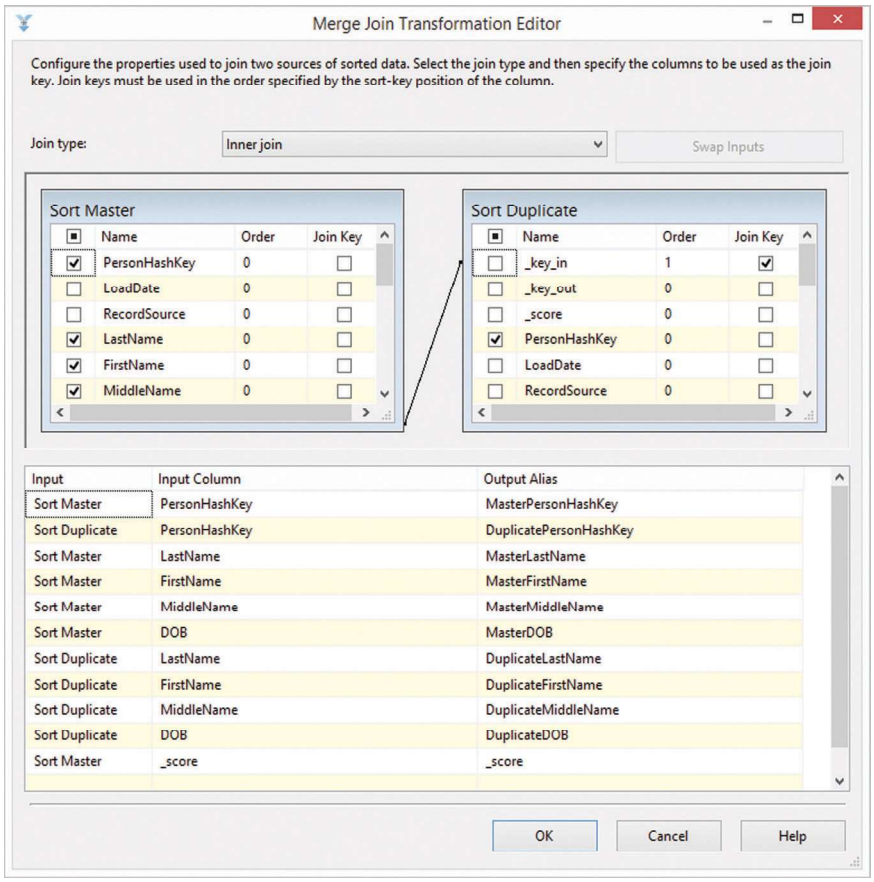

FIGURE 13.32

Merge join transformation editor for joining the master record to the duplicate record.

13.10 CREATING DIMENSIONS FROM SAME-AS LINKS

The same-as link created in the previous section can be used as the foundation for a de-duplicated dimension:

```
CREATE VIEW DimPassenger AS
SELECT
       src.PersonHashKey
       ,sat.FirstName
       ,sat.MiddleName
       ,sat.LastName
       ,sat.DOB
```

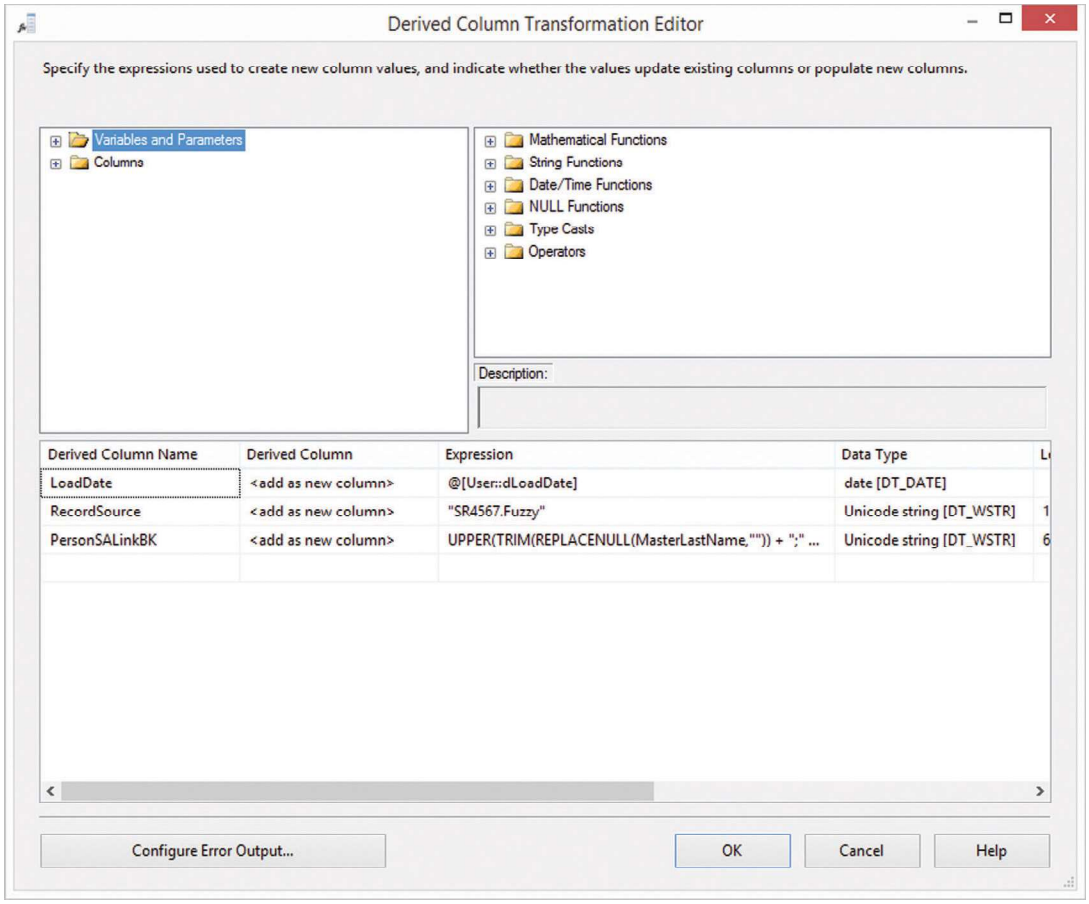

FIGURE 13.33

Derived column transformation editor.

```
FROM (
       SELECT DISTINCT PersonMasterHashKey AS PersonHashKey
       FROM [DataVault].[biz].[SALPerson]
       WHERE Score >= 0.5
       UNION ALL
       SELECT DISTINCT PersonDuplicateHashKey AS PersonHashKey
       FROM [DataVault].[biz].[SALPerson]
       WHERE Score < 0.5
) src
LEFT JOIN [DataVault].[biz].[SatCleansedPassenger] sat ON (
              sat.PersonHashKey = src.PersonHashKey AND sat.LoadEndDate IS NULL
)
WHERE
       sat.LoadEndDate IS NULL
       AND sat.PersonHashKey IS NOT NULL
```

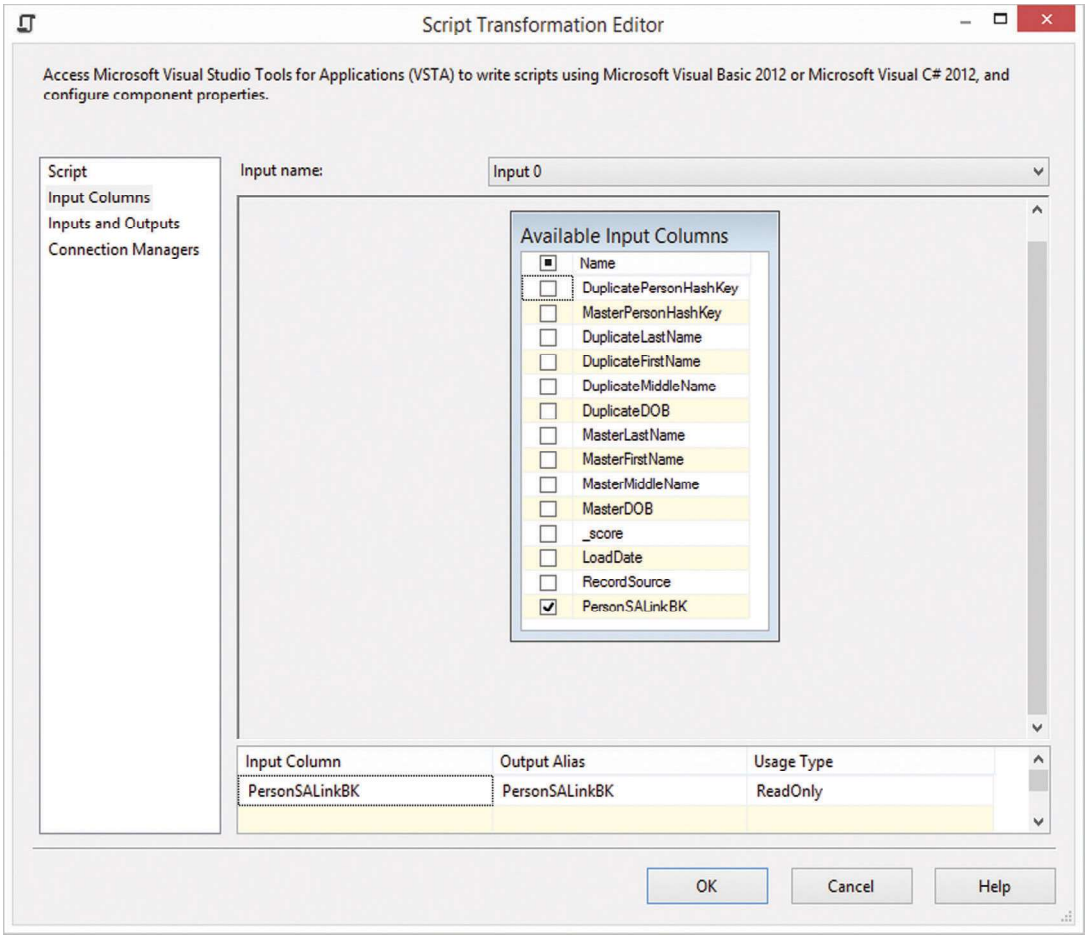

FIGURE 13.34

Configuring the input columns in the script transformation editor.

Instead of sourcing the virtual view from the actual hub and its satellites, the same-as link is used as the source and the satellites, which still hang off the hub, are joined to the same-as link to provide descriptive data.

The previous statement uses the score value from the fuzzy grouping transformation to dynamically select the mappings between the (potential) duplicate and the master record in the parent hub. By providing this score value, a power user can adjust this constant value (in the above statement 0.5) to include more or fewer mappings into the dimension: the lower the constant is set, the more business keys are mapped to a master in the dimension. A higher value means that the confidence of the algorithm should be higher before mapping takes place.

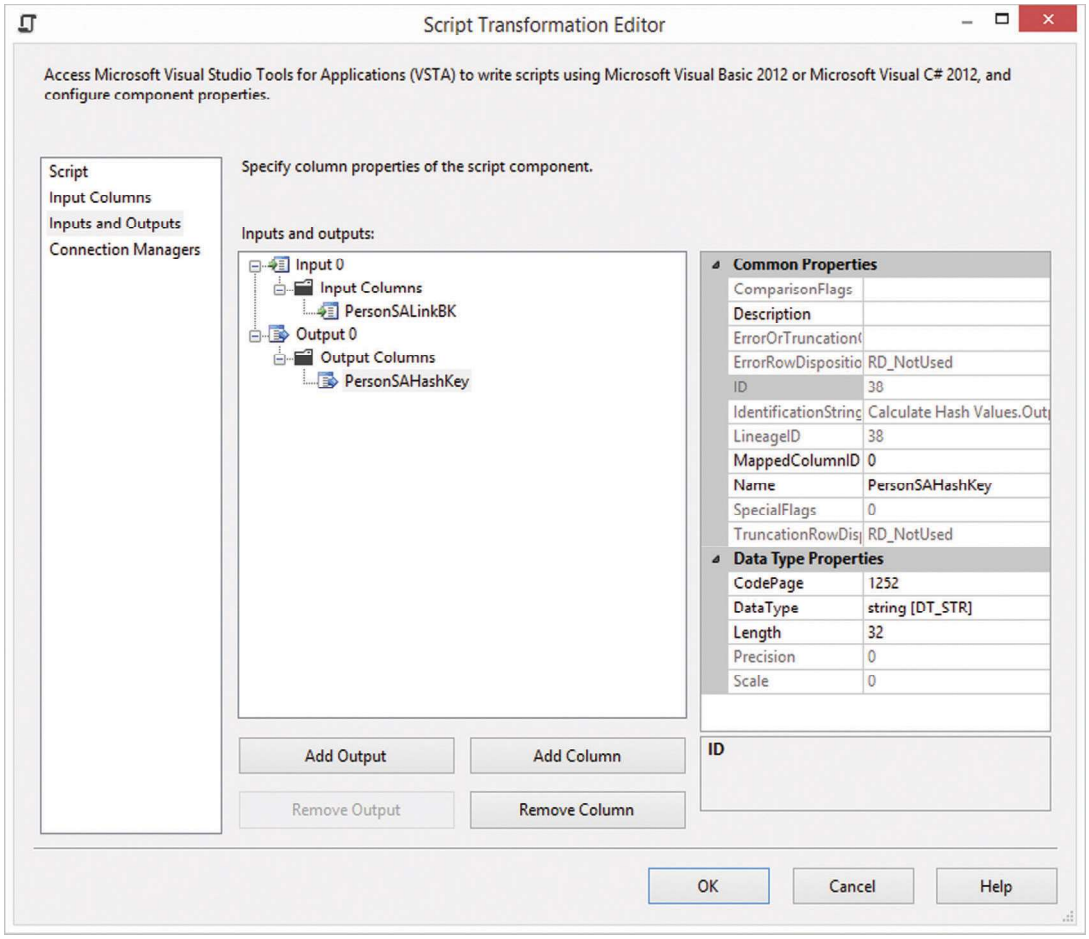

FIGURE 13.35

Configuring input and output columns in the script transformation editor.

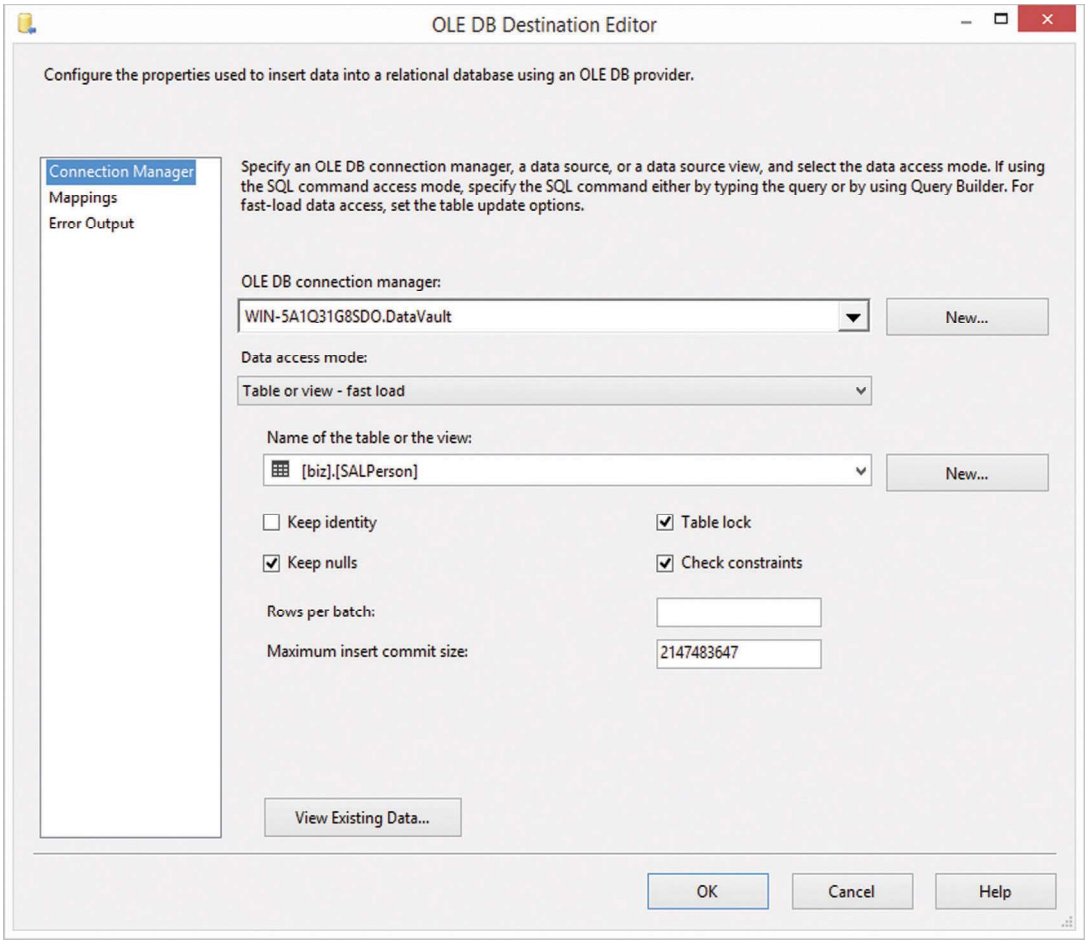

FIGURE 13.36

Configuring the same-as link target in the OLE DB destination editor.

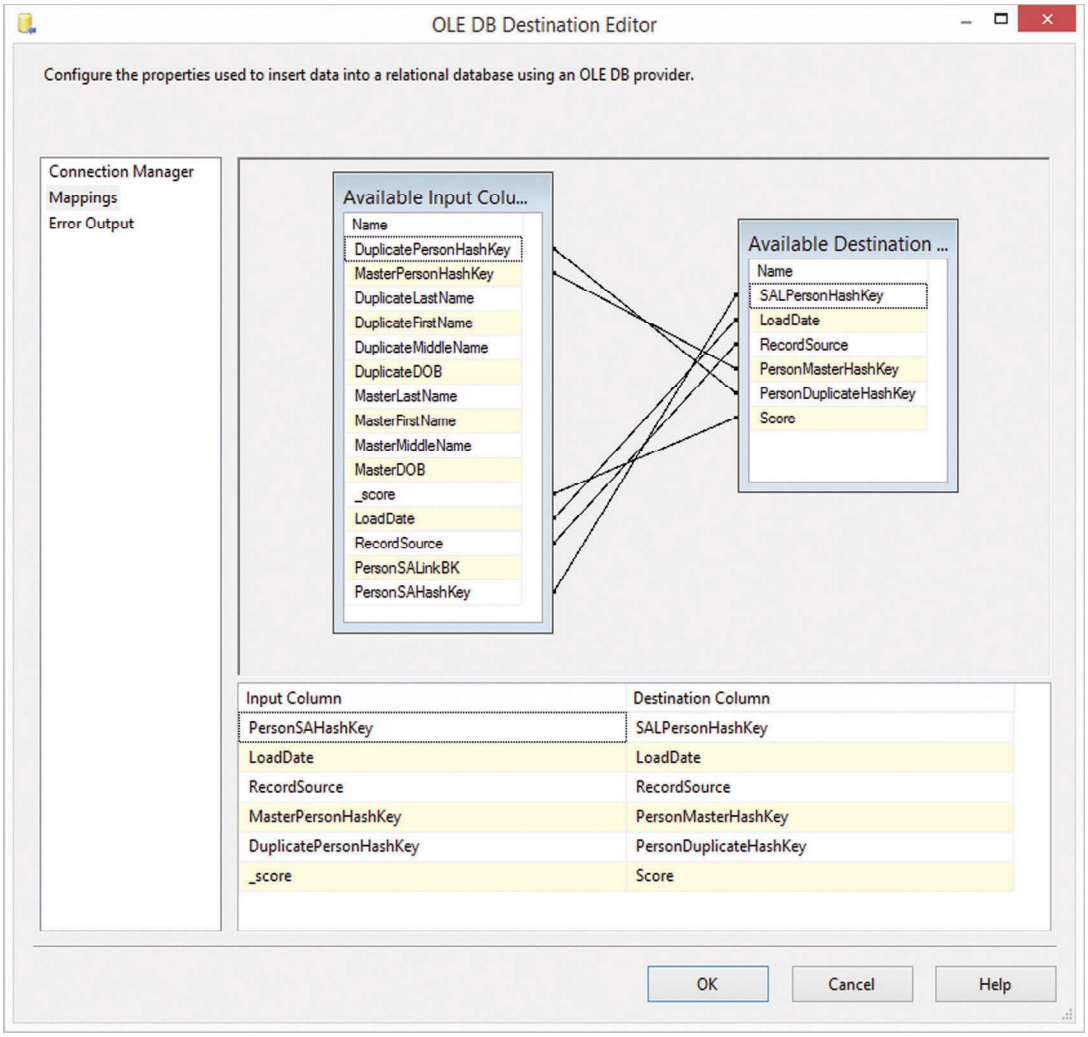

FIGURE 13.37

Configuring column mapping for the same-as link in the OLE DB destination editor.

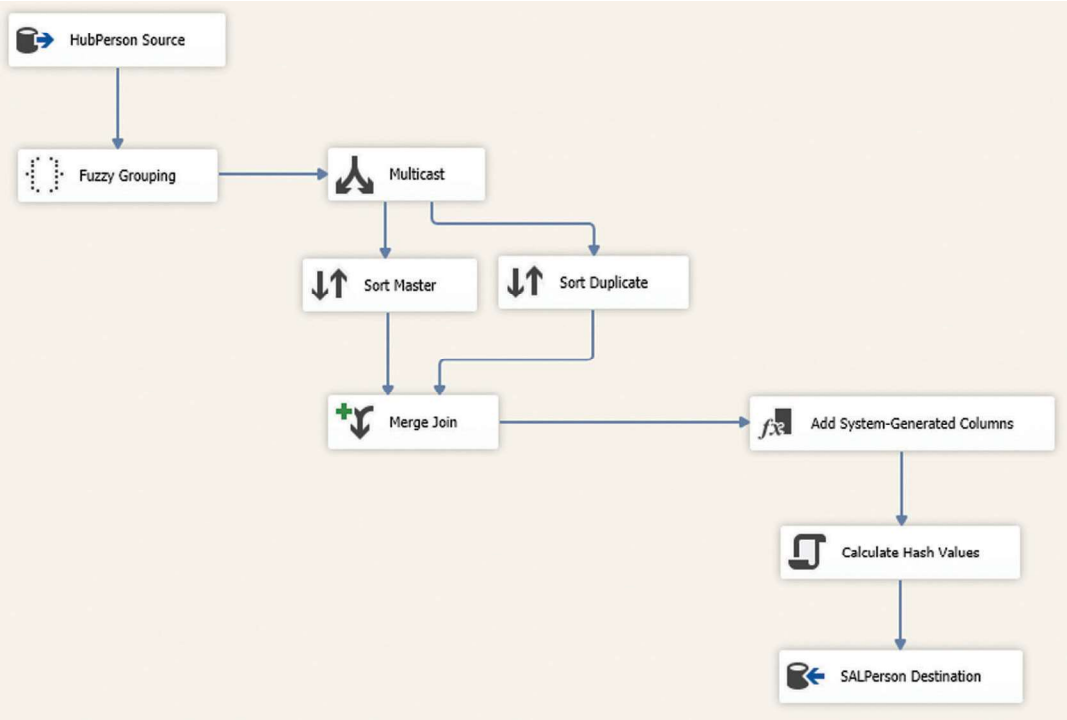

FIGURE 13.38

Data flow for de-duplication and loading same-as links.

REFERENCES

[1] David Loshin: "The Practitioner's Guide to Data Quality Improvement," pp. 4–6, 6f, 314, 314f, 270, 294f.
[2] Thomas C. Redman: "Data Quality for the Information Age", pp. 6f, 7ff, 8f, 9f, 10, 11, 22f.
[3] Larry P. English: "Information Quality Applied," pp. 251ff, 329, 332, 338, 345ff, 348ff, 351ff, 353, 356ff.
[4] Fisher et al. "Introduction to Information Quality," pp. 236f, 238f.
[5] Scott Ambler: "Refactoring Databases," p. 24f.
[6] Larry P. English: "Improving Data Warehouse and Business Information Quality," pp. 252, 260f, 261, 262, 267f, 274, 257ff.
[7] "DAMA Guide to the Data Management Body of Knowledge," pp. 305, 311.
[8] CDC Immunization Information Systems (IIS): "Deduplication Toolkit," retrieved from http://www.cdc.gov/vaccines/programs/iis/technical-guidance/deduplication.html.
[9] Leonard et al. "SQL Server 2012 Integration Services Design Patterns," p. 103f.
[10] "DAMA Guide to the Data Management Body of Knowledge," p. 310.
[11] https://msdn.microsoft.com/en-us/library/ms190768.aspx.

CHAPTER

LOADING THE DIMENSIONAL INFORMATION MART

14

Once the raw data has been loaded from the operational source systems into the Raw Data Vault, the next step is to process the raw data and load the results from this processing into the information marts. This chapter covers both steps.

14.1 USING THE BUSINESS VAULT AS AN INTERMEDIATE TO THE INFORMATION MART

The Business Vault serves as an intermediate between the Raw Data Vault and information marts. By doing so, it stores intermediate results from processed (soft) business rules that are stored for reusability. The next sections provide examples for implementing reusable business logic and storing the results for later usage when loading one or multiple information marts.

14.1.1 COMPUTED SATELLITE

Computed satellites are one of the artifacts in the Data Vault architecture to implement soft business rules. They are also used a lot for data quality cleansing, as described in the next chapter.

In many cases, computed satellites are provided in a virtual manner, using SQL views. The following DDL creates a computed satellite that hangs off **HubAirport** and provides the cultural region of the airport's location:

```
CREATE VIEW [biz].[SatDestAirportCulturalRegion] AS
SELECT
       [AirportHashKey]
      ,[LoadDate]
      ,[LoadEndDate]
      ,'SR9376' AS [RecordSource]
      ,[HashDiff]
      ,[DestCityName]
      ,[DestState]
      ,[DestStateName]
      ,[DestCityMarketID]
      ,[DestStateFips]
      ,[DestWac]
      ,CASE
```

```
            WHEN DestState IN ('CT', 'ME', 'MA', 'NH', 'RI', 'VT') THEN 'New England'
            WHEN DestState IN ('DE', 'MD', 'NJ', 'NY', 'PA', 'DC') THEN 'Mid Atlantic'
            WHEN DestState IN ('AL', 'AR', 'FL', 'GA', 'KY', 'LA') THEN 'The South'
            WHEN DestState IN ('MS', 'NC', 'SC', 'TN', 'VA', 'WV') THEN 'The South'
            WHEN DestState IN ('IL', 'IN', 'IA', 'KS', 'MI', 'MN') THEN 'Midwest'
            WHEN DestState IN ('MO', 'NE', 'ND', 'OH', 'SD', 'WI') THEN 'Midwest'
            WHEN DestState IN ('AZ', 'NM', 'OK', 'TX') THEN 'The Southwest'
            WHEN DestState IN ('AK', 'CO', 'CA', 'HI', 'ID', 'MT') THEN 'The West'
            WHEN DestState IN ('NV', 'OR', 'UT', 'WA', 'WY') THEN 'The West'
            ELSE NULL
    END AS CulturalRegion
FROM
    [DataVault].[raw].[SatDestAirport] src
```

The example satellite is based on an already existing satellite in the Raw Data Vault and introduces a computed attribute **CulturalRegion** based on the airport's state [1]. Essentially, this computed attribute is based on a mapping between the state abbreviation and a hard-coded mapping rule. Note that there are better options to implement such mapping: the use of analytical master data allows the business user to take control over this mapping and change it without IT involvement if necessary.

Another consideration is the use of the **record source** attribute. Because the computed satellite has changed the data to some extent, the record source is not the original source system anymore. Instead, the identifier of the soft business rule in the meta mart is used because the computed satellite implements the soft business rule. To some extent, the data was generated by the soft business rule and not by the source system (however, the generation is based on the data from the source system). Using the soft rule identifiers is only one option for the record source. Chapter 4, Data Vault 2.0 Modeling, has already stated that the record source attribute is an attribute that serves a debugging purpose. The IT organization should use it in the best manner that serves this purpose. Another choice would be to set the record source to "SYSTEM" or leave the original record source value. Throughout the book, we will set a soft business rule identifier whenever data was changed only slightly. If data was only filtered, we'll leave the original record source (because data values haven't changed).

There are multiple advantages of implementing soft business rules using virtualized computed satellites:

- **The implementation is simple and comprehensible:** usually, for each soft business rule definition, there should be an implementation. By using virtual satellites, this implementation is very compact. The alternative is to use a more complex ETL process, especially if the business rule is too complex to be covered in a SQL statement.
- **Quick development:** developing a SQL view is certainly faster than developing an ETL process with similar functionality (however, it might depend on the tools used).
- **Quick deployment:** it is also often faster to deploy a new or modified SQL view than deploying ETL processes.

These advantages are especially helpful when developing the data warehouse using the agile Data Vault 2.0 methodology. However, there are also some disadvantages:

- **Limited complexity:** if the soft business rule definition requires a too complex implementation, it might be required to split the implementation into multiple computed satellites (or other entities in the Business Vault) or implement the soft business rule using an ETL tools such as SSIS.

- **Performance**: while virtualization works well in many cases, some soft business rules require too much computing power, for example because many calculations are required or many joins are involved. In this case, it might be better to materialize the computed satellite by using ETL tools.

Computed satellites are covered more extensively in Chapter 13, Implementing Data Quality, when they are used for more complex data quality cleansing operations.

14.1.2 BUILDING AN EXPLORATION LINK

In some cases, however, virtualization is not an option. This is the case if external tools are involved, for example data mining tools such as the data mining extensions within SQL Server Analysis Services or another external tool that produces output that is calculated based on the raw data and not sourced directly from the source system (such as the output from Data Quality Services, which is covered in the next chapter).

An exploration link is such an example. The entity, the definition of which is described in detail in Chapter 5, Intermediate Data Vault Modeling, is used to store links that are artificially generated and not found in the source system. One way to implement such exploration links is by using an **association rule** algorithm, such as **Apriori** or **FP Growth**. Microsoft SQL Server provides such an algorithm in the Data Mining extension of SQL Server Analysis Services.

The following example implements and uses a data mining model that provides airlines with information about frequent connections between airports. The main answer provided by this model is:

Which airport destinations should be provided given the current connections offered by my airline?

The goal of this example is to provide new destinations that fit the current offerings of the airline. The suggestions are based on the offerings of all airlines. However, the data mining example in this chapter is far from being completely realistic:

- No data cleansing has been performed
- The loaded data in this chapter might be incomplete and not all patterns are covered
- Trends in the data set are not accounted for
- Errors might exist in the following code and the setup of the algorithm

In the end, the example has been developed quickly to show how to use a data mining model to populate an exploration link and not how to provide such predictions. Keep this in mind when using the model.

In order to use the example from this section, it is required to create a new **Analysis Server** database on the data warehouse infrastructure and to modify two configuration settings (Figure 14.1).

Open the properties dialog of the **Analysis Server** database using the context menu of the database and **General** tab. Make sure to check the option **show advanced (all) properties**, because some of the options that need to be modified are hidden by default.

The first setting that needs to be set is the **Data Mining \ AllowAdHocOpenRowsetQueries** to make sure that the model can be loaded directly from a source entity in the Raw Data Vault. The other option is to set up the providers that can be used in such an open rowset query by setting the **Data Mining \ AllowedProvidersInOpenRowset** to **"sqloledb,MSDataShape"** which enables both data providers.

Once these configuration options are modified, it is possible to create and train a data mining model on the server without using SSIS. Without modifying these settings, model training requires loading

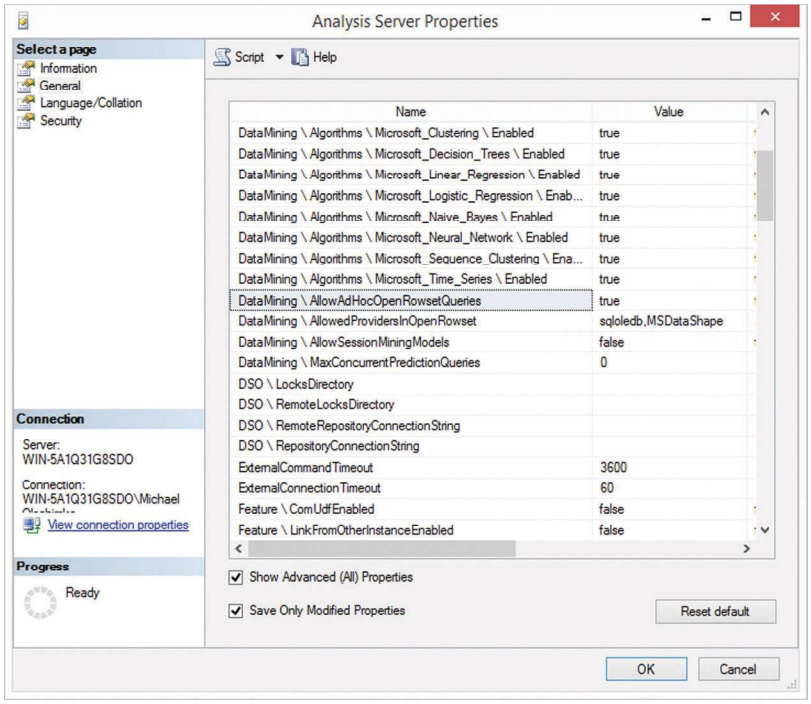

FIGURE 14.1

Analysis server properties.

the data using SSIS or another tool, and using the model for populating the exploration link becomes more complex as well.

Open a new **DMX query** in **Microsoft SQL Server Management Studio**. Enter the following statement to create a new data mining model using Microsoft's implementation of an association rule algorithm:

```
CREATE MINING MODEL ConnectionAssociation (
      CarrierOrigin text key,
      Origin text discrete predict,
      Connection table predict (
            Dest text key
      )
)
Using Microsoft_Association_Rules(Minimum_Support = 0.02, Minimum_Probability =
0.40);
```

This statement will create the model in the SQL Server Analysis Services database. The model will predict the destinations that are applicable for a given origin of the carrier. The parameters **minimum support** and **minimum probability** influence the number of frequent patterns that are accepted into the model. In the end, a pattern (here: flight connection) is considered as frequent if it is in at least 2%

of all flight connections in the source data (parameter **minimum support**). It will be included into the model if the pattern is able to predict a destination airport with at least 40% probability (parameter **minimum probability**) [2].

Once the model has been created, it needs to be trained using the existing connections of all airlines (this is actually realistic to do so, because all the data is available in public). The following DMX statement trains the model (finds frequent patterns) by inserting data into the data mining model:

```
INSERT INTO ConnectionAssociation (CarrierOrigin, Origin, Connection(Skip, Dest))
OPENROWSET('MSDataShape', 'data provider=SQLOLEDB;Server=localhost;UID=sa;PWD=datavault',
'Shape
{ SELECT DISTINCT
       carrier.Carrier+origin.AirportCode AS CarrierOrigin,
       origin.AirportCode AS Origin
  FROM
       [DataVault].[raw].[TLinkFlight] flight
INNER JOIN
       [DataVault].[raw].[HubCarrier] carrier ON carrier.CarrierHashKey =
flight.CarrierHashKey
INNER JOIN
       [DataVault].[raw].[HubAirportCode] origin ON origin.AirportCodeHashKey =
flight.OriginHashKey

        }
APPEND (
{      SELECT DISTINCT
       carrier.Carrier+origin.AirportCode AS CarrierOrigin,
       dest.AirportCode AS Dest
  FROM
       [DataVault].[raw].[TLinkFlight] flight
INNER JOIN
       [DataVault].[raw].[HubCarrier] carrier ON carrier.CarrierHashKey =
flight.CarrierHashKey
INNER JOIN
       [DataVault].[raw].[HubAirportCode] origin ON origin.AirportCodeHashKey =
flight.OriginHashKey
INNER JOIN
       [DataVault].[raw].[HubAirportCode] dest ON dest.AirportCodeHashKey =
flight.DestHashKey }
RELATE CarrierOrigin to CarrierOrigin) AS Connection');
```

Without going too much into detail, this command loads the data from the existing no-history link **TLinkFlight** in the Raw Data Vault. These connections are used to train the model.

After these two statements have been executed, the model is ready to use. It has found frequent patterns (flight connections). The next step is to create a SSIS data flow that uses these frequent patterns to predict additional destination airports for a given pair of carrier and origin airport ("if you are a carrier that operates out of X, you might be interested in offering flight connections to Y, based on the frequent patterns found").

Create a new data flow and insert an **OLE DB source** to the data flow. The data flow has to load all carrier and origin airport combinations from the Raw Data Vault source. Open the editor for the OLE DB source (Figure 14.2).

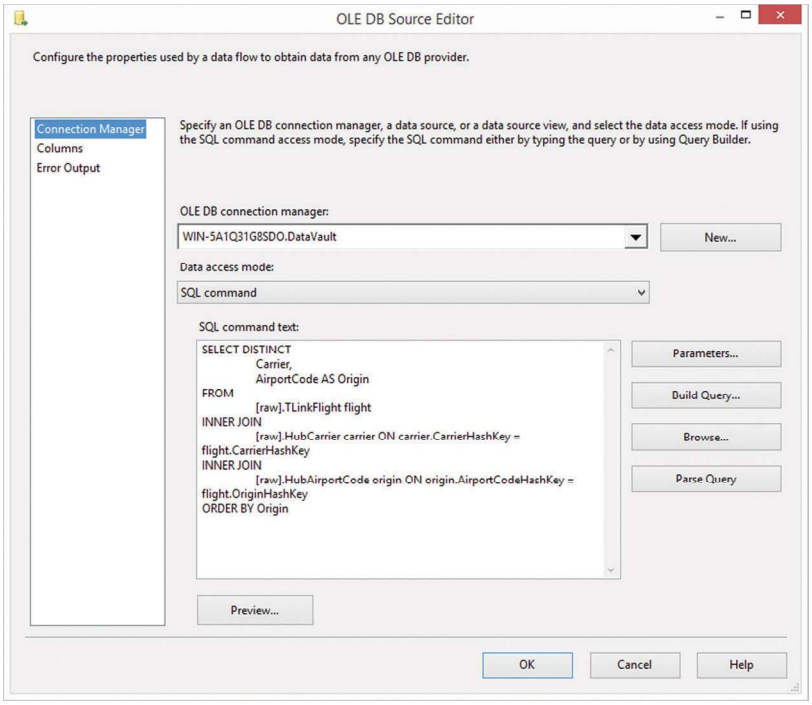

FIGURE 14.2

OLE DB source editor for the data mining source.

Enter the following SQL statement as SQL command text:

```
SELECT DISTINCT
       Carrier,
       AirportCode AS Origin
FROM
       [raw].TLinkFlight flight
INNER JOIN
       [raw].HubCarrier carrier ON carrier.CarrierHashKey = flight.CarrierHashKey
INNER JOIN
       [raw].HubAirportCode origin ON origin.AirportCodeHashKey = flight.OriginHashKey
ORDER BY
       Origin
```

This statement loads all combinations from the same no-history link in the Raw Data Vault. However, it doesn't take the current destination airports into account, because we're interested in retrieving additional destinations, currently not served by the carrier from this origin airport.

Add a **Data Mining Query** task to the data flow and connect it to the OLE DB source. Open the editor and create a new connection using the **New** button. The dialog is shown in Figure 14.3.

Connect to the Analysis Services database where the data mining model was created before. Close the dialog and select the mining structure in the data mining query transformation editor (Figure 14.4).

Make sure that the mining structure **ConnectionAssociation_Structure** and the mining model **ConnectionAssociation** are selected and switch to the query tab (Figure 14.5).

This statement loads all the connections of the carrier into the model and retrieves recommendations for additional flight destination for the currently offered set of flights. Enter the following SQL statement into the data mining query text box:

```
SELECT FLATTENED
  (PredictAssociation(ConnectionAssociation.Connection,3)) AS Recommendation
FROM
  ConnectionAssociation
NATURAL PREDICTION JOIN
SHAPE {
       @InputRowset
}
APPEND ({
       OPENROWSET(
              'SQLOLEDB',
              'Integrated Security=SSPI; Data Source=localhost;
               Initial Catalog=DataVault',
              'SELECT DISTINCT
                     origin.AirportCode AS Origin,
                     dest.AirportCode AS Dest
              FROM
                     [raw].TLinkFlight flight
              INNER JOIN
                     [raw].HubAirportCode origin ON origin.AirportCodeHashKey =
flight.OriginHashKey
              INNER JOIN
                     [raw].HubAirportCode dest ON dest.AirportCodeHashKey =
flight.DestHashKey
              ORDER BY
                     Origin'
       )
}
RELATE Origin TO Origin) AS Connection AS t
```

This statement basically provides the connections of the current carrier in @InputRowset to the trained model and asks for up to three recommended destinations based on the current connections of the carrier. For each carrier and origin, the data mining task returns the top three recommended destinations that the carrier should consider for the current origin airport. The recommendations are provided as records in the data flow.

Once the recommendations are in the data flow, the business keys should be hashed. The model was created with the business keys (instead of the hash keys) because the link is easier to produce afterwards: the primary hash key is calculated from the business key of the carrier, the origin airport and the recommended airport.

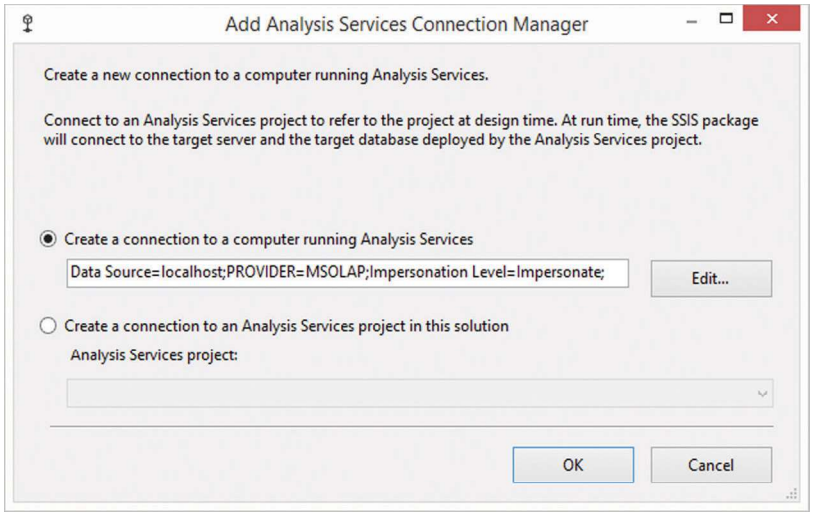

FIGURE 14.3

Add Analysis Services connection manager.

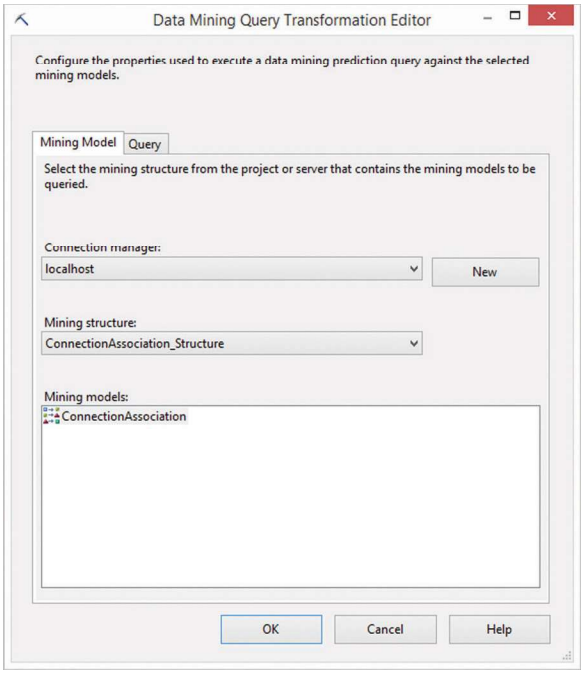

FIGURE 14.4

Data mining query transformation editor.

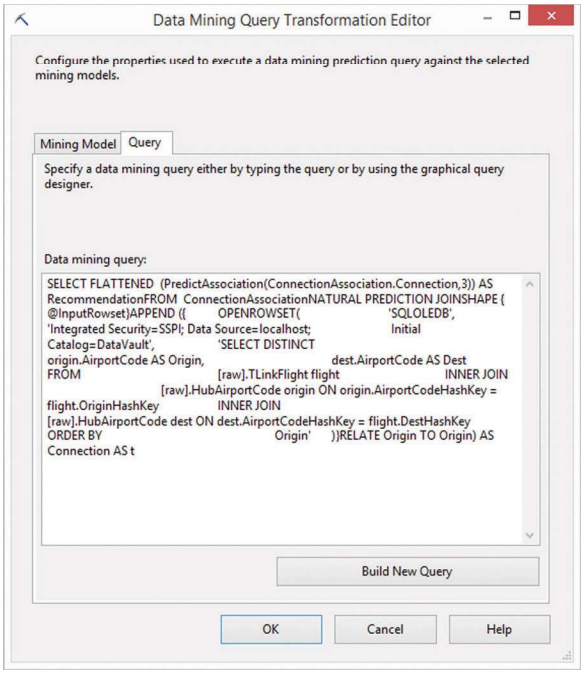

FIGURE 14.5

Editing the data mining query in the data mining query transformation editor.

The hashing approach follows the approach outlined in Chapter 11, Data Extraction: first, the columns are concatenated using a derived column transformation (Figure 14.6).

The derived columns are created using the provided expressions (Table 14.1).

Note the record source attribute, which is set to the identifier of the soft rule that defines the requirements and the process of the data mining task in the meta mart. The load date is retrieved from the SSIS variable dLoadDate in the User namespace. By doing so, all records in the target will use the same load date, which makes it easier to group the records in the target.

Because the hash key was used in the data mining model, the keys from the source data and the business key for the predicted destination airport are readily available for the hash key computation in the next step. Drag a **script transformation** to the data flow and connect it to the output path of the previous step (Figure 14.7).

This step also follows the process outlined in Chapter 11. Check all columns ending with HubBK or LinkBK as inputs to the script component. Copy the hashing script from Chapter 11 and compile the script. Create corresponding output columns on the next page, shown in Figure 14.8.

Ensure that each input column has a corresponding output column. Make sure that the data type is set to string and the length of the column is set to 32 characters (if MD5 hash keys are used). Close the script transformation editor and drag an OLE DB destination to the data flow. After connecting it to the existing components, open the editor (Figure 14.9).

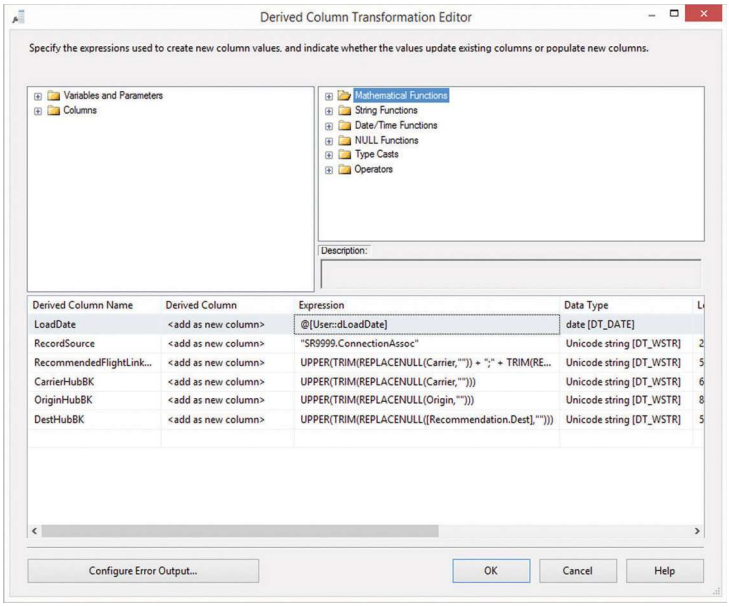

FIGURE 14.6

Derived column transformation editor.

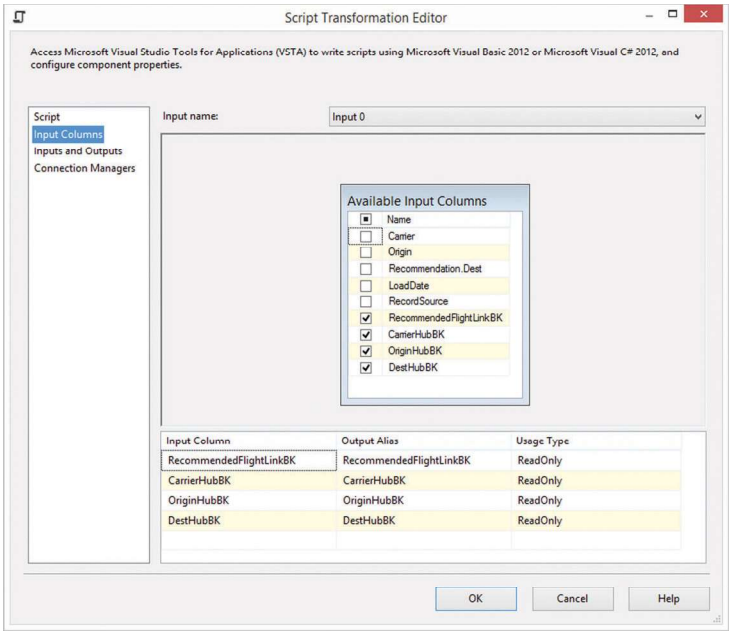

FIGURE 14.7

Script transformation editor.

Table 14.1 Derived Columns and Their Expressions

Derived Column Name	Expression
LoadDate	@[User::dLoadDate]
RecordSource	“SR9999.ConnectionAssoc”
RecommendedFlightLinkBK	UPPER(TRIM(REPLACENULL(Carrier,“”)) + “;” + TRIM(REPLACENULL(Origin,“”)) + “;” + TRIM(REPLACENULL([Recommendation.Dest],“”)))
CarrierHubBK	UPPER(TRIM(REPLACENULL(Carrier,“”)))
OriginHubBK	UPPER(TRIM(REPLACENULL(Origin,“”)))
DestHubBK	UPPER(TRIM(REPLACENULL([Recommendation.Dest],“”)))

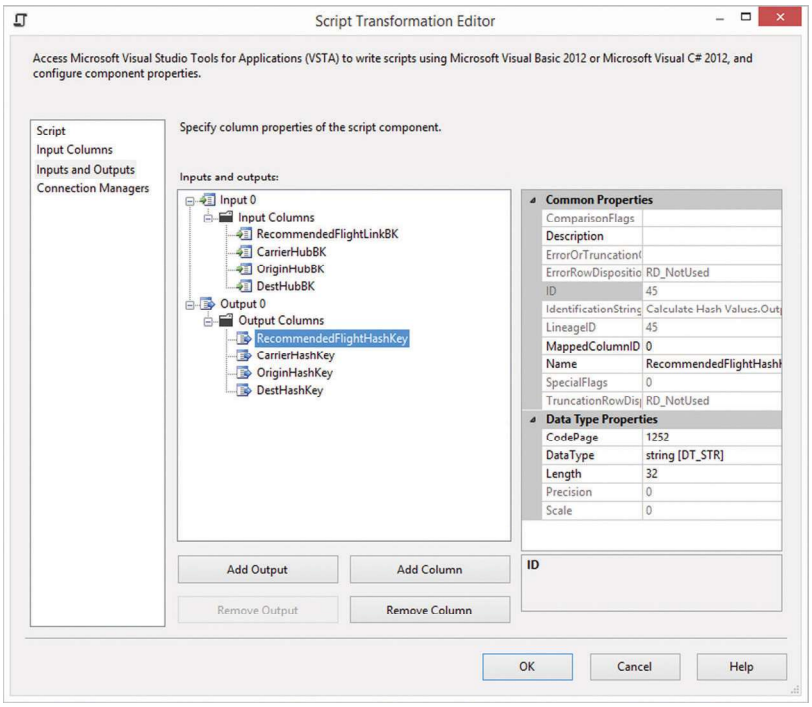

FIGURE 14.8

Input and output columns in the script transformation editor.

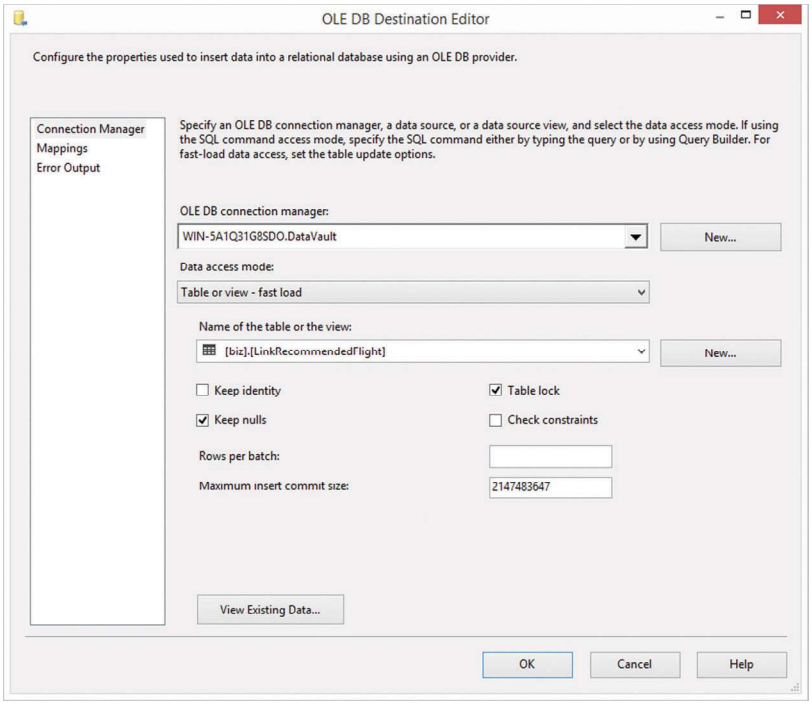

FIGURE 14.9

Setting up the OLE DB destination connection for exploration link.

The data is loaded into an exploration link table in the Business Vault. Create the table using the following DDL script:

```
CREATE TABLE [biz].[LinkRecommendedFlight](
       [RecommendedFlightHashKey] [char](32) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [CarrierHashKey] [char](32) NOT NULL,
       [OriginHashKey] [char](32) NOT NULL,
       [DestHashKey] [char](32) NOT NULL,
 CONSTRAINT [PK_LinkRecommendedFlight] PRIMARY KEY NONCLUSTERED
(
       [RecommendedFlightHashKey] ASC
) ON [INDEX],
 CONSTRAINT [UK_LinkRecommendedFlight] UNIQUE NONCLUSTERED
(
       [CarrierHashKey] ASC,
       [OriginHashKey] ASC,
       [DestHashKey] ASC
) ON [INDEX]
) ON [DATA]
```

Select the table in the dialog and make sure that **keep nulls** and **table lock** options are selected. Switch to the **mappings** page (Figure 14.10).

Make sure that each destination column has a corresponding source column and close the dialog. The data flow is completed and shown in Figure 14.11.

Because the frequent patterns in the model change over time, it is advisable to run this data flow frequently, for example once a week or month. Before running this data flow, the best choice is to truncate the target table first. In many cases, the patterns, if they change, are too different to run an update on the data. Using this approach is absolutely valid from an auditing perspective: the Business Vault may not be auditable and its tables can be reloaded at any point in time.

14.2 MATERIALIZING THE INFORMATION MART

The previous examples have shown how to implement the Business Vault. The ultimate goal, however, is to build and populate the information marts that will be used by the business users. The Business Vault plays only an intermediate step towards this goal, as outlined as in section 14.1.

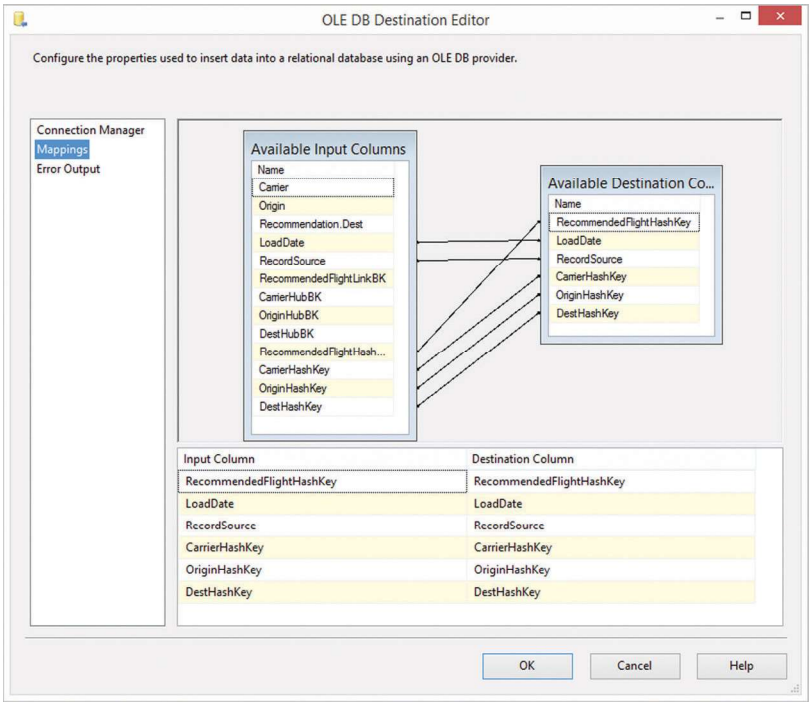

FIGURE 14.10

Mapping columns in the OLE DB destination editor.

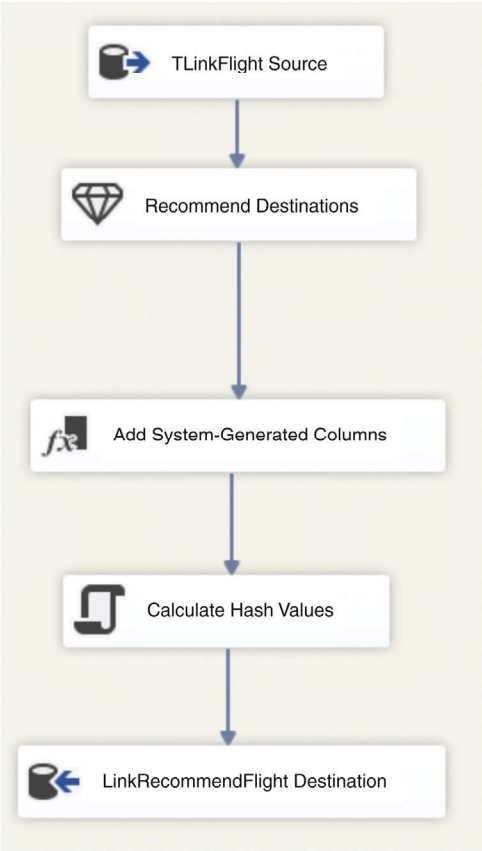

FIGURE 14.11

Data flow for loading an exploration link.

14.2.1 LOADING TYPE 1 DIMENSIONS

The first example creates and loads a Type 1 dimension, which provides no history of the dimension members but only the most current version of the descriptive data. In many cases, this involves all members ever loaded to the data warehouse, because historical data should be analyzed. For example, if a product has reached its end of line, it is still available in the product dimension because there might be historical data that depends on this now-outdated record. Or the product may be still active in the operational processes, for example in warranty requests from customers. Therefore, dimension members usually stay and are filtered later in the presentation layer, when not used by a specific data set.

The following DDL statement is used to create the dimension table in the information mart:

```
CREATE TABLE [dbo].[DimAirport](
	[AirportKey] [char](32) NOT NULL,
	[AirportCode] [nvarchar](3) NULL,
	[CityName] [nvarchar](100) NOT NULL,
	[State] [nvarchar](2) NOT NULL,
	[StateName] [nvarchar](100) NOT NULL,
	[CityMarketID] [int] NOT NULL,
	[StateFips] [smallint] NOT NULL,
	[Wac] [smallint] NOT NULL,
 CONSTRAINT [PK_DimAirport] PRIMARY KEY NONCLUSTERED
(
	[AirportKey] ASC
) ON [DATA]
) ON [DATA]
```

Note that **AirportCodeKey** identifies the rows of the dimension table, which is the hash key from the Data Vault 2.0 model. Using the hash key instead of a sequence number improves the provision of dimension tables because the hash key is already available in the Data Vault 2.0 model. This is described in more detail in Chapter 7, Dimensional Modeling.

In order to load this table for a Type 1 dimension, a simple INSERT statement in combination with a TRUNCATE TABLE statement is sufficient:

```
TRUNCATE TABLE FlightInformationMart.dbo.DimAirport;
GO

INSERT INTO FlightInformationMart.dbo.DimAirport
SELECT
	COALESCE(hub.AirportCodeHashKey, '') AS AirportKey
	,IIF(hub.AirportCode IS NOT NULL AND hub.AirportCode <> '',
		hub.AirportCode, '?') AS AirportCode
	,COALESCE(satd.[DestCityName], sato.[OriginCityName], 'Unknown') AS CityName
	,COALESCE(satd.[DestState], sato.[OriginState], '?') AS [State]
	,COALESCE(satd.[DestStateName], sato.[OriginStateName], 'Unknown') AS StateName
	,COALESCE(satd.[DestCityMarketID], sato.[OriginCityMarketID], 0) AS CityMarketID
	,COALESCE(satd.[DestStateFips], sato.[OriginStateFips], 0) AS StateFips
	,COALESCE(satd.[DestWac], sato.[OriginWac], 0) AS Wac
FROM
	DataVault.[raw].HubAirportCode hub
LEFT JOIN
	DataVault.[raw].SatDestAirport satd ON (
		satd.AirportHashKey = hub.AirportCodeHashKey
	)
LEFT JOIN
	DataVault.[raw].SatOriginAirport sato ON (
		sato.AirportHashKey = hub.AirportCodeHashKey
	)
WHERE
	satd.LoadEndDate IS NULL AND sato.LoadEndDate IS NULL;
GO
```

The table is first truncated and then completely reloaded in order to avoid dealing with updates. Another option is to use a MERGE statement if there are not many members in the dimension.

While this approach is a naïve approach, it demonstrates an advantage and characteristic of the Data Vault 2.0 model: many if not most dimensions are built by querying the data from a hub and joining descriptive data from one or more satellites. Each satellite might come from different source systems and it is part of the business logic in the dimension load to decide which satellite is representing the leading system if multiple systems provide contradicting raw data. Since only the latest version of the descriptive data should be included in Type 1 dimensions, a WHERE condition is applied on the satellite data to select only the active (most current) records. If the dimension should contain only members that are not deleted in the source system, the effectivity satellite on the hub should be taken into consideration by joining it in the query and adding a filter to remove deleted members.

However, the sources for dimensions are not limited to hubs and satellites. In fact, the data can come from multiple source entities in the Data Vault 2.0 model, including links. Therefore, when building dimension tables in the information mart, the first step is to identify the table (or set of tables) that provides the desired grain for the destination table. From our experience, the grain for dimension tables comes from a hub table in 80% of the cases. Other typical sources for building dimensions are same-as links because they are the outcome of de-duplication efforts. This will be demonstrated in the next chapter.

The complexity of the statement is due to the COALESCE statements that deal with potential airports that are only described in one satellite. In such a case, there is no descriptive data available from all satellites and the columns are NULL. This is also the reason why a LEFT JOIN was used, which is not the most frequently performed join operation in most databases, including Microsoft SQL Server. Section 14.3 explains how to use PIT tables in conjunction with ghost records to achieve INNER JOINs with equi-join conditions.

14.2.2 LOADING TYPE 2 DIMENSIONS

Joining the data becomes a bigger problem when dealing with Type 2 dimensions. In this case, multiple if not all versions of the descriptive data from the satellites should be sourced into the target. Before looking at the loading statement, review the DDL statement for creating a Type 2 dimension table that can be used in the dimensional model:

```
CREATE TABLE [dbo].[DimAirport2](
       [AirportKey] [char](32) NOT NULL,
       [AirportCodeHashKey] [char](32) NOT NULL,
       [LoadDate] [datetime2](7) NOT NULL,
       [LoadEndDate] [datetime2](7) NULL,
       [AirportCode] [nvarchar](3) NULL,
       [CityName] [nvarchar](100) NOT NULL,
       [State] [nvarchar](2) NOT NULL,
       [StateName] [nvarchar](100) NOT NULL,
       [CityMarketID] [int] NOT NULL,
       [StateFips] [smallint] NOT NULL,
       [Wac] [smallint] NOT NULL,
```

```
 CONSTRAINT [PK_DimAirport2] PRIMARY KEY NONCLUSTERED
(
       [AirportKey] ASC
) ON [INDEX],
 CONSTRAINT [UK_DimAirport2] UNIQUE NONCLUSTERED
(
       [AirportCodeHashKey] ASC,
       [LoadDate] ASC
) ON [INDEX]
) ON [DATA]
```

The table is an extended version of the same table used in the previous section. In Type 2 dimensions, the key column is not directly sourced from the hash key column in the Raw Data Vault 2.0 hub table. Instead, it is a value that is calculated from the business key and a date (for example the load date or a snapshot date introduced a little later). This way, every row gets a unique hash key assigned, which is required for identifying the record in the dimension.

Because the key by itself is not from the Raw Data Vault, additional attributes are required to identify not only the member (using **AirportCodeHashKey**) but also the right version (using **LoadDate**) when loading the fact table. Once both table types, the facts and the dimensions, are loaded, the key is used for joining the data.

Both the hash key and load date columns are also used when loading the Type 2 dimension table using the following, incremental approach:

```
INSERT INTO FlightInformationMart.dbo.DimAirport2
SELECT
    UPPER(CONVERT(char(32),
       HASHBYTES('MD5', CONCAT(
              UPPER(RTRIM(LTRIM(hub.AirportCode)))
              , ';'
              , CONVERT(NVARCHAR(30), src.LoadDate, 126)
       ))
    ,2)) AS AirportKey
    ,src.AirportHashKey AS AirportCodeHashKey
    ,src.LoadDate
    ,NULL AS LoadEndDate
    ,IIF(hub.AirportCode IS NOT NULL AND hub.AirportCode <> '',
       hub.AirportCode, '?') AS AirportCode
    ,COALESCE(satd.[DestCityName], sato.[OriginCityName], 'Unknown') AS CityName
    ,COALESCE(satd.[DestState], sato.[OriginState], '?') AS [State]
    ,COALESCE(satd.[DestStateName], sato.[OriginStateName], 'Unknown') AS StateName
    ,COALESCE(satd.[DestCityMarketID], sato.[OriginCityMarketID], 0) AS CityMarketID
    ,COALESCE(satd.[DestStateFips], sato.[OriginStateFips], 0) AS StateFips
    ,COALESCE(satd.[DestWac], sato.[OriginWac], 0) AS Wac
FROM
       (SELECT AirportHashKey, LoadDate FROM DataVault.[raw].SatOriginAirportMod
        UNION
        SELECT AirportHashKey, LoadDate FROM DataVault.[raw].SatDestAirportMod) src
```

```
INNER JOIN DataVault.[raw].HubAirportCode hub ON (
       src.AirportHashKey = hub.AirportCodeHashKey
)
LEFT OUTER JOIN DataVault.[raw].SatDestAirportMod satd ON (
       satd.AirportHashKey = src.AirportHashKey
       AND src.LoadDate BETWEEN
              satd.LoadDate AND COALESCE(satd.LoadEndDate, '9999-12-31 23:59:59.999')
)
LEFT OUTER JOIN DataVault.[raw].SatOriginAirportMod sato ON (
       sato.AirportHashKey = src.AirportHashKey
       AND src.LoadDate BETWEEN
              sato.LoadDate AND COALESCE(sato.LoadEndDate, '9999-12-31 23:59:59.999')
)
WHERE NOT EXISTS (
       SELECT
              1
       FROM
              FlightInformationMart.dbo.DimAirport2 tgt
       WHERE
              src.AirportHashKey = tgt.AirportCodeHashKey
                     AND src.LoadDate = tgt.LoadDate
)
```

New records from the Raw Data Vault source are loaded into the target, based on the hash key and the load date. Because the target is a Type 2 dimension table, all versions from the Raw Data Vault that are not currently in the destination should be loaded into the dimension table. The hub that provided the grain in the last section is not a good candidate for the FROM clause because it provides the wrong grain. Instead, the versions of descriptive data are stored in the satellites – actually, spread over all satellite tables that depend on the hub. Therefore, the grain for this statement is provided by a subquery in the FROM statement that returns the potential **load date** and **hash key** combinations. For each combination that is not yet in the target, which is checked by the WHERE condition, the corresponding descriptive data is sourced from each satellite using a LEFT OUTER JOIN. While the processing of the descriptive data in the SELECT clause is similar to the statement for Type 1 dimension loads in the previous section, the join condition is much more complex. This is true not only from a reader's perspective, but also from the perspective of the SQL optimizer.

To achieve higher performance of Type 2 dimension loads, an equi-join is the preferred solution. However, if performance is not an issue because there are not many joins involved in the actual loading statement or if the number of records is small, this approach might work very well (at least for the majority of dimension tables). However, note that, due to independent changes to the satellites and the naïve approach implemented for selecting the descriptive data from the source, it might happen that the Type 2 dimension provides two unchanged entries, for example if the secondary satellite (in the previous example, the source airport) changes but the primary satellite (destination airport) overrides

this change. Depending on the technology used for the information mart (or dependent OLAP cubes), this might pose a problem.

To complete the loading process, the load end date needs to be maintained. This can be done using an end-dating SQL statement, similar to the one for end-dating Data Vault 2.0 satellite tables:

```
UPDATE DimAirport2 SET
       LoadEndDate = (
              SELECT
                     DATEADD(ss,-1,MIN(z.LoadDate))
              FROM
                     DimAirport2 z
              WHERE
                     z.AirportCodeHashkey = a.AirportCodeHashkey
              AND
                     z.LoadDate > a.LoadDate
       )
FROM
       DimAirport2 a
WHERE
       LoadEndDate IS NULL AND AirportCodeHashkey = ?
```

In order to maintain the performance of the statement, it should be applied per hash key in the dimension. Otherwise, the scalability of the statement could be limited if the number of records in the satellite table is large.

14.2.3 LOADING FACT TABLES

The following DDL statement creates a fact table with many columns and references to dimension tables:

```
CREATE TABLE [dbo].[FactFlight](
       [CarrierKey] [char](32) NOT NULL,
       [FlightNumKey] [char](32) NOT NULL,
       [TailNumKey] [char](32) NOT NULL,
       [OriginKey] [char](32) NOT NULL,
       [DestKey] [char](32) NOT NULL,
       [FlightDateKey] int NOT NULL,
       [FlightDate] [datetime2](7) 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_FactFlight] PRIMARY KEY NONCLUSTERED
(
        [CarrierKey] ASC,
        [FlightNumKey] ASC,
        [TailNumKey] ASC,
        [OriginKey] ASC,
        [DestKey] ASC,
        [FlightDateKey] ASC
) ON [INDEX]
) ON [DATA]
```

The fact table itself is not identified by a key because no other tables depend on it, as described in Chapter 7. Instead, the primary key is based on the hash values of the dimension tables including the flight date key because they are defining the grain of the fact table.

Note that the FlightDateKey is an integer value for better readability of the date value. This approach follows the standard approach for building and using date dimensions (covered in the next chapter).

The following statement populates the data into the fact table:

```
INSERT INTO FlightInformationMart.dbo.FactFlight
SELECT
      link.CarrierHashKey AS CarrierKey
      ,link.FlightNumHashKey AS FlightNumKey
      ,link.TailNumHashKey AS TailNumKey
      ,UPPER(CONVERT(char(32), HASHBYTES('MD5',
         CONCAT(UPPER(RTRIM(LTRIM(hubOrigin.AirportCode))), ';', CONVERT(NVARCHAR(30),
         SatOrigin.LoadDate, 126))),2)) AS OriginKey
      ,UPPER(CONVERT(char(32), HASHBYTES('MD5',
         CONCAT(UPPER(RTRIM(LTRIM(hubDest.AirportCode))), ';', CONVERT(NVARCHAR(30),
         SatDest.LoadDate, 126))),2)) AS DestKey
      ,DATEPART(YEAR,
         link.FlightDate)*10000+DATEPART(MONTH, link.FlightDate)*100
         +DATEPART(DAY, link.FlightDate) AS FlightDateKey
      ,link.[FlightDate]
      ,sat.[Year]
      ,sat.[Quarter]
      ,sat.[Month]
      ,sat.[DayOfMonth]
      ,sat.[DayOfWeek]
      ,sat.[CRSDepTime]
      ,sat.[DepTime]
      ,sat.[DepDelay]
      ,sat.[DepDelayMinutes]
      ,sat.[DepDel15]
      ,sat.[DepartureDelayGroups]
      ,sat.[DepTimeBlk]
      ,sat.[TaxiOut]
      ,sat.[WheelsOff]
      ,sat.[WheelsOn]
      ,sat.[TaxiIn]
      ,sat.[CRSArrTime]
      ,sat.[ArrTime]
      ,sat.[ArrDelay]
      ,sat.[ArrDelayMinutes]
      ,sat.[ArrDel15]
      ,sat.[ArrivalDelayGroups]
      ,sat.[ArrTimeBlk]
      ,sat.[Cancelled]
      ,sat.[CancellationCode]
      ,sat.[Diverted]
      ,sat.[CRSElapsedTime]
      ,sat.[ActualElapsedTime]
      ,sat.[AirTime]
      ,sat.[Flights]
      ,sat.[Distance]
      ,sat.[DistanceGroup]
      ,sat.[CarrierDelay]
      ,sat.[WeatherDelay]
      ,sat.[NASDelay]
      ,sat.[SecurityDelay]
      ,sat.[LateAircraftDelay]
      ,sat.[FirstDepTime]
```

```
    ,sat.[TotalAddGTime]
    ,sat.[LongestAddGTime]
FROM
    [DataVault].[raw].[TLinkFlight] link
INNER JOIN [DataVault].[raw].[HubAirportCode] HubOrigin ON (
            HubOrigin.AirportCodeHashKey = link.OriginHashKey
)
INNER JOIN [DataVault].[raw].[SatOriginAirportMod2] SatOrigin ON (
            SatOrigin.AirportHashKey = link.OriginHashKey
            AND link.LoadDate BETWEEN
                  SatOrigin.LoadDate
                  AND COALESCE(SatOrigin.LoadEndDate, '9999-12-31 23:59:59.999')
)
INNER JOIN [DataVault].[raw].[HubAirportCode] HubDest ON (
            HubDest.AirportCodeHashKey = link.DestHashKey
)
INNER JOIN [DataVault].[raw].[SatDestAirportMod2] SatDest ON (
            SatDest.AirportHashKey = link.DestHashKey
            AND link.LoadDate BETWEEN
                  SatDest.LoadDate
                  AND COALESCE(SatDest.LoadEndDate, '9999-12-31 23:59:59.999')
)
INNER JOIN DataVault.[raw].TSatFlight sat ON (
            sat.FlightHashKey = link.FlightHashKey
)
WHERE
            NOT EXISTS (SELECT
                        1
                  FROM
                        FlightInformationMart.dbo.FactFlight tgt
                  WHERE
                        COALESCE(UPPER(CONVERT(char(32), HASHBYTES('MD5',

                        CONCAT(UPPER(RTRIM(LTRIM(hubOrigin.AirportCode))), ';',
      CONVERT(NVARCHAR(30), SatOrigin.LoadDate, 126))),2)),
      REPLICATE('0', 32)) = tgt.OriginKey
      AND COALESCE(UPPER(CONVERT(char(32), HASHBYTES('MD5',
      CONCAT(UPPER(RTRIM(LTRIM(hubDest.AirportCode))), ';',
      CONVERT(NVARCHAR(30), SatDest.LoadDate, 126))),2)),
      REPLICATE('0', 32)) = tgt.DestKey
      AND link.CarrierHashKey = tgt.CarrierKey
      AND link.FlightNumHashKey = tgt.FlightNumKey
      AND link.TailNumHashKey = tgt.TailNumKey
      AND link.FlightDate = tgt.FlightDate
)
```

The facts are loaded from the nonhistorized link **TLinkFlights**. Each Type 2 dimension that is used by the target requires the recalculation of the key, which is based on the business key and the load date in the above example with standard satellites. The key is then stored in the fact table to support later joins between the fact table and dimension tables. Section 14.3.2 introduces a method for reusing such calculations. Because the business key is required, the hub has to be joined.

The complex calculation is not necessary when using Type 1 dimensions because the **key** is the same as the hash key in the Raw Data Vault. Note the INNER JOIN, which requires that the dimension table provide a record, even in the NULL case. For this reason, the dimension table should include an unknown record.

Another approach to avoid the complex calculation is to retrieve the key from the dimension in the information mart. However, this would add additional and unnecessary dependencies to the loading processes, which should be avoided in order to improve the parallelization of the loading processes that load the information marts.

Joining the record from the satellite is based on the hash key from the hub. However, the hash key by itself is not sufficient for the join. Instead, the **LoadDate** and **LoadEndDate** from the satellite are used to find the active record for the fact record.

The WHERE clause is required to support incremental loading and includes all elements of the alternate key from the target table, which is based on the elements of the source table, in this case. It also includes the transaction date (**FlightDate**). In other cases, the transaction ID would be used because it is part of the alternate key. If a record with this combination is not found in the target, it is loaded from the source to the target. Note that the key for the originating and destination airports are recalculated again in order to support the condition. Again, section 14.3.2 introduces a method for reusing such calculations and avoiding this recalculation.

Additional descriptive data that is not included in the source link is sourced from the no-history satellite **TSatFlight** which hangs off **TLinkFlight**. This satellite is joined on the hash key only because there should be only one record per transaction, which eases the integration of both data sources in this loading statement.

The previously mentioned unknown record can be populated using the following statement into the dimension tables:

```
INSERT INTO [dbo].[DimCarrier]
           ([CarrierKey]
           ,[Carrier]
           ,[Code]
           ,[Name]
           ,[Corporate Name]
           ,[Abbreviation]
           ,[Unique Abbreviation]
           ,[Group_Code]
           ,[Region_Code]
           ,[Satisfaction Rank]
           ,[Sort Order]
           ,[External Reference]
           ,[Comments])
     VALUES (
                 REPLICATE('0', 32)
                 , '?'
                 , 0
                 , 'Unknown'
                 , 'Unknown'
                 , '?'
                 , '?'
                 , '?'
                 , '?'
                 , 9999
                 , 0
                 , ''
                 , ''
                );
```

This unknown record is required due to the use of the `REPLICATE('0', 32)` statement when loading the fact table. If this record is missing, the integrity of the information mart is compromised.

14.2.4 LOADING AGGREGATED FACT TABLES

It is also possible to change the grain when loading the data from the Data Vault model into the information mart. For example, the following DDL creates a **FactConnection** fact table that removes carrier and tail number references and aggregates the measures of the source table:

```
CREATE TABLE [dbo].[FactConnection](
	[CarrierKey] [char](32) NOT NULL,
	[OriginKey] [char](32) NOT NULL,
	[DestKey] [char](32) NOT NULL,
	[FlightDateKey] [int] NOT NULL,
	[FlightDate] [datetime2](7) NOT NULL,
	[Year] [smallint] NOT NULL,
	[Quarter] [smallint] NOT NULL,
	[Month] [smallint] NOT NULL,
	[DayOfMonth] [smallint] NOT NULL,
	[DayOfWeek] [smallint] NOT NULL,
	[SumDepDelay] [int] NOT NULL,
	[SumDepDelayMinutes] [int] NOT NULL,
	[SumTaxiOut] [int] NOT NULL,
	[SumWheelsOff] [int] NOT NULL,
	[SumWheelsOn] [int] NOT NULL,
	[SumTaxiIn] [int] NOT NULL,
	[SumArrDelay] [int] NOT NULL,
	[SumArrDelayMinutes] [int] NOT NULL,
	[SumCancelled] [int] NOT NULL,
	[SumDiverted] [int] NOT NULL,
	[SumAirTime] [int] NOT NULL,
	[SumFlights] [int] NOT NULL,
	[SumDistance] [int] NOT NULL,
	[SumCArrierDelay] [int] NOT NULL,
	[SumWeatherDelay] [int] NOT NULL,
	[SumNASDelay] [int] NOT NULL,
	[SumSecurityDelay] [int] NOT NULL,
	[SumLateAircraftDelay] [int] NOT NULL,
CONSTRAINT [PK_FactConnection] PRIMARY KEY NONCLUSTERED
(
	[CarrierKey] ASC,
	[OriginKey] ASC,
	[DestKey] ASC,
	[FlightDateKey] ASC
) ON [INDEX]
) ON [DATA]
```

In order to incrementally load the data into the new aggregated fact table, the following statement is executed once a day:

```
INSERT INTO FactConnection
SELECT
      link.CarrierHashKey AS CarrierKey
      ,UPPER(CONVERT(char(32), HASHBYTES('MD5',
             CONCAT(UPPER(RTRIM(LTRIM(hubOrigin.AirportCode))), ';',
             CONVERT(NVARCHAR(30), SatOrigin.LoadDate, 126))),2)) AS OriginKey
      ,UPPER(CONVERT(char(32), HASHBYTES('MD5',
             CONCAT(UPPER(RTRIM(LTRIM(hubDest.AirportCode))), ';', CONVERT(NVARCHAR(30),
             SatDest.LoadDate, 126))),2)) AS DestKey
      ,DATEPART(YEAR, link.FlightDate)*10000
             +DATEPART(MONTH, link.FlightDate)*100
             +DATEPART(DAY, link.FlightDate) AS FlightDateKey
      ,link.[FlightDate]
      ,sat.[Year]
      ,sat.[Quarter]
      ,sat.[Month]
      ,sat.[DayOfMonth]
      ,sat.[DayOfWeek]
      ,SUM(sat.[DepDelay]) AS SumDepDelay
      ,SUM(sat.[DepDelayMinutes]) AS SumDepDelayMinutes
      ,SUM(sat.[TaxiOut]) AS SumTaxiOut
      ,SUM(sat.[WheelsOff]) AS SumWheelsOff
      ,SUM(sat.[WheelsOn]) AS SumWheelsOn
      ,SUM(sat.[TaxiIn]) AS SumTaxiIn
      ,SUM(sat.[ArrDelay]) AS SumArrDelay
      ,SUM(sat.[ArrDelayMinutes]) AS SumArrDelayMinutes
      ,SUM(CASE WHEN sat.Cancelled=1 THEN 1 ELSE 0 END) AS SumCancelled
      ,SUM(CASE WHEN sat.Diverted=1 THEN 1 ELSE 0 END) AS SumDiverted
      ,SUM(sat.[AirTime]) AS SumAirTime
      ,SUM(sat.[Flights]) AS SumFlights
      ,SUM(sat.[Distance]) AS SumDistance
      ,SUM(sat.[CarrierDelay]) AS SumCArrierDelay
      ,SUM(sat.[WeatherDelay]) AS SumWeatherDelay
      ,SUM(sat.[NASDelay]) AS SumNASDelay
      ,SUM(sat.[SecurityDelay]) AS SumSecurityDelay
      ,SUM(sat.[LateAircraftDelay]) AS SumLateAircraftDelay
  FROM [DataVault].[raw].[TLinkFlight] link
  INNER JOIN [DataVault].[raw].[HubAirportCode] HubOrigin ON (
             HubOrigin.AirportCodeHashKey = link.OriginHashKey
  )
  INNER JOIN [DataVault].[raw].[SatOriginAirportMod2] SatOrigin ON (
             SatOrigin.AirportHashKey = link.OriginHashKey
             AND link.LoadDate BETWEEN SatOrigin.LoadDate AND
                    COALESCE(SatOrigin.LoadEndDate, '9999-12-31 23:59:59.999')
  )
  INNER JOIN [DataVault].[raw].[HubAirportCode] HubDest ON (
             HubDest.AirportCodeHashKey = link.DestHashKey
  )
  INNER JOIN [DataVault].[raw].[SatDestAirportMod2] SatDest ON (
             SatDest.AirportHashKey = link.DestHashKey
             AND link.LoadDate BETWEEN SatDest.LoadDate AND
```

```
                    COALESCE(SatDest.LoadEndDate, '9999-12-31 23:59:59.999')
)
INNER JOIN DataVault.[raw].TSatFlight sat ON (
     sat.FlightHashKey = link.FlightHashKey
)
WHERE
            NOT EXISTS (SELECT
                        1
                    FROM
                        FlightInformationMart.dbo.FactConnection tgt
                    WHERE
                        COALESCE(UPPER(CONVERT(char(32), HASHBYTES('MD5',
                        CONCAT(UPPER(RTRIM(LTRIM(hubOrigin.AirportCode))), ';',
                        CONVERT(NVARCHAR(30), SatOrigin.LoadDate, 126))),2)),
                        REPLICATE('0', 32)) = tgt.OriginKey
                        AND COALESCE(UPPER(CONVERT(char(32), HASHBYTES('MD5',
                        CONCAT(UPPER(RTRIM(LTRIM(hubDest.AirportCode))), ';',
                        CONVERT(NVARCHAR(30), SatDest.LoadDate, 126))),2)),
                        REPLICATE('0', 32)) = tgt.DestKey
                        AND link.CarrierHashKey = tgt.CarrierKey
                        AND link.FlightDate = tgt.FlightDate
                    )
GROUP BY
     link.FlightDate, sat.[Year], sat.[Quarter], sat.[Month], sat.[DayOfMonth],
     sat.[DayOfWeek], link.CarrierHashKey, hubOrigin.AirportCode, SatOrigin.LoadDate,
     hubDest.AirportCode, SatDest.LoadDate
```

This approach only works if the data is grouped over the date, among other dimensions (sourced from the hubs). If the data is aggregated on other dimensions or if the data is loaded multiple times over the day, this statement would fail to update the existing information in the target. However, there are advanced concepts that allow such incremental loads of aggregated fact tables, but this is out of the scope of this book.

14.3 LEVERAGING PIT AND BRIDGE TABLES FOR VIRTUALIZATION

The previous sections have shown some examples for providing materialized dimension and fact tables. Some of them are based on naïve approaches that require truncating the target first.

The major drawback of such materialization is that the data needs to be moved in order to support the materialization. However, it provides optimal performance when directly querying the relational information mart. Consider the following virtual view that provides an aggregated fact "table" which is based on the incremental loading of aggregated fact data in section 14.2.4:

```
CREATE VIEW dbo.FactConnection2 AS
SELECT
      link.CarrierHashKey AS CarrierKey
      ,UPPER(CONVERT(char(32), HASHBYTES('MD5',
             CONCAT(UPPER(RTRIM(LTRIM(hubOrigin.AirportCode))), ';',
             CONVERT(NVARCHAR(30), SatOrigin.LoadDate, 126))),2)) AS OriginKey
      ,UPPER(CONVERT(char(32), HASHBYTES('MD5',
```

```
            CONCAT(UPPER(RTRIM(LTRIM(hubDest.AirportCode))), ';', CONVERT(NVARCHAR(30),
            SatDest.LoadDate, 126))),2)) AS DestKey
    ,DATEPART(YEAR, link.FlightDate)*10000
            +DATEPART(MONTH, link.FlightDate)*100
            +DATEPART(DAY, link.FlightDate) AS FlightDateKey
    ,link.[FlightDate]
    ,sat.[Year]
    ,sat.[Quarter]
    ,sat.[Month]
    ,sat.[DayOfMonth]
    ,sat.[DayOfWeek]
    ,SUM(sat.[DepDelay]) AS SumDepDelay
    ,SUM(sat.[DepDelayMinutes]) AS SumDepDelayMinutes
    ,SUM(sat.[TaxiOut]) AS SumTaxiOut
    ,SUM(sat.[WheelsOff]) AS SumWheelsOff
    ,SUM(sat.[WheelsOn]) AS SumWheelsOn
    ,SUM(sat.[TaxiIn]) AS SumTaxiIn
    ,SUM(sat.[ArrDelay]) AS SumArrDelay
    ,SUM(sat.[ArrDelayMinutes]) AS SumArrDelayMinutes
    ,SUM(CASE WHEN sat.Cancelled=1 THEN 1 ELSE 0 END) AS SumCancelled
    ,SUM(CASE WHEN sat.Diverted=1 THEN 1 ELSE 0 END) AS SumDiverted
    ,SUM(sat.[AirTime]) AS SumAirTime
    ,SUM(sat.[Flights]) AS SumFlights
    ,SUM(sat.[Distance]) AS SumDistance
    ,SUM(sat.[CarrierDelay]) AS SumCArrierDelay
    ,SUM(sat.[WeatherDelay]) AS SumWeatherDelay
    ,SUM(sat.[NASDelay]) AS SumNASDelay
    ,SUM(sat.[SecurityDelay]) AS SumSecurityDelay
    ,SUM(sat.[LateAircraftDelay]) AS SumLateAircraftDelay
FROM [DataVault].[raw].[TLinkFlight] link
INNER JOIN [DataVault].[raw].[HubAirportCode] HubOrigin ON (
            HubOrigin.AirportCodeHashKey = link.OriginHashKey
)
INNER JOIN [DataVault].[raw].[SatOriginAirportMod2] SatOrigin ON (
            SatOrigin.AirportHashKey = link.OriginHashKey
            AND link.LoadDate BETWEEN SatOrigin.LoadDate AND
                  COALESCE(SatOrigin.LoadEndDate, '9999-12-31 23:59:59.999')
)
INNER JOIN [DataVault].[raw].[HubAirportCode] HubDest ON (
            HubDest.AirportCodeHashKey = link.DestHashKey
)
INNER JOIN [DataVault].[raw].[SatDestAirportMod2] SatDest ON (
            SatDest.AirportHashKey = link.DestHashKey
            AND link.LoadDate BETWEEN SatDest.LoadDate AND
                  COALESCE(SatDest.LoadEndDate, '9999-12-31 23:59:59.999')
)
INNER JOIN DataVault.[raw].TSatFlight sat ON (
     sat.FlightHashKey = link.FlightHashKey
)
GROUP BY
     link.FlightDate, sat.[Year], sat.[Quarter], sat.[Month], sat.[DayOfMonth],
     sat.[DayOfWeek], link.CarrierHashKey, hubOrigin.AirportCode, SatOrigin.LoadDate,
     hubDest.AirportCode, SatDest.LoadDate
```

14.3.1 FACTORS THAT AFFECT PERFORMANCE OF VIRTUALIZED FACTS

Because the produced fact "table" is based on a virtual view, the aggregation has to occur whenever data is sourced from the fact table. Depending on the number of records in the source table, these calculations can be resource intensive and hinder the deployment of virtualized fact tables. Also joining the data might become a problem if complex or a larger number of joins is required. Major factors that affect the performance are

- **Joins**, which might require complex join conditions
- **Aggregations and grain changes**, which require resource intensive computations to aggregate or recompute the data.

Once the source tables have been joined, selecting the measures that should be included in the virtualized fact table is not very resource intensive anymore and allows the customization of fact data for different information marts.

A similar problem exists if dimension tables should be provided in a virtual manner: instead of materializing the data in the destination, only a virtual view is provided that sources the data directly from the hub and the involved satellites:

```
CREATE VIEW DimCarrier2 AS
SELECT
       hub.CarrierHashKey AS CarrierKey
       , CASE WHEN hub.Carrier IS NOT NULL AND hub.Carrier <> ''
           THEN hub.Carrier
           ELSE '?'
         END AS Carrier
       , CASE WHEN sat.Code IS NOT NULL AND sat.Code <> ''
           THEN sat.Code
           ELSE '?'
         END AS Code
       , CASE WHEN sat.Name IS NOT NULL AND sat.Name <> ''
           THEN sat.Name
           ELSE 'Unknown'
         END AS Name
       , COALESCE(sat.[Corporate Name], '') AS [Corporate Name]
       , COALESCE(sat.Abbreviation, '') AS Abbreviation
       , COALESCE(sat.[Unique Abbreviation], '') AS [Unique Abbreviation]
       , COALESCE(sat.[Group_Code], '') AS [Group Code]
       , COALESCE(sat.[Region_Code], '') AS [Region Code]
       , COALESCE(sat.[Satisfaction Rank], 9999) AS [Satisfaction Rank]
       , COALESCE(sat.[Sort Order], 9999) AS [Sort Order]
       , COALESCE(sat.[External Reference], '') AS [External Reference]
       , COALESCE(sat.Comments, '') AS Comments
FROM
       DataVault.[raw].HubCarrier hub
LEFT JOIN
       DataVault.[raw].SatCarrier sat ON (
              sat.CarrierHashKey = hub.CarrierHashKey
       )
WHERE
       sat.LoadEndDate IS NULL
```

```
UNION ALL
SELECT
       REPLICATE('0', 32)
       , '?'
       , '?'
       , 'Unknown'
       , 'Unknown'
       , '?'
       , '?'
       , '?'
       , '?'
       , 9999
       , 0
       , ''
       , ''
       ;
```

The above view for a Type 1 dimension is based on a SELECT statement that sources its grain from the hub and joins descriptive data from only one satellite (an easy case). The unknown record is also provided in a virtual manner by adding it using a UNION ALL clause.

There are multiple problems with this approach: first, the source query requires joins, which have to be resolved by the SQL optimizer. The joins also require complex conditions because the dependent satellites don't provide data for each snapshot date but only when changes have been detected in the source system. In addition, business logic is required to implement the soft business rule, for example to select between potentially contradicting raw data from multiple source systems or to recalculate raw data into the desired format. If multiple targets should be supported, we see that the joins are always the same and, in many cases, the most resource-intensive operations. Business logic depends on the target that should be produced and the specific requirements of the business users. In many cases, the soft business rule is not as resource intensive compared to the join operations.

14.3.2 ADVANTAGES OF VIRTUALIZATION

Despite the problems with providing virtualized dimension and fact tables, there are several advantages that drive the need for virtualization in the data warehouse, especially when providing multiple information marts [3]:

- **Simplified solution**: providing facts and dimensions using virtualized approaches is more agile and responsive to user requests than ETL-based integration. These approaches require no data moving or data materialization and are far easier to design and develop.
- **Agile development process**: because it is easy to create and modify virtual facts and dimensions, it is possible to use such an approach in an agile development process that provides the final solution in multiple iterations. This requires that the cost to modify an existing solution is low.
- **Ease of change**: because the cost to modify an existing virtualized fact or dimension table is low, it is also possible to react on changes from the business that occur later in the lifetime of the data warehouse, for example if new products are introduced or the market changes and requires new business logic.

- **Improved developer productivity**: instead of relying on complex ETL-based solutions that are hard to develop, test, and deploy and are prone to changes once deployed, developers can quickly build solutions and demonstrate them to the end-user. If they have produced the wrong solution, they will fail fast, and have the time to start from scratch without generating too much cost.
- **Lower total cost of ownership (TCO)**: in addition to lower development costs (due to higher developer productivity), organizations save storage costs because data doesn't need to be materialized, which consumes disk space. While materializing data, for example using index views or using ETL, improves query performance, it often introduces data inconsistencies that increase the overall costs of the data warehouse.

To take advantage of these characteristics of virtualized information delivery, the Data Vault 2.0 model uses PIT and bridge tables to maintain the desired performance requirements by the business users, despite the fact that joins are required to collect the data, grain shifts occur and aggregations and other resource-intensive computations are performed when delivering the information.

The next sections demonstrate how to leverage these standard entities to provide virtualized fact and dimension tables, customizable for multiple information marts with different requirements, while maintaining superior query performance.

14.3.3 LOADING PIT TABLES

The last section has introduced two activities with delivering dimension tables:

1. **Joining the data from multiple satellites**: the data that is required for information delivery is stored in multiple satellites, often dependent on the same hub.
2. **Implementing the business logic**: a specific dimension for an information mart is based on individual selection of descriptive data from specific satellites and the application of business logic to this data. The actual definition of the business logic depends on the actual target and is user driven (expressed by user requirements).

The first activity is the most resource-intensive operation in the majority of cases. However, it is also possible to separate both activities in an easy way. This separation can be used to reuse the joins by materializing the intermediate result and providing it in a way that optimizes the customization for different information mart targets. This is where the PIT table comes into play.

As outlined in Chapter 6, Advanced Data Vault Modeling, the PIT table is like an index used by the query and provides information about the active satellite entries per snapshot date. The goal is to materialize as much of the join logic as possible and end up with an equi-join only. This join type is the most performant version of joining on most (if not all) relational database servers, including Microsoft SQL Server.

In order to maximize the performance of the PIT table while maintaining low storage requirements, one and only one ghost record is required in each satellite used by the PIT table. This ghost record is used when no record is active in the referenced satellite and serves as the unknown or NULL case. By using the ghost record, it is possible to avoid NULL checks in general, because the join condition will always point to an active record in the satellite table: either an actual record which is active at the given snapshot date or the ghost record.

The following statement is used to create the ghost record in the satellite:

```
INSERT INTO [raw].[SatDestAirportMod]
           ([AirportHashKey]
           ,[LoadDate]
           ,[LoadEndDate]
           ,[RecordSource]
           ,[HashDiff]
           ,[DestCityName]
           ,[DestState]
           ,[DestStateName]
           ,[DestCityMarketID]
           ,[DestStateFips]
           ,[DestWac])
     VALUES
           (REPLICATE('0', 32)
           ,'0001-01-01 00:00:00.000'
           ,'9999-12-31 23:59:59.999'
           ,'SYSTEM'
           ,REPLICATE('0', 32)
           ,'Unknown City'
           ,'?'
           ,'Unknown State'
           ,0
           ,0
           ,0)
```

This statement is usually executed after the satellite table has been created using the CREATE TABLE DDL statement to make sure that each satellite has a ghost record. Because the record is calculated and not sourced from a source system, the record source has been set to "SYSTEM" and the record is not part of the auditable raw data from the source system. Therefore, the ghost record can be modified at any time, for example when new attributes are added to the satellite table. Note that the hash key and the hash diff values are set to 32 zero values and the load date set to the first day and timestamp of all times, while the load end date is set to the last date and timestamp of all times. All other descriptive columns are set to default values that should be shown if no better data is available.

The PIT table is usually loaded in an incremental approach, based on the snapshot date. Each ETL load inserts the data that is missing for the given snapshot date. The following DDL statement creates the PIT table in the Business Vault:

```
CREATE TABLE [biz].[PITAirportCode](
       [AirportKey] [char](32) NOT NULL,
       [AirportCodeHashKey] [char](32) NOT NULL,
       [SnapshotDate] [datetime2](7) NOT NULL,
       [OriginAirportHashKey] [char](32) NOT NULL,
       [OriginLoadDate] [datetime2](7) NOT NULL,
       [DestAirportHashKey] [char](32) NOT NULL,
       [DestLoadDate] [datetime2](7) NOT NULL,
```

```
 CONSTRAINT [PK_PITAirportCode] PRIMARY KEY NONCLUSTERED
(
      [AirportKey] ASC
) ON [INDEX],
 CONSTRAINT [UK_PITAirportCode] UNIQUE NONCLUSTERED
(
      [AirportCodeHashKey] ASC,
      [SnapshotDate] ASC
) ON [INDEX]
) ON [DATA]
```

The primary key **AirportKey** is of a hash value derived from the business key of the hub **HubAirportCode** and the **snapshot date**, which is a rolling date, controlled by the ETL process. The alternate key of the PIT table is the **AirportCodeHashKey** from the parent hub and the **snapshot date**. All other columns are pointing to one satellite entry, identified by a hash key and a load date. There are two additional columns per referenced satellite. In this case, two satellites have been referenced; therefore, four additional columns are required. If satellites are added to the hub, the PIT has to be modified as part of the process.

There are multiple options to load the PIT table. One is to use an **Execute SQL Task** in the control flow that is executed whenever the Business Vault is populated. This is usually done just after the loading procedures for all satellites in the Raw Data Vault and in the Business Vault are completed. The Business Vault satellites are required, because they can be added to the PIT as well, which makes sense if they are load-date oriented and not based on a snapshot date.

Note that implementing this approach using only SQL is another option and uses similar statements to those used in the following description.

To implement a task to load a PIT table in SSIS, add an Execute SQL Task to the control flow of your SSIS package. Open the task editor (Figure 14.12).

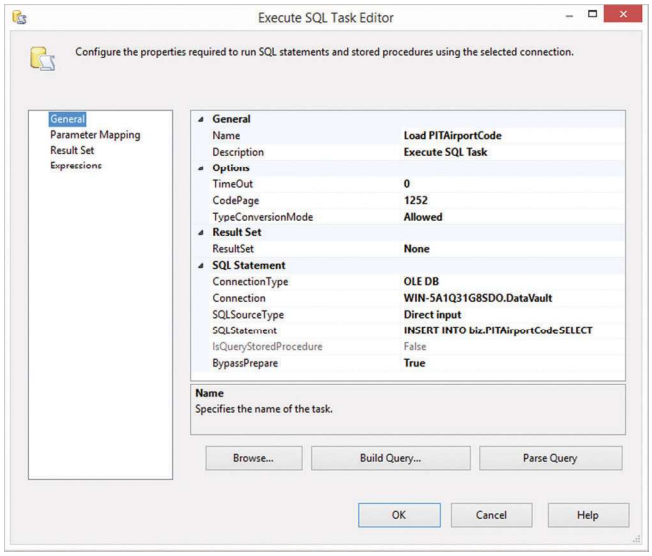

FIGURE 14.12

Execute SQL task editor for PIT table.

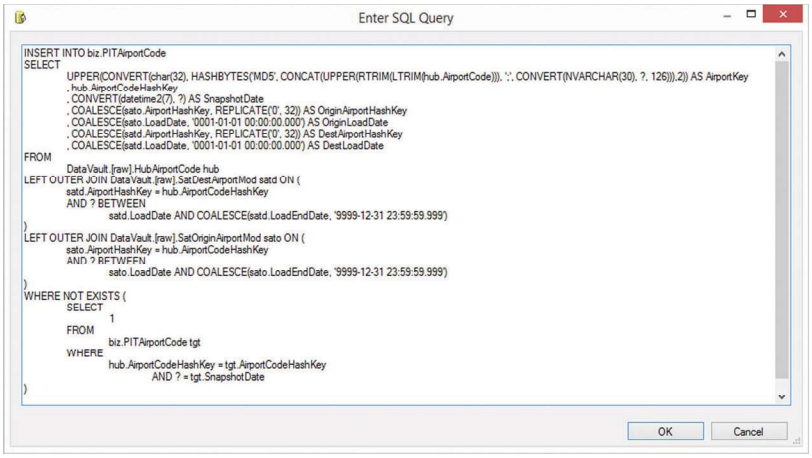

FIGURE 14.13

Enter SQL query dialog for PIT table.

Because the Business Vault is part of the EDW layer, set the connection of the **Execute SQL Task** to the **DataVault** connection manager used in other chapters. Open the SQL query editor by clicking the ellipse button in the properties value column. The dialog in Figure 14.13 is shown.

The following statement is used for loading one snapshot to the PIT. Set the query to the following SQL text:

```
INSERT INTO biz.PITAirportCode
SELECT
       UPPER(CONVERT(char(32),
              HASHBYTES('MD5', CONCAT(
                     UPPER(RTRIM(LTRIM(hub.AirportCode)))
                     , ';'
                     , CONVERT(NVARCHAR(30), ?, 126)
              ))
       ,2)) AS AirportKey
       , hub.AirportCodeHashKey
       , CONVERT(datetime2(7), ?) AS SnapshotDate
       , COALESCE(sato.AirportHashKey, REPLICATE('0', 32)) AS OriginAirportHashKey
       , COALESCE(sato.LoadDate, '0001-01-01 00:00:00.000') AS OriginLoadDate
       , COALESCE(satd.AirportHashKey, REPLICATE('0', 32)) AS DestAirportHashKey
       , COALESCE(satd.LoadDate, '0001-01-01 00:00:00.000') AS DestLoadDate
FROM
       DataVault.[raw].HubAirportCode hub
LEFT OUTER JOIN DataVault.[raw].SatDestAirportMod satd ON (
       satd.AirportHashKey = hub.AirportCodeHashKey
       AND ? BETWEEN
              satd.LoadDate AND COALESCE(satd.LoadEndDate, '9999-12-31 23:59:59.999')
)
LEFT OUTER JOIN DataVault.[raw].SatOriginAirportMod sato ON (
       sato.AirportHashKey = hub.AirportCodeHashKey
       AND ? BETWEEN
```

```
            sato.LoadDate AND COALESCE(sato.LoadEndDate, '9999-12-31 23:59:59.999')
)
WHERE NOT EXISTS (
      SELECT
            1
      FROM
            biz.PITAirportCode tgt
      WHERE
            hub.AirportCodeHashKey = tgt.AirportCodeHashKey
                  AND ? = tgt.SnapshotDate
)
```

Similar to other statements in this chapter, the BETWEEN statement assumes that the **load date** and the **load end date** are not overlapping (the **load end date** of the previous record is the **load date** minus one nanosecond of the next record). The goal is to load the data from this statement into a target PIT table. This key value in the primary key column (here: **AirportKey**) is later used in the dimensional tables of the dependent information marts.

The source statement generates a record per hash key in the parent hub and the snapshot date (which is set by a variable in the statement and set by SSIS). Due to the WHERE condition, the approach is recoverable and could be used to ensure that new business keys are also populated into the target for past snapshot dates to ensure equi-joins at all points in time. This is optional and requires running past snapshot dates again in SSIS.

The snapshot date is stored in a variable **dSnapshotDate** in the SSIS control flow. In order to bind the variable to the parameters in this SQL statement, close the SQL query editor and switch to the **parameter mapping** page of the execute SQL task editor. The page in Figure 14.14 is shown.

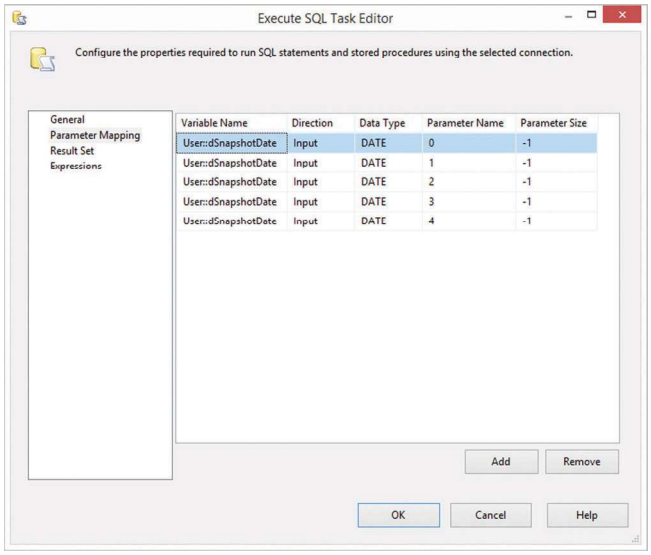

FIGURE 14.14

Parameter mapping in execute SQL task editor for the PIT table.

Because the snapshot date is used multiple times in the SQL statement, and the parameters are not named in OLE DB statements, the variable has to be bound multiple times. For each question mark, add a parameter mapping to the list and select the **dSnapshotDate** variable in the User namespace. Select the **DATE** data type and set each parameter to one instance of the parameter reference (from 0 to 4). Leave the parameter size as is.

Set the variable **dSnapshotDate** to a desired value (usually a date with a time of 00:00:00.000) and run the task. It is also possible to execute the task in a **for loop container** in order to insert multiple snapshot dates, for example during an initial load.

In some cases, the dimension tables should not include one entry per snapshot date but only when changes in the descriptive data occur. This is the actual case for Type 2 dimensions of SSAS OLAP cubes. In this case, it is possible to modify the WHERE condition of the above statement and load only changes into the PIT table. The load date of the referenced satellites is used for the change detection. This is a sufficient approach because the load date changes per change that is added to the satellite.

14.3.4 CREATING VIRTUALIZED DIMENSIONS

In order to implement the Type 2 dimension introduced in section 14.2.2 by using the PIT table from the previous section, the following T-SQL view is created:

```
CREATE VIEW [dbo].[DimAirport3] AS
SELECT
       pit.AirportKey
       ,pit.AirportCodeHashKey
       ,pit.SnapshotDate
       ,CASE
         WHEN hub.AirportCode IS NOT NULL AND hub.AirportCode <> '' THEN hub.AirportCode
         ELSE '?'
        END AS AirportCode
       ,COALESCE(satd.[DestCityName], sato.[OriginCityName], 'Unknown') AS CityName
       ,COALESCE(satd.[DestState], sato.[OriginState], '?') AS [State]
       ,COALESCE(satd.[DestStateName], sato.[OriginStateName], 'Unknown') AS StateName
       ,COALESCE(satd.[DestCityMarketID], sato.[OriginCityMarketID], 0) AS CityMarketID
       ,COALESCE(satd.[DestStateFips], sato.[OriginStateFips], 0) AS StateFips
       ,COALESCE(satd.[DestWac], sato.[OriginWac], 0) AS Wac
FROM
       DataVault.[biz].[PITAirportCode] pit
INNER JOIN DataVault.[raw].HubAirportCode hub ON (
       pit.AirportCodeHashKey = hub.AirportCodeHashKey
)
INNER JOIN DataVault.[raw].SatOriginAirportMod sato ON (
       sato.AirportHashKey = pit.OriginAirportHashKey
       AND sato.LoadDate = pit.OriginLoadDate
)
INNER JOIN DataVault.[raw].SatDestAirportMod satd ON (
       satd.AirportHashKey = pit.DestAirportHashKey
       AND satd.LoadDate = pit.DestLoadDate
)
```

This view implements the same business logic as in the original example: it provides a list of airports with descriptive information from the destination airport source or, if this primary source doesn't provide any data at that snapshot date, it alternatively loads descriptive data from the origin airport satellite.

The biggest difference from the original example is that the joins have changed from LEFT OUTER JOINs to INNER JOINs. In addition, all joins are equi-joins to further increase the performance of the join operations. Another advantage is that the query is based on the PIT table, instead of collecting load dates from multiple satellites, which provides further performance improvements. It joins the airport code hub, in order to retrieve the business key, which is stored in the hub only. It also joins both satellites that provide the descriptive data.

Note that the business logic is problematic, because the checks are based on individual fields. A better approach would check whether the leading satellite provides any useful data by performing a check on the referenced hash key in the PIT table:

```
CREATE VIEW [dbo].[DimAirport4] AS
SELECT
       pit.AirportKey
      ,pit.AirportCodeHashKey
      ,pit.SnapshotDate
      ,CASE
        WHEN hub.AirportCode IS NOT NULL AND hub.AirportCode <> ''
        THEN hub.AirportCode
        ELSE '?'
       END AS AirportCode
      ,CASE
        WHEN pit.DestAirportHashKey = '00000000000000000000000000000000'
        THEN sato.[OriginCityName]
        ELSE satd.[DestCityName]
       END AS CityName
      ,CASE
        WHEN pit.DestAirportHashKey = '00000000000000000000000000000000'
        THEN sato.[OriginState]
        ELSE satd.[DestState]
       END AS [State]
      ,CASE
        WHEN pit.DestAirportHashKey = '00000000000000000000000000000000'
        THEN sato.[OriginStateName]
        ELSE satd.[DestStateName]
       END AS StateName
      ,CASE
        WHEN pit.DestAirportHashKey = '00000000000000000000000000000000'
        THEN sato.[OriginCityMarketID]
        ELSE satd.[DestCityMarketID]
       END AS CityMarketID
      ,CASE
        WHEN pit.DestAirportHashKey = '00000000000000000000000000000000'
        THEN sato.[OriginStateFips]
        ELSE satd.[DestStateFips]
       END AS StateFips
```

```
       ,CASE
         WHEN pit.DestAirportHashKey = '00000000000000000000000000000000'
         THEN sato.[OriginWac]
         ELSE satd.[DestWac]
        END AS Wac
FROM
       DataVault.[biz].[PITAirportCode] pit
INNER JOIN DataVault.[raw].HubAirportCode hub ON (
       pit.AirportCodeHashKey = hub.AirportCodeHashKey
)
INNER JOIN DataVault.[raw].SatOriginAirportMod sato ON (
       sato.AirportHashKey = pit.OriginAirportHashKey
       AND sato.LoadDate = pit.OriginLoadDate
)
INNER JOIN DataVault.[raw].SatDestAirportMod satd ON (
       satd.AirportHashKey = pit.DestAirportHashKey
       AND satd.LoadDate = pit.DestLoadDate
)
```

If the referenced hash key is the ghost record, the data is retrieved from the other satellite. Implementing this change is relatively cheap because all the data is provided by virtual means while meeting the performance requirements of the business users.

The very same PIT table can be used to produce other virtual dimensions that implement different business logic. For example, the following view creates a virtual dimension that provides descriptive information from the destination airport source only:

```
CREATE VIEW [dbo].[DimAirportCalif] AS
SELECT
       pit.AirportKey
       ,pit.AirportCodeHashKey
       ,pit.SnapshotDate
       ,CASE
         WHEN hub.AirportCode IS NOT NULL AND hub.AirportCode <> '' THEN hub.AirportCode
         ELSE '?'
        END AS AirportCode
       ,satd.[DestCityName]
       ,satd.[DestState]
       ,satd.[DestStateName]
       ,satd.[DestCityMarketID]
       ,satd.[DestStateFips]
       ,satd.[DestWac]
FROM
       DataVault.[biz].[PITAirportCode] pit
INNER JOIN DataVault.[raw].HubAirportCode hub ON (
       pit.AirportCodeHashKey = hub.AirportCodeHashKey
)
INNER JOIN DataVault.[raw].SatDestAirportMod satd ON (
       satd.AirportHashKey = pit.DestAirportHashKey
       AND satd.LoadDate = pit.DestLoadDate
)
WHERE
       satd.DestState = 'CA'
```

The query statement only joins the satellites that are actually required for the purpose of the dimension. In this example, there is an additional filter on airports from California, which is part of the business rule implemented in this view.

The examples presented in this section should have given you an impression of the advantages that are achieved by separating the join operation from the rest of the business logic. Similar concepts are applied to fact tables in the following sections.

14.3.5 LOADING BRIDGE TABLES

The purpose of a bridge table is to ensure that the performance requirements of business users are met for virtual fact tables. The performance of fact tables, when sourced from a Data Vault 2.0 model, depends on three important factors:

1. **Joins between links**: because the desired grain is often not found in only one Data Vault 2.0 link, multiple links are joined in order to achieve the right grain. A prejoining improves the performance of this factor.
2. **Required aggregations and otherwise computed values**: other issues that limit the performance of virtual fact tables are required aggregations that have to be performed on the raw data, often involving a grain shift in addition. In other cases, measures are computed from raw data and added to the fact table. The latter might require complex business logic for the calculation.
3. **Applying additional customization**: a specific fact table includes a number of dimensions and measures that have to be derived from the raw data. The actual number and characteristic is defined by the individual business specification.

Joining and aggregating the data and running complex computational logic are the most resource-intensive operations of these activities. On the other hand, the results from these activities are often reusable for multiple targets (fact tables). Therefore, an approach that materializes the first two activities and separates it from the customization provides an advantage in the information delivery of fact tables, just as the PIT table does for dimension tables. Bridge tables are used for exactly this purpose: to provide the materialized basis for virtual fact tables that meet the performance requirements of the business users.

The following DDL statement creates a simple and minimal bridge table in the Business Vault schema of the EDW:

```
CREATE TABLE [biz].[BrDiversionFlight](
       [SnapshotDate] [datetime2](7) NOT NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [FlightHashKey] [char](32) 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,
       [DivTailNumHashKey] [char](32) NOT NULL,
       [DivAirportHashKey] [char](32) NOT NULL,
       [Diversion] [int] NOT NULL,
```

```
CONSTRAINT [PK_BrDiversionFlight] PRIMARY KEY NONCLUSTERED
(
       [FlightNumHashKey]
       ,[FlightDate]
       ,[OriginHashKey]
       ,[Diversion]
) ON [INDEX],
INDEX IX_BrDiversionFlight_FlightHashKey ( FlightHashKey ASC ) ON [INDEX],
INDEX IX_BrDiversionFlight_DestHashKey ( DestHashKey ASC, SnapshotDate ASC ) ON [INDEX]
) ON [DATA]
```

The bridge table performs only a prejoining of the nonhistorized links **TLinkFlight** and **TDiversionFlight**, which is required to produce a later fact table that is used to analyze the flight diversions and should include information from the planned flight.

The grain defines the primary key of the bridge table. It is also used by the next SQL statement that incrementally loads the bridge table.

Note that the bridge table implements some indices on the hash keys in order to improve the performance of joins between the bridge table and hubs or satellites. This is necessary for those columns, which are heavily used by ad-hoc queries, because foreign keys are not implemented in the EDW and therefore the implied indices are missing from the table.

In order to load the bridge, the following incremental INSERT statement is used within SSIS:

```
INSERT INTO DataVault.biz.BrDiversionFlight
SELECT
       DATEADD(DAY, 1, CONVERT(date, DATEADD(MCS, -1, div.LoadDate))) AS SnapshotDate
       , div.RecordSource
       , flight.FlightHashKey
       , flight.CarrierHashKey
       , flight.FlightNumHashKey
       , flight.TailNumHashKey
       , flight.OriginHashKey
       , flight.DestHashKey
       , flight.FlightDate
       , div.DivTailNumHashKey
       , div.DivAirportHashKey
       , div.Diversion
FROM
       [DataVault].[raw].[TLinkFlight] flight,
       [DataVault].[raw].[TLinkDiversionFlight] div
WHERE
       flight.FlightNumHashKey = div.FlightNumHashKey
       AND flight.FlightDate = div.FlightDate
       AND flight.OriginHashKey = div.StartAirportHashKey
       AND NOT EXISTS (
              SELECT
                     1
              FROM
                     DataVault.biz.BrDiversionFlight tgt
              WHERE
                     flight.FlightNumHashKey = tgt.FlightNumHashKey
                            AND div.FlightDate = tgt.FlightDate
                            AND flight.OriginHashKey = tgt.OriginHashKey
                            AND div.Diversion = tgt.Diversion
       )
```

This example implements a bridge table based on nonhistorized links. However, the loading procedures for standard Data Vault 2.0 links are the same. This bridge joins both links together and avoids a Cartesian product by using appropriate conditions in the WHERE clause. The bridge actually "bridges" two links and multiple hubs. However, the hubs are not joined at this time, because the business keys are not prejoined into the bridge by default. Only the hash key is added to the bridge table, but this key is already available in the link structures.

The sub-select statement in the WHERE clause is required to support incremental loading and is based on the primary key of the bridge table.

The load date from the source is calculated to the next snapshot date. This is easy if the snapshot date follows a regular pattern because it can be calculated in this case. If the snapshot date follows a different pattern, a lookup into a reference table for the snapshots of the target might be required. Having calculated the snapshot date makes it easier when retrieving information from or via PIT tables in the virtual fact table, for example to retrieve additional descriptive information from dependent satellites of the PIT table. If there is a need for the (technical) load date in the target, it is possible to add it to the bridge in addition. But in most cases, other dates provide more business value, for example the flight date in our example.

Note that the calculation might differ in your case. The one provided in the previous statement requires daily snapshot dates and ensures that load dates without time are mapped to the same snapshot date.

However, the goal of the bridge table is to provide the basis for virtual fact tables with superior performance. If the overall query performance is improved by prejoining information from hubs and their dependent satellites, it should be done, for example by joining the business keys from hubs into the bridge table. However, keep in mind that the performance of the bridge table also drops if the table becomes too wide. In the end, it requires some experimentation to find the right mixes for every individual bridge table.

More performance gains are achieved by ensuring that the bridge table is in the same grain as the target fact table. For example, the following bridge table changes the grain of the previous bridge table:

```
CREATE TABLE [biz].[BrConnection](
       [SnapshotDate] [datetime2](7) NOT NULL,
       [RecordSource] [nvarchar](50) NOT NULL,
       [CarrierHashKey] [char](32) NOT NULL,
       [OriginHashKey] [char](32) NOT NULL,
       [DestHashKey] [char](32) NOT NULL,
       [FlightDate] [datetime2](7) NOT NULL,
       [Year] [smallint] NOT NULL,
       [Quarter] [smallint] NOT NULL,
       [Month] [smallint] NOT NULL,
       [DayOfMonth] [smallint] NOT NULL,
       [DayOfWeek] [smallint] NOT NULL,
       [SumDepDelay] [int] NOT NULL,
       [SumDepDelayMinutes] [int] NOT NULL,
       [SumTaxiOut] [int] NOT NULL,
       [SumWheelsOff] [int] NOT NULL,
       [SumWheelsOn] [int] NOT NULL,
       [SumTaxiIn] [int] NOT NULL,
       [SumArrDelay] [int] NOT NULL,
       [SumArrDelayMinutes] [int] NOT NULL,
       [SumCancelled] [int] NOT NULL,
```

```
        [SumDiverted] [int] NOT NULL,
        [SumAirTime] [int] NOT NULL,
        [SumFlights] [int] NOT NULL,
        [SumDistance] [int] NOT NULL,
        [SumCArrierDelay] [int] NOT NULL,
        [SumWeatherDelay] [int] NOT NULL,
        [SumNASDelay] [int] NOT NULL,
        [SumSecurityDelay] [int] NOT NULL,
        [SumLateAircraftDelay] [int] NOT NULL,
CONSTRAINT [PK_BrConnection] PRIMARY KEY NONCLUSTERED
(
        [CarrierHashKey]
        ,[OriginHashKey]
        ,[DestHashKey]
        ,[FlightDate]

) ON [INDEX],
) ON [DATA]
```

The grain was changed by removing a hub reference from the bridge table. This is also called a "grain shift" and there are two options for it:

- **Removing hub references**: reduces the granularity, for example, by using a GROUP BY clause in the INSERT statement.
- **Adding hub references**: increases the granularity, for example by joining additional links to the current set of used links.

The table is loaded with the following incremental statement that is typically used from an Execute SQL task in SSIS again:

```
INSERT INTO biz.BrConnection
SELECT
      link.FlightDate AS SnapshotDate
      ,'SR9483'
      ,link.CarrierHashKey
      ,link.OriginHashKey
      ,link.DestHashKey
      ,link.[FlightDate]
      ,sat.[Year]
      ,sat.[Quarter]
      ,sat.[Month]
      ,sat.[DayOfMonth]
      ,sat.[DayOfWeek]
      ,SUM(sat.[DepDelay]) AS SumDepDelay
      ,SUM(sat.[DepDelayMinutes]) AS SumDepDelayMinutes
      ,SUM(sat.[TaxiOut]) AS SumTaxiOut
      ,SUM(sat.[WheelsOff]) AS SumWheelsOff
      ,SUM(sat.[WheelsOn]) AS SumWheelsOn
      ,SUM(sat.[TaxiIn]) AS SumTaxiIn
      ,SUM(sat.[ArrDelay]) AS SumArrDelay
      ,SUM(sat.[ArrDelayMinutes]) AS SumArrDelayMinutes
      ,SUM(CASE WHEN sat.Cancelled=1 THEN 1 ELSE 0 END) AS SumCancelled
      ,SUM(CASE WHEN sat.Diverted=1 THEN 1 ELSE 0 END) AS SumDiverted
```

```
    ,SUM(sat.[AirTime]) AS SumAirTime
    ,SUM(sat.[Flights]) AS SumFlights
    ,SUM(sat.[Distance]) AS SumDistance
    ,SUM(sat.[CarrierDelay]) AS SumCArrierDelay
    ,SUM(sat.[WeatherDelay]) AS SumWeatherDelay
    ,SUM(sat.[NASDelay]) AS SumNASDelay
    ,SUM(sat.[SecurityDelay]) AS SumSecurityDelay
    ,SUM(sat.[LateAircraftDelay]) AS SumLateAircraftDelay
FROM
      [DataVault].[raw].[TLinkFlight] link
INNER JOIN DataVault.[raw].TSatFlight sat ON (
    sat.FlightHashKey = link.FlightHashKey
)
WHERE
          NOT EXISTS (SELECT
                      1
                FROM
                      [DataVault].[biz].[BrConnection] tgt
                WHERE
                      link.CarrierHashKey = tgt.CarrierHashKey
                      AND link.OriginHashKey = tgt.OriginHashKey
                      AND link.DestHashKey = tgt.DestHashKey
                      AND link.FlightDate = tgt.FlightDate
                )

GROUP BY
    link.CarrierHashKey, link.OriginHashKey, link.DestHashKey, link.FlightDate,
    sat.[Year], sat.[Quarter], sat.[Month], sat.[DayOfMonth], sat.[DayOfWeek]
```

The INSERT statement implements the grain shift by removing hub references (to **HubCarrier**, **HubTailNumber** and **HubFlightNumber**) and applying a GROUP BY clause on the data. It further improves the performance of virtualized fact tables by adding the results of required aggregations to the materialized bridge. Because the aggregations represent business logic that is implemented by this bridge, the **record source** is set to a fixed value in the load statement. The **snapshot date** was not calculated from the **load date** because it is not sufficient to do so (due to the aggregation). Instead, the **flight date** was used as a **snapshot date**, which is more appropriate for the target information mart.

These bridge tables can now be used as the basis to provide virtual fact tables.

14.3.6 CREATING VIRTUALIZED FACTS

The virtualized fact tables implement the additional customization that is required by individual fact tables in various information marts and can differ in the representation of the information. However, the grain of fact tables that depend on the same bridge table should be the same. If the grain differs between the virtual fact table and the bridge table, a new bridge should be introduced in the Business Vault to ensure that the performance meets the requirements of the business users.

The following DDL statement creates a virtual fact entity based on **BrDiversionFlight**:

```
CREATE VIEW FactDiversions AS
SELECT
      lflight.FlightDate
      ,DATEPART(YEAR, lflight.FlightDate)*10000
             +DATEPART(MONTH, lflight.FlightDate)*100
             +DATEPART(DAY, lflight.FlightDate) AS FlightDateKey
      ,c.Carrier
      ,CONCAT(fn.Carrier, fn.FlightNum) AS FlightNum
      ,tn.TailNum
      ,pitOrigin.AirportKey AS OriginAirportKey
      ,pitDest.AirportKey AS DestAirportKey
      ,dtn.TailNum AS DivTailNum
      ,pitDivAirport.AirportKey AS DivAirportKey
      ,br.Diversion
      ,sat.[CRSDepTime]
      ,sat.[DepTime]
      ,sat.[DepDelay]
      ,sat.[DepDelayMinutes]
      ,sat.[DepDel15]
      ,sat.[DepartureDelayGroups]
      ,sat.[DepTimeBlk]
      ,sat.[TaxiOut]
      ,sat.[WheelsOff]
      ,sat.[WheelsOn]
      ,sat.[TaxiIn]
      ,sat.[CRSArrTime]
      ,sat.[ArrTime]
      ,sat.[ArrDelay]
      ,sat.[ArrDelayMinutes]
      ,sat.[ArrDel15]
      ,sat.[ArrivalDelayGroups]
      ,sat.[ArrTimeBlk]
      ,sat.[Cancelled]
      ,sat.[CancellationCode]
      ,sat.[Diverted]
      ,sat.[CRSElapsedTime]
      ,sat.[ActualElapsedTime]
      ,sat.[AirTime]
      ,sat.[Flights]
      ,sat.[Distance]
      ,sat.[DistanceGroup]
      ,sat.[CarrierDelay]
      ,sat.[WeatherDelay]
      ,sat.[NASDelay]
      ,sat.[SecurityDelay]
      ,sat.[LateAircraftDelay]
      ,sat.[FirstDepTime]
      ,sat.[TotalAddGTime]
      ,sat.[LongestAddGTime]
```

```
FROM
       [DataVault].[biz].[BrDiversionFlight] br
INNER JOIN [DataVault].[raw].[TLinkFlight] lflight ON (
       lflight.FlightHashKey = br.FlightHashKey
)
INNER JOIN DataVault.[raw].TSatFlight sat ON (
            sat.FlightHashKey = br.FlightHashKey
)
INNER JOIN [DataVault].[raw].[HubCarrier] c ON (
    c.CarrierHashKey = br.CarrierHashKey
)
INNER JOIN [DataVault].[raw].[HubFlightNum] fn ON (
    fn.FlightNumHashKey = br.FlightNumHashKey
)
INNER JOIN [DataVault].[raw].[HubTailNum] tn ON (
    tn.TailNumHashKey = br.TailNumHashKey
)
INNER JOIN [DataVault].[biz].[PITAirportCode] pitOrigin ON (
    pitOrigin.AirportCodeHashKey = br.OriginHashKey
       AND pitOrigin.SnapshotDate = br.SnapshotDate
)
INNER JOIN [DataVault].[biz].[PITAirportCode] pitDest ON (
    pitDest.AirportCodeHashKey = br.DestHashKey
       AND pitDest.SnapshotDate = br.SnapshotDate
)
INNER JOIN [DataVault].[raw].[HubTailNum] dtn ON (
    dtn.TailNumHashKey = br.DivTailNumHashKey
)
INNER JOIN [DataVault].[biz].[PITAirportCode] pitDivAirport ON (
    pitDivAirport.AirportCodeHashKey = br.DivAirportHashKey
       AND pitDivAirport.SnapshotDate = br.SnapshotDate
)
```

At first glance, the statement looks large. But for the SQL optimizer, most activities required to execute this statement are relatively cheap (performance-wise). The statement joins measures from the nonhistorized satellite **TSatFlight**. This join, as all the other joins in this statement, is based on an inner join with an equi-join condition. If a business key is required in the fact table (for example, because not all dimensions are provided via dimension tables in this example), the hub is joined on the hash key only in order to retrieve the business key or composite key from the hub. This statement concatenates the composite key from hub **HubFlightNum** into a format that the user is familiar with. This business logic could be extended to serve other requirements by the end-user, which might differ per information mart or fact table.

Note that the **flight date** could be easily sourced from the bridge table source. This statement joins **TLinkFlight** for demonstrative purposes: because the grain is the same, it is easily possible to join the original source of the bridge table in order to include additional or missing data into the fact table. In order to support such joins, the link hash key, in this example **FlightHashKey**, should be included.

If dimension tables should be provided for some dimensions, it is easy to retrieve the required key values for the dimension entry. This hash value comes from the PIT table. However, because the PIT table provides multiple snapshots for the same hash key, the equi-join condition for the PIT table is

extended to include the snapshot date of the fact. This snapshot date was calculated from the technical load date and helps us to retrieve the appropriate entry in the PIT table. We could also use the PIT table to retrieve a load date for satellites included in the PIT table to retrieve additional descriptive data from dependent satellites:

```
CREATE VIEW FactDiversions2 AS
SELECT
      lflight.FlightDate
      ,DATEPART(YEAR, lflight.FlightDate)*10000
             +DATEPART(MONTH, lflight.FlightDate)*100
             +DATEPART(DAY, lflight.FlightDate) AS FlightDateKey
      ,c.Carrier
      ,CONCAT(fn.Carrier, fn.FlightNum) AS FlightNum
      ,tn.TailNum
      ,oa.AirportCode
      ,soa.OriginCityName
      ,soa.OriginState
      ,soa.OriginStateName
      ,pitDest.AirportKey AS DestAirportKey
      ,dtn.TailNum AS DivTailNum
      ,pitDivAirport.AirportKey AS DivAirportKey
      ,br.Diversion
      ,sat.[CRSDepTime]
      ,sat.[DepTime]
      ,sat.[DepDelay]
      ,sat.[DepDelayMinutes]
      ,sat.[DepDel15]
      ,sat.[DepartureDelayGroups]
      ,sat.[DepTimeBlk]
      ,sat.[TaxiOut]
      ,sat.[WheelsOff]
      ,sat.[WheelsOn]
      ,sat.[TaxiIn]
      ,sat.[CRSArrTime]
      ,sat.[ArrTime]
      ,sat.[ArrDelay]
      ,sat.[ArrDelayMinutes]
      ,sat.[ArrDel15]
      ,sat.[ArrivalDelayGroups]
      ,sat.[ArrTimeBlk]
      ,sat.[Cancelled]
      ,sat.[CancellationCode]
      ,sat.[Diverted]
      ,sat.[CRSElapsedTime]
      ,sat.[ActualElapsedTime]
      ,sat.[AirTime]
      ,sat.[Flights]
      ,sat.[Distance]
      ,sat.[DistanceGroup]
      ,sat.[CarrierDelay]
      ,sat.[WeatherDelay]
      ,sat.[NASDelay]
      ,sat.[SecurityDelay]
```

```
    ,sat.[LateAircraftDelay]
    ,sat.[FirstDepTime]
    ,sat.[TotalAddGTime]
    ,sat.[LongestAddGTime]

FROM
       [DataVault].[biz].[BrDiversionFlight] br
INNER JOIN [DataVault].[raw].[TLinkFlight] lflight ON (
       lflight.FlightHashKey = br.FlightHashKey
)
INNER JOIN DataVault.[raw].TSatFlight sat ON (
     sat.FlightHashKey = br.FlightHashKey
)
INNER JOIN [DataVault].[raw].[HubCarrier] c ON (
    c.CarrierHashKey = br.CarrierHashKey
)
INNER JOIN [DataVault].[raw].[HubFlightNum] fn ON (
    fn.FlightNumHashKey = br.FlightNumHashKey
)
INNER JOIN [DataVault].[raw].[HubTailNum] tn ON (
    tn.TailNumHashKey = br.TailNumHashKey
)
INNER JOIN [DataVault].[biz].[PITAirportCode] pitOrigin ON (
    pitOrigin.AirportCodeHashKey = br.OriginHashKey
       AND pitOrigin.SnapshotDate = br.SnapshotDate
)
INNER JOIN [DataVault].[raw].[HubAirportCode] oa ON (
       oa.AirportCodeHashKey = br.OriginHashKey
)
INNER JOIN [DataVault].[raw].[SatOriginAirportMod] soa ON (
       soa.AirportHashKey = pitOrigin.OriginAirportHashKey
       AND soa.LoadDate = pitOrigin.OriginLoadDate
)
INNER JOIN [DataVault].[biz].[PITAirportCode] pitDest ON (
    pitDest.AirportCodeHashKey = br.DestHashKey
       AND pitDest.SnapshotDate = br.SnapshotDate
)
INNER JOIN [DataVault].[raw].[HubTailNum] dtn ON (
    dtn.TailNumHashKey = br.DivTailNumHashKey
)
INNER JOIN [DataVault].[biz].[PITAirportCode] pitDivAirport ON (
    pitDivAirport.AirportCodeHashKey = br.DivAirportHashKey
       AND pitDivAirport.SnapshotDate = br.SnapshotDate
)
```

In this example, the key value for the origin airport dimension is replaced by attributes directly joined into the fact table. However, the PIT is still used in order to retrieve the appropriate load date in the dependent satellite **SatOriginAirportMod**. Joining the satellite requires only INNER JOINs with equi-join condition, due to the available PIT table. Without the PIT, a more complex join would be required to find the appropriate delta record. It would involve a BETWEEN condition and a LEFT JOIN because it is not guaranteed that the satellite provides a record for the given snapshot date.

Another example implements a virtual fact table on the aggregated bridge **BrConnection**:

```
CREATE VIEW FactConnection3 AS
SELECT
      c.Carrier
      ,pitOrigin.AirportKey AS OriginAirportKey
      ,pitDest.AirportKey AS DestAirportKey
      ,DATEPART(YEAR, br.FlightDate)*10000
             +DATEPART(MONTH, br.FlightDate)*100
             +DATEPART(DAY, br.FlightDate) AS FlightDateKey
      ,br.[FlightDate]
      ,br.[Year]
      ,br.[Quarter]
      ,br.[Month]
      ,br.[DayOfMonth]
      ,br.[DayOfWeek]
      ,br.[SumDepDelay]
      ,br.[SumDepDelayMinutes]
      ,br.[SumTaxiOut]
      ,br.[SumWheelsOff]
      ,br.[SumWheelsOn]
      ,br.[SumTaxiIn]
      ,br.[SumArrDelay]
      ,br.[SumArrDelayMinutes]
      ,br.[SumCancelled]
      ,br.[SumDiverted]
      ,br.[SumAirTime]
      ,br.[SumFlights]
      ,br.[SumDistance]
      ,br.[SumCArrierDelay]
      ,br.[SumWeatherDelay]
      ,br.[SumNASDelay]
      ,br.[SumSecurityDelay]
      ,br.[SumLateAircraftDelay]
  FROM
        [DataVault].[biz].[BrConnection] br
  INNER JOIN [DataVault].[raw].[HubCarrier] c ON (
      c.CarrierHashKey = br.CarrierHashKey
  )
  INNER JOIN [DataVault].[biz].[PITAirportCode] pitOrigin ON (
      pitOrigin.AirportCodeHashKey = br.OriginHashKey
         AND pitOrigin.SnapshotDate = br.SnapshotDate
  )
  INNER JOIN [DataVault].[biz].[PITAirportCode] pitDest ON (
      pitDest.AirportCodeHashKey = br.DestHashKey
         AND pitDest.SnapshotDate = br.SnapshotDate
  )
```

This table includes no business logic for the aggregations, because these operations were already performed when loading the bridge table. Because these aggregated measures are included in the bridge table, no additional data from the no-history link or satellite are required as well. In addition to the measures, this fact table includes the key values to dimension tables for origin airport and destination airport. Additional descriptive data could be joined from hubs and satellites if required for a specific target.

14.4 IMPLEMENTING TEMPORAL DIMENSIONS

The examples from sections 14.2 and 14.3 have demonstrated how to provide Type 2 dimensions using joins between the hub table and dependent satellites or the PIT table and dependent satellites. All of these joins were based on the **load date** to find the record in the dependent satellite that is current for a given snapshot date. The load date was used because it provides information about the technical validity of a record in the history of the data: which data was current at a given point in time, from a technical perspective.

However, in some cases, business users don't want to analyze the data from a technical perspective. Instead, they are interested in a temporal perspective that is based on the effectivity dates, defined by the business. These effectivity dates come in various ways, for example **valid from** and **valid to** dates and **membership start** and **membership end** dates. Chapter 5, Intermediate Data Vault Modeling, has shown how to store such effectivity dates in effectivity satellites, which are added to hubs and links to indicate if business keys in hubs are deleted in the source system or have become invalid, and the validity of relationships between business keys in Data Vault 2.0 links. These satellites also ensure that changes to these effectivity dates are tracked in an auditable manner.

In order to create Type 2 dimensions that reflect the temporal perspective, a special form of a PIT table can be used. The following table implements such a temporal PIT:

```
CREATE TABLE [biz].[TPITAirportCode](
       [AirportKey] [char](32) NOT NULL,
       [AirportCodeHashKey] [char](32) NOT NULL,
       [SnapshotDate] [datetime2](7) NOT NULL,
       [OriginAirportHashKey] [char](32) NOT NULL,
       [OriginLoadDate] [datetime2](7) NOT NULL,
       [DestAirportHashKey] [char](32) NOT NULL,
       [DestLoadDate] [datetime2](7) NOT NULL,
 CONSTRAINT [PK_TPITAirportCode] PRIMARY KEY NONCLUSTERED
(
       [AirportKey] ASC
) ON [INDEX],
 CONSTRAINT [UK_TPITAirportCode] UNIQUE NONCLUSTERED
(
       [AirportCodeHashKey] ASC,
       [SnapshotDate] ASC
) ON [INDEX]
) ON [DATA]
```

Notice that there is no difference in the structure between a standard PIT and a temporal PIT. However, instead of prejoining the data based on load date, the effectivity date or any other descriptive date is used when loading the temporal PIT table:

```
INSERT INTO biz.TPITAirportCode
SELECT
       UPPER(CONVERT(char(32),
              HASHBYTES('MD5', CONCAT(
                     UPPER(RTRIM(LTRIM(hub.AirportCode)))
                     , ';'
                     , CONVERT(NVARCHAR(30), ?, 126)
              ))
```

```
       ,2)) AS AirportKey
       , hub.AirportCodeHashKey
       , CONVERT(datetime2(7), ?) AS SnapshotDate
       , COALESCE(sato.AirportHashKey, REPLICATE('0', 32)) AS OriginAirportHashKey
       , COALESCE(sato.LoadDate, '0001-01-01 00:00:00.000') AS OriginLoadDate
       , COALESCE(satd.AirportHashKey, REPLICATE('0', 32)) AS DestAirportHashKey
       , COALESCE(satd.LoadDate, '0001-01-01 00:00:00.000') AS DestLoadDate
FROM
       DataVault.[raw].HubAirportCode hub
LEFT OUTER JOIN DataVault.[raw].SatDestAirportMod2 satd ON (
       satd.AirportHashKey = hub.AirportCodeHashKey
       AND satd.LoadEndDate IS NULL
       AND ? BETWEEN
              satd.ValidFrom AND COALESCE(satd.ValidTo, '9999-12-31 23:59:59.999')
)
LEFT OUTER JOIN DataVault.[raw].SatOriginAirportMod2 sato ON (
       sato.AirportHashKey = hub.AirportCodeHashKey
       AND sato.LoadEndDate IS NULL
       AND ? BETWEEN
              sato.ValidFrom AND COALESCE(sato.ValidTo, '9999-12-31 23:59:59.999')
)
WHERE NOT EXISTS (
       SELECT
              1
       FROM
              biz.TPITAirportCode tgt
       WHERE
              hub.AirportCodeHashKey = tgt.AirportCodeHashKey
                     AND ? = tgt.SnapshotDate
)
```

The join conditions are based on the currently active record from the satellite, as indicated by a **load end date** of NULL, and the **snapshot date** from the PIT between the **valid from** and **valid to** dates from the descriptive satellite (not an effectivity satellite). However, if the effectivity dates change in the raw data, the PIT needs to be updated for past records or records are deleted from the PIT and the above statement inserts the current view requested by the business. Updating the table will be costly from a performance standpoint. Instead, add a load date to the temporal PIT table and partition over it and remove old partitions whenever the current partition was successfully loaded. By doing so, the temporal PIT table is turned into a rolling history of joins.

In order to present the data to the business user, a similar view can be used as in section 14.3.3:

```
CREATE VIEW [dbo].[DimAirport5] AS
SELECT
       pit.AirportKey
       ,pit.AirportCodeHashKey
       ,pit.SnapshotDate
       ,CASE
         WHEN hub.AirportCode IS NOT NULL AND hub.AirportCode <> '' THEN hub.AirportCode
         ELSE '?'
        END AS AirportCode
       ,COALESCE(satd.[DestCityName], sato.[OriginCityName], 'Unknown') AS CityName
```

```
       ,COALESCE(satd.[DestState], sato.[OriginState], '?') AS [State]
       ,COALESCE(satd.[DestStateName], sato.[OriginStateName], 'Unknown') AS StateName
       ,COALESCE(satd.[DestCityMarketID], sato.[OriginCityMarketID], 0) AS CityMarketID
       ,COALESCE(satd.[DestStateFips], sato.[OriginStateFips], 0) AS StateFips
       ,COALESCE(satd.[DestWac], sato.[OriginWac], 0) AS Wac
FROM
       DataVault.[biz].[TPITAirportCode] pit
INNER JOIN DataVault.[raw].HubAirportCode hub ON (
       pit.AirportCodeHashKey = hub.AirportCodeHashKey
)
INNER JOIN DataVault.[raw].SatOriginAirportMod sato ON (
       sato.AirportHashKey = pit.OriginAirportHashKey
       AND sato.LoadDate = pit.OriginLoadDate
)
INNER JOIN DataVault.[raw].SatDestAirportMod satd ON (
       satd.AirportHashKey = pit.DestAirportHashKey
       AND satd.LoadDate = pit.DestLoadDate
)
```

The only difference between the two views is the primary source of the fact table, which is the above temporal PIT table instead of the standard PIT. Other than that, the view definition is exactly as before. Therefore, it is also easy to use for power users who directly access the Data Vault 2.0 model, because they only need to change the PIT source in order to access a temporal view of the data instead of the technically historized view.

14.5 IMPLEMENTING DATA QUALITY USING PIT TABLES

Another application of PIT tables is to use it for data cleansing purposes. In some (rare) cases, it might be appropriate to use a PIT table for master and duplicate resolution and other data cleansing activities. The following DDL creates just another version of the PIT table used before:

```
CREATE TABLE [biz].[QPITAirportCode](
       [AirportKey] [char](32) NOT NULL,
       [AirportCodeHashKey] [char](32) NOT NULL,
       [SnapshotDate] [datetime2](7) NOT NULL,
       [OriginAirportHashKey] [char](32) NOT NULL,
       [OriginLoadDate] [datetime2](7) NOT NULL,
       [DestAirportHashKey] [char](32) NOT NULL,
       [DestLoadDate] [datetime2](7) NOT NULL,
 CONSTRAINT [PK_QPITAirportCode] PRIMARY KEY NONCLUSTERED
(
       [AirportKey] ASC
) ON [INDEX],
 CONSTRAINT [UK_QPITAirportCode] UNIQUE NONCLUSTERED
(
       [AirportCodeHashKey] ASC,
       [SnapshotDate] ASC
) ON [INDEX]
) ON [DATA]
```

Again, the only difference between this PIT table and the ones used before is the name. The column definitions remain the same. However, the loading statement differs:

```
INSERT INTO biz.QPITAirportCode
SELECT
       UPPER(CONVERT(char(32), HASHBYTES('MD5',
              CONCAT(UPPER(RTRIM(LTRIM(hub.AirportCode))), ';', CONVERT(NVARCHAR(30), ?,
       126))),2)) AS AirportKey
       , hub.AirportCodeHashKey
       , CONVERT(datetime2(7), ?) AS SnapshotDate
       , COALESCE(sato.AirportHashKey, REPLICATE('0', 32)) AS OriginAirportHashKey
       , COALESCE(sato.LoadDate, '0001-01-01 00:00:00.000') AS OriginLoadDate
       , CASE
          WHEN satd.DestCityName = 'Frisco, CA' THEN REPLICATE('0', 32)
          ELSE COALESCE(satd.AirportHashKey, REPLICATE('0', 32))
        END AS DestAirportHashKey
       , CASE
          WHEN satd.DestCityName = 'Frisco, CA' THEN '0001-01-01 00:00:00.000'
          ELSE COALESCE(satd.LoadDate, '0001-01-01 00:00:00.000')
        END AS DestLoadDate
FROM
       DataVault.[raw].HubAirportCode hub
LEFT OUTER JOIN DataVault.[raw].SatDestAirportMod satd ON (
       satd.AirportHashKey = hub.AirportCodeHashKey
       AND ? BETWEEN
              satd.LoadDate AND COALESCE(satd.LoadEndDate, '9999-12-31 23:59:59.999')
)
LEFT OUTER JOIN DataVault.[raw].SatOriginAirportMod sato ON (
       sato.AirportHashKey = hub.AirportCodeHashKey
       AND ? BETWEEN
              sato.LoadDate AND COALESCE(sato.LoadEndDate, '9999-12-31 23:59:59.999')
)
WHERE NOT EXISTS (
       SELECT
              1
       FROM
              biz.QPITAirportCode tgt
       WHERE
              hub.AirportCodeHashKey = tgt.AirportCodeHashKey
                     AND ? = tgt.SnapshotDate
)
```

The preceding statement loads the DQ PIT but cleanses the data by setting the **DestAirportHashKey** and **DestLoadDate** to the ghost record in the case that the **DestCityName** is "Frisco, CA". Similarly any other business logic could be applied when loading the DQ PIT table. The business logic in the dependent dimension view would automatically pick a record from an alternate satellite if the ghost record is found. The advantage of this approach is that it is transparent to the user: similarly to temporal dimensions, power users could just use the DQ PIT to source cleansed dimension data. On the other hand, the business logic is hidden in the loading procedure of the PIT table. A similar approach could be achieved by adding a computed satellite with cleansed data that could also be added to the standard (or temporal) PIT table. If the power user wants to use cleansed information, this computed satellite is joined to the PIT table instead of the raw satellite.

However, this example shows the power of PIT tables (and similarly bridge tables) for delivering the data that is required by the business user.

14.6 DEALING WITH REFERENCE DATA

In many cases, reference codes are included in Raw Data Vault satellites. These codes vary from source system to source system but business users expect a conformed view on such codes in order to run analytical statements across source systems.

Instead of modifying the code in the Raw Data Vault satellite, which would compromise the auditability of the Raw Data Vault, the code from a specific source system is replaced or enriched by descriptive data from reference tables when the Business Vault or information marts are being built. The following DDL creates a simplified computed satellite in the Business Vault:

```
CREATE VIEW biz.TSatFlightDelay AS
SELECT
       [FlightHashKey]
       ,[LoadDate]
       ,[RecordSource]
       ,[DepDelay]
       ,[DepDelayMinutes]
       ,[DepDel15]
       ,[DepartureDelayGroups]
       ,[ArrDelay]
       ,[ArrDelayMinutes]
       ,[ArrDel15]
       ,[ArrivalDelayGroups]
       ,[CarrierDelay]
       ,[WeatherDelay]
       ,[NASDelay]
       ,[SecurityDelay]
       ,[LateAircraftDelay]
FROM
       [DataVault].[raw].[TSatFlight]
```

This computed satellite is directly based on the satellite **TSatFlight** in the Raw Data Vault. It is merely a selection of some columns without any filtering or computing of rows that only serves as the playground for the example in this section.

The next statement modifies this computed satellite by adding descriptive data from a reference table to the satellite:

```
CREATE VIEW biz.TSatFlightDelay2 AS
SELECT
       s.[FlightHashKey]
       ,s.[LoadDate]
       ,s.[RecordSource]
       ,s.[DepDelay]
       ,s.[DepDelayMinutes]
       ,s.[DepDel15]
       ,s.[DepartureDelayGroups]
```

```
        ,rd.Name AS DepDelayName
        ,rd.Abbreviation AS DepDelayAbbr
        ,s.[ArrDelay]
        ,s.[ArrDelayMinutes]
        ,s.[ArrDel15]
        ,s.[ArrivalDelayGroups]
        ,ra.Name AS ArrDelayName
        ,ra.Abbreviation AS ArrDelayAbbr
        ,s.[CarrierDelay]
        ,s.[WeatherDelay]
        ,s.[NASDelay]
        ,s.[SecurityDelay]
        ,s.[LateAircraftDelay]
FROM
        [DataVault].[raw].[TSatFlight] s
INNER JOIN
        [DataVault].[raw].[RefDelayGroup] rd ON (s.[DepartureDelayGroups] = rd.Code)
INNER JOIN
        [DataVault].[raw].[RefDelayGroup] ra ON (s.[ArrivalDelayGroups] = ra.Code)
```

The descriptive data is joined from the reference table **RefDelayGroup**, which is just a view on a master data table in MDS. The reference table is joined twice because it is used to describe two codes in the satellite (**DepartureDelayGroups** and **ArrivalDelayGroups**).

By doing so, the descriptive data is added to the computed satellite by adding some of the attributes from the joined reference table to the view. This approach is perfectly fine, especially for creating star schemas. Instead of joining the data in the computed satellite, it could also be joined when creating the dimension tables or fact tables.

However, it is also possible to create a dimension based on reference tables:

```
CREATE VIEW DimDelayGroup AS
SELECT
        [Code] AS DelayGroupKey
        ,[Name]
        ,[Abbreviation]
        ,[Sort Order]
FROM
        [DataVault].[raw].[RefDelayGroup]
```

In this case, the **code** is used as the dimension key because it is defined as a unique value that never changes. Because of simplicity, the code is not hashed when creating dimension tables for reference data. Once the dimension table is defined, the dimension key is referenced by a fact table (using the code). It is also possible to reference the dimension from another dimension, creating a snow-flake schema.

In many cases, there are mapping tables that help to map between codes from individual source systems to conformed codes for the data warehouse. The easiest approach is to create a master data entity with the code column used for the conformed code and individual columns per source system code. There are also options if multiple codes from the source system need to be mapped to the same conformed code in the data warehouse.

14.7 ABOUT HASH KEYS IN THE INFORMATION MART

The information mart uses hash keys to define the relationships between facts and dimensions. Because the information mart is provided virtually, this presents no problem in most cases. However, the OLAP cubes are often materialized and storage might become an issue. In other cases, performance might be affected to some extent.

The following sections provide recommended solutions and best practices to deal with storage and performance bottlenecks that might be due to hash keys.

14.7.1 ADVANTAGES OF USING HASH KEYS IN THE INFORMATION MART

The advantage of using hash keys in the information mart is the ease of use when sourcing the data. All other options require more complex solutions and loading processes. Chapter 15, Multidimensional Database, shows that the hash keys can be used when creating the multidimensional database without any problems from a structural perspective.

For Type 1 dimensions, the hash key from the hub is used to populate the key attribute in the dimension. If Type 2 dimensions should be provided, the key from the PIT table is used because it identifies each change in the PIT (and thus the dimension table) uniquely.

Using hash keys in the data warehouse, including in the dimensional model, is future-proof for all requirements regarding the volume, variety and velocity of data and thus the recommended approach for building information marts and multidimensional databases.

We truly believe that you should only deviate from this recommendation if you *really* need to.

14.7.2 REDUCE THE NUMBER OF DIMENSIONS IN CUBE

If you have a performance or storage issue in your solution, our first recommendation is to review the number of dimensions in your information mart and dependent cubes. In most cases, it is possible to reduce the number of dimensions by providing multiple cubes that are more tailored for specific business cases. This way, the required storage is reduced per cube, which improves the overall performance of the solution while maintaining ease of development and use.

14.7.3 USE FIXED BINARY DATA TYPE FOR HASH VALUES

If you still have an issue after reducing the number of dimensions in your solution or if reducing the dimensions is not an option, you should consider storing the hash keys using a binary datatype (but not a BLOB, CLOB, TEXT, or LOB data type if you're using other environments). For example, it is possible to store the result from the MD5 hash function in Microsoft SQL Server using a binary(16) column. Instead of storing it as a hexadecimal string, which requires 32 characters (thus 32 bytes), the hash value is stored in its binary format, requiring only 16 bytes.

The drawback of this solution is that some tools that are part of the BI stack of your organization might not support reading or writing to binary columns. This might become a problem if transparent access using views is also not possible. You should review the tools to be used in the future, including enterprise service buses (ESBs) (such as Microsoft BizTalk, which doesn't directly support binary datatypes) and the workarounds available (SQL views, stored procedures, etc.).

The advantage of storing the hash keys as character strings is that the string is supported in any tool.

14.7.4 REDUCE THE SIZE OF THE HASH KEY

Depending on the selected hash function, the size of the hash values might differ (refer to Chapter 11, Data Extraction, for a detailed discussion). If you're using SHA-1 (or higher) to hash your business keys, consider using MD5 because it provides an appropriate solution while maintaining a manageable hash value size. All other options require more storage and might affect performance in a more serious way. Avoid using anything higher than SHA-1 because you will probably not gain anything for the additional storage or reduced performance.

On the other hand, avoid using CRC or MD4 functions, due to increased risk for collisions. Our general recommendation is to use the MD5 hash function for calculating the hash keys.

14.7.5 INTRODUCE ADDITIONAL SEQUENCE NUMBERS

It is also possible to introduce sequence numbers in the PIT tables (for using them in Type 2 dimensions) or *close to hubs* (for example in hubs themselves or other structures). This should be your last choice, because it makes dealing with the data more complex.

Also note that such a solution requires that the sequence number be materialized (again, for example in the PIT table) to be stable for virtualization. If the sequence number is not materialized somewhere, it is not possible to incrementally load information marts and OLAP cubes. It is also not possible to perform cross-business queries that join multiple information marts or cubes.

This solution would require the replacement of the key column (the hash value that is calculated per parent business key and snapshot date) by an integer sequence value. This value can be used to identify Type 2 dimensions. Because the PIT table probably contains many records, a bigint data type is required. For Type 1 dimensions, a hash key should be introduced per business key in hubs.

REFERENCES

[1] U.S. Diplomatic Mission to Germany (website), 2015, "About the USA", available at http://usa.usembassy.de/travel-regions.htm.

[2] Microsoft, Microsoft Association Algorithm Technical Reference, 2015, available at https://msdn.microsoft.com/en-us/library/cc280428.aspx.

[3] Judith R. Davis, Robert Eve: "Data Virtualization," p. 47ff.

CHAPTER

MULTIDIMENSIONAL DATABASE

15

The previous chapters of this book have described how to set up the relational part of the data warehouse. It provides the raw data and information for users to perform their analytical work, either using ad-hoc SQL queries or, more commonly, analytical applications, such as Microsoft SQL Server Reporting Services or spreadsheets, e.g., PowerPivot for Microsoft Excel.

However, most casual business users are not dealing with the relational data warehouse. Instead, many of them prefer multidimensional databases as their main interface for accessing the information for analytical purposes. These multidimensional databases are designed to support Online Analytic Processing (OLAP) and are an effective tool to consume information because it allows business users to formulate queries and get quick responses [1]. A multidimensional database is best suited when the end-user wants to work with aggregated information instead of raw facts.

Microsoft SQL Server Analysis Services (SSAS) provides an OLAP solution developed by Microsoft as part of their SQL Server Business Intelligence stack. SSAS also provides a Tabular mode that contains the detailed data [2]. It provides the presentation database that includes aggregations and indexes to provide high query performance [1]. SSAS has some interesting characteristics [1]:

- **User-defined metadata:** the multidimensional database consists of one or more OLAP cubes, which are based on facts and dimensions, closely following the definitions known from the information marts created in the previous chapter. SSAS also supports hierarchies, and the option to combine facts from multiple OLAP cubes. If the relational foundation was built using the principles in the previous chapter, it becomes possible to run queries across multiple OLAP cubes.
- **Query performance:** SSAS provides superior query performance when dealing with analytic queries. Such queries are characterized by grouping and aggregating when executed against a relational database.
- **Aggregation management:** one of the reasons why OLAP cubes provide superior performance for analytic queries is the fact that they precompute the data at different grain levels. This aggregation management is transparent to the business user.
- **Calculations:** in some cases, it is required to add calculations to the OLAP cube, for example when dealing with percentages. SSAS provides such calculated measures among other calculations.
- **Security provisions:** SSAS provides complex security rules to protect the aggregated data against unauthorized use.

As a summary, SSAS is best for providing aggregated information. It is not a good database to provide detailed information, typically dealt with in operational business processes. This is where a relational database or a NoSQL environment becomes more appropriate.

The goal of this last chapter is to complete the end-to-end discussion about how to build a scalable data warehouse. It is not going to replace an OLAP or SSAS book. It will, however, focus on some

characteristics that are unique for information marts prepared and built during the previous chapters (even though there are only a few, mainly due to the fact that we're dealing with virtualized information marts).

15.1 ACCESSING THE INFORMATION MART

The information mart provides ***all*** the information that should be presented to the end-user. If this is not the case, because some data from the Raw Data Vault is missing, it should be included by the information mart first, before the SSAS database is being built. Accessing the Raw Data Vault or the Business Vault directly to retrieve additional information or raw data that is not available in the information mart should be avoided.

Also, there might be multiple information marts. Each OLAP cube should only access one information mart in most cases. If data is required from other information marts, this data should be added to the primary information mart. With virtualized facts and dimensions, this becomes very easy and doesn't require additional storage. In some cases, it requires that business logic needs to be moved upstream from the loading procedures of the second information mart into the Business Vault in order to be accessible for the primary information mart. This is the desired solution because the Business Vault should provide reusable business rules, not any one of the information marts.

15.1.1 CREATING A DATA SOURCE

Therefore, there should be only one information mart required to build the SSAS database. Create a new **Analysis Services Multidimensional and Data Mining Project** in Microsoft SQL Server Data Tools and add a new data source to the project.

Create a new data source in the **solution explorer**. The **data source wizard** is presented. After the initial welcome page, a page appears that is used to select the connection string. If the connection string for the information mart is not available yet, create one by selecting the **new** button. The dialog in Figure 15.1 is shown.

Enter the server name, your user credentials to the information mart and the database that provides the relational information mart entities. After selecting the OK button, the connection appears in the previous dialog, as shown in Figure 15.2.

Make sure that the connection to the information mart is selected and select the **next** button. The next dialog configures the impersonation information of Analysis Services (Figure 15.3).

The provided settings are used by Analysis Services to connect to the relational information mart. The following options are available [3]:

1. **Use a specific Windows user name and password:** this option allows a specific user account and its password to be provided, to access the relational information mart. It is a useful option if a dedicated Windows user account was created to access the information mart, for example with only read-only privileges.
2. **Use the service account:** the service account that is running the Analysis Services database is used to access the information mart. This is the default option. In order to use this option, a database login should be created for the service account. It also requires granting of access to the information mart.

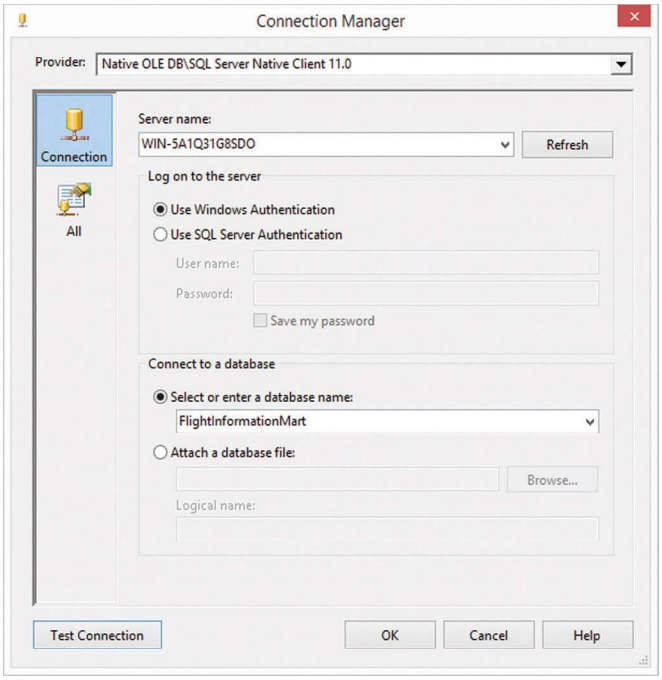

FIGURE 15.1

Create connection to information mart.

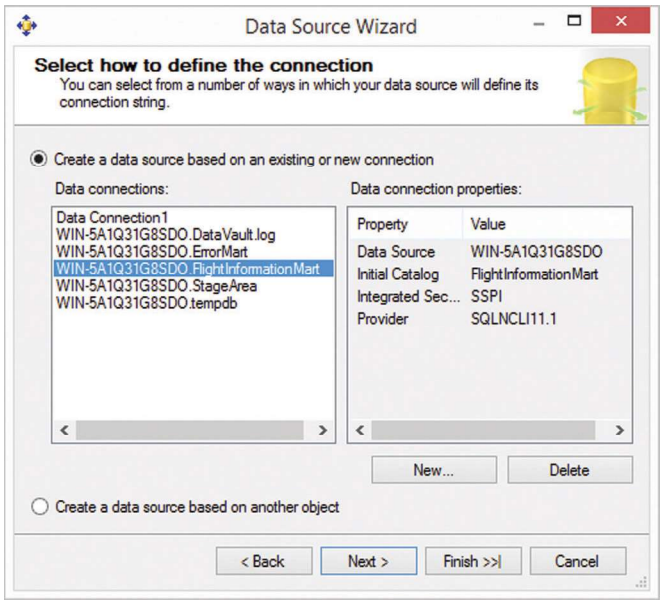

FIGURE 15.2

Select information mart connection in the data source wizard.

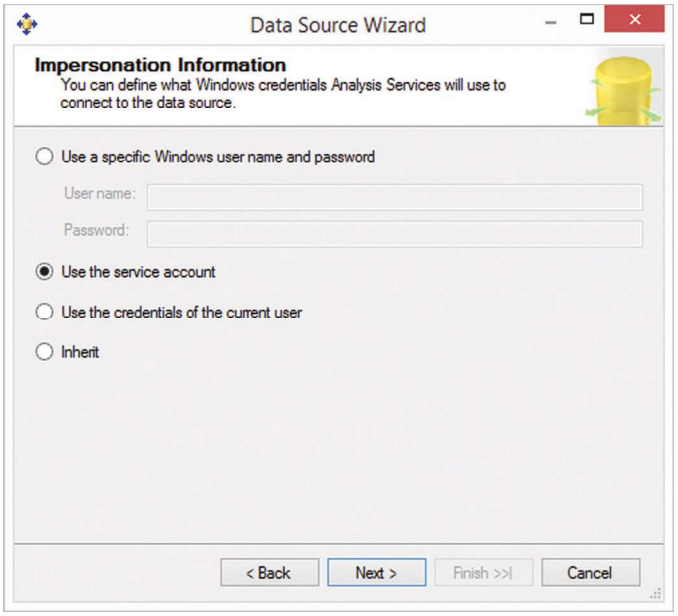

FIGURE 15.3

Impersonation information configuration.

3. **Use the credentials of the current user:** In some rare cases, the current user should be used for accessing the relational information mart. Note that this option is not available for multidimensional databases if they are located on the database backend.
4. **Inherit:** this option uses the impersonation options of the parent database (the Analysis Service database). It is helpful when setting the options centrally.

Select the appropriate setting and provide user credentials if necessary. If you don't know what option to choose, ask your database administrator or select the **use the service account** option as a best guess. It will require that the service account (the Windows account that is running Analysis Services) has access to the relational information mart.

Select the **next** button to continue. The data source wizard is completed with the page shown in Figure 15.4.

Make sure that the name of the data source is correct and select the **finish** button to complete the wizard.

15.1.2 CREATING DATA SOURCE VIEW

Once the data source for the information mart has been successfully configured, a data source view needs to be created which is used to access the relational database model in the information mart. From the solution explorer, select **new data source view** from the context menu of the **data source views** folder. The **data source view wizard** appears. After the initial welcome page of the wizard, the page in Figure 15.5 is shown.

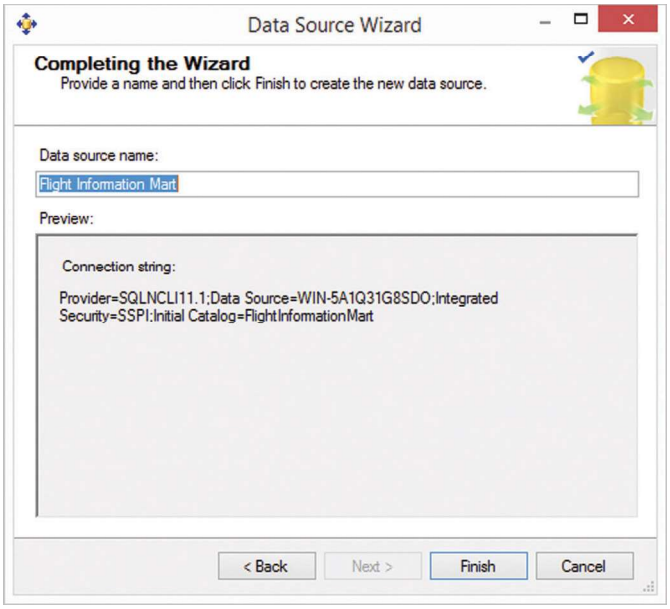

FIGURE 15.4

Complete the data source wizard.

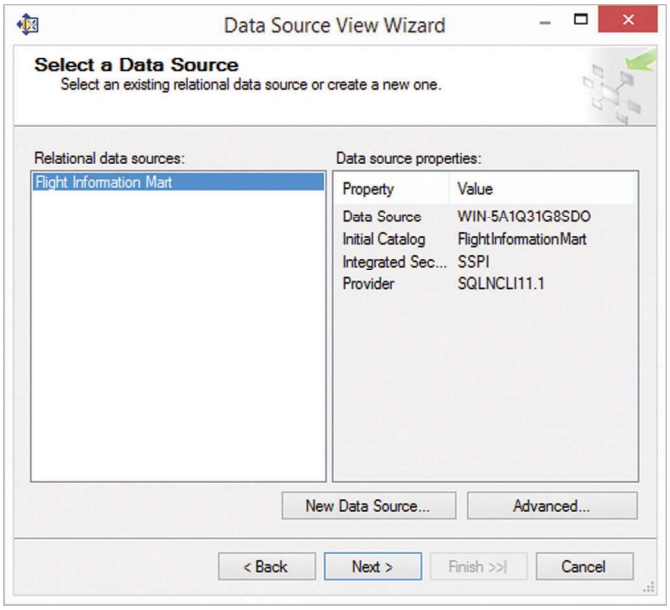

FIGURE 15.5

Selecting a data source in the data source view wizard.

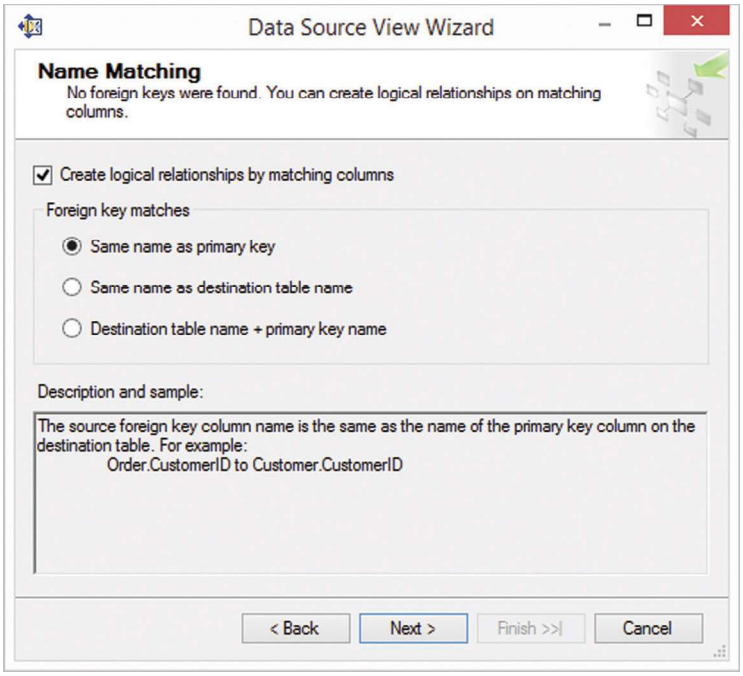

FIGURE 15.6

Configure name matching for the data source view.

Select the data source to the relational information mart that was just created and select the **next** button. The page presented in Figure 15.6 is shown.

In most cases, information marts don't use foreign key references. Therefore, Analysis Services offers the option to create logical relationships from the metadata of the tables found in the information mart. The relationships are found based on the column and table names. However, as shown in Figure 15.9, this matching is not 100% accurate, especially if naming conventions are not thoroughly followed. SSAS offers the following methods to detect the foreign key relationships in the information mart [4]:

1. **Same name as primary key:** the logical relationship is created based on the equality of the fact table column name and the name of the dimension's primary key.
2. **Same name as destination table name:** the logical relationship is created when the fact table column name matches the name of the dimension table.
3. **Destination table name + primary key name:** only if the fact table column name matches the dimension's table name concatenated with the name of the primary key column.

Select an appropriate matching method, based on your naming conventions and select the **next** button. The next dialog is presented in Figure 15.7.

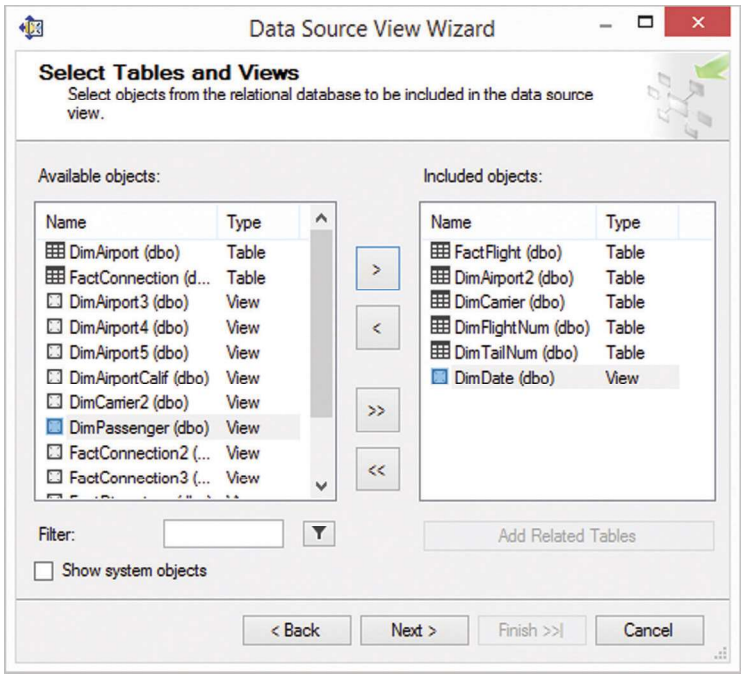

FIGURE 15.7

Select tables and views from the information mart.

Select the tables and views (usually the facts and dimensions in the dimensional model) that should be included in the data source view. Make sure that there is at least one selected fact table and all required dimension tables on the right list of **included objects**. Select the **next** button. The next page shows the settings of the configured data source view (Figure 15.8).

Review the settings and make sure that the name of the data source view meets your expectations. Select the **finish** button to complete the wizard. The selected tables from the information mart are added to the data source view and presented as a logical model in the center of the application (Figure 15.9).

Note that there are some missing logical relationships. For example, the relationships between the fact table and the **airport** dimension are missing because the fact columns are not using the same name as the primary key column in the dimension. For this reason, the automatic mapping did not work. The same applies to **FlightNumKey** in **DimFlightNum**, which is not mapped to the column with the same name in the fact table. Drag the column from the fact table to the primary key of the corresponding dimension to create a logical relationship in both cases. Note that **DimAirport2** is referenced twice: once for **OriginKey** and **DestKey** in the fact table. The corrected data source view is presented in Figure 15.10.

This completes the first step to access the information mart that was created throughout the book. The next steps are creating dimensions based on the dimension tables and the cube itself, based on the fact table.

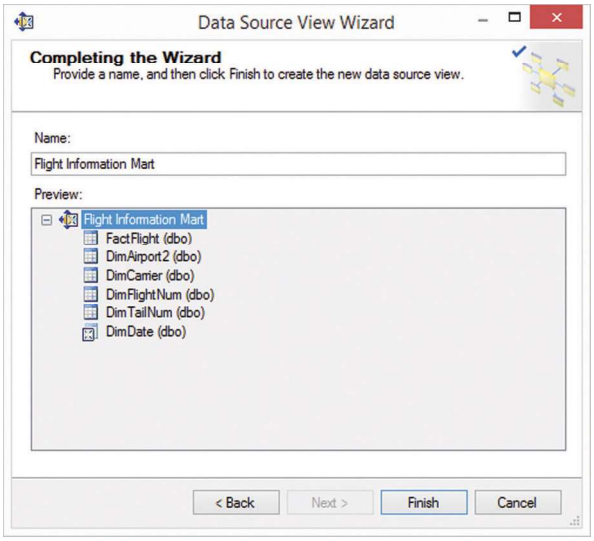

FIGURE 15.8

Complete the data source view wizard.

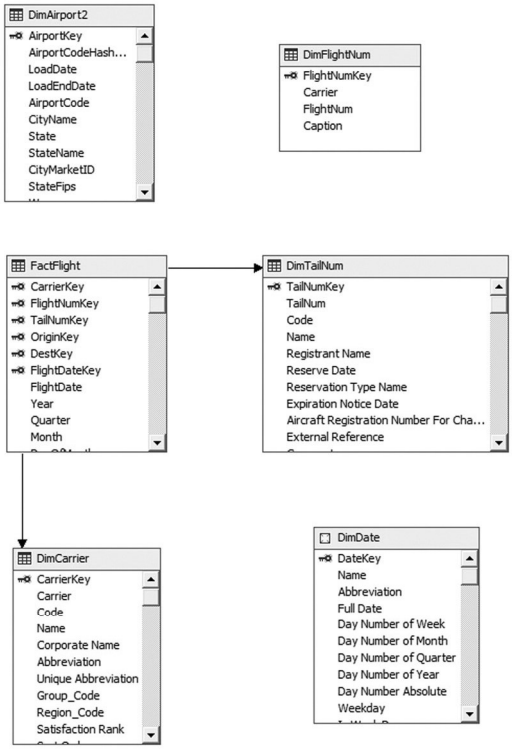

FIGURE 15.9

Initial data source view with missing logical relationships.

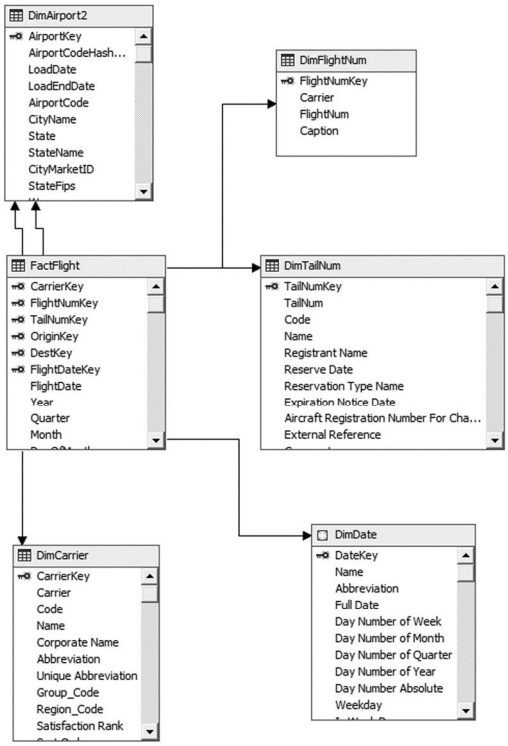

FIGURE 15.10

Data source view after logical relationships have been fixed.

15.2 CREATING DIMENSIONS

The data source view, created in section 15.1.2, provides SSAS access to and defines the relational information mart. The following sections of this chapter describe how the multidimensional database and the OLAP cubes are defined based on this data source view. The first step is to define the dimensions of the database and where the dimension data is being sourced.

To create a new dimension, select **new dimension** from the context menu of the **dimension** folder in the **solution explorer**. Click the **next** button on the welcome page of the dimension wizard. The page shown in Figure 15.11 is presented.

The page allows you to define the data source of the dimension or to create new data for some specific cases. The following options are available [5]:

1. **Use an existing table:** create a dimension from an existing table in the information mart. The information mart table will influence which attributes will be available for inclusion to the dimension.
2. **Generate a time table in the data source:** create a new time table in the information mart and use this table as the source for the dimension.

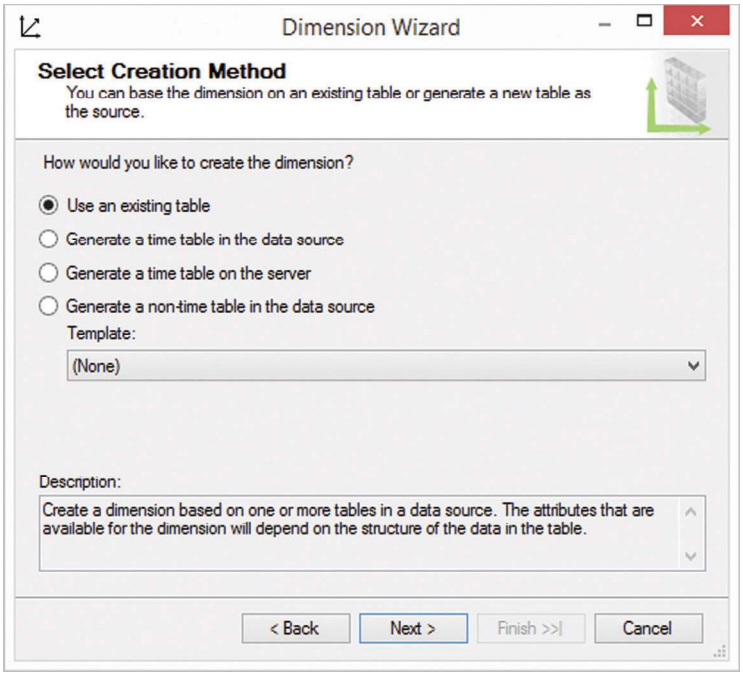

FIGURE 15.11

Selecting a creation method in the dimension wizard.

3. **Generate a time table on the server:** create a new time table on the SSAS server database and not the information mart.
4. **Generate a nontime table in the data source:** create a new table in the information mart, based on dimension attributes created in this wizard. An ETL job is required to load the dimension table created in this top-down approach.

Select the first option to create a new dimension in the multidimensional database based on data from the information mart. This is the recommended option for both standard dimensions and date dimensions. Refer to the next section which provides an alternate method to create a time table with more business-user involvement than the standard approach in SSIS.

After the **next** button is selected, the page in Figure 15.12 is shown.

This page asks for the table and columns information to source a new dimension from an existing table, as selected on the previous page. Select the data source view that was created in section 15.1.2 and one of the dimension tables.

Make sure that the **key column** was selected as key column. There should be only one key column per dimension table.

The **name column** defines the default caption for the member in the dimension. Select an appropriate and useful column that provides a distinguishable and understandable name for each member in the dimension.

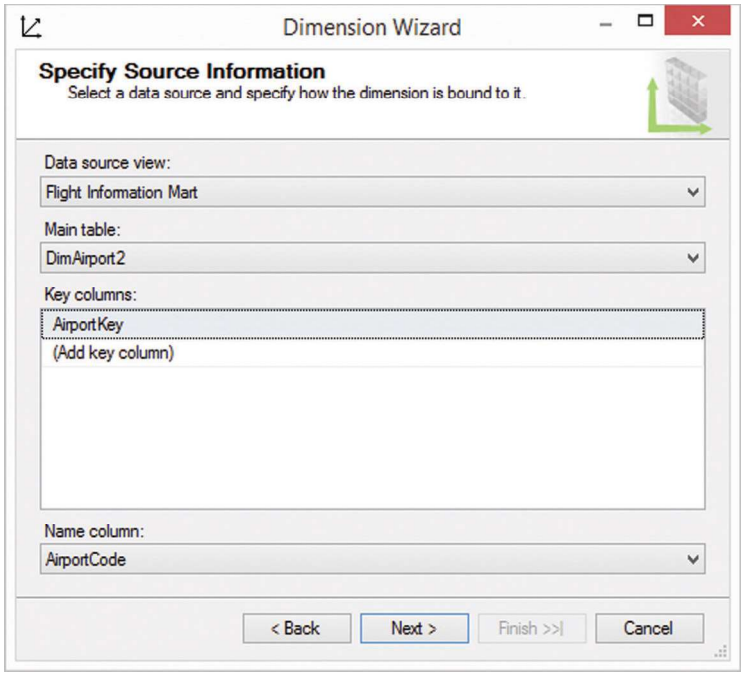

FIGURE 15.12

Specify source information in the dimension wizard.

After setting up this configuration, select the **next** button to select the attributes of the dimension on the next page, as shown in Figure 15.13.

Select all the columns from the source table that should be usable by the business user and include them into the dimension. It is possible to set up various attribute types that change the behavior of SSAS for common use cases, such as currency and geography dimensions [6].

After setting up the attributes, click the **next** button to proceed to the next page, which completes the dimension wizard (Figure 15.14).

Review the settings for the dimension and make sure to provide a dimension name that is meaningful for the business user. Click **finish** to create the dimension in the multidimensional database.

Repeat the process for other dimensions in the dimensional model, such as **DimCarrier**, **DimFlightNum** and **DimTailNum**. The next section describes how to set up a date dimension, based on the **DimDate** entity in the information mart.

15.2.1 DATE DIMENSION

While Analysis Services provides the capability to set up a date dimension (time table) from the dimension wizard, there is some advantage of running the process on your own. Instead of using the time data provided by SSAS, the recommended managed self-service BI approach is to provide an analytical

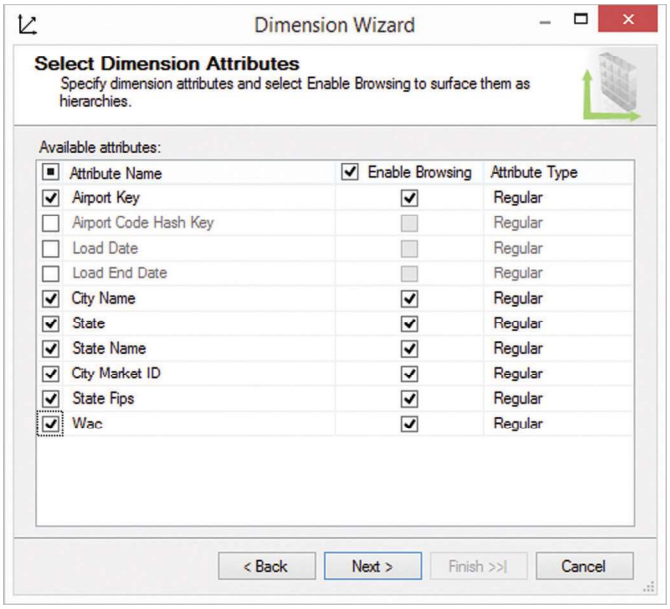

FIGURE 15.13

Select dimension attributes in the dimension wizard.

FIGURE 15.14

Complete the dimension wizard.

master data table with the dates that should be included in the date dimension and their corresponding descriptive fields. The advantage of this approach is that the descriptive fields can be modified by the business user, for example when changing date and months abbreviations or captions. The Microsoft Data Services (MDS) **DWH** model supplied with this book provides members for a date dimension with descriptive attributes that can be overwritten in MDS.

The information mart on the companion Web site includes a **DimDate** table that sources a limited number of descriptive attributes for the date dimension:

```
CREATE VIEW [dbo].[DimDate] AS
SELECT CONVERT(int, [Code]) AS DateKey
      ,[Name]
      ,[Abbreviation]
      ,[Full Date]
      ,[Quarter Number of Year]
      ,[Quarter Abbreviation]
      ,[Month Number of Year]
      ,[Month Abbreviation]
      ,[Day Number of Week]
      ,[Day Number of Month]
      ,[Day Number of Quarter]
      ,[Day Number of Year]
      ,[Day Number Absolute]
      ,[Year Number]
      ,[Weekday]
      ,[Is Week Day]
      ,[Is Last Day of Week]
      ,[Is Last Day of Month]
      ,[Is Last Day of Quarter]
      ,[Is Last Day of Year]
      ,[Sort Order]
      ,[External Reference]
      ,[Comments]
  FROM [DataVault].[raw].[RefDate]
```

This view in the information mart is directly based on a reference table in the Raw Data Vault and follows the approach for providing reference tables as dimensions outlined in Chapter 14, Loading the Information Mart. The reference table itself is a virtual view as well, because the analytical master data is under full control of the data warehouse and the other requirements for virtually providing reference data outlined in Chapter 12, Loading the Data Vault, are fully met.

In order to create a date dimension based on reference data from analytical master data, create a new dimension from the solution explorer and select the creation method (Figure 15.15).

Because the date dimension is sourced from reference data provided by the **DimDate** table in the information mart, select **use an existing table** again, following the approach in the previous section. Select **next** to continue (see Figure 15.16).

Select the **DateKey** dimension which is a code attribute (instead of a hash value, as in most other dimensions). Also, specify an appropriate **name** column before selecting the **next** button.

Set up the attributes of the date dimension on the page shown in Figure 15.17.

Select the attributes that should be provided by the date dimension. Also, consider setting the correct **attribute types**, which has been simplified in this example. After setting up the attributes, select

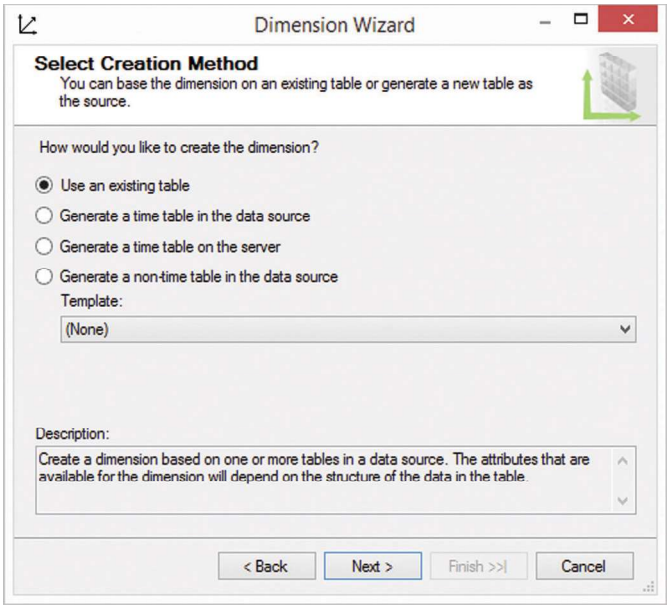

FIGURE 15.15

Select creation method for date dimension.

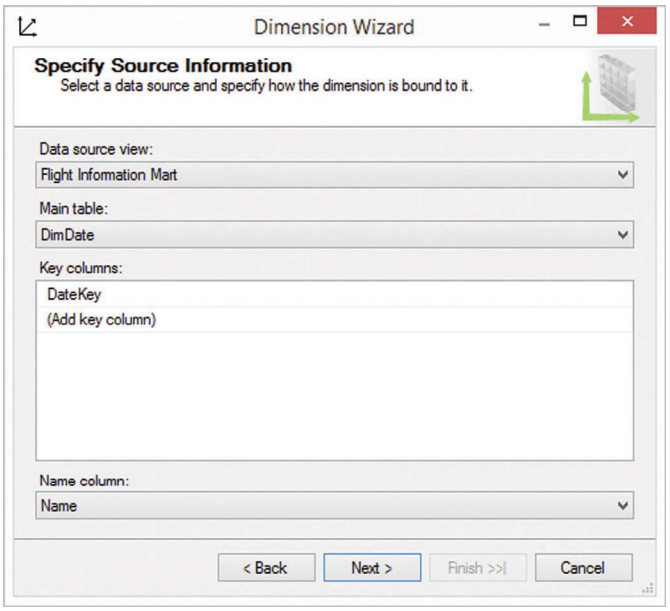

FIGURE 15.16

Specify source information for date dimension.

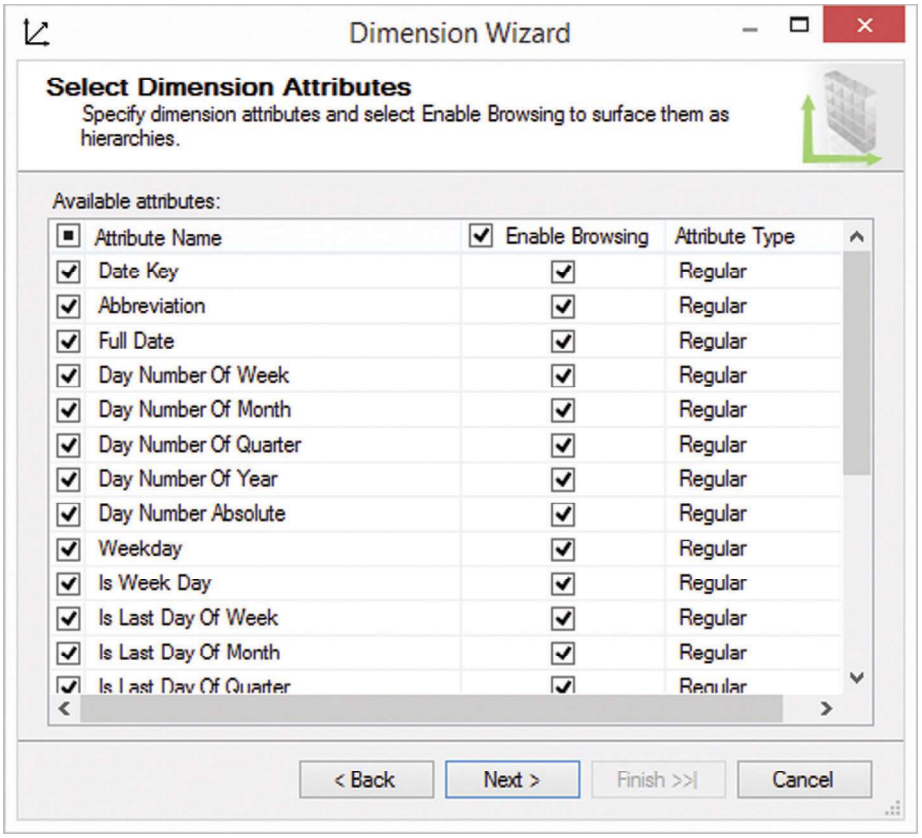

FIGURE 15.17

Select dimension attributes for date dimension.

the **next** button to proceed to the next page of the dimension wizard. The summary page will be shown as presented in Figure 15.18.

Review the settings and provide a meaningful dimension name before clicking the **finish** button. This completes the dimension wizard and adds the date dimension to the multidimensional database.

The final step is to define a date hierarchy that allows the end-user to analyze the data in the multidimensional table by different levels of grain, such as year, quarter, month or date. To define the hierarchy, open the **date** dimension in design mode (by using the context menu of the dimension in the solution explorer) and drag the **year number** attribute to the hierarchy canvas in the center of the screen. If you don't see the canvas shown in Figure 15.19, make sure that you're on the **dimension structure** tab. Add the following attributes to the hierarchy:

1. Year Number
2. Quarter Abbreviation
3. Month Abbreviation
4. Full Date

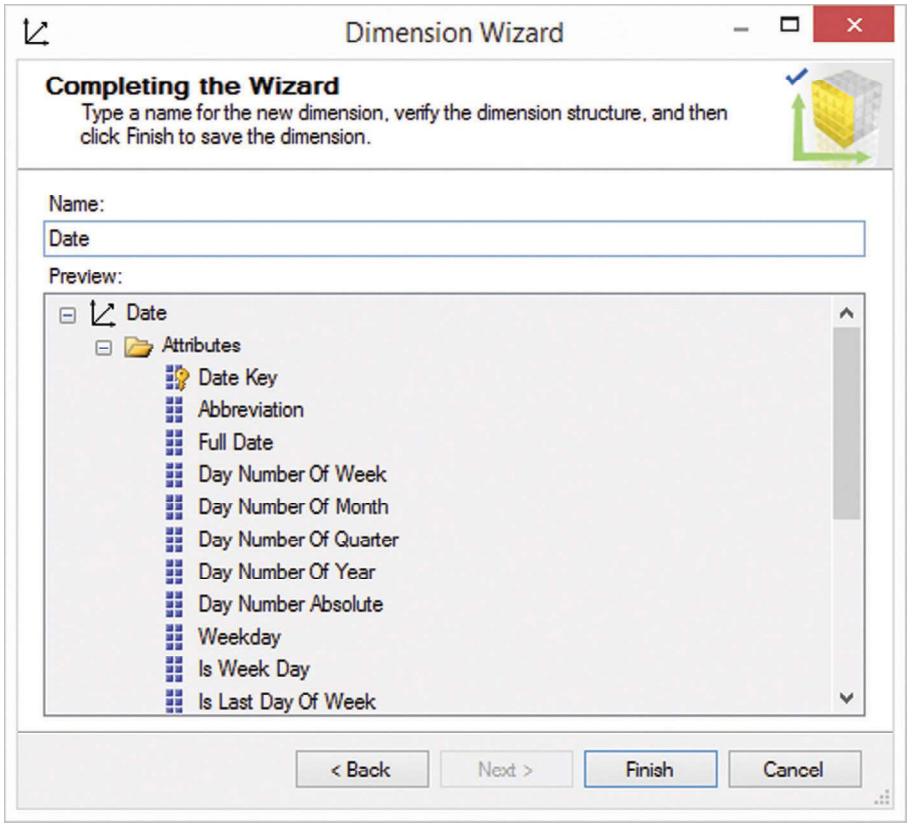

FIGURE 15.18

Completing the dimension wizard for the date dimension.

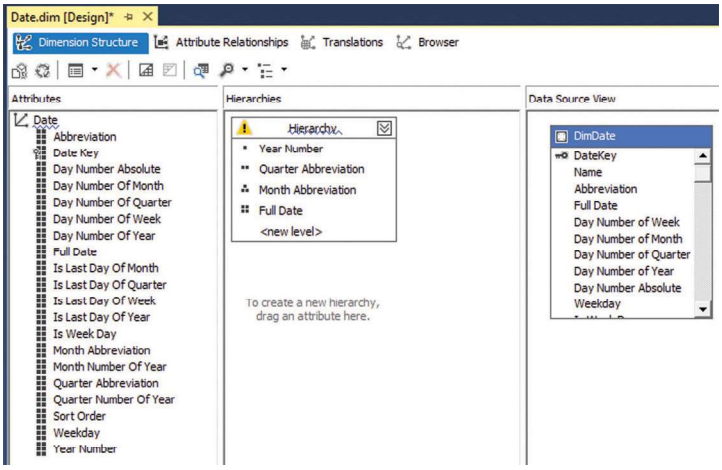

FIGURE 15.19

Defining the hierarchy of the date dimension.

The end result is shown in Figure 15.19.

It is also possible to set a hierarchy name to distinguish the hierarchy from other hierarchies, because it is possible to define multiple hierarchies per dimension. Save the dimension once your dimension structure is complete.

The example presented in this section provides an alternative to the standard date dimension in SSAS. Sourcing a date dimension from master data has the advantage that the business user has more control over the descriptive data but it also requires to create a relatively large table with analytical master data. This table is primarily intended for power users who know how to maintain the descriptive date information in the table and will not be managed by casual users.

15.3 CREATING CUBES

The dimensions created in this chapter are part of the multidimensional SSAS database and are used by multiple cubes. Each cube is based on one or more fact tables, but the recommendation is to create one virtual fact table in the information mart per cube. This way, it is possible to optimize the use of bridge tables for the final targets, because it improves the general performance as discussed in Chapter 14, Loading the Information Mart.

To create a new cube based on a fact table in the information mart, start the **cube wizard** by selecting **new cube** from the context menu of the **cubes** folder in the solution explorer. Click next on the welcome page of the wizard and select the creation method in the following page, shown in Figure 15.20.

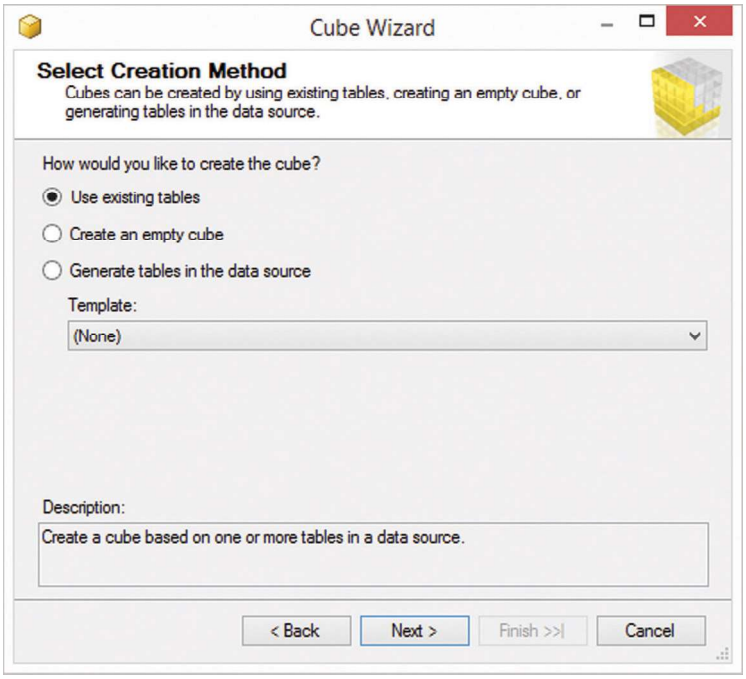

FIGURE 15.20

Selecting creation method in the cube wizard.

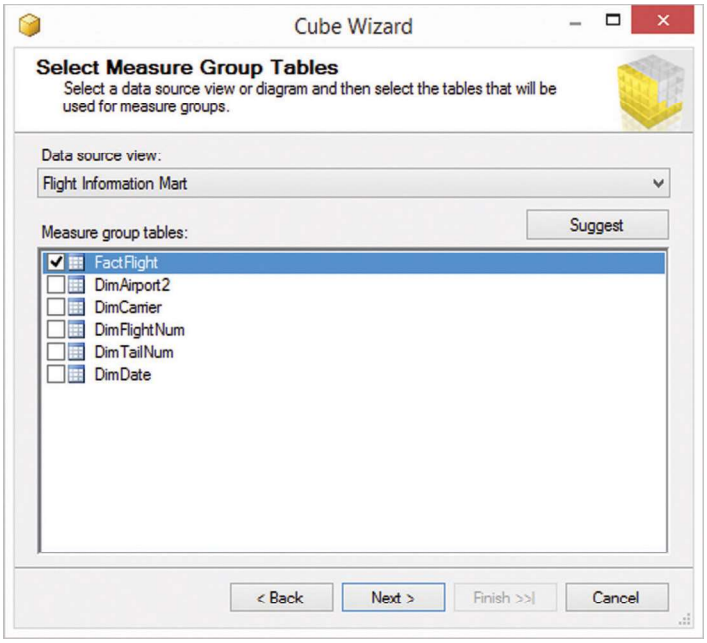

FIGURE 15.21

Select measure group tables in the cube wizard.

The page asks for the source of the data to be added as facts into the cube. The following options are available [7]:

1. **Use existing tables:** source the facts from an existing table (or view) in the information mart.
2. **Create an empty cube:** create a cube without sourcing any data at this time. This option is useful if all configurations should be performed manually without this wizard.
3. **Generate tables in the data source:** create a new fact table in the information mart based on the settings in this dialog. An ETL job is required to load the fact table.

Because the facts should be sourced from the fact table in the information mart, select **use existing tables** and click the **next** button. This will allow selection of the measure group table (Figure 15.21).

The measure group table is the fact table in the information mart that provides the facts and their measures. You should avoid selecting multiple fact tables here. Instead of doing so, create a new virtual fact table and (if required) another bridge table with the corresponding grain to improve the reusability of business logic that is responsible for defining the right grain in the bridge table and how it is presented to the business user (the virtual fact entity in the information mart).

Select the fact table and click the **next** button. The next page allows you to select the measures from the source table that should be included in the cube (Figure 15.22).

In this example, the following measures should be included in the flight cube:

- Dep Delay
- Dep Delay Minutes

- Taxi Out
- Taxi In
- Arr Delay
- Arr Delay Minutes
- CRS Elapsed Time
- Actual Elapsed Time
- Air Time

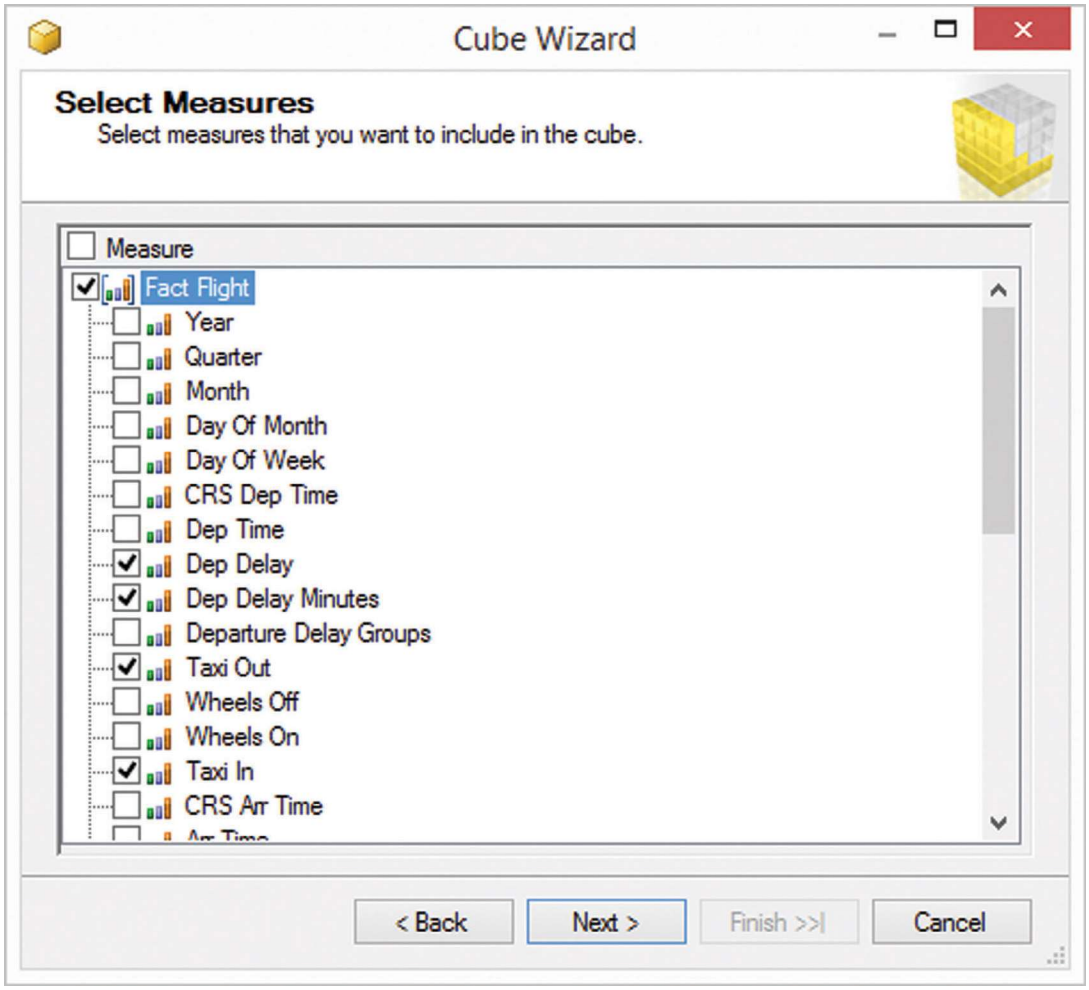

FIGURE 15.22

Select measures for the cube.

- Flights
- Distance
- Carrier Delay
- Weather Delay
- NAS Delay
- Security Delay
- Late Aircraft Delay
- Fact Flight Count

Select the above measures and click the **next** button. The next page is shown in Figure 15.23.

The page presented in Figure 15.23 allows you to include existing dimensions, which have been created in section 15.2. Select all required dimensions and select the **next** button to add additional dimensions (Figure 15.24).

If the fact table includes dimensional attributes which are not provided as dimension tables in the information mart, the dimensions can be set up using this page. Because there are no additional dimensions to be created in the **FactFlight** table, click the **next** button to continue. The next page, shown in Figure 15.25, presents the configuration of the cube for reviewing.

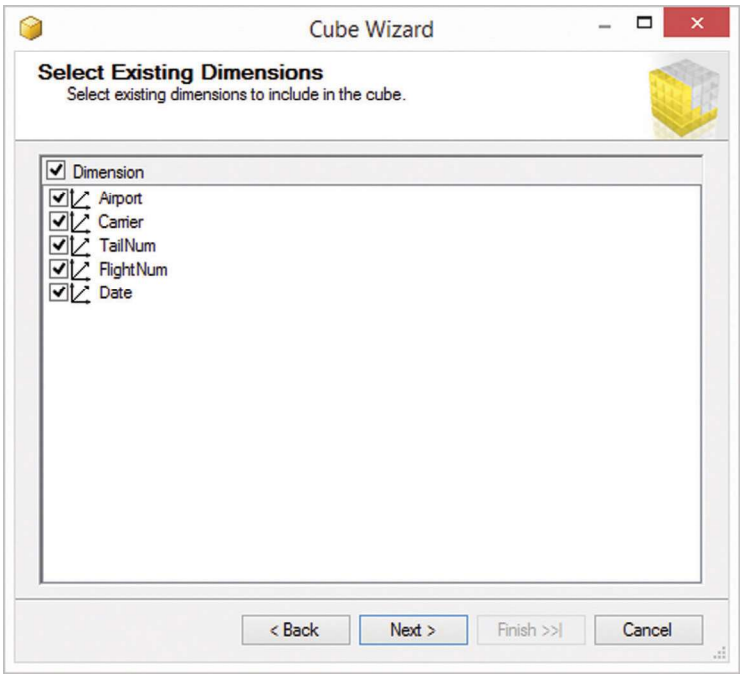

FIGURE 15.23

Select existing dimensions to be included in the cube.

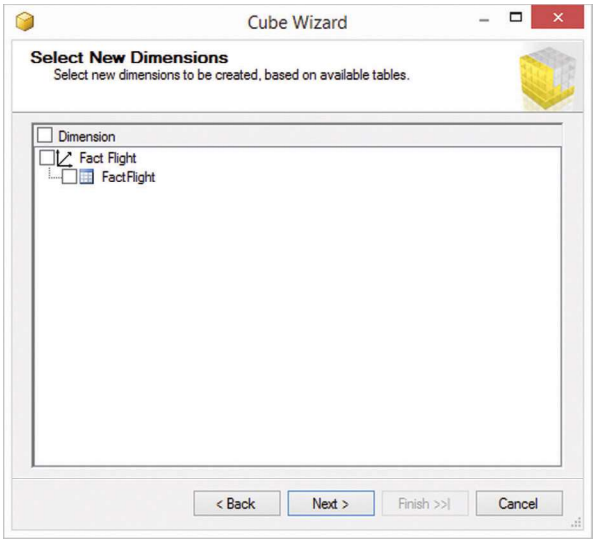

FIGURE 15.24

Select new dimensions to be included in the cube.

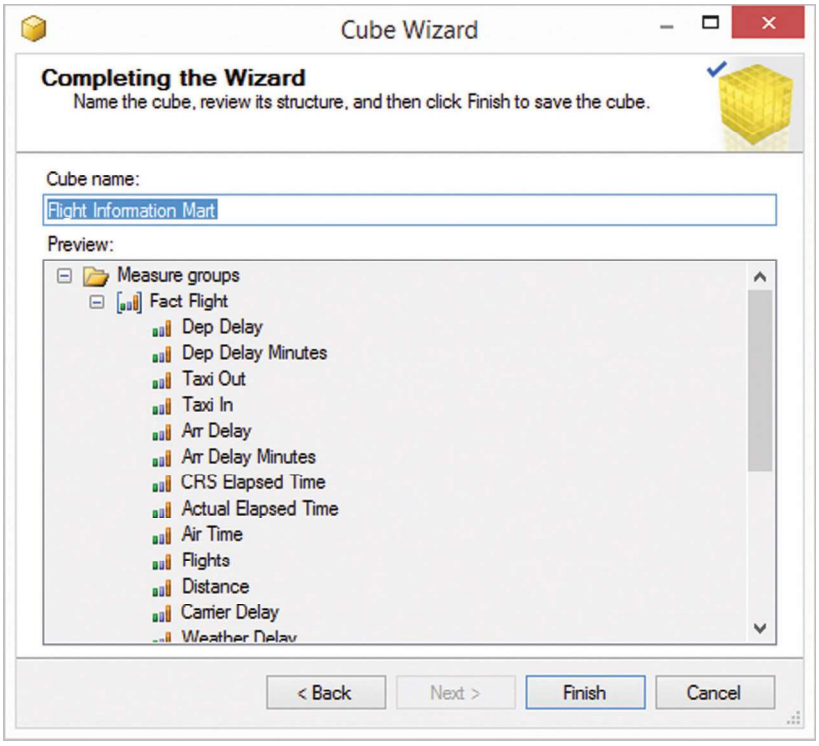

FIGURE 15.25

Completing the cube wizard.

Review the cube configuration presented on the page and provide a meaningful **cube name** for the business user. Select the **finish** button to complete and close the cube wizard. The cube is put into design mode and the logical model of the cube should be shown (Figure 15.26).

The model should include the fact table **FactFlight** and all selected dimension tables. Review the logical model before starting the cube processing and deployment that is required for using it in the front-end tool of your choice.

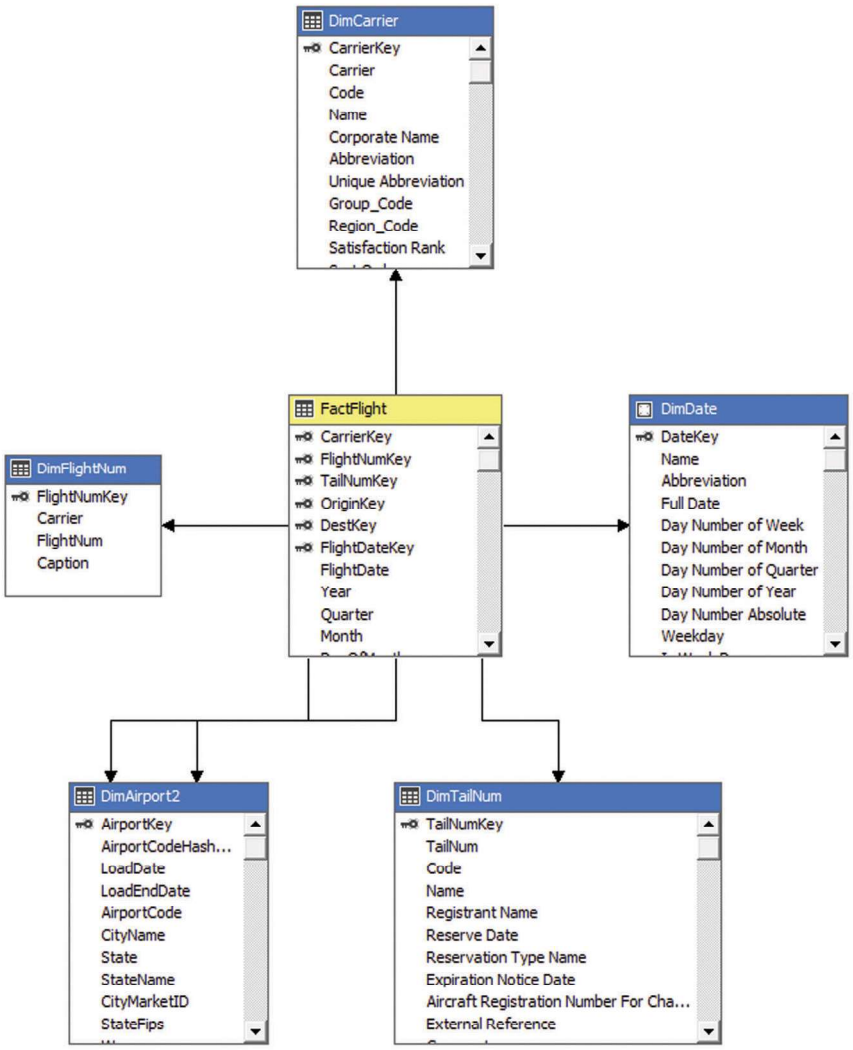

FIGURE 15.26

Logical model of the flight cube.

Note that the logical model for the cube, presented in Figure 15.26, provides only a simplified cube for flight information. There are many dimensions and measures missing from the cube that are required to put this cube into production. However, it serves as an appropriate final example for the information mart created in this book.

15.3.1 PROCESSING THE CUBE

Once the cube has been defined, it needs to be processed and deployed to the Analysis Services database on the database back-end. To process the cube, open the context menu of the cube in the solution explorer and select **process**. Make sure that the cube definition is saved. The dialog shown in Figure 15.27 appears on the screen.

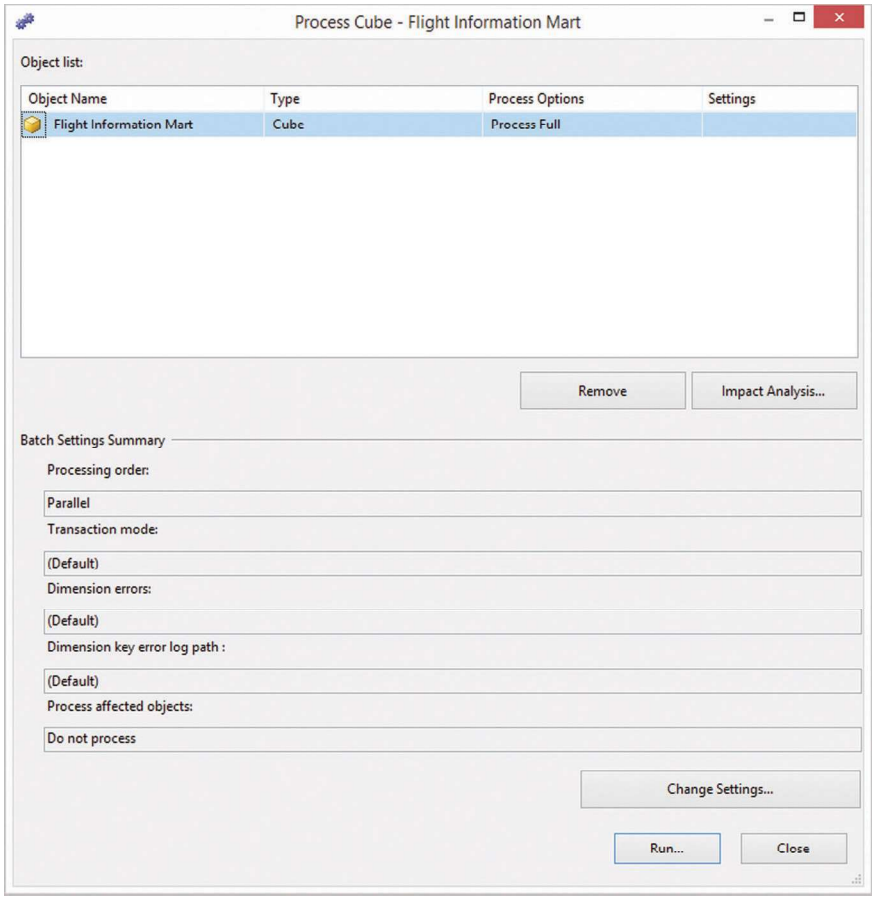

FIGURE 15.27

Process flight information mart cube.

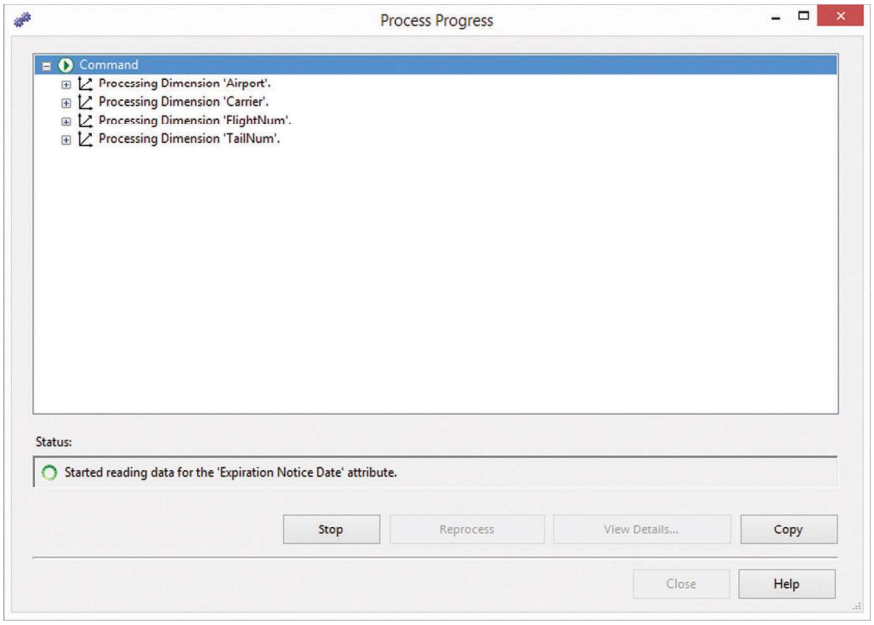

FIGURE 15.28

Process progress.

Click the **run** button in the dialog to initiate the cube processing. The next dialog, presented in Figure 15.28, shows the progress of the cube processing.

Wait until the cube processing completes and close the dialog using the **close** button. The cube is now ready for use.

15.4 ACCESSING THE CUBE

To retrieve aggregated information from the cube, open Microsoft SQL Server Management Studio, connect to the Analysis Services database and create a new MDX query on the database just created. Enter the following MDX statement:

```
SELECT
       CROSSJOIN([Date].[Full Date].members, [Measures].[Distance]) ON COLUMNS,
       [Carrier].[Carrier].[Carrier] ON ROWS
FROM
       [Flight Information Mart]
WHERE
       [Date].[Hierarchy].[Year Number].[2003].[Q4'03].[Oct'03]
```

This statement returns the distance flown in October 2003 by carrier. It is only one way of retrieving information from the cube, giving end-users full freedom to choose the measures and dimensions provided by the cube for ad-hoc access.

REFERENCES

[1] Joy Mundy and Warren Thornthwaite: The Microsoft Data Warehouse Toolkit, Second Edition, pp. 245, 247, 247ff.
[2] https://msdn.microsoft.com/en-us/library/hh212940.aspx.
[3] https://msdn.microsoft.com/en-us/library/ms187597.aspx.
[4] https://msdn.microsoft.com/en-us/library/ms186995.aspx.
[5] https://msdn.microsoft.com/en-us/library/ms178681.aspx.
[6] https://msdn.microsoft.com/en-us/library/ms175662.aspx.
[7] https://msdn.microsoft.com/en-us/library/ms187975.aspx.

Subject Index

D

E

F

G

H

M

N

O

P

R

S

T

U

V

W