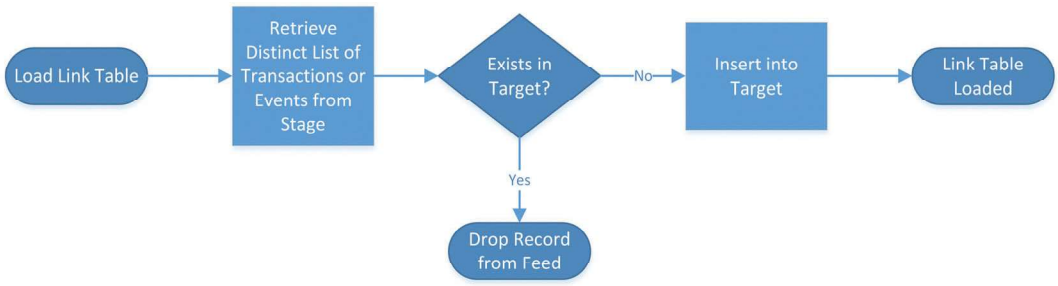

FIGURE 12.29

Template for loading nonhistorized link tables.

slower in the case of restarting the loading process (due to the lookup), it doesn't need any special handling. The ETL process is just restarted and loads the remaining records into the target table.

12.1.3.1 T-SQL Example

From a structural perspective, the no-history link target to be loaded in this section is not different to any other link table in the Raw Data Vault. The only difference is the name, which starts with "TLink" instead of "Link":

```
CREATE TABLE [raw].[TLinkFlight](
       [FlightHashKey] [char](32) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [CarrierHashKey] [char](32) NOT NULL,
       [FlightNumHashKey] [char](32) NOT NULL,
       [TailNumHashKey] [char](32) NOT NULL,
       [OriginHashKey] [char](32) NOT NULL,
       [DestHashKey] [char](32) NOT NULL,
       [FlightDate] [datetime2](7) NOT NULL,
 CONSTRAINT [PK_TLinkFlight] PRIMARY KEY NONCLUSTERED
(
       [FlightHashKey] ASC
) ON [INDEX],
 CONSTRAINT [UK_LinkFlight] UNIQUE NONCLUSTERED
(
       [CarrierHashKey] ASC,
       [FlightNumHashKey] ASC,
       [TailNumHashKey] ASC,
       [OriginHashKey] ASC,
       [DestHashKey] ASC,
       [FlightDate] ASC
) ON [INDEX]
) ON [DATA]
```

The no-history link consists of **FlightHashKey** as an identifying hash key (used in the primary key), a load date and record source and the hash keys of the referenced hubs: **HubFlightNum**, **HubTailNum**, **HubAirport**. The latter was referenced twice and two hash keys are stored: one for the origin airport and the other for the destination airport.

The source data provides no unique event or transaction number: the flight data itself is not a good candidate to be used as a transaction or event number in the non-historized link. But such a number

is required to achieve the proper grain, which fits the grain of the actual flights. Using only the hub references is not enough, because the flight number is reused from one day to the other. The goal is to achieve two goals for the loading process of no-history links:

1. **Same grain as transactions or events:** each transaction or event should have one record in the no-history link table in order to easily source fact tables from this table.
2. **Unique hash key for primary key:** the hash key in a link table is based on the referenced business keys. In order to achieve the first goal, another identifying element is required because, in most cases, the hub references alone have another grain than the transactions themselves (think of reoccurring customers, buying the same product in the same store).

Because no such transaction number is available in the source data, the flight date is added. It is not unique by itself, but fits the purpose of achieving the required grain and producing a unique hash key for the link when the date is included in the hash key calculation. The underlying elements, that is the hash keys of the referenced hubs and the flight date, are included in the alternate key for uniqueness and to speed up (optional) lookups.

Because the flight date was already included when calculating the hash key in the staging area, it is possible to simply load the data from the staging area into the no-history link:

```
INSERT INTO DataVault.[raw].TLinkFlight (
       FlightHashKey,
       LoadDate,
       RecordSource,
       CarrierHashKey,
       FlightNumHashKey,
       TailNumHashKey,
       OriginHashKey,
       DestHashKey,
       FlightDate
)
SELECT
       FlightHashKey,
       LoadDate,
       RecordSource,
       CarrierHashKey,
       FlightNumHashKey,
       TailNumHashKey,
       OriginHashKey,
       DestHashKey,
       FlightDate
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD s
WHERE
       NOT EXISTS (SELECT
                     1
              FROM
                     DataVault.[raw].TLinkFlight l
              WHERE
                     s.CarrierHashKey = l.CarrierHashKey
                     AND s.FlightNumHashKey = l.FlightNumHashKey
                     AND s.TailNumHashKey = l.TailNumHashKey
                     AND s.OriginHashKey = l.OriginHashKey
                     AND s.DestHashKey = l.DestHashKey
                     AND s.FlightDate = l.FlightDate
              )
```

Also, the order in which the data is loaded into the link table doesn't matter, that's why there is no filter for a specific load date. All the data is simply loaded into the target table. To achieve a recoverable process, a lookup is required that checks whether the record to be loaded is already in the target table. The NOT EXISTS condition in the WHERE clause is taken from the statement to load standard link tables. Note that the flight date was included in this condition to make sure that duplicates are filtered out accordingly.

12.1.3.2 SSIS Example

To implement the T-SQL based loading process from the previous section in SSIS, a similar data flow is required to the previous SSIS example for loading Data Vault 2.0 links. Drop an OLE DB source transformation to a new data flow and open the editor (Figure 12.30).

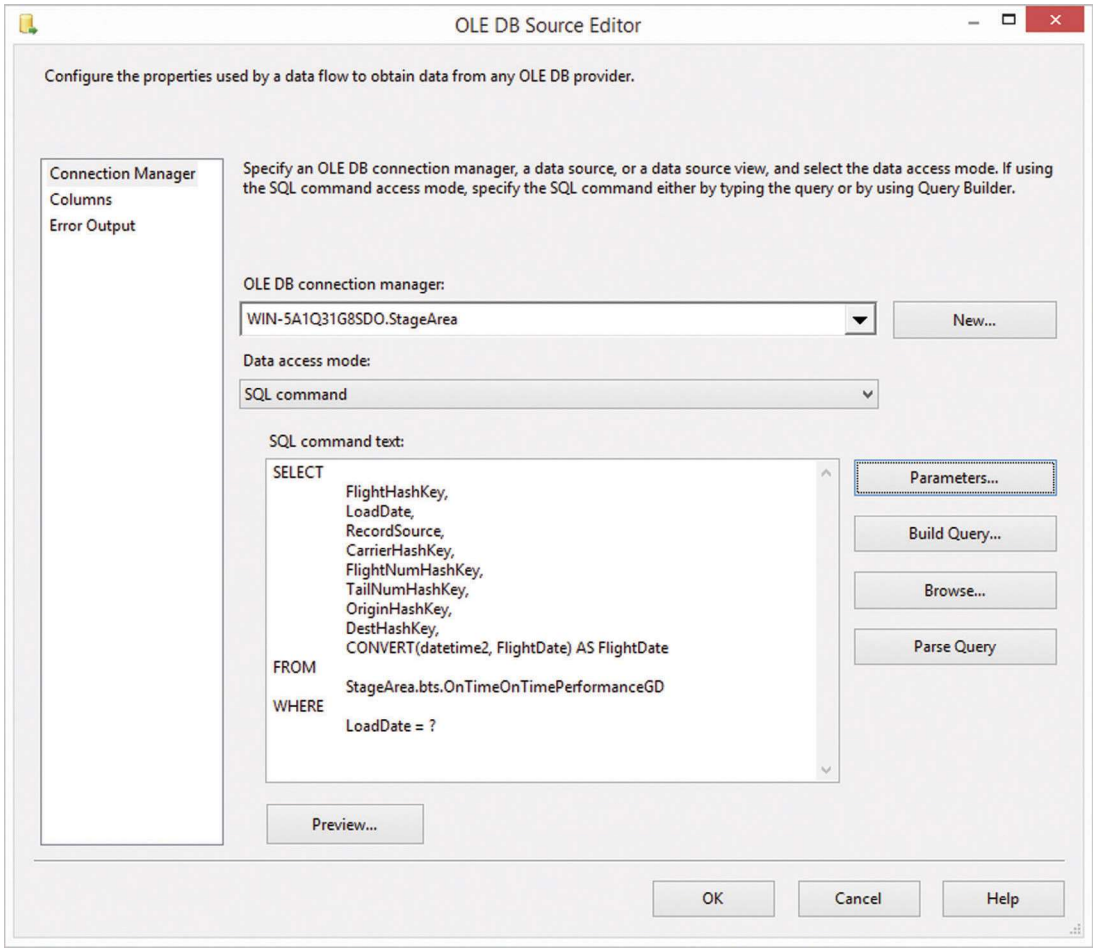

FIGURE 12.30

OLE DB source editor for no-history links.

The OLE DB source transformation loads the data from the source table in the staging area by using the following SQL command text:

```
SELECT
       FlightHashKey,
       LoadDate,
       RecordSource,
       CarrierHashKey,
       FlightNumHashKey,
       TailNumHashKey,
       OriginHashKey,
       DestHashKey,
       CONVERT(datetime2, FlightDate) AS FlightDate
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD
WHERE
       LoadDate = ?
```

Because the **flight date** in the source table is stored as an nvarchar, it is converted into the correct data type on the way out of the staging area and into the Raw Data Vault no-history link table. This is the latest point that it should be done. If possible, it should already be done in the staging area, but in some cases (refer to Chapter 11, Data Extraction), it has to be done on the way out. The **load date** is also included in this statement to process only one batch at a time. As usual, use the **Parameters…** button to map the parameter to a SSIS variable (Figure 12.31).

Once the load date variable has been mapped to the parameter, close the dialog and the OLE DB source editor. Insert a lookup transformation into the data flow and connect it to the output of the OLE DB source. Open the lookup transformation editor, shown in Figure 12.32.

Again, make sure that **redirect rows to no match output** has been selected in the combo box on the first page. After that, switch to the next page by selecting **connection** from the list on the left of the dialog. The page is used to set up the target transaction link as the lookup table (Figure 12.33).

Make sure that **TLinkFlight** in the **raw** schema of the **DataVault** database is used for the connection and switch to the **columns** page of the dialog (Figure 12.34).

Similar to the standard link, the hash keys of the referenced hubs are used to perform the lookup. However, because the hash keys alone are not enough to correctly identify the link, the **flight date** is added into the equi-join of the lookup as well. No columns from the lookup table are returned because we're only interested in finding out if the no-history link already exists in the target or not.

Close the dialog and add an OLE DB destination to the data flow. Connect the output path from the lookup transformation to the destination transformation. Because unknown links are redirected into another output, the dialog shown in Figure 12.35 appears.

Select the **lookup no match output** because it contains all the records where no corresponding link record has been found in the target link table.

Close the dialog and open the OLE DB destination editor, as shown in Figure 12.36.

Select the target table, **TLinkFlight** in the **raw** schema of the **DataVault** database, and make sure that **keep nulls** and **table lock** are checked. The other options (**keep identity** and **check constraints**) should not be checked. **Rows per batch** and the **maximum insert commit size** should be adjusted to your data warehouse infrastructure. Check that the column mapping of the data flow to the destination table is correct by switching to the **mappings** page of the dialog (Figure 12.37).

Each destination column should have a corresponding input column. Select **OK** to confirm the changes to the dialog.

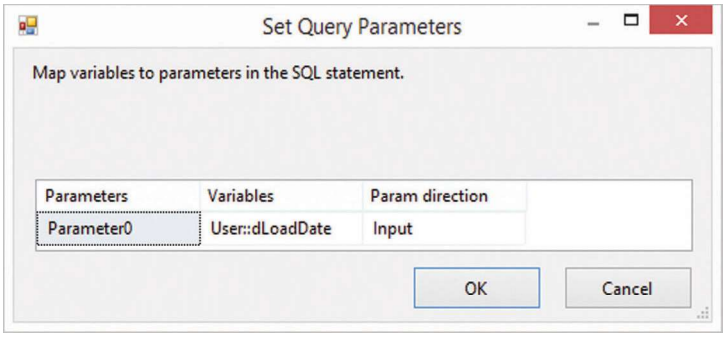

FIGURE 12.31

Set query parameter for nonhistorized link.

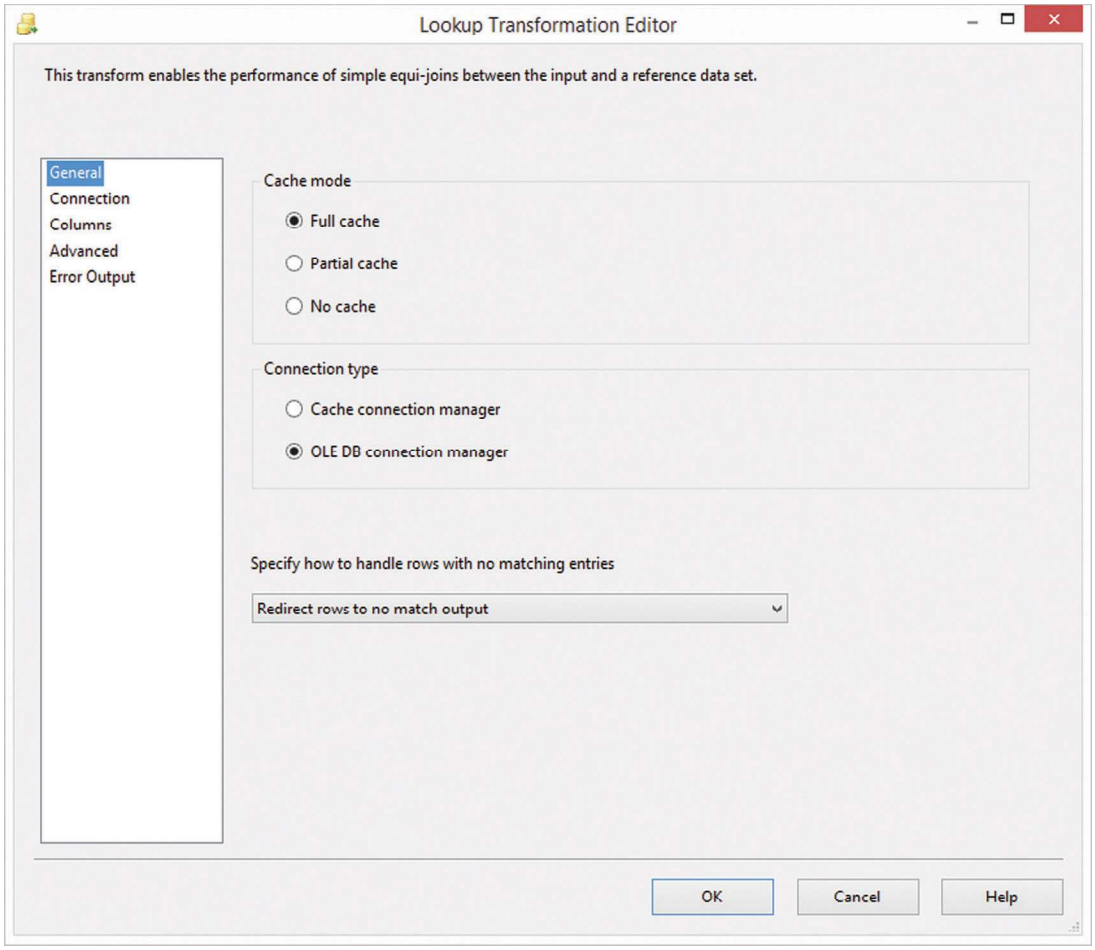

FIGURE 12.32

Lookup transformation editor for no-history links.

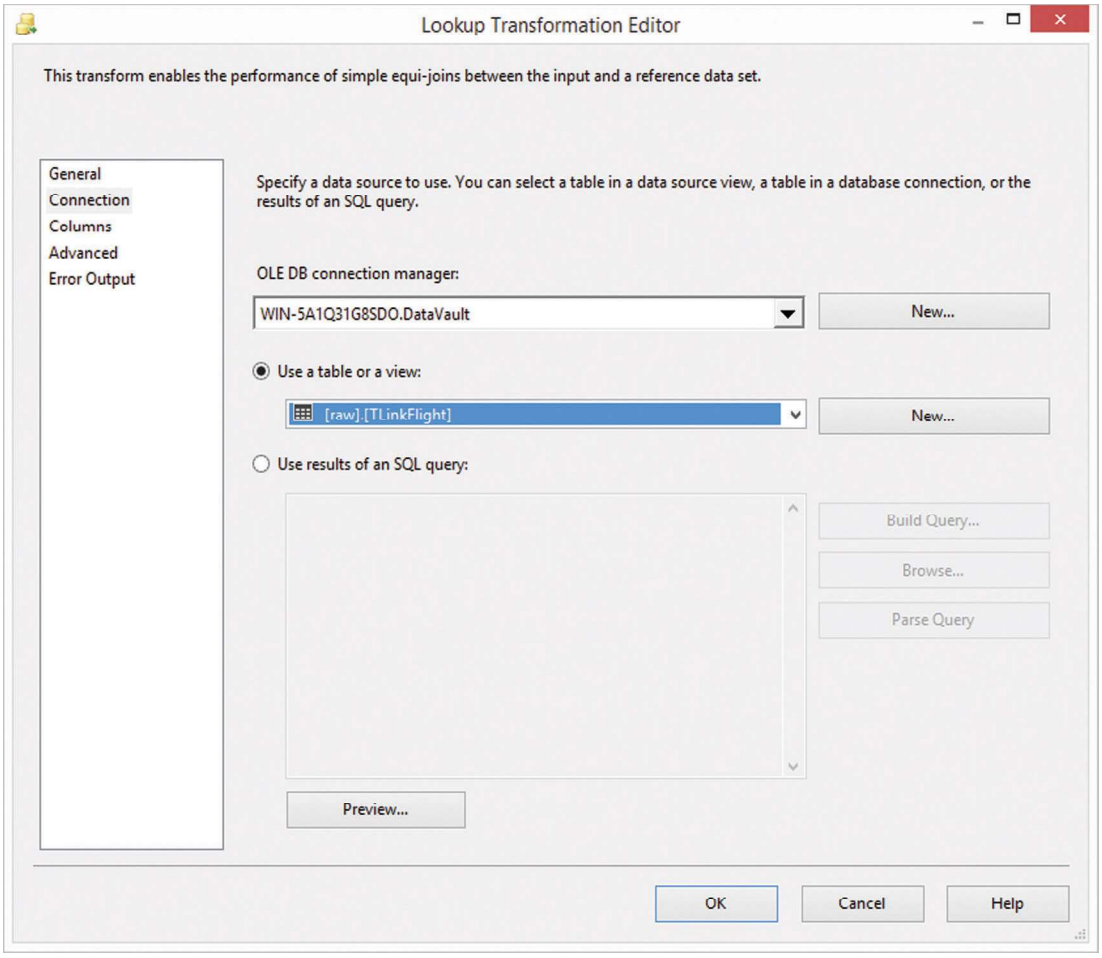

FIGURE 12.33

Setting up the connection in the lookup transformation editor for no-history links.

Figure 12.38 presents the final SSIS data flow for loading no-history links.

In addition to this link table, no-history links often have nonhistorized descriptive data stored in no-history satellites. Their loading process is covered in one of the coming sections on loading satellite entities.

12.1.4 SATELLITES

While the loading templates for hubs and links, even for special cases such as no-history links, are fairly simple, satellites are a little more complicated. However, the loading template for satellites follows the same design principles as discussed in the introduction to this chapter, thus becoming a fairly simple template as well.

The goal of the satellite loading process is to source the data from the staging area, find changes in the data and load these changes (and only these changes) into the target satellite. The default loading template for satellites is presented in Figure 12.39.

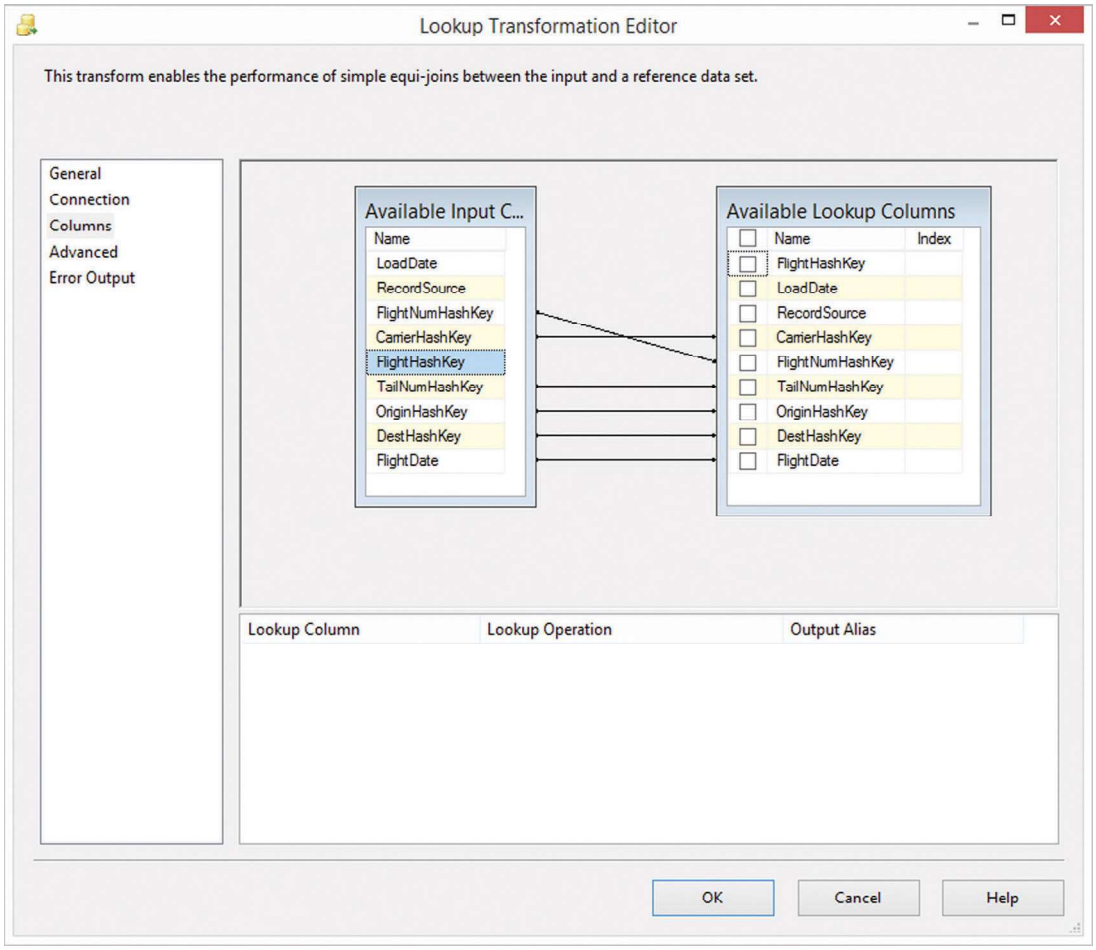

FIGURE 12.34

Configuring lookup columns for the no-history link.

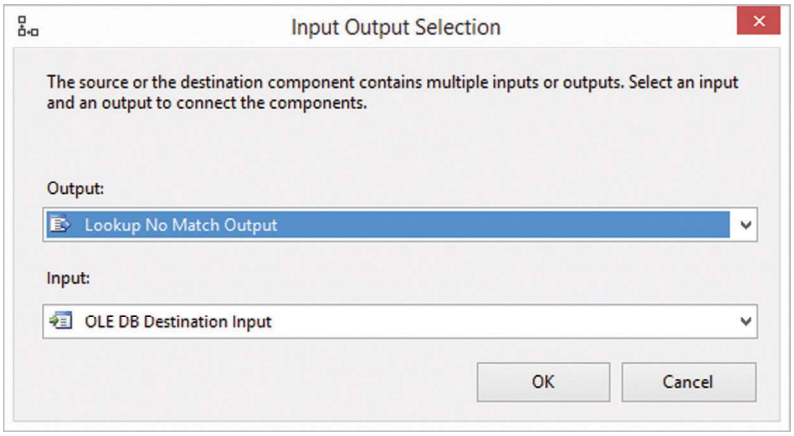

FIGURE 12.35

Input output selection for no-history links.

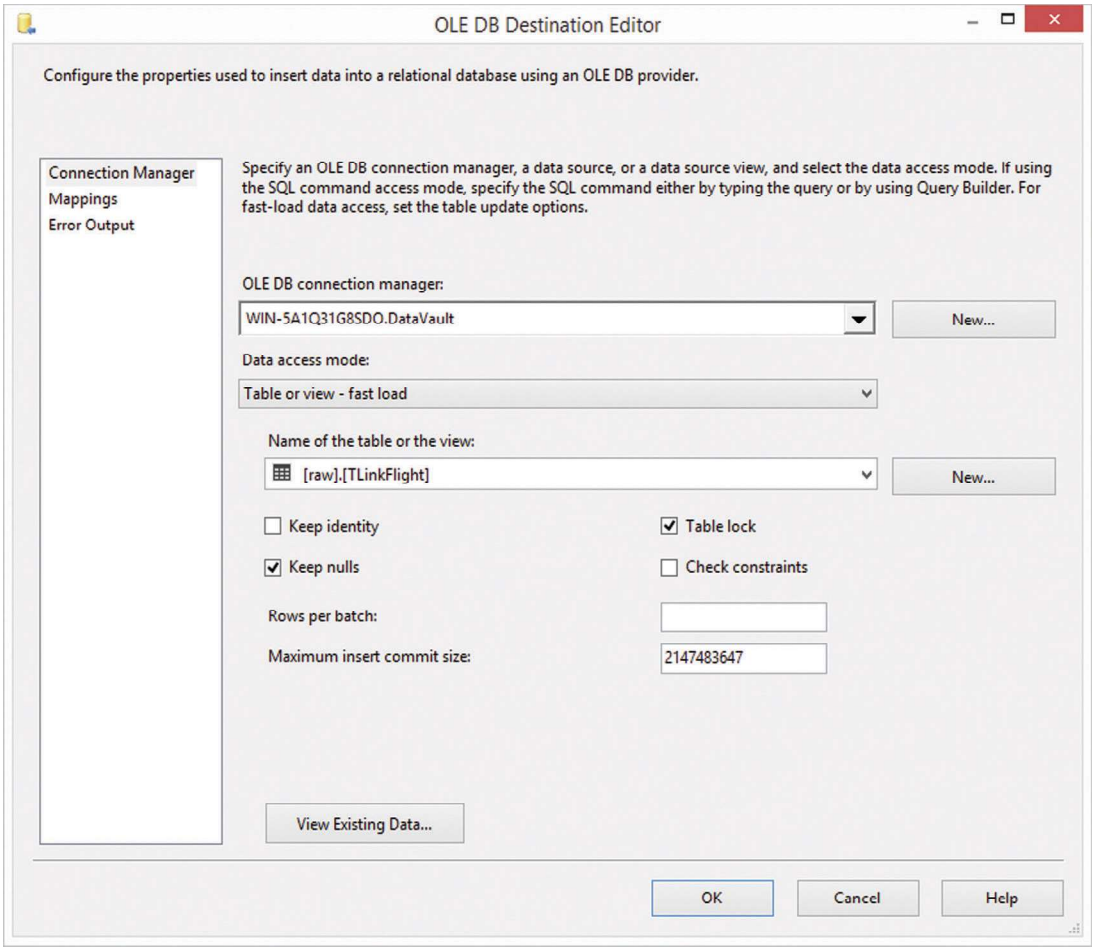

FIGURE 12.36

OLE DB destination editor of the no-history link.

The first step is to retrieve a distinct list of records from the staging table that might be changes. This step should omit **true** duplicates in the source because the satellite wouldn't capture them anyway. By doing so, the data flow reduces the amount of data early in the process, following the design principles for the loading templates.

Note that we define true duplicates as actual duplicates that are added to the staged data due to technical issues and provide no value to the business. We assume that these records should not be loaded into the Raw Data Vault and are filtered out in the loading process. If duplicates should be loaded, consider using a multi-active satellite for this purpose (refer to Chapter 5, Intermediate Data Vault Modeling, for more details).

Once the data (which contains both the descriptive data as well as the hash key of the parent hub or link table) has been sourced from the staging area, the latest record from the target satellite table that

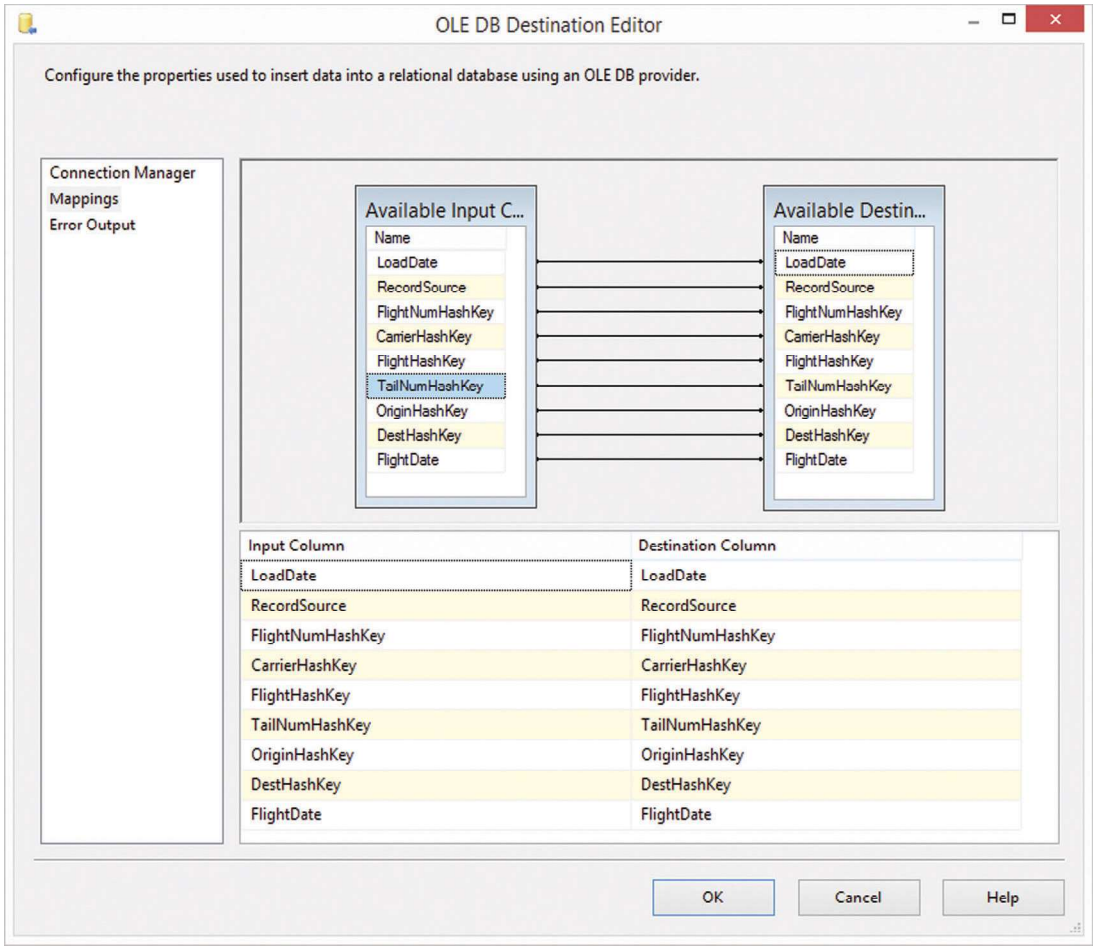

FIGURE 12.37

Mapping columns in the OLE DB destination editor of the no-history link.

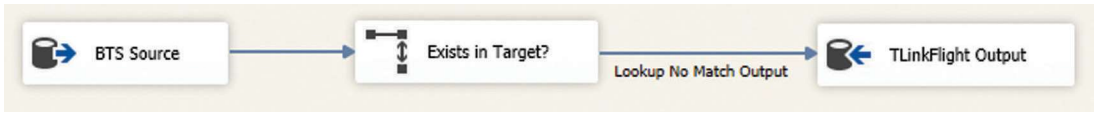

FIGURE 12.38

Complete SSIS data flow for loading no-history links.

matches the hash key is retrieved. This step is easy, because the latest record should have the latest load date of all the records for this hash key and a NULL or maximum load end date.

Both records, the record from the staging area and the latest satellite record obtained in the previous step, are compared column-wise. If all columns match, that means that no change has been detected, and the record is dropped from the feed because the satellite should not capture it. Remember that Data Vault satellites capture only deltas (new or changed records). If a record has not changed, it is not loaded into the satellite table.

If there is a change in at least one of the columns, the record is added to the target satellite table. Note that, in most cases, the change detection relies on all columns. However, there might be cases where some of the columns are ignored during column comparison. This would result in undetected changes. If all other columns were unmodified, no record would be added. Therefore, such a limited column comparison should be used only in specific and rare cases. Another thing to consider is if the column comparison should be case sensitive or case insensitive. Depending on the source data, both options make sense and are frequently used.

The major issue with the column comparison is that satellites sometimes contain a large number of columns that are required to be compared. This comparison can take time, especially if the number of records in the stage area that might contain changes to be captured is high as well. In order to speed up the performance of the loading template presented in Figure 12.39, hash diffs can be used. The hash diffs have been introduced as optional attributes to satellites in Chapter 4 and explained in great detail in the previous chapter.

Once the hash diff has been added to the satellite entity, it can be used to compare the input record with the target record with a comparison on one column only. When loading the Data Vault satellite, both required hash diffs from the source and the target are already calculated. The source data has already been hashed in the staging area, while the target satellites have the hash diffs included for each row by design.

Because of this preparatory work, the modified loading process doesn't require any hashing and is merely a simplified version of the standard loading template for satellites (Figure 12.40).

The difference between both loading patterns is only in the comparison step that either drops the record from the feed or loads it into the target table. This comparison is reduced to compare the hash diffs instead of all individual columns. It also removes the need to account for NULL values as this has been done in the hash diff calculation already by replacing the NULL values with empty strings in conjunction with delimiters. Other than the changed comparison, the template is the same as in Figure 12.39.

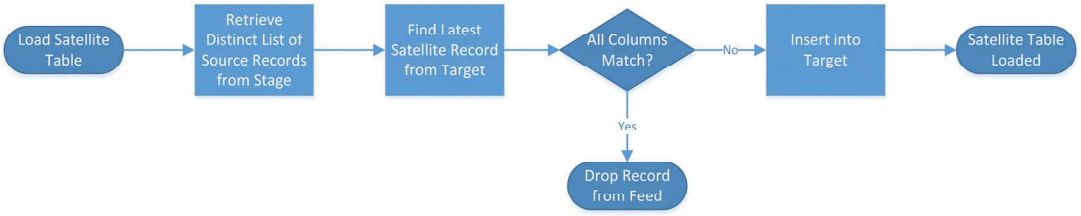

FIGURE 12.39

Template for loading satellite tables.

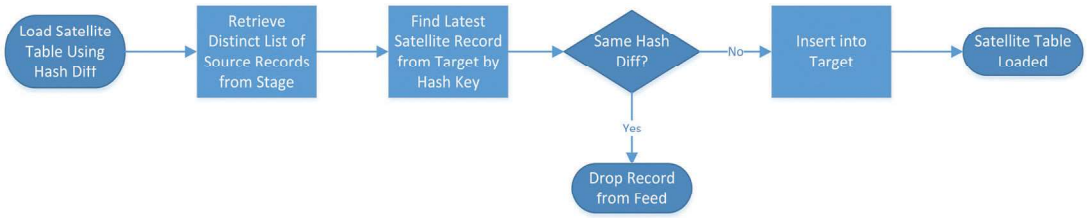

FIGURE 12.40

Template for loading satellite tables using hash diffs.

12.1.4.1 T-SQL Example

This section loads the descriptive data for the airports. However, there are two different sets of columns, each describing either the airport of origin or airport of destination. This data could be merged into one satellite, but aligning it might require additional business logic: what happens if data from both sets of columns contradict each other, e.g., using different descriptive data for the same airport and the same load date time? The load processes of the Raw Data Vault should not depend on any conditional business logic. Therefore, both sets of descriptive data are distributed to different satellites, each hanging off **HubAirport**. The first satellite captures the descriptive data for the originating airport and is created using the following DDL statement:

```
CREATE TABLE [raw].[SatOriginAirport](
       [AirportHashKey] [char](32) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [LoadEndDate] [datetime2](7) NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [OriginCityName] [nvarchar](100) NOT NULL,
       [OriginState] [nvarchar](2) NOT NULL,
       [OriginStateName] [nvarchar](100) NOT NULL,
       [OriginCityMarketID] [int] NOT NULL,
       [OriginStateFips] [smallint] NOT NULL,
       [OriginWac] [smallint] NOT NULL,
 CONSTRAINT [PK_SatOriginAirport] PRIMARY KEY NONCLUSTERED
(
       [AirportHashKey] ASC,
       [LoadDate] ASC
) ON [INDEX]
) ON [DATA]
```

The satellite contains the parent hash key and the load date in its primary key. The load end date is NULLable and indicates if and when the record has been replaced by a newer version. The descriptive "payload" of the satellite is defined by the columns **OriginCityName**, **OriginState**, **OriginStateName**, **OriginCityMarketID**, **OriginStateFips** and **OriginWac**. Because the primary key contains a randomly distributed hash key, a nonclustered index should be used.

Note that this satellite does not use the hash diff value calculated in the staging area for demonstrative purposes. The next satellite, also presented in this section, uses the hash diff value.

It is easily possible to implement the template shown in Figure 12.39 in T-SQL only:

```
INSERT INTO DataVault.[raw].SatOriginAirport (
       AirportHashKey,
       LoadDate,
       LoadEndDate,
       RecordSource,
       OriginCityName,
       OriginState,
       OriginStateName,
       OriginCityMarketID,
       OriginStateFips,
       OriginWac
)
SELECT DISTINCT
       stg.OriginHashKey,
       stg.LoadDate,
       NULL,
       stg.RecordSource,
       stg.OriginCityName,
       stg.OriginState,
       stg.OriginStateName,
       stg.OriginCityMarketID,
       stg.OriginStateFips,
       stg.OriginWac
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD stg
LEFT OUTER JOIN
       DataVault.[raw].SatOriginAirport sat
       ON (stg.OriginHashKey = sat.AirportHashKey AND sat.LoadEndDate IS NULL)
WHERE
       (ISNULL(stg.OriginCityName, '') != ISNULL(sat.OriginCityName, '')
       OR ISNULL(stg.OriginState, '') != ISNULL(sat.OriginState, '')
       OR ISNULL(stg.OriginStateName, '') != ISNULL(sat.OriginStateName, '')
       OR ISNULL(stg.OriginCityMarketID, 0) != ISNULL(sat.OriginCityMarketID, 0)
       OR ISNULL(stg.OriginStateFips, 0) != ISNULL(sat.OriginStateFips, 0)
       OR ISNULL(stg.OriginWac, 0) != ISNULL(sat.OriginWac, 0))
       AND stg.LoadDate = '1995-10-14 00:00:00.000'
```

This statement selects the data, including the parent hash key, the load date, the record source and the descriptive data from the source table in the staging area. The select is a DISTINCT operation because the airport is used in multiple flights (most airports serve multiple flights per day). If the DISTINCT option is left out, duplicate entries would be sourced and the subsequent insert operation would violate the primary key constraint of the target satellite. The load end date is explicitly set to NULL for demonstration purposes. As an alternative, it could also be removed from the list of columns and set implicitly. Because only changed data should be sourced into the target satellite, each column from the source stage table needs to be compared with its corresponding column in the target satellite table including a check for NULL values in the descriptive columns. This check could also be implemented using the ANSI-SQL standard function COALESCE. This column comparison is implemented in the WHERE clause of the statement. It requires that the target satellite be joined into the statement in order to have the current target values available. For this reason, a LEFT OUTER JOIN is used to join the data into the source dataset. The join

condition compares both hash keys of the parent and requires that the record should be active. The latter is checked by selecting only records from the target with a load end date of NULL. When comparing the records, only new and active records with a load end date of NULL are interesting.

The additional filter on the source load date ensures that only one batch is loaded. This is very important in the loading process of satellites because the delta detection depends on the fact that only one batch is evaluated at the same time. Changing this is possible but complicates the process. Also, it is important to run the batches in the staging area in the right order, to make sure that the final satellite has captured the changes as desired.

The second satellite, **SatDestAirport**, which captures the descriptive data for the flight's destination airport, uses the hash diff value to improve the performance of the column comparison. Therefore, the hash diff column was added to a structure similar to the **SatOriginAirport**:

```
CREATE TABLE [raw].[SatDestAirport](
       [AirportHashKey] [char](32) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [LoadEndDate] [datetime2](7) NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [HashDiff] [char](32) NOT NULL,
       [DestCityName] [nvarchar](100) NOT NULL,
       [DestState] [nvarchar](2) NOT NULL,
       [DestStateName] [nvarchar](100) NOT NULL,
       [DestCityMarketID] [int] NOT NULL,
       [DestStateFips] [smallint] NOT NULL,
       [DestWac] [smallint] NOT NULL,
 CONSTRAINT [PK_SatDestAirport] PRIMARY KEY NONCLUSTERED
(
       [AirportHashKey] ASC,
       [LoadDate] ASC
) ON [INDEX]
) ON [DATA]
```

The technical metadata of the satellite, including the parent hash key **AirportHashKey**, the load and end date and the record source, are exactly the same as in the previous satellite table. In addition, the hash diff column was added and the names of the descriptive columns are slightly different, due to different names in the source table.

To load the data from the source table in the staging area into the target satellite table, the following statement is used:

```
INSERT INTO DataVault.[raw].SatDestAirport (
       AirportHashKey,
       LoadDate,
       LoadEndDate,
       RecordSource,
       HashDiff,
       DestCityName,
       DestState,
       DestStateName,
       DestCityMarketID,
       DestStateFips,
       DestWac
)
```

```
SELECT DISTINCT
       stg.DestHashKey,
       stg.LoadDate,
       NULL,
       stg.RecordSource,
       stg.DestAirportHashDiff,
       stg.DestCityName,
       stg.DestState,
       stg.DestStateName,
       stg.DestCityMarketID,
       stg.DestStateFips,
       stg.DestWac
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD stg
LEFT OUTER JOIN
       DataVault.[raw].SatDestAirport sat
       ON (stg.DestHashKey = sat.AirportHashKey AND sat.LoadEndDate IS NULL)
WHERE
       (sat.HashDiff IS NULL OR stg.DestAirportHashDiff != sat.HashDiff)
       AND stg.LoadDate = '1995-10-26 00:00:00.000'
```

The only difference from the previous INSERT statement, apart from the fact that it loads the table **SatDestAirport** instead of **SatOriginAirport** (thus sourcing different columns from the staging table), is that the column compare was simplified: only one column is compared instead of all descriptive columns. If the hash diff in the target is different from the one in the stage area, or if there is no record in the target satellite that fits to the hash key (which leads to a hash diff of NULL), the record is loaded into the target.

Both INSERT statements rely on an OUTER JOIN in order to join the data into one result set. This is required in order to compare the incoming data with the existing data or produce NULL values if there is no satellite entry with the same parent hash key as the incoming record. If another join type is used, the column compare would not compare a wrong record or data would be lost on the way from the staging table to the Raw Data Vault. If the performance of the loading process is too slow due to the OUTER JOIN, the best approach is to divide both datasets (the new and the changed records from the staging area) and process them separately. This is covered in section 12.1.6.

The descriptive data in both satellites can be combined later in the Business Vault, for example using PIT tables or computed satellites. These approaches are covered in Chapter 14, Loading the Dimensional Information Mart.

12.1.4.2 SSIS Example

It is also possible to implement the loading template for satellites using SSIS. This approach also requires the load date variable in the SSIS package to ensure that only one batch is loaded into the Raw Data Vault during a single SSIS execution.

The first step is to set up the source components in the SSIS data flow. Unlike the SSIS processes for hubs and links, the SSIS data flow presented in this example uses two different source components from different layers of the data warehouse:

1. **The source staging table:** this table provides the new and changed records along with unchanged records.
2. **The target satellite table:** this table provides the current version of the data in the target.

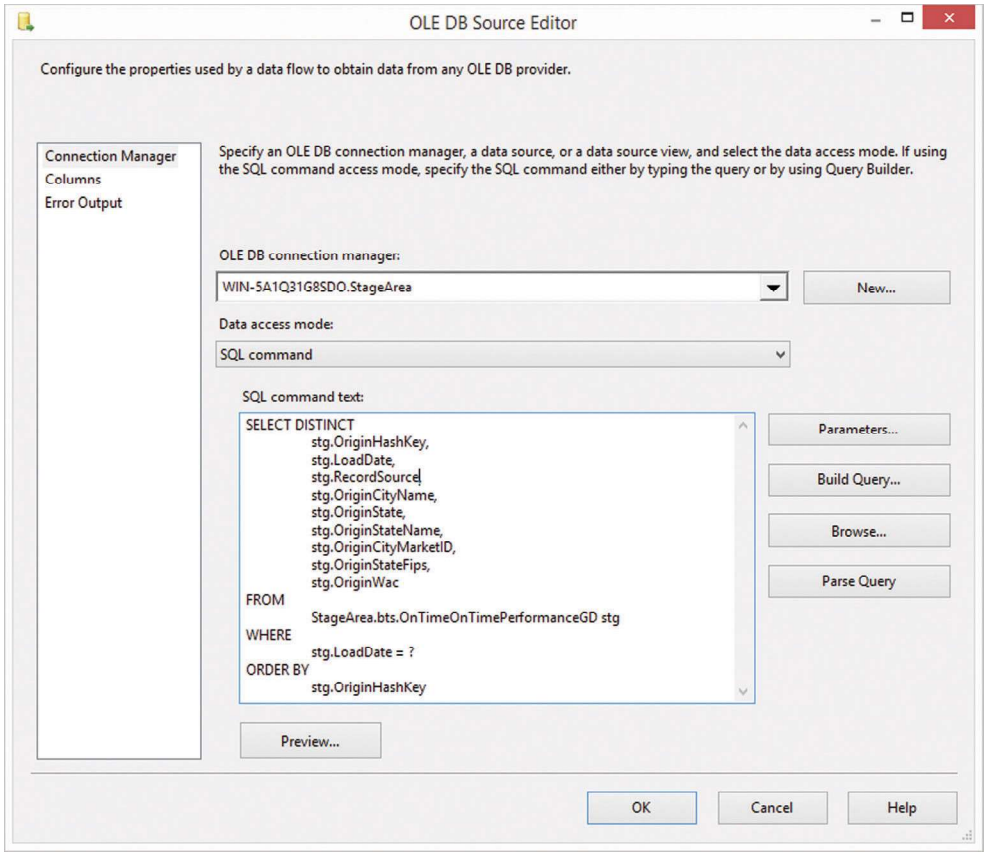

FIGURE 12.41

Set up the source component for the staging area data source.

The data from both sources will be merged in the data flow and compared during column comparison. Essentially, this approach implements a JOIN operation instead of a lookup (sub-query in T-SQL) for performance reasons. The first source component, an **OLE DB Source**, is set up using a SQL command, as Figure 12.41 shows.

The following SQL statement is used as SQL command text:

```
SELECT DISTINCT
       stg.OriginHashKey,
       stg.LoadDate,
       stg.RecordSource,
       stg.OriginCityName,
       stg.OriginState,
       stg.OriginStateName,
       stg.OriginCityMarketID,
       stg.OriginStateFips,
       stg.OriginWac
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD stg
WHERE
       stg.LoadDate = ?
ORDER BY
       stg.OriginHashKey
```

There are two lines noteworthy: first, the ORDER BY clause is required for the merge operation of both data sources because it improves performance if the key that is used for merging the data streams already sorts both data flows. In this case, this will be the hash key. The second interesting line is the use of 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 shown in Figure 12.42 is presented.

Select the variable 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. However, the component is not completely set up yet. While the incoming data is ordered by the hash key to optimize merging the data flows, this setting has to be indicated to the output columns of the source component in addition.

Open the advanced editor from the context menu of the source component (Figure 12.43).

Select the **OLE DB Source Output** node in the tree view from the left. Set the **IsSorted** property of the output to true. The last setting is to indicate the actual column that was used for sorting. Select the column from the **Output Columns** folder in the tree view, as shown in Figure 12.44.

Set the **SortKeyPosition** property to the column number of the hash key in the source query. Because the hash key is the first column in the SQL statement used in Figure 12.41, the position value 1 is set.

This completes the setup of the staging table source. The next dialog shows the setup of the source component that sources the data from the target satellite, which is required to compare the incoming values with the existing values (Figure 12.45).

The following SQL statement is used as the SQL command text:

```
SELECT
       AirportHashKey,
       OriginCityName,
       OriginState,
       OriginStateName,
       OriginCityMarketID,
       OriginStateFips,
       OriginWac
FROM
       [raw].[SatOriginAirport]
WHERE
       LoadEndDate IS NULL
ORDER BY
       AirportHashKey
```

The statement loads all records from the target satellite, which are currently active (having a **load date** of NULL). The data is also ordered to enable merging the data flows in the next step. Because the data is ordered, this has to be indicated to the SSIS engine using the same approach by setting the **IsSorted** property of the output and the **SortKeyPosition** of the hash key output column as described in Figure 12.43 and Figure 12.44.

Once the data from both sources is available in the data flow, the next step is to merge both data streams in order to be able to compare the values from both the staging area and the target satellite. Drag a **Merge Join Transformation** to the data flow canvas and open the editor (Figure 12.46).

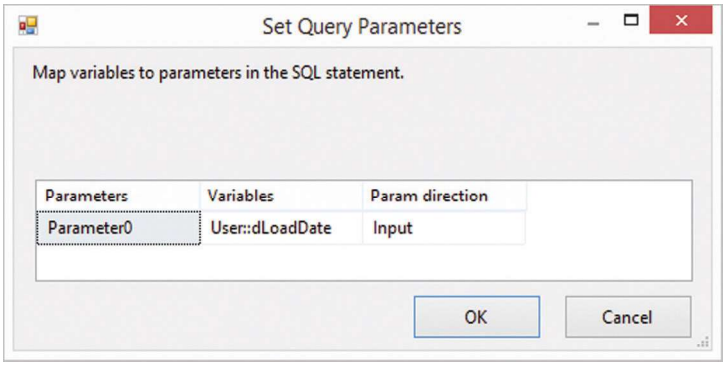

FIGURE 12.42

Set query parameters dialog.

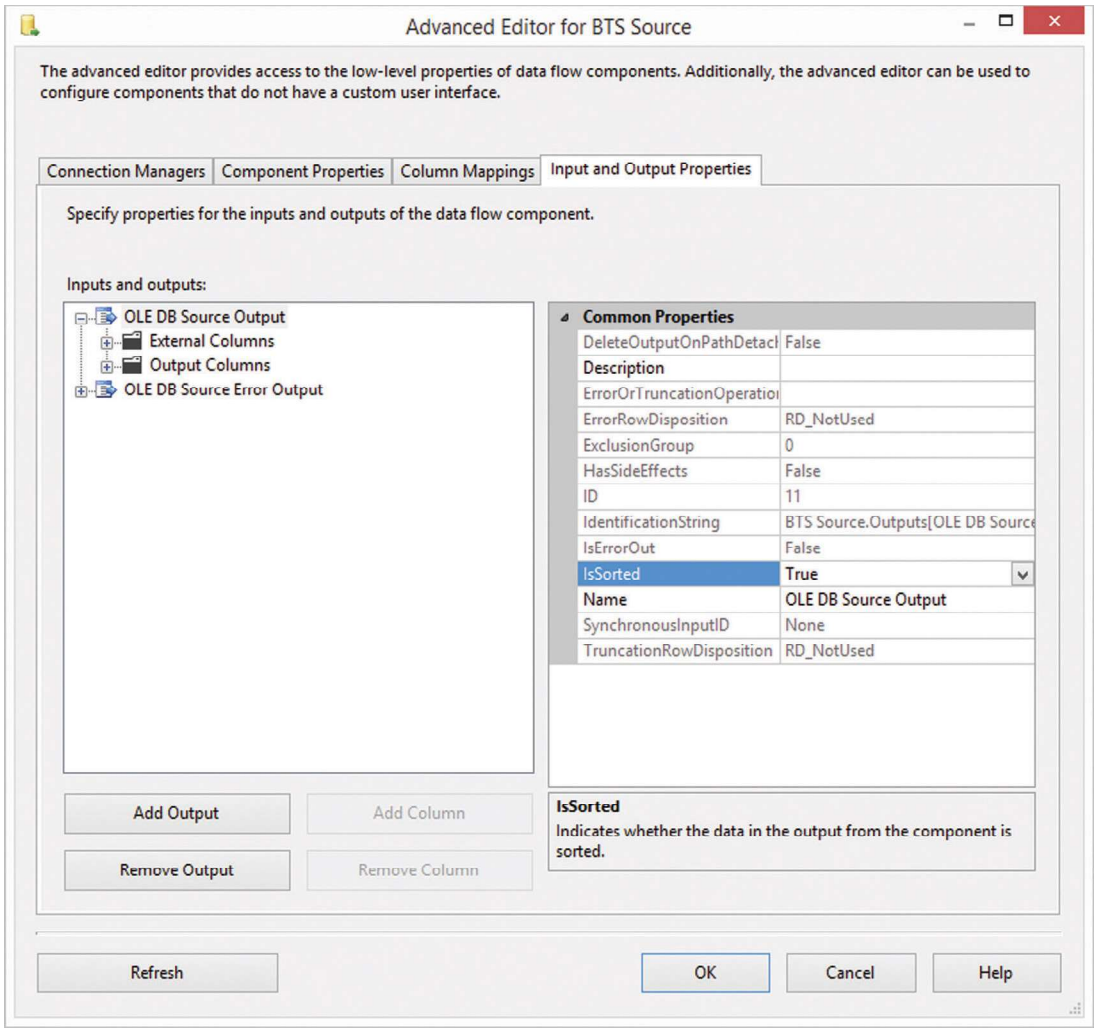

FIGURE 12.43

Advanced editor of source component.

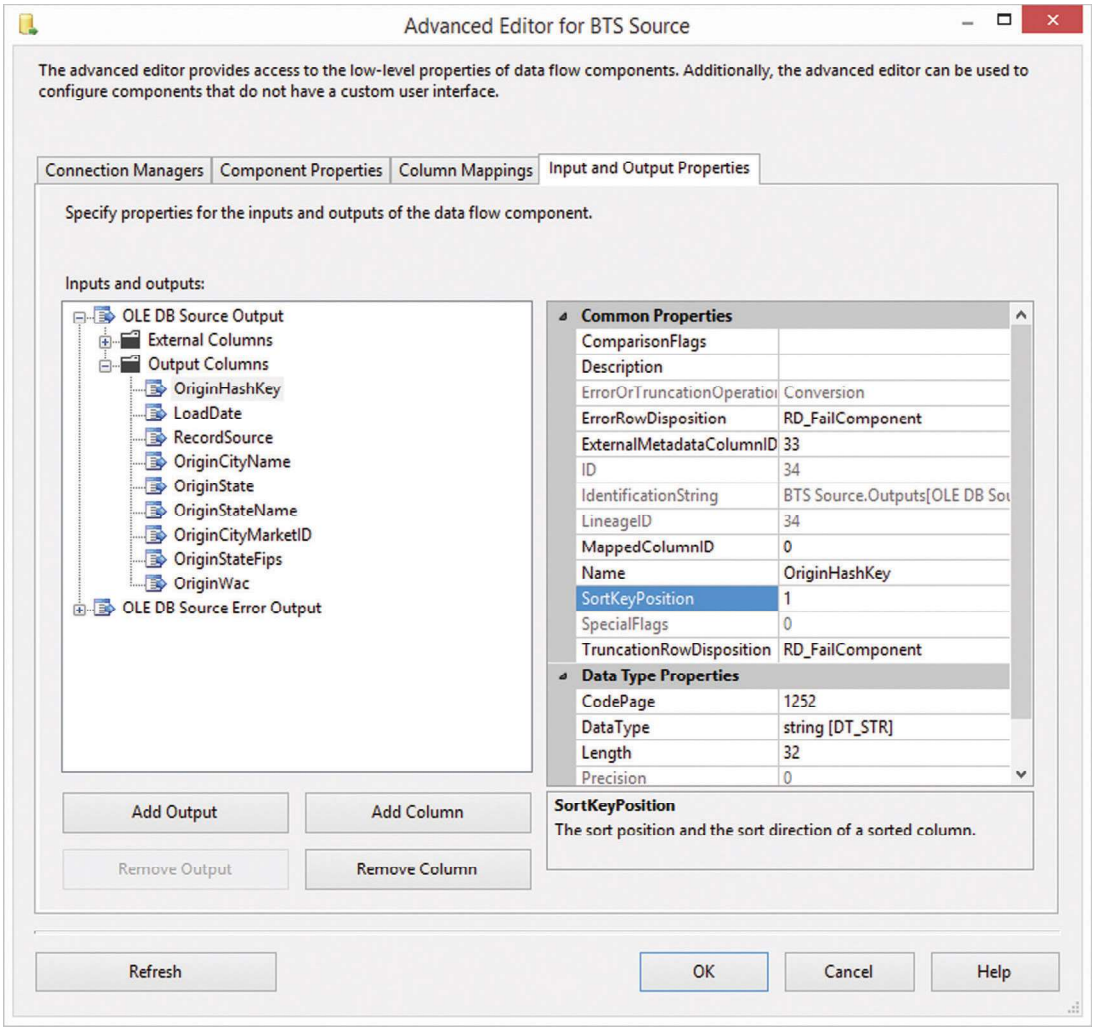

FIGURE 12.44

Setting the SortKeyPosition in the advanced editor.

There are two tasks to complete in this dialog: first, the columns that are used for the merge operation have to be connected. In this case, this is the hash key from each data stream because the streams are merged on this value. Drag the **OriginHashkey** column over the **AirportHashKey** in order to connect both columns. Also, make sure that the **join type** is set to **left outer join**. The second task is to select the columns that should be included in the merged data stream. This should include all columns from the staging area and the descriptive data from the target satellite because all of this data is required for either the column compare or the loading of the target satellite table. Because the descriptive columns are named the same in both the source and the target, rename one or both sides as in Figure 12.46.

Close the dialog and drag a conditional split transformation to the canvas of the data flow. This component is used to filter the records from the staging area source that don't represent a change from the data that is already in the target satellite (Figure 12.47).

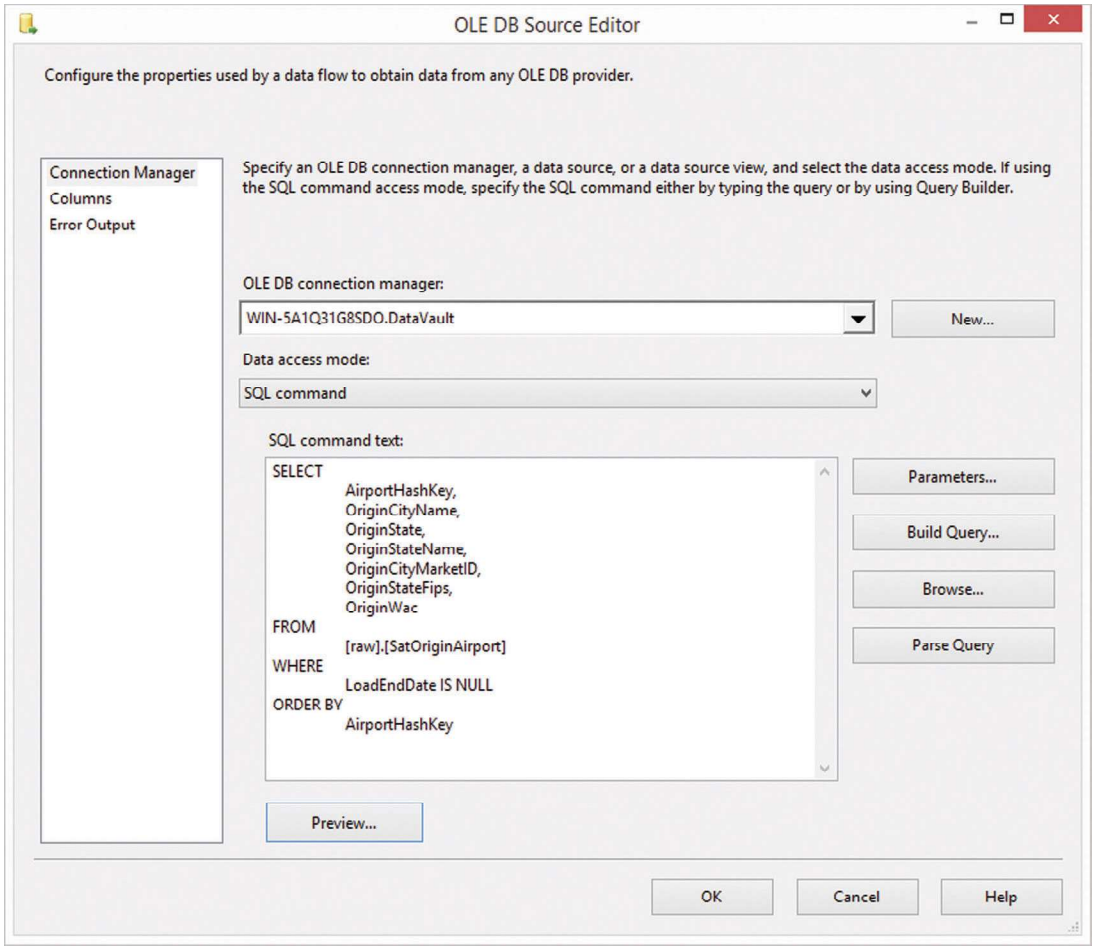

FIGURE 12.45

Source Editor for target data.

There should be two outputs configured in this dialog: one output for records that are new or changed and should be loaded into the target satellite and another output for the records that do not represent a change and should be ignored. The first is configured by adding another output to the list of outputs in the center of the dialog and setting the following condition:

```
REPLACENULL(OriginCityName,"") != REPLACENULL(TgtOriginCityName,"")
|| REPLACENULL(OriginState,"") != REPLACENULL(TgtOriginState,"")
|| REPLACENULL(OriginStateName,"") != REPLACENULL(TgtOriginStateName,"")
|| REPLACENULL((DT_I4)OriginCityMarketID,0) != REPLACENULL(TgtOriginCityMarketID,0)
|| REPLACENULL((DT_I2)OriginStateFips,0) != REPLACENULL(TgtOriginStateFips,0)
|| REPLACENULL((DT_I2)OriginWac,0) != REPLACENULL(TgtOriginWac,0)
```

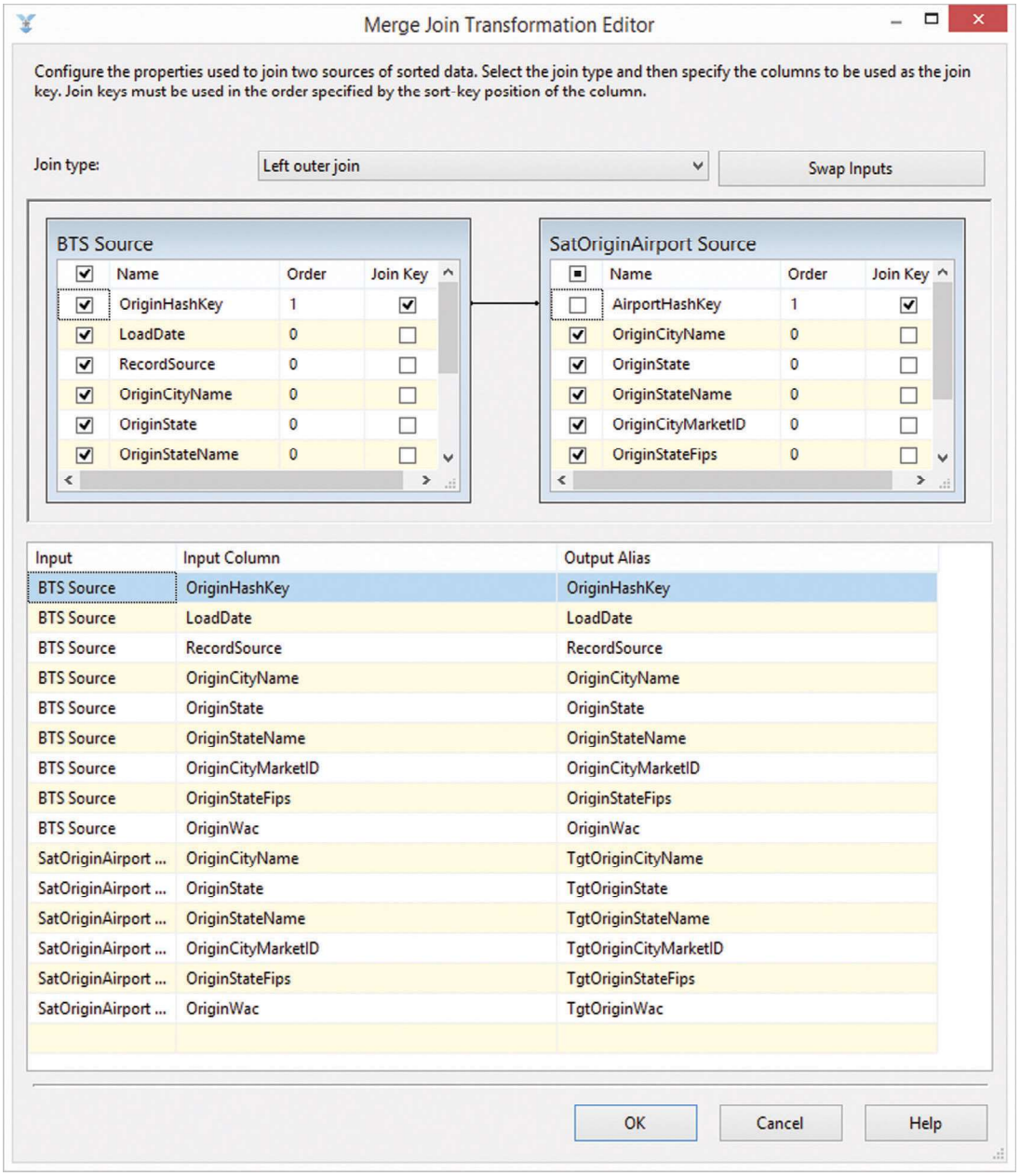

FIGURE 12.46

Merge join transformation editor.

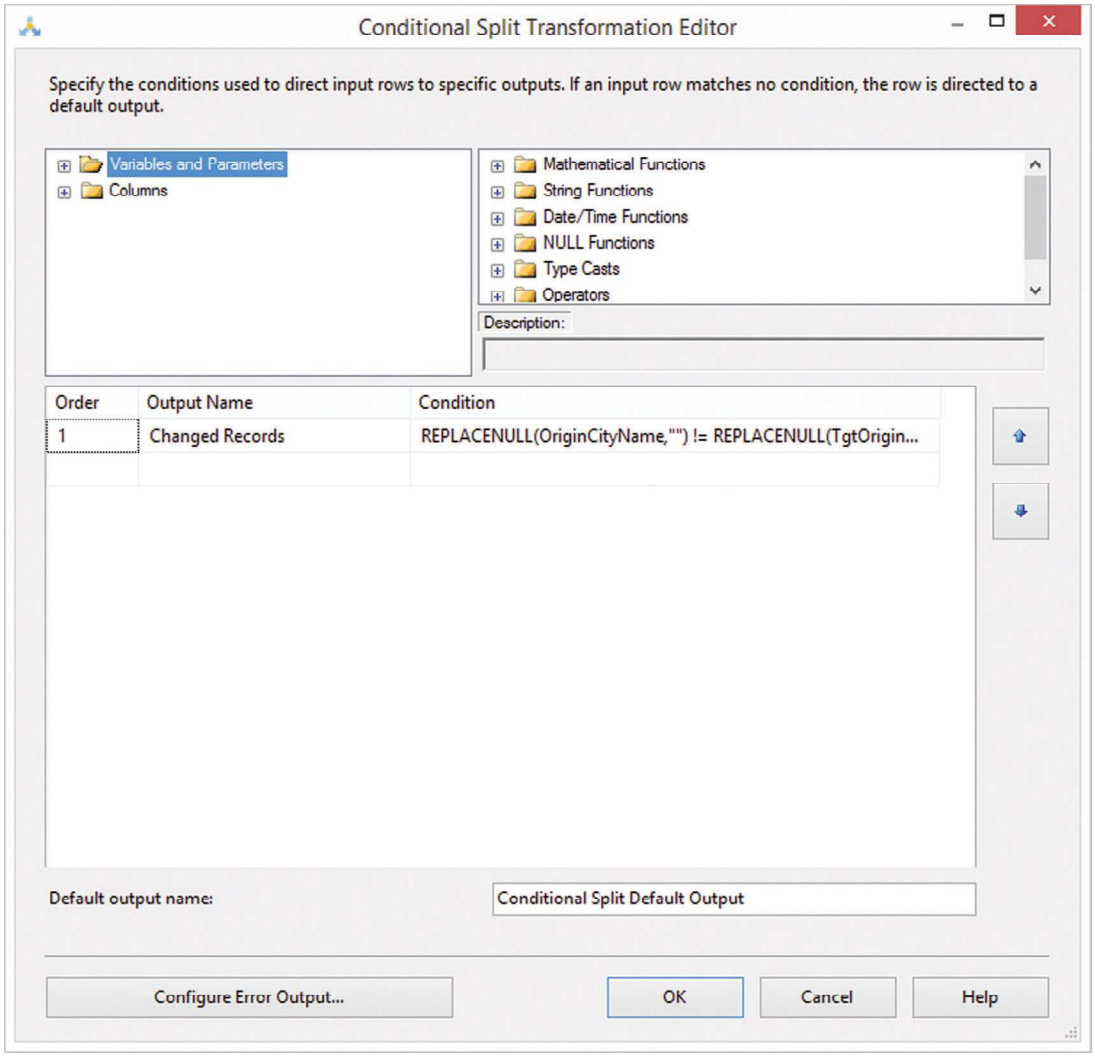

FIGURE 12.47

Conditional split transformation editor.

This condition implements a columnar-based, case-sensitive compare operation that takes potential NULL values into account. In order to implement a case-insensitive version of this operation, the expression should be extended by **UPPER** functions on all columns from both the staging area and the target satellite. It is also important to take special care for columns with a float data type. If any of the fields in the payload are flow, then converting them to a string zero without a forced numeric (fixed decimal point) may actually fail the comparison for equality.

The default output is left as is and is responsible for dealing with the records that are already known to the target satellite. These records will be ignored.

Close the dialog and drag an OLE DB destination to the canvas. Connect the conditional split transformation to the destination component by dragging the data flow output of the conditional split transformation to the destination. The dialog in Figure 12.48 will be shown to let you select the desired output that should be written into the destination.

Select the **changed records** output from the conditional split transformation and the **OLE DB destination input** from the OLE DB destination component. Select the OK button to close the dialog. Open the OLE DB destination editor to set up the target (Figure 12.49).

On the first tab, select the target satellite table. Because NULL values should be written into the target, it is important to check the **keep null** option. To ensure highest performance, **table lock** should be turned on. Parallel loading of the same satellite table should not be required because the recommendation is to load only data from one source system into a satellite (separate data by source system). The other options should be adjusted; especially the **rows per batch** and the **maximum insert commit size** option. **Check constraints** should not be necessary as they often implement soft business rules, which should be implemented later.

Select mappings from the left to edit the column mappings from the columns in the data stream to the destination table columns (Figure 12.50).

Because the columns in the data flow and in the destination table use the same name, the mapping editor should have mapped most columns already. Make sure that each destination column except the **load end date** has a source column. The load end date is left NULL for now because end-dating is separated from this process and is covered in section 12.1.5. In most cases, the hash key needs to be mapped, because the name often differs in the source and the target table.

This completes the setup of the loading process for satellites. The complete data flow is presented in Figure 12.51.

The final loading process presented in the figure can be optimized by comparing the source data with the target data based on the hash diff value instead of each individual column value. In order to do so, the hash diff values have to be included in the data flow and used in the conditional split transformation instead of the individual column values. Therefore, a couple of modifications are required to the data flow presented before. In the following example, the previous data flow has been copied and adopted for another target satellite **SatDestAirport**.

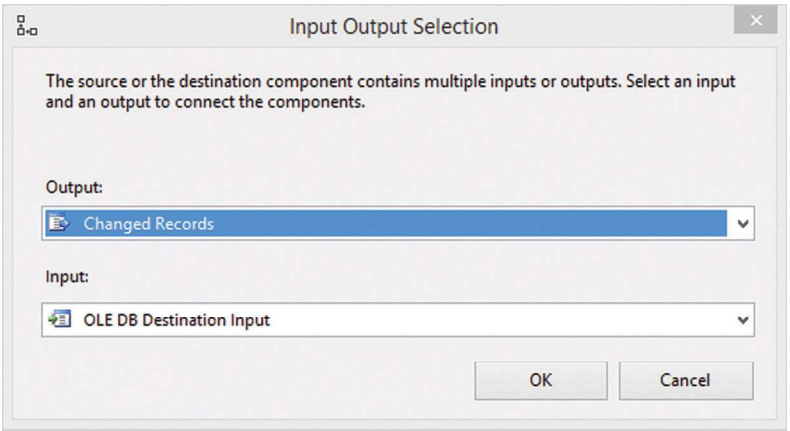

FIGURE 12.48

Input output selection dialog.

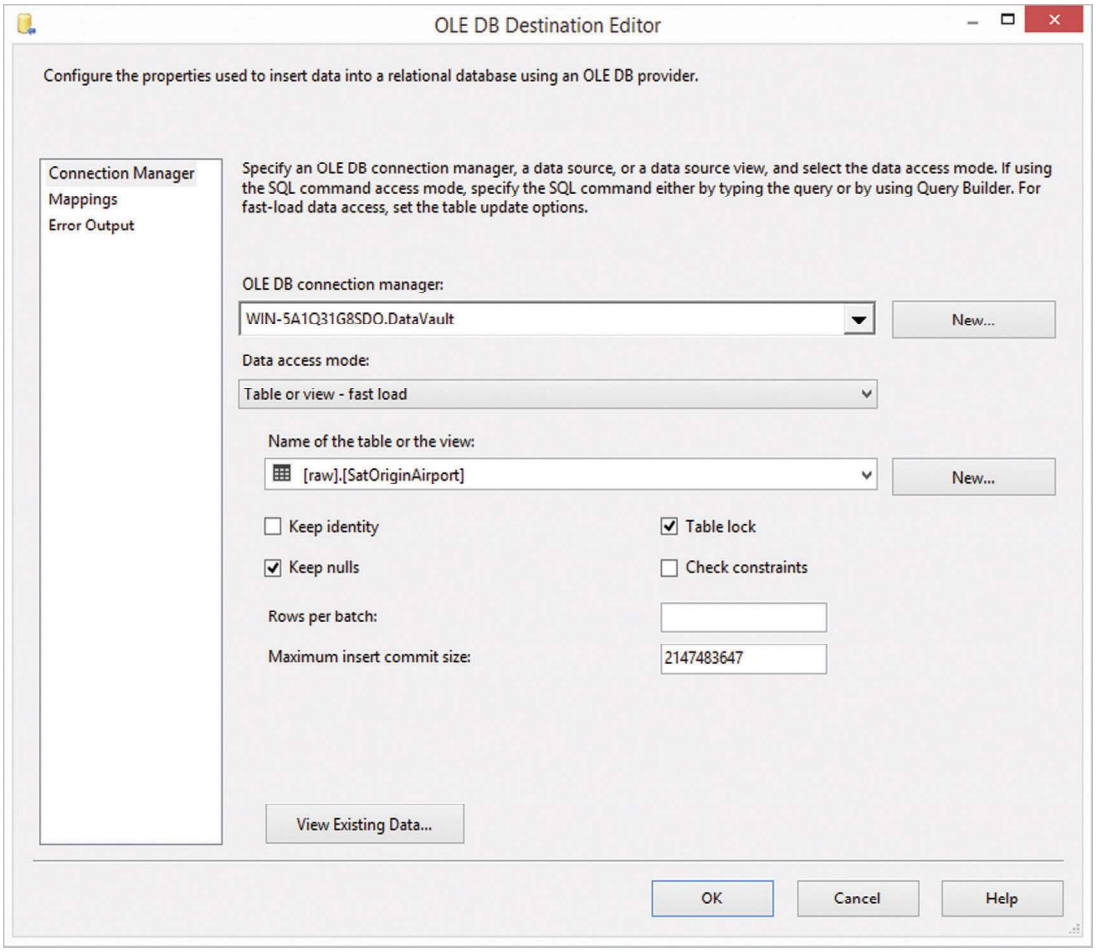

FIGURE 12.49

OLE DB destination editor.

Open the OLE DB source editor for the source table in the staging area (Figure 12.52).
In this dialog, copy and paste the following SQL statement into the **SQL command text** editor:

```
SELECT DISTINCT
       stg.DestHashKey,
       stg.LoadDate,
       stg.RecordSource,
       stg.DestAirportHashDiff,
       stg.DestCityName,
       stg.DestState,
       stg.DestStateName,
       stg.DestCityMarketID,
       stg.DestStateFips,
       stg.DestWac
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD stg
WHERE
       stg.LoadDate = ?
ORDER BY
       stg.DestHashKey
```

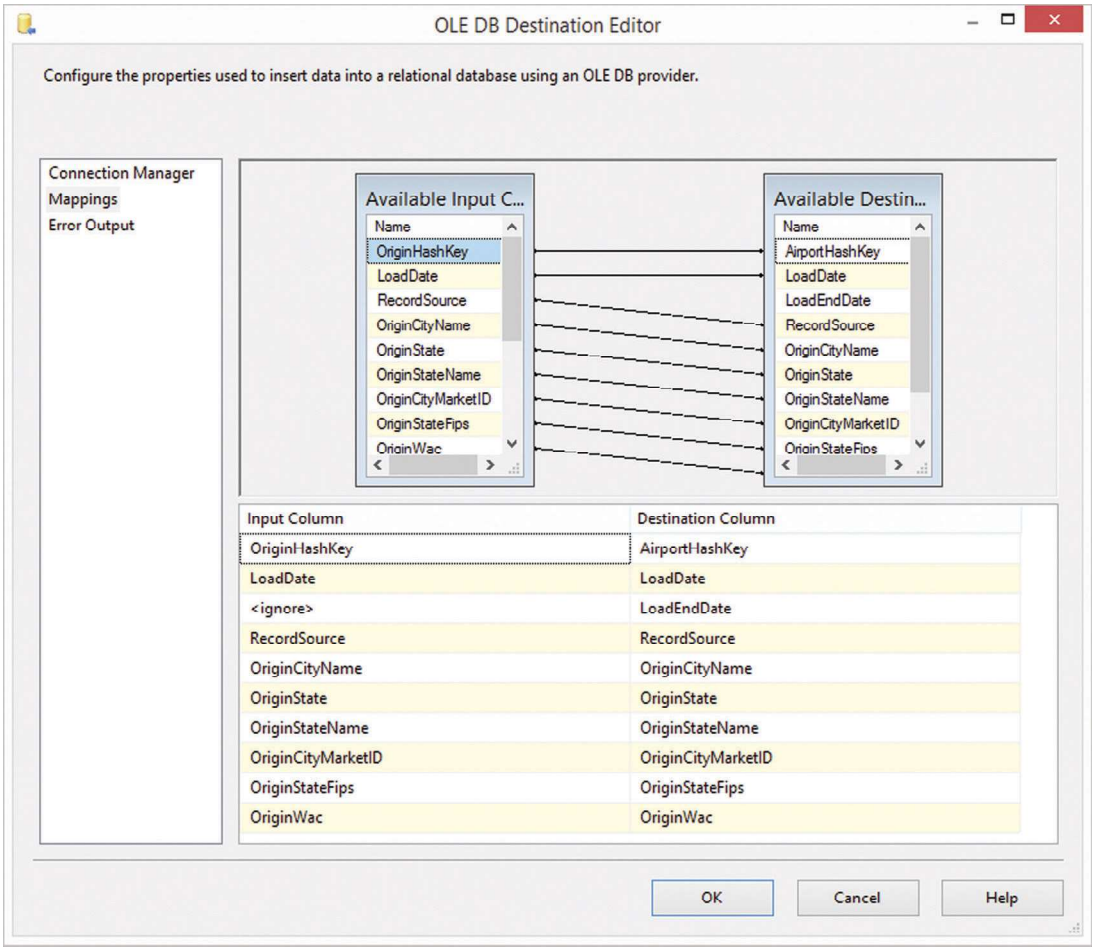

FIGURE 12.50

Mapping the columns from the data stream to the destination.

Make sure that the parameter is still bound to the **load date** variable previously created. Notice that the metadata of the output is out of sync when closing the dialog. Double-click the path in the data flow and select **delete unmapped input columns** to fix the issue.

Open the editor for the second source on the target satellite table (Figure 12.53).

Because the payload of the target satellite is not required for the delta checking, remove all descriptive columns from the SQL command text and add the **hash diff** column:

```
SELECT
       AirportHashKey,
       HashDiff
FROM
       [raw].[SatDestAirport]
WHERE
       LoadEndDate IS NULL
ORDER BY
       AirportHashKey
```

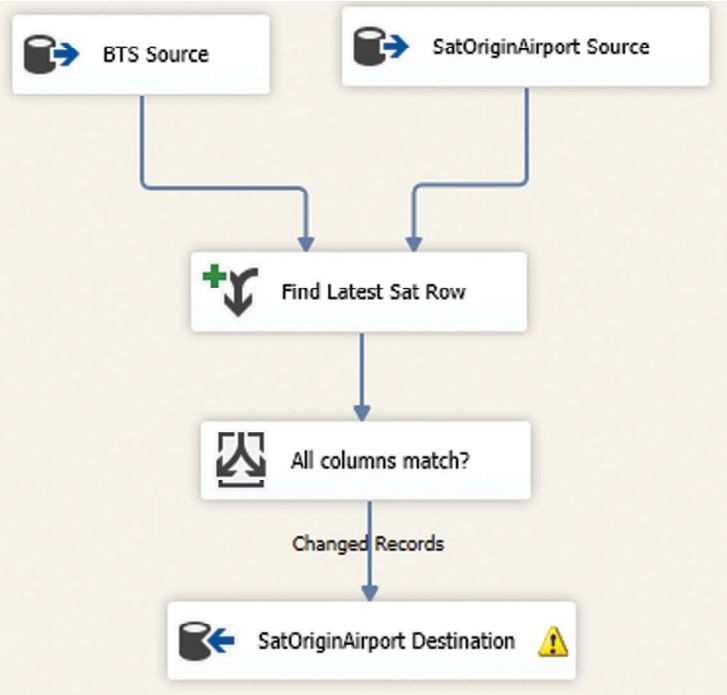

FIGURE 12.51

Satellite loading process based on column compare.

No other columns except the parent hash key and the hash diff values are required from the target. However, make sure that only active records are returned by limiting the result to records with a **load end date** of NULL.

After closing the dialog, fixing the metadata of the path in the data flow might be required for this source as well. Also make sure that both sources have **IsSorted** set to true for their output and the **SortKeyPosition** of the respective hash key column is set to 1.

The next task is to fix the merge join by using the merge join transformation editor, shown in Figure 12.54.

Make sure that the hash key from the staging area is mapped to the hash key in the target satellite table. In Figure 12.54, the **DestHashKey** is mapped to the **AirportHashKey**. In addition, select all columns from the staging area, because they contain the descriptive data that should be loaded into the target satellite. In order to check if any columns have changed, add the **hash diff** column from the target satellite to the data flow.

Close the dialog and open the editor of the **conditional split transformation** (Figure 12.55).

Replace the condition in the dialog shown in Figure 12.55 by the following expression:

```
ISNULL([HashDiff]) || ([DestAirportHashDiff]!=[HashDiff])
```

This expression only uses the hash diffs from the source and the target to perform the delta checking. It takes possible NULL values in the target hash diff into consideration to ensure that new records

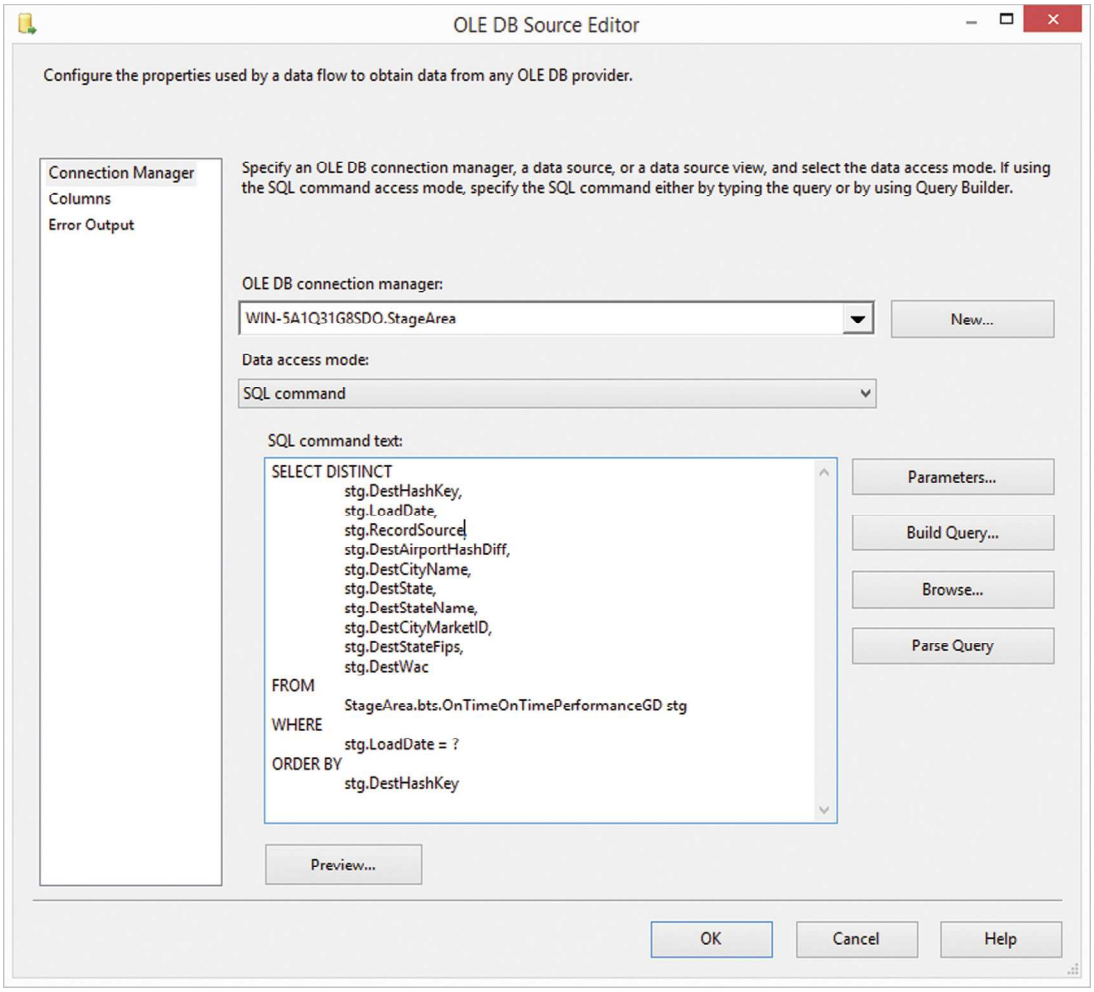

FIGURE 12.52

OLE DB source editor for destination airport staging data.

are loaded as well (they don't have a corresponding record in the target, thus their target hash diff is NULL).

Finally, the OLE DB destination has to be set up. Figure 12.56 shows the first page that sets up the table.

Change the name of the target to **SatDestAirport**. Make sure that keep nulls are enabled and switch to the mappings pane (Figure 12.57).

Make sure that all descriptive columns are mapped to the target and that the hash diff value from the source table in the staging area is mapped to the hash diff column of the target satellite. This is important to ensure that the hash diff is available when running this process another time.

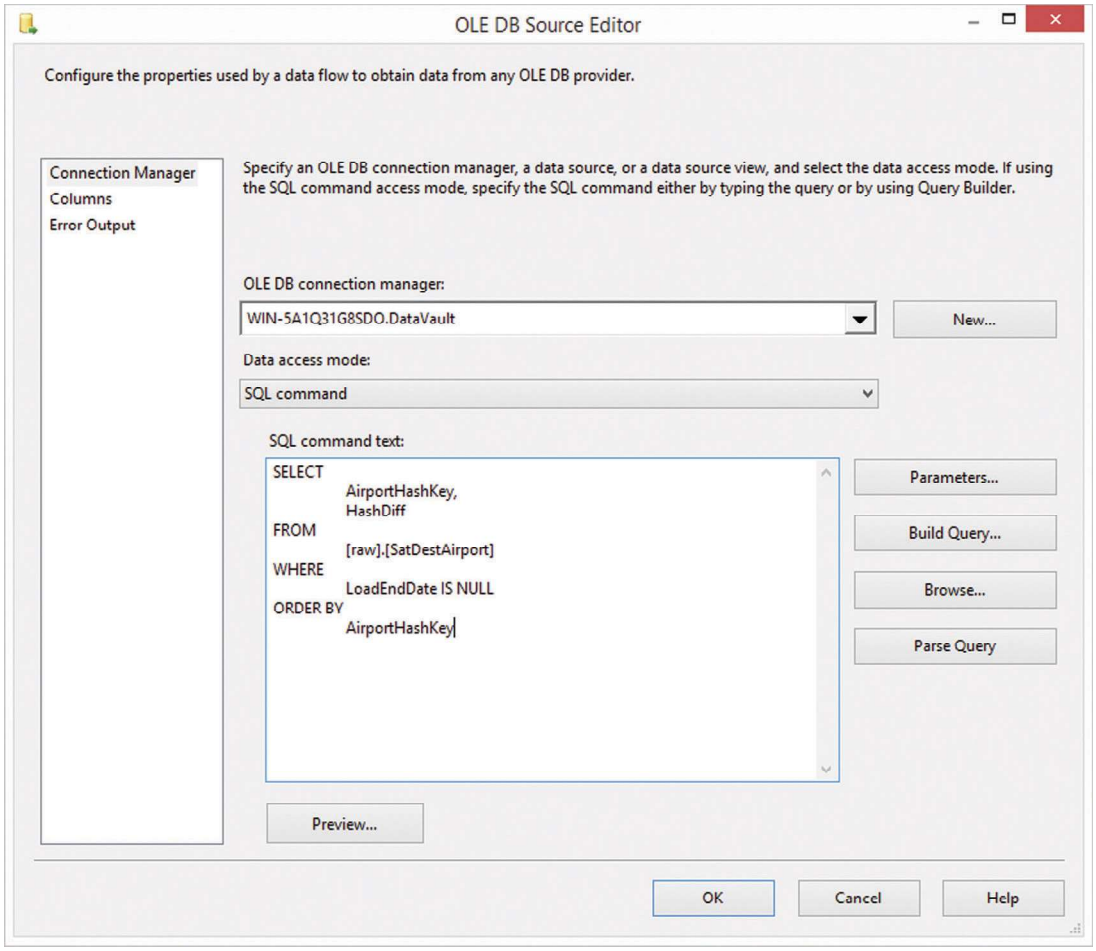

FIGURE 12.53

OLE DB source editor of the target satellite table.

The final data flow is presented in Figure 12.58.

In addition to the presented setup, a production-ready loading process would write erroneous data from the source into the error mart if it fails to load the data. In order to add the error mart to the data flows presented in this section, send error outputs to OLE DB destinations for the error mart as described in Chapter 10, Metadata Management.

12.1.5 END-DATING SATELLITES

The standard loading process of satellites is not complete yet. After the loading process for a satellite has been completed, there are multiple satellite entries active: two or more records have a load end date of NULL or 9999-12-31. This is invalid because there should be only one record active in a consistent

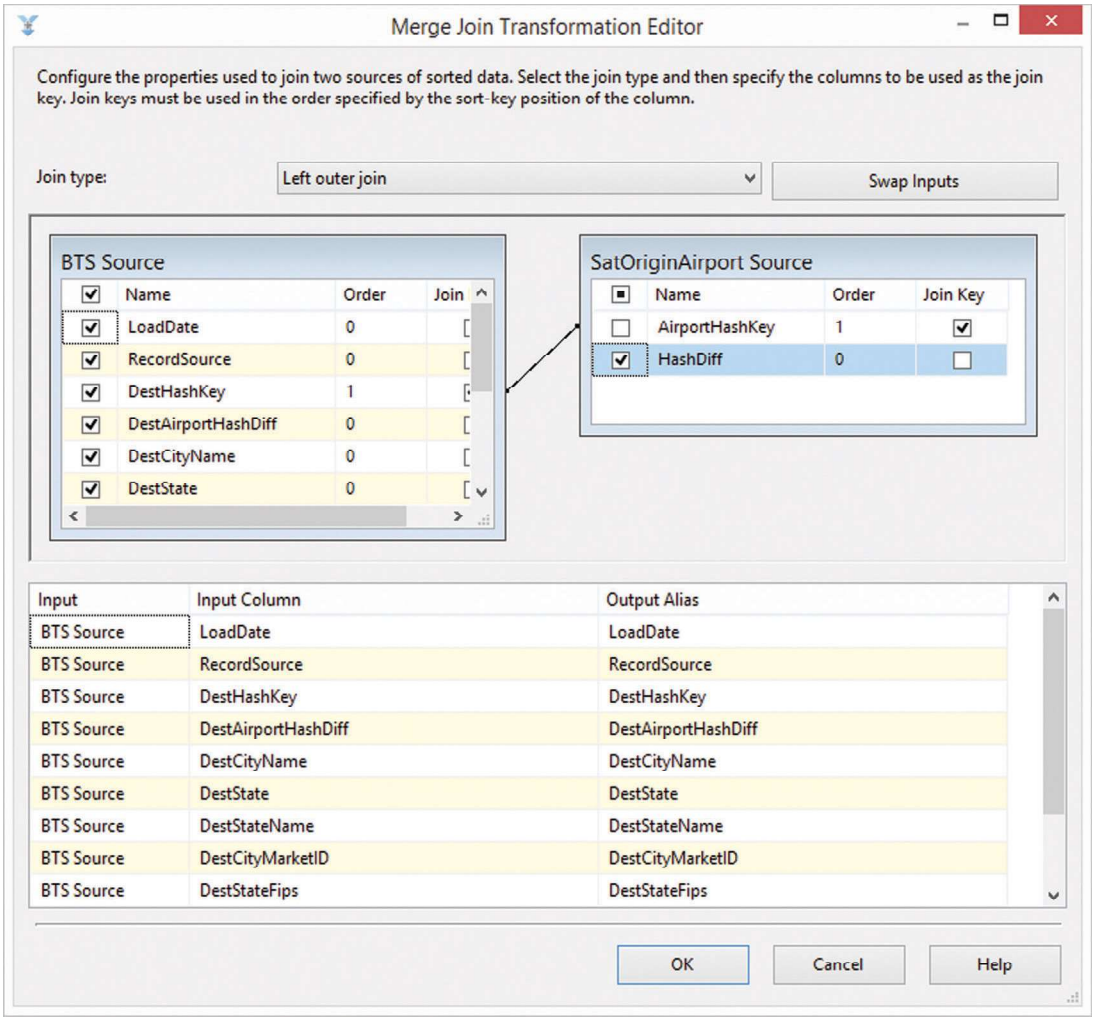

FIGURE 12.54

Merge join transformation editor for hash diff column compare.

Data Vault satellite (except for multi-active satellites). Therefore, an end-dating process is required after having loaded the data of one batch (one load date) from the staging table into the target satellite. The overall process is presented in Figure 12.59.

The first step in this process is to retrieve the data from the satellite table. Once the records have been retrieved, the records are sorted per group and per load date (descending) in the second step.

Within each group, the first record needs to be calculated (step 3) because this record should remain active. Therefore, the process sets the load end date to NULL for this record (or makes sure that this record keeps its NULL or future load end date). After that, the load end-dates of the remaining records

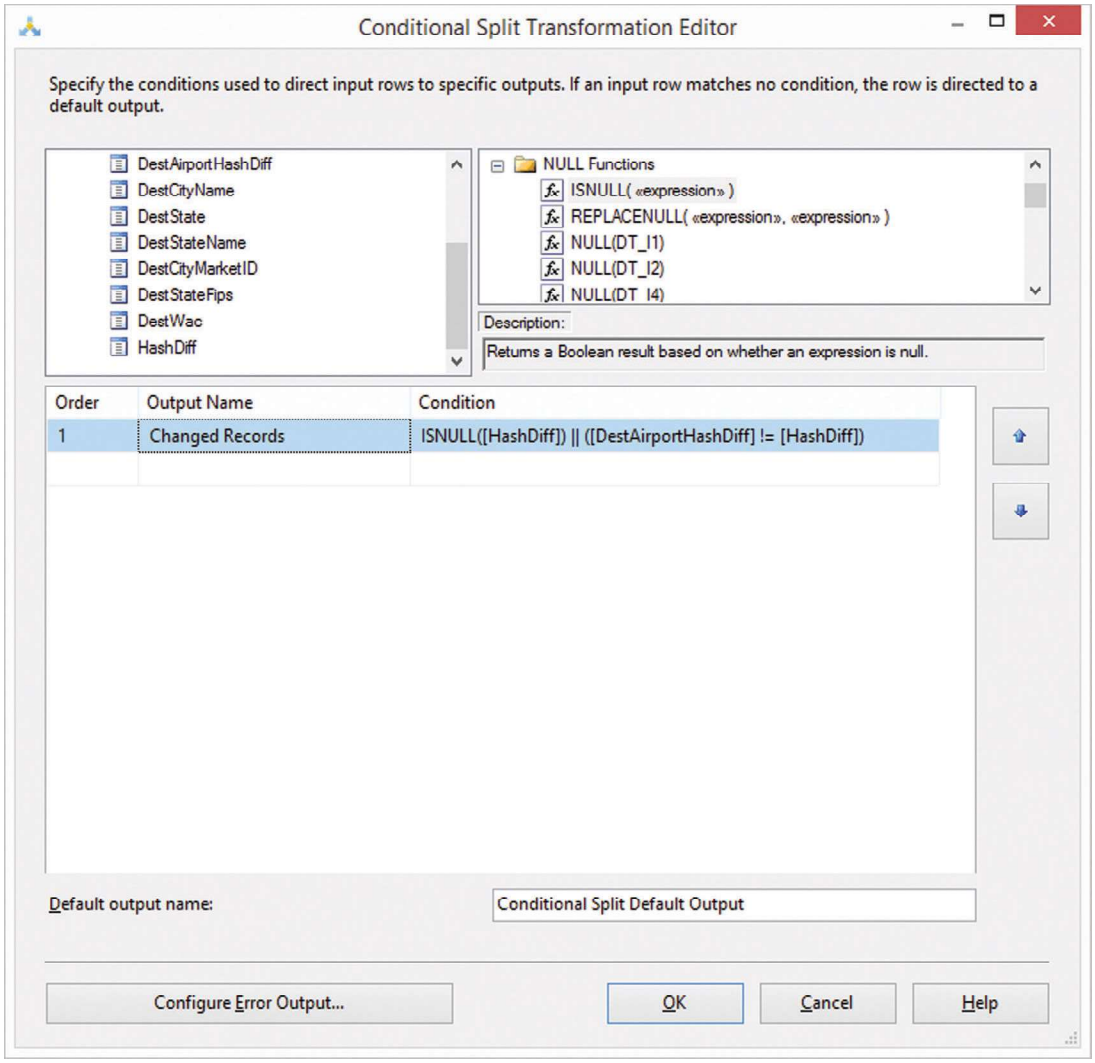

FIGURE 12.55

Conditional split transformation editor for delta checking based on hash diff.

are calculated for each group. This process is described next. Once all load end-dates are calculated, the target satellite is updated with the new load end-dates and the process completes.

Note that the sort operation in the second step of this process forces the use of the tempdb database in Microsoft SQL Server if the process is implemented in SQL Server. If implemented in SSIS, the sort operation will release the lock on the source component's connection of the satellite table.

The subprocess to calculate the load end-dates is shown in Figure 12.60.

In the first step, the incoming record set is reduced to those groups of data per parent hash key which have more than one active record per group. Active records are defined by records with a load end date

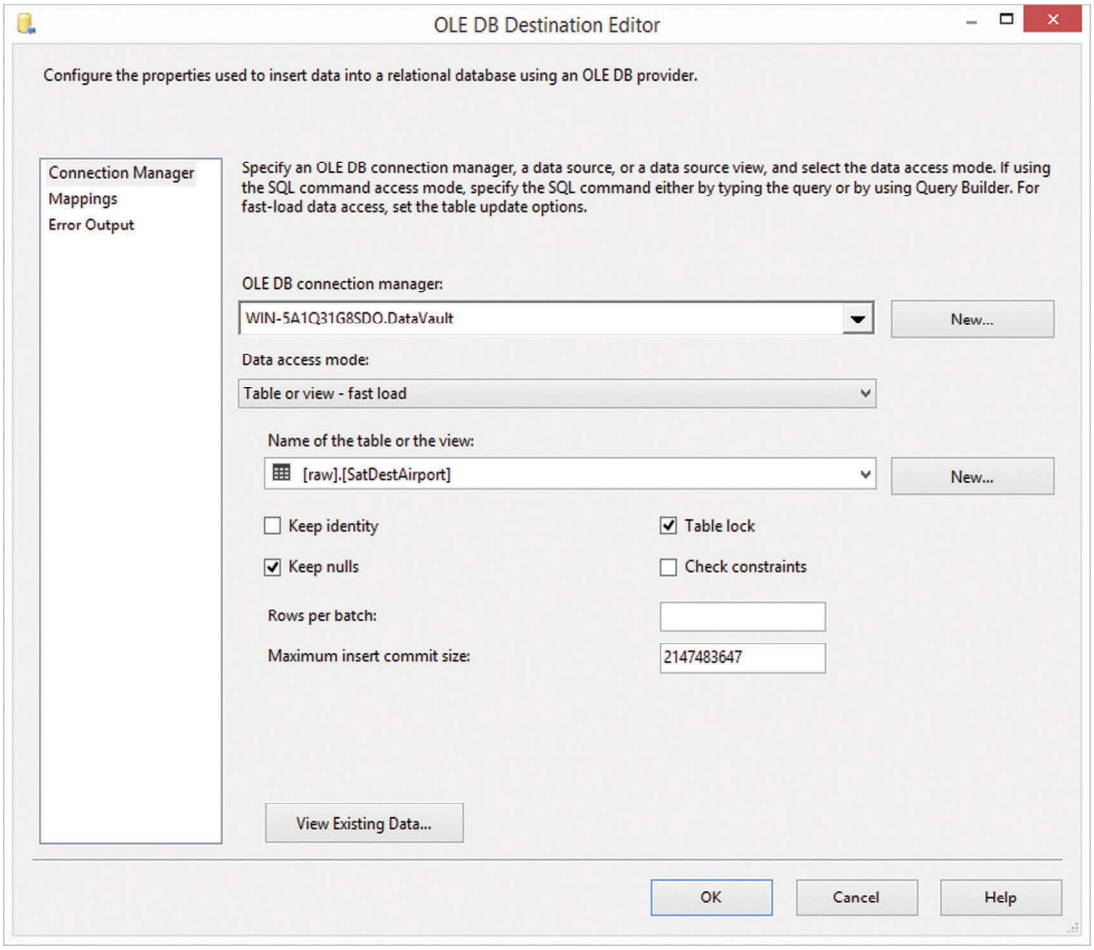

FIGURE 12.56

OLE DB destination editor for SatDestAirport.

of NULL or 9999-12-31 (the future date). All other records are not of interest for this process, because if there are not at least two active records per parent hash key, the single record has to remain open and, therefore, remains untouched.

After these records have been removed, each group is ordered by load date. This is achieved by ordering the whole data flow by hash key (to group all records belonging to one parent hash key together) and load date (descending). Ordering the load dates in a descending order is important because the process has to calculate "carry over" values for each record. This process is shown in Figure 12.61.

The first table on the left includes four parent hash keys. Only the data that belongs to the hash keys 2abcff… and 4444dd… are processed by this template because the other two hash keys don't have more than two active records in the satellite. For example, the data for the parent hash key 1bef79…

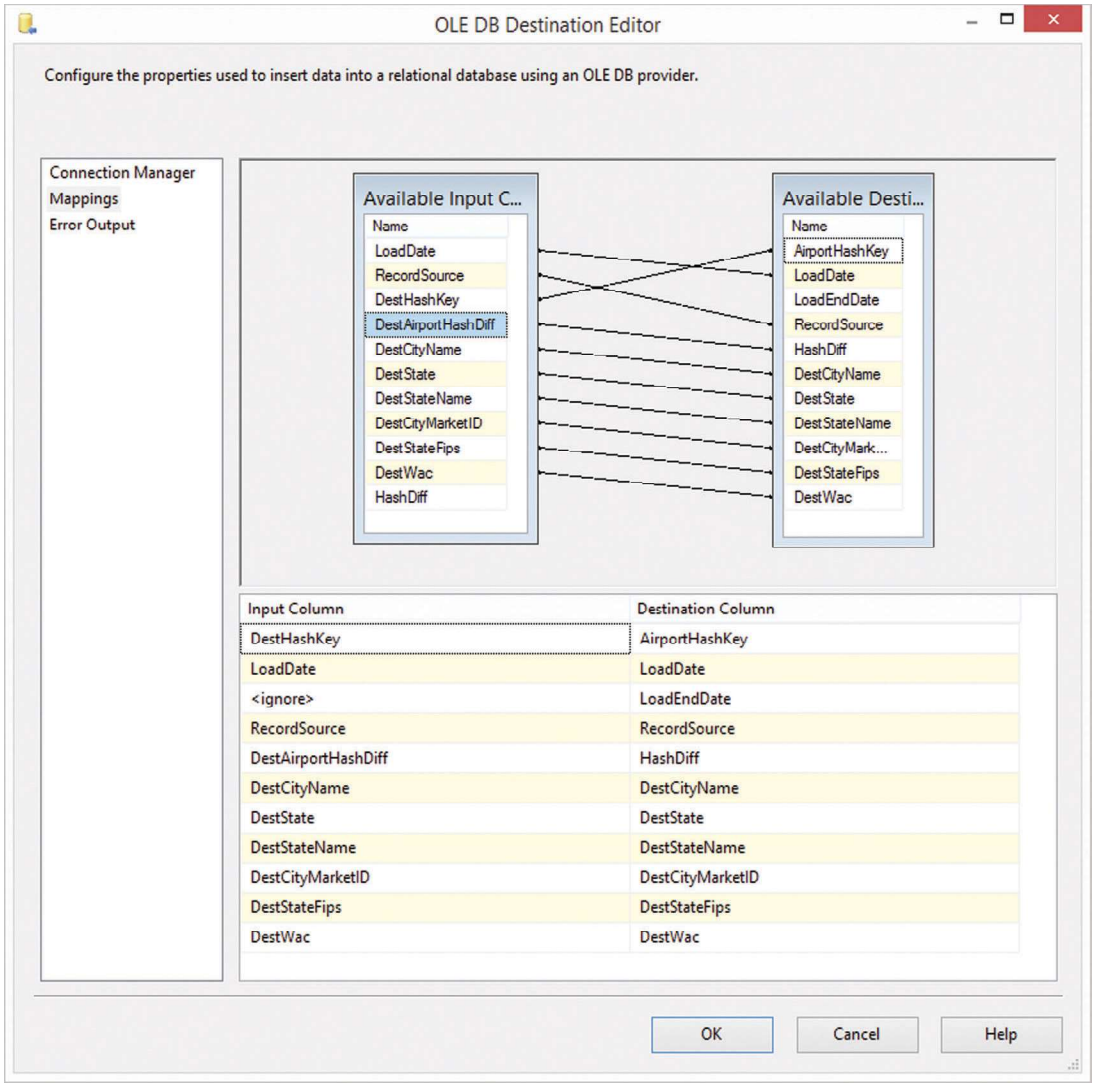

FIGURE 12.57

OLE DB destination editor for mapping the source columns to the target.

consists of one active record and one record already end-dated in the past. Once the data has been filtered out, the data is sorted by hash key and load end-date in the second step. The result is shown in the upper-right table.

Then, the "carry-over" value is calculated for each record within the group, except the first one (which should remain active). Each record except the first in the group should have a load end-date set to the load date of the next record in the group (ordered on a time-scale). Because of that, the load

date of the newer record is copied over to the next older record, as shown in the lower-right table of Figure 12.61. Once all load end-dates have been set, the data is sent as updates to the target satellite.

The following T-SQL statement can be used to perform the end-dating process:

```
UPDATE SatPreferredDish SET
       LoadEndDate = (
              SELECT
                     DATEADD(ss,-1,MIN(z.LoadDate))
              FROM
                     SatPreferredDish z
              WHERE
                     z.PassengerHashKey = a.PassengerHashKey
              AND
                     z.LoadDate > a.LoadDate
       )
FROM
       SatPreferredDish a
WHERE
       LoadEndDate IS NULL AND PassengerHashKey = ?
```

This statement end-dates the satellite **SatPreferredDish**. It assumes open satellite entries (records which are not end-dated yet) to be are indicated by a NULL **LoadEndDate**. The sub-query in the statement retrieves the **LoadDate** of the next entry for each **PassengerHashKey**. By doing so, it implements the process as outlined in Figure 12.61.

The previous statement should be executed for each parent hash key individually, in order to ensure that it scales to higher amounts of data. While the statement is not a fast approach, the statement scales linearly on the number of parent hash keys. If the load end-dates of all hash keys are updated at once, in a single transaction, the resource requirements could become too high and the database server may start to swap data out of memory to disk in order to deal with the amounts of data. Depending on the database management system used, other approaches than the one presented in the previous statement are more feasible as well.

The process of end-dating the satellite data has been separated from the loading process in order to deal with each problem separately. This simplifies the loading process and ensures that the performance remains high. Another important issue is that the statement deals with all the whole group of data per parent hash key at once, in order to make sure that previously loaded data becomes end-dated as well. This is important because a previous loading process might have been aborted and may left the satellite data in an inconsistent state. Therefore, this separation becomes part of the restartability characteristics of the Data Vault 2.0 loading patterns.

12.1.6 SEPARATE NEW FROM CHANGED ROWS

The performance of the templates shown in the previous section relied on the separation of duties to some extent. For example, several operations have already been conducted in the staging area, such as hashing. Other operations are performed once the data have been loaded, such as setting the load end-date but also performing any recalculations.

The performance of the above templates can be improved further by more separation. This relies on the fact that the column comparisons are only required for parent hash keys that are already known to the satellite, i.e., potential updates to the same hash key. If the source system contains a record that describes a business key or business key relationship that is already known to the data warehouse, the column comparison is used to find out if the source system has changed between the loads.

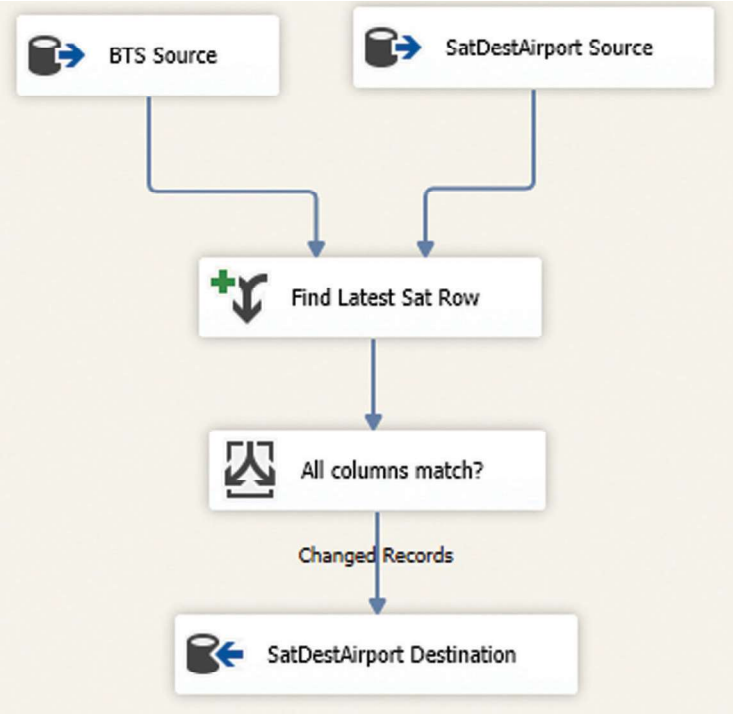

FIGURE 12.58

Complete data flow for loading satellites using hash diffs.

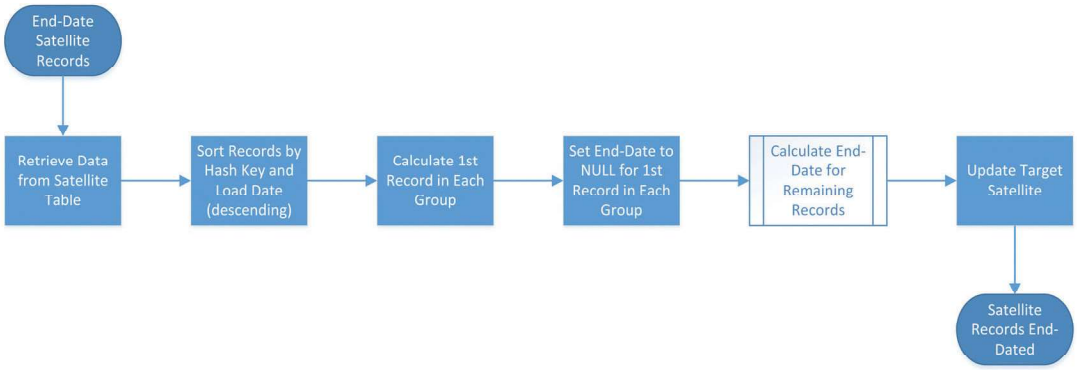

FIGURE 12.59

Template for setting the load end-date in satellite rows.

FIGURE 12.60

Calculate load end-date for remaining records subprocess.

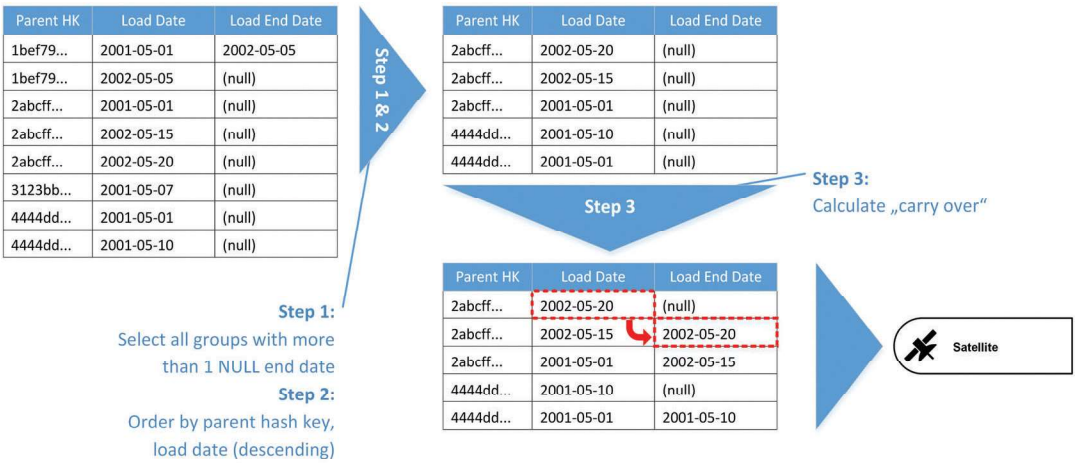

FIGURE 12.61

Calculate load end-date walkthrough.

But if the source system described a business key that is unknown to the satellite, the column comparison is not required: there will be no record available in the target that could be used in the update detection process. The new record has to be loaded into the target satellite in any case.

The previous templates always perform the column comparison. If SQL is used to implement the template, the compare might require an outer join because changed rows are detected using direct index match. But even if ETL is used, the column comparison is not required and reduces performance of the overall load. Therefore, it is possible to improve the performance of the satellite loads by separating new and changed records. This is especially useful if the number of new records in the source system outweighs the number of changed records. The programming logic that is required to detect changes should not be applied to new records because it reduces the performance of the loading pattern.

The first resulting template, which loads changed rows from the staging table only, remains mostly unchanged to the default satellite loading template in section 12.1.4 (Figure 12.62).

The only difference is in the first step because it only loads changed records from the staging tables. But implementing this step requires the identification of records that have potentially changed directly in the staging area. However, this is easy to do if the staging area is on the same infrastructure as the rest of the data warehouse. In this case, it is possible to do the source query on the staging area

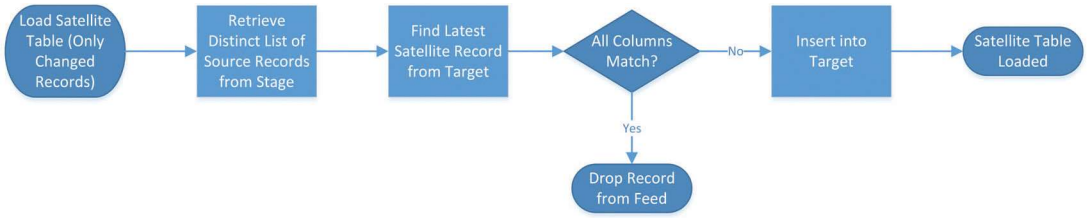

FIGURE 12.62

Template for loading only changed records into satellite tables.

and perform a look-ahead into the target to identify which hash keys are already known to the data warehouse:

```
SELECT DISTINCT
       stg.OriginHashKey,
       stg.LoadDate,
       NULL,
       stg.RecordSource,
       stg.OriginCityName,
       stg.OriginState,
       stg.OriginStateName,
       stg.OriginCityMarketID,
       stg.OriginStateFips,
       stg.OriginWac
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD stg
INNER JOIN
       DataVault.[raw].SatOriginAirport sat
       ON (stg.OriginHashKey = sat.AirportHashKey AND sat.LoadEndDate IS NULL)
WHERE
       (
       (CASE WHEN (stg.OriginCityName IS NULL AND sat.OriginCityName IS NULL
              OR stg.OriginCityName = sat.OriginCityName)
              THEN 1 ELSE 0 END) = 0
       OR (CASE WHEN (stg.OriginState IS NULL AND sat.OriginState IS NULL
              OR stg.OriginState = sat.OriginState)
              THEN 1 ELSE 0 END) = 0
       OR (CASE WHEN (stg.OriginStateName IS NULL AND sat.OriginStateName IS NULL
              OR stg.OriginStateName = sat.OriginStateName)
              THEN 1 ELSE 0 END) = 0
       OR (CASE WHEN (stg.OriginCityMarketID IS NULL AND sat.OriginCityMarketID IS NULL
              OR stg.OriginCityMarketID = sat.OriginCityMarketID)
              THEN 1 ELSE 0 END) = 0
       OR (CASE WHEN (stg.OriginStateFips IS NULL AND sat.OriginStateFips IS NULL
              OR stg.OriginStateFips = sat.OriginStateFips)
              THEN 1 ELSE 0 END) = 0
       OR (CASE WHEN (stg.OriginWac IS NULL AND sat.OriginWac IS NULL
              OR stg.OriginWac = sat.OriginWac)
              THEN 1 ELSE 0 END) = 0
       )
       AND stg.LoadDate = '2011-01-22 00:00:00.000'
```

This statement shows that the staging area is not only used to reduce the workload on the source systems, but it can also be used to perform intermediate tasks that improve the performance of the data warehouse loading processes. And it shows the additional value of having calculated the hash keys in the staging area: it does not only improve the reusability of the hash computations, but can also be used to improve the loading processes even further (as described in this section).

In order to load the new records only, the loading template in Figure 12.62 can be greatly simplified because no change detection is required by the loading process anymore (Figure 12.63).

Not only is the column compare removed, but also the retrieval of the latest record from the target satellite table. New records are just loaded into the target. This template also requires that the source query on the staging area provides only new records with hash keys unknown to the target satellite:

```
SELECT DISTINCT
       stg.OriginHashKey,
       stg.LoadDate,
       NULL,
       stg.RecordSource,
       stg.OriginCityName,
       stg.OriginState,
       stg.OriginStateName,
       stg.OriginCityMarketID,
       stg.OriginStateFips,
       stg.OriginWac
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD stg
LEFT OUTER JOIN
       DataVault.[raw].SatOriginAirport sat
       ON (stg.OriginHashKey = sat.AirportHashKey AND sat.LoadEndDate IS NULL)
WHERE
       (
       (CASE WHEN (stg.OriginCityName IS NULL AND sat.OriginCityName IS NULL
              OR stg.OriginCityName = sat.OriginCityName)
              THEN 1 ELSE 0 END) = 0
       OR (CASE WHEN (stg.OriginState IS NULL AND sat.OriginState IS NULL
              OR stg.OriginState = sat.OriginState)
              THEN 1 ELSE 0 END) = 0
       OR (CASE WHEN (stg.OriginStateName IS NULL AND sat.OriginStateName IS NULL
              OR stg.OriginStateName = sat.OriginStateName)
              THEN 1 ELSE 0 END) = 0
       OR (CASE WHEN (stg.OriginCityMarketID IS NULL AND sat.OriginCityMarketID IS NULL
              OR stg.OriginCityMarketID = sat.OriginCityMarketID)
              THEN 1 ELSE 0 END) = 0
       OR (CASE WHEN (stg.OriginStateFips IS NULL AND sat.OriginStateFips IS NULL
              OR stg.OriginStateFips = sat.OriginStateFips)
              THEN 1 ELSE 0 END) = 0
       OR (CASE WHEN (stg.OriginWac IS NULL AND sat.OriginWac IS NULL
              OR stg.OriginWac = sat.OriginWac)
              THEN 1 ELSE 0 END) = 0
       )
       AND sat.AirportHashKey IS NULL --new records only
       AND stg.LoadDate = '2011-01-22 00:00:00.000'
```

Both processes for new and changed records can run in parallel, because they operate on different sets of input data which are loaded into nonoverlapping primary keys of the target table (again, because the processed hash keys of each ETL process are different).

FIGURE 12.63

Template for loading only new records into satellite tables.

12.1.7 NO-HISTORY SATELLITES

The approach described in the previous section can also be used to load nonhistorized satellites. The reason is that the no-history satellite does not track changes in the source system. Instead, there are only new records that need to be loaded into the target table. Because of that, the same template as presented in Figure 12.63 can be used for loading satellites with no history.

The following DDL statement creates a no-history satellite in the Raw Data Vault:

```
CREATE TABLE [raw].[TSatFlight](
       [FlightHashKey] [char](32) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [Year] [smallint] NULL,
       [Quarter] [smallint] NULL,
       [Month] [smallint] NULL,
       [DayOfMonth] [smallint] NULL,
       [DayOfWeek] [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,
 CONSTRAINT [PK_TSatFlight] PRIMARY KEY NONCLUSTERED
(
	[FlightHashKey] ASC,
	[LoadDate] ASC
) ON [INDEX]
) ON [DATA]
```

There is no **load end date** in the satellite structure because changes are not tracked. For the same reason, the **load date** has been degraded to technical field without any importance for the model. The primary application is to be used during debugging: it exists in order to understand when data was loaded into the target. However, there is a secondary application to it: when incrementally loading subsequent information marts, it is helpful to load only new records into the information mart layer. This is why the load date was included in the primary key constraint: by doing so, the field is used for indexing, which improves the load performance of the information mart.

Before the data can be loaded to the information mart, which is described in Chapter 14, the data has to be loaded from the staging area into the no-history satellite. The following statement can be used for this purpose:

```
INSERT INTO DataVault.[raw].TSatFlight (
	[FlightHashKey]
	,[LoadDate]
	,[RecordSource]
	,[Year]
	,[Quarter]
	,[Month]
	,[DayOfMonth]
	,[DayOfWeek]
	,[CRSDepTime]
	,[DepTime]
	,[DepDelay]
	,[DepDelayMinutes]
	,[DepDel15]
	,[DepartureDelayGroups]
	,[DepTimeBlk]
	,[TaxiOut]
	,[WheelsOff]
	,[WheelsOn]
	,[TaxiIn]
	,[CRSArrTime]
	,[ArrTime]
	,[ArrDelay]
	,[ArrDelayMinutes]
	,[ArrDel15]
	,[ArrivalDelayGroups]
	,[ArrTimeBlk]
	,[Cancelled]
	,[CancellationCode]
	,[Diverted]
	,[CRSElapsedTime]
	,[ActualElapsedTime]
```

```
,[AirTime]
,[Flights]
,[Distance]
,[DistanceGroup]
,[CarrierDelay]
,[WeatherDelay]
,[NASDelay]
,[SecurityDelay]
,[LateAircraftDelay]
,[FirstDepTime]
,[TotalAddGTime]
,[LongestAddGTime]
)
SELECT
       stg.[FlightHashKey]
      ,stg.[LoadDate]
      ,stg.[RecordSource]
      ,stg.[Year]
      ,stg.[Quarter]
      ,stg.[Month]
      ,stg.[DayOfMonth]
      ,stg.[DayOfWeek]
      ,stg.[CRSDepTime]
      ,stg.[DepTime]
      ,stg.[DepDelay]
      ,stg.[DepDelayMinutes]
      ,stg.[DepDel15]
      ,stg.[DepartureDelayGroups]
      ,stg.[DepTimeBlk]
      ,stg.[TaxiOut]
      ,stg.[WheelsOff]
      ,stg.[WheelsOn]
      ,stg.[TaxiIn]
      ,stg.[CRSArrTime]
      ,stg.[ArrTime]
      ,stg.[ArrDelay]
      ,stg.[ArrDelayMinutes]
      ,stg.[ArrDel15]
      ,stg.[ArrivalDelayGroups]
      ,stg.[ArrTimeBlk]
      ,stg.[Cancelled]
      ,stg.[CancellationCode]
      ,stg.[Diverted]
      ,stg.[CRSElapsedTime]
      ,stg.[ActualElapsedTime]
      ,stg.[AirTime]
      ,stg.[Flights]
      ,stg.[Distance]
      ,stg.[DistanceGroup]
      ,stg.[CarrierDelay]
      ,stg.[WeatherDelay]
      ,stg.[NASDelay]
      ,stg.[SecurityDelay]
      ,stg.[LateAircraftDelay]
      ,stg.[FirstDepTime]
      ,stg.[TotalAddGTime]
      ,stg.[LongestAddGTime]
```

```
FROM
        StageArea.bts.OnTimeOnTimePerformanceGD stg
LEFT OUTER JOIN
        DataVault.[raw].TSatFlight sat ON (stg.FlightHashKey = sat.FlightHashKey)
WHERE
        sat.FlightHashKey IS NULL
```

This statement implements a simple copy process. Restartability is guaranteed by adding a LEFT OUTER JOIN that checks if the hash key of the parent no-history link is already in the target satellite. If this is the case, the record from the staging area is ignored, otherwise loaded. The load date is not required for this join, because there should only be one record per parent hash key in the target satellite.

12.1.8 SOFT-DELETING DATA IN HUBS AND LINKS

The processes presented in sections 12.1.1 and 12.1.2 load business keys or relationships between business keys into the data warehouse that have been inserted into the operational source system since the last batch. However, business keys are not only inserted into the data warehouse; they are also changed or deleted. Changing business keys is more of a business problem and requires special care up to the extent of asking the question whether the business key that is loaded into the hub is actually the right business key. Review the requirements for business keys in Chapter 4.

However, deleting keys is a common and valid operation and happens frequently. A product is removed from the catalog and not offered anymore, users lose access and their user account is removed from the operational system, etc. The same applies for relationships: employees quit and hire on with a new organization, products are moved into another category, and so on. In all these cases, the business demands that these changes be loaded into the data warehouse to be available for analysis. But the data warehouse doesn't only reflect the current state of the data, but also the complete, nonvolatile history of the business keys, their relationships and all the descriptive data. For this reason, historic data usually cannot be deleted from the data warehouse.

Hubs and links play a vital role in the Data Vault 2.0 model to integrate the data of the data warehouse. If business keys are deleted from hubs or business key relationships are deleted from links, this integration breaks because descriptive data is stored in dependent satellites and references these business keys or relationships. Therefore, deleting records from hubs or links is not an option if the model is to remain intact.

The same applies to end-dating hubs and links. This approach would require a **load end date** in hub and link tables and could be used to end the employment of an employee with a specific company, for example. However, this approach introduces another problem (among others): what if the employee realizes that the new company was a bad choice and returns to the old organization? The HR people in the organization don't care that hiring back the employee would break the Data Vault model due to duplicate keys in link tables.

The solution is to use effectivity satellites, introduced in Chapter 5, Intermediate Data Vault Modeling, to model such effectivity dates. The hub only provides information as to which business keys ever existed in the source systems and the link table provides the information about all relationships that ever existed. The corresponding effectivity satellite on the table provides the information about the effectivity that is the start and the end date of the business key existence or relationship validity.

Actually deleting data from the data warehouse is only performed when legal or privacy requirements have to be met, for example in a data destruction strategy.

12.1.8.1 T-SQL Example

The following DDL creates an effectivity satellite which is used to track the validity of an airline ID:

```
CREATE TABLE [raw].[SatAirlineIDEffectivity](
       [AirlineIDHashKey] [char](32) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [LoadEndDate] [datetime2](7) NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [Year Start] int NOT NULL,
       [Year End] int NOT NULL,
       [ValidFrom] [datetime2](7) NOT NULL,
       [ValidTo] [datetime2](7) NULL,
 CONSTRAINT [PK_SatAirlineIDEffectivity] PRIMARY KEY NONCLUSTERED
(
       [AirlineIDHashKey] ASC,
       [LoadDate] ASC
) ON [INDEX]
) ON [DATA]
```

Because the satellite is dependent on the **HubAirLineID**, it includes the hash key from its parent hub and the **load date** of the record in the primary key. In addition, the table definition includes a **load end date** and the **record source** as technical fields. By doing so, it follows the definition of a standard Data Vault 2.0 satellite and treats the dates in the payload (**ValidFrom** and **ValidTo**) as ordinary descriptive fields, which can be changed by the source system at any time. This is called bi-temporal modeling [1].

The other two fields in the payload of the satellite, year start and year end, are the actual raw data fields from the source table. They have been used to calculate the **ValidFrom** and **ValidTo** fields in the loading statement of the effectivity satellite:

```
INSERT INTO DataVault.[raw].SatAirlineIDEffectivity (
       AirlineIDHashKey,
       LoadDate,
       LoadEndDate,
       RecordSource,
       [Year Start],
       [Year End],
       ValidFrom,
       ValidTo
)
SELECT DISTINCT
       stg.AirlineIDHashKey,
       stg.LoadDate,
       NULL,
       stg.RecordSource,
       stg.[Year Start],
       stg.[Year End],
       DATEFROMPARTS(stg.[Year Start], 1, 1),
       DATEFROMPARTS(stg.[Year End], 12, 31)
FROM
       StageArea.my.CarrierHistory stg
LEFT OUTER JOIN
       DataVault.[raw].SatAirlineIDEffectivity sat
       ON (stg.AirlineIDHashKey = sat.AirlineIDHashKey AND sat.LoadEndDate IS NULL)
WHERE
       ISNULL(stg.[Year Start], 0) != ISNULL(sat.[Year Start], 0)
       AND ISNULL(stg.[Year End], 9999) != ISNULL(sat.[Year End], 9999)
```

This statement aligns the data type of the integer-based year data in the source system to the required timestamp by applying the **DATEFROMPARTS** function on both **year start** and **year end** columns. Because this is a data type alignment only without recalculating the value, it counts as a hard rule and can be applied when loading the Raw Data Vault. However, in order to be able to trace errors, the original values are stored along with the aligned values, as well. Other recalculations should be avoided to include in the loading procedures of the Raw Data Vault and moved into the Business Vault or the loading processes of the Information Marts. If the source data changes, the satellite captures this change just as it does any other descriptive data.

Note that the above example relies on a table that has been artificially created and is based on the carrier history entity in the example MDS entity on the companion Web site. The MDS table contains multiple entries per carrier, which was simplified for demonstration purposes. The resulting table was placed in a custom schema.

12.1.9 DEALING WITH MISSING DATA

The problem with deleted business keys and relationships is that it is not always possible to detect actual deletes properly: under some circumstances, the source feed might not provide a business key that is actually in the source system. There are various reasons why a business key or other data is missing from the source:

- **Source system not in scope:** the source system simply isn't being loaded to the data warehouse, for example when the source system did not deliver its exported source files to the data warehouse (due to a different schedule, for example).
- **Technical error:** in other cases the data exists in the source system, but is not exported, for technical reasons. The export could fail because there is no disk space left on the target disk for the export.
- **Source filter:** a source filter could have prevented the export of the data into the source file, even though the data is in the source system. When using delta loads, for example, only new data is exported into the file. Data that hasn't changed at all is not being exported in such settings.
- **ETL loading error:** sometimes, data is not loaded into the staging area, even if the data is in the source files. This could be due to implementation errors in the ETL loading routines.

For these reasons, it is not guaranteed that a business key has been deleted from the source system just because it is not loaded into the stage area. However, we'd like to know which records have been actually deleted in order to mark them as such (so called "soft deletes") in the data warehouse. Typically, this happens in effectivity satellites on hubs and links, as described in the previous section. In order to do so, we need to distinguish between data that has been left out of the source file for accidental reasons or by user action: that is, the record has been actually deleted from the source system.

In the best case, the source system provides an audit trail that tells the data warehouse what has happened to the source data between loads. This is the case when change data capture (CDC) has been activated in the source system and the CDC data is provided to the data warehouse as an audit trail. In such a case, the approach as outlined in section 12.1.8 can be used to load effectivity satellites from these sources.

However, in many cases, such an audit trail is not provided by the source system. Instead, a deleted record just disappears from the source data. It is not possible to distinguish between data that has been not provided by the source system and truly deleted data.

In order to solve this issue, the naïve approach is to retrieve all keys from the hub table and check if the hub keys are provided by the staging area (Figure 12.64).

This assumes that the staging area provides a full load and not a delta load of the incoming data. If this is the case, the business key still exists in the source system and it is clear that it hasn't been deleted. On the other hand, if the staging area doesn't provide the key, it could be assumed that the key doesn't exist in the source system anymore and therefore marked as deleted in the accompanying effectivity satellite.

The problem with this approach is that it requires many lookups, especially if the hub table contains a lot of business keys. For each key, the lookup into the source table has to be executed. In order to detect a deleted business key in hubs or deleted relationship in links, the process has to find the data that meets the following criteria:

1. The business key from the hub (or the relationship from the link) is not available in the source
2. The business key is not already soft-deleted in the target hub or link.

However, because of the full-scan on the target table, this process becomes unsustainable when the data grows. Instead of checking to see whether each hub key is still in the source table, the process should avoid the full-scan on the target table.

There is no solution to distinguish the missing and deleted business keys with 100% security. The best approach is to make an educated guess. The **last seen date** helps to make such a guess. This column is added to hubs (and links) where needed and used to build an inverted index tree. It is used only when CDC or an audit trail is not available from the source system (Figure 12.65).

The last seen date is used to track when the business key stopped appearing on the feed to the data warehouse. By using this system-driven attribute, it is possible to minimize the data set that is required to scan. This approach works only on hubs and links. It is not intended to be used with satellite tables.

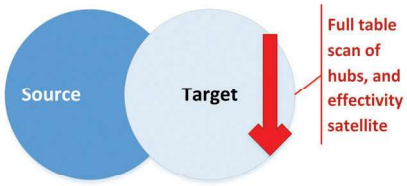

FIGURE 12.64

Full table scan required to detect deletes in the source system.

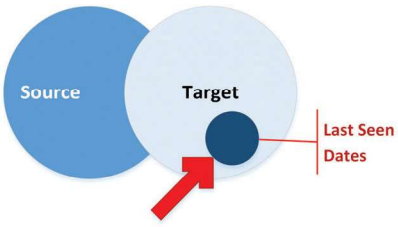

FIGURE 12.65

Introducing the last seen date in the loading process.

12.1.9.1 Data Aging

The last seen date helps to manage or reduce the table scan to a maintainable and viable set by introducing data aging (Figure 12.66).

There are four data sources in Figure 12.66: flights, passengers, carriers, and airports. No source provides an audit trail and might have missing data. To simplify the problem, each source provides only one business key in this example. In the first week, the flight system, the carriers and the airports system provide "their" business key to the data warehouse. In the second week, flights and airports keep providing the key, but the carrier key is gone. Instead, the passenger key is provided. In the third week, only the carrier key is provided, and all others are missing. In week four, only the airport key is delivered to the data warehouse.

The data warehouse needs to find out which of the keys have been deleted and disappeared from further loads and which business keys are only missing from some loads. The carrier and airport keys are such examples, but also the passenger key that will appear in week five again. The flight key, on the other hand, seems to be deleted, because it doesn't appear anymore. However, what if it appears in week eight?

The answer to this question cannot be given from the data warehouse team. Instead, the business has to answer this question. For each key, there might be different requirements. For example, the carrier key might be considered as deleted if it doesn't appear for three weeks. This is due to the fact that the source system is relatively unreliable delivering the data to the data warehouse. On the other hand, the flights system is very reliable and whenever a key has not been delivered in a load, it can be considered as being deleted from the source. It is also important if the source data is provided in full loads or delta loads. However, the same logic can be applied to delta loads: if a delta does not provide any data for a business key for several weeks, it can be assumed that the business key has been deleted (Figure 12.67).

In this figure, the business stated that flight numbers, which did not appear for three weeks, should be marked as deleted in the data warehouse in week 5. Such rules are only applied if the business decides to do so. Each key and each source system requires its own definition for data aging. The definition for each one should be provided by the business user and put into writing in the service level agreement between the data warehouse team and the business user.

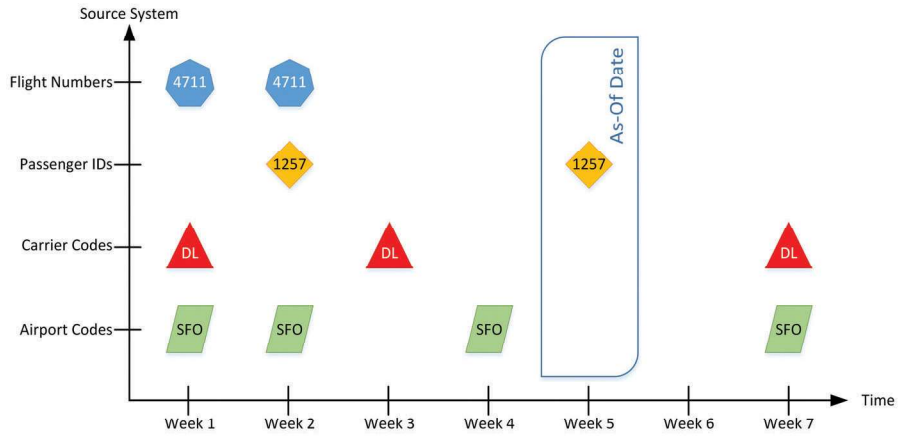

FIGURE 12.66

Data aging.

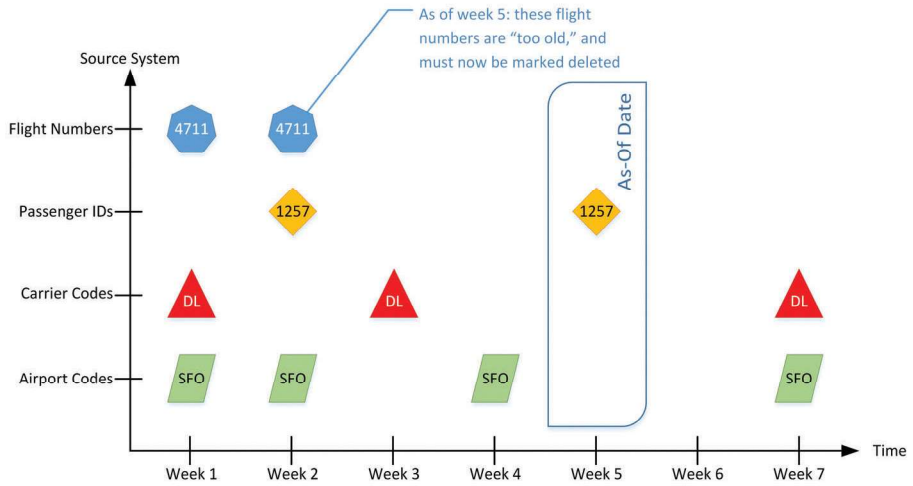

FIGURE 12.67

Mark business keys deleted.

If the business wants to avoid this discussion or cannot make a statement, the only alternative is to provide an audit trail to the data warehouse in order to detect deletes.

Using a load end represents a pragmatic approach to detect deltas on large data sets. However, it is a form of soft rule and therefore this approach violates the general recommendation that soft business rules should not be considered when loading the Raw Data Vault. If this should be avoided, status-tracking satellites could be used. They have been described in Chapter 5 and increase the performance by changing the update into an insert statement.

12.1.9.2 T-SQL Example

The last seen date is added to the hub structure as shown in the following DDL statement:

```
CREATE TABLE [raw].[HubFlightNum](
       [FlightNumHashKey] [char](32) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [LastSeenDate] [datetime2](7) NOT NULL,
       [Carrier] [nvarchar](2) NULL,
       [FlightNum] [smallint] NULL,
 CONSTRAINT [PK_HubFlightNum] PRIMARY KEY NONCLUSTERED
(
       [FlightNumHashKey] ASC
) ON [INDEX],
 CONSTRAINT [UK_HubFlightNum] UNIQUE NONCLUSTERED
(
       [Carrier] ASC,
       [FlightNum] ASC
) ON [INDEX]
) ON [DATA]
```

The **last seen date** has been added to the system-generated attributes of the hub table. Other than that, the table follows the DDL for hubs, presented in section 12.1.1. In some cases, it might be recommended

to create an index on the last seen date (first element of the index) and the hash key (second element of the index) to increase the performance of the table. This depends on the volume of data and if any partitioning schemes are used.

The following statement inserts new keys into the target hub:

```
INSERT INTO DataVault.[raw].HubFlightNum (
       FlightNumHashKey, LoadDate, RecordSource, LastSeenDate, Carrier, FlightNum
)
SELECT DISTINCT
       FlightNumHashKey, LoadDate, RecordSource, LoadDate, Carrier, FlightNum
FROM
       StageArea.bts.OnTimeOnTimePerformanceGD s
WHERE
       NOT EXISTS (SELECT
                     1
              FROM
                     DataVault.[raw].HubFlightNum h
              WHERE
                     s.Carrier = h.Carrier
                     AND s.FlightNum = h.FlightNum
              )
       AND LoadDate = '2003-10-01 00:00:00.000'
```

The last seen date for new keys, which are loaded to the hub, is set to the current load date. The WHERE clause is more complex than the statement for loading hubs presented in section 12.1.1, because of the composite business key.

After executing the previous statement, the following statement is executed in order to update the last seen date for those records that are already in the hub and are provided in the staging table:

```
UPDATE DataVault.[raw].HubFlightNum SET
       LastSeenDate = stg.LoadDate
FROM
       DataVault.[raw].HubFlightNum hub
INNER JOIN
       StageArea.bts.OnTimeOnTimePerformanceGD stg
ON
       stg.Carrier = hub.Carrier AND stg.FlightNum = hub.FlightNum
       AND hub.LastSeenDate < stg.LoadDate
WHERE
       stg.LoadDate = '2003-10-01 00:00:00.000'
```

This statement updates all records that have a last seen date that is older than the current load date (the load date currently loaded into the data warehouse). This is to reduce the number of updates on the target table.

Updating the last seen date is a separate process that doesn't affect the process that inserts new business keys into the target hub (which follows the standard process outlined in section 12.1.1).

12.2 LOADING REFERENCE TABLES

The purpose of reference tables is not only to simplify the Raw Data Vault 2.0 model but also the processes that deal with loading the data or using the data later on. The next sections describe multiple approaches to load reference data from the staging area into reference tables in the Raw Data Vault.

12.2.1 NO-HISTORY REFERENCE TABLES

If reference data should be loaded without taking care of the history, the loading process can be drastically simplified by using SQL views to create virtual reference tables. A similar approach was described in Chapter 11, Data Extraction, when staging master data from Microsoft Master Data Services (MDS) or any other master data management solution that is under control of the data warehouse team and primarily used for analytical master data. This approach can be used under the following conditions:

1. **History not required:** again, this solution is applicable for cases of nonhistorized reference tables only.
2. **Full load in staging area:** the source table in the staging area provides a full load and not a delta load.
3. **Same infrastructure:** the staging area is located on the same infrastructure as the data warehouse. If a different database server is used to house the staging area, the performance of the virtual reference tables could be impacted.
4. **Full control over staging area:** the staging area is under full control of the data warehouse team and the team decides about structural changes. The last thing that should happen in production is an uncontrolled update to the staging area that breaks a virtual reference table.
5. **Reference data in staging area is virtualized as well:** this condition rules out most applications but is important because the staging area should not be used as the central storage location. If reference data in the data warehouse layer is virtually depending on data in the staging area, the Data Vault 2.0 architecture has been violated.
6. **All required data is available:** in some cases, the source system loses old records (e.g., old countries). If this is OK, because old records are not required in the reference table, then this condition is negligible. However, because the data warehouse provides historic data, all codes referenced in satellites have to be dissolved by the reference table in the data warehouse layer.
7. **No data transformation required:** the data in the staging area is already in a format that requires no processing of soft business rules in order to prevent the execution of conditional logic when loading the Raw Data Vault.

If all these conditions are met, a virtual SQL view can be created in order to virtually provide the reference data to the users of the Raw Data Vault. This approach is typically used when providing reference data from an analytical MDM solution that is under control and managed by the data warehouse team. Such data is also staged virtually and centrally stored in the MDM application. The following DDL creates an example view that implements a nonhistorized reference table in the Raw Data Vault:

```
CREATE VIEW [raw].[RefRegion]
AS
SELECT
    [Code]
    ,[Name]
    ,[Abbreviation]
    ,[Sort Order]
    ,[External Reference]
    ,[Comments]
FROM
    [StageArea].[mds].[BTS_Region_DWH]
```

The view selects data from the table in the staging area, which is also a virtually provided staging view (refer to Chapter 11 for details). All columns are provided explicitly to avoid taking over unrequired columns but also to prevent taking over unforeseen changes to the underlying structure into the data warehouse. The view doesn't implement any soft business rules, but might implement hard business rules, such as data type alignment. It does however, brings the reference data from the staging area into the desired structure of a reference table, as discussed in Chapter 6, Advanced Data Vault Modeling.

This approach is most applicable for loading analytical master data from a master data management application such as Microsoft Master Data Services. Virtual reference tables are especially used in the agile Data Vault 2.0 methodology to provide the reference data as quickly as possible. If the business user agrees with the implemented functionality and materialization is required, the reference data can be materialized in a subsequent sprint, stretching the actual implementation of new functionality over multiple sprints.

12.2.1.1 T-SQL Example

In many other cases, especially if the data is already staged in the staging area, it should be materialized into the data warehouse layer to ensure that data is not spread over multiple layers. This decoupling from the staging area prevents any undesired side-effects if other parties change the underlying structure of the staging area. In such cases, the reference table is created in the data warehouse layer, for example by a statement such as the following:

```
CREATE TABLE [raw].[RefRegion](
       [Code] [nvarchar](2) NOT NULL,
       [Name] [nvarchar](250) NOT NULL,
       [Abbreviation] [nvarchar](2) NOT NULL,
       [Sort Order] [int] NOT NULL,
       [External Reference] [nvarchar](255) NULL,
       [Comments] [nvarchar](max) NULL,
 CONSTRAINT [PK_RefRegion] PRIMARY KEY CLUSTERED
(
       [Code] ASC
) ON [DATA]
) ON [DATA] TEXTIMAGE_ON [DATA]
```

The structure of the reference table follows the definition for nonhistorized reference tables outlined in Chapter 6. The primary key of the reference table consists of the **Code** column. Because this column holds a natural key instead of a hash key, the primary key uses a clustered index. There are multiple options for loading the reference table during the loading process of the Raw Data Vault. The most commonly used adds new and unknown reference codes from the staging area into the target reference table and updates records in the target that have changed in the source table. This way, no codes that could be used in any one of the satellites is lost. While it is not recommended to use the MERGE statement in loading the data warehouse, it is possible to load the reference table this way:

```
MERGE DataVault.[raw].RefRegion AS ref
USING (
       SELECT
              [Code],
              [Name],
              [Abbreviation],
              [Sort Order],
              [External Reference],
              [Comments]
```

```
        FROM
                [StageArea].[mds].[BTS_Region_DWH]
) AS stg ([Code], [Name], [Abbreviation], [Sort Order], [External Reference], [Comments])
ON (ref.[Code] = stg.[Code])
WHEN MATCHED THEN
        UPDATE SET
                ref.[Name] = stg.[Name],
                ref.[Abbreviation] = stg.[Abbreviation],
                ref.[Sort Order] = stg.[Sort Order],
                ref.[External Reference] = stg.[External Reference],
                ref.[Comments] = stg.[Comments]
WHEN NOT MATCHED THEN
        INSERT (
                [Code],
                [Name],
                [Abbreviation],
                [Sort Order],
                [External Reference],
                [Comments]
        )
        VALUES (
                stg.[Code], stg.[Name],
                stg.[Abbreviation],
                stg.[Sort Order],
                stg.[External Reference],
                stg.[Comments]
        );
```

Because the code column identifies the reference table, it becomes the search condition of the MERGE statement. If the code from the staging table is found in the target, the record in the reference table is updated. If it is unknown, it is inserted. If codes are deleted from the source system, they are ignored in order to preserve all codes in the reference table. Deletes are implemented by adding a WHEN NOT MATCHED BY SOURCE clause:

```
MERGE DataVault.[raw].RefRegion AS ref
USING (
        SELECT
                [Code],
                [Name],
                [Abbreviation],
                [Sort Order],
                [External Reference],
                [Comments]
        FROM
                [StageArea].[mds].[BTS_Region_DWH]
) AS stg ([Code], [Name], [Abbreviation], [Sort Order], [External Reference], [Comments])
ON (ref.[Code] = stg.[Code])
WHEN MATCHED THEN
        UPDATE SET
                ref.[Name] = stg.[Name],
                ref.[Abbreviation] = stg.[Abbreviation],
                ref.[Sort Order] = stg.[Sort Order],
                ref.[External Reference] = stg.[External Reference],
                ref.[Comments] = stg.[Comments]
```

```
WHEN NOT MATCHED BY TARGET THEN
       INSERT (
              [Code],
              [Name],
              [Abbreviation],
              [Sort Order],
              [External Reference],
              [Comments]
       )
       VALUES (
              stg.[Code], stg.[Name],
              stg.[Abbreviation],
              stg.[Sort Order],
              stg.[External Reference],
              stg.[Comments]
       )
WHEN NOT MATCHED BY SOURCE THEN DELETE;
```

The MERGE statement is generally not recommended to use in the loading processes of the data warehouse because of performance reasons and other issues with the MERGE statement on SQL Server [2]. Instead, the operations should be separated into individual statements to maintain performance. On the other hand, reference tables often have a relatively small size and performance doesn't become an issue. Therefore, using the MERGE statement might be simpler in some cases. If the reference table is large or performance becomes an issue, the statement should be separated.

12.2.2 HISTORY-BASED REFERENCE TABLES

History-based reference tables consist of two tables in fact (refer to Chapter 6, Advanced Data Vault Modeling, for more details about their definitions). The first table has the same structure as the no-history reference table and provides a list of codes and optionally some nonhistorized attributes of the codes in the list. The loading process is the same as described in the previous section.

12.2.2.1 T-SQL Example

The second table is a satellite that hangs off the reference table. The following statement creates a satellite on a reference table:

```
CREATE TABLE [raw].[RefSatRegion](
       [Code] [nvarchar](2) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [LoadEndDate] [datetime2](7) NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [HashDiff] [char](32) NOT NULL,
       [ID] [int] NOT NULL,
       [MUID] [uniqueidentifier] NOT NULL,
       [VersionName] [nvarchar](50) NOT NULL,
       [VersionNumber] [int] NOT NULL,
       [VersionFlag] [nvarchar](50) NULL,
       [Name] [nvarchar](250) NULL,
       [ChangeTrackingMask] [int] NOT NULL,
       [Abbreviation] [nvarchar](2) NULL,
       [Sort Order] [decimal](38, 0) NULL,
```

```
      [External Reference] [nvarchar](255) NULL,
      [Record Source] [nvarchar](100) NULL,
      [Comments] [nvarchar](4000) NULL,
      [EnterDateTime] [datetime2](3) NOT NULL,
      [EnterUserName] [nvarchar](100) NULL,
      [EnterVersionNumber] [int] NULL,
      [LastChgDateTime] [datetime2](3) NOT NULL,
      [LastChgUserName] [nvarchar](100) NULL,
      [LastChgVersionNumber] [int] NULL,
      [ValidationStatus] [nvarchar](250) NULL,
 CONSTRAINT [PK_RefSatRegion] PRIMARY KEY CLUSTERED
(
      [Code] ASC,
      [LoadDate] ASC
) ON [DATA]
) ON [DATA]
```

Instead of a hash key, this satellite uses the code to reference its parent table. This is because there is no hash key in the parent reference table due to the goal of reference tables to increase readability of the data. Because the hash key is not used, a clustered index is used for the primary key to improve performance during reads. This satellite uses a hash diff column to improve the column comparison when inserting new records.

Many of the satellite columns are already in the parent reference table. However, this example shows how to track the changes to the source system data in addition to the simple reference table without history, which is provided for the business users. In other cases, only the attributes, which are not used in the parent reference table, are added to the historizing satellite. The best approach is left to the data warehousing team.

In order to load the satellite table on the reference table, the following statement can be used:

```
INSERT INTO DataVault.[raw].RefSatRegion (
       Code,
       LoadDate,
       LoadEndDate,
       RecordSource,
       HashDiff,
       ID,
       MUID,
       VersionName,
       VersionNumber,
       VersionFlag,
       Name,
       ChangeTrackingMask,
       Abbreviation,
       [Sort Order],
       [External Reference],
       [Record Source],
       Comments,
       EnterDateTime,
       EnterUserName,
       EnterVersionNumber,
       LastChgDateTime,
       LastChgUserName,
       LastChgVersionNumber,
       ValidationStatus
)
```

```
SELECT DISTINCT
       stg.Code,
       stg.LoadDate,
       NULL,
       stg.RecordSource,
       stg.RegionHashDiff,
       stg.ID,
       stg.MUID,
       stg.VersionName,
       stg.VersionNumber,
       stg.VersionFlag,
       stg.Name,
       stg.ChangeTrackingMask,
       stg.Abbreviation,
       stg.[Sort Order],
       stg.[External Reference],
       stg.[Record Source],
       stg.Comments,
       stg.EnterDateTime,
       stg.EnterUserName,
       stg.EnterVersionNumber,
       stg.LastChgDateTime,
       stg.LastChgUserName,
       stg.LastChgVersionNumber,
       stg.ValidationStatus
FROM
       StageArea.mds.BTS_Region_DWH stg
LEFT OUTER JOIN
       DataVault.[raw].RefSatRegion sat
       ON (stg.Code = sat.Code AND sat.LoadEndDate IS NULL)
WHERE
       (sat.HashDiff IS NULL OR stg.RegionHashDiff != sat.HashDiff)
       AND stg.LoadDate = '2015-02-16 11:31:22.537'
```

This statement uses the same loading approach as standard Data Vault satellites, described in section 12.1.4. The **left outer join** is based on the satellite's parent key, consisting of the reference **code** and the **load date**. The column compare in this statement is based on the **hash diff** to improve loading performance. The statement has to be executed for each load date in the staging area, replacing the hard-coded load date in the shown SQL statement by a variable.

This satellite also requires being end-dated afterwards, similar to the process described in section 12.1.5.

12.2.3 CODE AND DESCRIPTIONS

A code and descriptions reference table provides a convenient method to deal with a large number of reference code groups. Typically, source systems provide various groups with only a couple of codes (refer to Chapter 6). The number of code groups that a source system provides can go to several hundreds with large systems. To avoid creating many reference tables with only a small number of codes, a common practice is to use only one reference table to capture code and descriptions. This table is a generalized table that provides only standard attributes, such as descriptions, sort orders, etc. The standard attributes are defined by the data warehouse team and provided for each group. If additional, nonstandard attributes are required for a code group, an individual reference table is required (see sections 12.2.1 and 12.2.2).

If the code and descriptions table should provide no history, it is possible to provide the table as a virtual SQL view, as standard, nonhistory reference tables. The DDL for such a virtual code and descriptions table is presented here:

```
CREATE VIEW [raw].[RefCodes] AS
SELECT
      'BTS.Region' AS [Group]
      ,[Code]
      ,[Name]
      ,[Abbreviation]
      ,[Sort Order]
      ,[External Reference]
      ,[Record Source]
      ,[Comments]
  FROM [StageArea].[mds].[BTS_Region_DWH]
  UNION ALL
  SELECT
      'FAA.AircraftCategory' AS [Group]
      ,[Code]
      ,[Name]
      ,[Abbreviation]
      ,[Sort Order]
      ,[External Reference]
      ,[Record Source]
      ,[Comments]
  FROM [StageArea].[mds].[FAA_AircraftCategory_DWH]
  UNION ALL
  SELECT
      'FAA.AircraftType' AS [Group]
      ,[Code]
      ,[Name]
      ,[Abbreviation]
      ,[Sort Order]
      ,[External Reference]
      ,[Record Source]
      ,[Comments]
  FROM [StageArea].[mds].[FAA_AircraftType_DWH]
```

In this example, the view creates a UNION ALL over multiple staging tables to create a code and descriptions table. In other cases, the source system provides the table including all available groups. The problem with the union is that adding many staging tables increases the complexity of the view. The **record source** is not required but might be helpful to trace the source of reference data in the resulting view. Again, the **record source** column is used by the data warehouse team for debugging purposes.

Note that the reference table should not consolidate reference codes from multiple sources. This would require business logic, which is applied after loading the raw data. Instead, it should provide descriptive information for codes used in the raw data that can be used to create business rules that transform the incoming raw data into useful information. Therefore, it is possible that similar groups exist in this table. If the codes in the group are not changed across multiple source systems, for example

due to modifications to the group, multiple groups are used to keep all possible reference codes from the source systems. Therefore, an integration doesn't take place at this point.

12.2.3.1 T-SQL Example

The approach to virtually providing code and description tables has the same requirements as outlined in section 12.2.1. If any of these conditions are not met, or if the complexity of the view is becoming too high, the table could be materialized using the following DDL statement:

```
CREATE TABLE [raw].[RefCodes](
       [Group] [nvarchar](50) NOT NULL,
       [Code] [nvarchar](250) NOT NULL,
       [Name] [nvarchar](250) NOT NULL,
       [Abbreviation] [nvarchar](2) NULL,
       [Sort Order] [int] NOT NULL,
       [External Reference] [nvarchar](255) NULL,
       [Comments] [nvarchar](max) NULL,
       [Record Source] [nvarchar](100) NOT NULL,
 CONSTRAINT [PK_RefCodes] PRIMARY KEY CLUSTERED
(
       [Group] ASC,
       [Code] ASC
) ON [DATA]
) ON [DATA] TEXTIMAGE_ON [DATA]
```

As all other reference tables, the code and descriptions table relies on a clustered primary key because of the natural key in the **group** and **code** columns. It is possible to load the code and descriptions table using a MERGE statement again, but it becomes much more complex, due to the multiple source tables in the staging area:

```
MERGE DataVault.[raw].RefCodes AS ref
USING (
      SELECT
        'BTS.Region' AS [Group]
      ,[Code]
      ,[Name]
      ,[Abbreviation]
      ,[Sort Order]
      ,[External Reference]
      ,[Record Source]
      ,[Comments]
  FROM [StageArea].[mds].[BTS_Region_DWH]
  UNION ALL
  SELECT
        'FAA.AircraftCategory' AS [Group]
      ,[Code]
      ,[Name]
      ,[Abbreviation]
      ,[Sort Order]
      ,[External Reference]
      ,[Record Source]
      ,[Comments]
  FROM [StageArea].[mds].[FAA_AircraftCategory_DWH]
  UNION ALL
  SELECT
```

```
          'FAA.AircraftType' AS [Group]
        ,[Code]
        ,[Name]
        ,[Abbreviation]
        ,[Sort Order]
        ,[External Reference]
        ,[Record Source]
        ,[Comments]
    FROM [StageArea].[mds].[FAA_AircraftType_DWH]
) AS stg ([Group], [Code], [Name], [Abbreviation], [Sort Order], [External Reference],
[Record Source], [Comments])
ON (ref.[Group] = stg.[Group] AND ref.[Code] = stg.[Code])
WHEN MATCHED THEN
       UPDATE SET
              ref.[Name] = stg.[Name],
              ref.[Abbreviation] = stg.[Abbreviation],
              ref.[Sort Order] = stg.[Sort Order],
              ref.[External Reference] = stg.[External Reference],
              ref.[Record Source] = stg.[Record Source],
              ref.[Comments] = stg.[Comments]
WHEN NOT MATCHED BY TARGET THEN
       INSERT (
              [Group],
              [Code],
              [Name],
              [Abbreviation],
              [Sort Order],
              [External Reference],
              [Record Source],
              [Comments]
       )
       VALUES (
              stg.[Group], stg.[Code],
              stg.[Name],
              stg.[Abbreviation],
              stg.[Sort Order],
              stg.[External Reference],
              stg.[Record Source],
              stg.[Comments]
       );
```

This MERGE statement does not delete any codes in the target code and description table. The UNION ALL is still included in the SELECT clause in order to avoid dealing with parallel running of INSERT statements on the same table, for example by adding locks on the target table in the loading processes.

12.2.4 CODE AND DESCRIPTIONS WITH HISTORY

The last reference table example in this chapter is focused on historized code and description tables. The implementation follows the same approach as standard reference tables and involves a satellite on the code and description table for keeping historized attributes.

12.2.4.1 T-SQL Example

The following DDL is used to create such a satellite on the code and descriptions table:

```
CREATE TABLE [raw].[RefSatCodes](
       [Group] [nvarchar](50) NOT NULL,
       [Code] [nvarchar](2) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [LoadEndDate] [datetime2](7) NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [HashDiff] [char](32) NOT NULL,
```

```
	[ID] [int] NOT NULL,
	[MUID] [uniqueidentifier] NOT NULL,
	[VersionName] [nvarchar](50) NOT NULL,
	[VersionNumber] [int] NOT NULL,
	[VersionFlag] [nvarchar](50) NULL,
	[Name] [nvarchar](250) NULL,
	[ChangeTrackingMask] [int] NOT NULL,
	[Abbreviation] [nvarchar](2) NULL,
	[Sort Order] [decimal](38, 0) NULL,
	[External Reference] [nvarchar](255) NULL,
	[Record Source] [nvarchar](100) NULL,
	[Comments] [nvarchar](4000) NULL,
	[EnterDateTime] [datetime2](3) NOT NULL,
	[EnterUserName] [nvarchar](100) NULL,
	[EnterVersionNumber] [int] NULL,
	[LastChgDateTime] [datetime2](3) NOT NULL,
	[LastChgUserName] [nvarchar](100) NULL,
	[LastChgVersionNumber] [int] NULL,
	[ValidationStatus] [nvarchar](250) NULL,
 CONSTRAINT [PK_RefSatCodes] PRIMARY KEY CLUSTERED
(
	[Group] ASC,
	[Code] ASC,
	[LoadDate] ASC
) ON [DATA]
) ON [DATA]
```

The DDL closely follows the DDL in section 12.2.2 but includes the **group** column in the primary key of the satellite, in addition to the **code** and **load date**. The satellite is loaded similarly using the following INSERT statement:

```
INSERT INTO DataVault.[raw].RefSatCodes (
	[Group],
	Code,
	LoadDate,
	LoadEndDate,
	RecordSource,
	HashDiff,
	ID,
	MUID,
	VersionName,
	VersionNumber,
	VersionFlag,
	Name,
	ChangeTrackingMask,
	Abbreviation,
	[Sort Order],
	[External Reference],
	[Record Source],
	Comments,
	EnterDateTime,
	EnterUserName,
	EnterVersionNumber,
	LastChgDateTime,
	LastChgUserName,
	LastChgVersionNumber,
	ValidationStatus
)
```

```
SELECT DISTINCT
    'BTS.Region' AS [Group],
       stg.Code,
       stg.LoadDate,
       NULL,
       stg.RecordSource,
       stg.RegionHashDiff,
       stg.ID,
       stg.MUID,
       stg.VersionName,
       stg.VersionNumber,
       stg.VersionFlag,
       stg.Name,
       stg.ChangeTrackingMask,
       stg.Abbreviation,
       stg.[Sort Order],
       stg.[External Reference],
       stg.[Record Source],
       stg.Comments,
       stg.EnterDateTime,
       stg.EnterUserName,
       stg.EnterVersionNumber,
       stg.LastChgDateTime,
       stg.LastChgUserName,
       stg.LastChgVersionNumber,
       stg.ValidationStatus
FROM
       StageArea.mds.BTS_Region_DWH stg
LEFT OUTER JOIN
       DataVault.[raw].RefSatCodes sat
       ON ('BTS.Region' = sat.[Group] AND stg.Code = sat.Code AND sat.LoadEndDate IS
NULL)
WHERE
       (sat.HashDiff IS NULL OR stg.RegionHashDiff != sat.HashDiff)
       AND stg.LoadDate = '2015-02-16 11:31:22.537'
UNION ALL
SELECT DISTINCT
       'FAA.AircraftCategory' AS [Group],
       stg.Code,
       stg.LoadDate,
       NULL,
       stg.RecordSource,
       stg.AircraftCategoryHashDiff,
       stg.ID,
       stg.MUID,
       stg.VersionName,
       stg.VersionNumber,
       stg.VersionFlag,
       stg.Name,
       stg.ChangeTrackingMask,
       stg.Abbreviation,
       stg.[Sort Order],
       stg.[External Reference],
       stg.[Record Source],
       stg.Comments,
       stg.EnterDateTime,
       stg.EnterUserName,
       stg.EnterVersionNumber,
       stg.LastChgDateTime,
       stg.LastChgUserName,
       stg.LastChgVersionNumber,
       stg.ValidationStatus
```

```
FROM
       StageArea.mds.FAA_AircraftCategory_DWH stg
LEFT OUTER JOIN
       DataVault.[raw].RefSatCodes sat
       ON ('FAA.AircraftCategory' = sat.[Group] AND stg.Code = sat.Code AND
sat.LoadEndDate IS NULL)
WHERE
       (sat.HashDiff IS NULL OR stg.AircraftCategoryHashDiff != sat.HashDiff)
       AND stg.LoadDate = '2015-02-16 11:31:22.537'
UNION ALL
SELECT DISTINCT
       'FAA.AircraftType' AS [Group],
       stg.Code,
       stg.LoadDate,
       NULL,
       stg.RecordSource,
       stg.AircraftTypeHashDiff,
       stg.ID,
       stg.MUID,
       stg.VersionName,
       stg.VersionNumber,
       stg.VersionFlag,
       stg.Name,
       stg.ChangeTrackingMask,
       stg.Abbreviation,
       stg.[Sort Order],
       stg.[External Reference],
       stg.[Record Source],
       stg.Comments,
       stg.EnterDateTime,
       stg.EnterUserName,
       stg.EnterVersionNumber,
       stg.LastChgDateTime,
       stg.LastChgUserName,
       stg.LastChgVersionNumber,
       stg.ValidationStatus
FROM
       StageArea.mds.FAA_AircraftType_DWH stg
LEFT OUTER JOIN
       DataVault.[raw].RefSatCodes sat
       ON ('FAA.AircraftType' = sat.[Group] AND stg.Code = sat.Code AND sat.LoadEndDate
IS NULL)
WHERE
       (sat.HashDiff IS NULL OR stg.AircraftTypeHashDiff != sat.HashDiff)
       AND stg.LoadDate = '2015-02-16 11:31:22.537'
```

Again, this statement uses the precalculated hash diff attribute to improve the performance of the column comparisons. It should be run for every load date in the staging area as it is required by all loading statements presented in this chapter.

This satellite requires being end-dated as described in section 12.1.5.

12.3 TRUNCATING THE STAGING AREA

Once the data has been loaded into the Raw Data Vault, the staging area should be cleaned up. This is because the storage consumption of the staging area should be kept to a minimum to reduce maintenance overhead and in order to improve the performance of subsequent loads of the data warehouse.

If the staging area contains only data that needs to be loaded into the data warehouse, no additional processes are required to manage which batches have been loaded into the data warehouse.

There are two options to truncate the staging area. Depending on the frequency of the incoming batches, it is feasible to use a TRUNCATE TABLE statement or it requires deleting only data with specific load dates only:

- **Truncate table:** the TRUNCATE TABLE statement can be used if it is guaranteed that all data from the staging table has been loaded into the Raw Data Vault and no new data has arrived between the load of the Raw Data Vault and the TRUNCATE TABLE statement. This is often the case when data is delivered only on a daily schedule but not more often.
- **Delete specific records:** if the staging area receives multiple batches over the day and the data warehouse team cannot guarantee that all data has been loaded into the Raw Data Vault, the TRUNCATE TABLE statement might accidentally delete data that is already in the staging area but not in the Raw Data Vault yet. Consider a delivery schedule of 15 minutes: the staging process loads the data from a source system in intervals of 15 minutes, for example by loading it directly off a relational database. In most cases, the data is loaded into the Raw Data Vault within this 15-minute cycle. But the data warehouse team cannot guarantee that this is the case in all loads. For example, during peak hours when other data sources are loaded as well, the loading process into the Raw Data Vault might take more time. If a TRUNCATE TABLE statement is used each time the loading process of the Raw Data Vault completes, data loss might occur in such cases. Therefore, only the records with the load date that has been loaded into the Raw Data Vault should be deleted.
- **Delete specific partitions:** because the DELETE statement is much slower than the TRUNCATE TABLE statement, data warehouse teams often rely on table partitioning to delete old records from the staging area. The tables in the staging area are partitioned on the load date. Whenever a batch has been completed, the partition with the corresponding load date is deleted as a whole.

These three options are typical examples of how to delete the data that has been processed by the Raw Data Vault loading processes from the staging area. By doing so, the staging area is kept smaller and more manageable.

REFERENCES

[1] Date C. An introduction to database systems. 7th ed. Reading, Menlo Park, New York: Addison-Wesley-Longman; 2000.

[2] Bertrand A. Use caution with SQL Server's MERGE statement, 2013, MSSQLTips.com website, available from http://www.mssqltips.com/sqlservertip/3074/use-caution-with-sql-servers-merge-statement/.

CHAPTER

IMPLEMENTING DATA QUALITY

13

The goal of Data Vault 2.0 loads is to cover all data, regardless of its quality ("the good, the bad, and the ugly"). All data that is provided by the source systems should be loaded into the enterprise data warehouse, every time the source is in scope. In particular, these should be avoided:

- **Load bad data into an error file:** if data cannot be accepted into the Raw Data Vault for any reason, load it into the error mart.
- **Filter data because of technical issues:** duplicate primary keys (in the source system), violated NOT NULL constraints, missing indices or problems with joins should not prevent the raw data from being loaded into the Raw Data Vault. Adjust your loading procedures for these issues.
- **Prohibit NULL values:** instead of preventing NULL values in the Raw Data Vault, make sure to remove NOT NULL constraints from the Data Vault tables or set a default value.

With all the rules for acceptance of data relaxed, what if bad or ugly data is loaded into the data warehouse? At the very least, it should not be presented without further processing to the business user to be used in the decision-making process. This chapter covers the best practices for dealing with low data quality.

13.1 BUSINESS EXPECTATIONS REGARDING DATA QUALITY

Business users expect that the data warehouse will present correct information of high quality. However, data quality is a subjective concept. Data that is correct for one business user might be wrong for another with different requirements or another understanding of the business view. There is no "golden copy" or "single version of the truth" in data warehousing [1]. Just consider the business user who wants to compare reports from the data warehouse with the reports from an operational system, including all the calculations, which might differ. In many cases, data warehouse projects divert from the calculations and aggregations in operational systems due to enterprise alignment or to overcome errors in the source system. This is not a desired solution, but is reality grounded in the fact that the data warehouse often fixes issues that should have been fixed in the source system or the business processes. In any case, the data warehouse should provide both "versions of the truth." That's why Data Vault 2.0 focuses on the "single version of the facts."

The expectations regarding data quality should be distinguished by two categories [1]:

- **Data quality expectations:** these are expressed by rules, which measure the validity of the data values. This includes the identification of missing or unusable data, the description of data in conflict, the determination of duplicate records and the exploration of missing links between the data [1].
- **Business expectations:** these are measures related to the performance, productivity, and efficiency of processes. Typical questions raised by the business are focused on the throughput

decreased due to errors, the percentage of time that is spent in order to rework failed processes, and the loss in value of transactions that failed because of wrong transactions [1].

In order to meet the business expectations regarding data quality, both expectations have to be brought in line. The conformance to business expectations and its corresponding business value should be set in relation to the conformance to data quality expectations. This can provide a framework for data quality improvements [1].

Another question is where these data quality improvements should take place. Typically, there are different types of errors in a data warehouse and there are recommended practices for fixing and creating an alignment between the expectations of the business and the data quality expectations:

- **Data errors:** the best approach is to fix data errors in the source application and the business process that generates the data. Otherwise, the root cause of the error has not been fixed and more errors will emerge and need to be fixed later in a more costly approach.
- **Business rule errors:** this is an error that actually needs to be fixed in the data warehouse because the business rules, which are implemented in the Business Vault or in the loading routines for the information marts, are erroneous.
- **Perception errors:** if the business has a wrong perception of the data produced by the business process, the requirements for the data warehouse should be changed in most cases.

Because the data is owned by the business and IT should only be responsible for delivering it back to the business in an aggregated fashion, the IT should reduce the amount of work dedicated to implementing business rules. Instead, this responsibility should be given to business users as much as possible. This concept, called managed self-service BI, has been introduced in Chapter 2, Scalable Data Warehouse Architecture. If this approach is followed, then IT only governs the process, and provides the raw data for the business user to implement their business rules. The responsibility to fix business rule errors moves to the business user, in effect.

13.2 THE COSTS OF LOW DATA QUALITY

When data of low quality is loaded into the data warehouse, it incurs costs often overseen by business users. These costs can be put into several categories:

- **Customer and partner satisfaction:** in some cases, especially if external parties also use the data warehouse, low data quality can affect customer satisfaction [2]. For example, in some cases, the data warehouse is used to create quarterly invoices or financial statements to external partners. If the data on these statements is wrong, it can seriously damage the business relationship between customers or partners.[3]
- **Regulatory impacts:** in other cases, the data warehouse is used to generate statements required by government regulations or the law, industry bodies or internal units to meet self-imposed policies [1]. If these reports are wrong, and if you know it, you might end up in court. There are industries that have a large number of regulations to be met, for example the USA PATRIOT Act, Sarbanes-Oxley Act and the Basel II Accords in the financial industry [1].
- **Financial costs:** improving the data quality might cost substantial amounts of money. However, not improving the data quality and making wrong decisions will cost multiple times more [2]

This includes increased operating costs, decreased revenues, reduction or delays in cash flow, increased penalties, fines or other charges [1].

- **Organizational mistrust:** if end-users know that a data warehouse provides erroneous reports, they try to avoid the system by using the operational system to pull the reports or build their own silo data warehouse instead of using the enterprise data warehouse system [2] Therefore, providing reports with high data quality is a requirement for any successful enterprise data warehouse strategy.
- **Re-engineering:** if the data warehouse provides reports with low data quality and the organization cannot fix the issues (either in the source system or the data warehouse itself), many organizations decide to fail the project and start from the beginning [2].
- **Decision-making:** delivering bad data, even temporarily, provides a serious problem for business managers in meeting their goals. Many of their decisions are based on the information provided by the data warehouse. If this information is wrong, they cannot make the right decisions and lose time in meeting the goals set by higher management. Even if the data warehouse provides corrected information some months later in the year, the time is gone for the managers to fix the decisions made in the past, which were based on wrong assumptions [2].
- **Business strategy:** in order to achieve organizational excellence, top management requires outstanding data [2].

The cost of low data quality outweighs the cost of assessments, inspection and fixing the issue by multiple times. Therefore, it is advisable to improve the data quality before information in reports is given to business users. The best place for such improvements lies in the business processes and the operational systems to achieve total quality. However, if this is not possible, the data warehouse can be used for fixing bad data as well.

13.3 THE VALUE OF BAD DATA

In legacy data warehousing, bad data is fixed before it is loaded into the data warehouse. The biggest drawback of this approach is that the original data is not available for analysis anymore, at least not easily. This prevents any analysis for understanding the gap between business expectations, as outlined in the previous section, and the data quality of the source systems (Figure 13.1).

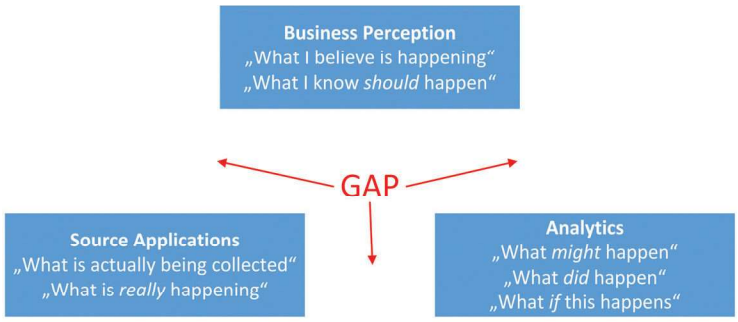

FIGURE 13.1

Gap analysis.

During the gap analysis presented in Figure 13.1, the business users explain to the data warehouse team what they believe happens in the business and what they think should happen in a perfect process. They also tell the data warehouse team how the data used in the business processes should be collected from the source systems.

But when the data warehouse team actually loads the data from the source systems, they often find these expectations to be broken because the source systems provide the truth about the data being collected and what is really happening from a data perspective. In addition, the source application provides insights into the process layer of the enterprise and often shows which processes are broken from a business perspective.

These are the two "versions of the truth." Both sides are right: the business has an expectation about the business processes that should be in place and the source systems are right because they show what kind of data (and of what quality) is actually being tracked, because of the actual business processes in place. Between both versions is a gap that should be closed.

A good data warehouse solution should assist the business in this gap analysis. It should be possible to discover where the requirements for the current business views or beliefs don't match what is being collected or being run. If the data warehouse team wants to help the business see the gaps, it is required to show them the bad data and help the business to see problems in their applications in order to correct them. Therefore, the gap analysis involves multiple layers, which are presented in Figure 13.2.

The business requirements represent the business expectations, while the raw data represents the source system data. The information layer provides altered, matched and integrated information in the enterprise data warehouse and in the information marts.

Sourcing raw data, including bad and ugly data, provides some values to the data warehouse:

- If problems are found (now or in the future) it is possible to handle them when the raw data is available.
- If the data is translated or interpreted before loaded into the data warehouse, a lot of gaps are sourced. These gaps are represented by the "fixed" data.
- Self-service BI isn't possible without IT involvement which is required to take back some of the interpretations of the raw data.
- It is possible to push more business rules into front-end tools for managed self-service BI usage because all the raw data is still available.
- IT is not responsible for interpreting the raw data, except the programming that is required.

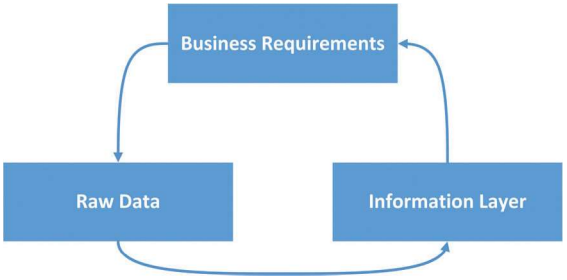

FIGURE 13.2

Layers of gap analysis.

Having the bad data in the data warehouse helps to reconcile integrated and altered data with the source systems and with business requirements. This is the reason why bad data, and ugly-looking data, is kept in the data warehouse and not modified, transformed or fixed in any sense on the way into the data warehouse.

13.4 DATA QUALITY IN THE ARCHITECTURE

Data quality routines correct errors, complete raw data or transform it to provide more business value. This is exactly the purpose of soft business rules. Therefore, data quality routines are considered as soft business rules in the reference architecture of Data Vault 2.0 (Figure 13.3).

Just like soft business rules, data quality routines change frequently: for example, if the data quality improvement is based on fixed translation rules, kept in a knowledge database, these rules evolve over time. And if a data mining approach is used to correct errors, inject missing data, or match entities, there are no fixed rules. Instead, the data-mining algorithm can adapt to new data and is often trained on a regular basis.

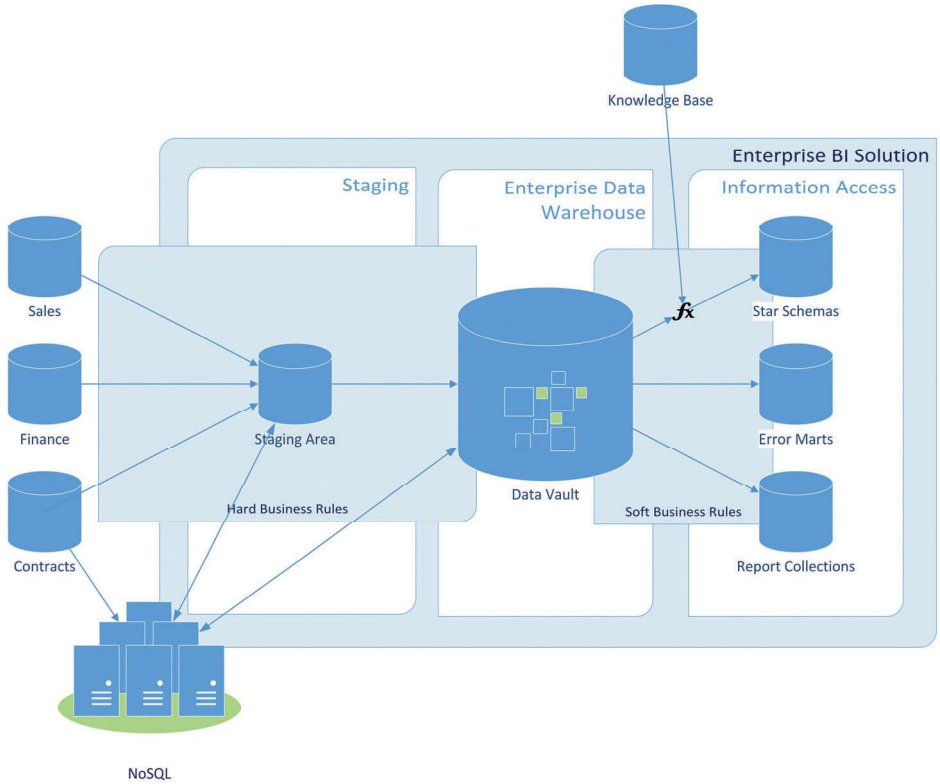

FIGURE 13.3

Data quality in the Data Vault 2.0 architecture.

Because soft business rules are implemented in the Business Vault or in the loading processes of the Information Marts, any data quality routines should be implemented in these places as well. By implementing data quality as soft business rules, the incoming raw data is not modified in any way and remains in the enterprise data warehouse for further analysis. If the data quality rules change, or new knowledge regarding the data is obtained, it is possible to adjust the data quality routines without reloading any previous raw data. The same is true if the business expectations change and with them, the understanding of high quality data.

The concept of multiple Information Marts is also helpful if there are multiple views of high quality data. In some cases, information consumers want to use information processed with enterprise-agreed data corrections. Other information consumers have their own understanding of data quality and the required corrections to the incoming data. And another type of user wants to use the raw data because they are interested in analyzing erroneous data from the operational system. In all these cases, individual information marts can be used to satisfy each type of information consumer and their individual requirements. It is also possible for information consumers in a self-service BI approach to mix and match the data quality routines as they see fit for their given task. This requires that data quality routines be implemented in the Business Vault if possible, so the cleansed results can be reused for multiple information marts. This is why the examples in this chapter are all based on the Business Vault. However, it is also valid to directly load the data (virtualized or materialized) into the target Information Mart.

13.5 CORRECTING ERRORS IN THE DATA WAREHOUSE

The correction of errors in the data warehouse represents only a suboptimal choice. The best choice is to fix data errors in the information processes of the business and in the operational systems. The goal is to eliminate not the errors, but the need to fix them. This can be achieved by improving the information processes and by designing quality into these processes to prevent any defects that require fixing [3].

While this represents the best choice, fixing the data in the source systems is often rejected as out-of-scope. Instead, erroneous data is loaded into the data warehouse and the business expects the errors to be fixed on the way downstream towards the end-user. This requires costly data correction activity, which cannot be complete. In many cases, not all existing errors are detected by data quality software and new errors are introduced. The resulting data set, which is propagated to the end-user, contains valid data from the source, along with fixed data, undetected errors and newly introduced errors [3].

Because of these limitations, the approach described in this chapter ensures that the raw data is kept in the data warehouse. By doing so, it is possible to fix additional issues that are found later in the process or to take back any erroneous modifications of the raw data made earlier. Typically, the following errors can be fixed in the business intelligence system [3]:

- **Transform, enhance, and calculate derived data:** the calculation of derived data is often necessary when the raw data is not in the form required by the business user. The

transformation increases the value of the information and is considered to be an enhancement of the data [3].

- **Standardization of data:** standardized data is required by the business user to communicate effectively with peers. It also involves standardizing the data formats [3].
- **Correct and complete data:** in some cases, the data from the operational system is incorrect or incomplete and cannot be fixed at the root of the failure. In this suboptimal case, the data warehouse is responsible for fixing the issue on the way towards the business user [3].
- **Match and consolidate data:** matching and consolidating data is often required when dealing with multiple data sources that produce the same type of data and duplicate data is involved. Examples include customer or passenger records, product records and other cases of operational master data [3].
- **Data quality tagging:** another common practice is to provide information about the data quality to downstream users [4].

Note that these "corrections" are not directly applied on the raw data, i.e., in form of updates. Instead, these modifications are implemented after retrieving the raw data from the Raw Data Vault and loading the data into the Business Vault or information mart layers where the cleansed data is materialized or provided in a virtual form.

The next sections describe the best practices for each case and provide an example using T-SQL or SSIS (whatever fits best). However, understand that the best case is to fix these issues in the operational system or in the business process that generates the data. Fixing these issues in the data warehouse represents only a suboptimal choice. Too often, this is what happens in reality. Data quality tagging is shown in section 13.8.3.

13.6 TRANSFORM, ENHANCE AND CALCULATE DERIVED DATA

Raw data from source systems often have data quality issues, including [4,5]:

- **Dummy values:** the source might compensate for missing values by the use of default values.
- **Reused keys:** business keys or surrogate keys are reused in the source system, which might lead to identification issues.
- **Multipurpose fields:** fields in the source database are overloaded and used for multiple purposes. It requires extra business logic to ensure the integrity of the data or use it.
- **Multipurpose tables:** similarly, relational tables can be used to store different business entities, for example, both people and corporations. These tables contain many empty columns because many columns are only used by one type of entity and not the others. Examples include the first and last name of an individual person versus the organizational name.
- **Noncompliance with business rules:** often, the source data is not in conformance with set business rules due to a lack of validation. The values stored in the source database might not represent allowed domain values.

- **Conflicting data from multiple sources:** a common problem when loading data from multiple source systems is that the raw data might be in conflict. For example, the delivery address of a customer or the spelling of the customer's first name ("Dan" versus "Daniel") might be different.
- **Redundant data:** some operational databases contain redundant data, primarily because of data modeling issues. This often leads to inconsistencies in the data, similar to the conflicts from multiple source systems (which are redundant data, as well).
- **Smart columns:** some source system columns contain "smart" data, that is data with structural meaning. This is often found in business keys (review smart keys in Chapter 4, Data Vault 2.0 Modeling) but can be found in descriptive data as well. XML and JSON encoded columns are other examples of such smart columns, because they also provide structural meaning.

Business users often take advantage of transforming this data into more valuable data. Combining conflicting data might provide actionable insights to business users [1]. For example, when screening passengers for security reasons, context information is helpful: where did the passenger board the plane, where is the passenger heading to, what is the purpose of the trip, etc. It is also helpful to integrate information from other data sources, such as intelligence and police records about the passenger, to identify potential security threats. The value of the raw data increases greatly when such information is added in an enhancement process [1].

Because of this, organizations often purchase additional data from external information suppliers [6]. There are various types of data that can be added to enhance the data [7]:

- **Geographic information:** external information suppliers can help with address standardization and provide geo-tagging, including geographic coordinates, regional coding, municipality, neighborhood mapping and other kinds of geographic data.
- **Demographic information:** a large amount of demographic data can be bought, even on a personal or individual enterprise level, including age, martial status, gender, income, ethnic coding or the annual revenue, number of employees, etc.
- **Psychographic information:** these enhancements are used to categorize populations regarding their product and brand preferences and their organizational memberships (for example, political parties). It also includes information about leisure activities, vacation preferences and shopping time preferences.

There is also business value in precalculating or aggregating data for frequently asked queries [6].

13.6.1 T-SQL EXAMPLE

Computed satellites provide a good option for transforming, enhancing or calculating derived values. The following DDL creates a virtual computed satellite, which calculates derived columns based on the **Distance** column:

```
CREATE VIEW [biz].[TSatFlight] AS
SELECT
       [FlightHashKey]
      ,[LoadDate]
      ,'SR4711' AS RecordSource
      ,[Year]
      ,[Quarter]
      ,[Month]
      ,[DayOfMonth]
      ,[DayOfWeek]
      ,[CRSDepTime]
      ,[DepTime]
      ,[DepDelay]
      ,[DepDelayMinutes]
      ,[DepDel15]
      ,[DepartureDelayGroups]
      ,[DepTimeBlk]
      ,[TaxiOut]
      ,[WheelsOff]
      ,[WheelsOn]
      ,[TaxiIn]
      ,[CRSArrTime]
      ,[ArrTime]
      ,[ArrDelay]
      ,[ArrDelayMinutes]
      ,[ArrDel15]
      ,[ArrivalDelayGroups]
      ,[ArrTimeBlk]
      ,[Cancelled]
      ,[CancellationCode]
      ,[Diverted]
      ,[CRSElapsedTime]
      ,[ActualElapsedTime]
      ,[AirTime]
      ,[Flights]
      ,[Distance] AS DistanceM
      ,[Distance] * 0.87 AS DistanceNM
      ,[Distance] * 1.6 AS DistanceKM
      ,[Distance] / IIF([AirTime] = 0, NULL, [AirTime]*1.0) AS Speed
      ,[DistanceGroup]
      ,[CarrierDelay]
      ,[WeatherDelay]
      ,[NASDelay]
      ,[SecurityDelay]
      ,[LateAircraftDelay]
      ,[FirstDepTime]
      ,[TotalAddGTime]
      ,[LongestAddGTime]
FROM
      [DataVault].[raw].[TSatFlight];
```

Because the data was modified, a new **record source** is used. This approach follows the one in Chapter 14, Loading the Dimensional Information Mark, and helps to identify which soft business rule was implemented in order to produce the data in the satellite.

In addition to the raw data, which is provided for convenience to the power user, the **Distance** column is renamed and used to calculate the distance in nautical miles (**DistanceNM**), kilometers (**DistanceKM**) and the ground speed (column **Speed**).

13.7 STANDARDIZATION OF DATA

During standardization raw data is transformed into a standard format with the goal to provide the data in a shareable, enterprise-wide set of entity types and attributes. The formats and data values are standardized to further enhance the potential business communication and facilitate the data cleansing process because a consistent format helps to consolidate data from multiple sources and identify duplicate records [6]. Examples for data standardization include [1,6]:

- **Stripping extraneous punctuation or white spaces:** some character strings have additional punctuations or white spaces that need to be removed, for example by character trimming.
- **Rearranging data:** in some cases, individual tokens, such as first name and last name, are rearranged into a standard format.
- **Reordering data:** in other cases, there is an implicit order of the elements in a data element, for example in postal addresses. Reordering brings the data into a standardized order.
- **Domain value redundancy:** data from different sources (or even within the same source) might use different unit of measures. In some cases, airline miles might be provided in aeronautic miles, in other cases, ground kilometers.
- **Format inconsistencies:** phone numbers, tax identification numbers, zip codes, and other data elements might be formatted in different ways. Having no enterprise-wide format hinders efficient data analysis or prevents the automated detection of duplicates.
- **Mapping data:** often, there are mapping rules to standardize reference codes used in an operational system to enterprise-wide reference codes or to transform other abbreviations into more meaningful information.

The latter can often be implemented with the help of analytical master data, which are loaded into reference tables in Data Vault 2.0.

13.7.1 T-SQL EXAMPLE

Because the analytical master data has been provided as reference tables to the enterprise data warehouse, it is easy to perform lookups into the analytical master data, even if the Business Vault is virtualized. The following DDL statement is based on the computed satellite created in the previous section and joins a reference table to resolve a system-wide code to an abbreviation requested by the business user:

```
CREATE VIEW [biz].[TSatCleansedFlight] AS
SELECT
       [FlightHashKey]
      ,[LoadDate]
      ,'SR4711.DQ' AS RecordSource
      ,[Year]
      ,[Quarter]
      ,[Month]
      ,[DayOfMonth]
      ,[DayOfWeek]
      ,[CRSDepTime]
      ,[DepTime]
      ,[DepDelay]
      ,[DepDelayMinutes]
      ,[DepDel15]
      ,[DepartureDelayGroups]
      ,[DepTimeBlk]
      ,[TaxiOut]
      ,[WheelsOff]
      ,[WheelsOn]
      ,[TaxiIn]
      ,[CRSArrTime]
      ,[ArrTime]
      ,[ArrDelay]
      ,[ArrDelayMinutes]
      ,[ArrDel15]
      ,[ArrivalDelayGroups]
      ,[ArrTimeBlk]
      ,[Cancelled]
      ,[CancellationCode]
      ,[Diverted]
      ,[CRSElapsedTime]
      ,[ActualElapsedTime]
      ,[AirTime]
      ,[Flights]
      ,[DistanceM]
      ,[DistanceNM]
      ,[DistanceKM]
      ,[Speed]
      ,[DistanceGroup]
      ,[DataVault].[raw].[RefDistanceGroup].Abbreviation AS DistanceGroupText
      ,[CarrierDelay]
      ,[WeatherDelay]
      ,[NASDelay]
      ,[SecurityDelay]
      ,[LateAircraftDelay]
      ,[FirstDepTime]
      ,[TotalAddGTime]
      ,[LongestAddGTime]
FROM
      [DataVault].[biz].[TSatFlight] flight
LEFT JOIN
      [DataVault].[raw].[RefDistanceGroup]
      ON (flight.[DistanceGroup] = [DataVault].[raw].[RefDistanceGroup].Code)
```

The source attribute **DistanceGroup** is a code limited to the source system. It is not used organization-wide. Therefore, the business requests that the code should be resolved into an abbreviation known and used by the business. The mapping is provided in the analytical master data, which is loaded into the Data Vault 2.0 model as reference tables. The resolution of the system codes into known codes can be done by simply joining the reference table **RefDistanceGroup** to the computed satellite and adding the abbreviation as a new attribute, **DistanceGroupText**.

Because the computed satellite has modified the data, a new **record source** is provided.

Another use-case is to align the formatting of the raw data to a common format, for example when dealing with currencies, dates and times, or postal addresses.

13.8 CORRECT AND COMPLETE DATA

The goal of this process is to improve the quality of the data by correcting missing values, known errors and suspect data [6]. This requires that the data errors are known to the data warehouse team or can be found easily using predefined constraints. The first often replaces attribute values of specific records, while the latter is often defined by relatively open business rules involving domain knowledge [30]. Examples include [2]:

- **Single data fields:** this business rule ensures that all data values are following a predefined format, and are within a specific range or domain value.
- **Multiple data fields:** this business rule ensures the integrity between multiple data fields, for example by ensuring that passenger IDs and passenger's country (who issued the ID) match.
- **Probability:** flags unlikely combinations of data values, for example the gender of a passenger and the name.

There are multiple options for the correction of such data errors [7]:

- **Automated correction:** this approach is based on rule-based standardizations, normalizations and corrections. Computed satellites often employ such rules and provide the cleansed data to the next layer in the loading process. Therefore, it is possible to implement this approach in a fully virtualized manner without materializing the cleansed data in the Business Vault.
- **Manual directed correction:** this approach is using automated correction, but the results require manual review (and intervention) before the cleansed data is provided to the next layer in the loading process. To simplify the review process, a confidence level is often introduced and stored along the corrected data. This approach requires materialization of the cleansed data in the Business Vault and should include a confidence value and the result of the manual inspection process. The data steward might also overwrite the cleansed data in a manual correction process. In this case, an external tool is required in addition.
- **Manual correction:** in this case, data stewards inspect the raw data manually and perform their corrections. This approach is often combined with external tools and might involve MDM applications such as Microsoft Master Data Services when dealing with master data.

In some cases, it is not possible to correct all data errors in the raw data. It has to be determined by the business user how to handle such uncorrectable or suspect data. There are multiple options [6]:

- **Reject the data:** if the data is not being used in aggregations or calculations, it might be reasonable to exclude the data from the information mart.
- **Accept the data without a change:** in this case, the data is loaded into the information mart without further notice to the business user.
- **Accept the data with tagging:** instead of just loading the data into the information mart, the error is documented by a tag or comment that is presented to the business user.
- **Estimate the correct or approximate values:** in some cases, it is possible to approximate missing values. This can be used to avoid excluding the data but entails some risks if the estimates are significantly overstating or understating the actual value. The advantage is that this approach helps to keep all occurrences of the data, which is helpful when counting the data.

It is helpful to attach a tag to the resulting data that identifies the applied option. This is especially true for the last case, to indicate to the business user that they are not dealing with actual values but estimates [6].

13.8.1 T-SQL EXAMPLE

In order to reject data without further notice, the virtual satellite can be extended by a WHERE clause:

```
CREATE VIEW [biz].[TSatCleansedFlight2] AS
SELECT
       [FlightHashKey]
      ,[LoadDate]
      ,[RecordSource]
      ,[Year]
      ,[Quarter]
      ,[Month]
      ,[DayOfMonth]
      ,[DayOfWeek]
      ,[CRSDepTime]
      ,[DepTime]
      ,[DepDelay]
      ,[DepDelayMinutes]
      ,[DepDel15]
      ,[DepartureDelayGroups]
      ,[DepTimeBlk]
      ,[TaxiOut]
      ,[WheelsOff]
      ,[WheelsOn]
      ,[TaxiIn]
      ,[CRSArrTime]
      ,[ArrTime]
      ,[ArrDelay]
      ,[ArrDelayMinutes]
      ,[ArrDel15]
      ,[ArrivalDelayGroups]
      ,[ArrTimeBlk]
      ,[Cancelled]
      ,[CancellationCode]
      ,[Diverted]
      ,[CRSElapsedTime]
```

```
      ,[ActualElapsedTime]
      ,[AirTime]
      ,[Flights]
      ,[DistanceM]
      ,[DistanceNM]
      ,[DistanceKM]
      ,[Speed]
      ,[DistanceGroup]
      ,[DistanceGroupText]
      ,[CarrierDelay]
      ,[WeatherDelay]
      ,[NASDelay]
      ,[SecurityDelay]
      ,[LateAircraftDelay]
      ,[FirstDepTime]
      ,[TotalAddGTime]
      ,[LongestAddGTime]
FROM
      [DataVault].[biz].[TSatCleansedFlight]
WHERE
      NOT ([Cancelled] = 0 AND [AirTime] = 0)
```

Some of the source data is erroneous and the business decided to exclude these records completely from further analysis. The WHERE clause filters these faulty records, which are actual flights that haven't been **cancelled** but which have no **airtime**. Because the data itself is not modified, the record source from the base satellite **TSatCleansedFlight** is used. However, this approach removes the ability to identify the soft business rule that is responsible for filtering the data.

13.8.2 DQS EXAMPLE

Correcting raw data can also be performed using data quality management software, such as Microsoft Data Quality Services (DQS), which is included in Microsoft SQL Server. DQS consists of three important components:

- **Server component:** this component is responsible for the execution of cleansing operations and the management of knowledge bases, which keep information about domains and how to identify errors.
- **Client application:** the client application is used by the administrator to set up knowledge bases and by data stewards to define the domain. Data stewards also use it to manually cleanse the data.
- **SSIS transformation:** in order to automatically cleanse the data, a SSIS transformation is available that can be used in SSIS data flows.

Before creating a SSIS data flow that uses DQS for automatic data cleansing, a knowledge base has to be created and domain knowledge implemented.

The following example uses an artificial dataset on passenger records required for security screening [8]. This dataset requires cleansing operations because some of the passenger names and other attributes are misspelled and the dataset contains duplicate records.

Open the DQS client application, connect to the DQS server and create a new knowledge base. The dialog in Figure 13.4 is shown.

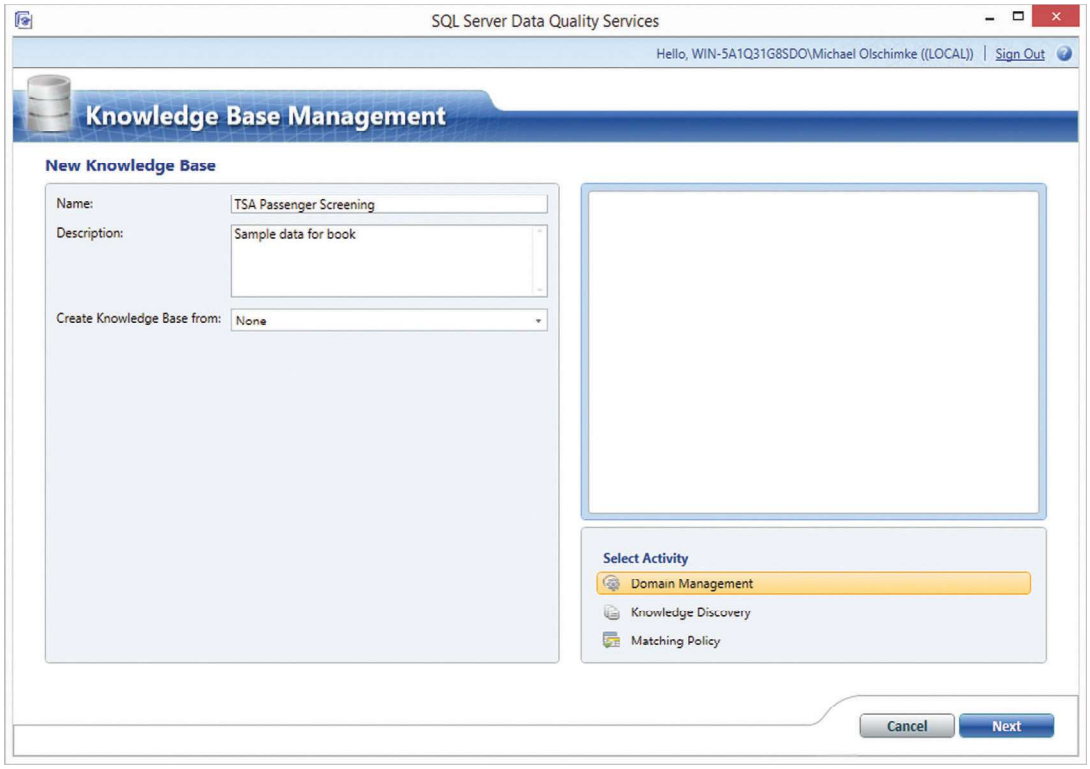

FIGURE 13.4

Create new knowledge base in DQS client.

Provide a name for the new knowledge base and an optional description. There are three possible activities available [9]:

- **Domain management:** this activity is used to create, modify and verify the domains used within the knowledge base. It is possible to change rules and reference values that define the domain. It is possible to verify raw data against the domains in the knowledge base in SSIS.
- **Knowledge discovery:** the definition of domain knowledge can be assisted using this activity where domain knowledge is discovered from source data.
- **Matching policy:** this activity is used to prepare the knowledge base for de-duplication of records. However, this activity is not yet supported by SSIS and only manual-directed correction is supported.

Select **domain management** and click the **next** button to continue. The knowledge base is shown and domains can be managed in the dialog, shown in Figure 13.5.

The dialog presents all available domains in the list on the left side. The right side is used to present the definitions, rules, and other properties of the selected domain. Because the knowledge base is empty, create a new domain by selecting the button on the top left of the left side (Figure 13.6).

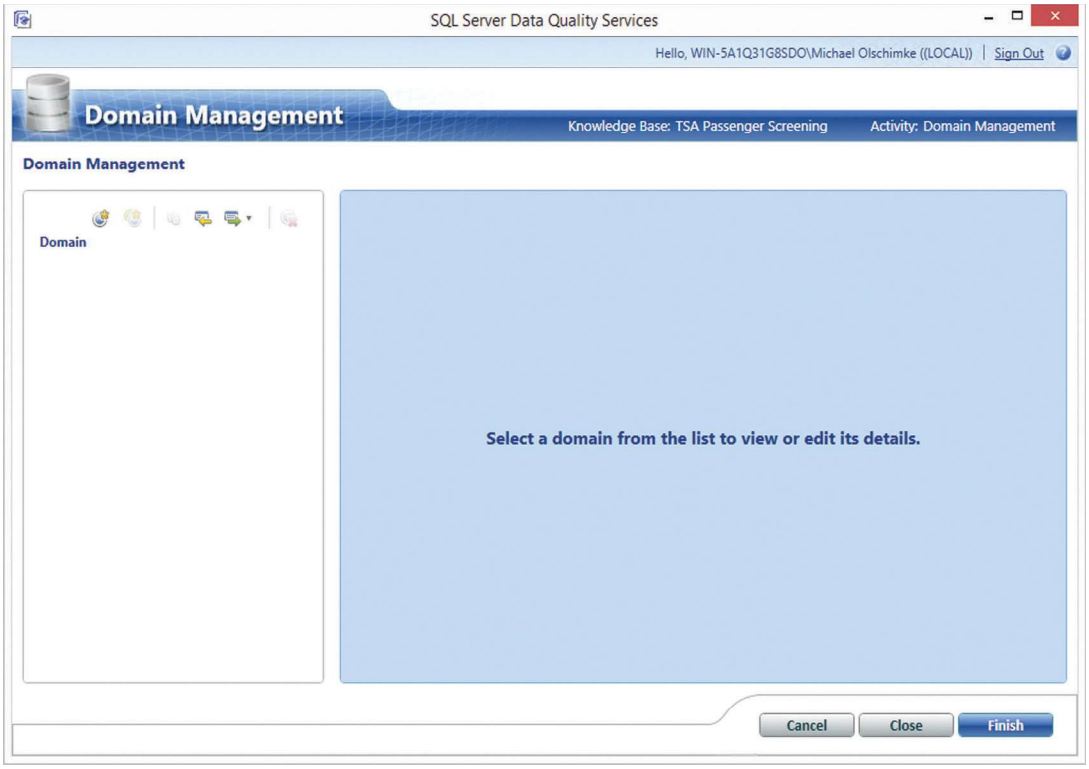

FIGURE 13.5

Domain management in the DQS client.

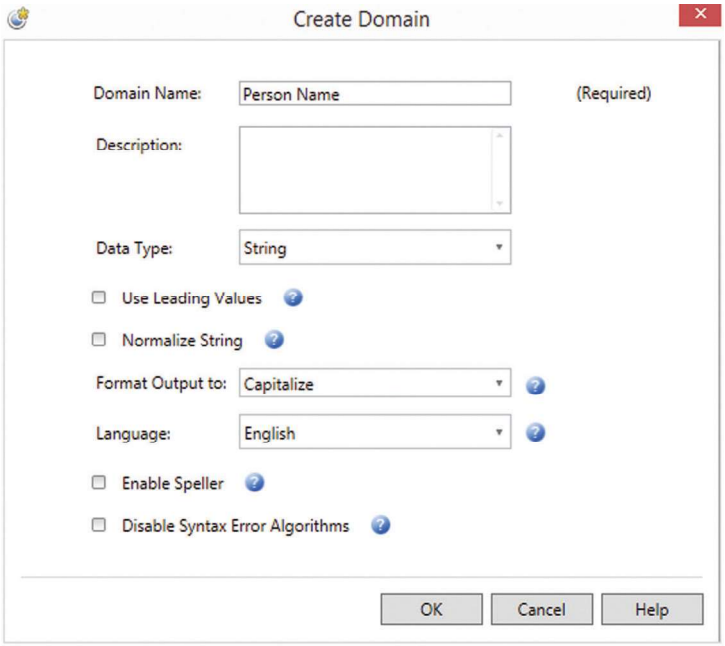

FIGURE 13.6

Create domain in DQS client.

Domains in DQS are defined by the following attributes:

- **Domain name:** the name of the domain.
- **Description:** an optional description of the domain.
- **Data type:** the data type of the domain.
- **Use leading values:** indicates that a leading value should be used instead of synonyms that have the same meaning.
- **Normalize string:** indicates if punctuation should be removed from the input string when performing data quality operations.
- **Format output to:** indicates the data format that should be used when domain values are returned.
- **Language:** the language of the domain's values. This is an important attribute for string domains, required by data quality algorithms.
- **Enable speller:** indicates if the language speller should be applied to the incoming raw data.
- **Disable syntax error algorithms:** this option is used in knowledge discovery activities to check for syntax errors.

Some of these options are only available when the **data type** has been set to **string**.

Close the dialog and switch to the **domain rules** tab (Figure 13.7).

Create a new domain rule that will be used to ensure the maximum string length of a name part. Give it a meaningful name and build the rule as shown in Figure 13.7. This completes the setup of the first domain. Use the dialog to set up the remaining domains in the knowledge database (Table 13.1).

Note that the other domains do not use domain rules (even though **Person DOB** contains some interesting errors in the test dataset). **Person sex** and **person name suffix** define domain values instead (Figure 13.8 and Figure 13.9).

Only the characters "F" and "M" are valid domain values for the domain **person sex**. All other values in raw data will be considered invalid. NULL values are corrected to empty strings.

The domain person name suffix contains a defined set of valid domain values, including "II", "III", "IV" and "Jr.". However, the raw data contains other values as well, for example "JR" which is corrected to "Jr.". In addition, after initial analysis, some errors in the data have been found, for example the suffix "IL" which is corrected to "II".

Whenever a correction is performed by DQS, a warning is provided to the user. It is possible to return this warning to SSIS as well for further processing.

Table 13.1 Domains to Set Up in the Knowledge Database

Domain Name	Data Type	Options	Format	Rule
Person Name	String		Capitalize	Length is less than or equal to <50>
Person DOB	Integer		None	none
Person Sex	String	Use Leading Values	Upper Case	None
Person Name Suffix	String	Use Leading Values Normalize String	None	none

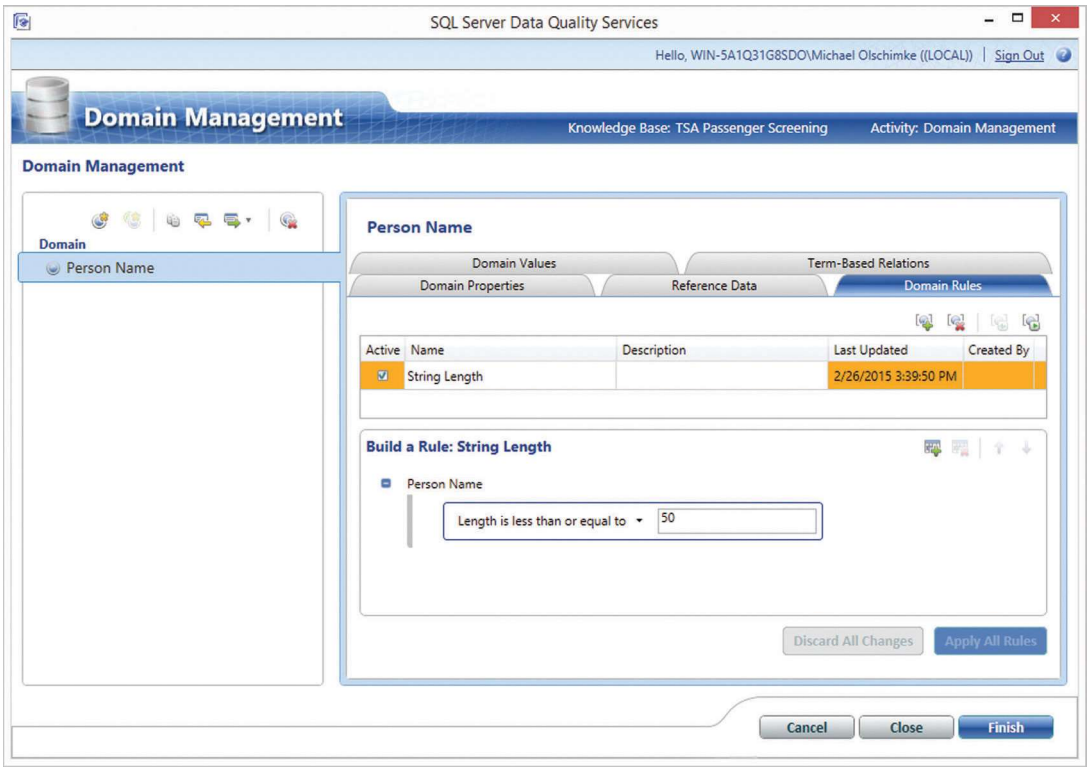

FIGURE 13.7

Defining domain rules for a domain in the DQS client.

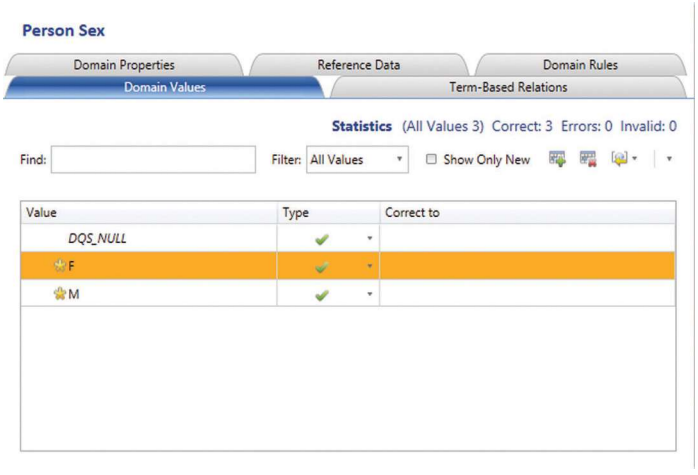

FIGURE 13.8

Domain values of the person sex domain.

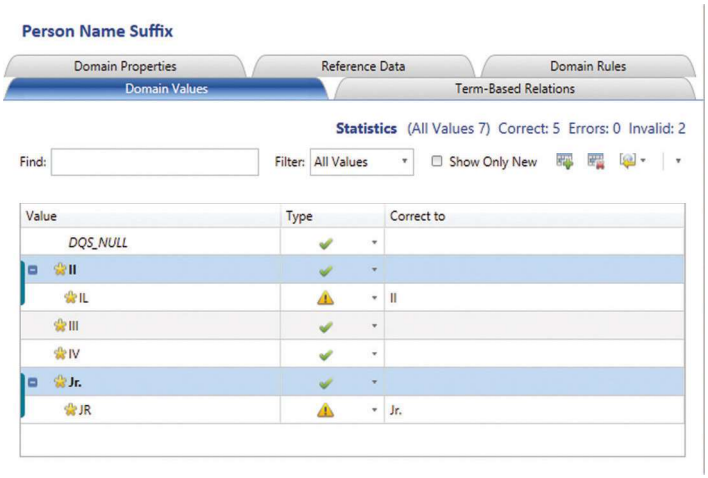

FIGURE 13.9

Domain values of the person name suffix domain.

The domain person name is used in multiple instances, for example as **person first name**, **person last name** and **person middle name**. Instead of creating multiple, independent domains with redundant definitions, it is possible to derive new domains from existing domains by creating a linked domain (Figure 13.10).

Create all three linked domains and derive them from the person name domain. Commit the changes to the knowledge base by selecting the **finish** button on the bottom right of the DQS client. The domains are published to the knowledge base and are available to data stewards or SSIS processes.

13.8.3 SSIS EXAMPLE

The next step is to use the DQS knowledge base in SSIS to retrieve descriptive data from a raw satellite, cleanse the data and write it into a new materialized satellite in the Business Vault. The source data is stored in the raw satellite SatPassenger. Figure 13.11 shows some descriptive data from the satellite.

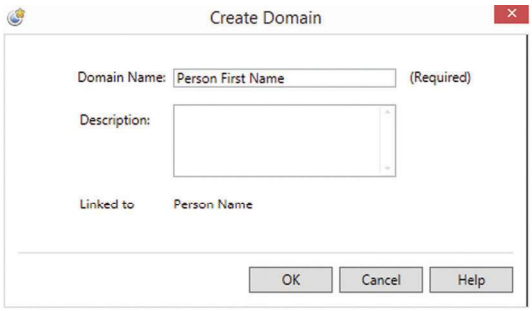

FIGURE 13.10

Creating a linked domain in DQS.

LastName	FirstName	MiddleName	Suffix	DOB	Sex
GROSS	KAYLA	KATHY		19960619	F
JACKSON	DANIEL	ALLEN		20000326	M
HAYS	ISIAH	JORDAN	JR	19970612	M
JOHNSON	MAURICE			19961122	M
DE JESUS	LUIS			19970828	M
CARLSEN	AMY	LEEANN		19970728	F
CLARK	MIKE	SAMUEL		19970129	M
O BRYANT	SEAN			19970801	M
SHEPPARD	STEVEN	T		19970130	M
HUGHES	JOSEPH			19970609	M
KING	CHRISTY			19990920	F
HAYAKAWA	ATSUSHI			19960609	F
BARRON	ANTHONY	ALFONSO		19970801	M

FIGURE 13.11

Sample data from SatPassenger.

The following DDL statement is used to create the target satellite:

```
CREATE TABLE [biz].[SatCleansedPassenger](
       [PersonHashKey] [varchar](32) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [LoadEndDate] [datetime2](7) NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [Sequence] [int] NOT NULL,
       [LastName] [nvarchar](50) NOT NULL,
       [LastName_Status] [nvarchar](100) NULL,
       [LastName_Confidence] [nvarchar](100) NULL,
       [LastName_Reason] [nvarchar](4000) NULL,
       [FirstName] [nvarchar](50) NOT NULL,
       [FirstName_Status] [nvarchar](100) NULL,
       [FirstName_Confidence] [nvarchar](100) NULL,
       [FirstName_Reason] [nvarchar](4000) NULL,
       [MiddleName] [nvarchar](50) NOT NULL,
       [MiddleName_Status] [nvarchar](100) NULL,
       [MiddleName_Confidence] [nvarchar](100) NULL,
       [MiddleName_Reason] [nvarchar](4000) NULL,
       [Suffix] [nvarchar](50) NULL,
       [Suffix_Status] [nvarchar](100) NULL,
       [Suffix_Confidence] [nvarchar](100) NULL,
       [Suffix_Reason] [nvarchar](4000) NULL,
       [DOB] [date] NULL,
       [DOB_Status] [nvarchar](100) NULL,
       [DOB_Confidence] [nvarchar](100) NULL,
       [DOB_Reason] [nvarchar](4000) NULL,
       [Sex] [varchar](1) NOT NULL,
       [Sex_Status] [nvarchar](100) NULL,
       [Sex_Confidence] [nvarchar](100) NULL,
       [Sex_Reason] [nvarchar](4000) NULL,
       [Record Status] [nvarchar](100) NULL,
       [Appended Data] [nvarchar](4000) NULL,
       [Appended Data Schema] [nvarchar](4000) NULL,
 CONSTRAINT [PK_SatCleansedPassenger] PRIMARY KEY NONCLUSTERED
(
       [PersonHashKey] ASC,
       [LoadDate] ASC
) ON [INDEX]
) ON [DATA]
```

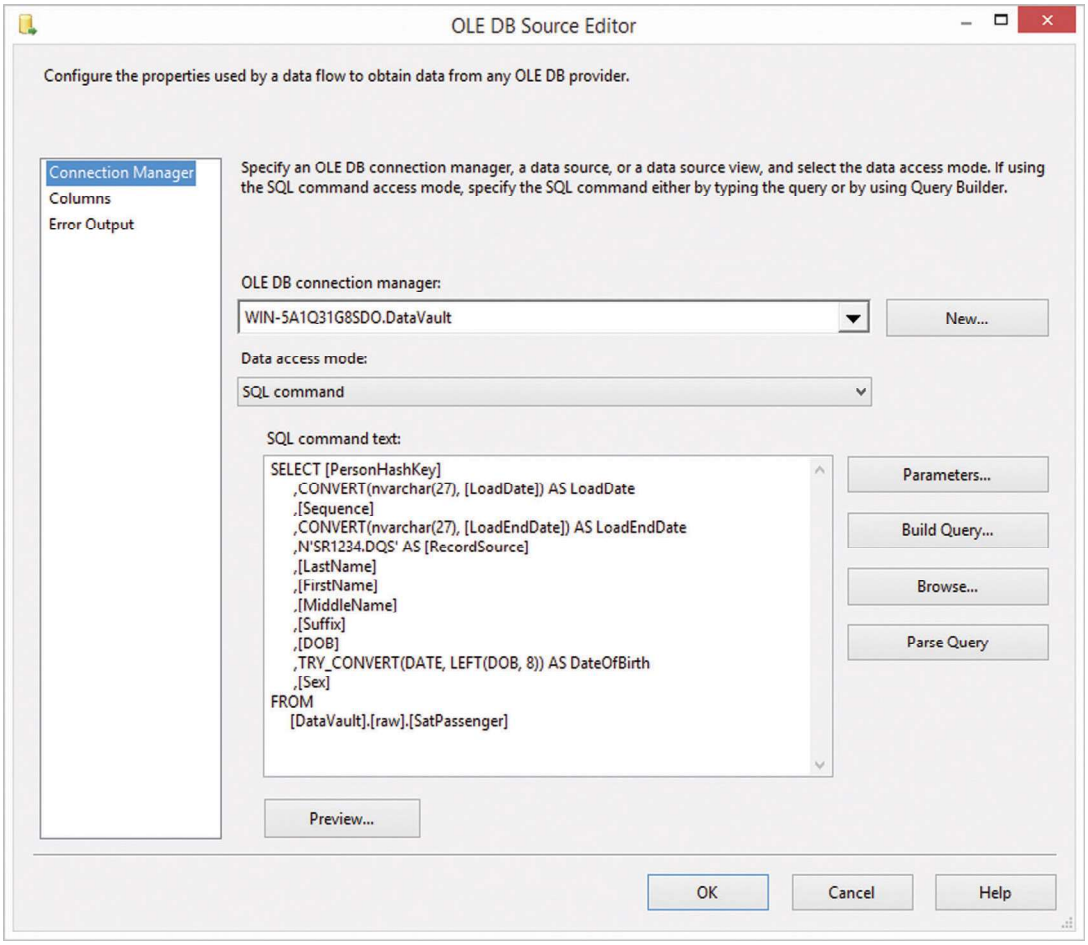

FIGURE 13.12

OLE DB source editor to retrieve raw data from source satellite.

Despite the existence of a sequence attribute, the previous satellite is a standard (computed) satellite and not a multi-active satellite. The sequence number is not included in the primary key definition. It is used for reference purposes only, in order to retrieve the raw data for a cleansed record.

Create a new data flow and add an OLE DB source to it. Open the editor to configure the source connection and columns (Figure 13.12).

Select the source database **DataVault** and choose **SQL command** as data access mode. Set the **SQL command text** to the following query:

```
SELECT [PersonHashKey]
      ,CONVERT(nvarchar(27), [LoadDate]) AS LoadDate
      ,[Sequence]
      ,CONVERT(nvarchar(27), [LoadEndDate]) AS LoadEndDate
      ,N'SR1234.DQS' AS [RecordSource]
      ,[LastName]
      ,[FirstName]
      ,[MiddleName]
      ,[Suffix]
      ,[DOB]
      ,TRY_CONVERT(DATE, LEFT(DOB, 8)) AS DateOfBirth
      ,[Sex]
FROM
     [DataVault].[raw].[SatPassenger]
```

This query retrieves the data to be cleansed from the raw satellite. Because DQS doesn't support timestamps from the database, the **load date** and **load end date** from the source are converted into strings. The statement also tries to convert the date of birth (DOB) into a date column (which is supported by DQS). The problem is that the DOB column contains invalid values which are either out of range or do not specify a valid date (such as the 20011414 or 20010231). Because the data flow modifies the data, the record source is set to the identifier of the soft business rule definition in the meta mart (formatted as Unicode string).

On the next page, make sure that all columns are selected and close the dialog.

Another problem of the source data is that it contains duplicate data describing the same person (which is identified by last name, first name, middle name, date of birth in the parent hub). This is why the source satellite in the Raw Data Vault is a multi-active satellite to be able to store multiple different sets of descriptive records for the same (composite) business key.

However, in this example, the business has decided to store only one passenger description in the target business satellite. Because it doesn't really matter which one is chosen, it is possible to use a sorter to remove duplicates in a simple approach. Therefore, drag a sort transformation to the data flow and connect it to the OLE DB source transformation from the last step. Open the editor and configure it in the dialog as shown in Figure 13.13.

The loading process in this section follows a simple approach: each record from the source is loaded into the target satellite if it is not a duplicate of the same description within the same batch. That means multiple versions of the same record should be loaded into the target to show how the (cleansed) description changes over time.

The cleansed data is close to the original data in terms of granularity: strings are correctly capitalized, and the domain values are validated. Therefore, changes in the source satellite will likely lead to a required change in the target satellite and not many records are skipped because the cleansing results in an unchanged line in the target. If this were not the case, a more sophisticated change tracking should be applied on the cleansed data.

Because the approach used in this example is simplified, the data is sorted by the two columns **PassengerHashKey** and **LoadDate** only. When new entries are loaded in later batches, they are just taken over; however, duplicate records for the same passenger in the same batch are removed.

In order to ensure the restartability of the process, the data flow has to check if there is already a record with the same hash key and load date in the target satellite. This is done using a lookup component, especially because no descriptive data should be compared (because the data flow avoids true delta checking on the target). Add a lookup transformation to the data flow, connect it to the existing transformations and open the lookup transformation editor (Figure 13.14).

The setup is similar to other loading procedures, as rows without matching entries are redirected to a "no match output." The only difference is that the cache has been turned off: using the cache in this example makes no sense because the combination of hash key and load date is unique in the underlying data stream from the source (due to the de-duplication in the sort transformation from the last step) and, therefore, the lookup would not profit from an enabled cache. The lookup condition will be configured on the third page of this dialog (the **columns** page).

Select the next page to set up the connection (Figure 13.15).

Because the SQL command text used in the source has converted the load date to a string data type (due to DQS), the same has to be done in the lookup. Otherwise, the columns cannot be used in the lookup condition because of different data types. Connect to the **DataVault** database and use the following statement to retrieve the data for the lookup:

```
SELECT
      [PersonHashKey]
      ,CONVERT(nvarchar(27), [LoadDate]) AS LoadDate
FROM
      [DataVault].[biz].[SatCleansedPassenger]
```

Because no descriptive data is required from the lookup table, only the **hash key** and the converted **load date** are retrieved from the target satellite in the Business Vault. The lookup condition based on these two columns is configured on the next page, shown in Figure 13.16.

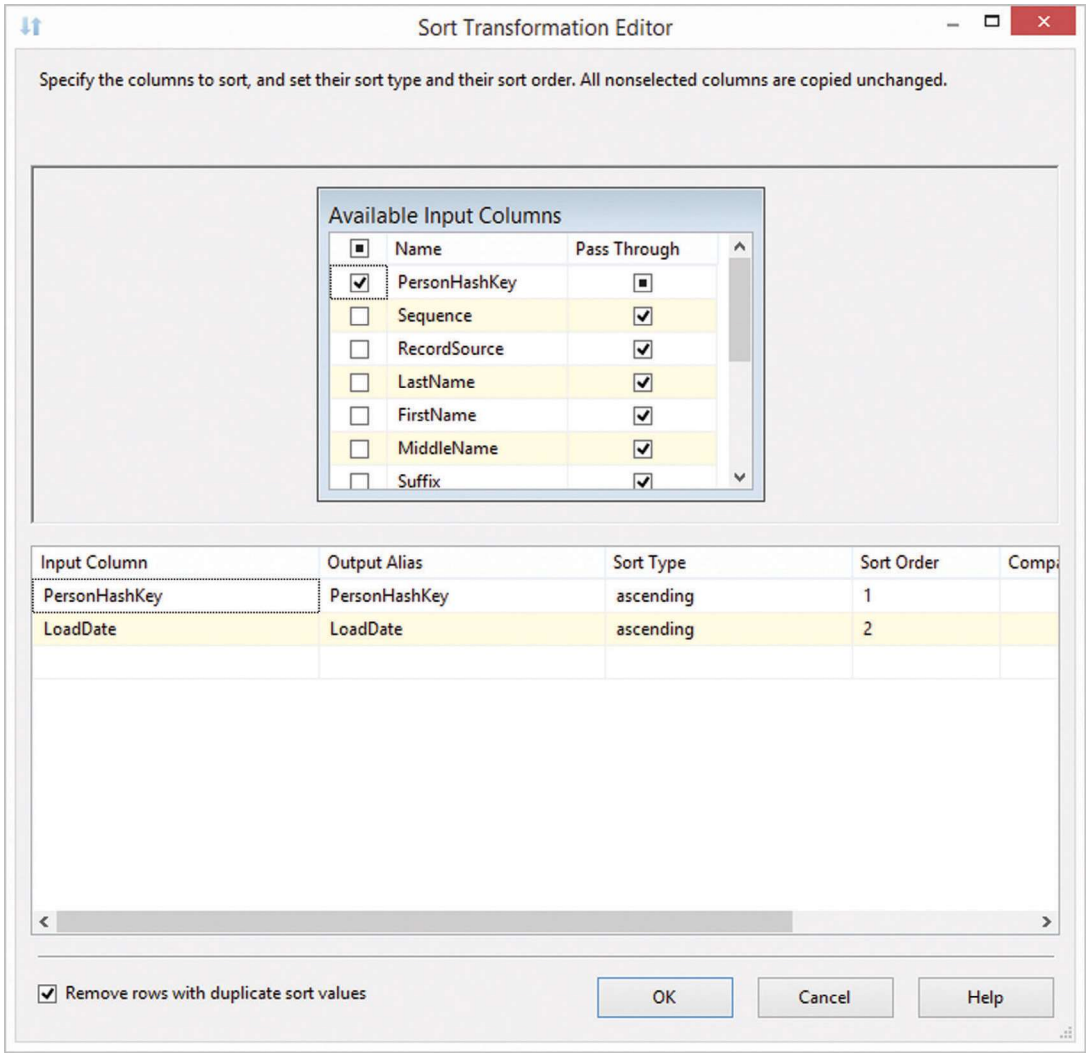

FIGURE 13.13

Sort transformation editor to remove duplicate passenger descriptions.

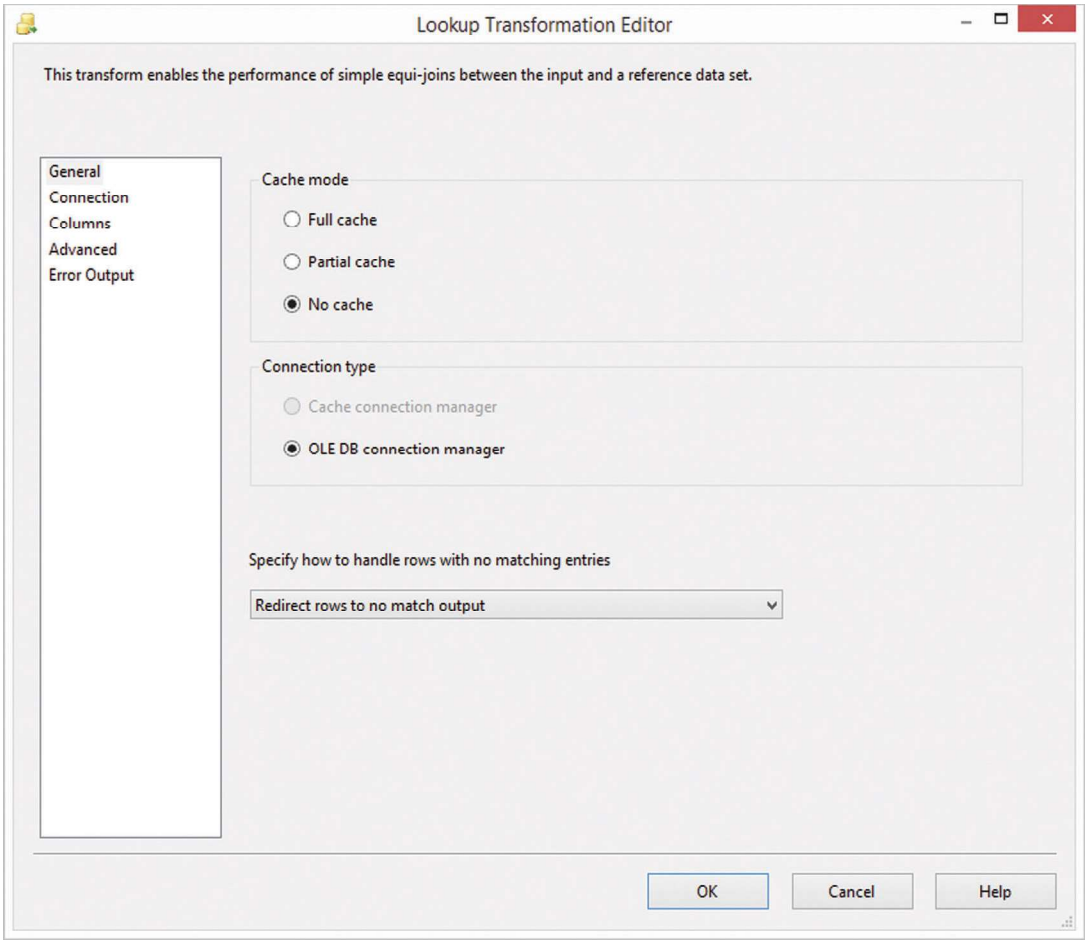

FIGURE 13.14

Lookup transformation editor to check if the key combination is already in the cleansed target satellite.

Connect the corresponding hash keys and load dates from both the source table and the lookup table and close the dialog by clicking the OK button.

Add a **DQS Cleansing** transformation to the data flow and connect it to the output of the lookup. There are two outputs from the lookup transformation: one for records already in the target satellite and therefore found in the lookup table and one output with records not found in the target and therefore unknown. We are only interested in the unknown records. Therefore, connect the **lookup no match output** to the DQS cleansing input, as shown in Figure 13.17.

After closing the dialog, the connection between components is configured. Open the configuration editor for the DQS cleansing transformation (Figure 13.18).

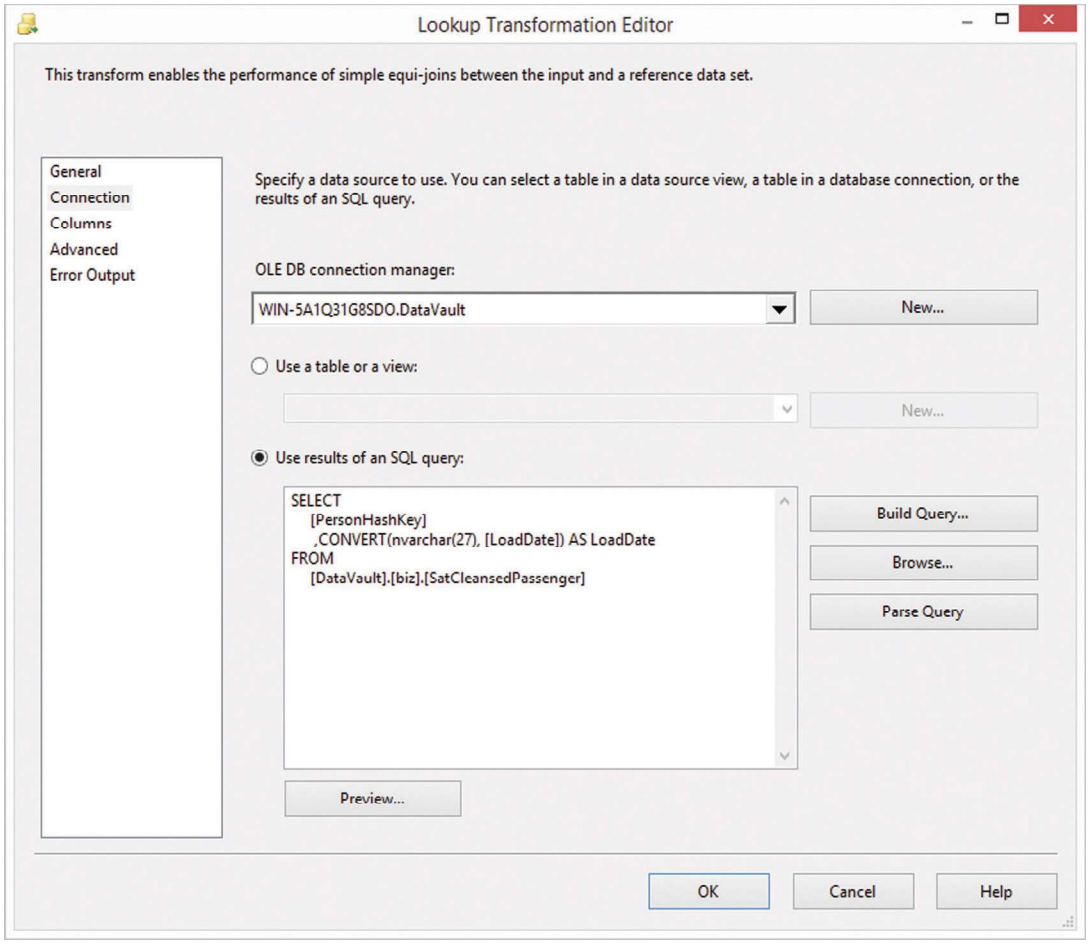

FIGURE 13.15

Set up the connection of the lookup transformation.

Create a new connection to the DQS server by using the **new** button. Once the connection is established, select the knowledge base configured in the previous section. The dialog will show domains found in the knowledge base.

Switch to the **mapping** tab to map the columns in the data flow to the domains in the knowledge base (Figure 13.19).

Select the columns from the data flow that should be cleansed in the grid on the upper half of the dialog. In order to cleanse a column, a corresponding DQS domain is required. If there are other columns in the data flow without a corresponding domain in the knowledge base, they have to be added in DQS first (as in the previous section).

Map each selected input column to a DQS domain. Leave all other options (source alias, output alias, status alias, confidence alias and reason alias), as they are. These columns will provide useful

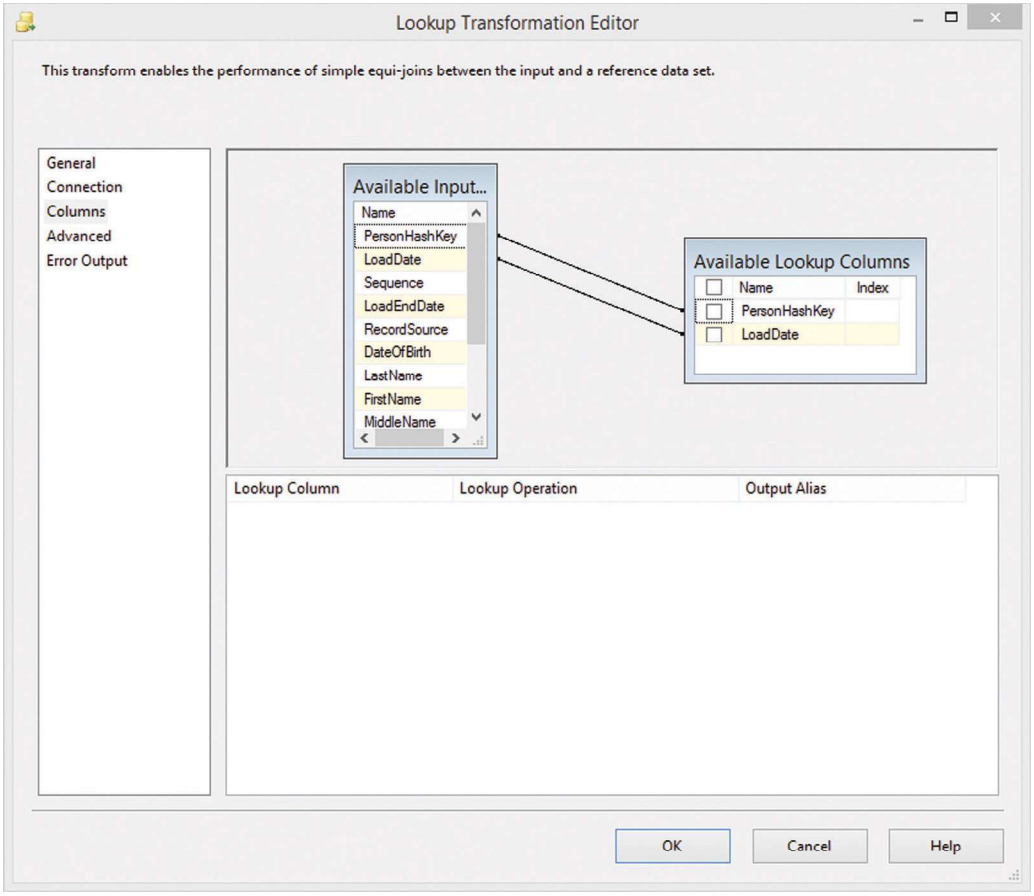

FIGURE 13.16

Configure lookup condition.

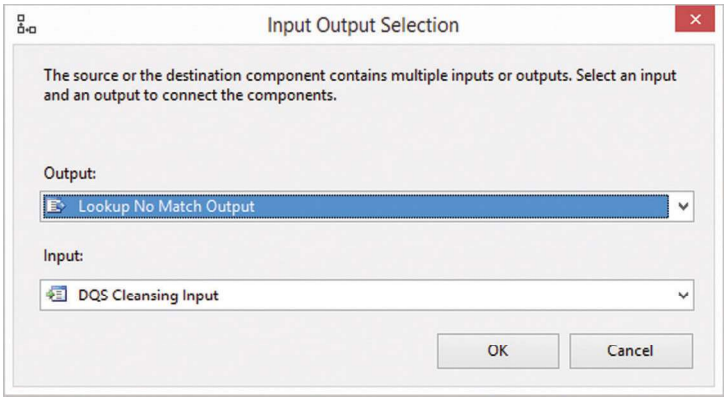

FIGURE 13.17

Connecting the lookup output to the DQS cleansing input.

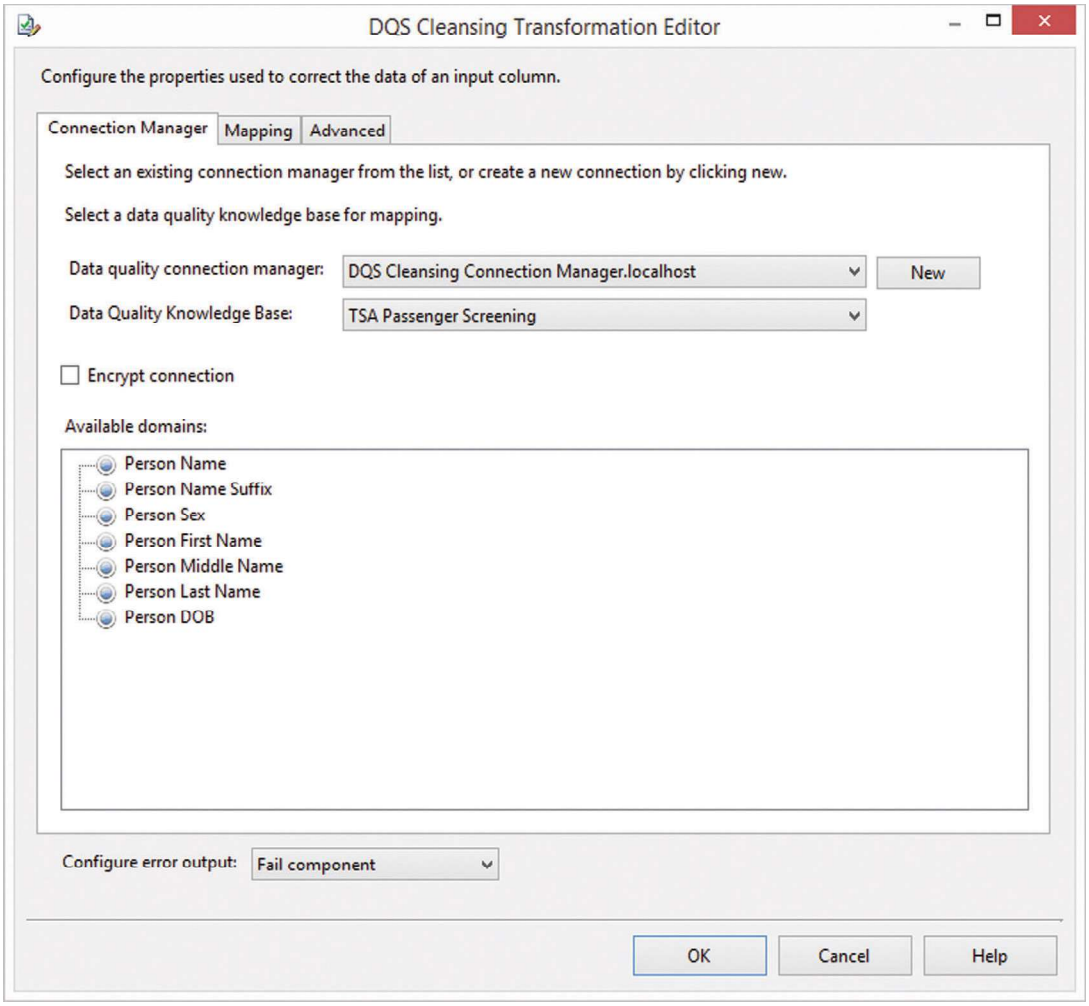

FIGURE 13.18

DQS cleansing transformation editor.

information about the data cleansing operations to the business user. However, in order to actually include these informative columns in the output of the DQS cleansing transformation, switch to the **advanced** tab (Figure 13.20).

Check all options to add all informative columns. The first option is used to standardize the output, which means that their standardized counterparts might replace domain values.

Close the dialog and add an OLE DB destination component to the data flow. Connect it to the DQS cleansing transformation. Open the editor of the OLE DB destination transformation to configure it (Figure 13.21).

Select the **DataVault** database and the target satellite in the Business Vault. Configure the column mapping between the data flow and the target table on the next page, shown in Figure 13.22.

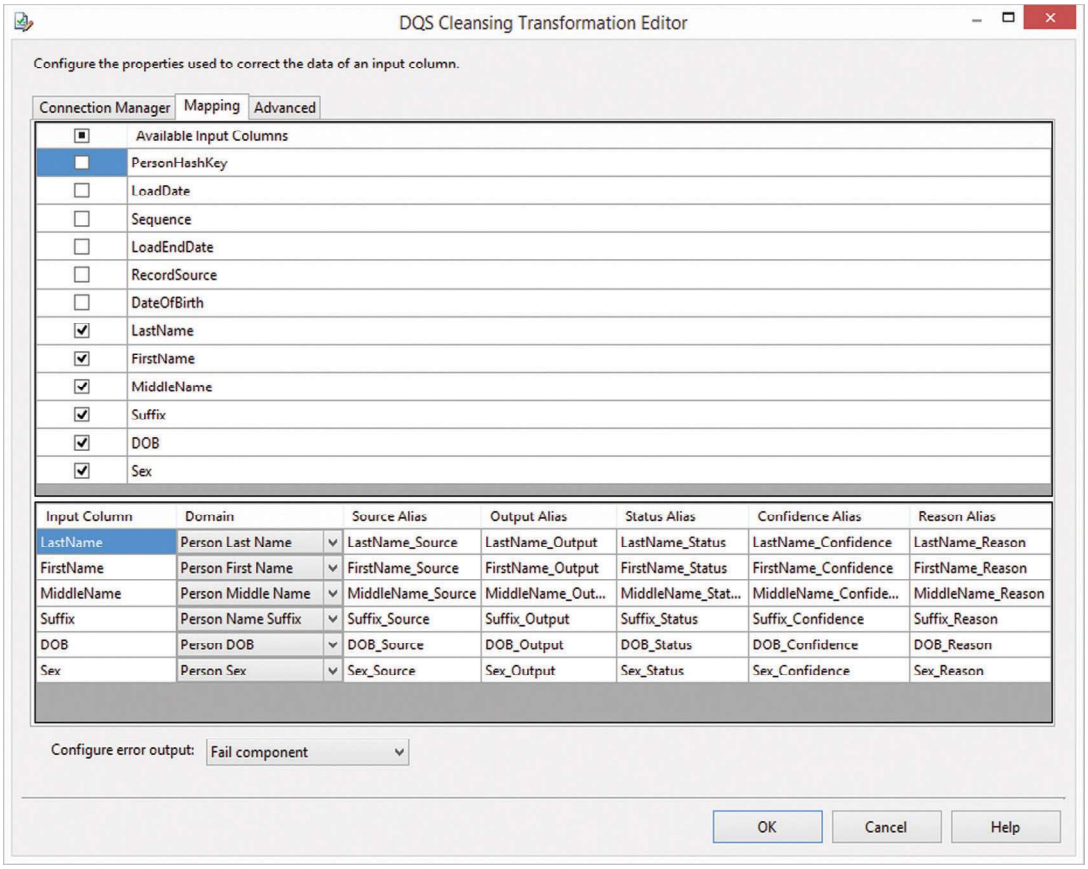

FIGURE 13.19

Mapping columns from the data flow to DQS domains.

The source columns from the DQS cleansing transformation are ignored because the same data is found in the raw satellite. The sequence from the source is added as a descriptive attribute to the target satellite (which is not a multi-active satellite) to retrieve the corresponding record from the source. The output columns are mapped to the cleansed attributes. All other attributes (status, confidence, and reason) are added and mapped to the target as well.

The final data flow is presented in Figure 13.23.

Because the lookup is executed before the DQS cleansing takes place, the data is reduced to the records that are unknown to the target satellite and actually require data cleansing: all known records have been cleansed in the past.

Note that the ETL process to build a business satellite often doesn't follow a prescribed template, as is the case when loading the Raw Data Vault as shown in Chapter 12, Loading the Data Vault. Instead, most business vault entities are loaded using individual processes, which is hard to automate.

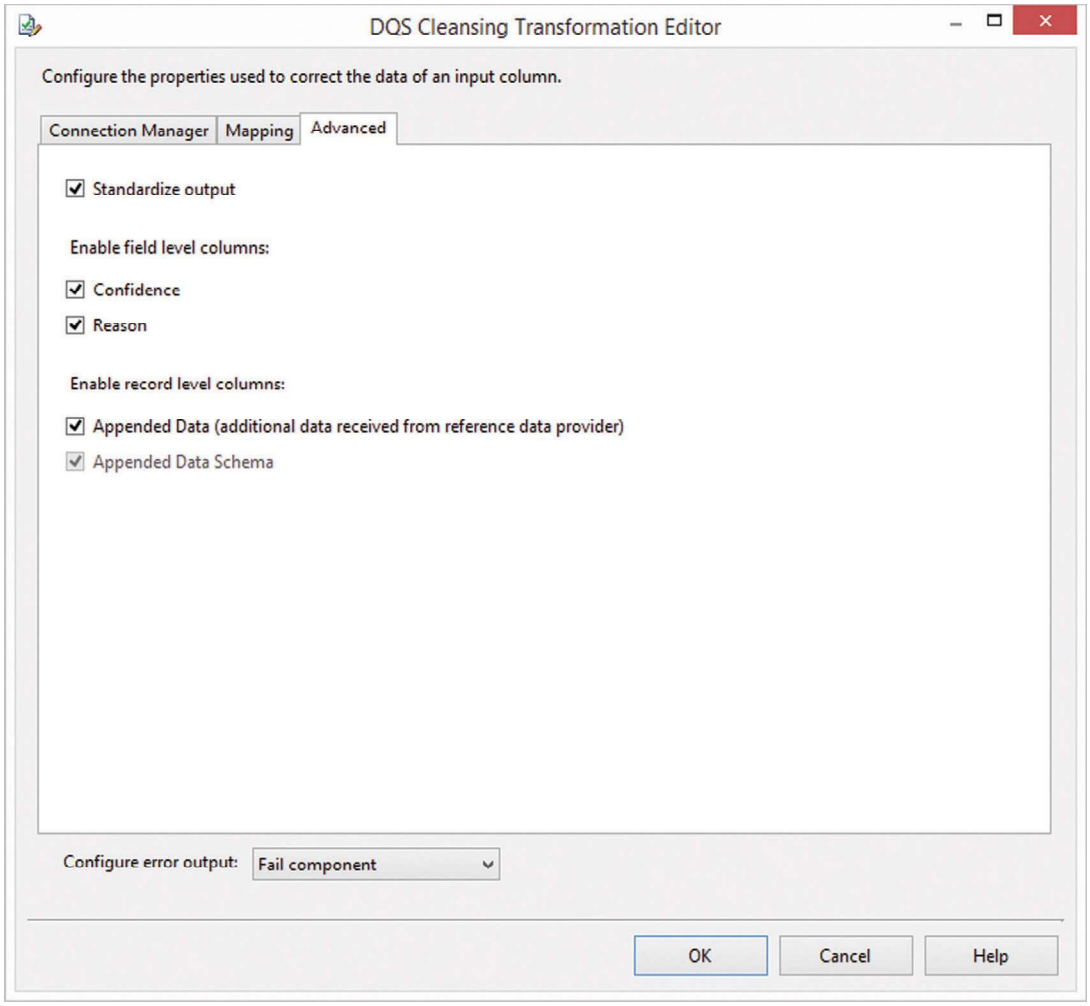

FIGURE 13.20

Advanced tab of DQS cleansing transformation editor.

To complete the process, it is required to end-date the target satellite. This approach, however, follows the standard end-dating process for satellites in the Raw Data Vault, outlined in Chapter 12.

To further improve the solution, the informative attributes from the DQS cleansing transformation could be loaded into a separate satellite instead of the same target. This would improve the usability and performance of the cleansed satellite.

The resulting satellite can be used as any other satellite in the Raw or Business Data Vault for loading the information marts.

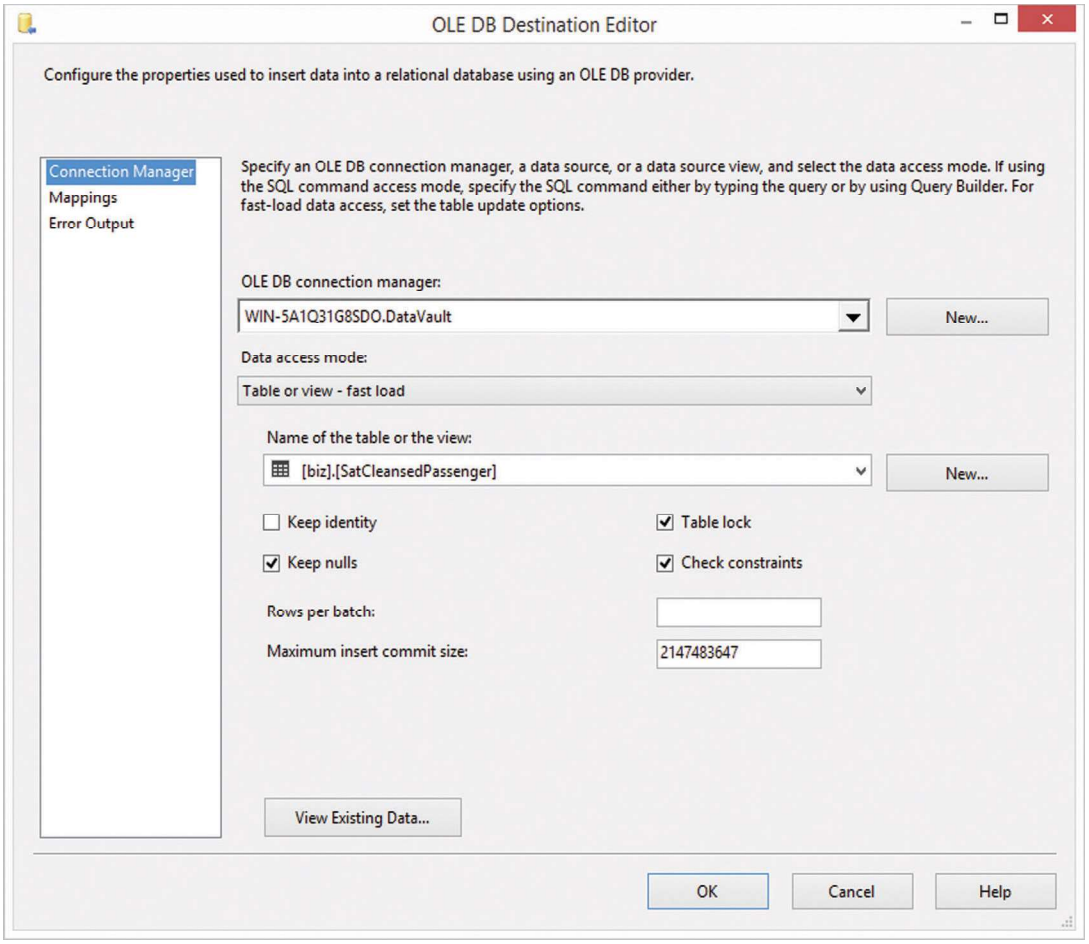

FIGURE 13.21

OLE DB destination editor for target satellite in Business Vault.

13.9 MATCH AND CONSOLIDATE DATA

A typical task in data warehousing is to resolve identities that [36]:

- **Represent the same entity:** here, duplicate business keys or relationships from the same or different source systems mean the same business entity. The typical goal is to merge the business entities into one and consolidate the data from both records or remove duplicate data in favor of a master record.
- **Represent the wrong entity:** there are cases where the business user thinks that a record is not in the system because of a slight variation in the descriptive data, which leads to an unfound (yet existing) record [10].

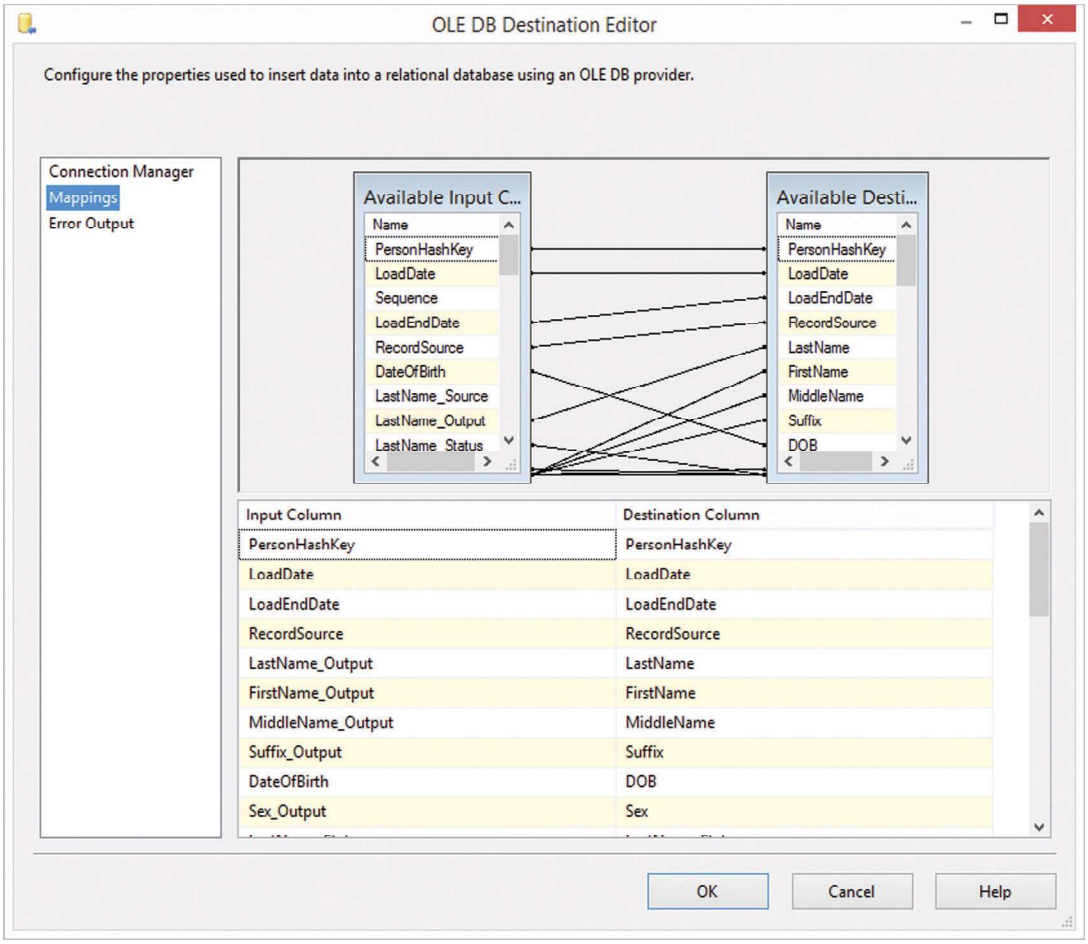

FIGURE 13.22

Column mapping in the OLE DB destination editor.

- **Represent the same household:** in other cases, there are different business entities (identified as such) that belong together. For example, they live in the same household or work in the same organization. The goal is to aggregate the data to the higher order entity.

For all cases, there are entities in the Data Vault 2.0 model that support each of them. The resolution of duplicate identifiers for the same business entities can be solved with a same-as-link (SAL) that was introduced in Chapter 5. Errors in descriptive data can be fixed using computed satellites, similar to the approaches discussed in section 13.8. The last case can be solved with a hierarchical link.

This approach can be supported by a variety of techniques and tools, for example [3]:

- **Data matching techniques:** it is possible to identify potential duplicates using matching techniques, such as phonetic, match code, cross-reference, partial text match, pattern matching, and fuzzy logic algorithms [6].

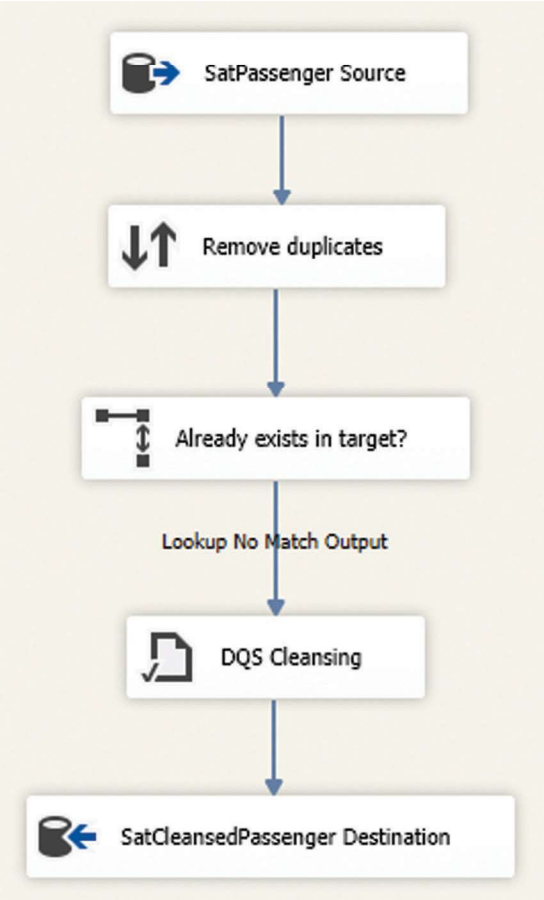

FIGURE 13.23

Data cleansing in SSIS.

- **Data mining software:** instead of relying on rules, data mining tools rely on artificial intelligence algorithms that find close matches.
- **Data correction software:** such software implements data cleansing and other algorithms to test for potential duplicate matches and merge the records.

The next section presents how to de-duplicate data using a fuzzy data matching algorithm included in SSIS.

13.9.1 SSIS EXAMPLE

Fuzzy logic represents a fairly easy approach for data de-duplication and is supported by SSIS. The example in section 13.8.3 has demonstrated how to cleanse descriptive attributes in satellites. The satellite also contained duplicate data, describing similar persons. However, in order to cleanse the data, the hub has to be cleansed first.

The recommended approach for data de-duplication in Data Vault 2.0 is to take advantage of a same-as link, introduced in Chapter 5. By doing so, the raw data is left untouched and the concept provides the most flexibility.

The **HubPerson** is defined using the following DDL:

```
CREATE TABLE [raw].[HubPerson](
	[PersonHashKey] [char](32) NOT NULL,
	[LoadDate] [datetime2](7) NOT NULL,
	[RecordSource] [nvarchar](50) NOT NULL,
	[LastName] [nvarchar](50) NOT NULL,
	[FirstName] [nvarchar](50) NOT NULL,
	[MiddleName] [nvarchar](50) NOT NULL,
	[DOB] [int] NOT NULL,
 CONSTRAINT [PK_HubPerson] PRIMARY KEY NONCLUSTERED
(
	[PersonHashKey] ASC
) ON [INDEX],
 CONSTRAINT [UK_HubPerson] UNIQUE NONCLUSTERED
(
	[LastName] ASC,
	[FirstName] ASC,
	[MiddleName] ASC,
	[DOB] ASC
) ON [INDEX]
) ON [DATA]
```

The business key of this hub is a composite key consisting of the attributes LastName, FirstName, MiddleName, and date of birth DOB, apparently because there was no passenger ID or person ID available in the source data. This is a weak way of identifying a person, with many possible false positive matches. The better approach is to use a business key for this identification purpose. However, the sample data doesn't provide such a business key and, in reality, there are some cases where no business key for persons is available.

Create the following same-as link so it can be used as the data flow's destination:

```
CREATE TABLE [biz].[SALPerson](
	[SALPersonHashKey] [char](32) NOT NULL,
	[LoadDate] [datetime2](7) NOT NULL,
	[RecordSource] [nvarchar](50) NOT NULL,
	[PersonMasterHashKey] [char](32) NOT NULL,
	[PersonDuplicateHashKey] [char](32) NOT NULL,
	[Score] [real] NOT NULL,
 CONSTRAINT [PK_SALPerson] PRIMARY KEY NONCLUSTERED
(
	[SALPersonHashKey] ASC
) ON [INDEX],
 CONSTRAINT [UK_SALPerson] UNIQUE NONCLUSTERED
(
	[PersonMasterHashKey] ASC,
	[PersonDuplicateHashKey] ASC
) ON [INDEX]
) ON [DATA]
```

The same-as link is defined by two hash keys, which point to the same referenced hub, **HubPerson**. One of the hash keys references the master record; the other hash key references the duplicate record that should be replaced by the master in subsequent queries.

Create a new data flow and drag an **OLE DB source** component to it. Open the editor (Figure 13.24).

Because only data is used as descriptive input for the following fuzzy group process, no other descriptive data needs to be joined to the source. Therefore, the **HubPerson** in the Raw Data Vault is selected. If additional descriptive data should be considered in the de-duplication process, for example data from dependent satellites, the data should be joined in a source SQL command.

Make sure that all columns from the hub are included in the OLE DB source and close the dialog. Add a **fuzzy grouping** transformation to the canvas of the data flow and connect it to the source.

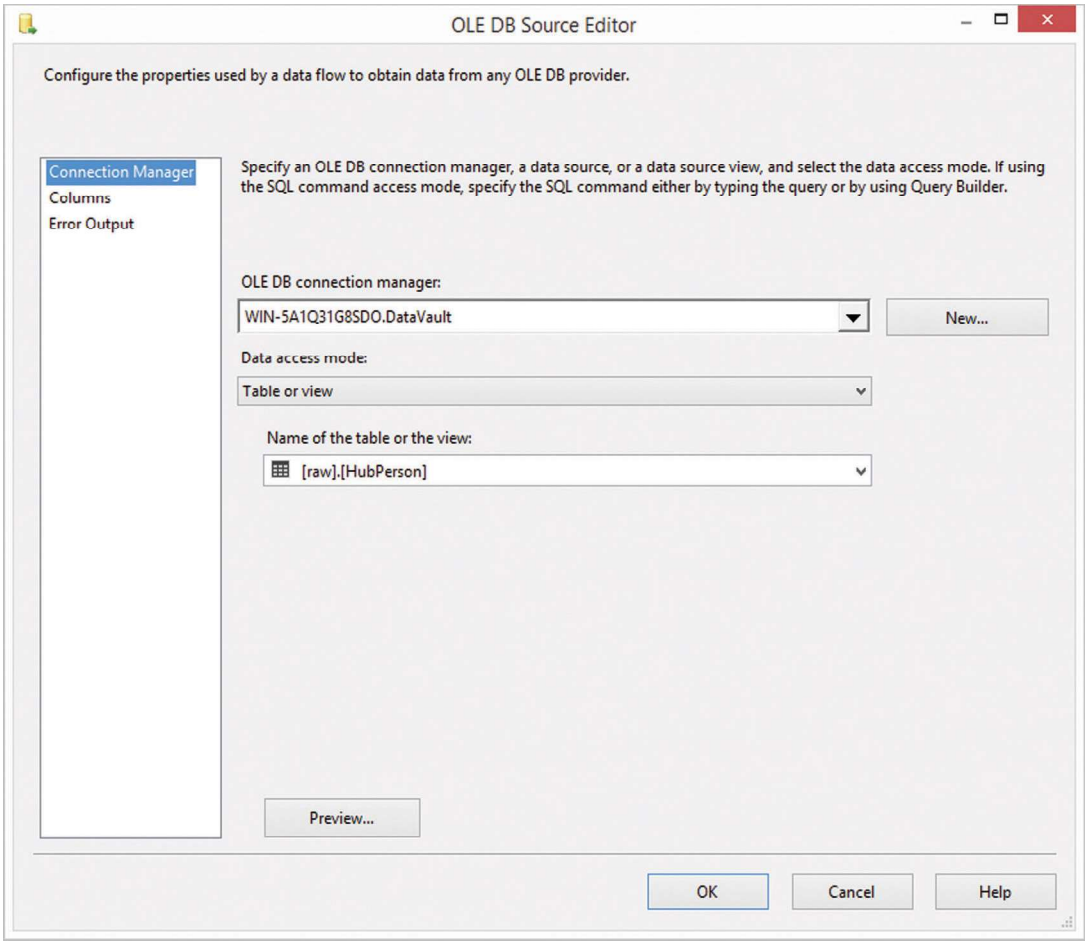

FIGURE 13.24

OLE DB source editor for HubPerson.

Because the fuzzy grouping algorithm is resource-intensive, it requires a database for storing temporary SQL Server tables. Create a new connection and specify the **tempdb** database (Figure 13.25).

The advantage of the **tempdb** database is that every user has access to the database and created objects are dropped on disconnect [11]. In most cases, it is also configured for such applications. As an alternative, it is also possible to use another database for storing these temporary objects. You should consult your data warehouse administrator for more information.

Make sure the created tempdb connection manager is selected on the first page of the fuzzy grouping transformation editor (Figure 13.26).

Select the **columns** tab to set up the descriptive columns that should be used in the de-duplication process (Figure 13.27).

Select all descriptive columns and set the match type. For string columns, this should be fuzzy. In some cases, it makes sense to enforce the exactness of other attributes, as this is the case for the DOB column. This domain knowledge is usually provided by the business user.

Make sure that all other columns that are not used in the fuzzy grouping algorithm are activated for pass-through and switch to the **advanced** tab, shown in Figure 13.28.

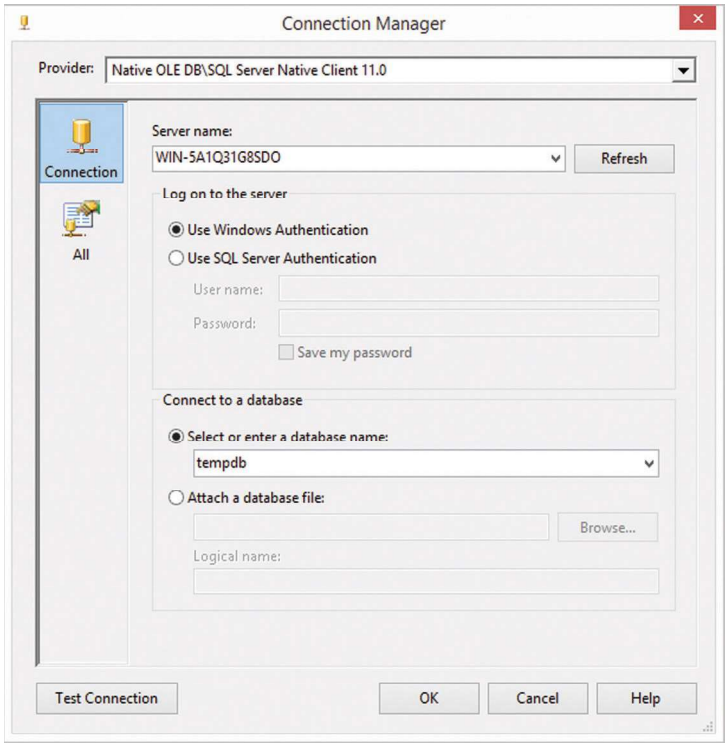

FIGURE 13.25

Configuring tempdb for fuzzy grouping.

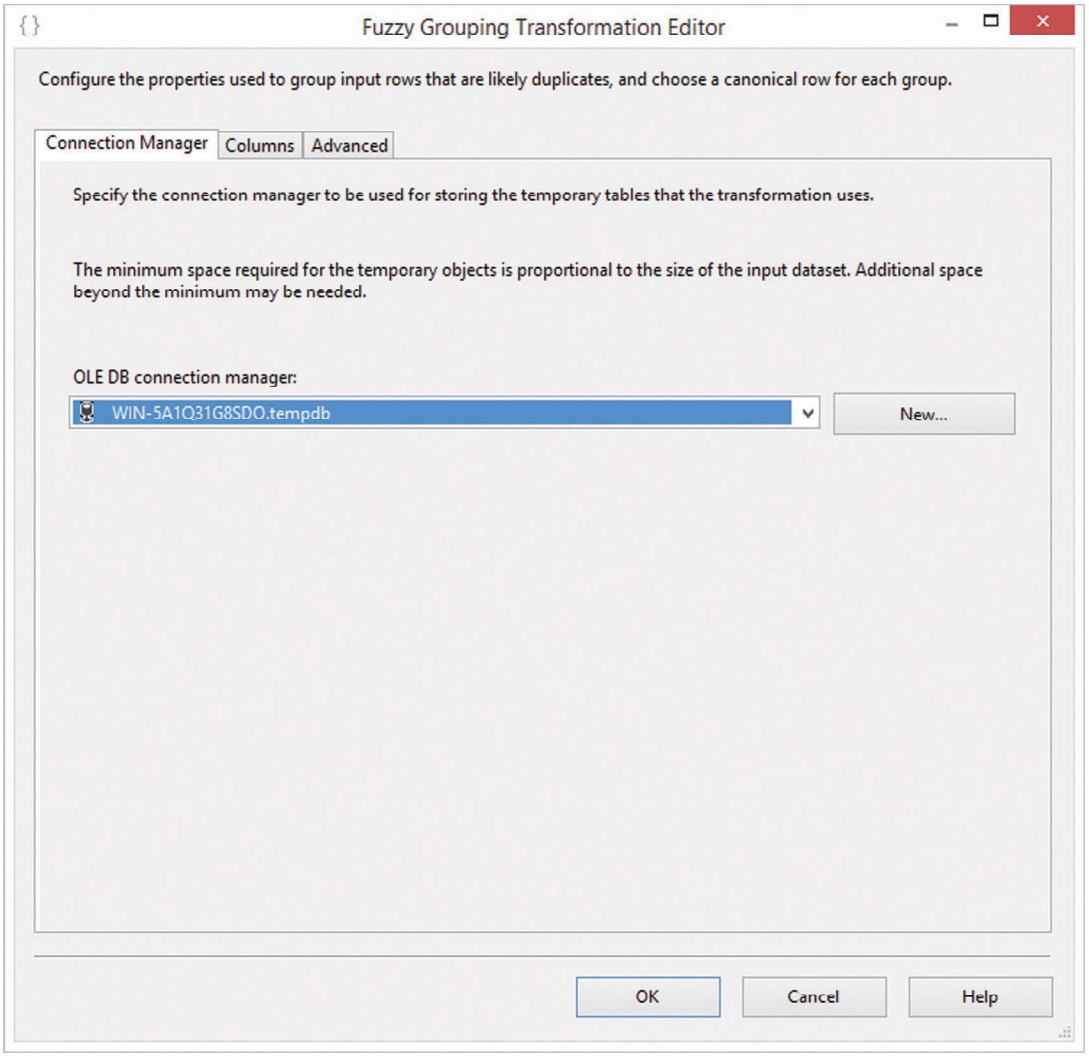

FIGURE 13.26

Configuring the connection manager of the fuzzy grouping transformation.

The similarity threshold influences which persons are considered as duplicates. The lower the threshold is, the more false-positive matches the algorithm will produce. On the other hand, keeping the value too high increases the number of false-negative matches:

- **False-positive match:** the fuzzy-grouping algorithm wrongly (false) thinks that two persons are the same (positive).
- **False-negative match:** the fuzzy-grouping algorithm wrongly (false) thinks that two persons are not the same (negative).

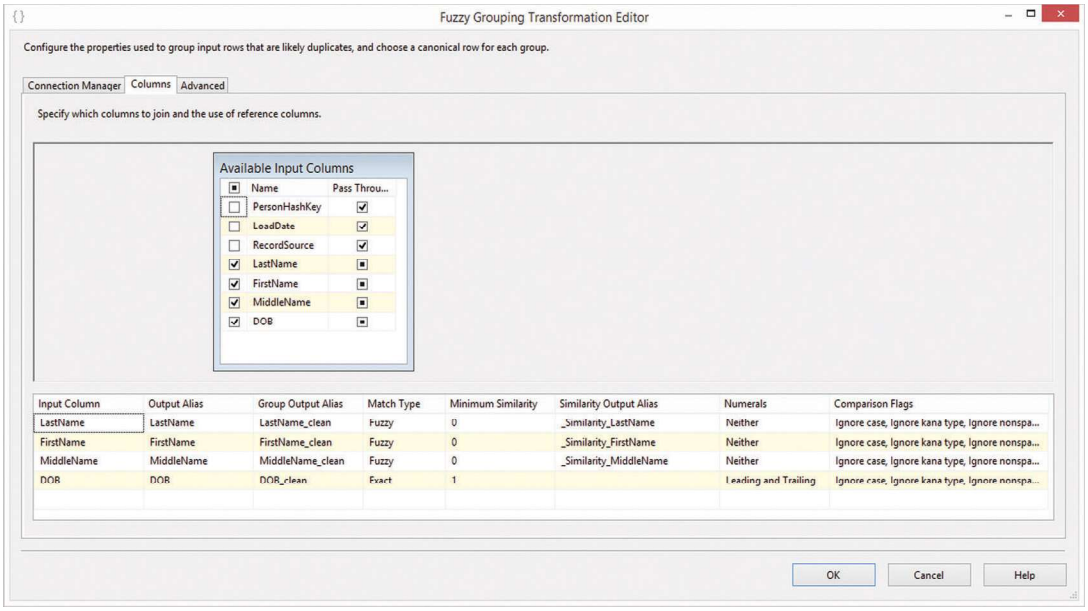

FIGURE 13.27

Setting up descriptive columns for de-duplication.

- **True-positive match:** the algorithms correctly (true) identified two persons as duplicates (positive).
- **True-negative match:** the algorithm correctly (true) classified both persons as different (negative).

While the default value of 0.80 is a good setting for most cases, the same-as link is capable of providing a better solution when setting this threshold to a lower value because it gives the user of the same-as link (the information mart or a power user) more choices. Therefore, set the threshold to a very low value of 0.05. This will produce a lot of false-negative matches, but section 13.10 will demonstrate that it doesn't matter (not from a storage or usage perspective) but allows the user to use a user-defined threshold instead of relying on this constant value. It is only recommended to set the threshold to a higher value when dealing with large volumes of business keys in the source hub (regardless of any descriptive data).

The result that will be produced by the fuzzy grouping transformation is shown in Figure 13.29. The transformation adds the columns defined in Figure 13.28 which are used as follows:

- **_key_in:** a surrogate key that identifies the original record. In the same-as link terminology, this is the duplicate key. The surrogate key was introduced by SSIS and is not coming from the source.
- **_key_out:** the surrogate key of the record that the record should be mapped to. This is the master key. If the record should not be mapped to another record, both surrogate keys are the same.
- **_score:** the similarity score of both records (the duplicate and the master record). If both are the same (and no mapping takes place), the similarity score is 1.0 (100% equal).

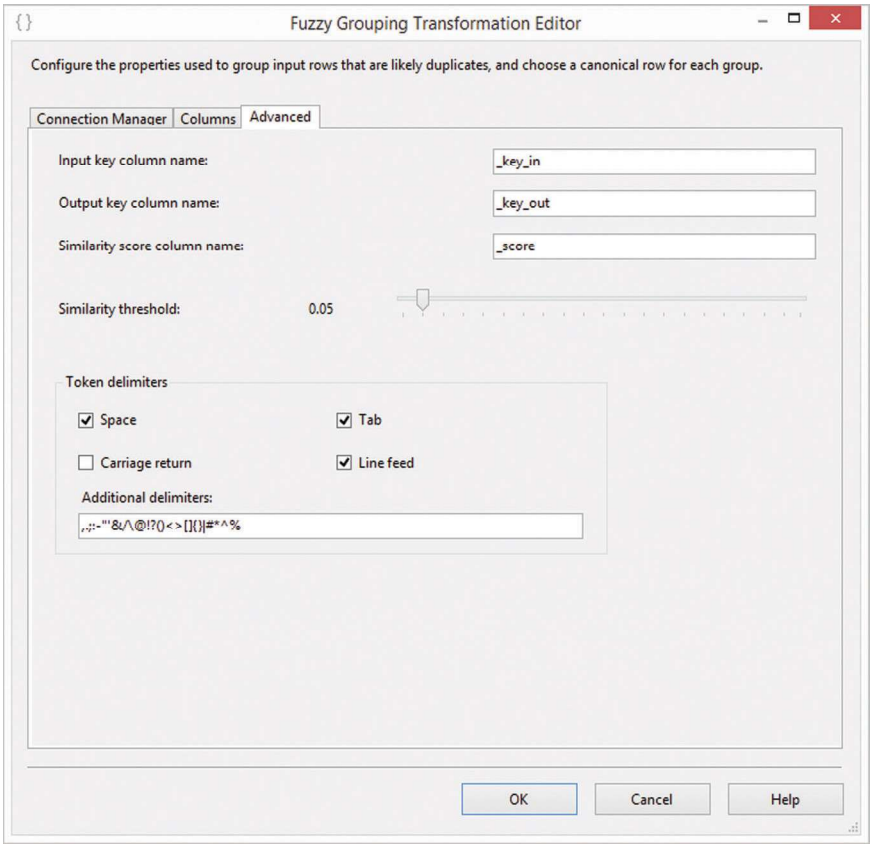

FIGURE 13.28

Configuring advanced options in the fuzzy grouping transformation editor.

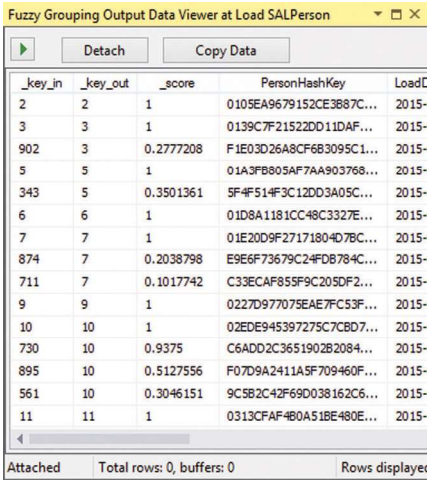

FIGURE 13.29

Output of fuzzy grouping transformation.

The problem with this output, which cannot be influenced much, is that the hash keys of the master and duplicate records are required for loading the target same-as link. The hash key that is in the data flow identifies the duplicate record. In order to add the hash key of the master, the easiest and probably the fastest way is to create a copy of the data and merge it back by joining over the surrogate keys.

Close the fuzzy grouping transformation editor, add a **multicast** transformation to the data flow and add two **sort** transformations, each fed from a path from the multicast transformation. The first sort transformation is configured to sort its own copy of the data flow by the surrogate key of the master record (Figure 13.30).

Select the column **_key_out** for sorting and pass through all other columns. Close the dialog and open the configuration dialog for the second sort transformation, which sorts its copy of the data flow by the surrogate key of the duplicate record (Figure 13.31).

Select the surrogate key of the duplicate, which is **_key_in**. Enable pass-through for all other columns. Close the dialog and add a **merge join** transformation. Connect it to the outputs of both sort transformations and open the transformation editor (Figure 13.32).

Connect **_key_in** from the duplicate data stream to **_key_out** from the master data stream with each other. This becomes the join condition. Select both hash keys and name them **MasterPersonHashKey** and **DuplicatePersonHashKey**. They will be written to the target link and reference the hub. Make sure to select all elements of the composite business key (LastName, FirstName, MiddleName, and DOB) from each stream. They are required in order to calculate the hash key for the same-as link. In addition, select the **_score** attribute from the duplicate data stream. It will be written to the target link as well.

Set the join type to inner join: there should be a matching master record in the second data flow.

Close the editor and connect the output of the **merge join** transformation to a new **derived column** transformation. It is used to calculate the hash key of the same-as link using the approach presented in Chapter 11, Data Extraction (see Figure 13.33).

Add a **load date** to the stream and retrieve the load date of the current batch from the SSIS variable **dLoadDate**. Set the **record source** to an identifier for the soft rule. Calculate the input to be used for the link hash key (SALPersonHashKey) calculation using the following expression:

```
UPPER(TRIM(REPLACENULL(DuplicateLastName,"")) + ";" + TRIM(REPLACENULL(DuplicateFirstName,""))
+ ";" + TRIM(REPLACENULL(DuplicateMiddleName,"")) + ";" +
TRIM(REPLACENULL((DT_WSTR,8)DuplicateDOB,"")) + ";" + TRIM(REPLACENULL(MasterLastName,"")) +
";" + TRIM(REPLACENULL(MasterFirstName,"")) + ";" + TRIM(REPLACENULL(MasterMiddleName,"")) + ";"
+ TRIM(REPLACENULL((DT_WSTR,8)MasterDOB,"")))
```

Make sure the name of the input column ends with "LinkBK" so that the hash function can catch up the business key automatically. Drag a script component to the canvas and connect it to the existing data flow. Its purpose is to calculate the hash key based on the input created in the last step. It follows the same approach as in Chapter 11.

Open the script transformation editor and switch to the **input** columns page (Figure 13.34).

Select the input column created in the previous step. Switch to the **inputs and outputs** tab and create a new output (Figure 13.35).

Add a new output column to the existing output. Use the exact same name as the selected input column but ending with "HashKey" (refer to Chapter 11 for details). Use a string data type with a length

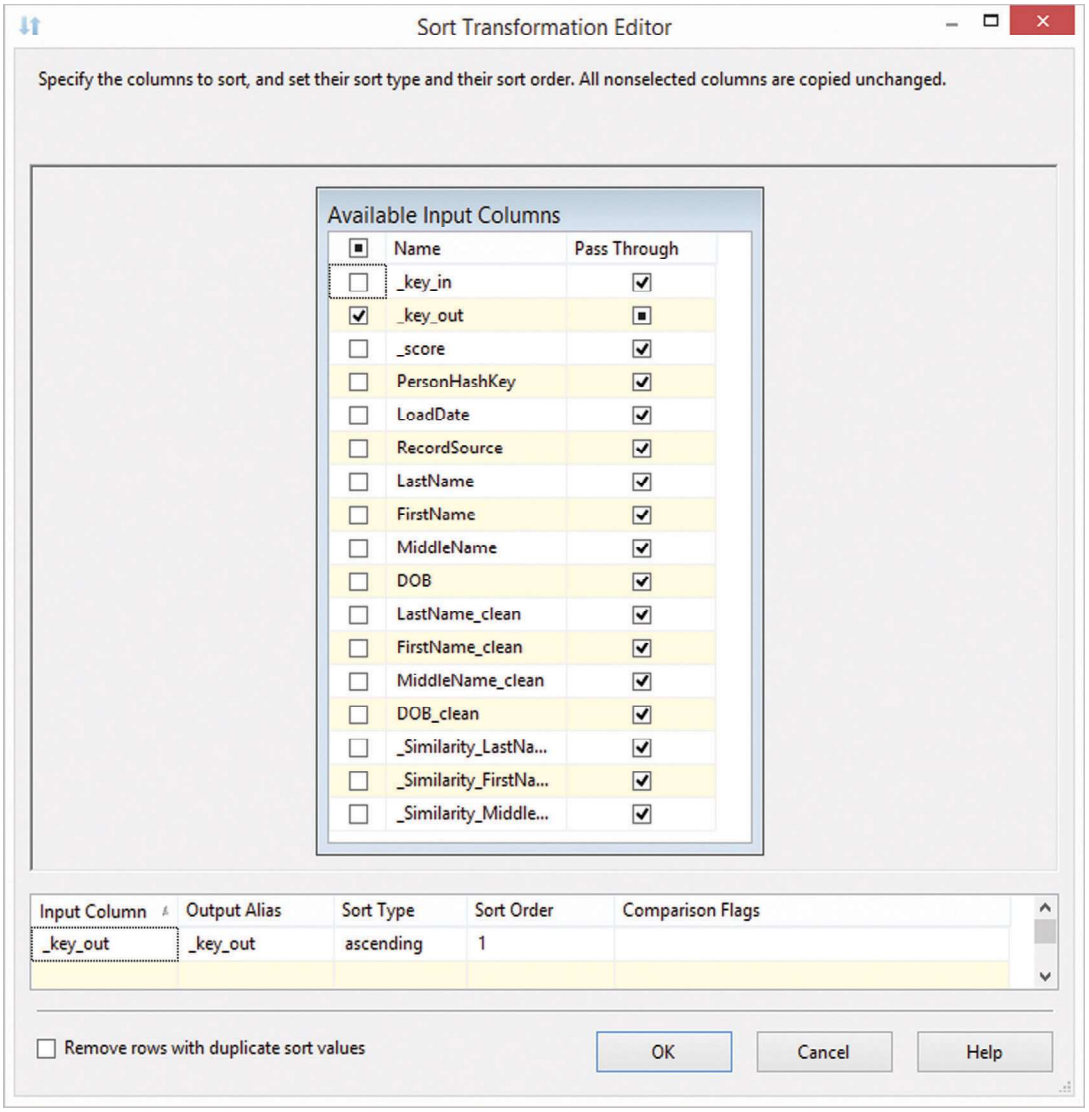

FIGURE 13.30

Configuration of the sort transformation of the master data flow.

of 32 characters and code page 1252 (or whatever code page is appropriate in your data warehouse environment). However, note that the hash key uses only characters from 0 to 9 and A to F. Switch to the script tab and add the script from Chapter 11.

Build the script and close both the script editor and the script transformation editor. Drag an **OLE DB destination** component to the data flow and connect it to the output of the script component. Open the editor to configure the target (Figure 13.36).

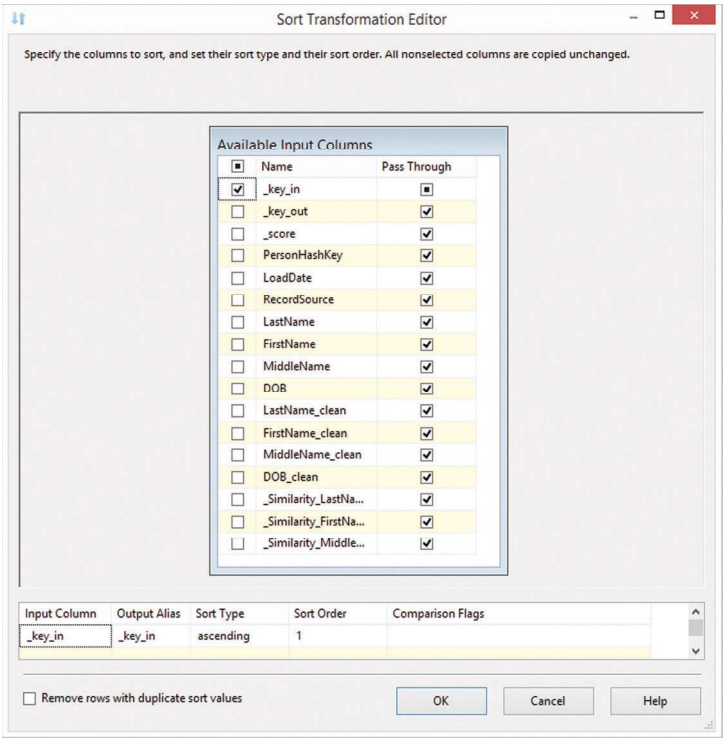

FIGURE 13.31

Configuration of the sort transformation of the duplicate data flow.

Select the target link SALPerson in the Business Vault. Check keep nulls and table lock. Switch to the mappings tab to configure the mapping between the data flow and the target table (Figure 13.37).

Make sure that each target column is sourced from the data flow. Close the dialog.

The final data flow is presented in Figure 13.38.

There are two options to run this data flow in practice:

1. Incremental load: this approach requires updating entries when the scores or the mapping (from duplicate to master) change.
2. Truncate target before full load: in order to avoid the update, the target is truncated and reloaded with every run.

In most cases, the second option is much easier to use while still feasible from a performance standpoint. Only if the hub contains many records is incremental loading required.