Figure 11.7 shows that both environments independently calculate hashes for the data stored in both worlds because the data is sourced independently. Both systems identify business keys and their relationships for the purpose of hashing. In addition, whole documents in Hadoop can be linked to the data warehouse by a Data Vault 2.0 link that references the hash key of the Hadoop document. It is also possible to join across both systems when building the information marts.

In order to make this approach work, make sure that all systems use the same hash function and the same approach to apply it to the input data (refer to section 11.2.2 for details).

11.2.3.5 Performance

From a theoretical standpoint, it requires more CPU power to calculate one hash value than to generate one sequence value. The previous chapters have already discussed the reason why hash values are favored over sequence values (with regards to performance):

- **Easy to distribute hash key calculation:** the hash key calculation depends only on the availability of the hash function (which might be a tool problem) and the business key that needs to be hashed. For that reason, the hashing can be distributed very easily, for example to other environments or to multiple cluster nodes.
- **Hash key calculation is CPU not I/O intensive:** calculating the hash key is a CPU intensive operation (consider the calculation of a large number of hash keys) but doesn't require much I/O (the only I/O workload required is when storing the hash keys in the stage area for reusability). Because CPU workload can be easily and cheaply distributed over multiple CPUs, it is generally favored over I/O workload.
- **Loading of dependent tables doesn't require lookups:** the biggest performance gain comes from the fact that calculating the hash key requires only the business key, as described in the first bullet point. This advantage makes the lookup to retrieve the business key's sequence number from a hub obsolete. Because lookups cost a high amount of I/O, this is a popular advantage with high performance benefits.
- **Reduce the need for column comparing:** another intensive operation in data warehousing is to detect a change in descriptive data for versioning. In Data Vault, this happens when loading new rows into satellites because only deltas are stored in satellites. In order to detect if the new row should be stored in the satellite (which requires that at least one column be changed in the source system), every column between the source system (in the staging table) and the satellite has to be compared for a change. It also requires dealing with possible NULL values. Hash diff values can reduce the necessary comparisons to only one comparison: the hash diff value itself. Section 11.2.5 will show how to take advantage of the hash diff value.

In summary, hash keys provide a huge performance gain in most cases, especially when dealing with large amounts of data. For these reasons (and the additional ones regarding the integration with other environments), sequence numbers have been replaced by hash keys in Data Vault 2.0.

The previous sections have shown some advantages of hash functions in data warehousing, but they are no silver bullets [26]. Depending on how they are used, new problems might be introduced that need to be dealt with. However, the advantages of hash keys outweigh their drawbacks.

11.2.4 HASHING BUSINESS KEYS

The purpose of hash keys is to provide a surrogate key for business keys, composite business keys and business key combinations. Hash keys are defined in parent tables, which are hubs and links in Data

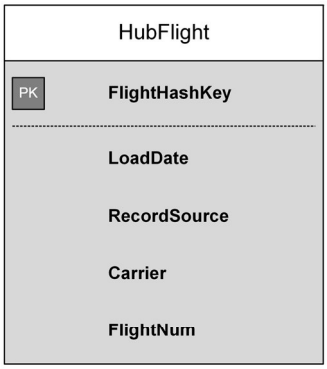

FIGURE 11.8

Hub with hash key (physical design).

Vault 2.0. They are used by all dependent Data Vault 2.0 structures, including links, satellites, bridges and PIT tables. Consider a hub example from Chapter 4 (Figure 11.8).

The hash key **FlightHashKey** is used for identification purposes of the business keys **Carrier** and **FlightNum** in the hub. All other attributes are metadata columns that are described later in this chapter. For link structures, the hash key that is used as an identifying element of the link is based on the business keys from the referenced hubs (Figure 11.9).

Note that there are a total of five hash keys in the **LinkConnection** link. Four of them are referencing other hubs. Only the **ConnectionHashKey** element is used as the primary key. This key is not calculated from the four hash values, but from the business keys they are actually based on. When loading the link structure, these business keys are usually available to the loading process.

Calculating a hash from other hash values is not recommended because it requires cascading hash operations. In order to calculate a hash based on other hashes, these hashes have to be calculated in the link loading process first. Because this operation might involve multiple layers of hashes, many unnecessary hash operations are introduced. Because they are CPU intensive operations, the primary goal of the loading patterns is to reduce them as much as possible.

In order to calculate hash key for hubs and links, it is required to concatenate all business key columns (think about composite business keys) and apply the guidelines from section 11.2.2 before the hash function is applied to the input. The following pseudo code shows the operation with n individual business keys in the composite key of the hub:

$$HashKey = Hash(UpperCase(Trim(BK_1) + d + Trim(BK_2) + \cdots + Trim(BK_n)))$$

BK_i represents the individual business key in the composite key and d the chosen delimiter. The order of the business keys should follow the definition of the physical table and should be part of the documentation. If the business key is not a composite business key, n equals to 1 and the above function is applied without delimiters. It assumes that all other standards, especially the endianness and character set, are taken care of. This also includes any control characters in the business keys and common conversion rules for various data types, for example floating values or dates and timestamps.

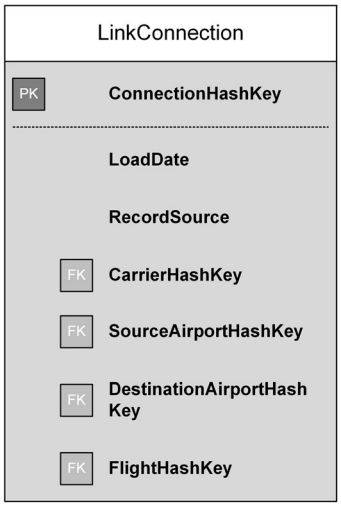

FIGURE 11.9

Link with hash keys (physical design).

The approach is the same for hubs and links: in the case of links, each BK_i represents a business key from the referenced hubs (see Figure 11.10). If hubs with composite business keys are used, all individual parts of the composite business key are included. For degenerated links, all degenerated fields have to be included as well. These weak hub references are part of the identifying hash key as any other hub reference.

While not intended on purpose, some of the individual business keys might be NULL. Because most database systems return NULL from string concatenations if one of the operators is a NULL value, NULL values have to be replaced by empty strings. Without the delimiter, the meaning of the business key combination would change in an erroneous way. Consider the example rows in Table 11.7.

After concatenating the four business keys without a delimiter, the string becomes in both cases "ABCDEFGHI". Because the input to the hash function is the same, the resulting hash value is the same as well. But it is obvious that both rows are different and should produce different hash values. This is achieved by using a semicolon (or any other character sequence) as a delimiter. The first row

Table 11.7 Importance of Delimiters

	BK 1	BK 2	BK 3	BK 4	Hash Result (without Delimiters)	Hash Result (with Delimiters)
Row 1	ABC	(null)	DEF	GHI	6FEB8AC01A4400A7 28B482D0506C4BEB	D9D33F7E6E80D174 7C45465025E9E6AF
Row 2	ABC	DEF	(null)	GHI	6FEB8AC01A4400A7 28B482D0506C4BEB	FD4D58C47B5C343A 4A9E23922ABE4C46

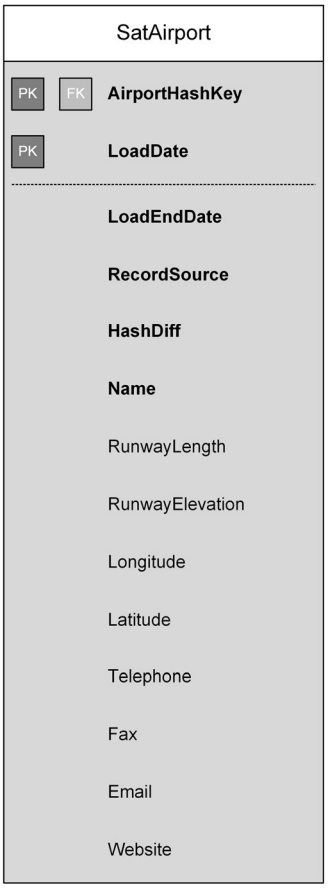

FIGURE 11.10

Satellite with multiple hub references (physical design).

is concatenated to "ABC;;DEF;GHI" and the second row to "ABC;DEF;;GHI". After sending the data through the hashing function, the desired output is achieved by retrieving different hash values.

In the previous chapter, a statement similar to the following T-SQL statement was used to perform the operation:

```
SELECT UPPER(CONVERT(char(32),HASHBYTES('MD5',
       UPPER(CONCAT(
              RTRIM(LTRIM(COALESCE(@event, ''))), ';',
              RTRIM(LTRIM(COALESCE(@computer, ''))), ';',
              RTRIM(LTRIM(COALESCE(@operator, ''))), ';',
              RTRIM(LTRIM(COALESCE(@sourceid, ''))), ';',
              RTRIM(LTRIM(COALESCE(@id, '')))
       ))
),2));
```

This statement implements the previous pseudocode using Microsoft's MD5 implementation in SQL Server 2014 and takes care of potential NULL values in addition. Note that each individual business key (the variables, such as **@event**, in the above statement) is checked for NULL values using the **COALESCE** function. The business keys are also removed from leading and trailing spaces on an individual basis. In this example, a semicolon is used to separate the business keys before they are being hashed using the **HASHBYTES** function. The result of this function is a varbinary value. This varbinary is converted to a char(32) value. The last **UPPER** function around the **CONVERT** function makes sure that the hexadecimal hash key is using only uppercase letters. If SHA-1 should be used instead, the result from the **HASHBYTES** function has to be converted to a char(40) instead.

Note that, in rare cases, business keys are case-sensitive. In this case, the upper case function has to be avoided in order to generate a valid hash key that is able to distinguish between the different cases. Keep in mind that the goal is to differentiate between the different semantics of each key, not to follow the rules at all cost.

It is also possible to implement this approach in ETL, for example by using a SSIS community component **SSIS Multiple Hash** [27] or using a **Script Component** (standard component in SSIS). The latter is demonstrated in section 11.6.3.

11.2.5 HASHING FOR CHANGE DETECTION

Hash functions can also be used to detect changes in descriptive attributes. As described in Chapter 4, descriptive attributes are loaded from the source systems to Data Vault 2.0 satellites. Satellites are delta-driven and store incoming records only when at least one of the source columns has changed. In this case, the whole (new) record is loaded to a new record in the satellite. The loading process of satellites (and other entities) is discussed in the next chapter.

In order to detect a change that requires an insert into the target satellite, the columns of the source system have to be compared with the current record in the target satellite. If any of the column values differ, a change is detected and the record is loaded into the target. To perform the change detection, every column in the source must be compared with its equivalent column in the target. Especially when loading large amounts of data, this process can become too slow in some cases. For such cases, there is an alternative that involves hash values on the column values to be compared. Instead of comparing each individual column values, only two hash values (the hash diffs) are compared, which can improve the performance of the change detection process and therefore the satellite loading process.

The basic idea of the hash diff is the same as the hash key in the previous section: it uses the fact that, given a specific input, the hash function will always return the same hash value. Instead of comparing individual columns, only the hash diff on these columns is used (Table 11.8).

Table 11.8 shows data in a stage table and the current record in the target satellite. Both records are hashed using a MD5 hash function, which results in a 32-character-long hash diff. Because the descriptive data is the same in the stage table as in the satellite table, both sides share the same hash diff value. After comparing the **hash diff** column (instead of the individual columns **title**, **first name** and **last name**) no change is detected and, as a result, no row is inserted into the satellite. If the data changes only a little, the hash diff value becomes different (Table 11.9).

Table 11.8 Example Data Compared by Hash Diff (without a Change)

Attribute Name	Stage Table Value	Satellite Table Value
Title	Mrs.	Mrs.
First Name	Amy	Amy
Last Name	Miller	Miller
Hash Diff	CADAB1708BF002A85C49FF78DCFD9A65	CADAB1708BF002A85C49FF78DCFD9A65

Table 11.9 Example Data Compared by Hash Diff (with Change)

Attribute Name	Stage Table Value	Satellite Table Value
Title	Mrs.	Mrs.
First Name	Amy	Amy
Last Name	Freeman	Miller
Hash Diff	66C17DF4D91EE9F0CF39490BFCC20B60	CADAB1708BF002A85C49FF78DCFD9A65

As a result of the changed data, the hash diff is completely different. Because of this different value, it is easy to detect changes in the stage table without comparing the values itself. While both hash diffs have to be calculated, the hash from the satellite can be reused whenever data is loaded, because it is stored with the descriptive data in the satellite, as Figure 11.10 shows.

The satellite **SatAirport** contains a number of descriptive attributes. The **HashDiff** attribute stores the hash diff over the descriptive attributes and can be reused whenever a record in the stage table is present that describes the parent airport (identified by **AirportHashKey**).

The hash diff is calculated in a similar manner to the hash keys in the previous section. All descriptive attributes are concatenated using a delimiter before the hash diff is calculated. Before concatenating the attributes, leading and trailing spaces are removed and the data is formatted in a standardized format. Especially the data type, date formats, and decimal separators are of importance, because all descriptive data, including dates, decimal values and all other data types have to be converted to strings before the data is hashed. The following pseudocode is used within the first, yet incomplete, approach:

$$HashDiff = Hash(Trim(Data_1) + d + Trim(Data_2) + \cdots + Trim(Data_n))$$

$Data_i$ indicates descriptive attributes, d a delimiter. The only difference between the pseudocode in this section and the previous section (to calculate hash keys on business keys) is that the upper case function has been removed: in many cases, change detection should trigger a new satellite entry if the case of a character is changing. However, this depends on the requirements given by the organization. In other cases, case-sensitive descriptive data is set on a per-satellite basis or even a per-attribute basis. How case-sensitivity is implemented depends on the definition of the source system and the data warehouse, as well.

Table 11.10 Different Parents with the Same Descriptive Data

Passenger HashKey	LoadDate	LoadEndDate	Record Source	Title	First Name	Last Name
9d4ef8a…	**2014-06-03**	9999-12-31	DomesticFlight	Mrs.	Amy	Miller
12af89e…	**2014-06-05**	9999-12-31	DomesticFlight	Mrs.	Amy	Miller

The above pseudocode is not complete yet. In order to increase the uniqueness among different parent values, the business keys of the parent are added to the hash diff function. Consider the example shown in Table 11.10.

In this case, there are two passengers Mrs. Amy Miller. They are distinguished by different hash keys, which result from different inputs, the business keys. If only the descriptive data is included in the hash diff calculation, both records would share the same hash diff value. This might not be very dangerous, but if the hash diffs were different, yet correct, we could use the hash diff to find a specific version of the descriptive data for one individual parent without using a combination of both the parent hash key and the hash diff. It is desired to have a hash diff that is unique over all satellite records, because then it would be sufficient to use only the hash diff to locate a particular version, which might improve the query performance for loading patterns in some circumstances.

We achieve this uniqueness of the hash key by adding the business keys of the parent to the hash diff calculation:

$$HashDiff = Hash(UpperCase(Trim(BK_1) + d + Trim(BK_3) + \cdots + Trim(BK_n)) + d + Trim(Data_1) + d + Trim(Data_2) + \cdots + Trim(Data_n))$$

The business keys of the parent are just added in front of the descriptive data, delimited by the same delimiter and following the same guidelines as outlined in the previous section. Both groups, the business keys and the descriptive data, must follow a documented order. A good practice is to use the column order of the table definition. In many cases, the business keys are uppercased while descriptive data remains case sensitive. Keep this in mind when standardizing and developing the hash diff function. Note that it's not the hash that is added to the front of the descriptive data but the business keys themselves. This follows the recommendation to avoid hashing hashes (hash-a-hash) in order to avoid cascaded calculations (refer to the previous section for more details).

The input to the hash function for the examples in Table 11.10 is presented in Table 11.11.

Because the input to the hash function is different (due to the included business key), the hash value for both inputs is different. When loading the satellites, it is ensured that this hash diff can be regained

Table 11.11 Example Input to Hash Diff Function

Input to Hash Function	Hash Diff
7878;Mrs.;Amy;Miller	`9DA0891434B92DF529B8CCCD86CC140B`
2323;Mrs.;Amy;Miller	`E890EE7980D5B13449704293A1BB4CCA`

at any time because both the descriptive data as well as the business keys are available. Therefore, this is the recommended practice.

11.2.5.1 Maintaining the Hash Diffs

Using the hash diff can improve the performance of the satellite loading processes, especially on tables with many columns. However, it incurs a maintenance cost or effort in addition to the calculation in the stage area. The reason for this additional maintenance is that satellite tables might change. Consider the following example, presented in Table 11.12 to Table 11.14, which adds a column **academic title** to the example provided previously:

The first table shows the old satellite structure, with three descriptive attributes, namely **title**, **first name** and **last name**. Because the source system has changed, a new descriptive attribute is added in Table 11.13, called **academic title**. Because the source systems never delivered any data for this new column in the past, it is set to NULL, or any other value representing this fact.

Once the new column has been added to the source table, it can capture incoming data. In this case, a new record is being added to the table that overrides the first record (Amy Miller marries Mr.

Table 11.12 Initial Satellite Structure

Passenger HashKey	LoadDate	LoadEndDate	Record Source	Title	First Name	Last Name
8473d2a…	**1991-06-26**	9999-12-31	DomesticFlight	Mrs.	Amy	Miller
9d8e72a…	**2001-06-03**	9999-12-31	DomesticFlight	Mr.	Peter	Heinz

Table 11.13 Satellite Structure After Adding a New Column

Passenger HashKey	LoadDate	LoadEndDate	Record Source	Title	Academic Title	First Name	Last Name
8473d2a…	**1991-06-26**	9999-12-31	DomesticFlight	Mrs.	(null)	Amy	Miller
9d8e72a…	**2001-06-03**	9999-12-31	DomesticFlight	Mr.	(null)	Peter	Heinz

Table 11.14 New Records are being Added to the New Satellite Structure

Passenger HashKey	LoadDate	LoadEndDate	Record Source	Title	Academic Title	First Name	Last Name
8473d2a…	**1991-06-26**	2003-03-03	DomesticFlight	Mrs.	(null)	Amy	Miller
9d8e72a…	**2001-06-03**	9999-12-31	DomesticFlight	Mr.	(null)	Peter	Heinz
8473d2a…	**2014-06-20**	9999-12-31	DomesticFlight	Mrs.	Dr.	Amy	Freeman

Freeman). In addition, the source system provides an academic title for Mrs. Freeman. It is unclear if she always held this title, but from a data warehousing perspective, it makes no difference. For audit reasons, the old records will not be updated because at the time when they have been loaded (indicated by the load date), the source system did not deliver an academic title for her (or any other record).

The issue arises when hash diffs are used in this satellite. Tables 11.15,11.16 and 11.17 show the hash diffs for the above examples (in the same order as previously).

The numbers in the curly brackets indicate the corresponding business keys. The first table shows the MD5 values for the descriptive data. Note that the hash diffs are different to the ones provided in Table 11 because the example was slightly different: other business keys were used in that example.

The semantic meaning of the data behind the hash diffs in Table 11.15 and Table 11.16 did not change (compare this to Table 11.12 and Table 11.13). However, the hash diffs changed, indicating to the change detection that the rows have changed. The only difference between these tables is the introduction of a new column, **academic title**. But the new column has changed everything because it influenced the input to the hash diff function (Table 11.18).

Table 11.15 Hash Diffs for Initial Satellite Structure

Passenger HashKey	LoadDate	LoadEndDate	Hash Diff
8473d2a… {4455}	1991-06-26	9999-12-31	`BDD9DD9208611F2A8CF3670053634FF0`
9d8e72a… {6677}	2001-06-03	9999-12-31	`B3B1724EF449DA9D9521FA95A88A82A3`

Table 11.16 Hash Diffs for Satellite Structure After Adding a New Column

Passenger HashKey	LoadDate	LoadEndDate	Hash Diff
8473d2a… {4455}	**1991-06-26**	9999-12-31	`4F860465EB585FABF3CBD28E7A29AEC0`
9d8e72a… {6677}	**2001-06-03**	9999-12-31	`FFCF6191C583F5B77273D4CABF3EB98F`

Table 11.17 Hash Diffs After New Records Have Been Added to the New Satellite Structure

Passenger HashKey	LoadDate	LoadEndDate	Hash Diff
8473d2a… {4455}	**1991-06-26**	2003-03-03	`4F860465EB585FABF3CBD28E7A29AEC0`
9d8e72a… {6677}	**2001-06-03**	9999-12-31	`FFCF6191C583F5B77273D4CABF3EB98F`
8473d2a… {4455}	**2014-06-20**	9999-12-31	`1C1E8C1799E39E567F746F720800B0E5`

The first row presents the input for Amy Miller before the structural change to the satellite, and the second row the input after the column has been added. Notice the added semicolon in the input on the left side, between the title and the first name. This semicolon is due to the fact that a NULL column was introduced and changed to an empty string. The advantage of the semicolon that it allows for NULL values now becomes a disadvantage.

However, we actually need the new hash diff value because, otherwise, we're unable to detect any changes in the source system. If we leave the old hash diffs in the satellite table, all current source records (modified or unmodified) are interpreted as modified, due to the different hash diff value. Therefore, we need to update all hash diffs in the satellite table to prevent unnecessary (and unwanted) inserts into the satellite table.

It is possible to avoid this maintenance overhead of the hash diff by following a simple strategy. The first idea is to update only the current records in the satellite. They are indicated by having no load end date (or in the examples of this book, an end-date of 9999-12-31), because only those records are required for any regular comparison during satellite loading. All end-dated satellite entries have been replaced by newer records already and are not required to compare with. This approach reduces the number of records to be updated after structural changes but still incurs many updates, especially when there are many different parent entries.

But it is actually possible to further reduce the maintenance overhead to zero: by making sure that the hash diffs remain valid, even after structural changes. However, there are some conditions to make this happen:

1. Columns are only added, not deleted.
2. All new columns are added at the end of the table.
3. Columns that are NULL and at the end of the table are not added to the input of the hash function.

The first condition is required because of auditability reasons in any case. The goal of the data warehouse is to provide all data that was sourced at a given time. If columns are deleted, the auditability is not achieved anymore.

Meeting the second and the third condition is best explained by an example (Table 11.19, modified from Table 11.18):

Table 11.18 Modified Example Input to Hash Diff Function

Input to Hash Function	Hash Diff
4455;Mrs.;Amy;Miller	BDD9DD9208611F2A8CF3670053634FF0
4455;Mrs.;;Amy;Miller	4F860465EB585FABF3CBD28E7A29AEC0

Table 11.19 Improving the Hash Diff Calculation

Input to Hash Function	Hash Diff
4455;Mrs.;Amy;Miller	BDD9DD9208611F2A8CF3670053634FF0
4455;Mrs.;Amy;Miller;	D64074FE047165874481A7B299C4A766
4455;Mrs.;Amy;Miller	BDD9DD9208611F2A8CF3670053634FF0

The first line shows the input to the hash function before the structural change. The second line shows the input after the structural change, meeting the first two conditions, but not the third. Instead of adding the new column in the middle between the title and the first name, the academic title is added at the end of the table. Without meeting the last condition, a semicolon is added and the NULL value is added after the semicolon as an empty string. Because the input has changed by the addition of the semicolon character, the hash diff has changed, indicating a change.

If the third condition is met in addition, all empty columns at the end of the input are removed before the hash function is called. It means that all trailing semicolons are being removed. That way, the input becomes the same again and the hash diffs indicate no change because they have to be the same. This approach works with multiple NULL columns at the end.

How does it improve the satellite loading process? First, the old hash diffs are still valid. There is no semantic difference between the rows in Table 11.19. The only difference is a structural change that should not have an impact on the satellite loading process by triggering an insert operation into the satellite. This is achieved by this improved hash diff calculation.

However, if a new column is added to the satellite and the source system provides a value for the new column, a new hash diff is being calculated (Table 11.20).

Because the input to the hash function has changed between the two loads, the hash function will return two different hash diffs. This is in line with the requirements of the satellite loading process because the satellite should capture the change in Table 11.20.

The biggest advantage of the presented approach is that the hash diff values are always valid if the conditions listed earlier are met. On the other hand, the disadvantage is that the satellite columns might be in a different order than the source table, for example when new columns are added to the source table after it was initially sourced into the data warehouse. However, because this approach doesn't affect the auditability at all, we recommend this approach in most cases.

11.3 PURPOSE OF THE LOAD DATE

The **load date** identifies "*the date and time at which the data is physically inserted directly into the database*" [6]. In the case of the data warehouse, this might be the staging area, the Raw Data Vault or any other area where the data arrives. Once the data has arrived in the data warehouse, the load date is set and is part of the auditable metadata. Except for satellites, the load date is primarily a metadata attribute that helps when debugging the data warehouse. However, because it is part of the satellite's primary key, it is in fact an essential part of the model that needs to be taken special care of.

If the database system that is used for the data warehouse supports it, the **load date** should represent the instant that the data is inserted to the database. The time of arrival should be accurately

Table 11.20 Changing the Semantic Meaning

Input to Hash Function	Hash Diff
4455;Mrs.;Amy;Miller	`BDD9DD9208611F2A8CF3670053634FF0`
4455;Mrs.;Amy;Miller;Dr.	`0044E798C5586E931A604D1E0CD09FA1`

identified down to a millisecond level in order to support potential real-time ingestion of data. If the database system doesn't support timestamps but only dates, the alternative is to add a sub-sequence number to the load date that represents pseudo millisecond counters. This is required for satellites in order to make the primary key (which consists of the parent hash key and the load date) unique. This uniqueness requires milliseconds because data might be delivered multiple times a day (in batches or in real time).

The data warehouse has to be in control of the load date. Again, this is especially important for the Data Vault satellites because the load date is included in the satellite's primary key. If the load date breaks for any reason, the model will break and stop the loading process of the data warehouse. A load date typically breaks if two different batches are identified by the same load date. In this case, the satellite loading process would try to insert records (due to a detected change) with a duplicate key. For example, if a create date or export date from the source system that is not controlled by the data warehouse is used as a load date, the following issues are implicated:

- **Mixed arrival latency:** not all data arrives at the same batch window timeframe. There is often some form of ongoing mixed arrival schedule for source data. When the load date is generated during the insert into the data warehouse, this mixed arrival date time can be assigned gracefully. The inserts to the Raw Data Vault (in this case) occur at the time of arrival. In one case, the business needs to be updated every 5 seconds with live feed data; in another case, the EDW needs to show a time-lineage of arrival to an auditor for traceability reasons.
- **Mixed time zones for source data:** not all data is being created in the same time zone. Some source systems are located in the USA, some in India, and so on. If a create date were to be used as a load date, these time zones have to be aligned on the way into the data warehouse. Any conversion, no matter how simple, slows down the loading process of the data warehouse and increases the complexity. Increased complexity will not only increase the maintenance effort but also the chance that errors happen.
- **Missing dates on sources:** not all source systems provide a create date or another candidate to be used as a load date. Dealing with these exceptions requires conditions in the loading procedures, which should be avoided because it adds complexity again. Instead, apply the same logic to all sources to keep the loading procedures as simple as possible.
- **Trustworthiness of dates on sources:** not all source systems run correctly. In some cases, the time of the source system has an offset due to a wrong configuration or hardware failures (consider a defunct motherboard battery). In other cases, the date and time configuration of the source system is changed between loads (for example to fix an erroneous configuration or after replacing the motherboard battery). This actually makes things worse as it complicates the matters when times now overlap.

If the data warehouse team decides to use an external timestamp as the load date, the data warehouse loading process will fail in the following scenarios:

- **Create date is modified by source system:** the data warehouse team has no control over the create date from the external system. If the source system owner or the source system application decides to change the create date for any reason, it has to be handled by the data warehouse. However, how should this change be handled if the load date is part of the immutable primary key within satellites? Changing it is not possible for auditing reasons.

- **Loading history with create date as load date:** in other cases, the data warehouse team has to source historical data that might or might not have create dates. For example, if the source system has introduced the create date in a later version, historical data from earlier versions don't provide the create date.

For all these reasons, the load date has to be created by the data warehouse and not sourced from an external system. This also ensures the auditability of the data warehouse system because it allows auditors to reconcile the raw data warehouse data to the source systems. The load date further helps to analyze errors by loading cycles or inspecting the arrival of data in the data warehouse. It is also possible to analyze the actual latency between time of creation and time of arrival in the data warehouse.

System-driven load dates (system-driven by the data warehouse) can be used to determine the most recent / most current data set available for release downstream to the business users. While it doesn't provide complete temporal information, it provides the technical view on the most recent raw data or at any given point in time.

But what should be done with the dates and timestamps from the source system? Typically, the following timelines exist in source systems:

- Data creation dates
- Data modified dates
- Data extract dates
- Data applied dates such as
  - Effective start and end dates
  - Built dates
  - Scheduled dates
  - Executed dates
  - Deleted dates
  - Planned dates
  - Financed dates
  - Cancelled dates

None of these dates or any other date from the source system qualify as load dates. However, they provide value to the business. As such, they should be included in the data warehouse as descriptive data and loaded to Data Vault satellites.

11.4 PURPOSE OF THE RECORD SOURCE

The record source has been added for debugging purposes only. It can and should be used by the data warehousing team to trace where the row data came from. In order to achieve this, the record source is a string attribute that provides a technical name of the source system, as detailed as it is required. In most cases, data warehouse teams decide to use some hierarchical structure to indicate not only the source system, but also the module or table name. If accessing a relational Microsoft SQL Server source, the record source should use the following format:

```
[Source Application Name].[Schema Name].[Table Name]
```

For example, the following record source would indicate a table from the **CRM** application, in the **Cust** schema:

```
CRM.Cust.Customers
```

Avoid using only the name of the source system because following this approach can be very helpful when tracing down errors. When analyzing the data warehouse due to a run-time error, either in development or production, it is helpful to have detailed information available.

The record source is added to all tables in the staging area, the Raw Data Vault, the Business Vault and probably the information marts. But what if data from multiple sources is combined or aggregated? In this case, the record source is not clear anymore. The recommended practice is to set the record source to the technical name of the business rule that generated the record. The technical name can be found in the metadata of the data warehouse (refer to Chapter 10, Metadata Management). If there is no business rule, the record source should be set to SYSTEM. Examples include the ghost record in satellites (see Chapter 6, Advanced Data Vault Modeling) or any other system-driven records.

Use of record sources that are dependent on a specific load should be avoided. For example, the file name is often not a good candidate for a record source, especially if it contains a date. The record source should group all data together that comes from the same origin of data. Having a date in the record source prevents this. The same applies for physical database or server names: what if the location of the data changes? The file name, the database name or the server name might be changed in the future, even if the source where the records came from remains the same.

11.5 TYPES OF DATA SOURCES

The data warehouse can source the raw data from a variety of data sources, including structured, semi-structured and unstructured sources. Data can also come from operational systems in batch loads or in real time, through the Enterprise Service Bus (ESB). Typical examples for source systems in data warehousing include [28]:

- **Relational tables:** operational systems often work with a relational database backend, such as Microsoft SQL Server, Oracle, MySQL, PostgreSQL, etc. This data can be directly accessed using OLE DB or ADO.NET sources in SSIS [29].
- **Text files:** if direct access of the relational database is not permitted, operational systems often export data into text files, such as comma-separated files (CSV) or files with fixed-length fields [29].
- **XML documents:** in other cases, operational systems also provide XML files that can be processed in SSIS [29].
- **JSON documents:** similar to XML documents, JSON documents provide semi-structured files that can be processed by SSIS using additional third-party components.
- **Spreadsheets:** a lot of business data is stored in spreadsheets, such as Microsoft Excel files, which can be directly sourced from SSIS [29] and Google Sheets, which can be sourced with third-party components.
- **CRM systems:** customer data is often stored in customer relationship management (CRM) systems, such as Microsoft Dynamics CRM, Salesforce CRM or SAP. Third-party data sources allow sourcing this data.

- **ERP systems:** organizations use ERP systems to store data from business activities, such as product planning, manufacturing, marketing and sales, inventory management or shipping and payment. Examples for ERP systems include Microsoft Dynamics ERP, Microsoft Dynamics GP, SAP ERP, and NetSuite. There are third-party data sources available for sourcing data from ERP systems.
- **CMS systems:** content management systems (CMS) provide the ability to create intranet applications for unstructured or semi-structured content. Third-party connectors allow connecting to CMS systems such as Microsoft SharePoint.
- **Accounting software:** this type of software is used to manage general ledger and perform other financial activities, such as payroll management. Examples include QuickBooks and Xero. Third-party components allow sourcing of data from these systems.
- **Unstructured documents:** include Word documents, PowerPoint presentations and other unstructured documents.
- **Semi-structured documents:** include emails, EDI messages, and OFX financial data formats which can be sourced with third-party components.
- **Cloud databases:** more and more data is stored in the cloud, for example in Microsoft Azure SQL Database, Amazon SimpleDB, or Amazon DynamoDB. Microsoft Azure SQL Database is supported by the OLE DB data source; other cloud databases are supported by third-party vendors.
- **Web sources:** the Internet provides a rich set of third-party data that can be added to the data warehouse in order to enrich the data from operational systems. The same applies to data from Intranet locations. Examples for file formats used typically in such settings include RSS feeds (for news syndication), OData and JSON (for standardized access to data sets) in addition to general XML documents. Some of these formats can be sourced by SSIS with built-in capabilities. Others require a third-party data driver.
- **Social networks:** social networks present another, yet more advanced, Web source. There are SSIS data sources to process data from social networks such as Twitter and Facebook.
- **Mainframe copybooks:** a lot of operational data is managed by mainframe systems and will be in the future. EBCDIC files are one of the standards and can be handled by SSIS with additional components.
- **File systems:** not all documents are stored in local file systems. Instead, some data resides on FTP servers, can be copied using secure copy (SCP) commands or is in the cloud, for example Amazon S3. Third-party components can assist in retrieving data from such remote stores.
- **Remote terminals:** In other cases, the data is on remote servers, but only accessible via remote terminals, such as telnet or secure shell (SSH). Again, third-party vendors extend SSIS if that is required.
- **Network access:** even TCP ports on the network might provide data that is sourced into the data warehouse.

In order to load the data using ETL tools, the access methods have to be supported by the tool itself. Not all tools support all data sources, but it is possible to extend them by custom components. Microsoft SSIS is such an example where the base functionality can be extended by custom components from third-party vendors. It is also possible to add generic data sources from the Web, especially by accessing REST or XML Web Services. Additional SSIS components can be found on CodePlex [30].

The remaining sections discuss how to source typical data sources, including the sample airline data that is used throughout the book.

11.6 SOURCING FLAT FILES

The first example loads flat files into the data warehouse. These flat files are typically exported from operational systems, which often use a relational database. Flat files include several of the previously listed formats, including comma-separated files (CSV), fixed-length files and Microsoft Excel Sheets.

When sourcing text-based files, such as CSV or fixed-length files, all data is formatted as strings. In order to convert the raw data during the export in the source system, which also uses other data types, such as integers, or decimal values, a format is applied to build the string text that is exported into the text file. By doing so, the source application defines, for example for decimal values, how many positions after the decimal point should be included and how many should be rounded off.

11.6.1 CONTROL FLOW

The companion Web site provides three CSV files from the BTS database, introduced in the opening of this chapter. In order to run the following example in SSIS, you have to extract the Zip file and place the CSV files into one folder. The final SSIS control flow will traverse through the folder and load all data from all CSV files in the folder into the staging area.

In order to traverse through the folder, drag a **Foreach Loop Container** on the canvas of the control flow. Double-click to edit it and set the following parameters as shown in Figure 11.11.

Change the **Enumerator** type to **Foreach File Enumerator**. Configure the file enumerator by setting the **Folder** to the location where the CSV files from the companion Web site are located. Change the **Files** wildcard to ***.csv** and **Retrieve file name** to **Name and extension**.

The last option defines how the file name of the traversed file should be stored in a variable. It is possible to store:

- the **name and extension:** only the file name and the file extension are stored in the variable.
- the **fully qualified name:** this includes the file location and the file name including the extension of the file.
- the **name only:** only the file name, without the file extension, is stored.

Select the option to store the **fully qualified name** because the full name is required by the data flow that will be created in the next step.

The variable that stores the file name of the currently traversed file is set on the next tab **Variable Mappings** (Figure 11.12).

To store the file name in a variable, either choose a pre-existing variable or create a new variable by focusing the cell in the **Variable** column of the grid and selecting <**New Variable…**>. The dialog as shown in Figure 11.13 will appear.

Set the **container** of the variable to **package**. For the purpose of this example, it would also be sufficient to set it to the foreach loop container itself to reduce the scope of the variable. Set the **name** of the variable to **sFileName**. We will later reference the variable using this variable name. The **namespace** should be set to **User** or a defined namespace of your choice that is used to store such variables. Set the **value type** to **String**, as it will store the file name and extension of the traversed file. The default **value** should be set to **default** or something similar. Make sure **read only** is not selected.

After selecting the **OK** button, the variable is added to the previous variable mapping dialog and shown in the table with index 0 (Figure 11.14).

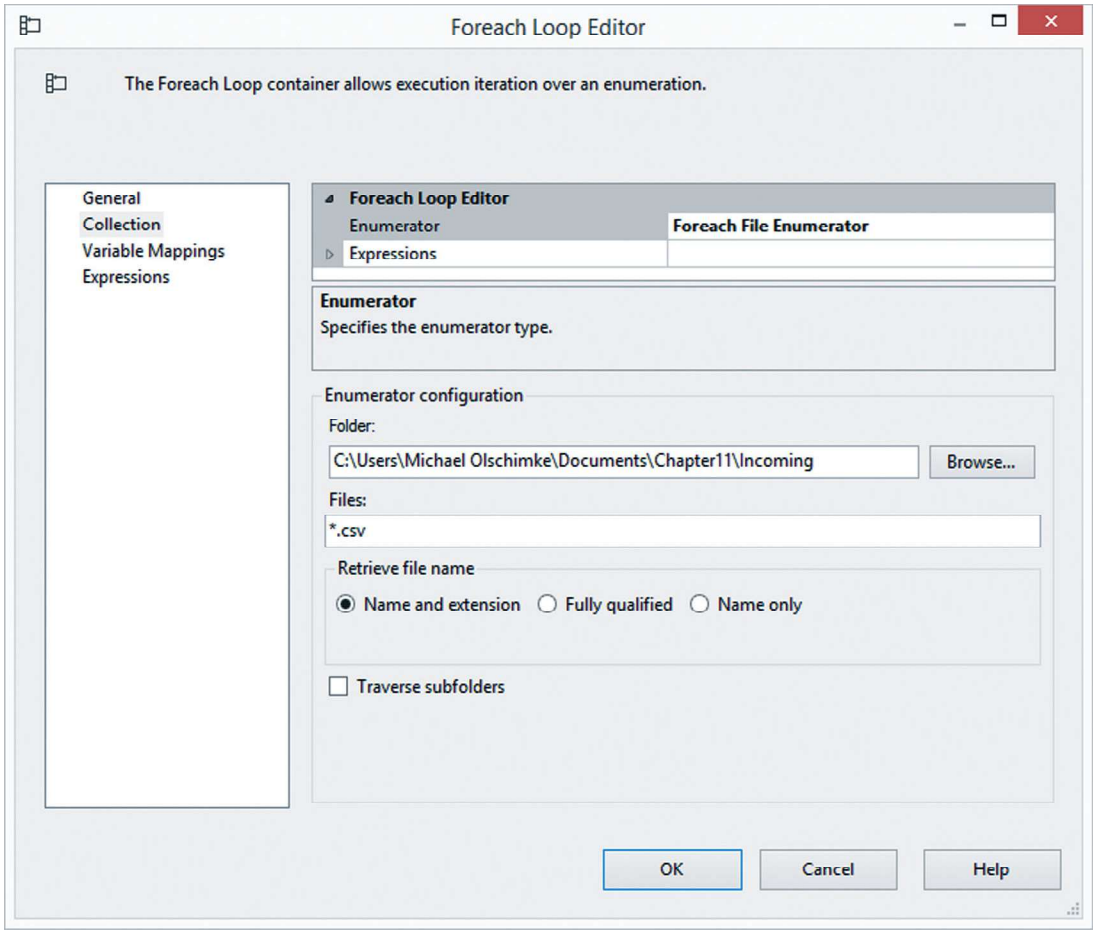

FIGURE 11.11

Configuring the collection of the foreach loop container.

Confirm the foreach loop editor by selecting **OK**. The next task ensures a consistent load date over all records in each file. For this purpose, the current timestamp is set to a SSIS variable using a Microsoft Visual C# script: whenever the container loops over a file, it first sets the current timestamp into the variable **User::dLoadDate** and then starts the data flow, which is discussed next. The data flow will use the timestamp in the variable to set the **load date** in the staging area. Follow the earlier approach and create a new SSIS variable by selecting the **Variables** menu entry in the context menu of the control flow. Create a new variable with name **dLoadDate** and the settings as shown in Table 11.21.

Close the dialog and drag a **Script Task** into the container. Open the editor as shown in Figure 11.15.

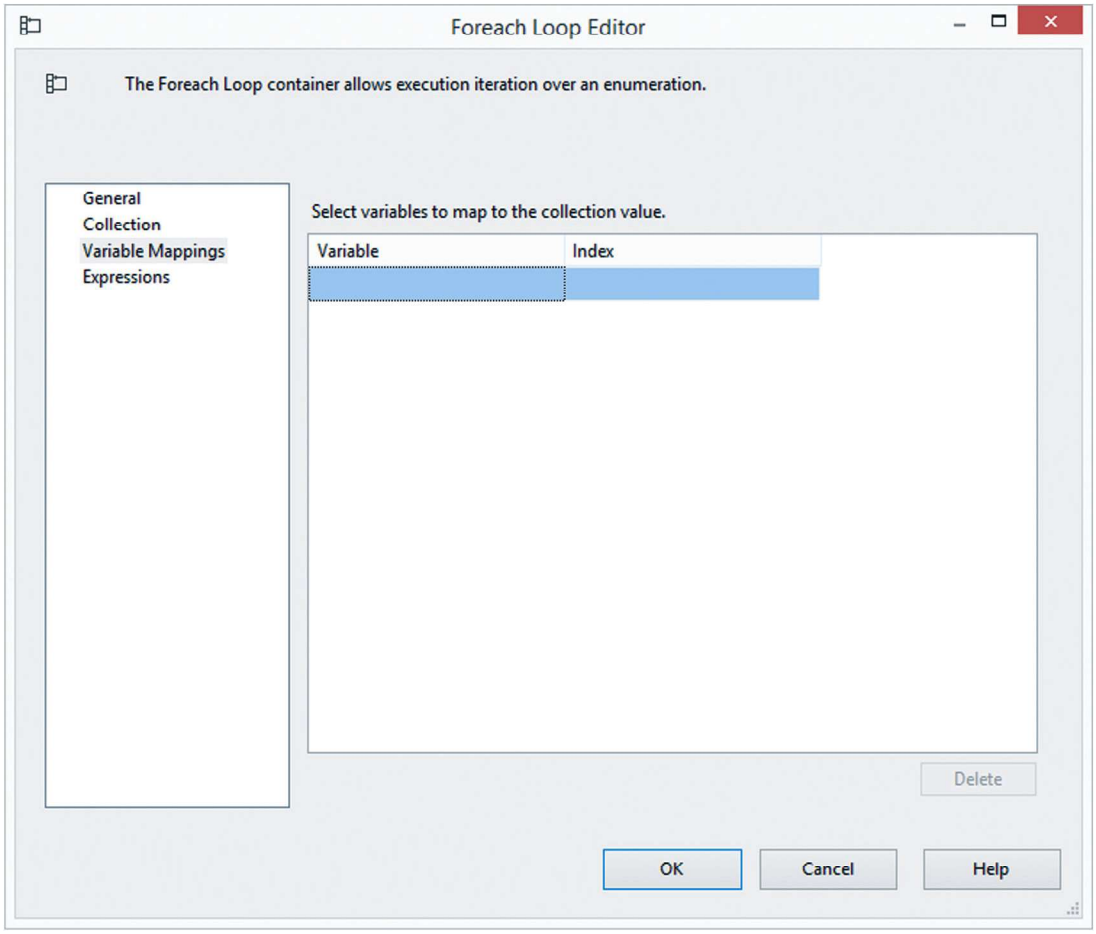

FIGURE 11.12

Map variable of foreach loop container.

Table 11.21 Variable dLoadDate Settings

Parameter	Value
Name	dLoadDate
Scope	Package
Data type	DateTime
Value	12/30/1899

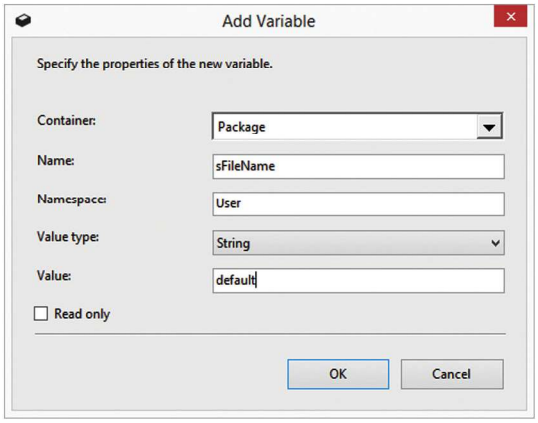

FIGURE 11.13

Add variable to store file name from foreach loop container.

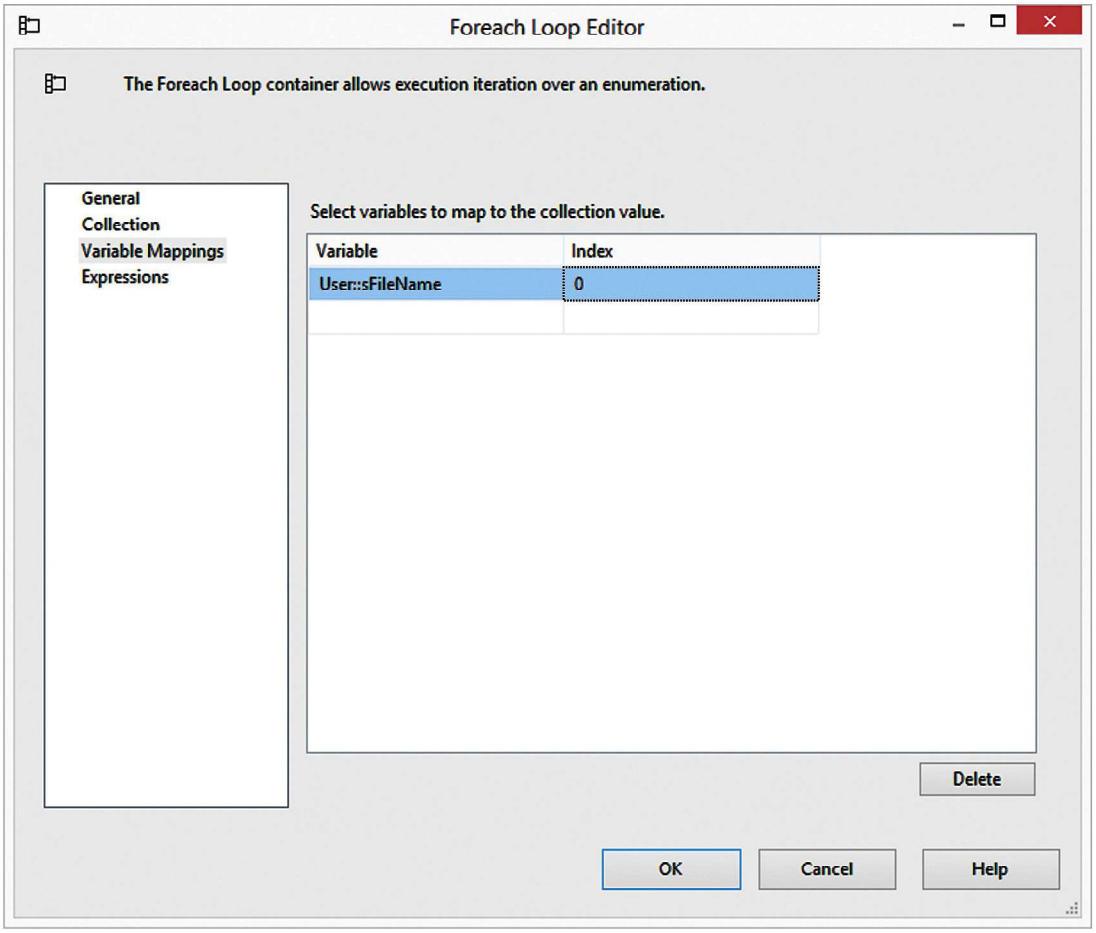

FIGURE 11.14

Added variable to the foreach loop container.

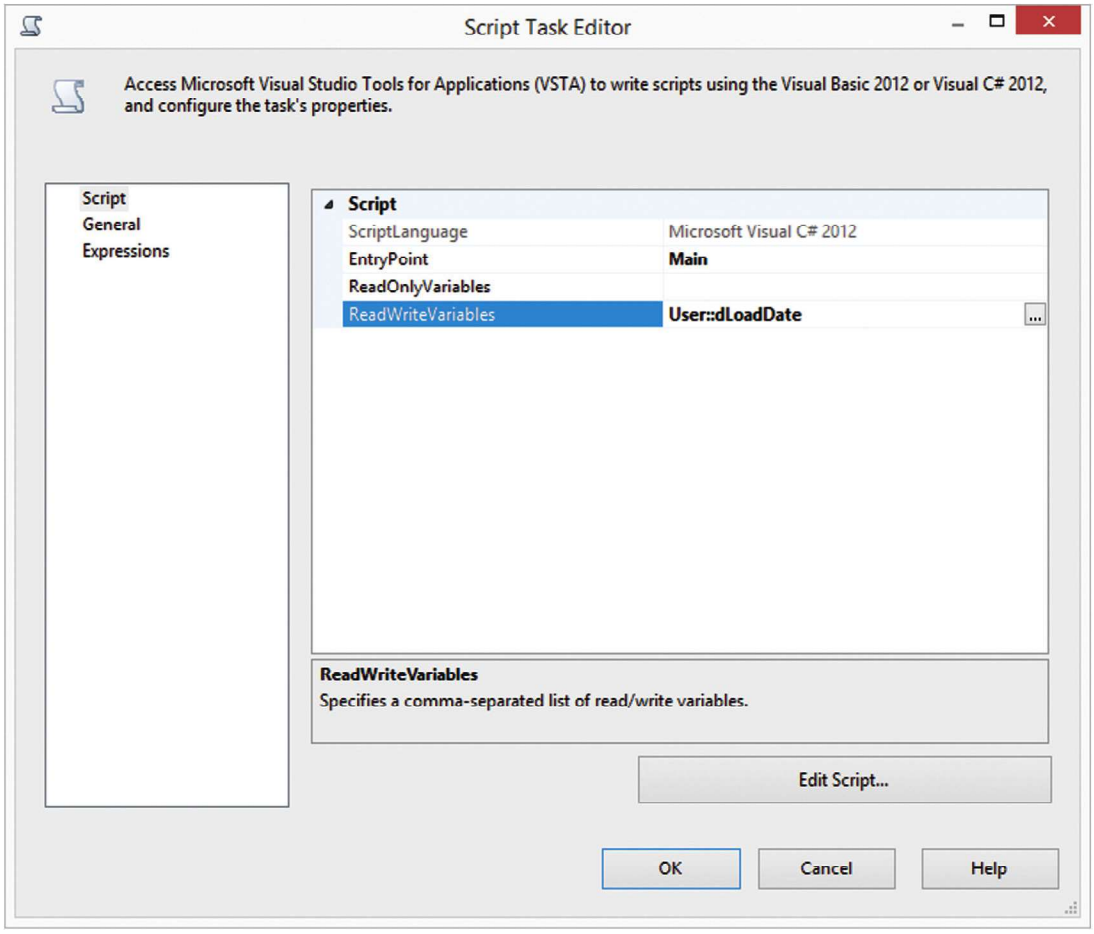

FIGURE 11.15

Script task editor to set the load date.

Add the variable **User::dLoadDate** to the **ReadWriteVariables** by using the dialog available after pressing the button with the … caption. Check the variable in the dialog and close it. Edit the script and enter the following code in the **Main** function:

```
Dts.Variables["User::dLoadDate"].Value = DateTime.Now;
Dts.TaskResult = (int)ScriptResults.Success;
```

This code writes the current timestamp of the system into the variable and reports a successful completion of the function to SSIS.

Add a **Data Flow Task** to the container and rename it to **Stage BTS On Time On Time Performance**. The final control flow is presented in Figure 11.16.

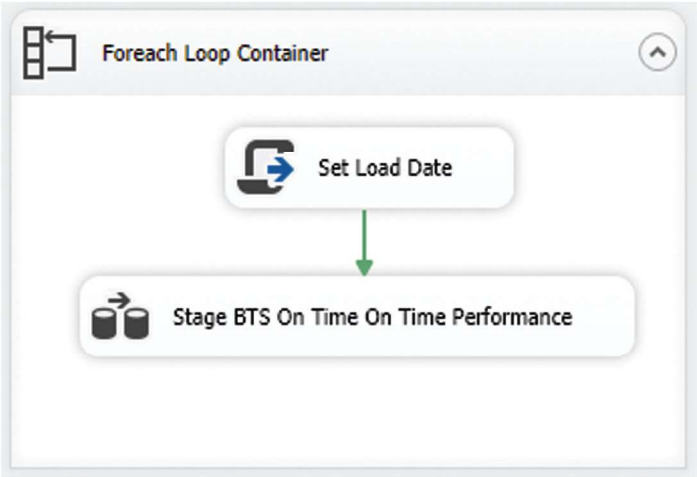

FIGURE 11.16

Control flow to source flat files.

The foreach loop container will enumerate over all files in the configured folder. Sort order of the file names appears to be in alphabetic order, but this is undocumented by Microsoft. There are custom components that allow sorting by file name and date among other options [31].

The next step is to configure the data flow task in order to stage the data in the flat files into the target tables.

11.6.2 FLAT FILE CONNECTION MANAGER

In order to set up the data flow for staging CSV flat files, the first step is to set up the **flat file connection manager**. Open the **Stage BTS On Time On Time Performance** data flow in the previous SSIS control flow. Drag a **Flat File Source** to the data flow (Figure 11.17).

Create a new flat file connection by selecting the **New…** button. In the following **Flat File Connection Manager Editor**, select one of the flat files in the source folder (Figure 11.18).

Make sure the **locale** is set to **English (United States)** and **Unicode** is not selected. The **format** should be set to **Delimited**, double quotes (“) should be used as **text qualifier** and **header row delimiter** is set to **{CR}{LF}** as usual under Microsoft Windows. The column names are provided in the first data row of the source file; therefore activate the corresponding check box.

The data from the source file can be previewed on the **columns** tab. Switch to the **advanced** tab to configure the data types and other characteristics of the columns (Figure 11.19).

This configuration is required because the CSV file doesn’t provide any metadata or description of the data, except the column headers. All data is formatted as strings only. While it is possible to manually configure the columns, it is also possible to use a wizard that analyzes the source data. Select the Suggest Types… button to start the wizard in Figure 11.20 (Figure 11.20).

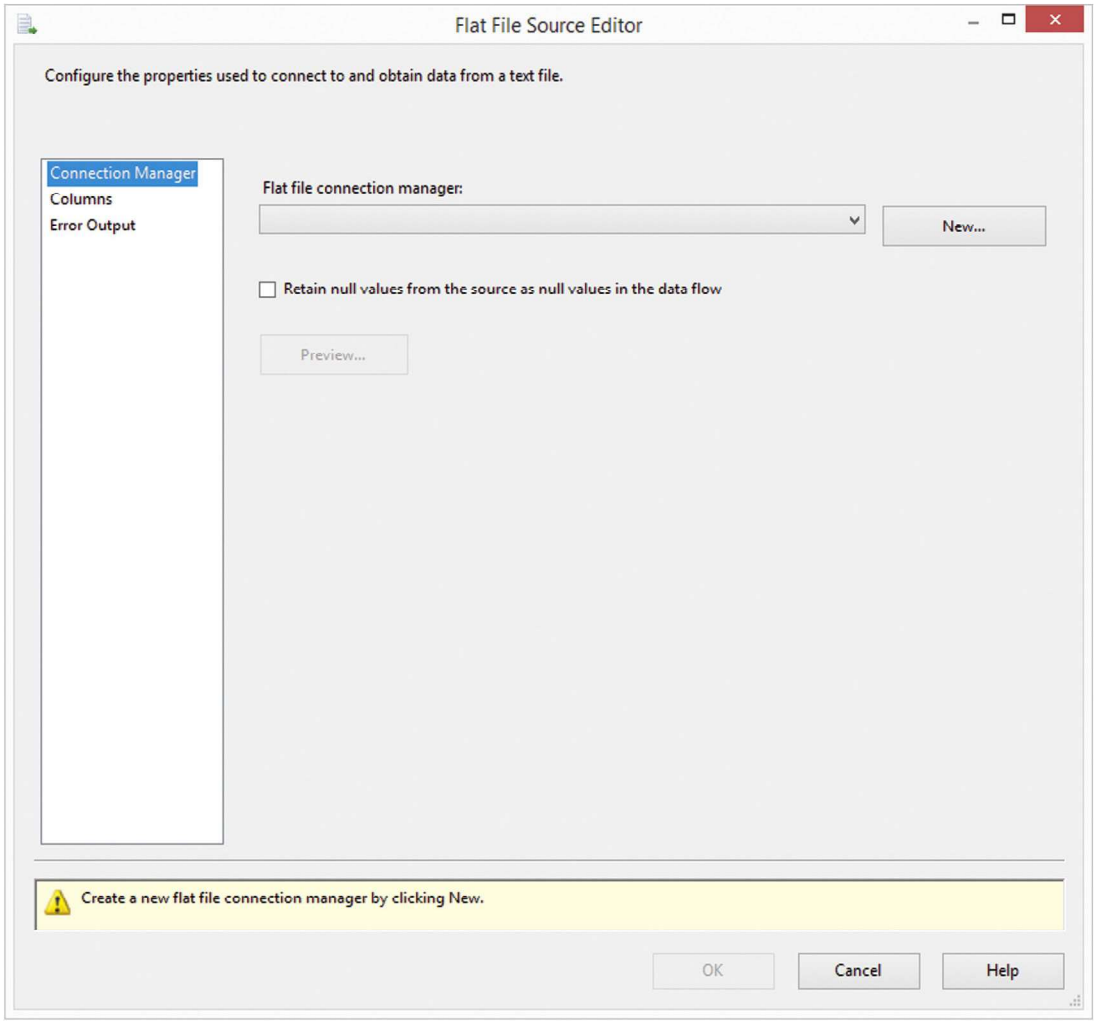

FIGURE 11.17

Flat file source editor.

The wizard analyzevs the source data by looking at a sample size only. This sample size should be large enough to capture all types of different characteristics of the data. Turn on the first two options and identify Boolean columns by 1 and 0. This follows the recommendation to use the actual data types of the raw data. Columns such as **delayed** are Boolean fields, which are formatted as 0 and 1. Make sure that **pad string columns** is turned off. After selecting the **OK** button, review the definitions of the source columns in the previous dialog. Note that some of the columns might have been incorrectly

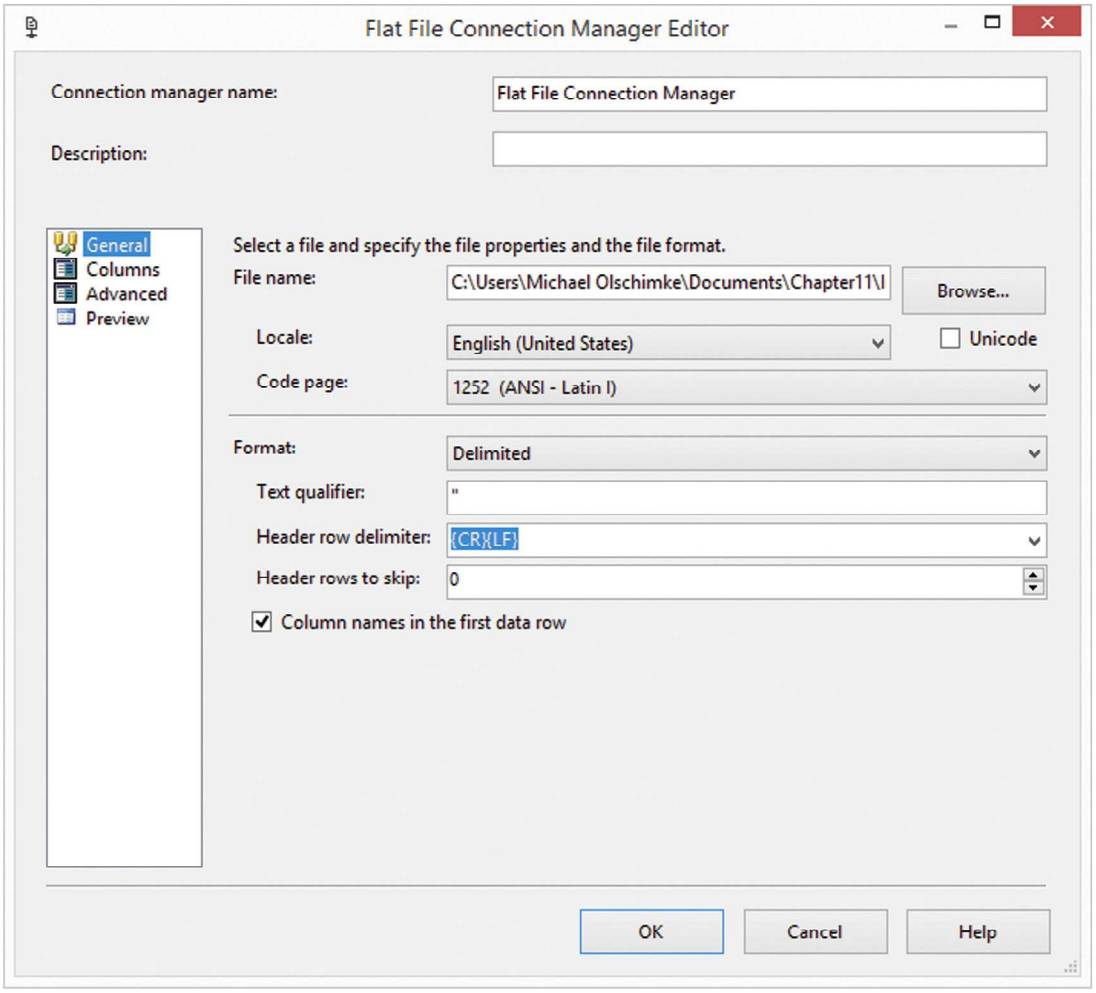

FIGURE 11.18

Setting up the flat file connection.

classified as Boolean because the data contains only records from January, which is in quarter 1 and month 1. Change them to better data types, such as two-byte signed integer. Also, you should increase some of the output column width of the string columns to allow longer city names, for example. Change all string columns from strings to Unicode strings. More information about the source columns can be found in the readme.html file that accompanies the source files.

After completion of the **flat file connection manager editor**, select **retain null values from the source as null values in the data flow** in the **flat file source editor**. Close the editor by selecting the **OK** button.

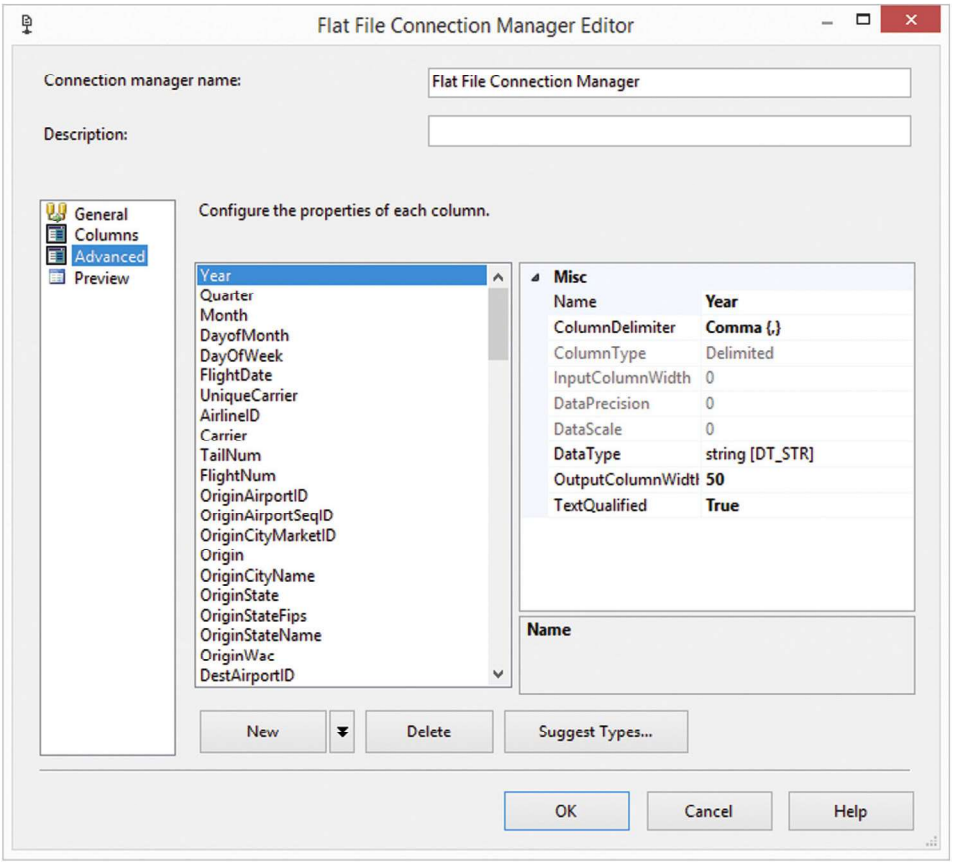

FIGURE 11.19

Configure columns of the flat file connection.

So far, the filename in the **Flat File Connection Manager** was configured using a static file name. This allows easy configuration of the columns and previewing data. In order to successfully traverse over all files in the source folder, we need to change the file name programmatically. Select the **Flat File Connection Manager** in the Connection Managers pane at the bottom of **Microsoft Visual Studio** and set the **DelayValidation** property to **True**. Open the expressions editor by clicking the ellipse button. The dialog in Figure 11.21 is shown.

Set the **ConnectionString** property by the following expression: **@[User::sFileName]**. This will set the file name to the file name stored in the SSIS variable obtained in the control flow.

11.6.3 DATA FLOW

After having configured the flat file connection, the next step is to add the system-generated attributes in the control flow and load the data into the destination.

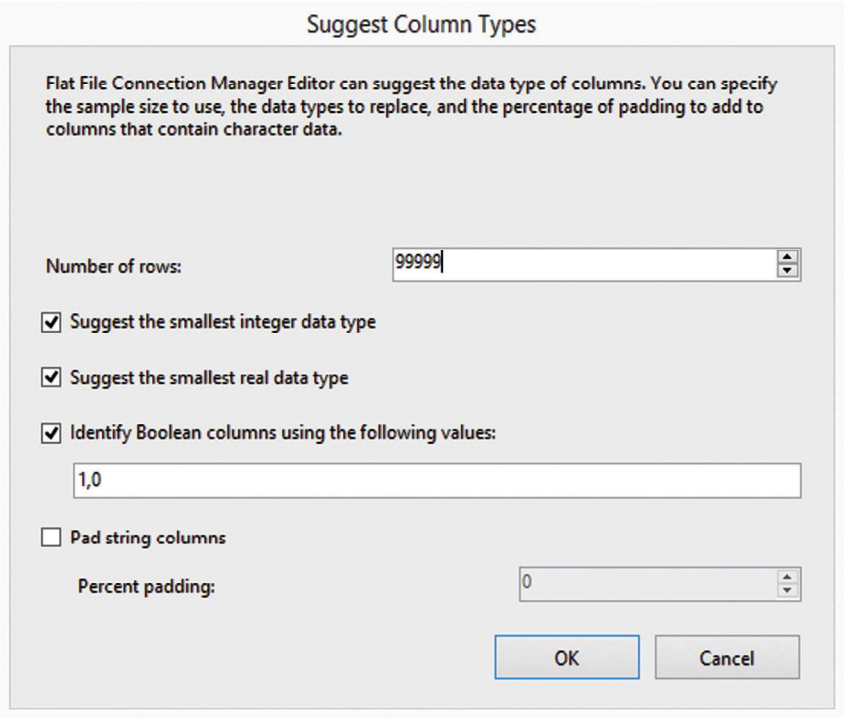

FIGURE 11.20

Suggest column types configuration.

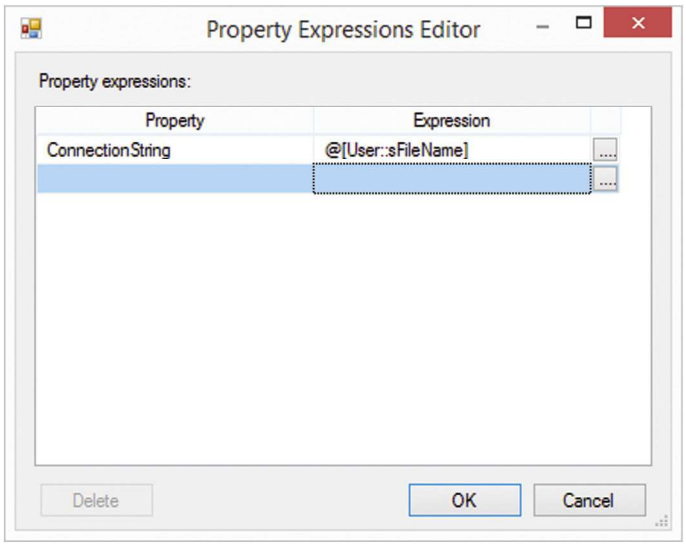

FIGURE 11.21

Property expression editor for flat file connection manager.

The following system-generated attributes should be added to the stage table:

- **Sequence number:** a number that is used to maintain the order of the source file in the relational database table.
- **Record source:** the source of origin of the data.
- **Load date:** the date and time when the record was loaded into the data warehouse. This should be consistent for all records in one file but different to other files.
- **Hash keys:** surrogate keys for each business key and their relations, based on a hash function.
- **Hash diffs:** hash values used to speed up column comparisons when loading descriptive data.

The primary key of the staging table consists of the sequence number and the load date. The combination of both fields must be unique for this reason.

The first step is to add the sequence number to the data flow. Microsoft SSIS doesn't provide a sequence generator. If the sequence should be created within SSIS, a **Script Component** is used. An alternative is to create an IDENTITY column in the Microsoft SQL Server table. The script component allows fully customized transformations, written in C# or Visual Basic.NET, to extend SSIS. Drag the component to the data flow canvas. The dialog in Figure 11.22 will be shown.

It is possible to use the script component in three different ways:

- **Source:** the script component acts as a source and sends records into the data flow. This is useful when loading data from data sources not directly supported by SSIS.
- **Destination:** the script component acts as a destination and supports writing records from the data flow into third-party locations such as custom APIs.
- **Transformation:** the script component transforms the input records from the data flow into different outputs and writes them back to the data flow.

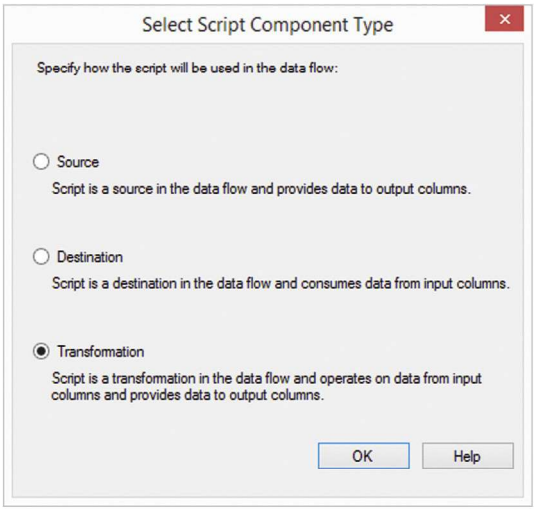

FIGURE 11.22

Select script component type.

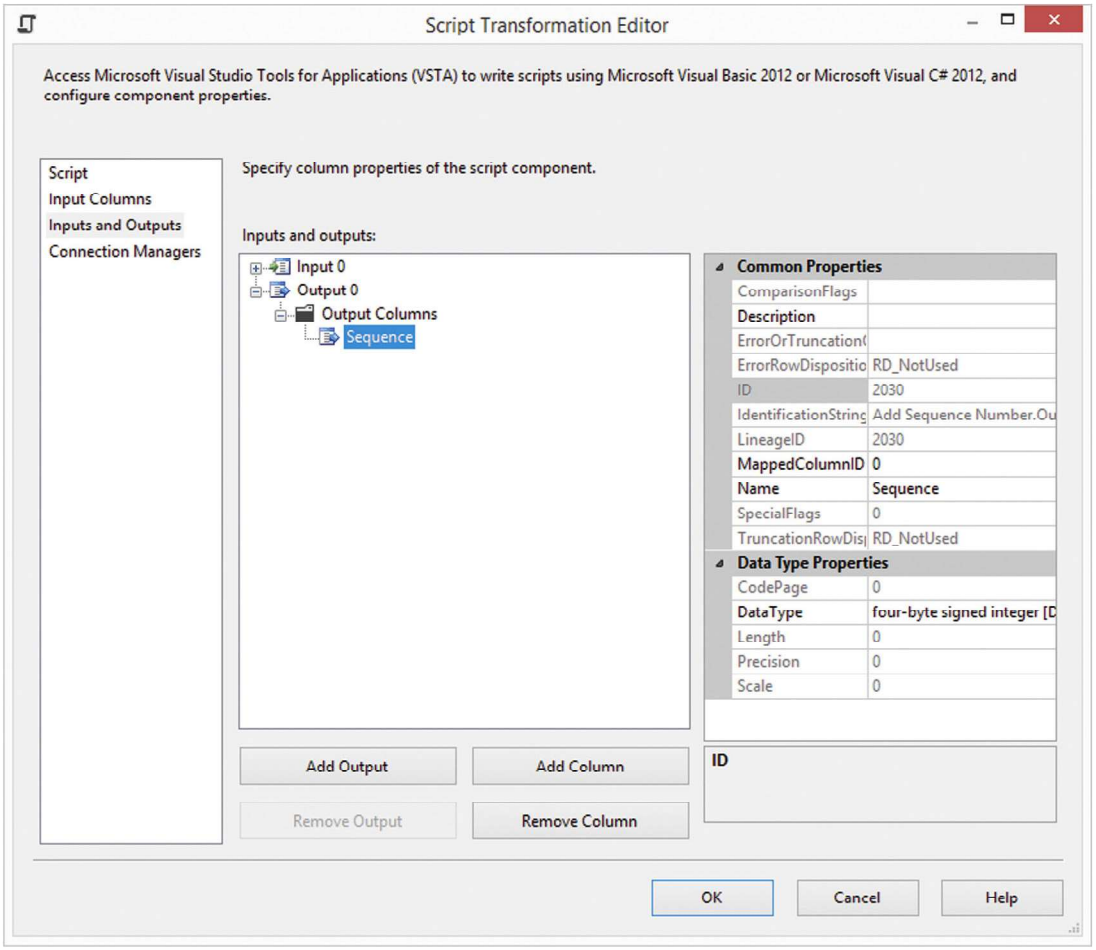

FIGURE 11.23

Script transformation editor to create a sequence output.

Because the goal of this step is to attach a sequence number to the data flow, select **transformation** before pressing the **OK** button. Connect the data flow of the **flat file source** to the script component. Open the **Script Transformation Editor** and add a new column on the **Inputs and Outputs** tab (Figure 11.23).

Select the folder **Output Columns** of the first output called **Output 0**. Select the **Add Column** button below the tree view. Name the output column **Sequence** and set the **DataType** property to

four-byte signed integer [DT_I4]. Switch back to the **Script** tab and select the **Edit Script…** button on the bottom right of the dialog. Replace the **Input0_ProcessInputRow** function by the following code:

```
private int seq = 1;
public override void Input0_ProcessInputRow(Input0Buffer Row)
{
    Row.Sequence = this.seq++;
}
```

This code introduces a new **seq** field that is increased for each row and returned into the data flow. Close the script and the script transformation editor.

The next step is to add the **record source** and the **load date**. Adding them is fairly simple: a **Derived Column** component can be added to the data flow. This is due to the simplicity of the required calculations. Drag a **Derived Column** component on the data flow canvas and connect it to the previous script component. Add the derived columns as shown in Table 11.22.

The first derived column creates a **LoadDate** column in the data flow. This column retrieves the load date from the variable, which was set in the control flow before the data flow was started. The **RecordSource** column is set statically to a detailed string that can be used for debugging purposes.

In addition, the same component can be used to prepare the hash calculations. For each hub and link in the target Data Vault model as shown in Figure 11.1, a hash key is required that is based on the business key or business key relationship in the model. There are five hubs and two links in the model. But the source file provides multiple columns that are mapped to the same target entity. For example, both **origin** and **dest**ination airports are mapped to the **HubAirport**. For that reason, multiple columns have to be hashed. Add the derived columns as shown in Table 11.23, which will represent the various inputs for the hash function.

Note that some of the input columns are converted to Unicode strings by the expression **(DT_WSTR,X)** where X is the configured length of the output string. All elements in the business key (BK) or the satellite payload (PL) for descriptive data are checked for NULL values by using the **REPLACENULL** function. Note the combination of case-insensitive business keys and case-sensitive payload in the last two derived columns.

The delimiter is hard-coded into the derived column because it can only be changed later with much effort. Changing the delimiter from semicolon to another character requires full reloading of the data warehouse. Also, it might be required to improve the expressions to support additional data types (such

Table 11.22 Derived Columns Required for Staging Process

Derived Column Name	Expression	Data Type
LoadDate	@[User::dLoadDate]	date
RecordSource	“BTS.OnTimeOnTimePerformance”	Unicode string

Table 11.23 Additional Derived Columns Required for Hashing in the Staging Process

Derived Column Name	Expression
FlightNumHubBK	UPPER(TRIM(REPLACENULL(Carrier,“”)) + “;” + TRIM((DT_WSTR,5)REPLACENULL(FlightNum,“”)))
OriginHubBK	UPPER(TRIM(REPLACENULL(Origin,“”)))
CarrierHubBK	UPPER(TRIM(REPLACENULL(Carrier,“”)))
TailNumHubBK	UPPER(TRIM(REPLACENULL(TailNum,“”)))
DestHubBK	UPPER(TRIM(REPLACENULL(Dest,“”)))
Div1AirportHubBK	UPPER(TRIM(REPLACENULL(Div1Airport,“”)))
Div2AirportHubBK	UPPER(TRIM(REPLACENULL(Div2Airport,“”)))
Div3AirportHubBK	UPPER(TRIM(REPLACENULL(Div3Airport,“”)))
Div4AirportHubBK	UPPER(TRIM(REPLACENULL(Div4Airport,“”)))
Div5AirportHubBK	UPPER(TRIM(REPLACENULL(Div5Airport,“”)))
FlightLinkBK	UPPER(TRIM(REPLACENULL((DT_WSTR,2)Carrier,“”)) + “;” + TRIM((DT_WSTR,5)REPLACENULL(FlightNum,“”)) + “;” + TRIM(REPLACENULL((DT_WSTR,10)TailNum,“”)) + “;” + TRIM(REPLACENULL((DT_WSTR,3)Origin,“”)) + “;” + TRIM(REPLACENULL((DT_WSTR,3)Dest,“”)) + “;” + TRIM((DT_WSTR,27)REPLACENULL(FlightDate,“”)))
Div1FlightLinkBK	UPPER(TRIM((DT_WSTR,5)REPLACENULL(FlightNum,“”)) + “;” + TRIM((DT_WSTR,10)REPLACENULL(Div1TailNum,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(Origin,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(Div1Airport,“”)) + “;” + “1” + “;” + TRIM((DT_WSTR,27)REPLACENULL(FlightDate,“”)))
Div2FlightLinkBK	UPPER(TRIM((DT_WSTR,5)REPLACENULL(FlightNum,“”)) + “;” + TRIM((DT_WSTR,10)REPLACENULL(Div2Tailnum,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(Origin,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(Div2Airport,“”)) + “;” + “2” + “;” + TRIM((DT_WSTR,27)REPLACENULL(FlightDate,“”)))
Div3FlightLinkBK	UPPER(TRIM((DT_WSTR,5)REPLACENULL(FlightNum,“”)) + “;” + TRIM((DT_WSTR,10)REPLACENULL(Div3TailNum,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(Origin,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(Div3Airport,“”)) + “;” + “3” + “;” + TRIM((DT_WSTR,27)REPLACENULL(FlightDate,“”)))
Div4FlightLinkBK	UPPER(TRIM((DT_WSTR,5)REPLACENULL(FlightNum,“”)) + “;” + TRIM((DT_WSTR,10)REPLACENULL(Div4TailNum,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(Origin,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(Div4Airport,“”)) + “;” + “4” + “;” + TRIM((DT_WSTR,27)REPLACENULL(FlightDate,“”)))
Div5FlightLinkBK	UPPER(TRIM((DT_WSTR,5)REPLACENULL(FlightNum,“”)) + “;” + TRIM((DT_WSTR,10)REPLACENULL(Div5TailNum,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(Origin,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(Div5Airport,“”)) + “;” + “5” + “;” + TRIM((DT_WSTR,27)REPLACENULL(FlightDate,“”)))

Table 11.23 Additional Derived Columns Required for Hashing in the Staging Process *(cont.)*

Derived Column Name	Expression
FlightNumCarrierLinkBK	UPPER(TRIM(REPLACENULL(Carrier,“”)) + “;” + TRIM((DT_WSTR,5)REPLACENULL(FlightNum,“”)) + “;” + TRIM(REPLACENULL(Carrier,“”)))
OriginAirportSatPL	UPPER(TRIM(REPLACENULL(Origin,“”))) + “;” + TRIM(REPLACENULL(OriginCityName,“”)) + “;” + TRIM(REPLACENULL(OriginState,“”)) + “;” + TRIM(REPLACENULL(OriginStateName,“”)) + “;” + TRIM((DT_WSTR,5)REPLACENULL(OriginCityMarketID,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(OriginStateFips,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(OriginWac,“”))
DestAirportSatPL	UPPER(TRIM(REPLACENULL(Dest,“”))) + “;” + TRIM(REPLACENULL(DestCityName,“”)) + “;” + TRIM(REPLACENULL(DestState,“”)) + “;” + TRIM(REPLACENULL(DestStateName,“”)) + “;” + TRIM((DT_WSTR,5)REPLACENULL(DestCityMarketID,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(DestStateFips,“”)) + “;” + TRIM((DT_WSTR,3)REPLACENULL(DestWac,“”))

as Boolean and float values) and to standardize on the date format to be used: it is recommended to use ISO-8601 for converting date timestamps into character strings.

The Div1FlightLinkBK to Div5FlightLinkBK columns include a constant value that helps to unpivot the incoming data. If the flight was diverted multiple times, the diversion information is provided in up to five sets of columns. When loading the data into the Raw Data Vault, it is loaded into a single link table with up to five records per incoming row.

The final setup is shown in Figure 11.24.

The new columns are shown in the grid on the bottom of the dialog. Select **OK** to close the dialog. The next step is to calculate the hash values for all business keys, their relationships and for descriptive attributes that should be added to the same satellites. Multiple options have been discussed in section 11.2. This section describes how to use a **Script Component** for hashing the inputs.

Drag another **Script Component** to the data flow canvas and connect it to the previous **Derived Column** transformation. Switch to the **Input Columns** tab and check all available input columns in the default input **Input 0**. Switch to the **Inputs and Outputs** tab. For each derived column that ends with **HubBK**, **LinkBK** or **SatPL**, add a corresponding **HashKey** or **HashDiff** column. For example, for **FlightNumHubBK**, add an output column **FlightNumHashKey** with a string (non-unicode) of length 32 and code page 1252. Because the hash value is based only on characters from 0 to 9 and A to F, a Western European code page, such as 1252 or even ASCII, is sufficient. The important step here is to define the one to be used and implement it consistently.

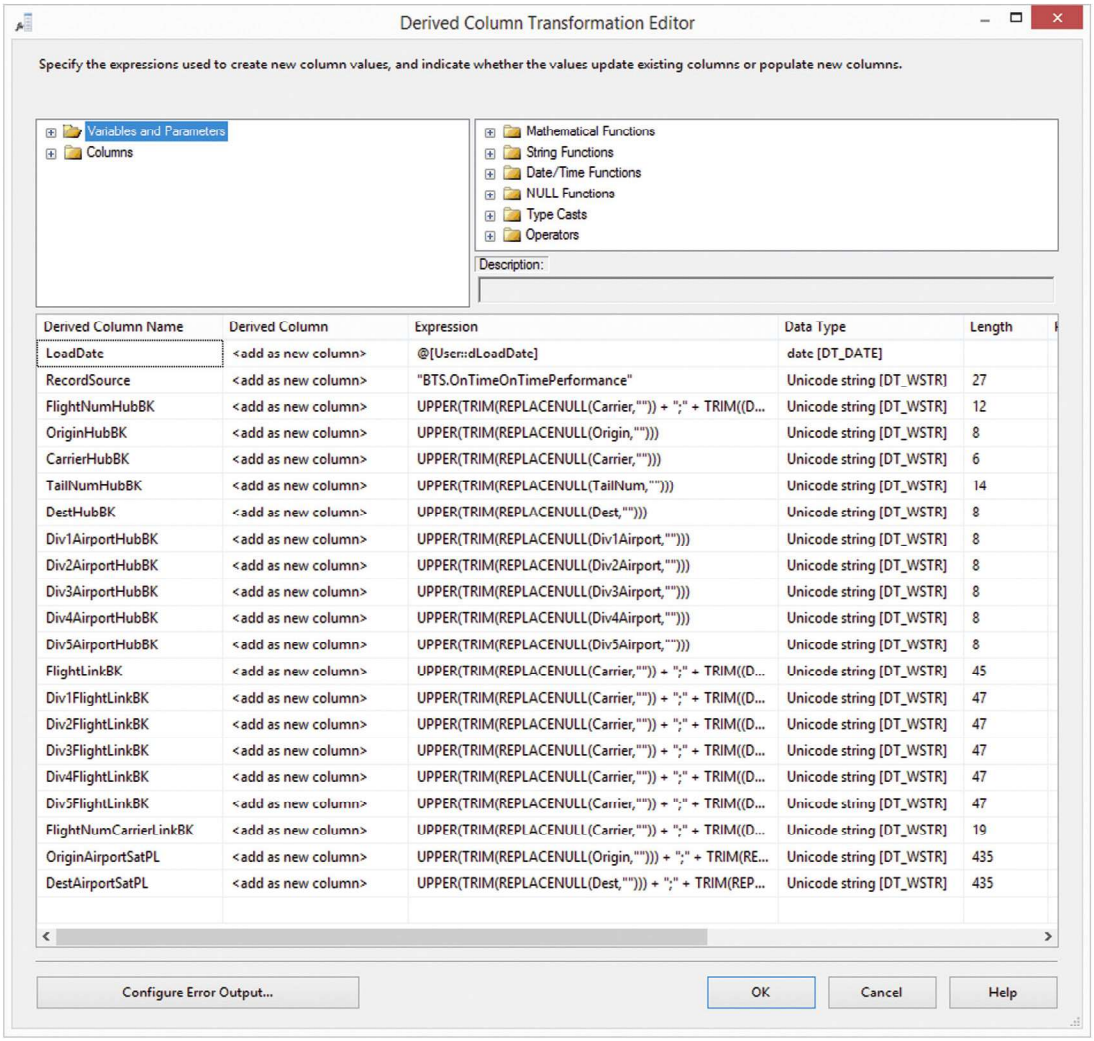

FIGURE 11.24

Derived column transformation editor to setup the system-generated columns.

Following this naming convention is important for the following script to work, because it hashes all input columns ending with either **HubBK**, **LinkBK** or **SatPL** and writes them into an output column with exactly the same name, ending with **HashKey** or **HashDiff** instead. This approach requires no programming at all; all configuration is done graphically in the Script Transformation Editor (Figure 11.25).

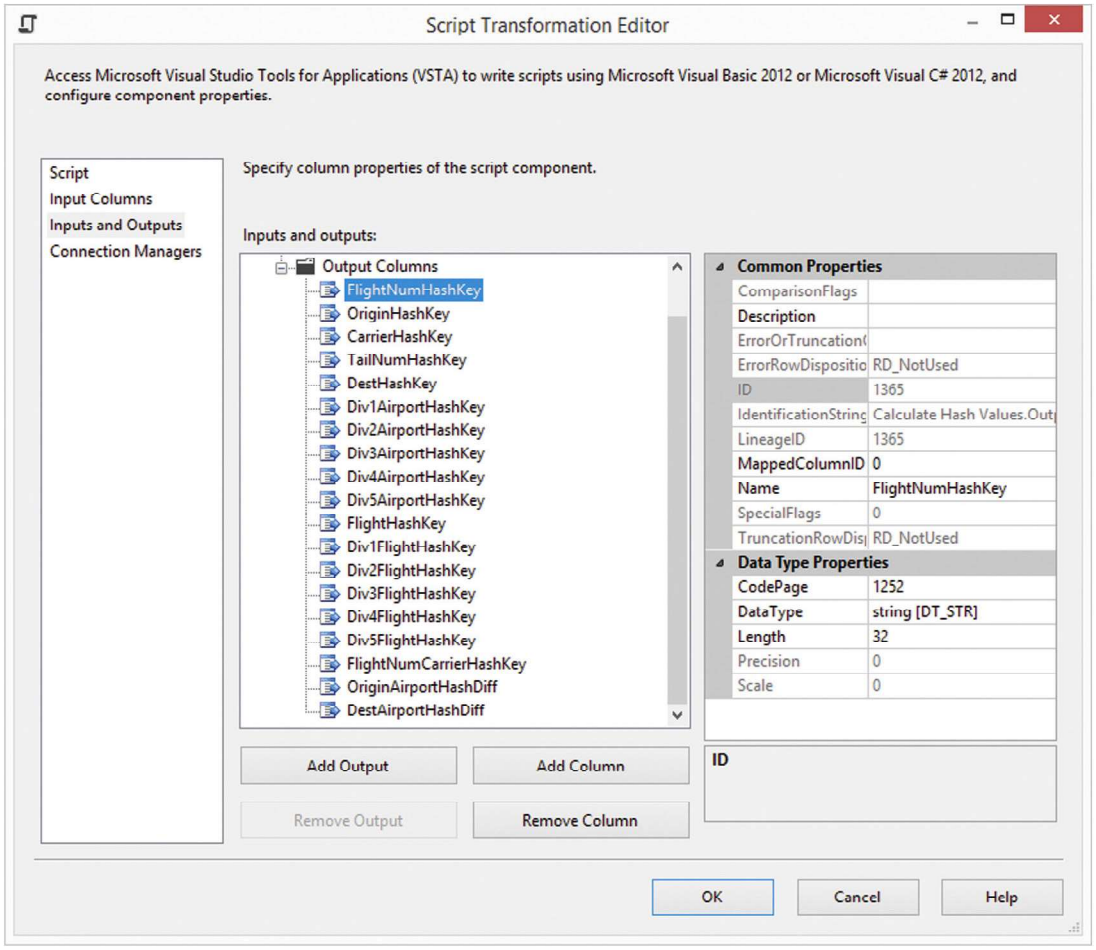

FIGURE 11.25

Output columns in the script transformation editor for applying the hash function.

Switch back to the **Script** tab and enter the following script using the **Edit Script…** button:

```
[Microsoft.SqlServer.Dts.Pipeline.SSISScriptComponentEntryPointAttribute]
public class ScriptMain : UserComponent
{
    MD5 md5 = System.Security.Cryptography.MD5.Create();
    System.Text.UnicodeEncoding encoding = new System.Text.UnicodeEncoding(false, false);

    /// <summary>
    /// This method is called once for every row that passes through the
    /// component from Input0.
    /// </summary>
    /// <param name="Row">The row that is currently passing through the
    /// component</param>
```

```
    public override void Input0_ProcessInputRow(Input0Buffer Row)
    {
        Type rowType = Row.GetType();
        String columnValue = "";
        String oColumnName = "";

        foreach (IDTSInputColumn100 iColumn in
            this.ComponentMetaData.InputCollection[0].InputColumnCollection) {

            if (iColumn.Name.EndsWith("HubBK")
                || iColumn.Name.EndsWith("LinkBK")
                || iColumn.Name.EndsWith("SatPL"))
            {
                oColumnName = iColumn.Name.Replace("HubBK", "HashKey");
                oColumnName = oColumnName.Replace("LinkBK", "HashKey");
                oColumnName = oColumnName.Replace("SatPL", "HashDiff");

                columnValue =
                    rowType.GetProperty(iColumn.Name).GetValue(Row, null).ToString();
                columnValue = BitConverter.ToString(
                    md5.ComputeHash(encoding.GetBytes(columnValue)));
                columnValue = columnValue.Replace("-", "");
                if (rowType.GetProperty(oColumnName) != null)
                    rowType.GetProperty(oColumnName).SetValue(Row, columnValue, null);
            }
        }
    }

}
```

This script traverses through all input columns, checks if their names end with either **HubBK**, **LinkBK** or **SatPL**, replaces these suffixes with **HashKey** or **HashDiff** and applies the MD5 hash function before storing the hash value in the output column. In order to make this script compile successfully, a reference to the assembly **System.Security** is required.

Note that this script implements a simplified hash diff calculation. It has not implemented the improved strategy to support changing satellite structures as described in section 11.2.5 with zero maintenance. In order to do so, any delimiters at the end of the input strings for hash diffs (columns ending with "SatPL") have to be removed from the input.

The last step is to set up the OLE DB destination to write the data flow into a staging table. Before doing so, create the target table in the staging area by executing the following script:

```
CREATE TABLE [bts].[OnTimeOnTimePerformance](
       [Sequence] [int] NOT NULL,
       [Year] [smallint] NULL,
       [Quarter] [smallint] NULL,
       [Month] [smallint] NULL,
```

```
[DayofMonth] [smallint] NULL,
[DayOfWeek] [smallint] NULL,
[FlightDate] [datetime] NULL,
[UniqueCarrier] [nvarchar](2) NULL,
[AirlineID] [smallint] NULL,
[Carrier] [nvarchar](2) NULL,
[TailNum] [nvarchar](6) NULL,
[FlightNum] [smallint] NULL,
[OriginAirportID] [smallint] NULL,
[OriginAirportSeqID] [int] NULL,
[OriginCityMarketID] [int] NULL,
[Origin] [nvarchar](3) NULL,
[OriginCityName] [nvarchar](100) NULL,
[OriginState] [nvarchar](2) NULL,
[OriginStateFips] [smallint] NULL,
[OriginStateName] [nvarchar](100) NULL,
[OriginWac] [smallint] NULL,
[DestAirportID] [smallint] NULL,
[DestAirportSeqID] [int] NULL,
[DestCityMarketID] [int] NULL,
[Dest] [nvarchar](3) NULL,
[DestCityName] [nvarchar](100) NULL,
[DestState] [nvarchar](2) NULL,
[DestStateFips] [smallint] NULL,
[DestStateName] [nvarchar](100) NULL,
[DestWac] [smallint] NULL,
[CRSDepTime] [smallint] NULL,
[DepTime] [smallint] NULL,
[DepDelay] [smallint] NULL,
[DepDelayMinutes] [smallint] NULL,
[DepDel15] [bit] NULL,
[DepartureDelayGroups] [smallint] NULL,
[DepTimeBlk] [nvarchar](9) NULL,
[TaxiOut] [smallint] NULL,
[WheelsOff] [smallint] NULL,
[WheelsOn] [smallint] NULL,
[TaxiIn] [smallint] NULL,
[CRSArrTime] [smallint] NULL,
[ArrTime] [smallint] NULL,
[ArrDelay] [smallint] NULL,
[ArrDelayMinutes] [smallint] NULL,
[ArrDel15] [bit] NULL,
[ArrivalDelayGroups] [smallint] NULL,
[ArrTimeBlk] [nvarchar](9) NULL,
[Cancelled] [bit] NULL,
[CancellationCode] [nvarchar](10) NULL,
[Diverted] [bit] NULL,
[CRSElapsedTime] [smallint] NULL,
[ActualElapsedTime] [smallint] NULL,
[AirTime] [smallint] NULL,
[Flights] [smallint] NULL,
[Distance] [int] NULL,
[DistanceGroup] [int] NULL,
[CarrierDelay] [smallint] NULL,
```

```
[WeatherDelay] [smallint] NULL,
[NASDelay] [smallint] NULL,
[SecurityDelay] [smallint] NULL,
[LateAircraftDelay] [smallint] NULL,
[FirstDepTime] [smallint] NULL,
[TotalAddGTime] [smallint] NULL,
[LongestAddGTime] [smallint] NULL,
[DivAirportLandings] [smallint] NULL,
[DivReachedDest] [bit] NULL,
[DivActualElapsedTime] [smallint] NULL,
[DivArrDelay] [smallint] NULL,
[DivDistance] [int] NULL,
[Div1Airport] [nvarchar](3) NULL,
[Div1AirportID] [smallint] NULL,
[Div1AirportSeqID] [smallint] NULL,
[Div1WheelsOn] [smallint] NULL,
[Div1TotalGTime] [smallint] NULL,
[Div1LongestGTime] [smallint] NULL,
[Div1WheelsOff] [smallint] NULL,
[Div1TailNum] [nvarchar](6) NULL,
[Div2Airport] [nvarchar](3) NULL,
[Div2AirportID] [smallint] NULL,
[Div2AirportSeqID] [smallint] NULL,
[Div2WheelsOn] [smallint] NULL,
[Div2TotalGTime] [smallint] NULL,
[Div2LongestGTime] [smallint] NULL,
[Div2WheelsOff] [smallint] NULL,
[Div2TailNum] [nvarchar](6) NULL,
[Div3Airport] [nvarchar](3) NULL,
[Div3AirportID] [smallint] NULL,
[Div3AirportSeqID] [smallint] NULL,
[Div3WheelsOn] [smallint] NULL,
[Div3TotalGTime] [smallint] NULL,
[Div3LongestGTime] [smallint] NULL,
[Div3WheelsOff] [smallint] NULL,
[Div3TailNum] [nvarchar](6) NULL,
[Div4Airport] [nvarchar](3) NULL,
[Div4AirportID] [smallint] NULL,
[Div4AirportSeqID] [smallint] NULL,
[Div4WheelsOn] [smallint] NULL,
[Div4TotalGTime] [smallint] NULL,
[Div4LongestGTime] [smallint] NULL,
[Div4WheelsOff] [smallint] NULL,
[Div4TailNum] [nvarchar](6) NULL,
[Div5Airport] [nvarchar](3) NULL,
[Div5AirportID] [smallint] NULL,
[Div5AirportSeqID] [smallint] NULL,
[Div5WheelsOn] [smallint] NULL,
[Div5TotalGTime] [smallint] NULL,
[Div5LongestGTime] [smallint] NULL,
[Div5WheelsOff] [smallint] NULL,
[Div5TailNum] [nvarchar](6) NULL,
```

```
	[LoadDate] [datetime] NOT NULL,
	[RecordSource] [nvarchar](27) NOT NULL,
	[FlightNumHashKey] [char](32) NOT NULL,
	[OriginHashKey] [char](32) NOT NULL,
	[CarrierHashKey] [char](32) NOT NULL,
	[TailNumHashKey] [char](32) NOT NULL,
	[DestHashKey] [char](32) NOT NULL,
	[Div1AirportHashKey] [char](32) NOT NULL,
	[Div2AirportHashKey] [char](32) NOT NULL,
	[Div3AirportHashKey] [char](32) NOT NULL,
	[Div4AirportHashKey] [char](32) NOT NULL,
	[Div5AirportHashKey] [char](32) NOT NULL,
	[FlightHashKey] [char](32) NOT NULL,
	[Div1FlightHashKey] [char](32) NOT NULL,
	[Div2FlightHashKey] [char](32) NOT NULL,
	[Div3FlightHashKey] [char](32) NOT NULL,
	[Div4FlightHashKey] [char](32) NOT NULL,
	[Div5FlightHashKey] [char](32) NOT NULL,
	[FlightNumCarrierHashKey] [char](32) NOT NULL,
	[OriginAirportHashDiff] [char](32) NOT NULL,
	[DestAirportHashDiff] [char](32) NOT NULL,
 CONSTRAINT [PK_OnTimeOnTimePerformance] PRIMARY KEY NONCLUSTERED
(
	[Sequence] ASC,
	[LoadDate] ASC
) ON [INDEX]
) ON [DATA]
```

In this script, all strings are configured as nvarchar to allow other languages to be added later on. It is also in line with the flat file source in the data flow, because the input columns have been configured as Unicode strings. The table is created in a schema called **bts**. The schemas as used in this chapter follow the convention that each source system uses its own schema.

The final step is to set up the destination. Drag an **OLE DB Destination** component to the data flow canvas and connect it to the **Script Component** that calculates the hash values. In the **OLE DB Destination Editor**, create a new connection manager. Set up the new connection to the **StageArea** database as shown in Figure 11.26.

After selecting the OK button, make sure the connection is taken over to the **OLE DB destination editor** and select the **OnTimeOnTimePerformance** table in the **bts** namespace as the destination (Figure 11.27).

Make sure to **keep null** values by activating the option. You might want to configure the other options, such as **rows per batch**. Switch to the **Mappings** tab and make sure that all columns from the data flow are mapped correctly to the destination (Figure 11.28).

Scroll down and make sure that the **HubBK**, **LinkBK** and **SatPL** columns have not found a destination column. They are only used in the data flow but not written to the destination. Instead, the corresponding hash key or hash diff value is written.

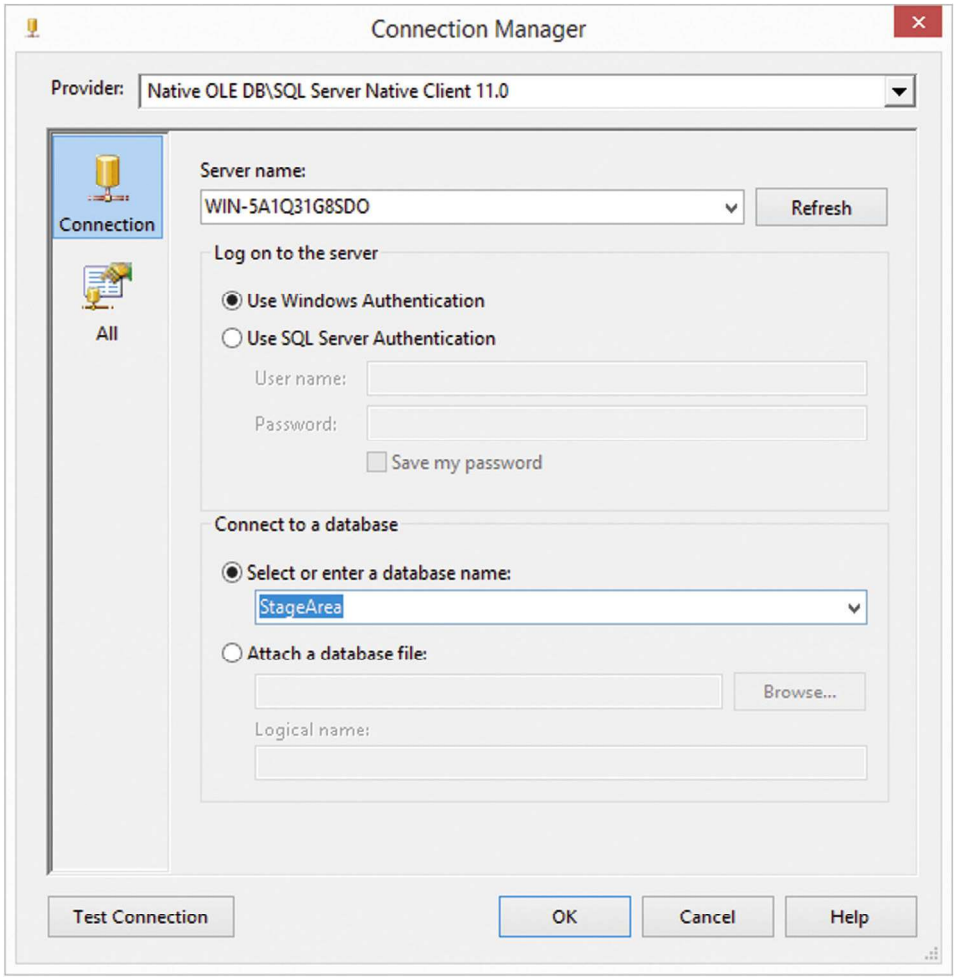

FIGURE 11.26

Setup connection manager.

Run the control flow by pressing the start button in the toolbar. After moving the data, open Microsoft SQL Server Management Studio and execute the following SQL statement to compare the hash calculation in TSQL with the one performed in SSIS:

```
SELECT TOP 10
       OriginHashKey AS OriginHashKey_SSIS,
       UPPER(CONVERT(char(32),HASHBYTES('MD5',
              UPPER(RTRIM(LTRIM(COALESCE(Origin, ''))))
       ),2)) AS OriginHashKey_TSQL,

       OriginAirportHashDiff AS OriginAirportHashDiff_SSIS,
       UPPER(CONVERT(char(32),HASHBYTES('MD5',
```

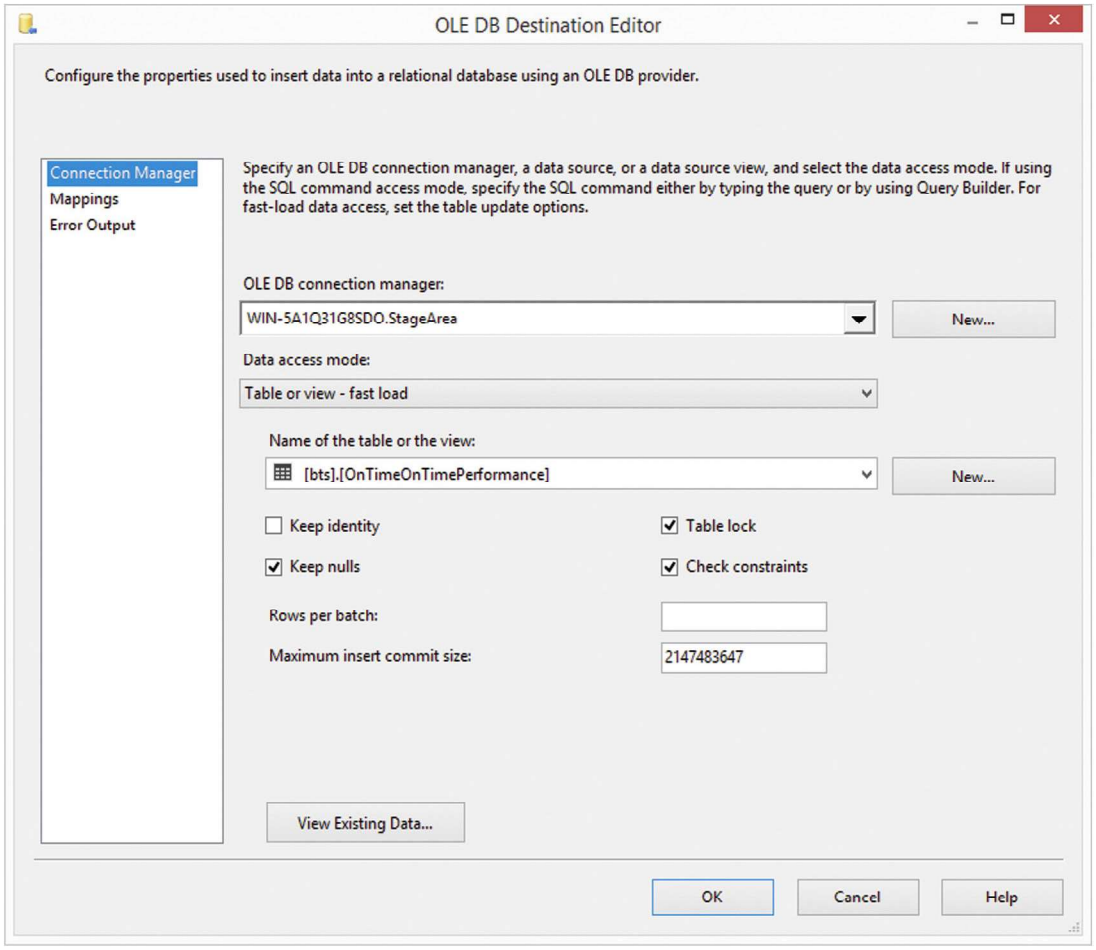

FIGURE 11.27

Set up OLE DB destination.

```
                CONCAT(
                        UPPER(RTRIM(LTRIM(COALESCE(Origin, '')))), ';',
                        RTRIM(LTRIM(COALESCE(OriginCityName, ''))), ';',
                        RTRIM(LTRIM(COALESCE(OriginState, ''))), ';',
                        RTRIM(LTRIM(COALESCE(OriginStateName, ''))), ';',
                        RTRIM(LTRIM(COALESCE(OriginCityMarketID, ''))), ';',
                        RTRIM(LTRIM(COALESCE(OriginStateFips, ''))), ';',
                        RTRIM(LTRIM(COALESCE(OriginWac, '')))
                )
        ),2)) AS OriginAirportHashDiff_TSQL
FROM
        [StageArea].[bts].[OnTimeOnTimePerformance]
```

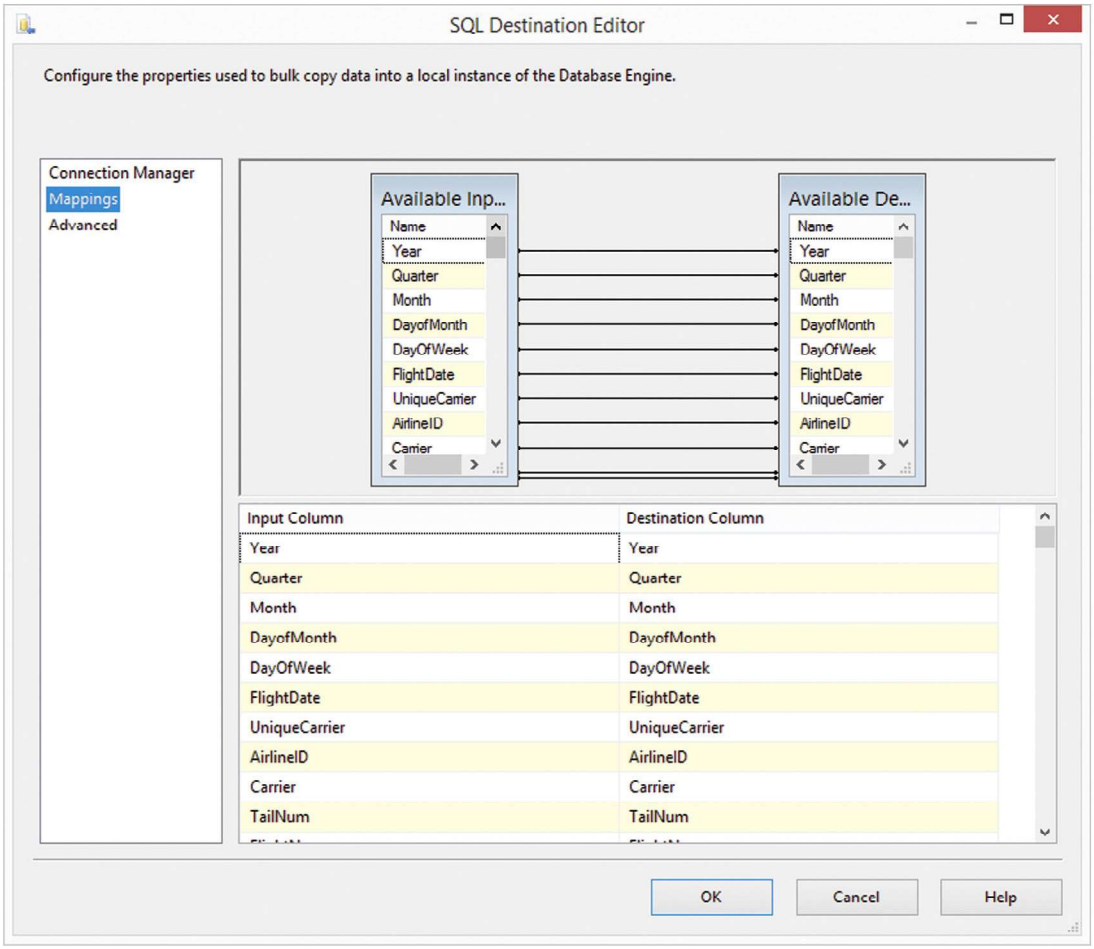

FIGURE 11.28

Map input columns to destination.

The statement should return four columns. The first two columns should return the **OriginHashKey** and provide exactly the same result. The last two columns should produce the same **OriginAirportHashDiff** value.

Note that there are two options to convert the columnValue variable in line 32 of the script used in the script component to calculate the hash values. This option is set in line 5 when the **System.Text.UnicodeEncoding** class is initialized:

```
System.Text.UnicodeEncoding encoding = new System.Text.UnicodeEncoding(false, false);
```

The first parameter in the constructor allows to set little endian (false) or big endian (true) for the conversion. When using big endian, the hash values produced in SSIS are different from those produced in T-SQL. Also make sure that the byte-order mark (the second parameter is set to false).

The control and data flows to source flat files into the staging area are now complete. Note that the Metrics Mart and Error Mart have been left out of the discussion by intention, due to space restrictions. In order to go productive, redirect errors into the Error Mart as shown in Chapter 10, Metadata Management, and capture base metrics in the same manner as discussed in Chapter 10 as well.

11.7 SOURCING HISTORICAL DATA

The previous section described the loading of flat file data sources. However, one special case remains that needs some thought. Before the data warehouse is put into production and loads data on a regular schedule, historical data is often loaded. The historical data often comes from the source application itself, from the archive or from a legacy data warehouse. Sourcing historical data allows the business to analyze trends that have started before the data warehouse has been put in place. Therefore, loading historical data is a common task in data warehousing.

Section 11.3 has described how the load date is applied to incoming data in order to identify the batch that loaded the data. When using this approach without modifications for loading historical data, a problem is incorporated into the loading process. Because all historical data is loaded in one batch, or at least in multiple batches around the same time, there is a risk that historical data cannot be loaded into the Data Vault any longer. If the same load date is used for all historical data, it would actually become impossible to load the data. The reason behind this problem is that the load date is part of the Data Vault 2.0 model because it is included in the primary key of satellites. Consider the three historical source files in Figure 11.29.

FIGURE 11.29

Historical source files generated on various dates.

All three files contain address data for the same passenger. The passenger has moved over time. Because this historical data is of interest for the business users, it needs to be loaded into the data warehouse before going into production. If the initial data load, based on the historical data, is performed on January 1, 2015, the load dates for all three documents would be set to this date. Consider the effects on the satellite **SatPassengerAddress** shown in Table 11.24.

The problem is that the load date in Table 11.24 has been set to the same date. In fact, loading this data into the satellite table will not work, because the primary key is made up of the passenger hash key column and the load date column (refer to Chapter 4). This combination has to be unique to meet the requirements of the primary key. These records would be in violation with these requirements because the combination is always the same for all three rows.

To resolve this issue, there are two options: either load the data in three different batches, or set an artificial load date. The first solution would work from a technical perspective, but from a design perspective, there should be no difference between historical data and actual data. The data warehouse should pretend that the historical data was loaded just as the actual data on the day it was generated (or at least near the date). In the case of the example shown in Table 11.24, all historical data seems to be loaded on January 1, 2015. This is true, but not the desired view of the business.

The second approach is to set the load date to an artificial date. This is the preferred solution when loading archived historical data. The load date is set to the date it would have received if the historical data had been loaded in the past. By doing so, it simulates that the data warehouse would have been in place and the loading procedures would have sourced the file in the past. This is the only exception to the rule that the load date should not be a source-system generated date, because we're using the date of generation, in many cases the extract date. On the other hand, this is a one-off historical load and the load date is only derived from the historical date of generation, but under full control of the data warehouse. For example, the data warehouse team might decide to override a load date derived from the historic date of generation during the initial node when required. For all these reasons, the date of generation could be used as the load date (Table 11.25).

Note that the time part of the load date has been set to 00:00:00.000. There are two reasons for this decision: first, in many cases, the actual time when the historical data has been extracted from the source system is unknown because the file name or some log file provides only the date of extraction. The second reason is that the convention to set the time part of the load date for historical data helps to distinguish the historical data from the actual data which is still desired in some cases, for example

Table 11.24 Erroneous SatPassengerAddress Satellite

Passenger HashKey	Load Date	Load End Date	Record Source	Address	City	Zip	State	Country
8473d2a…	**2015-01-01 08:34:12**	2015-01-01 08:34:12.999	Domestic Flight	31 Main St.	Norman	30782	OK	USA
8473d2a…	**2015-01-01 08:34:12**	2015-01-01 08:34:12.999	Domestic Flight	9612 Lincoln Road	Santa Clara	70831	CA	USA
8473d2a…	**2015-01-01 08:34:12**	9999-12-31 24:59:59.999	Domestic Flight	2050 1st street	San Francisco	94114	CA	USA

Table 11.25 Corrected Satellite Data for SatPassengerAddress

Passenger HashKey	Load Date	Load End Date	Record Source	Address	City	Zip	State	Country
8473d2a…	**1995-06-26 00:00:00.000**	2000-05-20 23:59:59.999	Domestic Flight	31 Main St.	Norman	30782	OK	USA
8473d2a…	**2000-05-20 00:00:00.000**	2007-04-21 23:59:59.999	Domestic Flight	9612 Lincoln Road	Santa Clara	70831	CA	USA
8473d2a…	**2007-04-21 00:00:00.000**	9999-12-31 23:59:59.999	Domestic Flight	2050 1st street	San Francisco	94114	CA	USA

for debugging purposes. Following this recommendation is a compromise between the ability to distinguish the data from each other and the desire to load the historical data as it would have been loaded in the past.

11.7.1 SSIS EXAMPLE FOR SOURCING HISTORICAL DATA

Because the load date is set in the staging area, it has to be overridden in its loading processes when historical data should be loaded. In section 11.6, the current date was set as the load date by writing the timestamp into a SSIS variable using a script task. This script task needs to be modified for loading historical data. The goal is to develop a SSIS package that is able to handle both types of data (historical data for the initial load and daily loads).

The first step is to allow read-only access to the current file name in the forloop container. Open the **Script Task Editor** and select the ellipsis button to modify the **ReadOnlyVariables** (Figure 11.30).

In the following dialog, add the **Users::sFileName** variable which holds the current file name of the container that traverses over all files in the source directory.

The second step is to modify the source code of the script task. Instead of setting the current date, the following source is used to extract the date of extraction from the filename:

```
public partial class ScriptMain : VSTARTScriptObjectModelBase
        {
        private static string pattern =
                @".*On_Time_On_Time_Performance_(\d{4})_(\d{1,2})_(\d{1,2}).*";
        private Regex r = new Regex(pattern, RegexOptions.IgnoreCase);

        /// <summary>
        /// This method is called when this script task executes in the control flow.
        /// Before returning from this method, set the value of Dts.TaskResult to indicate
        /// success or failure.
        /// </summary>
        public void Main()

        {
             Match m = r.Match(Dts.Variables["User::sFileName"].Value.ToString());
             if (m.Success && m.Groups.Count >= 4)
```

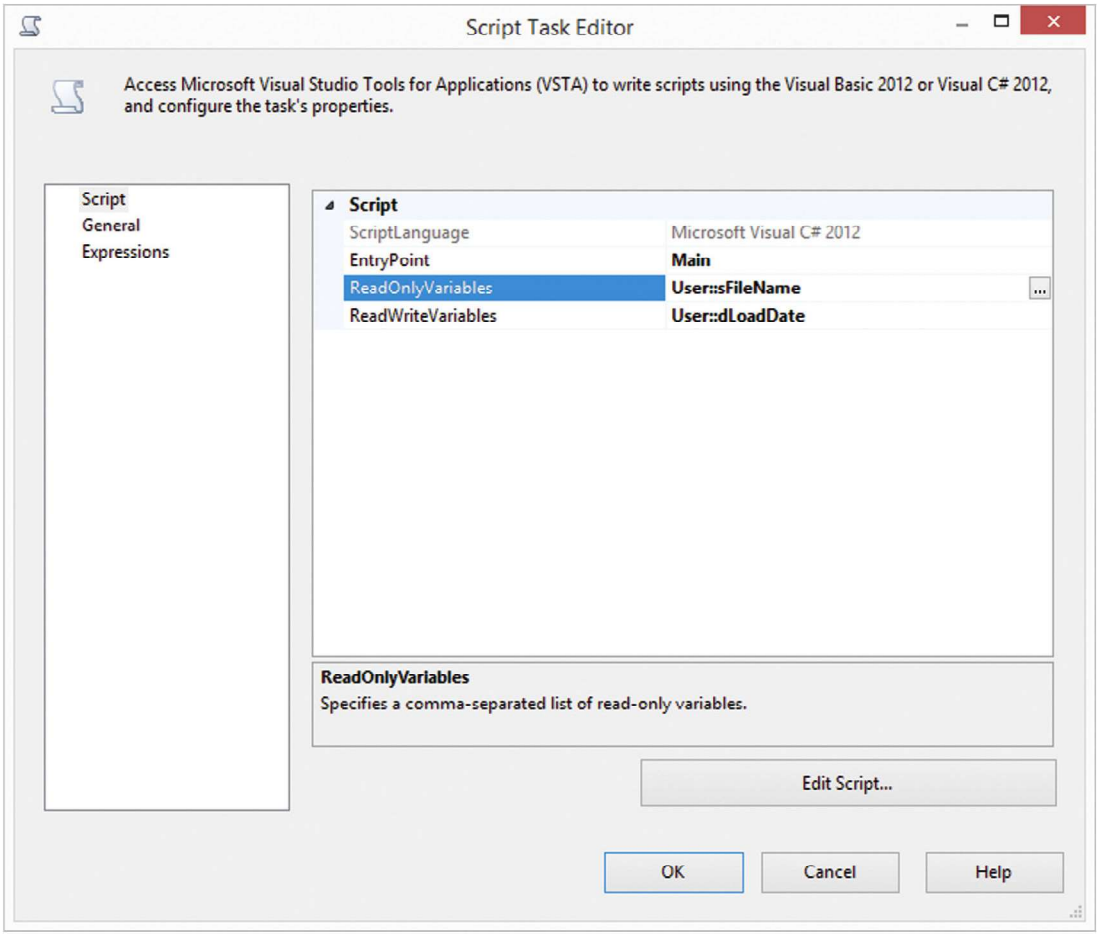

FIGURE 11.30

Script task editor to add ReadOnlyVariables.

```
            {
                int year = int.Parse(m.Groups[1].Value);
                int month = int.Parse(m.Groups[2].Value);
                int day = int.Parse(m.Groups[3].Value);

                Dts.Variables["User::dLoadDate"].Value = new DateTime(year, month, day);
            }
            else
            {
                // report error to the Error Mart
            }

            Dts.TaskResult = (int)ScriptResults.Success;
        }
    }
```

The year, month and day are extracted from the source file name using a regular expression that captures dates from file names with YYYY_MM_DD formatted dates included in the file name. These expressions search for patterns within strings and are useful to extract information from semi-structured text such as the file names. If an error occurs, the error is reported to the Error Mart.

11.8 SOURCING THE SAMPLE AIRLINE DATA

The BTS data is provided as Google Sheets on Google Drive. The folder is accessible by the general public under the following link: http://goo.gl/TQ1R63. We will load the data from the Google Sheet files into the staging area of the local data warehouse.

Microsoft SSIS does not provide a component for easy access to files on Google Drive. However, it is possible to extend SSIS by Google Sheets components that can directly access live Google Sheets. In order to do so, download the following products from CData.com:

- **CData ADO.NET Provider for Google Apps:** The CData ADO.NET Provider for Google Apps gives developers the power to easily connect .NET applications to popular Google Services including Google Docs, Google Calendar, Google Talk, Search, and more. We will use the components to load all Google Sheets from the public folder on Google Drive [32]. The product can be found here: http://www.cdata.com/drivers/google/ado/
- **CData SSIS Components for Google Spreadsheets:** Powerful SSIS Source & Destination Components that allows you to easily connect SQL Server with live Google Spreadsheets through SSIS Workflows. We will use the Google Spreadsheets Data Flow Components to source the airline data from Google Sheets [33]. The product can be downloaded from http://www.cdata.com/drivers/gsheets/ssis/

The vendor provides an extended free trial for the purpose of this book. Use the given trial subscription codes when installing the software on your development machine (Table 11.26).

Note: If you decide to buy the components for your projects, you can use the coupon code **DVAULT10** to receive a 10% discount on the software.

As an alternative, you can also download the files as zipped CSV files manually from Google Drive, store them unpacked in a local folder and load the files using the procedure described in the previous section.

The next sections describe how to traverse over Google Drive in order to find the location and name of the airline data spreadsheets. It then sources each spreadsheet into the staging area. Before doing so, access to the Google Drive account has to be granted using OAuth 2.0. Otherwise traversing files on Google Drive is not allowed due to security concerns.

Table 11.26 License Keys for CData Components Required to Download the Sample Data

Product Name: CData SSIS Components for Google Spreadsheets 2015 [RLSAA] License: EVALUATION COPY Product Key: `XRLSA-ASPST-D4FFD-2841W-MRREM`
Product Name: CData ADO.NET Provider for Google Apps 2015 [RGRAA] License: EVALUATION COPY Product Key: `XRGRA-AS1SR-V4FFD-2851T-ZWAFZ`

11.8.1 AUTHENTICATING WITH GOOGLE DRIVE

Before the control flow is modified in the next section, Google Drive has to be set up in order to provide the data to SSIS. This requires two steps:

1. Import the folder under http://goo.gl/TQ1R63 into your personal Google Drive account.
2. Allow SSIS to connect to your Google Drive account by registering it as a Desktop application.

The second step requires setting up a project under https://console.developers.google.com/project in order to enable authorized access for desktop applications. Sign-in to the console and create a new project (Figure 11.31).

Provide a unique project name and project id. Select **Create** to create the project in your account. Wait until the project has been created. Once the project dashboard shows up, configure the **consent screen** by selecting the corresponding menu item under **APIs & auth** (Figure 11.32).

The consent screen is presented to the end-user in order to grant access to the personal Google Drive account. No worries: this user will just be you and nobody else.

Enter some information that identifies the application to the user and select the **Save** button.

OAuth 2.0 authorization is required to access the entities on Google Drive that we need to access in order to traverse all the airline data spreadsheets. Without it, the SSIS job described in the next sections cannot find out which spreadsheets are available. Therefore, select the **Create new Client ID** button to advance to the next screen (Figure 11.33).

Depending on the application type, different mechanisms are used to authenticate the application to the user. There are different settings for Web applications, service accounts and applications installed on local desktops or handheld devices. From a cloud provider point-of-view, SSIS is an installed desktop application. Therefore, select the **Installed application** option and **Other** as the application type. Create the client ID by pressing the default button on the screen. The client ID will be created and presented to you (Figure 11.34).

The first items, the client ID and the client secret, are required to allow your local SSIS control flow to access your Google Drive account. Write down both pieces of information or keep your Web browser open. Also, make sure to keep this information secret. When entering the information in the control or data flow, the consent screen will be opened in the default Web browser. When that happens, log into your Google account and confirm the application.

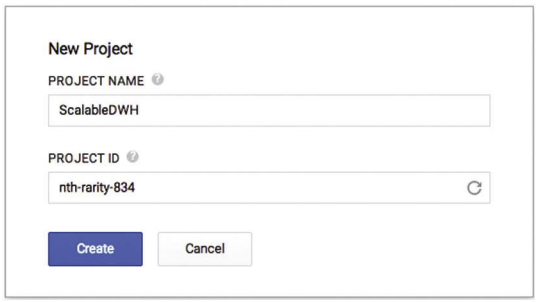

FIGURE 11.31

Create new project in Google Drive.

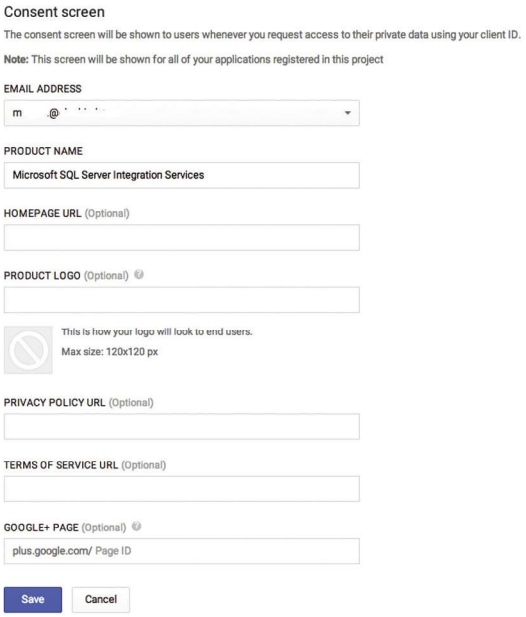

FIGURE 11.32

Setup Consent screen.

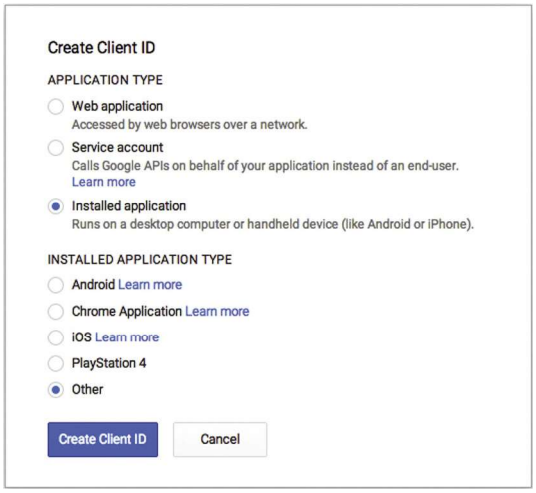

FIGURE 11.33

Create client ID for desktop application.

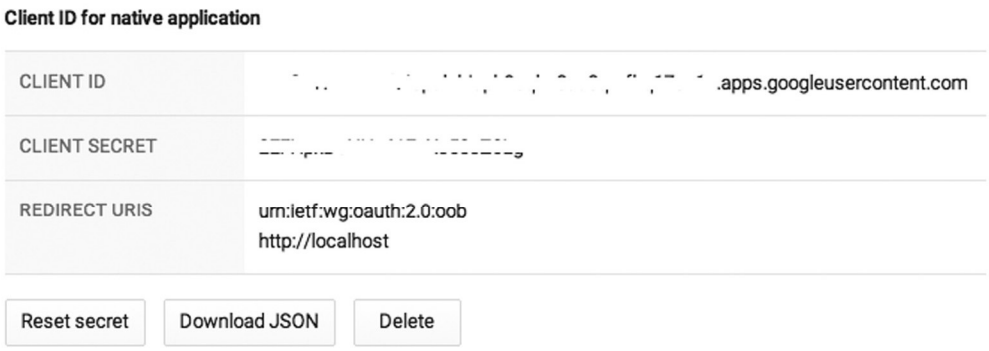

FIGURE 11.34

Client ID for native application.

Lastly, make sure to enable the **Drive API** and the **Drive SDK** under APIs.

11.8.2 CONTROL FLOW

The first step is to find out the Google Spreadsheets that are available on the Google Drive account. The basic idea is to use the ADO.NET provider for Google Apps to search for all spreadsheets with a name similar to "On_Time_On_Time" by performing a SELECT operation against the provider.

Drag an **Execute SQL Task** to the control flow, outside of the existing **Foreach Loop Container** (Figure 11.35).

Set the ConnectionType property to ADO.NET and create a new connection (Figure 11.36).

Set the properties as shown in Table 11.27 for the connection.

The first four properties are related to the authentication process. Note that there is also another authentication method available, based on user and password credentials. However, Google requires OAuth 2.0 for the operations performed by this SSIS control flow.

The pseudo columns setting activates all pseudo columns in all tables. The pseudo columns are required in order to perform a search on Google Drive. This is discussed next. Set timeout to 0 in order to avoid SSIS errors due to timeouts when selecting a large number of Google Spreadsheets.

Once the connection has been set up completely, select the **OK** button and finish the configuration of the Execute SQL Task: set the **ResultSet** property to **Full Result Set** and enter the following query in the **SQLStatement** property:

```
SELECT
       Title,
       CONCAT('https://spreadsheets.google.com/feeds/worksheets/',
              Id, '/private/full') AS FeedLink
FROM
       GDrive
WHERE
       Title LIKE '%TestCase%'
       AND Trashed=False
       AND MIMEType='application/vnd.google.apps.spreadsheet'
       AND Starred=true
```

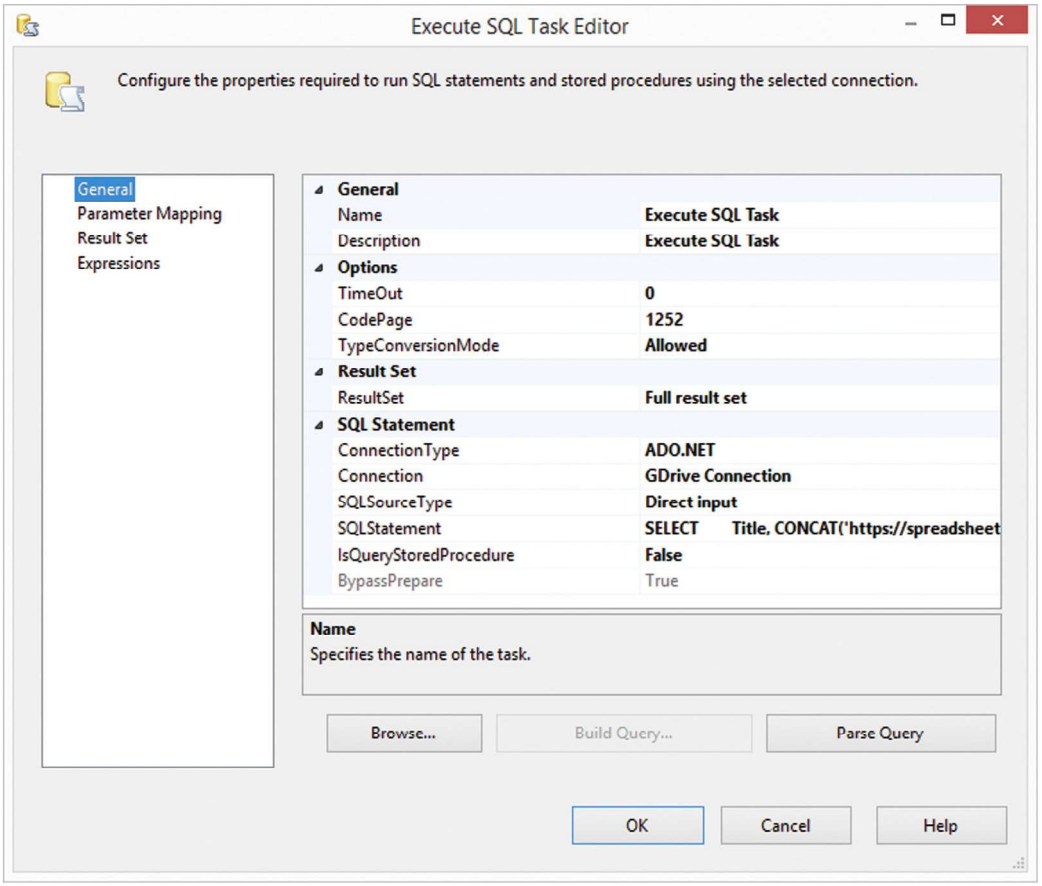

FIGURE 11.35

Execute SQL task editor for the Google Drive connection.

Table 11.27 Property Values for the Google Drive Connection

Property	Value
Initiate OAuth	True
OAuth Settings Location	A folder on the local file system accessible by SSIS
OAuth Client Id	Your client ID created in the previous section
OAuth Client Secret	Your client secret created in the previous section
Pseudo Columns	*=*
Readonly	True
Timeout	0

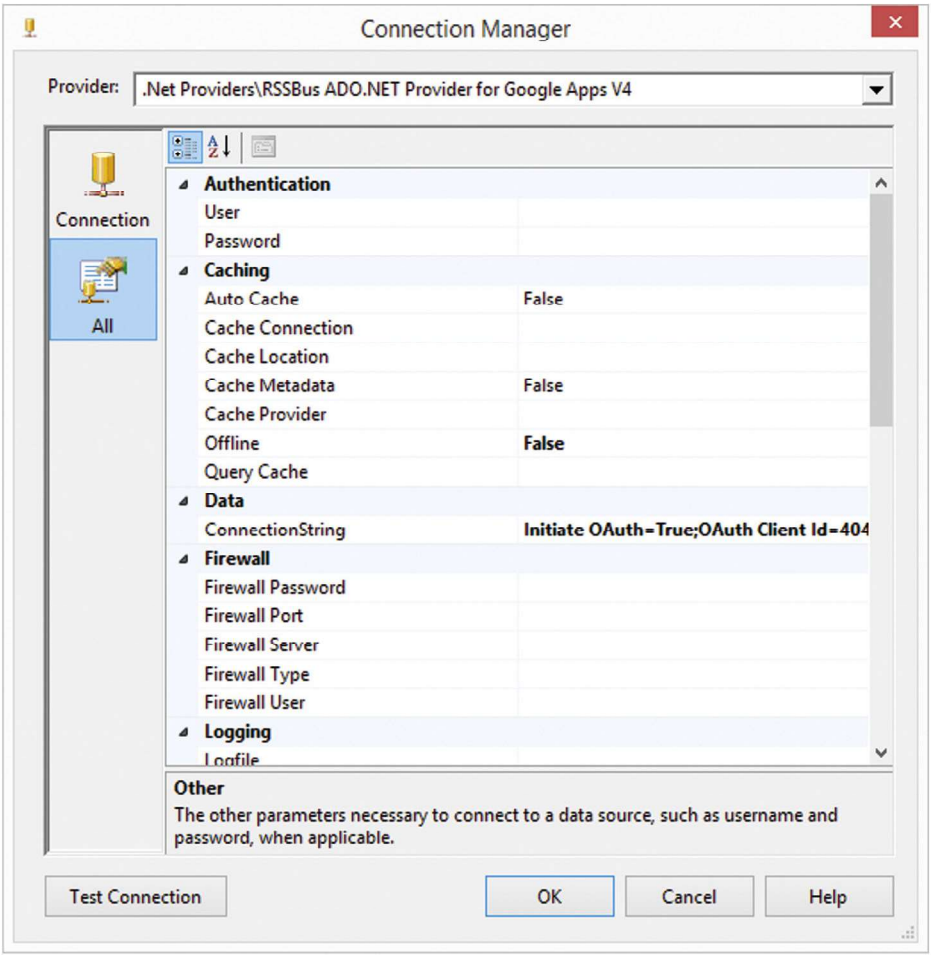

FIGURE 11.36

Connection manager for the Google Drive connection.

The statement selects the title and the spreadsheet ID of those files from Google Drive that meet the following conditions:

1. The title starts with "On_Time_On_Time": this makes sure that only files are selected that provide airline data.
2. The file is not trashed.
3. The file is a Google Spreadsheet.
4. The file is starred (optional).

The last condition reduces the number of spreadsheets returned by this query to a manageable number of documents: it only returns spreadsheets that have been starred. Remove this WHERE condition

to retrieve all spreadsheets for the airline data. If you leave this option in, make sure you have actually starred some of the airline spreadsheets.

The conditions are provided to Google Drive by a pseudo column called Query. This pseudo column doesn't exist in the source sheet but is used by the CData ADO.NET source for Google Apps to provide the query in a SQL friendly format. Additional search parameters can be found under the following URL: https://developers.google.com/drive/web/search-parameters.

Note that the Id returned by Google Drive is used to build the feed link of the spreadsheet. It follows the following format: https://spreadsheets.google.com/feeds/worksheets/[id]/private/full.

Both the title and the feed link are required: the title is used to extract the load date as described in section 11.7. The feed link is used to access the data from the spreadsheet in the data flow.

Because the ADO.NET provider returns one row per spreadsheet, the result set has to be stored in an object variable. This can be configured on the **Result Set** tab of the **Execute SQL Task Editor** (Figure 11.37).

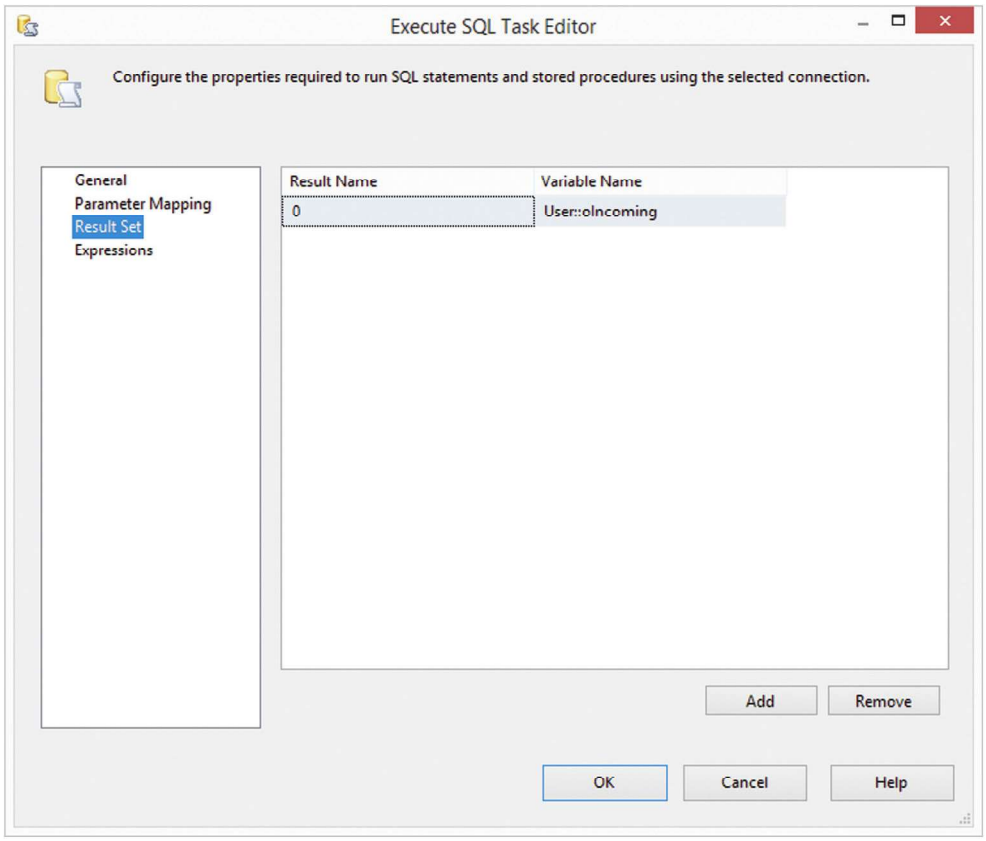

FIGURE 11.37

Configure the result set.

Create a new variable in the **User** namespace. Set the value type to **Object** because it will be used to store the whole dataset returned by the ADO.NET provider. Associate it to the result with name 0 as shown in Figure 11.37.

Close the editor and connect the Execute SQL Task to the Foreach Loop Container in the control flow (Figure 11.38).

The next step is to change the foreach loop container because it should no longer traverse over the local file system. Instead, it should traverse over the ADO.NET result set stored in the object variable. Open the editor of the container and switch to the **Collection** tab (Figure 11.39).

Change the enumerator to a **Foreach ADO Enumerator**. Set the **ADO object source variable** to the one that was configured in Figure 11.38. Make sure that the **enumeration mode** is set to **Rows in the first table**. Switch to the Variable Mappings tab in order to map from the columns in the result set to SSIS variables that can be used in the control and data flow (Figure 11.40).

The first column contains the title of the spreadsheet. This is the clear name that is also shown when viewing the Google Drive folder online. Make sure it is mapped to the sFileName variable created in section 11.6. The second column in the result set contains the feed link, which is required by the data flow to access the spreadsheet data. Create a new variable in the User namespace called sFeedLink with a value type of String and set the default value to a valid feed link, such as https://spreadsheets.google.com/feeds/worksheets/1gsbCxxTmfSZnoZQxCN1hoSZfnXFNgu-79JMwn35iVNc/private/full.

Setting this default value is required to configure the data flow properly. The feed link can be copied from the following Web site to avoid typing it in:

http://www.datavault.guru/2015/01/22/feedlink-for-sample-airline-data/.

You should also set the **sFileName** variable to the name of the sheet within the spreadsheet behind the feed link to avoid issues in the data flow later. Therefore, set the default value to "On_Time_On_Time_Performance_2006_2_27_DL".

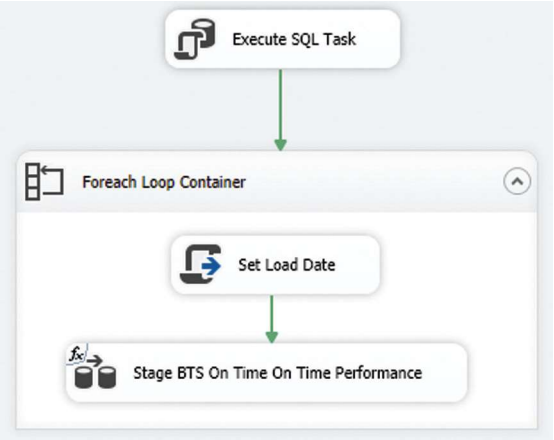

FIGURE 11.38

Control flow to source the sample airline data.

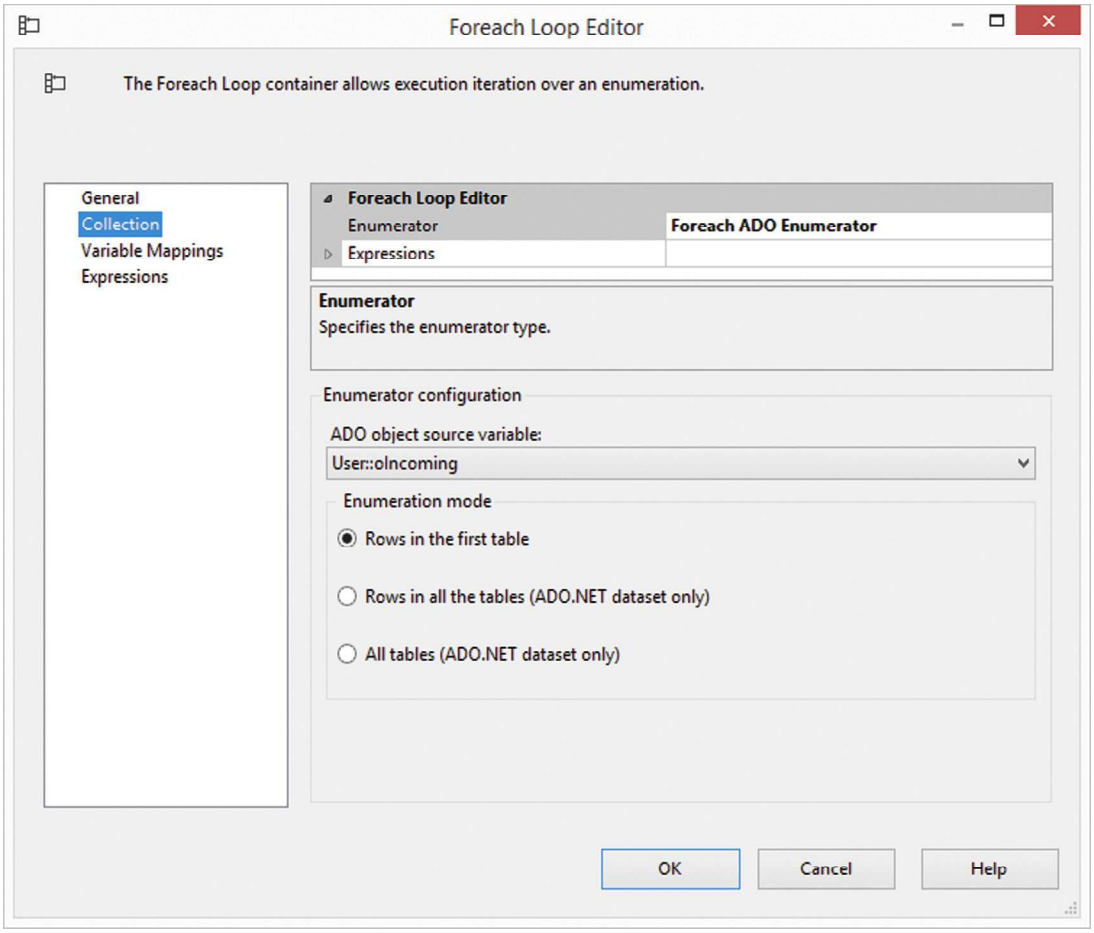

FIGURE 11.39

Modifying the enumerator of the foreach loop container.

Setting up the variables completes the setup of the control flow. The next step is to configure the data flow by changing the source.

11.8.3 GOOGLESHEETS CONNECTION MANAGER

After installation of both software packages, you will notice two new components in the SSIS Toolbox when switching to the Data Flow tab, as shown in Figure 11.41.

You can move the components into the **Other Source** or **Other Destination** folder by opening their context menu and selecting the respective menu item.

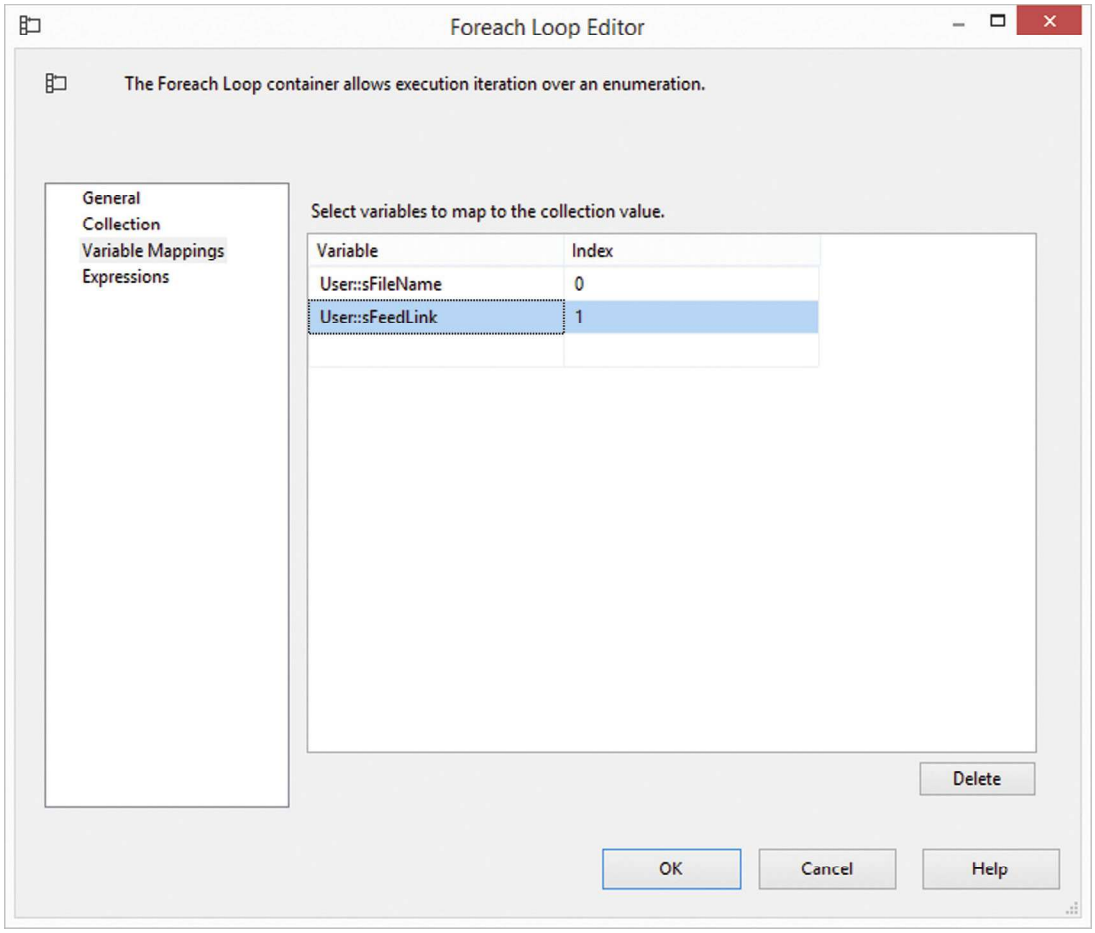

FIGURE 11.40

Variable mappings to map the columns in the result set to variables.

Drag a **CData GoogleSheets Source** to the data flow created in the previous section. It will replace the **Flat File Source** that was used before. Open the editor by double clicking the source and create a new connection using the **New…** button. The dialog shown in Figure 11.42 will appear.

Enter the information shown in Table 11.28 to set up the connection to a sample Google Sheets document.

The spreadsheet value should be set to the feed link that will be obtained in the control flow. It uniquely identifies a spreadsheet. For now, it is set as a constant value and replaced in an expression later. It is also possible to provide the name of the spreadsheet. However, because the API is limited

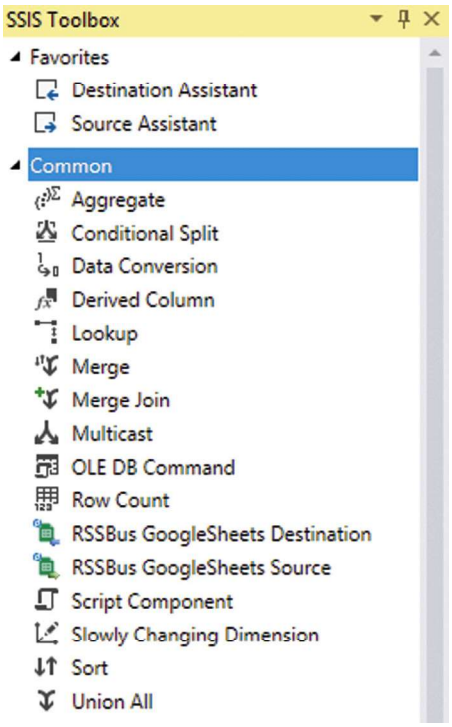

FIGURE 11.41

SSIS Toolbox with CData components.

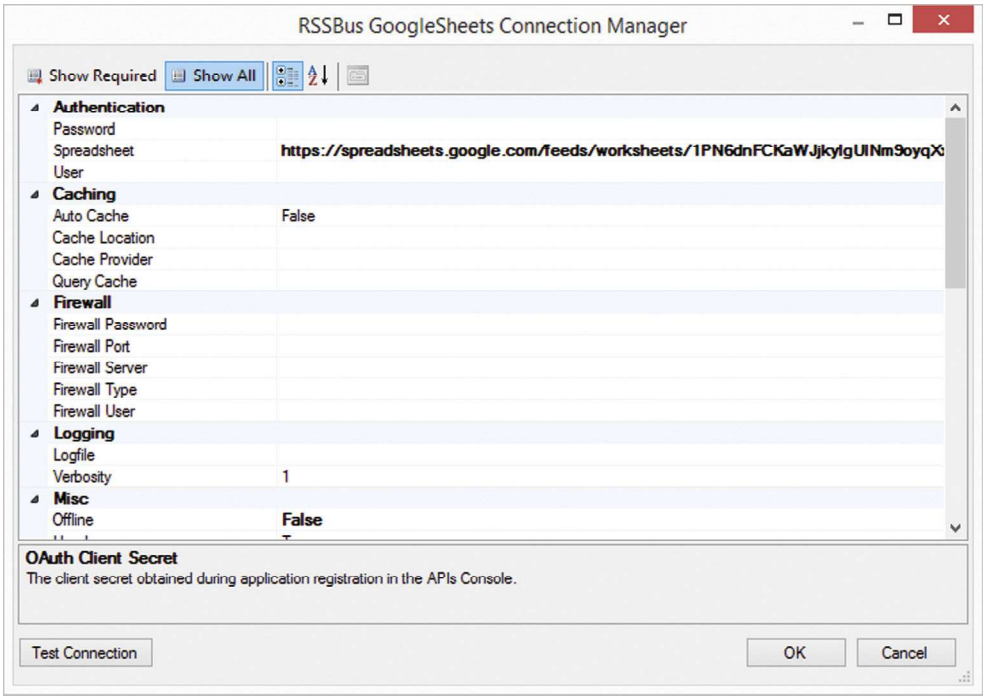

FIGURE 11.42

Connection manager to set up the Google Sheets connection.

Table 11.28 Required Property Values to Set Up the Google Sheets Connection

Property	Value
Initiate OAuth	True
OAuth Settings Location	A folder on the local file system accessible by SSIS. Make sure to use another folder than the one configured in the previous section.
OAuth Client Id	Your client ID created in section 11.8.1
OAuth Client Secret	Your client secret created in section 11.8.1
Spreadsheet	https://spreadsheets.google.com/feeds/worksheets/1gsbCxxTmfSZnoZQxCN1hoSZfnXFNgu-79JMwn35iVNc/private/full
Readonly	True
Timeout	0
Header	True
Detect Data Types	0

to a small number of spreadsheets that can be accessed by name (there is an API limit that allows only 500 spreadsheet names to return), providing the feed link is a more secure way. To avoid typing in the feed link, copy it from the following Web site:

http://www.datavault.guru/2015/01/22/feedlink-for-sample-airline-data/

The property **Detect Data Types** should be turned off by setting its value to 0. In most cases, this option is quite useful, because it automatically detects the data types of the columns in the spreadsheet by analyzing a number of records. The number of records it uses for this analysis can be provided here. The more records it analyzes, the more secure the data type detection becomes [34]. However, because not all the airline data uploaded to Google Drive has diversions (the columns towards the right), the data type detection might fail. In order to avoid any such problems, we have decided to completely turn off the feature and force the source component to set all data types to varchar(2000) instead. This will affect the staging table as well.

Close the dialog by selecting the **OK** button. The connection is taken over as the active connection manager in the previous dialog. Because each spreadsheet has multiple tabs, select the corresponding tab from the **Table or View** box (Figure 11.43).

Select the table On_Time_On_Time_Performance_2006_2_27_DL from the list. The **Spreadsheets** view is a virtual view that returns a list of available spreadsheets (but it is limited to the 500 records). You can preview the data in the spreadsheet using the **Preview** button (Figure 11.44).

Also, make sure that switching to the **columns** tab in the editor retrieves the columns in the source. Close the source editor.

11.8.4 DATA FLOW

The next step is to replace the flat file connection in the data flow by the Google Sheets Source (Figures 11.45 and 11.46).

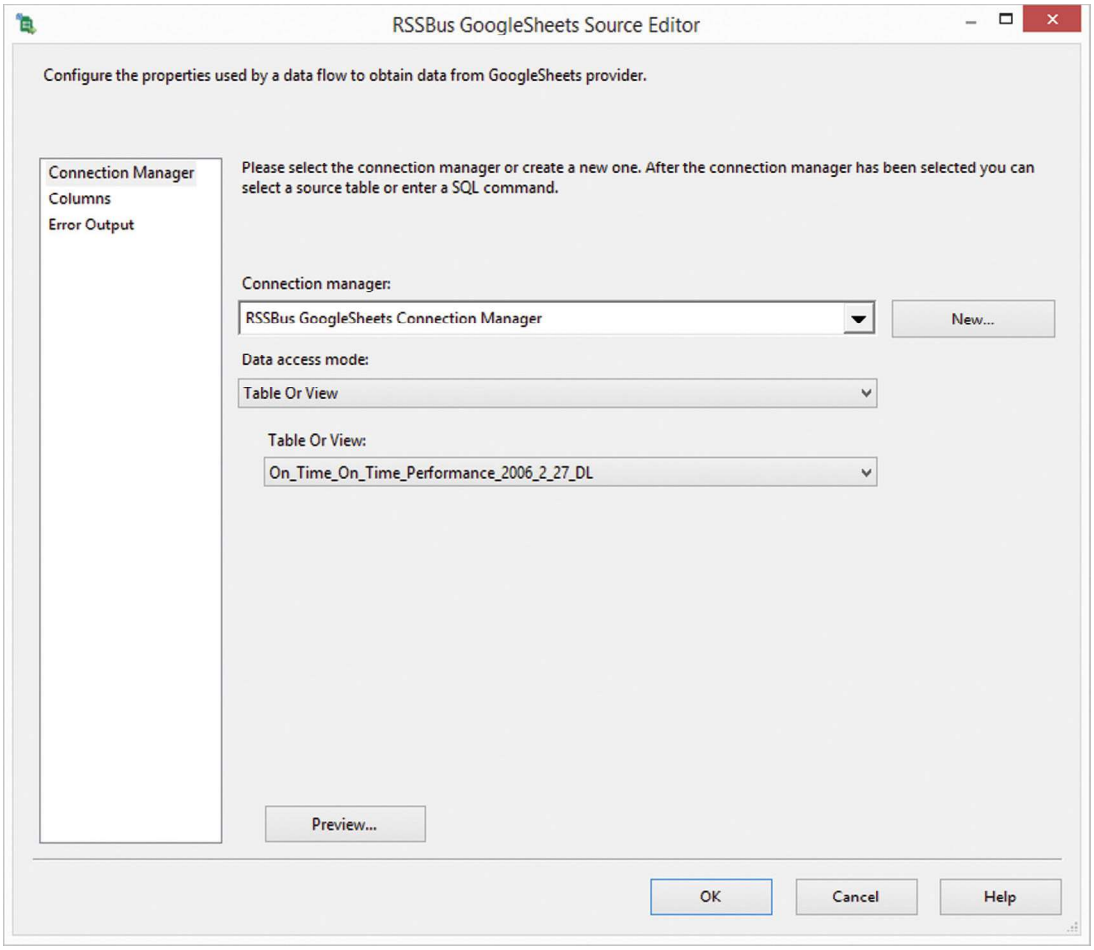

FIGURE 11.43

Setting up the connection in the CData GoogleSheets source editor.

Delete the old data flow path and connect the Google spreadsheet source to next component in the data flow (**Add Sequence Number**). After both components are connected, error messages are shown on two of the components.

The first error is on the script component that calculates the hash values. This is because the input column definition has changed and requires a rebuild of the script. Open the editor, open the script and rebuild it. After closing the script editor and the dialog, the error should be gone.

The second issue is a warning that indicates a column truncation. This warning appears because the destination table in the staging area is modeled after actual data types from the source file. Because the

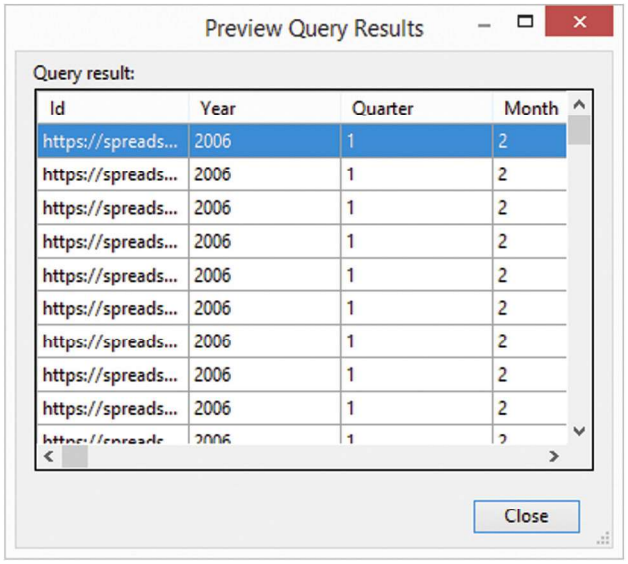

FIGURE 11.44

Previewing the data from the spreadsheet.

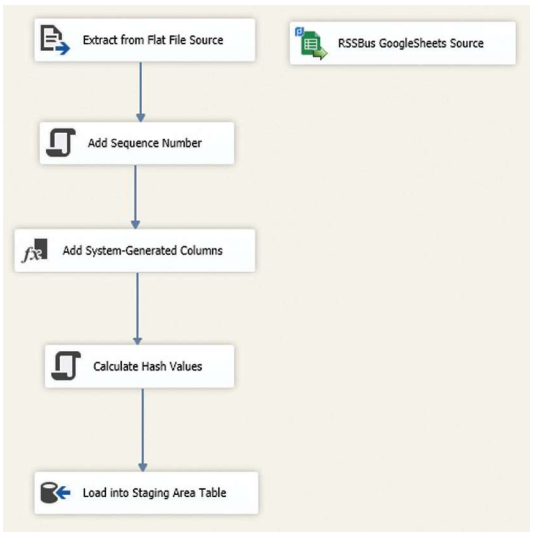

FIGURE 11.45

Data flow utilizing the flat file source.

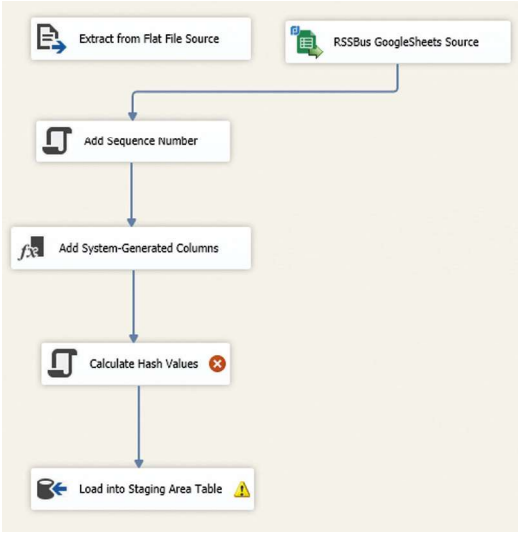

FIGURE 11.46

Data flow utilizing the Google spreadsheet.

Google Sheets Source only provides strings, another staging table is required. Use the following T-SQL script to create a new staging table*:

```
CREATE TABLE [bts].[OnTimeOnTimePerformanceGD](
       [Id] [nvarchar](255) NULL,
       [Year] [nvarchar](2000) NULL,
       [Quarter] [nvarchar](2000) NULL,
       [Month] [nvarchar](2000) NULL,
       [Dayofmonth] [nvarchar](2000) NULL,
       [Dayofweek] [nvarchar](2000) NULL,
       [Flightdate] [nvarchar](2000) NULL,
       [Uniquecarrier] [nvarchar](2000) NULL,
       [Airlineid] [nvarchar](2000) NULL,
       [Carrier] [nvarchar](2000) NULL,
       [Tailnum] [nvarchar](2000) NULL,
       [Flightnum] [nvarchar](2000) NULL,
       [Originairportid] [nvarchar](2000) NULL,
       [Originairportseqid] [nvarchar](2000) NULL,
       [Origincitymarketid] [nvarchar](2000) NULL,
       [Origin] [nvarchar](2000) NULL,
       [Origincityname] [nvarchar](2000) NULL,
       [Originstate] [nvarchar](2000) NULL,
       [Originstatefips] [nvarchar](2000) NULL,
       [Originstatename] [nvarchar](2000) NULL,
       [Originwac] [nvarchar](2000) NULL,
       [Destairportid] [nvarchar](2000) NULL,
       [Destairportseqid] [nvarchar](2000) NULL,
       [Destcitymarketid] [nvarchar](2000) NULL,
       [Dest] [nvarchar](2000) NULL,
       [Destcityname] [nvarchar](2000) NULL,
       [Deststate] [nvarchar](2000) NULL,
       [Deststatefips] [nvarchar](2000) NULL,
       [Deststatename] [nvarchar](2000) NULL,
       [Destwac] [nvarchar](2000) NULL,
       [Crsdeptime] [nvarchar](2000) NULL,
       [Deptime] [nvarchar](2000) NULL,
       [Depdelay] [nvarchar](2000) NULL,
       [Depdelayminutes] [nvarchar](2000) NULL,
       [Depdel15] [nvarchar](2000) NULL,
       [Departuredelaygroups] [nvarchar](2000) NULL,
       [Deptimeblk] [nvarchar](2000) NULL,
       [Taxiout] [nvarchar](2000) NULL,
       [Wheelsoff] [nvarchar](2000) NULL,
       [Wheelson] [nvarchar](2000) NULL,
       [Taxiin] [nvarchar](2000) NULL,
       [Crsarrtime] [nvarchar](2000) NULL,
       [Arrtime] [nvarchar](2000) NULL,
       [Arrdelay] [nvarchar](2000) NULL,
       [Arrdelayminutes] [nvarchar](2000) NULL,
       [Arrdel15] [nvarchar](2000) NULL,
       [Arrivaldelaygroups] [nvarchar](2000) NULL,
       [Arrtimeblk] [nvarchar](2000) NULL,
       [Cancelled] [nvarchar](2000) NULL,
       [Cancellationcode] [nvarchar](2000) NULL,
       [Diverted] [nvarchar](2000) NULL,
       [Crselapsedtime] [nvarchar](2000) NULL,
       [Actualelapsedtime] [nvarchar](2000) NULL,
```

*The file name of the source code file is provided on the companion site, please refer the site for more details: http://booksite.elsevier.com/9780128025109

```
[Airtime] [nvarchar](2000) NULL,
[Flights] [nvarchar](2000) NULL,
[Distance] [nvarchar](2000) NULL,
[Distancegroup] [nvarchar](2000) NULL,
[Carrierdelay] [nvarchar](2000) NULL,
[Weatherdelay] [nvarchar](2000) NULL,
[Nasdelay] [nvarchar](2000) NULL,
[Securitydelay] [nvarchar](2000) NULL,
[Lateaircraftdelay] [nvarchar](2000) NULL,
[Firstdeptime] [nvarchar](2000) NULL,
[Totaladdgtime] [nvarchar](2000) NULL,
[Longestaddgtime] [nvarchar](2000) NULL,
[Divairportlandings] [nvarchar](2000) NULL,
[Divreacheddest] [nvarchar](2000) NULL,
[Divactualelapsedtime] [nvarchar](2000) NULL,
[Divarrdelay] [nvarchar](2000) NULL,
[Divdistance] [nvarchar](2000) NULL,
[Div1airport] [nvarchar](2000) NULL,
[Div1airportid] [nvarchar](2000) NULL,
[Div1airportseqid] [nvarchar](2000) NULL,
[Div1wheelson] [nvarchar](2000) NULL,
[Div1totalgtime] [nvarchar](2000) NULL,
[Div1longestgtime] [nvarchar](2000) NULL,
[Div1wheelsoff] [nvarchar](2000) NULL,
[Div1tailnum] [nvarchar](2000) NULL,
[Div2airport] [nvarchar](2000) NULL,
[Div2airportid] [nvarchar](2000) NULL,
[Div2airportseqid] [nvarchar](2000) NULL,
[Div2wheelson] [nvarchar](2000) NULL,
[Div2totalgtime] [nvarchar](2000) NULL,
[Div2longestgtime] [nvarchar](2000) NULL,
[Div2wheelsoff] [nvarchar](2000) NULL,
[Div2tailnum] [nvarchar](2000) NULL,
[Div3airport] [nvarchar](2000) NULL,
[Div3airportid] [nvarchar](2000) NULL,
[Div3airportseqid] [nvarchar](2000) NULL,
[Div3wheelson] [nvarchar](2000) NULL,
[Div3totalgtime] [nvarchar](2000) NULL,
[Div3longestgtime] [nvarchar](2000) NULL,
[Div3wheelsoff] [nvarchar](2000) NULL,
[Div3tailnum] [nvarchar](2000) NULL,
[Div4airport] [nvarchar](2000) NULL,
[Div4airportid] [nvarchar](2000) NULL,
[Div4airportseqid] [nvarchar](2000) NULL,
[Div4wheelson] [nvarchar](2000) NULL,
[Div4totalgtime] [nvarchar](2000) NULL,
[Div4longestgtime] [nvarchar](2000) NULL,
[Div4wheelsoff] [nvarchar](2000) NULL,
[Div4tailnum] [nvarchar](2000) NULL,
[Div5airport] [nvarchar](2000) NULL,
[Div5airportid] [nvarchar](2000) NULL,
[Div5airportseqid] [nvarchar](2000) NULL,
[Div5wheelson] [nvarchar](2000) NULL,
[Div5totalgtime] [nvarchar](2000) NULL,
[Div5longestgtime] [nvarchar](2000) NULL,
[Div5wheelsoff] [nvarchar](2000) NULL,
[Div5tailnum] [nvarchar](2000) NULL,
[Sequence] [int] NOT NULL,
[LoadDate] [datetime] NOT NULL,
[RecordSource] [nvarchar](27) NOT NULL,
[FlightNumHashKey] [char](32) NOT NULL,
[OriginHashKey] [char](32) NOT NULL,
```

```
	[CarrierHashKey] [char](32) NOT NULL,
	[TailNumHashKey] [char](32) NOT NULL,
	[DestHashKey] [char](32) NOT NULL,
	[Div1AirportHashKey] [char](32) NOT NULL,
	[Div2AirportHashKey] [char](32) NOT NULL,
	[Div3AirportHashKey] [char](32) NOT NULL,
	[Div4AirportHashKey] [char](32) NOT NULL,
	[Div5AirportHashKey] [char](32) NOT NULL,
	[FlightHashKey] [char](32) NOT NULL,
	[Div1FlightHashKey] [char](32) NOT NULL,
	[Div2FlightHashKey] [char](32) NOT NULL,
	[Div3FlightHashKey] [char](32) NOT NULL,
	[Div4FlightHashKey] [char](32) NOT NULL,
	[Div5FlightHashKey] [char](32) NOT NULL,
	[FlightNumCarrierHashKey] [char](32) NOT NULL,
	[OriginAirportHashDiff] [char](32) NOT NULL,
	[DestAirportHashDiff] [char](32) NOT NULL,
 CONSTRAINT [PK_OnTimeOnTimePerformanceGD] PRIMARY KEY NONCLUSTERED
(
	[Sequence] ASC,
	[LoadDate] ASC
) ON [INDEX]
) ON [DATA]
```

Notice the nvarchar(2000) data types used for most columns (except the system-generated columns and the Id column that identifies each row in the Google Spreadsheets document). The alternative to convert all columns to the data types used in section 11.6.2 is too complicated in the staging process. Also, it would hinder automation efforts. Remember that another purpose for the staging area is to allow easy retrieval of the data from the source system. The data will be converted into the actual data types when it is loaded into the next layer, which is the Enterprise Data Warehouse (EDW) layer, modeled as a Data Vault 2.0 model.

Once the table has been created, it can be set as the target in the **OLE DB Destination Editor** (Figure 11.47).

Select the new staging table **[bts].[OnTimeOnTimePerformanceGD]** from the list of tables in the staging area. All other options should remain the same. However, make sure that the columns from the data flow are still mapped to the target columns because the name of the columns has changed. It follows the convention of the Google API to change the column name to lower-case, except the first letter. For example, the column name **DestAirportID** becomes **Destairportid**. Therefore, a remap of the columns is required. Change to the **Mappings** tab and select the **Map Items by Matching Names** menu item (Figure 11.48).

Make sure that all columns have been mapped. After selecting the OK button, start the process to load some data into the target table in the staging area. Generate and compare some of the hash values as described in section 11.6.3 to validate the process. If the hash values (either the hash keys or the hash diffs) are different, a problem has occurred in the SSIS process.

Finally, the configuration of the data flow needs to be adjusted in order to make use of the variables from the control flow. In the control flow, select the **CData GoogleSheets Connection Manager** and open the property expressions editor from its properties (Figure 11.49).

Set the expression of the **Spreadsheet** property to @[User::sFeedLink].

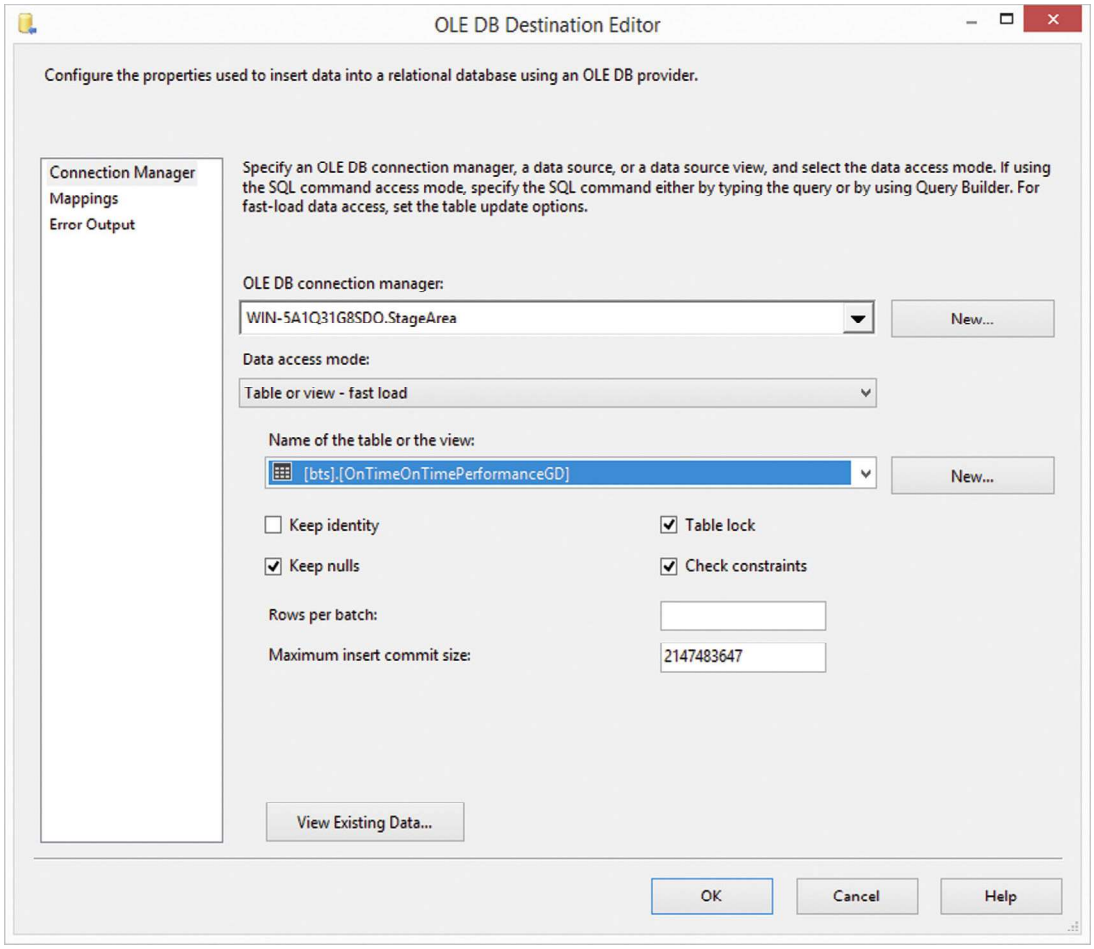

FIGURE 11.47

Configure OLE DB destination editor to use the new staging table.

Close the expression editor and set **DelayValidation** to **True**.

Select the canvas of the data flow and open the expression editor in its properties (Figure 11.50).

Select the property **[CData GoogleSheets Source].[TableOrView]** to dynamically set the name of the table (which is the sheet name within Google Sheets). Set it to the variable sFileName by adding the expression **@[User::sFileName]**. Confirm the dialog.

Finally, set the property **ValidateExternalMetadata** of the **CData GoogleSheets Source** to **False** in order to prevent validation of the variable at design-time.

To make this SSIS control flow less resource consuming (especially storage in the staging area), you could introduce command line parameters to the SSIS package and stage individual files, load them into the Raw Data Vault and truncate the staging tables before loading the next Google Sheet from the cloud.

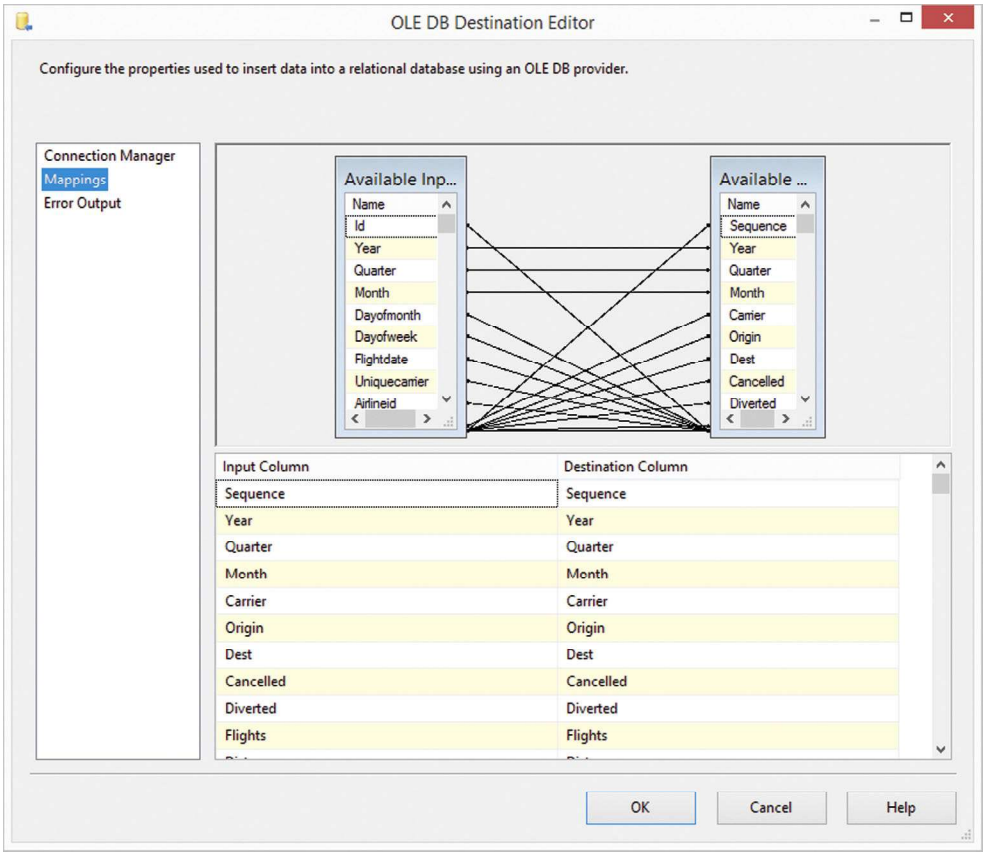

FIGURE 11.48

Remapping the columns after changing the data source.

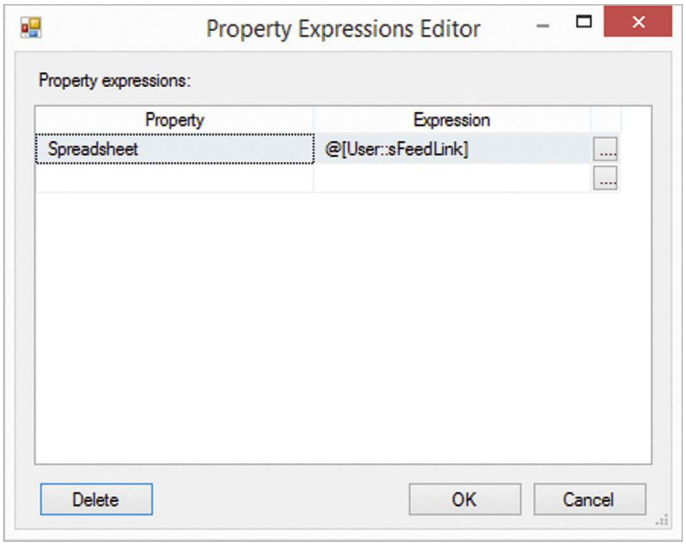

FIGURE 11.49

Property expression editor for GoogleSheets connection manager.

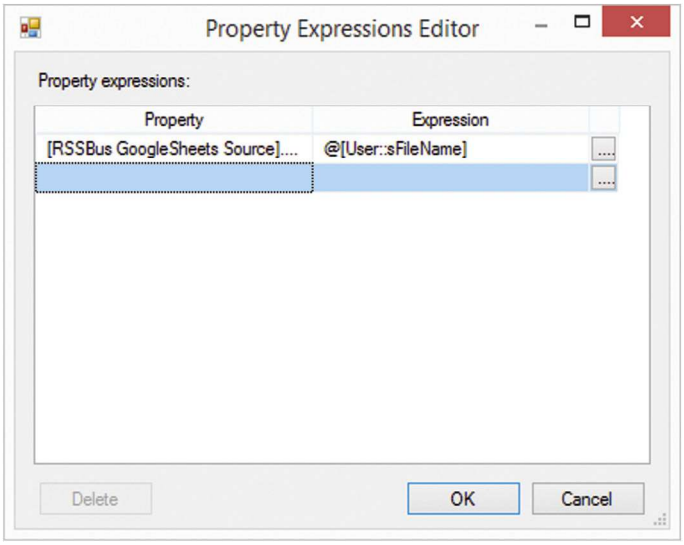

FIGURE 11.50

Property expression editor for data flow task.

Using this approach, only one batch is staged during this initial load at a given time. However, it requires that the Raw Data Vault be loaded just after staging an individual file. This is covered in the next chapter.

11.9 SOURCING DENORMALIZED DATA SOURCES

In some cases, source files are provided in a hierarchical or denormalized format. Examples for such files include XML files and COBOL copybooks. The flat file that was sourced in section 11.6 was also a denormalized flat file because it contained multiple diversions in the same row as the actual flight. This data was joined into the parent table.

The following code presents such a hierarchical XML file:

```
<?xml version="1.0" encoding="utf-8"?>
<flight id="DL1234">
   <passenger id="US4211">
      <title>Mrs.</title>
      <name>
        <firstname>Amy</firstname>
        <lastname>Miller</lastname>
      </name>
      <preferred-dish>Vegetarian</preferred-dish>
      <address-line-1>2050 1st Street</address-line-1>
      <address-line-2/>
      <city>San Francisco</city>
      <zip>94114</zip>
      <state>CA</state>
```

```
      <country>USA</country>
   </passenger>
  <passenger id="DE4949">
      <title>Mr.</title>
      <name>
        <firstname>Peter</firstname>
        <lastname>Heinz</lastname>
      </name>
      <preferred-dish>Meat</preferred-dish>
      <address-line-1>Auf dem Hofe 9</address-line-1>
      <address-line-2/>
      <city>Stadt Oldendorf</city>
      <zip>98472</zip>
      <state/>
      <country>Germany</country>
   </passenger>
</flight>
```

The XML file above contains two records in the following conceptual hierarchy:

- `flight` (id = DL1234)
  - `passenger` (id = US4211)
    - `name` (Amy Miller)
  - `passenger` (id = DE4949)
    - `name` (Heinz Peter)

XML files are not limited to three levels. The number of levels is unlimited. It is also possible to create an unbalanced XML file, which uses a different number of levels in each sub-tree.

In order to successfully load these formats into a relational staging area, they need to be normalized first. Otherwise, dealing with hierarchical tables becomes too complex when loading the data into the Enterprise Data Warehouse. Also, normalizing the data prepares the data for the Data Vault model, which requires that the data be normalized even further (Figure 11.51).

The normalization supports the divide-and-conquer approach of the Data Vault 2.0 System of Business Intelligence: instead of tackling the whole source file in its hierarchical format at once, in a complex staging process, the individual levels are sourced using the same approach as ordinary files. The only difference is that the source file is accessed multiple times and loaded into multiple target tables in the staging area.

To capture the data from the XML file in the beginning of this section, the following E/R model can be used (Figure 11.52).

There are three targets required: a flights table that covers flight information, a passenger table that covers the second level and a name table. Not only has the parent sequence been added, but also the respective business keys (**PassengerID** or **FlightID**) and their hash keys. The latter are required when loading the Data Vault model; the business keys are optional merely for debugging purposes. The **Passenger** staging table also includes an optional hash diff attribute on the descriptive attributes.

Note that only three entities are required because each level follows the same structure. If the structure of each sub-tree were different, as it is in XHTML (a XML derivative for presenting rich formatted information from Web servers using Web browsers, similar to HTML), this approach might not be

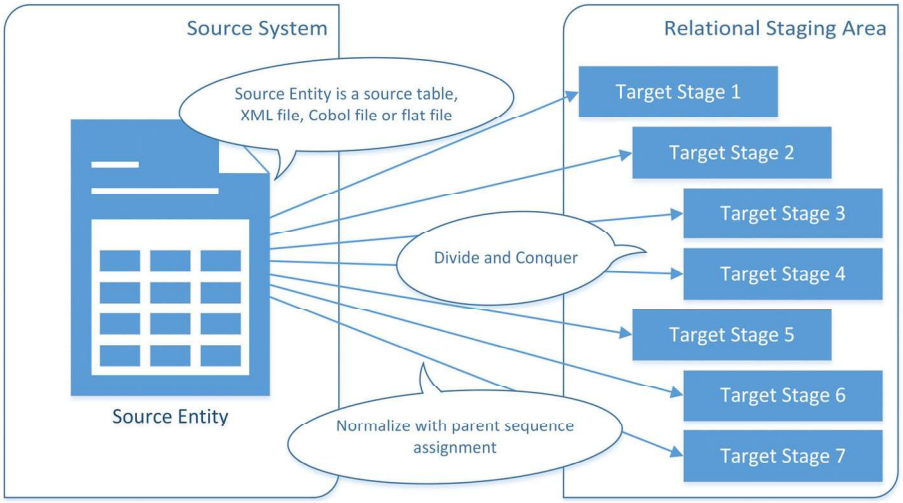

FIGURE 11.51

Normalizing source system files.

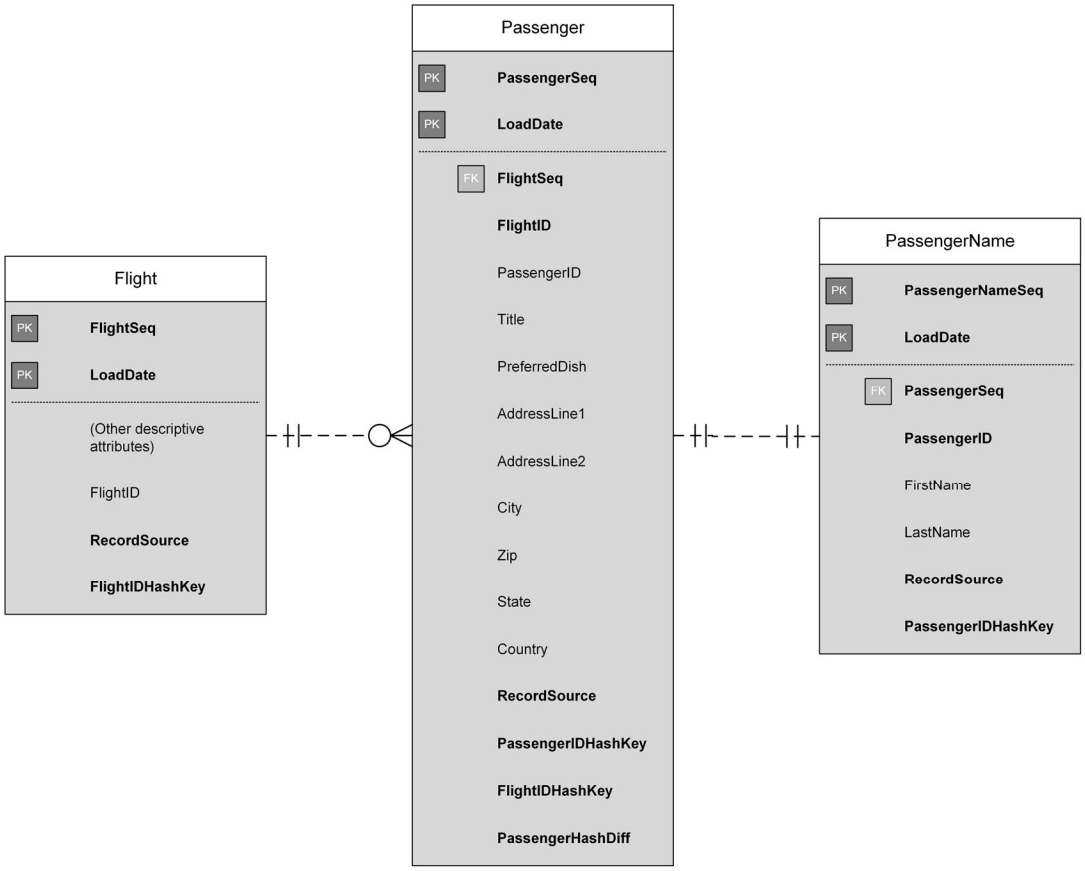

FIGURE 11.52

Normalized staging tables (physical design).

sufficient. If the input file doesn't use a common structure for all sub-trees, consider an approach using unstructured databases for staging, for example Hadoop or another NoSQL database.

Because the number of diversions in the flat file used in section 11.6 is limited to five and all columns from the child table (the diversions) are pivoted into the parent table, the normalization process is optional. If the number of diversions were unlimited and the structure in a hierarchical format, the normalization would be strongly suggested.

11.10 SOURCING MASTER DATA FROM MDS

Compared to data from flat files, relational data, such as master data from Microsoft Master Data Services (MDS), can be sourced much easier. The reason is that there is typically no need to source historical master data because the source only provides the current master data. In general, there is no difference from sourcing master data stored in Microsoft MDS to sourcing any other relational database for any kind of data.

However, if MDS houses analytical master data (refer to Chapter 9, Master Data Management), it is often located on the same infrastructure as the data warehouse. In this case, there is no need to actually load the data into the staging area because the primary reason of the staging area is to reduce the workload on the operational systems when loading the data warehouse. But if MDS is used to maintain only analytical master data, the additional workload on MDS is often negligible. Especially if the data warehouse team controls MDS, it is possible to make sure that data warehouse loads will not affect the business user.

For that reason, it is often possible to stage the master data using virtual views instead of materializing the data using ETL. But it is still required to attach the system-generated fields, including the sequence number, load date, record source and hashes. To virtually stage a master data entity, the following statement creates a stage area view in the **mds** schema on the **BTS_Region_DWH** subscription view of MDS (refer to Chapter 9 for instructions on creating the view in MDS):

```
CREATE VIEW [mds].[BTS_Region_DWH] AS
SELECT [ID]
      ,[MUID]
      ,[VersionName]
      ,[VersionNumber]
      ,[VersionFlag]
      ,[Name]
      ,[Code]
      ,[ChangeTrackingMask]
      ,[Abbreviation]
      ,[Sort Order]
      ,[External Reference]
      ,[Record Source]
      ,[Record Owner_Code]
      ,[Record Owner_Name]
      ,[Record Owner_ID]
      ,[Comments]
      ,[EnterDateTime]
      ,[EnterUserName]
      ,[EnterVersionNumber]
```

```
      ,[LastChgDateTime]
      ,[LastChgUserName]
      ,[LastChgVersionNumber]
      ,[ValidationStatus]

         -- Sequence ID
         ,ROW_NUMBER() OVER(ORDER BY ([Sort Order])) AS [Sequence]

         -- Load Date
         ,GETDATE() AS [LoadDate]

         -- Record Source
         ,'MDS.BTS.Region' AS [RecordSource]

         -- Hash key on region code
         ,UPPER(CONVERT(char(32),HASHBYTES('MD5',
              UPPER(RTRIM(LTRIM(COALESCE([Code], ''))))
         ),2)) AS RegionHashKey

         -- Hash key on record owner
         ,UPPER(CONVERT(char(32),HASHBYTES('MD5',
              UPPER(RTRIM(LTRIM(COALESCE([Record Owner_Name], ''))))
         ),2)) AS RecordOwnerHashKey

         -- Hash key on user name who created the record (optional)
         --,UPPER(CONVERT(char(32),HASHBYTES('MD5',
         --UPPER(RTRIM(LTRIM(COALESCE([EnterUserName], ''))))
         --),2)) AS EnterUserNameHashKey

         -- Hash key on user name who changed it the last time (optional)
         --,UPPER(CONVERT(char(32),HASHBYTES('MD5',
         --UPPER(RTRIM(LTRIM(COALESCE([LastChgUserName], ''))))
         --),2)) AS LastChgUserNameHashKey

         -- Hash diff on descriptive attributes
         ,UPPER(CONVERT(char(32),HASHBYTES('MD5',
              CONCAT(
                     UPPER(RTRIM(LTRIM(COALESCE([Code], '')))), ';',
                     RTRIM(LTRIM(COALESCE([ID], ''))), ';',
                     RTRIM(LTRIM(COALESCE([MUID], ''))), ';',
                     RTRIM(LTRIM(COALESCE([VersionName], ''))), ';',
                     RTRIM(LTRIM(COALESCE([VersionNumber], ''))), ';',
                     RTRIM(LTRIM(COALESCE([VersionFlag], ''))), ';',
                     RTRIM(LTRIM(COALESCE([Name], ''))), ';',
                     RTRIM(LTRIM(COALESCE([ChangeTrackingMask], ''))), ';',
                     RTRIM(LTRIM(COALESCE([Abbreviation], ''))), ';',
                     RTRIM(LTRIM(COALESCE([Sort Order], ''))), ';',
                     RTRIM(LTRIM(COALESCE([External Reference], ''))), ';',
                     RTRIM(LTRIM(COALESCE([Record Source], ''))), ';',
                     RTRIM(LTRIM(COALESCE([Comments], ''))), ';',
                     RTRIM(LTRIM(COALESCE([EnterDateTime], ''))), ';',
                     RTRIM(LTRIM(COALESCE([EnterUserName], ''))), ';',
                     RTRIM(LTRIM(COALESCE([EnterVersionNumber], ''))), ';',
                     RTRIM(LTRIM(COALESCE([LastChgDateTime], ''))), ';',
                     RTRIM(LTRIM(COALESCE([LastChgUserName], ''))), ';',
                     RTRIM(LTRIM(COALESCE([LastChgVersionNumber], ''))), ';',
                     RTRIM(LTRIM(COALESCE([ValidationStatus], '')))
              )
         ),2)) AS RegionHashDiff

   FROM [MDS].[mdm].[BTS_Region_DWH]
```

The view first selects all columns from the source system. It then generates the sequence number, load date and record source. The sequence number is generated using the **ROW_NUMBER** function and is just a sequence on the **sort order**. Avoid using the sort order or any other numerical value (such as the ID) because they don't have to be unique. However, uniqueness is a requirement for the sequence number. The ID might be unique, but using it means losing control over the generation process.

The hash keys are generated in a similar manner to that discussed before. Note that MDS provides two attributes that identify the member in the region entity: code and name. In many cases, the code column is an appropriate business key that should be hashed. But in other cases, it is just a sequence number without meaning to the business. In this case, the business key might be located in the name. Another business key in the entity identifies the record owner. This column is part of a domain attribute in MDS. Domain attributes are always represented in subscription views by three columns: the code, the name, and the ID of the referenced entity. In the case of the record owner, the name column provides the business key because the code column is a sequence number without meaning to the business. Therefore, the name is used as the input to the hash function.

If the data warehouse contains detailed user information, for example from the Active Directory of the organization, users might become business objects that should be modeled in the EDW. In this case, the two columns **EnterUserName** and **LastChgUserName** become business keys that need to be hashed. For the purpose of this chapter (and the remainder of this book), these attributes are considered as descriptive data.

The descriptive data is hashed to obtain the hash diff in the last step. The input to the hash function contains the case-insensitive business key of the proposed parent and the case-sensitive descriptive attributes. The columns, which are part of the referenced record owner entity, are not included in the descriptive data because they will not be included in the target satellite.

The query only implements a simplified hash diff calculation. In order to support the improved strategy to support changing satellite structures (additional columns at the end of the satellite table, as described in section 11.2.5), any trailing delimiters have to be removed from the concatenated hash diff input.

Note that, in many cases, the hash key is actually not required for master data sources; many master data tables are modeled as reference tables in the EDW. We have included the hash diff primarily for demonstration purposes.

Creating virtual views for master data provides a valid approach to quickly sourcing master data into the data warehouse. Using a virtual approach enables the data warehouse team to provide the master data within only one iteration in the Data Vault 2.0 methodology. If MDS or a similar master data management solution is located on the same hardware infrastructure as the data warehouse, the business users should not recognize any performance issues. However, it is also possible to source the master data using ETL processes, for example in SSIS. It follows a similar approach to that described in section 11.6 but with an OLE DB source instead of a flat file source.

REFERENCES

[1] http://www.rita.dot.gov/bts/.
[2] http://www.dot.gov/.
[3] http://www.transtats.bts.gov/DatabaseInfo.asp?DB_ID=120.

[4] http://www.swiss.com/corporate/en/company/about-us/facts-and-figures.
[5] http://msdn.microsoft.com/en-us/library/ms177456.aspx.
[6] Daniel Linstedt: DV2.0 and Hash Keys: Hash Keys and Architecture Changes, pp. 1, 2, 9.
[7] http://www.b-eye-network.com/blogs/linstedt/archives/2014/08/data_vault_20_b.php.
[8] Michael Coles, Rodney Landrum: Expert SQL Server 2008 Encryption, p. 151.
[9] Network Working Group: The MD5 Message-Digest Algorithm, http://tools.ietf.org/pdf/rfc1321.pdf.
[10] Information Technology Laboratory, National Institute of Standards and Technology: Secure Hash Standard (SHS), http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.
[11] RSA Laboratories: "PKCS #1: RSA Cryptography Standard," http://www.emc.com/emc-plus/rsa-labs/pkcs/files/h11300-wp-pkcs-1v2-2-rsa-cryptography-standard.pdf.
[12] http://support.microsoft.com/kb/889768.
[13] http://www.apprendre-en-ligne.net/crypto/bibliotheque/feistel/index.html.
[14] Microsoft: SQL Server to SQL Server PDW Migration Guide (AU2), p. 14.
[15] http://csrc.nist.gov/groups/STM/cavp/documents/shs/shaval.htm.
[16] http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-006.pdf.
[17] http://www.iso.org/iso/catalogue_detail?csnumber=40874.
[18] http://blogs.msdn.com/b/jeremykuhne/archive/2005/07/21/441247.aspx.
[19] http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html.
[20] https://hbase.apache.org/apidocs/org/apache/hadoop/hbase/util/ByteBufferUtils.html.
[21] Alastair Aitchison: "Beginning Spatial with SQL Server 2008," p. 93.
[22] Intel: Endianness White Paper, http://www.pascal-man.com/navigation/faq-java-browser/jython/endian.pdf
[23] Marc Stevens: Single-block collision attack on MD5, http://marc-stevens.nl/research/md5-1block-collision/md5-1block-collision.pdf.
[24] http://preshing.com/20110504/hash-collision-probabilities/.
[25] http://technet.microsoft.com/en-us/library/ms190969%28v=sql.105%29.aspx.
[26] Frederick P. Brooks Jr.: "The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)", p. 179.
[27] http://ssismhash.codeplex.com/.
[28] Ralph Kimball, Joe Caserta: The Data Warehouse ETL Toolkit, p. 353f.
[29] Brian Knight, et al.: Professional Microsoft SQL Server 2014 Integration Services, pp. 101ff, 105ff, 110, 104f.
[30] https://ssisctc.codeplex.com/.
[31] http://microsoft-ssis.blogspot.de/2012/01/custom-ssis-component-foreach-sorted.html.
[32] http://www.cdata.com/drivers/google/ado/.
[33] http://www.cdata.com/drivers/gsheets/ssis/.
[34] http://cdn.CData.com/help/RL1/rssis/RSBGSheets_p_DetectDataTypes.htm.

CHAPTER

LOADING THE DATA VAULT 12

This chapter focuses on the loading templates for the Raw Data Vault. These templates are built on some basic rules and best practices that have been accumulated over multiple years of experience. The patterns have evolved because of multiple performance issues with legacy ETL code. The top issues that affect the performance of the ETL loads are:

1. **Complexity:** the performance is affected by the variety of the data structures that need to be loaded into the data warehouse, but also by the data type alignment and the conformance of the data to set definitions. This is becoming a significant problem when dealing with highly unstructured data where the data structure changes from one record (or document) to the next.
2. **Data size:** volume also plays an important role regarding data warehouse loads. The more data is loaded, the more exposed are performance problems in the loading architecture and design.
3. **Latency:** the velocity of the data sources influences the frequency of the incoming data. If data needs to be loaded with high frequencies, small problems in the data flow will be exaggerated, leading to many more issues than before. Fast-arriving data also prohibits complexity in the processing stream because, depending on the infrastructure used, data might be lost if processing takes too long.

The complexity of the loading patterns is often additionally influenced by business rules that need to be processed upstream of the data warehouse. While the intention is to reduce the complexity of the data, the actual effect is that these business rules make the processes more complex because they change. The effect of the changed business rule has to be taken into consideration in later stages, which represents the majority of the increase in complexity.

When analyzing the fundamental issue of the ETL performance in many data warehouse projects, findings based on set logic (shown in Figure 12.1) are important to understand.

When dealing with a specific number of records that are loaded into the staging area, only about 60 to 80% are inserted records that have never been seen by the data warehouse before; 10 to 20% are updates of descriptive data to keys that are already in the data warehouse. And 5% of the incoming data describe deletes in the source system that should be tracked by the data warehouse as soft-deletes.

If only 20% of, e.g., 10 million rows (that is, 2 million rows) were identified by key, to be updates, then the aggregations of those 2 million rows would be even less – especially by month, quarter, and year. Those updates will identify specific sets of denormalized keys to be updated, along with a specific point in time. It would then be easy to meet performance objectives in a well-tuned relational database management system (RDBMS) environment.

However, many data warehouse developers construct their ETL routines to deal with the whole dataset in staging. If the problem were separated, each resulting individual process would have to deal with a decreasing process complexity, which makes it possible to increase performance, as we will learn a little later.

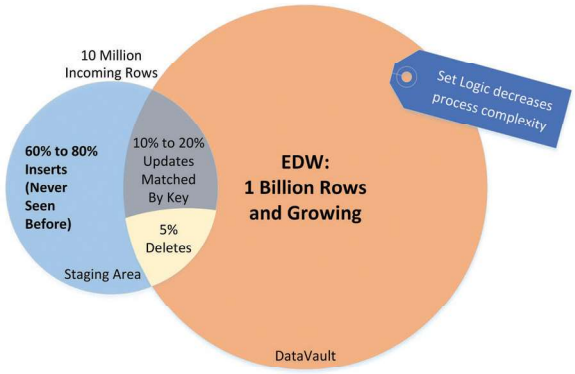

FIGURE 12.1

Set logic for data warehouses.

The key to improving the performance of data warehouse loading processes is based on two ideas:

1. **Divide and conquer the problem:** separate the processing of the loading procedures into separate groups in order to deal with smaller problems using a focused approach. In data warehousing, there are two different goals that should be reached individually: data warehousing and information processing.
2. **Apply set logic:** reduce the amount of data each process deals with by separating the data into different processes and reducing the amount of data as it is being processed.

The first idea is depicted in Figure 12.2: the two major goals are further divided into two separate activities each.

Data warehousing is different from sourcing of data because it has different issues:

- **Latency issues:** the performance of subsequent processes is affected by the latency of the source. Because the latency of the actual source system is not stable and varies over time, the staging area is used during sourcing to provide a stable latency of the incoming data. In other cases, data arrives in real-time which has to be taken into consideration by the sourcing application as well.
- **Arrival issues:** not all data arrives at the same time. The staging area is used to buffer the incoming data and makes sure that all dependencies are met if actually required.

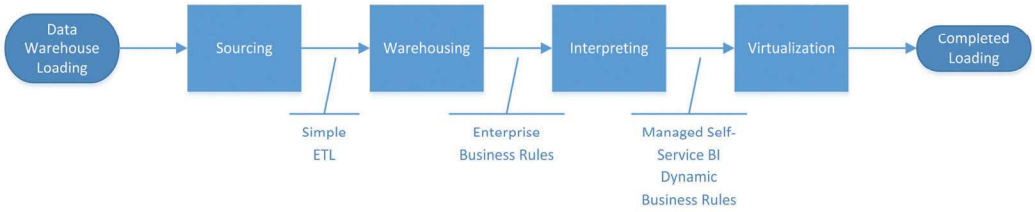

FIGURE 12.2

Overall loading process of the data warehouse.

- **Network issues:** the network could be the source of errors as well, for example when the network connection is too slow or important network devices or servers are unavailable.
- **Security issues:** in other cases, the data warehouse doesn't have access to the source system because the password was changed or expired.

This is the reason why the data is "dumped" into the staging area in order to separate these problems from the actual data warehousing. Each activity in Figure 12.2 deals with a separate problem. Some of these problems are also neutralized by using a NoSQL environment such as Hadoop or a similar technology.

In order to ensure the performance of the loading processes, the following rules should be followed:

1. **Decrease process complexity:** simplicity beats complexity. Simple loading processes are not only easier to maintain and administrate, they are also superior regarding performance.
2. **Decrease the amount of data:** reduce the amount of data that needs to be touched in order to load the target. This can also be achieved by using parallel processes where each process or thread is dealing with a smaller amount of data than the unparallelized process.
3. **Increase process parallelism:** the server is able to process multiple execution paths at the same time. For this reason, the intraoperation (inside one process) and interoperational (multiple processes) parallelism should be increased to take advantage of these capabilities.
4. **Combine all three:** to achieve superior performance, combine all three rules by reducing the complexity of the loading processes, decreasing the amount of data, and increasing the process parallelism, all at the same time. The best way to achieve this goal is to tackle one rule at a time.

Note that parallelism is **last** on the list, and should be avoided as long as possible, because any time a process is partitioned or parallelism is added, maintenance costs are increased. Instead, data warehouse teams should focus first on decreasing the complexity of the loading processes, yet many organizations don't even deal with this first rule. Also note that the parallelization requires that referential integrity be turned off in the Raw Data Vault, at least during load.

The overall loading process of the data warehouse implements these recommendations (Figure 12.3).

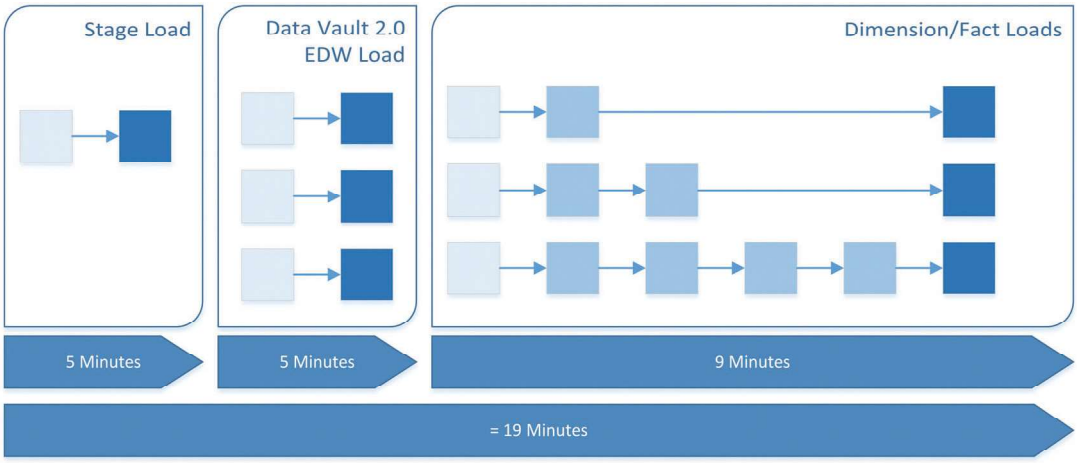

FIGURE 12.3

High-performance processes for the data warehouse.

Each phase of the loading process deals with a separate problem. By doing so, each phase also deals with progressively less data. Data is only loaded into the enterprise data warehouse (EDW) layer if it is new and unknown to the layer. And only data that is required for incrementally building the information mart is pushed into the next layer. This is also due to the fact that set logic has been applied to further reduce the amount of data to be dealt with. The separation of loading processes also favors highly simplified and focused loading patterns because each layer has a specific focus that is reflected in the loading patterns. The time each simplified phase requires is short due to high parallelization. Because of this short required time-frame, the overall process is short as well.

However, parallelization is not limited to one source system. The goal of the data warehouse loading processes is to load source systems as soon as they are ready to be loaded into the data warehouse. Waiting until a specific time in the night when all source systems are ready to be loaded into the data warehouse should be avoided. Instead, once a source system has provided its data to the data warehouse, it is staged, loaded to the data warehouse and, if possible, all information marts that directly depend on this data are processed. Figure 12.4 depicts this staggered approach.

In the figure, the data from multiple source systems is provided to the data warehouse at different times. Once the data is available to the data warehouse, it is staged immediately. It is also possible to load the data into the Raw Data Vault because there are no dependencies that require any waiting. In the case of Figure 12.4, all information marts depend on only one data source, which is a simplified example. In reality, there are synchronization points that have to be taken care of.

However, some of the dimension and fact tables that only depend on the one source system or have all other dependencies met (all other source systems loaded already) can be processed. This approach helps to take advantage of the available data warehouse infrastructure and increases the load window of the data warehouse because the peak in the loading process is reduced. It disperses the load of the engine over time and makes it possible to increase the number of loads for some of the source systems. Instead of loading the data only once a day, the data can be sourced multiple times a day because the loading processes are independent from each other and computing power remains available. Loading the data from a single source system multiple times a day also has the advantage that each single run can deal with less data if delta loading is used. However, it also requires that the data warehouse meet more operational requirements, such as a better uptime and more resiliency and elasticity of the data warehouse infrastructure.

This chapter presents the recommended templates for loading patterns of many Data Vault 2.0 entities. These templates are based on experience and optimization from projects and take full advantage of the Data Vault 2.0 model and the rules and recommendations outlined earlier in this chapter. Examples in T-SQL and SSIS are also provided for each presented template. It is easy to adapt the examples to ANSI SQL or other ETL engines.

12.1 LOADING RAW DATA VAULT ENTITIES

Similar to the staging area, the enterprise data warehouse layer has to be materialized. This EDW layer is responsible for storing the single version of the facts, at any given time. Therefore, virtualization is not an option for most entities in the EDW layer (some exceptions are discussed in section 12.2.1) and the Business Vault, which is discussed in Chapter 14, Loading the Dimensional Information Mart.

Implementing the loading processes for the Raw Data Vault only requires simple SQL statements, such as INSERT INTO <target> SELECT FROM <source> statements. However, organizations often

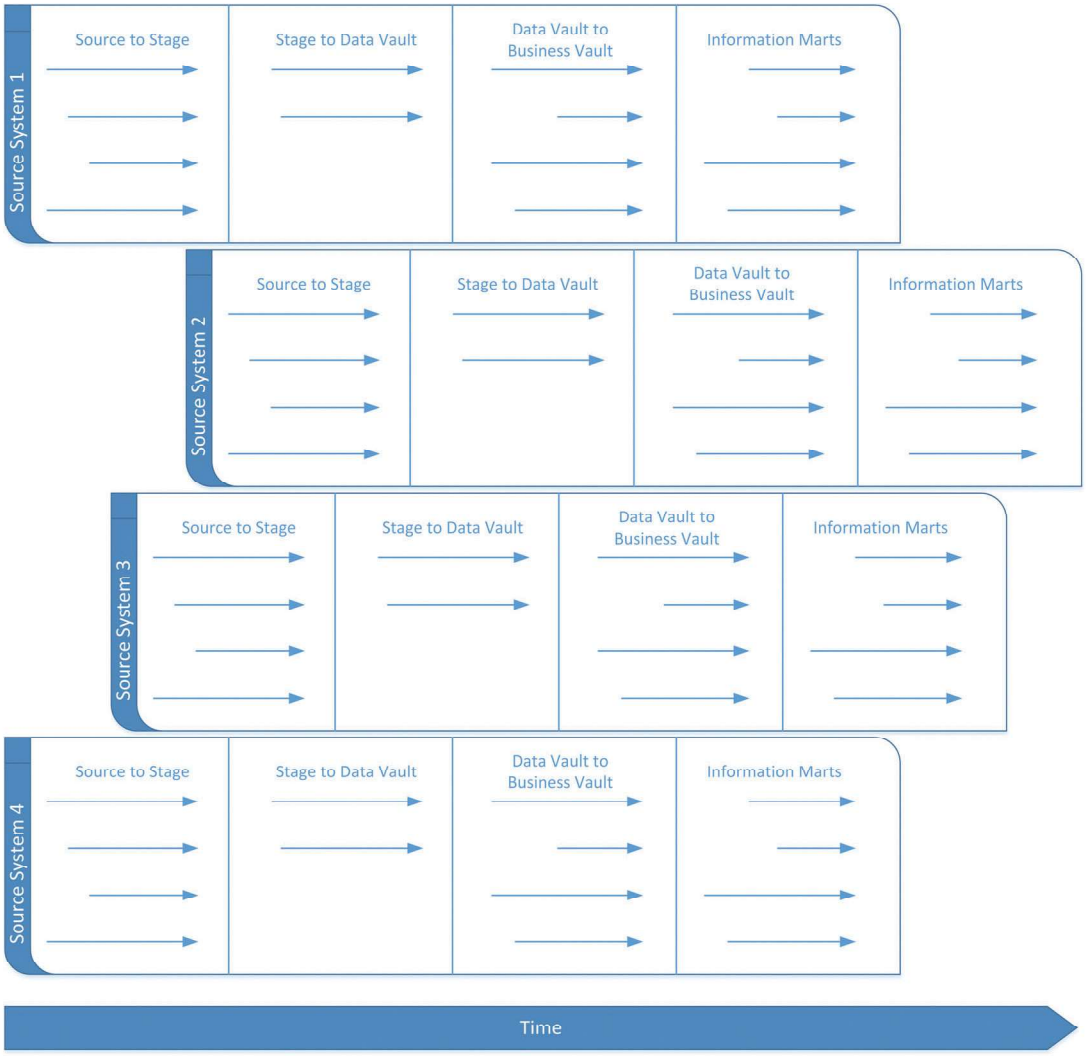

FIGURE 12.4

Staggering the load process of multiple source systems.

prefer to implement loading processes in ETL to more easily offload the required processing power to their existing ETL infrastructure, leveraging their ETL investments. For this reason, both options using SSIS and T-SQL are demonstrated in the next sections: each section focuses on one target entity, such as hubs, links, and satellites, including their special cases. The goal of the Raw Data Vault is to store raw data, so all patterns move raw data only. The data is extracted from the staging table and loaded into the target entity in the Raw Data Vault without modifications.

Note that the loading patterns use the load date from the staging area. However, it is also possible to create a new load date within the loading processes (for example, using GETDATE() or a similar

function). The key is to control the load date within the data warehouse and not rely on a third-party system such as an operational source application.

The performance of the loading patterns is not only due to the design principles for data warehouse loading, as outlined in the previous section. It is also due to the use of hash keys instead of sequence numbers. The hash keys are used to overcome lookups into parent tables, which hinder parallelization and reduce performance due to higher I/O requirements. The characteristics of using hash keys in data warehousing have been discussed in the Chapter 11, Data Extraction. The loading templates presented in this chapter take direct advantage of these characteristics.

12.1.1 HUBS

When loading data from the staging area into the enterprise data warehouse layer, the Data Vault hubs are supposed to store the business keys of the source systems. This list has to be updated only in a single regard: in each load cycle new keys that have been added to the source system are added to the target hub. If keys get deleted in the source system, they remain in the target hub. The template for loading hub tables is presented in Figure 12.5.

The first step is to retrieve all business keys from the source. This step is not as trivial as it sounds. In many cases, there are multiple source tables within one source system that provide the same business keys because they are shared across the system. For example, a passenger table provides a list of passenger identification numbers and a table holding ticket information references the passenger using its business key (see Figure 12.6).

In order to guarantee that all passenger identification numbers have been sourced to the target hub table **HubPassenger**, the keys from all source tables have to be sourced using a UNION operation, as shown in this figure, or by running the template in Figure 12.5 multiple times, for each source table. In many cases, the second approach is the most favorable because it makes automation of the loading processes more feasible, for example by allowing a metadata driven definition of the loading patterns (refer to Chapter 10, Metadata Management, for more details).

In both cases, the question becomes which record source is set in the target hub table. The record source in the target reflects the detailed source system table that has provided the business key to the data warehouse for the first time. If multiple tables (or even multiple source systems) provide the business key at the same time, a predefined master system table is set as the record source. This master system table

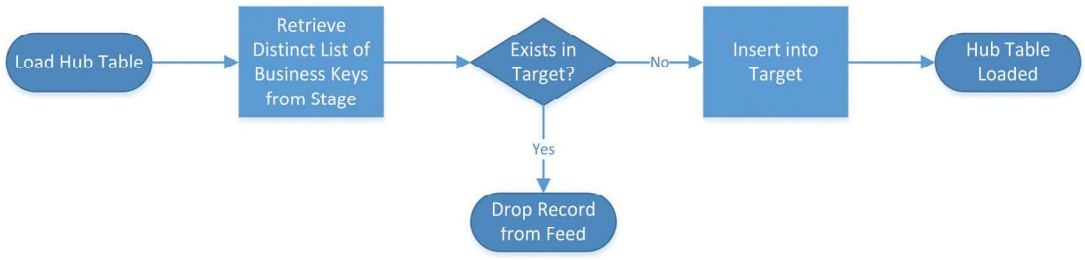

FIGURE 12.5

Template for loading hub tables.

FIGURE 12.6

Combining business keys from multiple sources.

is usually identified in the metadata and has the highest priority. If multiple patterns are used to load the same target hub table, the source table with the highest priority (as defined in the metadata) is sourced first. This way, the key found in the source tables with the highest priority will set the record source, because the duplicates in other sources will not be sourced anymore.

This behavior is guaranteed by the second step in Figure 12.5, which checks if the business key already exists in the target. This is done by performing a lookup into the target table to find out if the business key from the source already exists. Only new keys are processed further. Because this lookup should be performed only once per business key, only a distinct list of business keys were sourced from the source table in the staging area in the first step. This follows the approach of reducing the amount of data processed by the data flow as soon as possible.

Because the staging area contains both the business key and the hash key, both keys are sourced from the staging table and become available in the data flow. The lookup is performed using the business key and not the hash key. This is because of the rare risk of hash collisions described in the previous chapter. If two business keys produce the same hash key, a lookup on the hash key would return that the hash key already exists in the target. But due to the hash collision, this would mean a different business key. Performing the lookup on the business key will return a hit only if the business key already exists in the target table, regardless of the hash key. However, in the case of a hash collision, this will result in a primary key constraint violation when inserting the record in the next step. This is not the preferred solution, but it is the desired one, because we'd like to be notified in case of collision. Despite the low risk of a hash collision, the impact is high: if a collision happens, the hash function should be upgraded as described in Chapter 11, Data Extraction.

If a business key already exists in the target, the record is dropped from the data flow. Only if the business key doesn't exist in the target hub table, it is inserted into the target table. As already stated, the hash key is not calculated in this process because it is sourced from the staging table. If the business key consists of multiple parts, a so-called composite key, the process remains the same. The only difference is that the lookup has to include the whole composite key when performing the lookup operation.

Because the hub table stores all business keys that have been in the source systems in any point of time, this process doesn't delete any business keys or updates them for any reason. Once a business key has been inserted to the hub table, it remains there forever (not quite true if data has to be destructed for legal or other reasons).

12.1.1.1 T-SQL Example

The following DDL creates a target hub for the airport code found in the BTS performance data:

```
CREATE TABLE [raw].[HubAirportCode](
       [AirportCodeHashKey] [char](32) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [AirportCode] [nvarchar](3) NULL,
 CONSTRAINT [PK_HubAirportCode] PRIMARY KEY NONCLUSTERED
(
       [AirportCodeHashKey] ASC
) ON [INDEX],
 CONSTRAINT [UK_HubAirportCode] UNIQUE NONCLUSTERED
(
       [AirportCode] ASC
) ON [INDEX]
) ON [DATA]
```

Throughout this book, the **raw** schema is used to store the tables and other database objects of the Raw Data Vault. The hub contains a hash key, which is based on the airport code columns from the source. In addition, it contains a load date and a record source column. The defining column is the **AirportCode** column at the end of the table definition. The use of a nonclustered primary key is strongly recommended because the hash key is used as the primary key. Because the hash key has a random distribution, the keys will not arrive in any order. If a clustered primary key were used, Microsoft SQL Server would rearrange the data rows on disk whenever a new record arrived. This would slow down INSERT performance and provide no value to the retrieval of data from the table. Therefore, we always use nonclustered primary keys, except for very small tables with a small number of inserts. In addition to the primary key on the hash key column, an alternate key is added in order to speed up lookups on the business key.

In order to load newly arrived airport codes from the source table in the staging area, the following T-simplified SQL command can be used:

```
INSERT INTO DataVault.[raw].HubAirportCode (
       AirportCodeHashKey, LoadDate, RecordSource, AirportCode
)
SELECT DISTINCT
       OriginHashKey, LoadDate, RecordSource, Origin
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD
WHERE
       Origin NOT IN (SELECT AirportCode FROM DataVault.[raw].HubAirportCode)
       AND LoadDate = '1995-10-18 00:00:00.000'
```

Note that if the hub is defined by a composite business key, the NOT IN statement might not work, depending on the selected database management system. Microsoft SQL Server only supports one column in a NOT IN condition. However, it is possible to use a NOT EXISTS condition to overcome this limitation. This is done in the link loading statement in section 12.1.2.

The diversion airports are NULL if no flight diversion happened. This special case might be wrong from a business perspective, but nonetheless the NULL business key is also sourced into the hub because some links and satellites might use it as well (for example, see the link loading process in section 12.1.2).

Only keys which are unknown to the target hub should be inserted to avoid errors: if duplicate business keys or hash keys are inserted into the hub table, either the table's primary key on the hash key column or the alternate key on the business key column will raise a duplicate value error. To avoid the

error, the first check in the WHERE clause is to check whether the business key is already in the target hub. This approach is the safest, because it detects hash collisions (refer to Chapter 11). But it also requires an index on the business key for faster lookups.

Hashing the business keys is not required because this task was performed when loading the staging area. The loading process for hubs uses these hash keys. Other entities, such as links and hubs, which are hanging off this hub, will use the same hash key as well.

The **LoadDate** is included in each WHERE clause because the staging area could contain multiple loads. This is not the desired state, but if an error happened over the weekend, there might be multiple batches in the staging area that have not been processed yet. The goal of the loading process is to stage each batch in the order it was loaded into the staging area: the Friday batch comes before the Saturday batch, the batch at 08:00 comes before the batch at 10:00. Therefore, the previous statement should be executed for one load date only, not multiple. This is also important when loading the other Data Vault entities, especially links (due to the required change detection). Trying to load everything at once is possible, but complicates the loading process.

The previous statement is simplified because it loads the business keys only from the **Origin** column and not the other business key columns available in the source staging table **bts.OnTimeOnTimePerformance**. In order to load business keys from all source columns that provide airport codes, the following statements should be executed in parallel:

```
INSERT INTO DataVault.[raw].HubAirportCode (
       AirportCodeHashKey, LoadDate, RecordSource, AirportCode
)
SELECT DISTINCT
       OriginHashKey, LoadDate, RecordSource, Origin
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD
WHERE
       Origin NOT IN (SELECT AirportCode FROM DataVault.[raw].HubAirportCode)
       AND LoadDate = '1995-10-18 00:00:00.000';
GO

INSERT INTO DataVault.[raw].HubAirportCode (
       AirportCodeHashKey, LoadDate, RecordSource, AirportCode
)
SELECT DISTINCT
       DestHashKey, LoadDate, RecordSource, Dest
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD
WHERE
       Dest NOT IN (SELECT AirportCode FROM DataVault.[raw].HubAirportCode)
       AND LoadDate = '1995-10-18 00:00:00.000';
GO

INSERT INTO DataVault.[raw].HubAirportCode (
       AirportCodeHashKey, LoadDate, RecordSource, AirportCode
)
SELECT DISTINCT
       Div1AirportHashKey, LoadDate, RecordSource, Div1Airport
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD
WHERE
       Div1Airport NOT IN (SELECT AirportCode FROM DataVault.[raw].HubAirportCode)
       AND LoadDate = '1995-10-18 00:00:00.000';
```

```
GO

INSERT INTO DataVault.[raw].HubAirportCode (
       AirportCodeHashKey, LoadDate, RecordSource, AirportCode
)
SELECT DISTINCT
       Div2AirportHashKey, LoadDate, RecordSource, Div2Airport
FROM
       StageArea.bts.OnTimeOnTimePerformance
WHERE
       Div2Airport NOT IN (SELECT AirportCode FROM DataVault.[raw].HubAirportCode)
       AND LoadDate = '1995-10-18 00:00:00.000';
GO

INSERT INTO DataVault.[raw].HubAirportCode (
       AirportCodeHashKey, LoadDate, RecordSource, AirportCode
)
SELECT DISTINCT
       Div3AirportHashKey, LoadDate, RecordSource, Div3Airport
FROM
       StageArea.bts.OnTimeOnTimePerformance
WHERE
       Div3Airport NOT IN (SELECT AirportCode FROM DataVault.[raw].HubAirportCode)
       AND LoadDate = '1995-10-18 00:00:00.000';
GO

INSERT INTO DataVault.[raw].HubAirportCode (
       AirportCodeHashKey, LoadDate, RecordSource, AirportCode
)
SELECT DISTINCT
       Div4AirportHashKey, LoadDate, RecordSource, Div4Airport
FROM
       StageArea.bts.OnTimeOnTimePerformance
WHERE
       Div4Airport NOT IN (SELECT AirportCode FROM DataVault.[raw].HubAirportCode)
       AND LoadDate = '1995-10-18 00:00:00.000';
GO

INSERT INTO DataVault.[raw].HubAirportCode (
       AirportCodeHashKey, LoadDate, RecordSource, AirportCode
)
SELECT DISTINCT
       Div5AirportHashKey, LoadDate, RecordSource, Div5Airport
FROM
       StageArea.bts.OnTimeOnTimePerformance
WHERE
       Div5Airport NOT IN (SELECT AirportCode FROM DataVault.[raw].HubAirportCode)
       AND LoadDate = '1995-10-18 00:00:00.000';
GO
```

Because there are multiple columns that might provide an airport code, namely Origin, Dest, Div1Airport, Div2Airport, Div3Airport, Div4Airport and Div5Airport, all of these source columns and their corresponding hash keys have to be sourced. Because these statements insert into the same target table, a locking or synchronization mechanism might be required. Locking on the table level is most efficient and better from a performance standpoint, but requires that only one process at a time

inserts rows. Thus, the tables could either be executed in sequence or should recover from a deadlock by automatically restarting the process. Locking on the row-level and executing (and committing) only micro-batches enable the use of full-parallelized execution without further handling.

12.1.1.2 SSIS Example

It is also possible to utilize SSIS for loading the business keys into the target hubs. Using SSIS also requires that only one batch be loaded into the Raw Data Vault at a time. For this reason, the SSIS variable shown in Figure 12.7 is created.

This variable is used to select only the data for one batch in the data flow. After this batch has been loaded to the Raw Data Vault, the end-date is calculated for the records and the next batch is loaded into the target.

Note that it is also possible to load all batches in one job, but this requires a more complex loading process, especially for satellites. On the other hand, loading multiple batches at the same time is often done during initial loads, which is not the everyday case. An exception to this rule is real-time loading of data, which is out of the scope of this book. If the staging area doesn't provide multiple loads, the use of this variable can be omitted. It is only required for loading historical data or multiple batches from the staging area into the enterprise data warehouse layer.

To implement the hub loading statement from the previous section in SSIS, each statement is implemented as its own data flow. Before configuring the source components, create a new OLE DB connection manager to set up the database connection into the staging area (Figure 12.8).

Select the **StageArea** database from the list of available databases. Once all required settings are made, select the **OK** button to close the dialog. Drag an **OLE DB source** component to the canvas of the dataflow. Open the editor and configure the source component as shown in Figure 12.9.

Use the following SQL statement to set up the SQL command text:

```
SELECT DISTINCT
       OriginHashKey AS AirportCodeHashKey,
       LoadDate,
       RecordSource,
       CONVERT(nvarchar(3), Origin) AS AirportCode
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD
WHERE
       LoadDate = ?
```

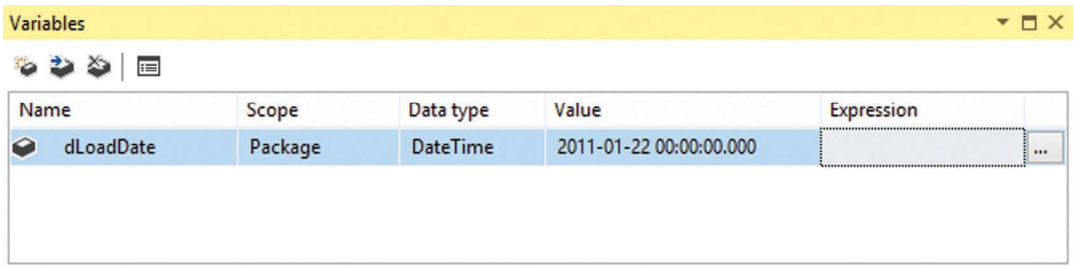

FIGURE 12.7

Adding load date variable to SSIS.

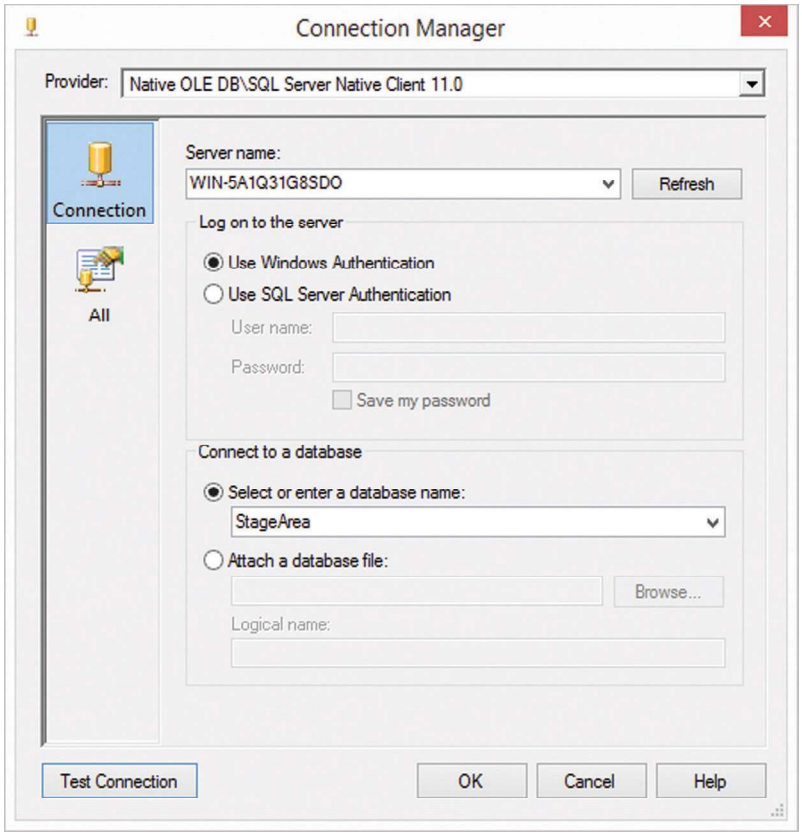

FIGURE 12.8

Setup connection to staging area.

Table 12.1 lists the other SQL statements that should be used in the data flows for additional columns from the source table in the staging area. Each SQL statement overrides the name of the incoming field to match the field name of the column in the destination hub table, which is not required but makes automation (or copy and paste) easier. It also makes the handling of the individual streams easier later on. Other than that, there are only two filters applied: the first filter is the DISTINCT in the SELECT clause that ensures that no duplicate business keys are retrieved from the source. The second filter is implemented in the WHERE clause to load only one batch from the staging area, based on the variable defined earlier. The parameter is referenced using a quotation mark in the SQL statement. In order to associate it with the variable, use the **Parameters…** button. The dialog in Figure 12.10 is presented.

Select the variable that was previously created and associate it with the first parameter. Select the **OK** button to complete the operation. Close the OLE DB Source Editor to save the SQL statement in the source component.

This completes the setup of the OLE DB source transformation. Close the dialog and drag a lookup transformation into the data flow. Connect the output of the OLE DB source transformation to the lookup and open the editor (Figure 12.11).

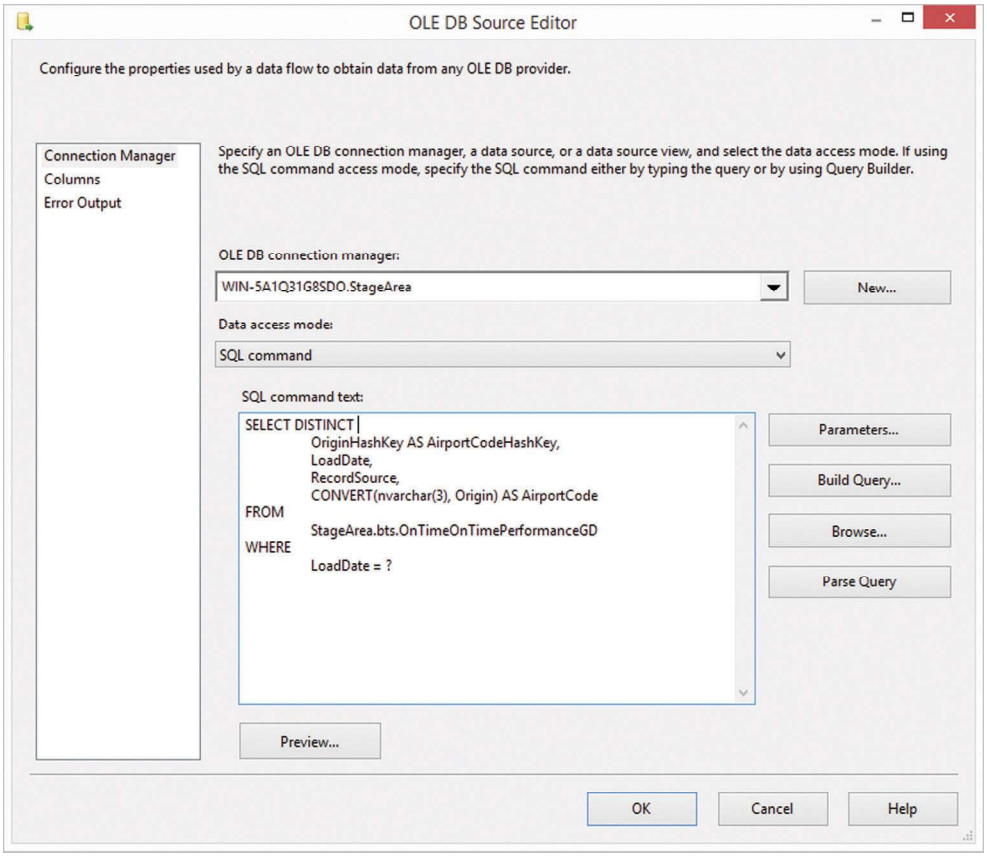

FIGURE 12.9

OLE DB source editor for retrieving the business keys from the staging area

The lookup is required to filter out the business keys that are already in the target hub from the data flow. After the lookup is performed, the records in the data flow are provided in two output paths:

- **Lookup match output:** this output contains all the business keys from the sources that are already known to the target hub. We are not interested in these business keys for further processing.
- **Lookup no match output:** this output provides all the business keys that have not been found in the target hub by this lookup. These keys should be added to the target.

This means that we are actually interested in those keys that haven't been found by the lookup. For this reason, it is important to **redirect rows to no match output** if the business key cannot be found in the destination. This setting can be set in the dialog page shown in Figure 12.11.

Switch to the next page, by selecting **Connection** from the list on the left. The dialog page is shown in Figure 12.12.

Table 12.1 SQL Statements for OLE DB Source Components

Name	SQL Command Text

Origin BTS Source	`SELECT DISTINCT`
	`  OriginHashKey AS AirportCodeHashKey,`
	`  LoadDate,`
	`  RecordSource,`
	`  CONVERT(nvarchar(3), Origin) AS AirportCode`
	`FROM`
	`  StageArea.bts.OnTimeOnTimePerformanceGD`
	`WHERE`
	`  LoadDate = ?`

Dest BTS Source	`SELECT DISTINCT`
	`  DestHashKey AS AirportCodeHashKey,`
	`  LoadDate,`
	`  RecordSource,`
	`  CONVERT(nvarchar(3), Dest) AS AirportCode`
	`FROM`
	`  StageArea.bts.OnTimeOnTimePerformanceGD`
	`WHERE`
	`  LoadDate = ?`

Div1Airport BTS Source	`SELECT DISTINCT`
	`  Div1AirportHashKey AS AirportCodeHashKey,`
	`  LoadDate,`
	`  RecordSource,`
	`  CONVERT(nvarchar(3), Div1Airport) AS AirportCode`
	`FROM`
	`  StageArea.bts.OnTimeOnTimePerformance`
	`WHERE`
	`  LoadDate = ?`

Div2Airport BTS Source	`SELECT DISTINCT`
	`  Div2AirportHashKey AS AirportCodeHashKey,`
	`  LoadDate,`
	`  RecordSource,`
	`  CONVERT(nvarchar(3), Div2Airport) AS AirportCode`
	`FROM`
	`  StageArea.bts.OnTimeOnTimePerformance`
	`WHERE`
	`  LoadDate = ?`

Div3Airport BTS Source	`SELECT DISTINCT`
	`  Div3AirportHashKey AS AirportCodeHashKey,`
	`  LoadDate,`
	`  RecordSource,`
	`  CONVERT(nvarchar(3), Div3Airport) AS AirportCode`
	`FROM`
	`  StageArea.bts.OnTimeOnTimePerformance`
	`WHERE`
	`  LoadDate = ?`

Table 12.1 SQL Statements for OLE DB Source Components *(cont.)*

Name	SQL Command Text

Div4Airport BTS Source	`SELECT DISTINCT`
	`  Div4AirportHashKey AS AirportCodeHashKey,`
	`  LoadDate,`
	`  RecordSource,`
	`  CONVERT(nvarchar(3), Div4Airport) AS AirportCode`
	`FROM`
	`  StageArea.bts.OnTimeOnTimePerformance`
	`WHERE`
	`  LoadDate = ?`

Div5Airport BTS Source	`SELECT DISTINCT`
	`  Div5AirportHashKey AS AirportCodeHashKey,`
	`  LoadDate,`
	`  RecordSource,`
	`  CONVERT(nvarchar(3), Div5Airport) AS AirportCode`
	`FROM`
	`  StageArea.bts.OnTimeOnTimePerformance`
	`WHERE`
	`  LoadDate = ?`

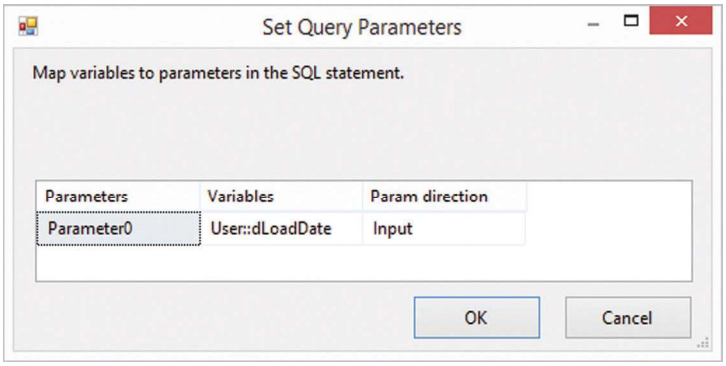

FIGURE 12.10
Set query parameters dialog.

Select the **DataVault** database connection and the **HubAirportCode** as the lookup table from the list of tables. You can use the **Preview…** button to check the data in the target. Select the **Columns** entry from the list on the left to switch to the columns page of the dialog (Figure 12.13).

This page actually influences whether the process is able to detect hash collisions. If the business key is used for the equi-join, hash collisions would be detected when inserting a new business key with an already existing hash key into the target hub, because it would violate the primary key on the hash key. If the hash key were used for the equi-join, duplicates wouldn't be detected because they share the same hash key. In fact, the new business key wouldn't be loaded into the target because the hash key on the business key is already present (has been found by the lookup). This would violate the definition of the hub, which is defined as a distinct list of business keys (and not hash keys). Therefore, the best choice is to use the business key

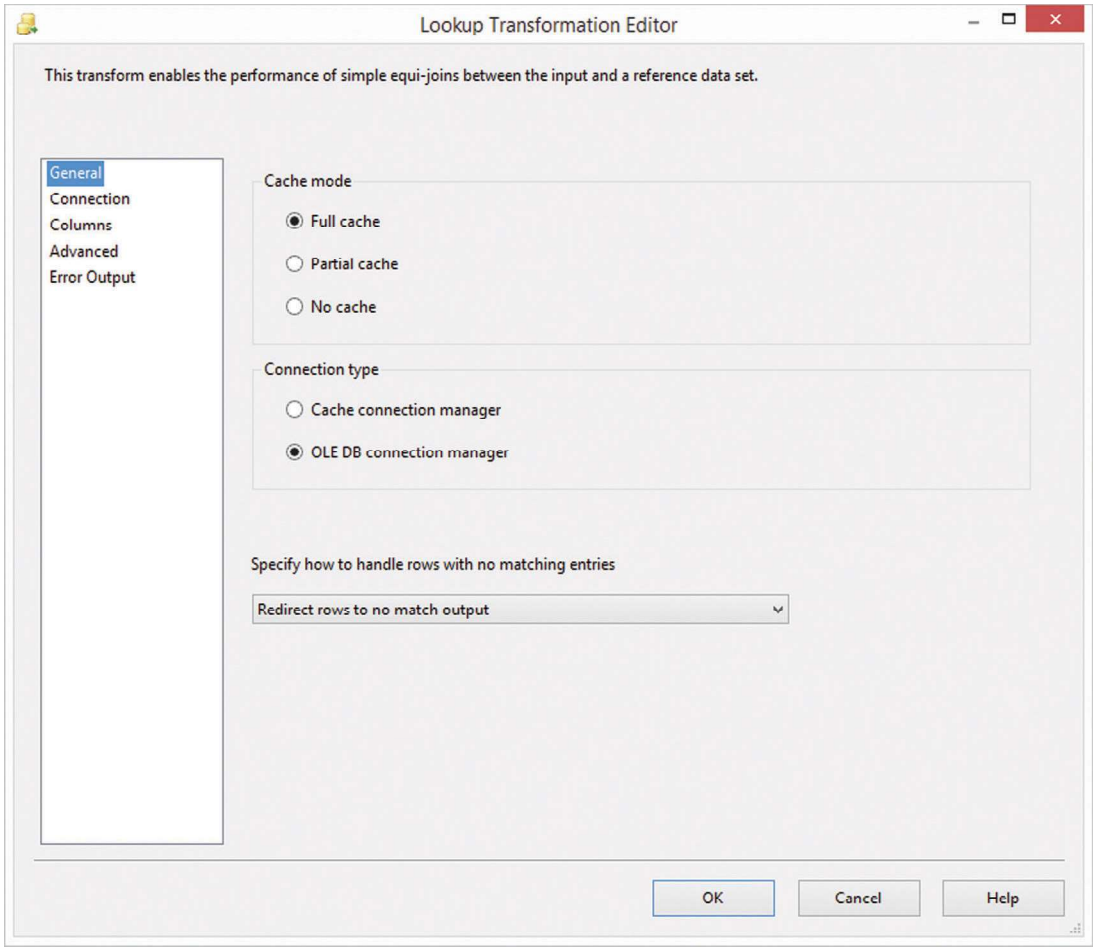

FIGURE 12.11

Lookup transformation editor to find out which business keys already exist.

in the equi-join operation of the lookup. It is not required to load any columns from the lookup into the data flow, because the only thing we're interested in is whether the business key is in the target hub table or not.

The last step in the process is to set up the destination. First, set up the OLE DB connection manager (Figure 12.14).

Once the connection manager has been set up, insert an OLE DB destination to the data flow canvas and connect the output from the lookup to the destination (Figure 12.15).

Once the path is connected, SSIS asks for the selection of the output (from the lookup transformation) that should be connected to the input of the OLE DB destination. This is because the lookup provides two outputs (plus the error output). Because we're interested in loading unknown business keys to the destination, select the **lookup no match output** in this dialog. Close the dialog and open the editor of the OLE DB destination (Figure 12.16).

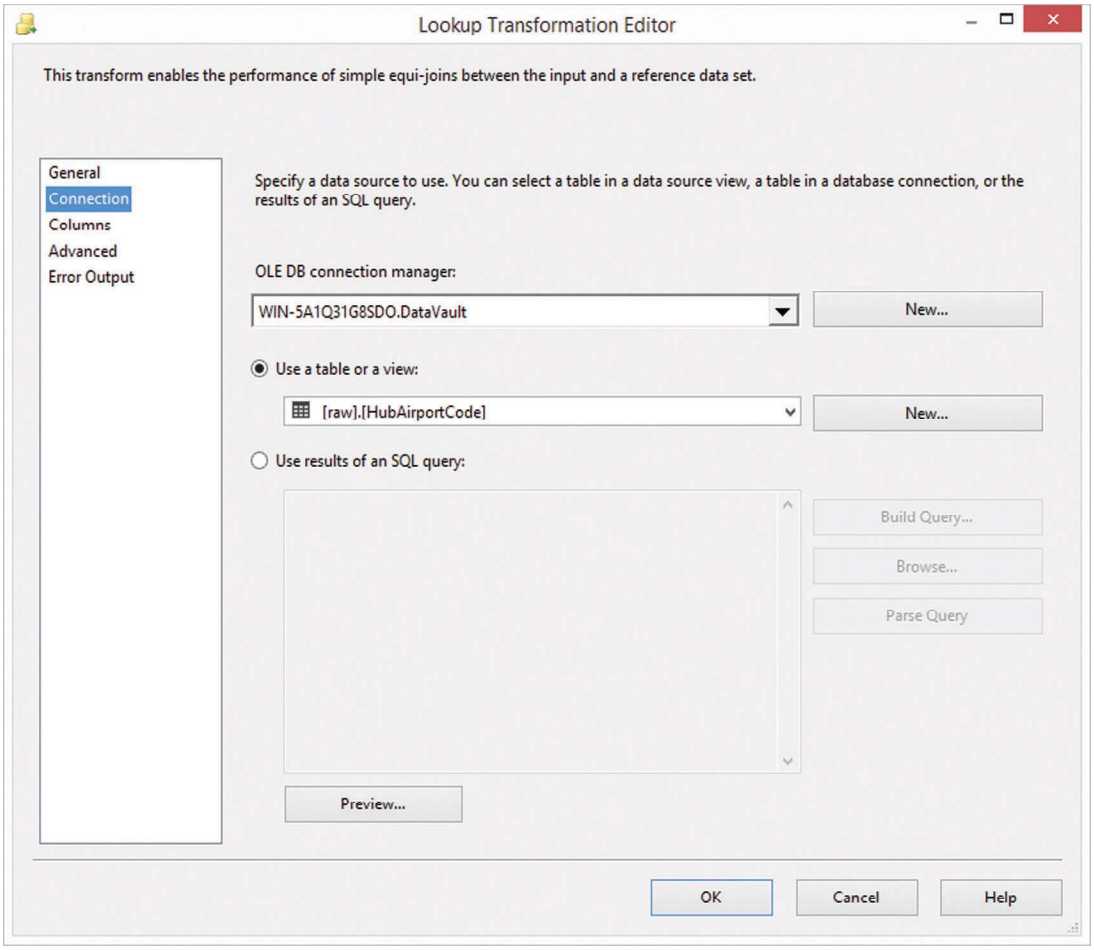

FIGURE 12.12

Setup lookup connection in the lookup transformation editor.

Because Data Vault 2.0 is based on hash keys, no identities are used. Therefore, **keep identity** is turned off. The option **keep nulls** should be checked in order to load a potential NULL business key, which is in line with the INSERT statement from the previous section. The table lock speeds up the loading process of the target table. However, it might require putting individual data flows with the same target table into a sequence in the control flow to avoid having to deal with deadlocks. Check constraints should be turned off as well. It is not recommended to use check constraints in the Raw Data Vault for two reasons: first, they reduce the performance of the loading process, and second, they often implement soft business rules, which should not be implemented at this point. Instead, they should be moved into the loading processes of the Business Vault or the information marts, which is covered in Chapter 14, Loading the Dimensional Information Mart.

Switch to the mapping page of the dialog by selecting the corresponding entry on the left side. The page shown in Figure 12.17 will appear.

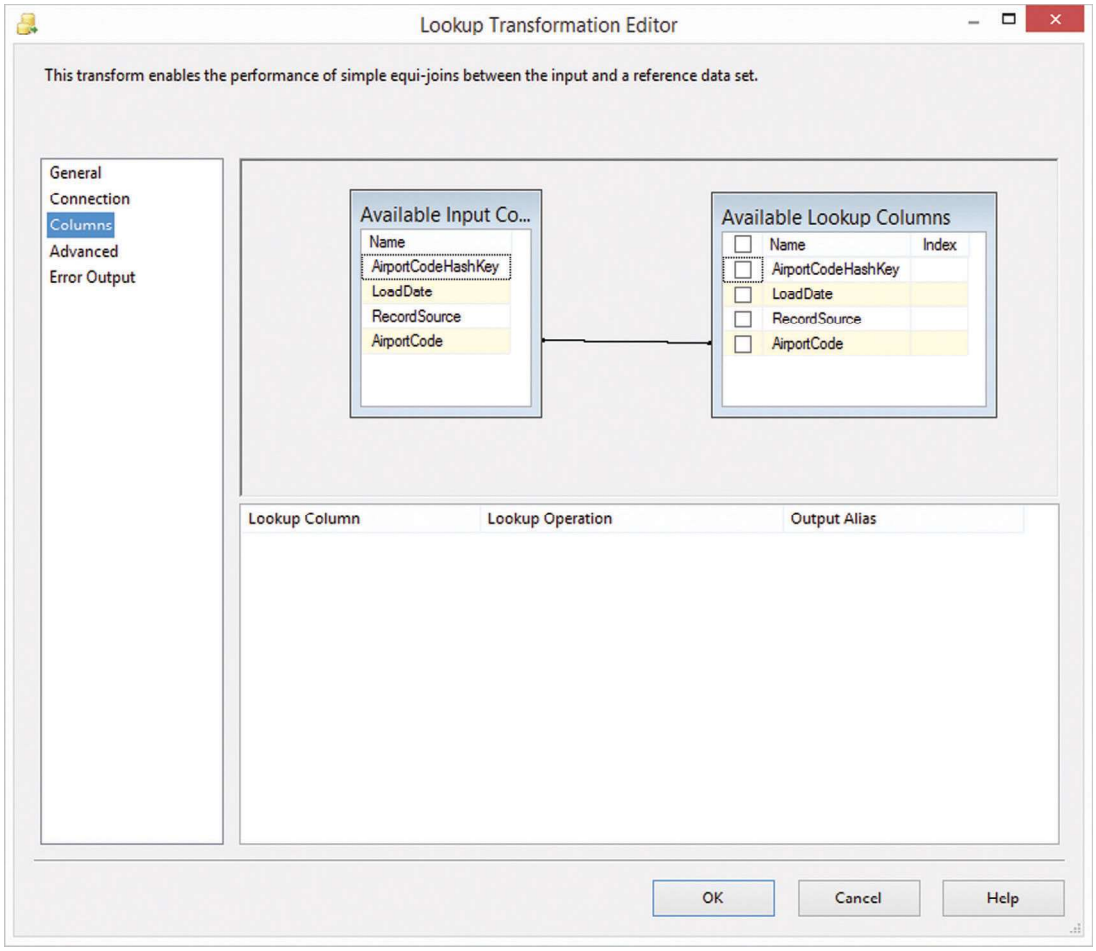

FIGURE 12.13

Configuring lookup columns in the lookup transformation editor.

Make sure that each column in the destination has been mapped to an input column. Select OK to close the dialog.

This completes the data flow for loading hub tables with SSIS. The final data flow is presented in Figure 12.18.

12.1.2 LINKS

The loading template for link tables in the Data Vault is actually very similar to the loading template of hub tables, especially if the hash keys are already calculated in the staging area, as is the recommended practice. The similarity of both templates is due to the fact that no lookups into the hubs referenced by the link are required. The loading template for links is completely independent of hub tables or any other tables (see Figure 12.19).

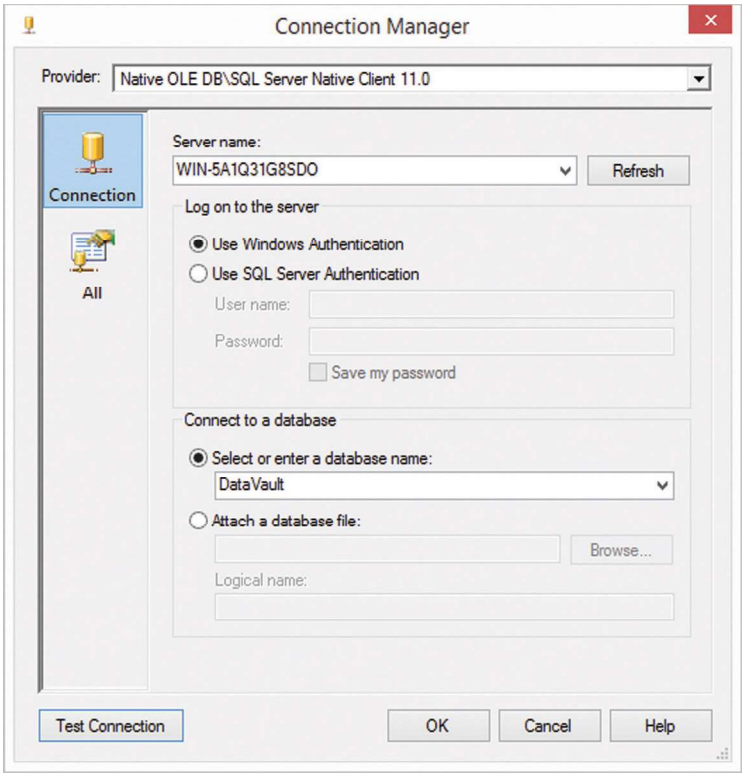

FIGURE 12.14

Configuring the OLE DB connection manager for the destination.

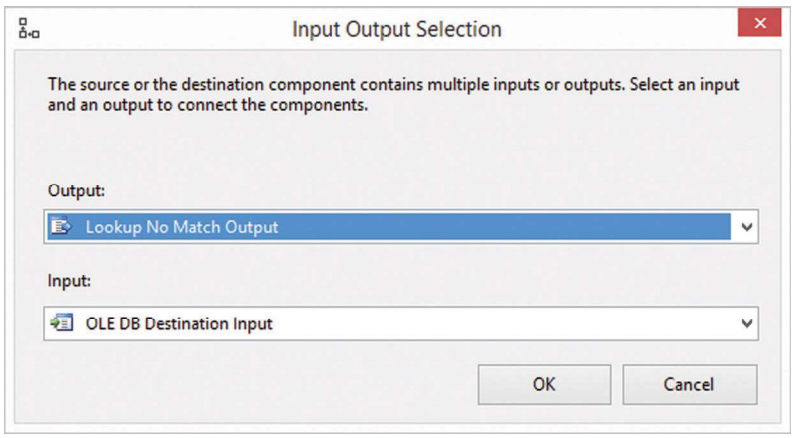

FIGURE 12.15

Output selection for destination component.

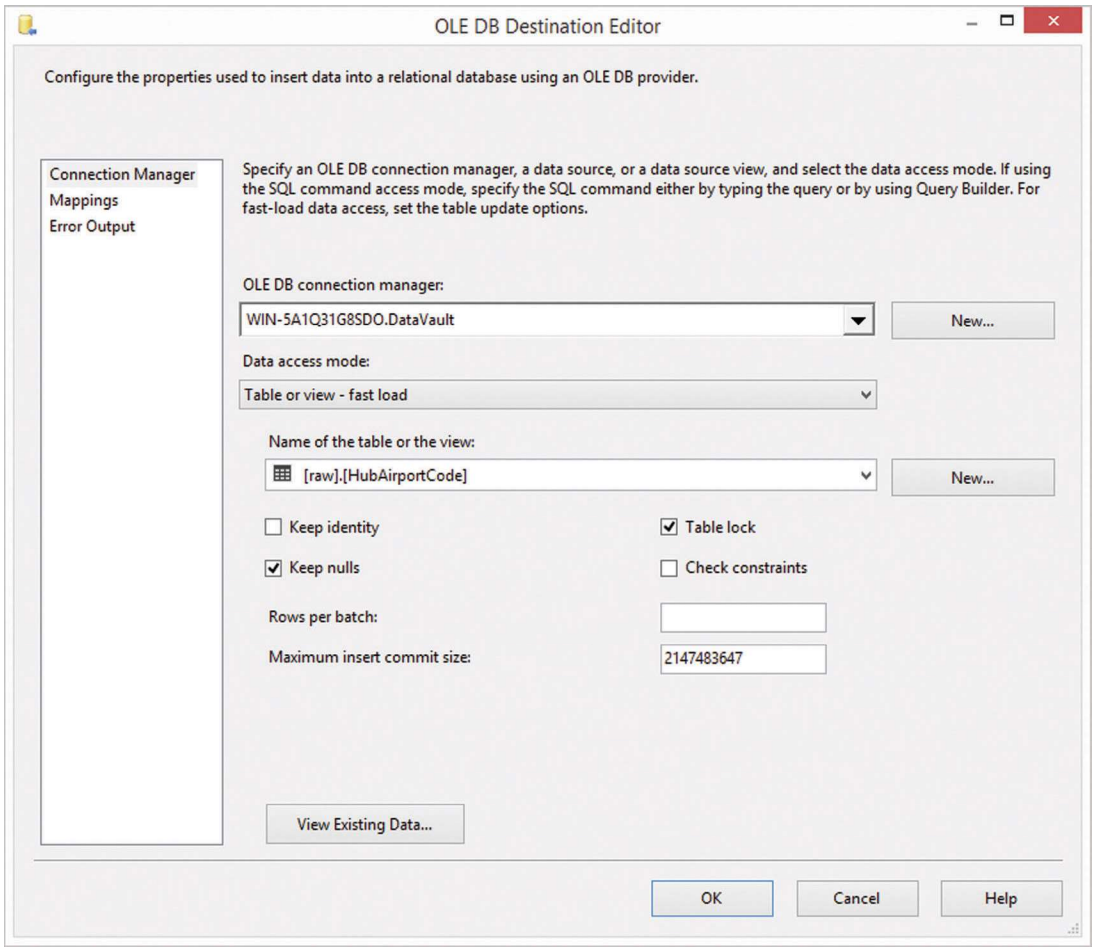

FIGURE 12.16

Setting up the connection in the OLE DB destination editor.

The first step in the loading process of links is to retrieve a distinct list of business key relationships from the source system. These relationships include not only the business keys; they also include the individual hash keys for the referenced hubs (where the business keys are defined) and the hash key of the relationship, which is the hash key of the link table structure.

Because only the hash keys are stored in the target link table, they are used to perform the lookup in the next step. This is sufficient in order to detect hash collisions in the link table. If the hash key of the input business key combination is the same for different inputs, the individual hash keys of the hub references will also be the same as the colliding record in the link table. For that reason, the lookup in the second step would return that the link combination doesn't exist in the target link table, but the subsequent insert would fail, due to the fact that both links (with different business key references, thus different hash keys used to reference the hubs in the link relationship) share the same link hash key which is used as the primary key of the table.

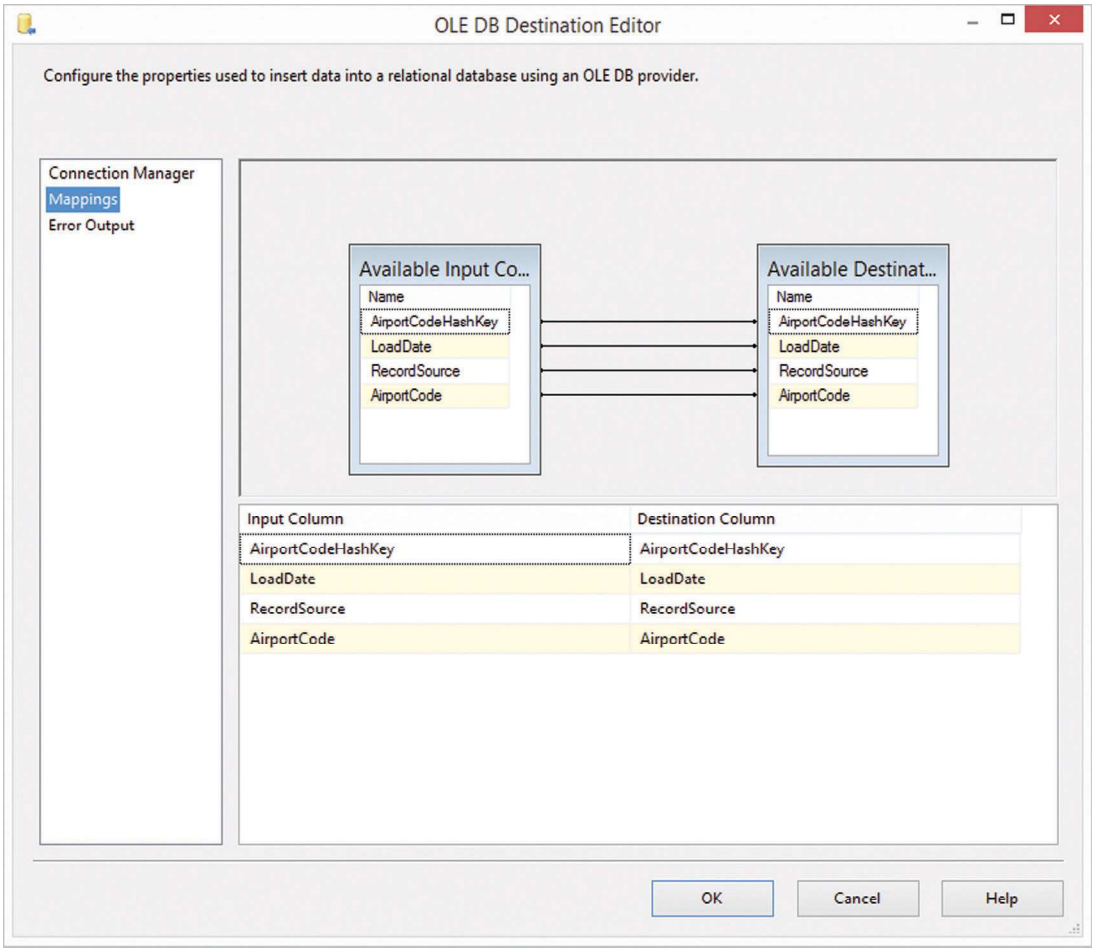

FIGURE 12.17

Mapping the columns from the data flow to the destination.

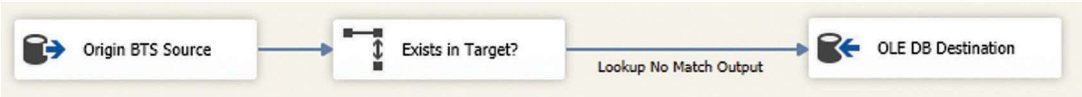

FIGURE 12.18

Data flow for loading Data Vault 2.0 hubs.

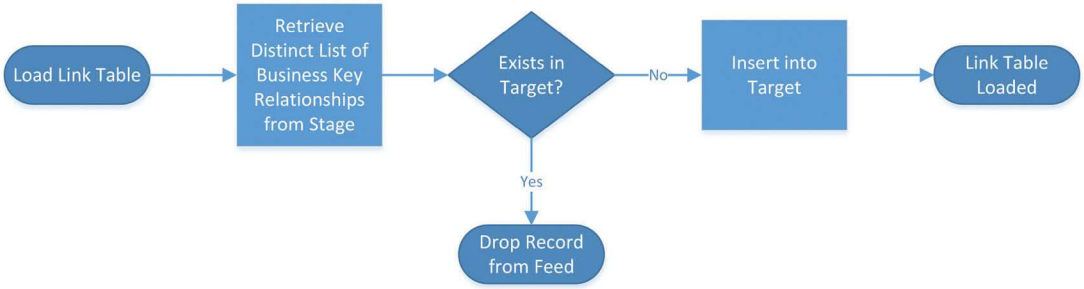

FIGURE 12.19

Template for loading link tables.

Note that the rare case in which there are hash collisions in all the hubs and the link table at the same time would be recognized as individual hash collisions in the hub table. However, such a risk is astronomically low.

In most, if not all, cases, where there is no hash collision, the link is inserted without any error into the target link table, assuming that the lookup returns no match. This insert operation includes the load date and record source from the source table in addition to the hash keys. If a match is returned, the link record is ignored and dropped from the data flow.

The only difference with the link loading template lies in the first step: instead of retrieving a single business key (or a composite business key) with its corresponding hash key, the link loading template retrieves the individual hub references: not the business keys, but their hash keys and the combined hash key, which will be used for the primary key of the link table. Another difference is the lookup, which does not use business keys but the hash keys of the hub references. The reason lies in the fact that the link table doesn't contain any business keys. Other than that, the link loading template is the same as the hub loading template. For this reason, it's a good practice to start implementing the hub loads first, and then adopt the hub loading template to be used with link tables.

Another similarity with the hub loading template is that the links might be sourced from multiple source tables, which might be spread over multiple source systems. In this case, the links from the various sources are combined into one target link table. This requires a similar approach as with hub loads: either the links are sourced in sequence or using the same approaches for parallelization. Note that combining links with different grain into the same target link table should be avoided. This leads to hub references that are NULL and is called "link overloading." Instead of using such an approach, separate the links by their grain, which is expressed by the hub references, and ensure that descriptive data for each grain can be attached to the link structure using satellites on individual link tables.

12.1.2.1 T-SQL Example

Because of the similarity of the loading templates for hubs and links, the T-SQL statement for loading link tables is very similar to the hub loading statement. It loads a standard link table that was created with the following statement:

```
CREATE TABLE [raw].[LinkFlightNumCarrier](
       [FlightNumCarrierHashKey] [char](32) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [FlightNumHashKey] [char](32) NOT NULL,
       [CarrierHashKey] [char](32) NOT NULL,
 CONSTRAINT [PK_LinkFlightNumCarrier] PRIMARY KEY NONCLUSTERED
(
       [FlightNumCarrierHashKey] ASC
) ON [INDEX],
 CONSTRAINT [UK_LinkFlightNumCarrier] UNIQUE NONCLUSTERED
(
       [FlightNumHashKey] ASC,
       [CarrierHashKey] ASC
) ON [INDEX]
) ON [DATA]
```

This table is also created in the **raw** schema, as are all other tables of the Raw Data Vault throughout this book. Following the definition outlined in Chapter 4, Data Vault 2.0 Modeling, the table contains a hash key, which is used as the table's primary key and identifies each link entry uniquely. The table contains the load date and record source in the same manner, and for the same purpose, as the hub table. The difference in the definition lies in the following hash keys, namely **FlightNumHashKey** and **CarrierHashKey**, which store the parent hash keys of the referenced hubs.

The primary key is on the hash key of the link again and is nonclustered. In addition, the link table contains an alternate key on the hub references to ensure uniqueness of the link combination. The implicit index on this unique key is also used as an index for a lookup required in the loading process.

The loading process for links should only load newly arrived business key relationships from the source table in the staging area. For this reason, the insert statement is comparable to the hub statement, but is based on the hash keys from the referenced hubs and not on the business keys found in the source:

```
INSERT INTO DataVault.[raw].LinkFlightNumCarrier (
       FlightNumCarrierHashKey, LoadDate, RecordSource, FlightNumHashKey, CarrierHashKey
)
SELECT DISTINCT
       FlightNumCarrierHashKey, LoadDate, RecordSource, FlightNumHashKey, CarrierHashKey
FROM
       StageArea.bts.OnTimeOnTimePerformance s
WHERE
       NOT EXISTS (SELECT
                           1
                     FROM
                           DataVault.[raw].LinkFlightNumCarrier l
                     WHERE
                           s.FlightNumHashKey = l.FlightNumHashKey
                           AND s.CarrierHashKey = l.CarrierHashKey
                     )
       AND LoadDate = '2015-01-22 09:51:56.000'
```

The SELECT part in this statement uses a DISTINCT clause again, in order to avoid checking the same business key relationship multiple times. All hash keys have been calculated in the staging area again, which is used to simplify the actual loading and increase the reusability of the hash computation. Only relationships that don't exist in the target link table are inserted. Because link tables use more than one referenced hub, the statement uses a NOT EXISTS condition in favor to a NOT IN statement. The sub-query searches for relationships that consist of the same hash keys.

Similar to the use of the load date in the statement for loading hubs (refer to section 12.1.1), the previous statement is filtering the data to one specific load date. This is done in order to sequentially load the batches in the staging area to ensure that the deltas are correctly loaded into the data warehouse. Therefore, another statement is required to find out the available load dates in the staging area first, order them (the batch with the oldest load date should be loaded into the data warehouse first) and then execute the above statement per load date found in the staging area.

The statement doesn't check the validity of the incoming links. For example, in some cases, the source system provides a NULL value instead of a business key. In this case, the above statement maps NULL business keys to a hash key, which is the result of the hash function on an empty string. In order to maintain the data integrity, this key has to be added to the hub, as well. However, there are two cases that might happen when loading business key relationships:

1. **At least one expected business key was not provided:** in this case, the source system provided erroneous data because an expected business key, which was used in a foreign key reference, was not provided and is NULL.
2. **At least one optional business key was not provided:** the business key in the source system is optional and was not provided. This is not an actual error.

It is important to cover both cases in order to analyze both errors and design issues. If only one extra hub record is used to cover the NULL business keys, it is not possible to distinguish between these cases. For that reason, there should be actually two extra records in the target hub (see Table 12.2).

The first record indicates the first option: the expected business key was not provided by the source. This is an actual error. The business key is set to an artificial, system-generated value, in this case −1. The hash key is set statically as well, because it makes the identification easier. It could also be derived from the artificial business key.

The second record is used for business keys that are optional and not provided. For this case, the business key −2 was artificially set. The hash key is set to 22222222222222222222222222222222 (32 times the character "2"). This way, it is easy to identify records in the source system that are attached to the missing key for some reason. The third record in the table is a valid business key.

When data is loaded from the source and cannot be associated with a business key, for example because the business key column in a source flat file is left empty, the data in the satellite is associated with one of these entries in the hub table. If a business key is missing in satellites on hubs, many of these cases are attached to the first option that covers missing business keys.

However, especially when loading link tables from foreign key references in the source system, both cases are very common. If a NULL reference is found in the source system, this information is added to the link table (Table 12.3).

The first line describes a connection between Denver and an unknown destination airport. Because the record was expected but not provided by the source system, the hash reference into the airport hub is set to the hash key reserved for erroneous data.

The last record in Table 12.3 references the second reserved record in the airport hub, reserved for cases when an optional business key is not provided. This is not perfect, but also not an error.

In both cases, the incoming NULL value is replaced by the artificial business key from the hub, with the artificial hash key (or the hash key derived from the artificial business key). Once the link is loaded into the Raw Data Vault, it is possible to load satellite data that describes the erroneous or weird data. This descriptive data is typically found next to the foreign key reference, in the same source table.

By following this process, the data warehouse integrates as much data as possible and allows the data warehouse team and the business user to analyze any issues with the source system directly in the Raw Data Vault. The data will later be "cleansed" by business logic, when loaded into the information marts.

Table 12.2 Ghost Records in Airport Hub Table

Airport Hash Key	Load Date	Record Source	Airline ID
1111…	0001-01-01 00:00:00.000	SYSTEM	-1
2222…	0001-01-01 00:00:00.000	SYSTEM	-2
1aab…	2013-07-13 03:12:11.000	BTS	JFK

Table 12.3 Linkconnection With Expected and Unexpected Null References

Connection Hash Key	Load Date	Record Source	Carrier Hash Key	Source Airport Hash Key	Destination Airport Hash Key	Flight Hash Key
87af…	**2013-07-13 03:12:11**	BTS	8fe9… {UA}	3de7… {DEN}	1111… {-1}	a87f… {UA942}
28db…	**2013-07-13 03:12:11**	BTS	8fe9… {UA}	3de7… {DEN}	1aab… {JFK}	8df7… {UA4711}
9de7…	**2013-07-14 02:11:10**	BTS	8fe9… {UA}	3de7… {DEN}	1aab… {JFK}	9eaf... {UA123}
9773…	**2013-07-15 03:14:12**	BTS.x	8fe9… {UA}	2222… {-2}	9bbe… {SFO}	821a… {UA883}

12.1.2.2 SSIS Example

Loading Data Vault 2.0 links in SSIS follows a similar approach to loading hubs. Because links can also be provided by multiple source tables (within the same source system), each source that provides link records desires its own data flow. After adding an OLE DB source transformation to the newly created data flow, open the OLE DB source editor (Figure 12.20).

Select the **StageArea** database from the list of databases and set the **data access mode** to **SQL command**. Insert the following SQL statement into the editor for the **SQL command text**:

```
SELECT DISTINCT
       FlightNumCarrierHashKey,
       LoadDate,
       RecordSource,
       FlightNumHashKey,
       CarrierHashKey
FROM
       StageArea.bts.OnTimeOnTimePerformance
WHERE
       LoadDate = ?
```

This statement selects all links for a given batch from the source table in the staging area. The batch is indicated by the load date. This approach follows the hub loading process and requires using the variable defined in section 12.1.1 (Figure 12.21).

Map the parameter in the SQL command text to a SSIS variable by selecting the variable in the second column. The **param direction** should be set to **input** as it is the default. Close the dialog. This completes the configuration of the source component. Drag a lookup component into the data flow and connect its input to the default output from the OLE DB source. Open the lookup transformation editor, which is shown in Figure 12.22.

Similar to the hub loading process, the process shown in this section is interested in links that don't exist in the target table. That is why this lookup is performed against the target. To prevent a failure of the SSIS process, select **redirect rows to no match output** in the selection box. This will open another output that provides the unknown link structures that should be loaded into the target. Those that are already found in the target link table are ignored. This follows the link loading template outlined in section 12.1.2.

Switch to the **connection** page of this dialog by selecting the corresponding entry on the left side (Figure 12.23).

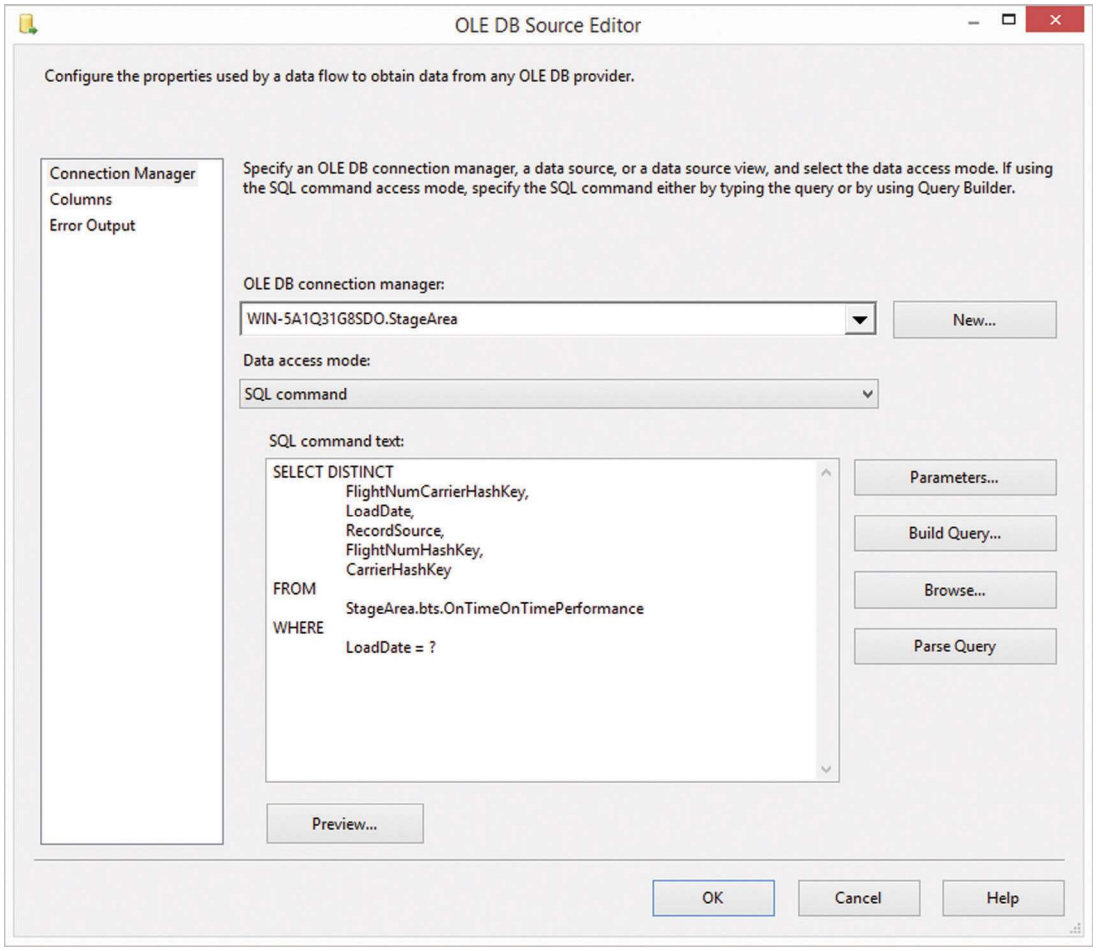

FIGURE 12.20

OLE DB source editor for loading links.

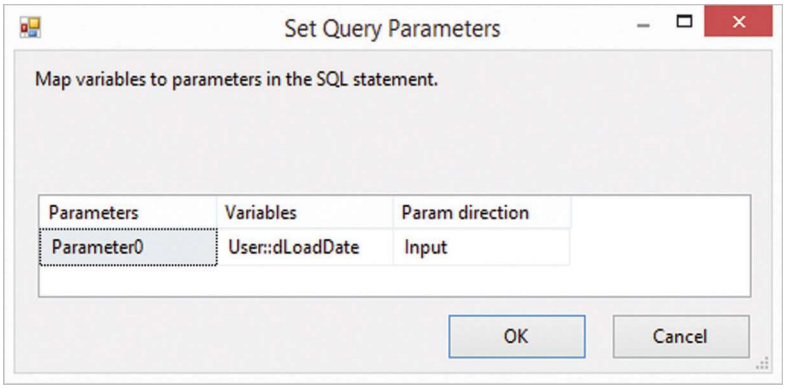

FIGURE 12.21

Set query parameter for load date timestamp.

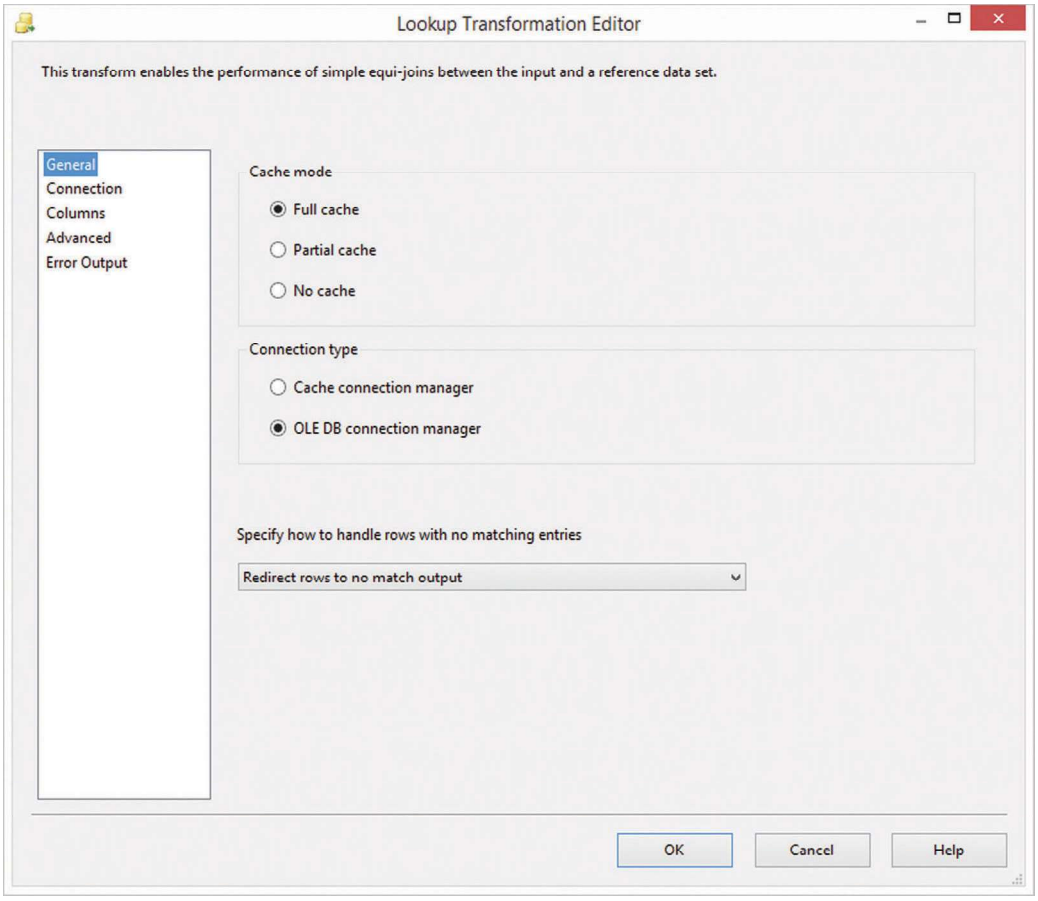

FIGURE 12.22

Set up the lookup transformation editor to find existing links.

This page sets up the connection to the lookup table. Because the goal of the lookup is to find out which links don't exist in the target table, select the **DataVault** database and the target table **Link-FlightNumCarrier**.

To complete the configuration of the lookup transformation, select the **columns** page on the left side of the dialog. The page shown in Figure 12.24 allows the configuration of the lookup columns and the columns that should be returned by the transformation.

Instead of using the hash key of the link (which is the primary key of the link table), the lookup should be performed on the hash keys of the referenced hubs. This improves the detection of hash key collisions. Because we're only interested in finding out which links are not in the target table yet, no columns are returned from the lookup table.

Select **OK** to close the lookup transformation editor. Insert an OLE DB destination and connect the output from the lookup to the destination. Because there are multiple outputs due to the redirection of unknown links in Figure 12.22, a dialog (Figure 12.25) is shown to map the output to the input.

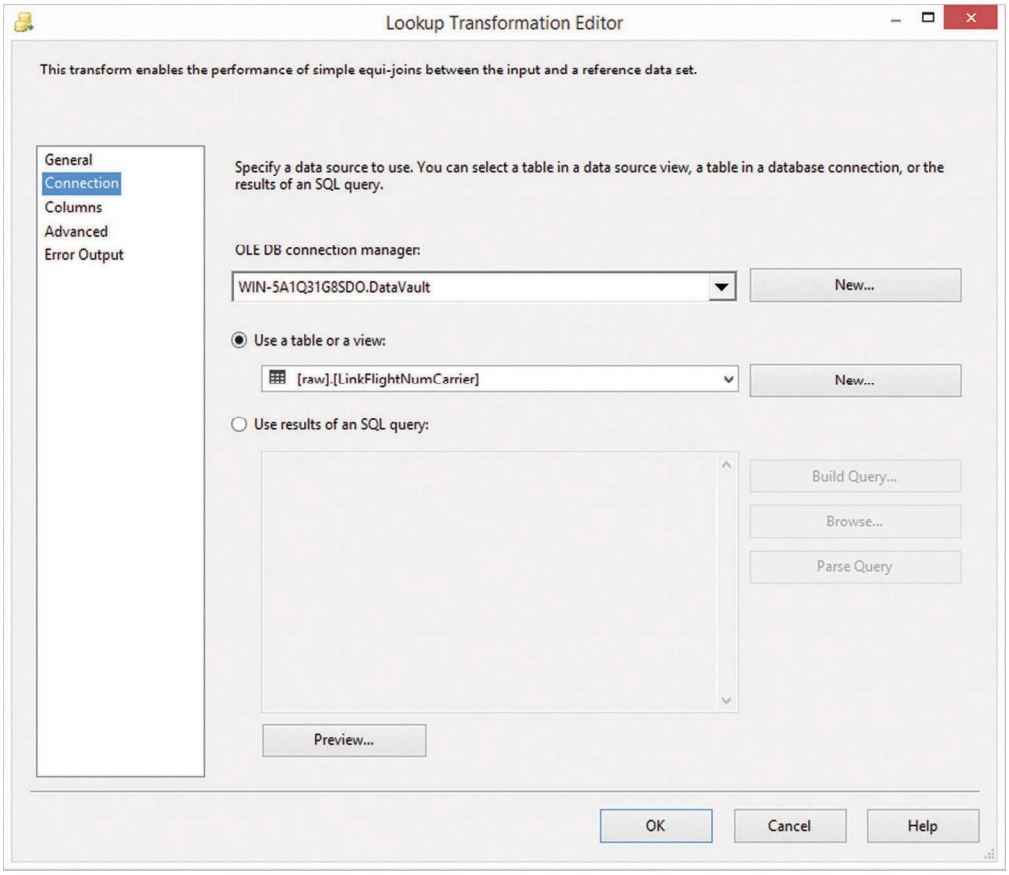

FIGURE 12.23

Setting up the connection in the lookup transformation editor for link tables.

The **lookup no match output** provides the link records from the source table in the staging area that have not been found in the target link table yet. Therefore, select this output and map it to the OLE DB destination input in order to load unknown link records to the target. The other output provides only link records that are already known to the target and will be ignored.

Close the dialog and open the **OLE DB destination editor**, shown in Figure 12.26.

Select the connection manager for the **DataVault** database and select the **LinkFlightNumCarrier** link table in the **raw** schema. Make sure that **keep nulls** and the **table lock** option are checked. **Keep identity** and **check constraints** should be unchecked for the same reasons as in the hub loading process. Consider changing the **rows per batch** and **maximum insert commit size** in order to adjust the SSIS load to your infrastructure.

Switch to the **mappings** page by selecting the entry in the list on the left of the dialog. The page shown in Figure 12.27 is presented.

Make sure that each column from the destination is mapped to a corresponding column in the source. Close the dialog. The final data flow is presented in Figure 12.28.

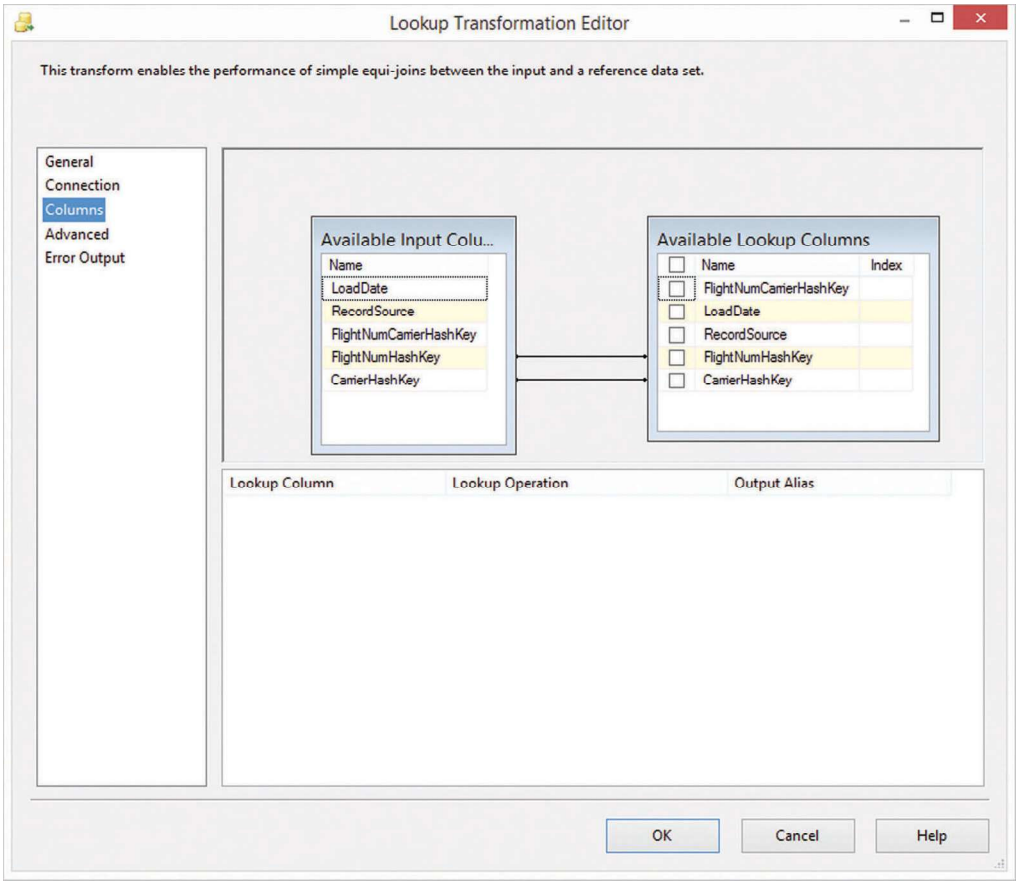

FIGURE 12.24

Configuring the columns of the link lookup.

This data flow might be extended by capturing errors in the error mart. Such an approach would follow the approach outlined in Chapter 10, Metadata Management.

12.1.3 NO-HISTORY LINKS

The loading pattern for no-history links (also known as transactional links) is even simpler than the loading template for standard links (described in section 12.1.2). The only difference is that the lookup in the loading template becomes optional (Figure 12.29).

Because no-history links are typically used for transactions or sensor data, one record needs to be loaded per transaction or event. It is not a distinct list of relationships. The lookup ensures that no duplicate transactions or events are loaded in the case of restarts.

This approach can be implemented in a fully recoverable approach, without the need to delete the partially loaded records in the target link table before restarting this process. While the process runs

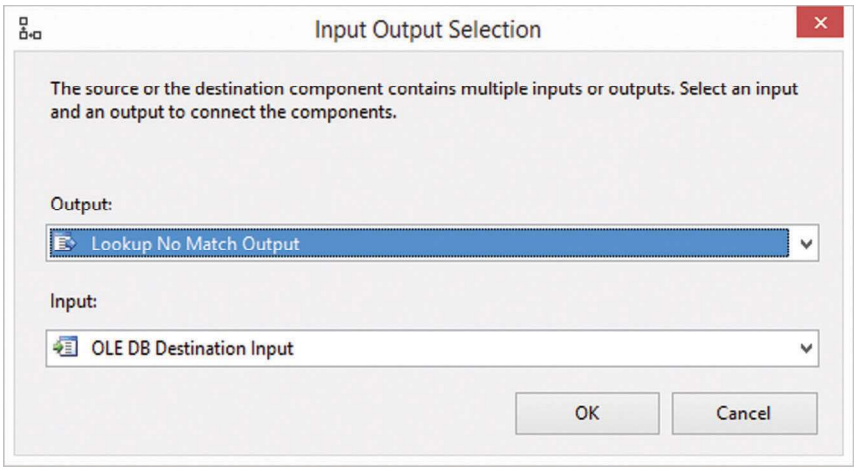

FIGURE 12.25

Input output selection for link loading process.

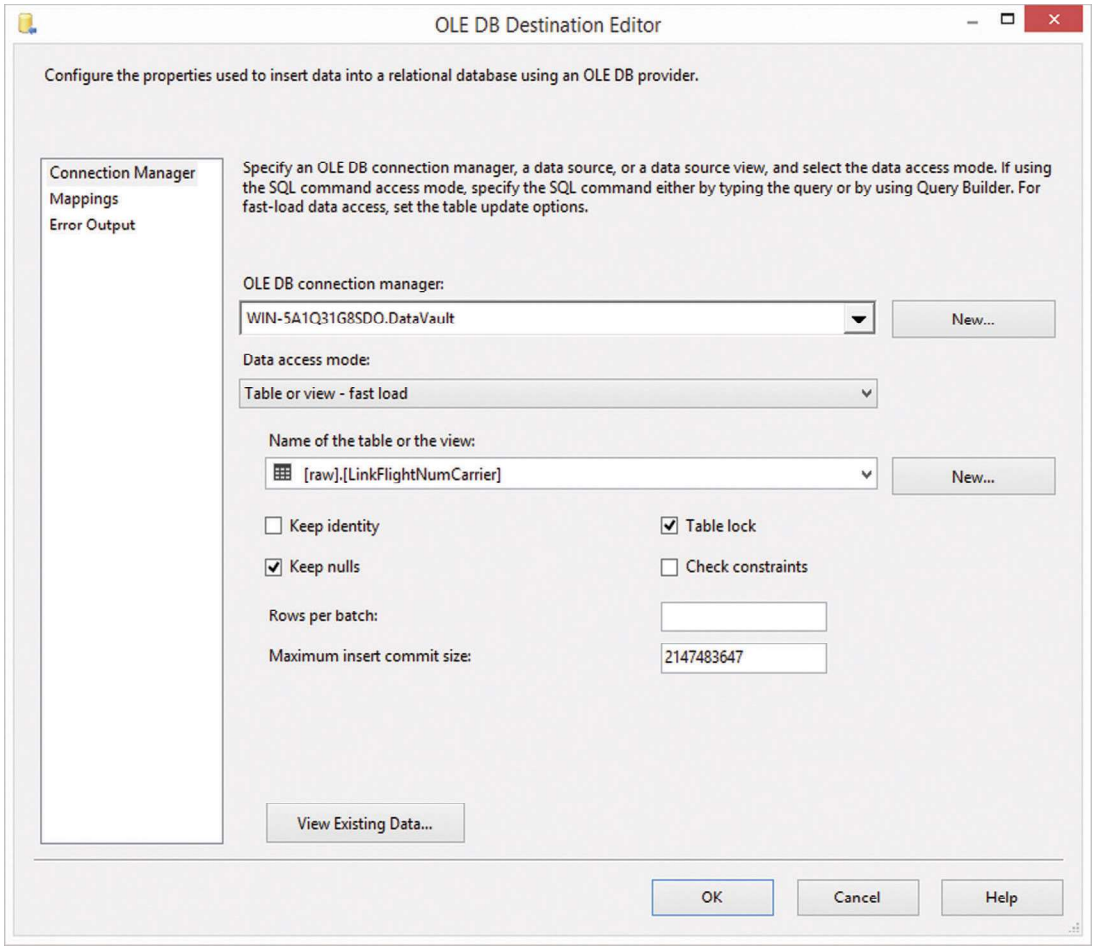

FIGURE 12.26

OLE DB destination editor for the link table destination.

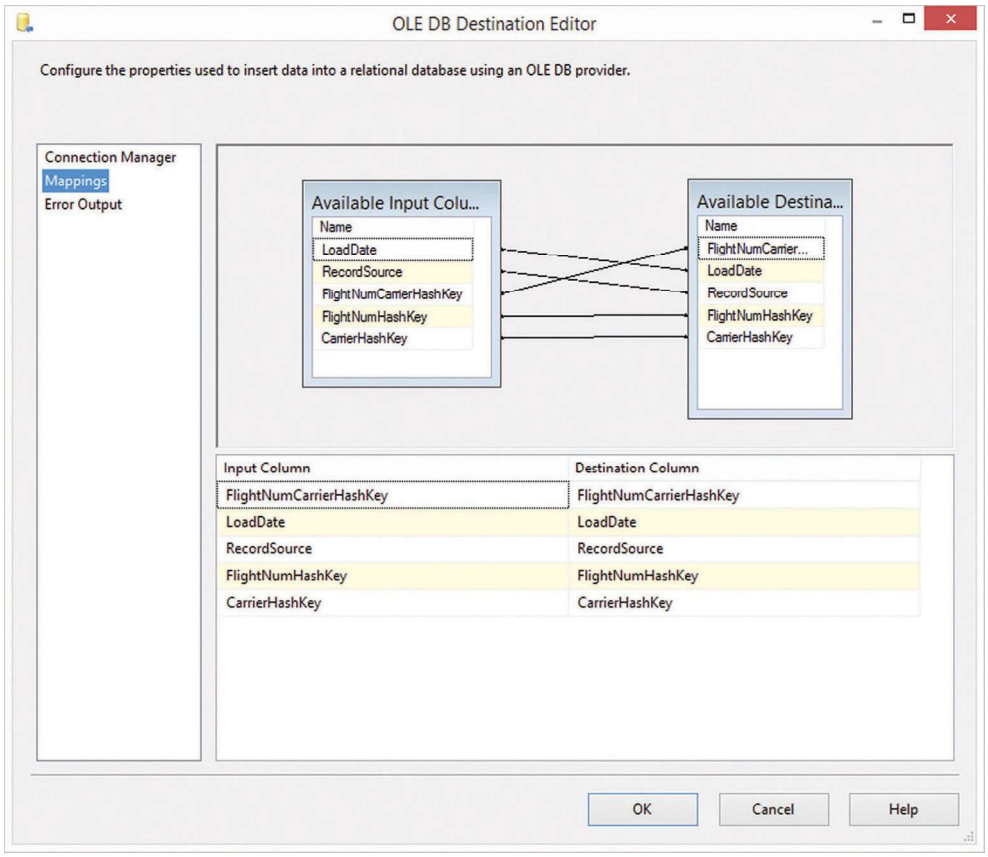

FIGURE 12.27

Mapping the columns in the OLE DB destination editor.

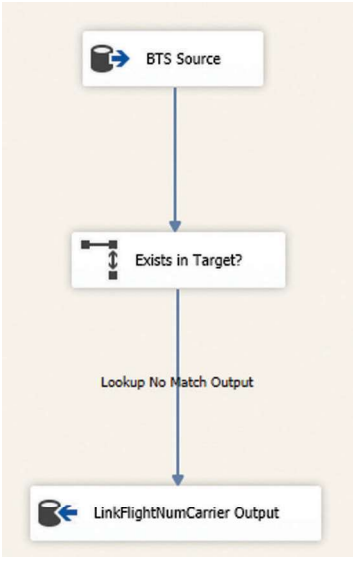

FIGURE 12.28

Data flow to load Data Vault 2.0 links.