SQLBI METHODOLOGY AT WORK

Draft 1.0 – October 1, 2008

Alberto Ferrari (alberto.ferrari@sqlbi.eu)

Marco Russo (marco.russo@sqlbi.eu)

Introduction

After having defined our methodology in the previous paper (Introduction to the SQLBI Methodology, available at www.sqlbi.com), we are going to test it and describe the deeper and technical details by providing a complete solution based on the AdventureWorks database.

The starting points are:

- **AWDataWarehouseRefresh**. It is a package provided by Microsoft as a sample ETL solution to build the adventure works data warehouse. We will rebuild it completely because we are going to make heavy changes to the underlying database structure.
- **Adventure Works DW**. Based on the data warehouse built by the previous package, it is a complete BI solution used to demonstrate several features of SSAS.

We are not going to describe in full detail neither the Adventure Works OLTP database nor the solutions provided by Microsoft. You can find several sources on the web that will document both. Our goal is to describe the problems in the solution and let you follow our thoughts in the process of rebuilding it. Clearly, we do not intend to criticize Microsoft's work, our intent is to show how to build a full solution and Adventures work is a perfect candidate for doing so.

Even if it is only a demo, we think it is very important to follow it in details. As consultants, we often work for customers who already have a working BI solution but want to improve its design to gain better performances and/or a better architecture to let the solution grow gracefully in the future. The considerations and thoughts that we do in those cases are exactly reproduced in this demo.

The steps will be:

- Analyze the SSIS package that builds AdventureWorksDW and modify it in order to make it easier to maintain and clearer to understand. This will lead us to a better comprehension of the steps needed to transform the OLTP database into the Kimball data marts.
- Analyze the data source view of the SSAS solution to check for computation hidden in the named queries and/or in the computed fields of the data source view. We will remove all those computations from the data source view and move them to the data warehouse level.
- When we will have a clear understanding of the ETL process, we will be ready to build the data warehouse level. We will produce a model of the data warehouse with a modeling tool and we will document all the steps needed to transform the OLTP database in the data warehouse.
- Deeply review all the data flow tasks in the SSIS package

Our goal is to start from the AdventureWorks OLTP database and build a complete BI solution using the techniques described up to now. The final solution will be much similar to that of Microsoft but, from an architectural point of view, it will show the power of our approach to the development of a BI solution.

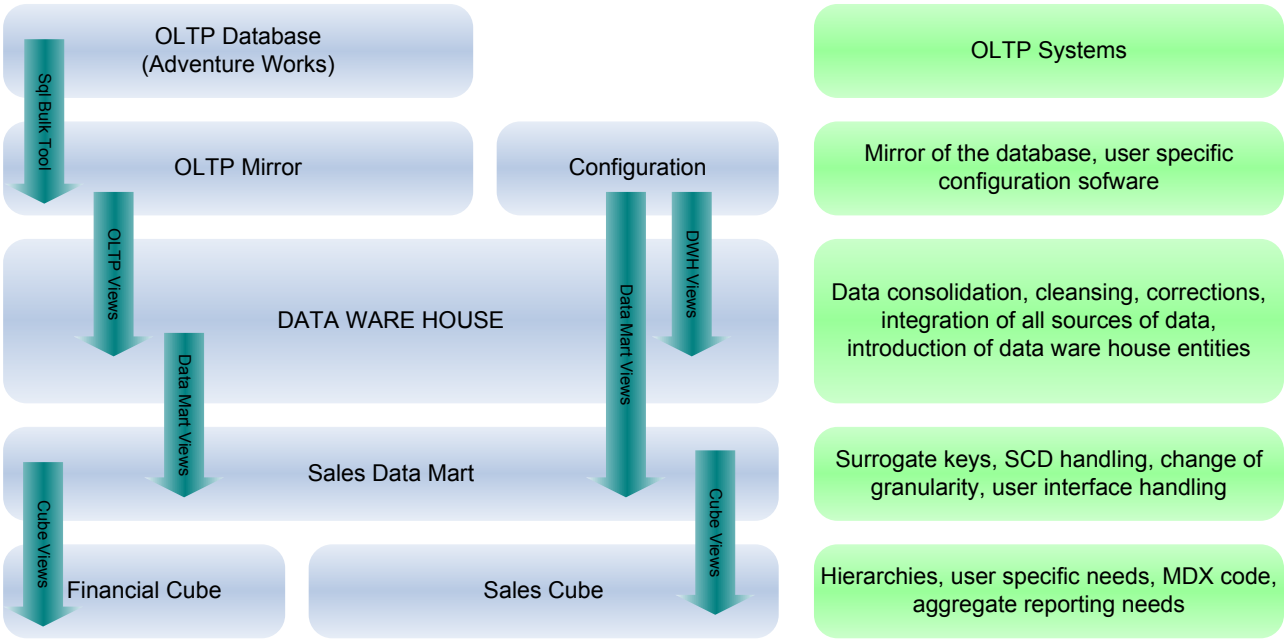
We will not describe the whole solution in the book; sources are available on the web and we have extensively commented them. What we think is important to get here are the principal concepts of the data warehouse architecture. Moreover, as the book is mainly focused on SSAS solutions, we will not spend too many words on the SSIS package implementation. We will bring to your attention some hint and tips, when we feel that they might be useful, but the focus will be on the architecture, not on the specific implementation of the ETL with SSIS.

A note for our readers: you can read the whole chapter without ever opening BIDS to see the solution or you can look at the solution while reading the chapter. The choice is up to you and depends on what you

want to learn from this chapter. If you are strong on SSIS and SSAS and you are mainly interested in the concepts of the application of our model to a real world situation, you can simply go on reading the chapter. If, on the other side, you want to dive into technical details of the solution and see some hint and tips about SSIS and SSAS, then the best will be to open the demo projects and to study it while reading the chapter.

THE WHOLE PICTURE

Before diving into the technical stuff of the solution development, let us review the final solution, according to our methodology:



IMG (0132): Complete Adventure Works Solution

Let us comment the picture in more detail:

- We start with the Adventure Works database and mirror it copying only the useful tables and columns. Our goals are speed and ease of management, no business logic should be inserted at this step. The step will be carried on with a free tool to mirror databases.
- We build a Configuration database that holds all the configuration data for the final solution. If this were a real world solution, we would have to write some user interface with the configuration database to let the end user interact with it.
- In the data warehouse, we consolidate data, cleanse it and create a complete view of all the information that will be used for subsequent reporting and analysis needs. The step between the OLTP mirror and the data warehouse is the hardest, at least from a conceptual point of view. We will create new entities thinking at the analysis of data and forgetting the operational point of view of the OLTP database. This step will be composed of views that gather data from the OLTP and feed an SSIS package that makes all the ETL operations.
- The data in the data warehouse will move into the data marts. In the data marts the point of view changes. We are no more concerned with the structure of data but we will focus on what the user wants to see. A date field might become a string, if the user wants to see it in a fancy way. The data warehouse maintains the structural view of data while the data mart will mix the logical structure with user specific requirements. Moreover, we adopt Kimball's methodology and define facts,

dimensions, bridges, slowly changing dimensions and all the features needed for a good data mart. A set of views will feed an SSIS package that builds the data marts. The views might be complex, as they might need to aggregate data, join tables and – in general – follow the data warehouse structure to fully de-normalize it and produce data marts.

- The final step will move data from the data marts in the SSAS solution that will hold the cubes. Technical requirements might lead us to define different views for the same data, to change some esthetic aspects of a single column, to rename fields according to the user requirements. This step will be composed solely of views, as no processing should occur; the views will (and need to) be very simple and fast.

Since we do not want to show only the final solution, we will write the chapter trying to make you following all the work and the considerations we did in order to make Adventure Works adapt to our vision of the data warehouse architecture.

The flow of the analysis will be different from the steps presented here. You must consider that no OLTP mirror database can be created until we know exactly which columns are needed from the OLTP database. Therefore, even if the OLTP mirror database comes first, it will be the last one in the development cycle. The flow of knowledge about the data warehouse is different from that of the data. First, we need to build and describe the data warehouse. Then, we will be able to load it and let it bring data to the data marts.

Moreover, even before being able to write down the data warehouse structure, we will spend some time analyzing the Adventure Works solution that is available on the web in order to use it as the starting point of our analysis process.

THE ANALYSIS PROCESS

The first step of our re-engineering of the AdventureWorks solution is to study the original one and detect all the potential errors in its structure.

The solution is composed of two different items: an ETL package that loads the data marts with data coming from the OLTP database and an SSAS solution that effectively creates the cube structures.

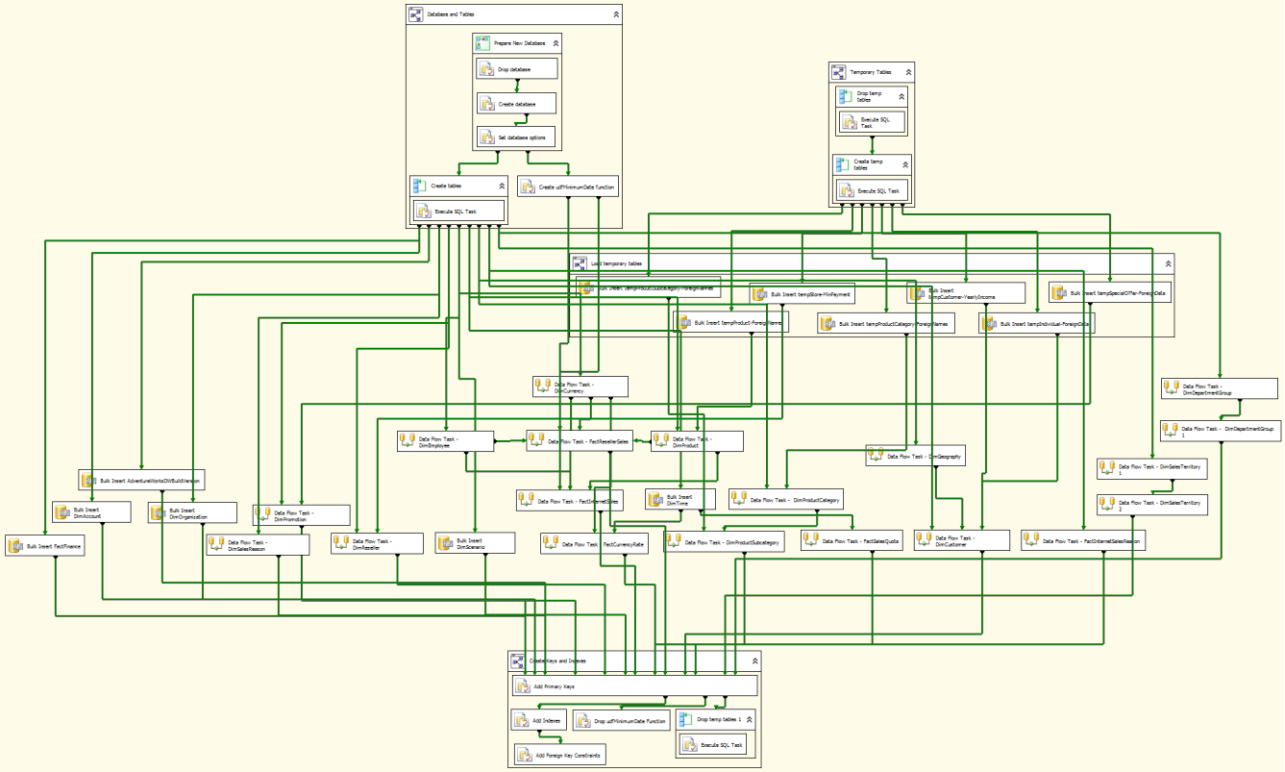
- The SSIS package that creates the AdventureWorksDW database is located under the program files folder in "Microsoft SQL Server\90\Samples\Integration Services\Package Samples\AWDataWarehouseRefresh".
- The SSAS solution is located under the program files folder in "Microsoft SQL Server\90\Tools\Samples\AdventureWorks Analysis Services Project\enterprise". We will use the enterprise solution as it is the most complete. However, we will not rebuild the whole SSAS solution as it contains data mining projects and several other features that are out of the scope of this example.

We will review both of them and will redefine their structure according to our methodology. The analysis process will require us to dive into both the ETL and the SSAS solution to gain knowledge on how the facts and dimensions are constructed starting from the OLTP.

ANALYSIS OF THE ETL PACKAGE

First, we open the ETL package to gain a good knowledge of how the data coming from the OLTP database is handled in order to create the data marts. As this is an example, we would expect it to be easy to read. Many people will start learning SSIS programming with that package and we believed that a good effort has been spent in making the package a good example. Opening the package has been a real surprise, at least from the clearness point of view.

In the following picture, you can see the original package:



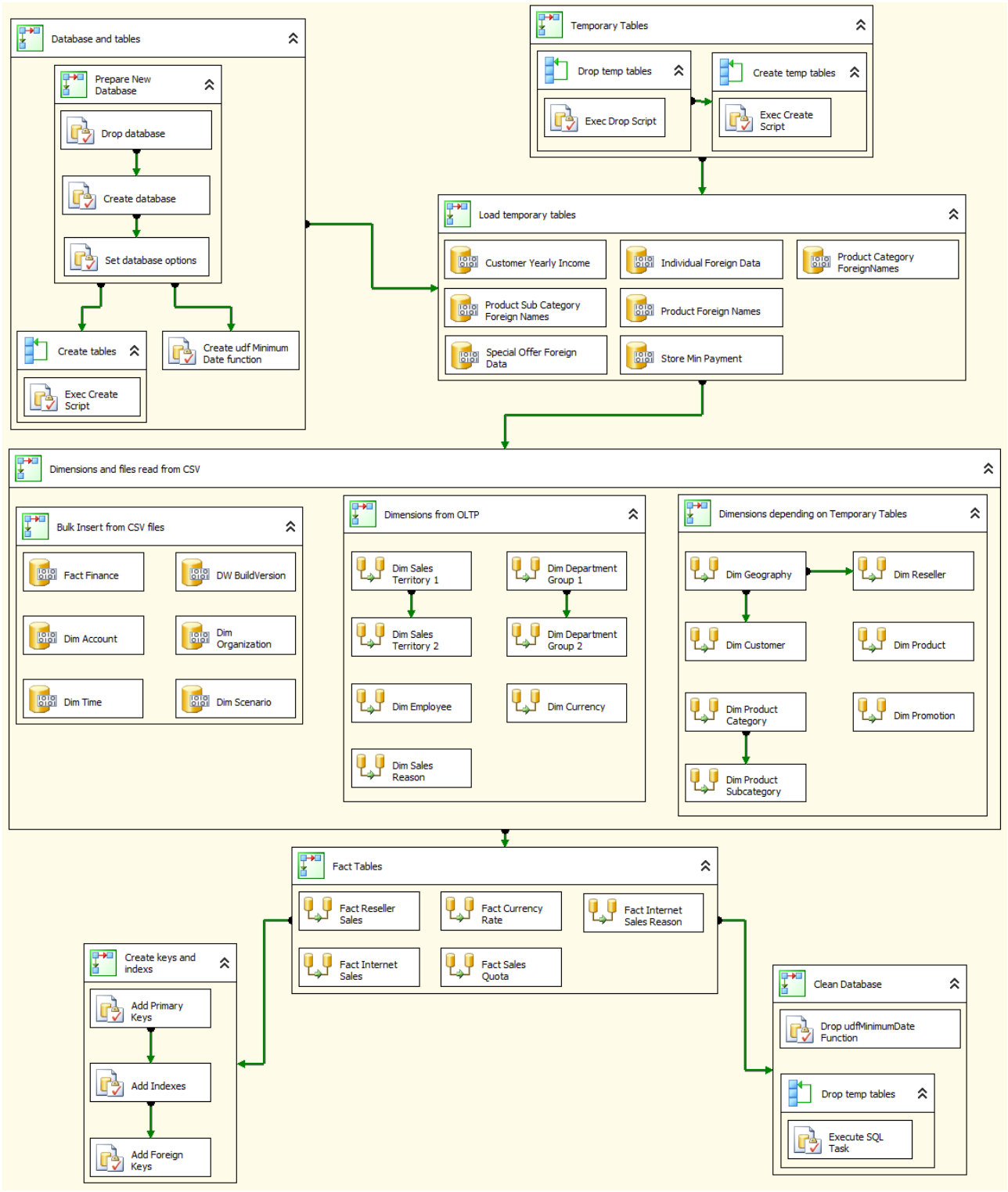
IMG (0116): Original ADW Package

We normally use Visual Studio on a 30” display with a resolution of 2560x1600 pixels. Even if this package fills the whole monitor, we still do not have a clear view of what it does. There are so many arrows intersecting each other that it is quite impossible to understand the real dependencies.

Therefore, the first work is that of making some esthetic improvements on the package, in order to make it easier to read on a normal display. We started with these modifications:

- Introduce sequence containers to group related data flow tasks and remove almost all the dependency arrows between simple tasks. This might reduce somewhat the degree of parallelism in the package but will greatly enhance readability. The correct balance between readability and performances is always a difficult decision to take but, because we are making a demonstration package, we will prefer readability against performances.
- Remove useless information from the names of the single data flow tasks to reduce their size. In our opinion it is useless to start the name of a data flow task with “data flow task”, it should be clear to any SSIS programmer what a task is, just by looking at its icon.
- Align all items and make their size reasonable. Tasks with varying size lead to bad alignment and make the whole package difficult to read. Again, this is only esthetic but it is very important.

We made a copy of the original package and named it “ADWDRefresh (Esthetic Changes)”. You can find it in the “Adv – ETL” project of the final solution, with all these modification already made. The package, after those updates, looks much better:



IMG (0117): ADW Package after esthetic updates

We can appreciate that esthetic is very important in SSIS packages, as it is normally in source code. If we do not pay attention to clearness, we will end up with a very difficult package to read. With those modifications, it is now very easy to understand the tasks that compose the package. Moreover, grouping related tasks in sequence containers, we greatly reduced the number of arrows in the whole package. Now that we have fewer relationships, we gained a much clearer structure and we can start to analyze the ETL phase.

There are two sources of data:

- CSV files: these files contain data that is not present in the OLTP database but is needed for the final data marts. All the financial information and structure is kept there. We will remove the flat files and move those data in the configuration database, in order to understand the data better.
- OLTP database: it is the well-known AdventureWorks database. Several complex queries retrieve the data from the OLTP database and move them in the data marts. Some of these queries are very complex and we will make them easier using the data warehouse technique.

Before going down in further updates, we can make some considerations about the structure of the ETL:

- The "Database and tables" container will destroy the existing data warehouse database and then will rebuild it, executing all the scripts with the name "CreateTable-*.SQL". Moreover, it will create an UDF function in the data warehouse. The package will destroy this function upon termination. We do not like this approach because the destruction and rebuild of the database is in the same package that uses it. If something goes wrong during the package debugging, you will end up with an incomplete database and the full package will fail to validate. For this reason, we will create a separate package that creates the database from scratch. If anything goes wrong, you can safely execute it to rebuild the empty database.
- The "Temporary Tables" container will drop and recreate several temporary tables directly into the OLTP database. These temporary tables will contain data coming from the CSV files (see "Load temporary tables" container). They represent external data coming into the data warehouse. In our structure there is no place at all for this kind of operation. The OLTP is read-only and we will eventually write CSV files in the configuration database or in the staging one, depending on their usage. We have to note that – from an architectural point of view – there is no difference between a "temporary table" and an "OLTP table". We have no place for temporary tables in our structure if they contain useful data that will definitely go into the data warehouse. This information must go into the configuration database and must have a name and a structure so that any package that needs them can safely use them.
- The temporary tables contains two kinds of data:
    - Translation data, i.e. names of some dimensional attributes in different languages
    - External data: i.e. information not found in the OLTP database

  From the package, it is not evident which data is used for translation and which is used for loading new data into the data warehouse. We will make the structure clearer using the configuration database schemas.
- The "Bulk Insert from CSV files" creates several dimensions that come from the external world (CSV files in the example). This container does not have a good structure, because it loads data directly from the external world into the data warehouse. We will store this information into the configuration database and will provide some loading mechanism to build them.
- The "Dim Time" task does a bulk load of the time dimension. This is wrong. We should create the time dimension inside the package with a clear and documented logic. Otherwise, it would be very hard to modify it, for example, in order to add some attributes to the dimension.
- The fact table queries contain several joins between the OLTP database and the data warehouse dimensions. The package will execute these joins before the creation of primary keys on the dimension tables. This will lead to poor performances. Moreover, in a well-designed SSIS package there should be no join between different databases, as it will break the "Connection driven" structure of the package.
- The final "Clean database" container is useless. It will clear the temporary tables from the data warehouse and the OLTP but, as we said before, we will not create any temporary table in the data warehouse nor in the OLTP database.

There are several other considerations to do about this package but we will make them when we will start modifying the single data flow tasks. For now, we want to focus on the structural point of view and the information we can gather from here are summarized as follows:

- Bulk Insert from CSV files
    - Fact_Finance, Dim_Account, Dim_Organization, Dim_Time, Dim_Scenario come from CSV files. Those CSV files do not contain any natural key or any defined structure. We will have to rebuild them in the configuration database or in the data warehouse.
- Dimensions from OLTP
    - Sales Territory contains a geographic structure built from the original OLTP sales territory table, enriched with the country information from the Person.CountryRegion table. It is composed of two data flows because the first one inserts data from the OLTP and the other creates the "unknown" record useful in all dimensions.
    - The department dimension from the HumanResources.Departments OLTP table. The first data flow creates a generic node while the second one loads data from the OLTP table.
    - Dim Sales Reasons is constructed from the Sales.SalesReasons OLTP table.
    - Dim Employee is much more complex. It gathers information from several tables in the OLTP database to get the whole SCD story of employees. It is not useful to analyze in deep detail the query. The important point here is that an SCD is built based on OLTP data. In the data warehouse, we will need to keep all the information needed to rebuild the same SCD.
- Dimensions depending on temporary tables
    - Dim Geography is built using seven different tables (the query has to follow complex relationships in the OLTP database) and one temporary table that holds the foreign names of the country regions.
    - Dim Promotion gathers its data from Sales.SpecialOffer and gets the foreign names for the special offers from a temporary table.
    - Reseller, Product, Customer are all very complex queries that take information from the OLTP system.

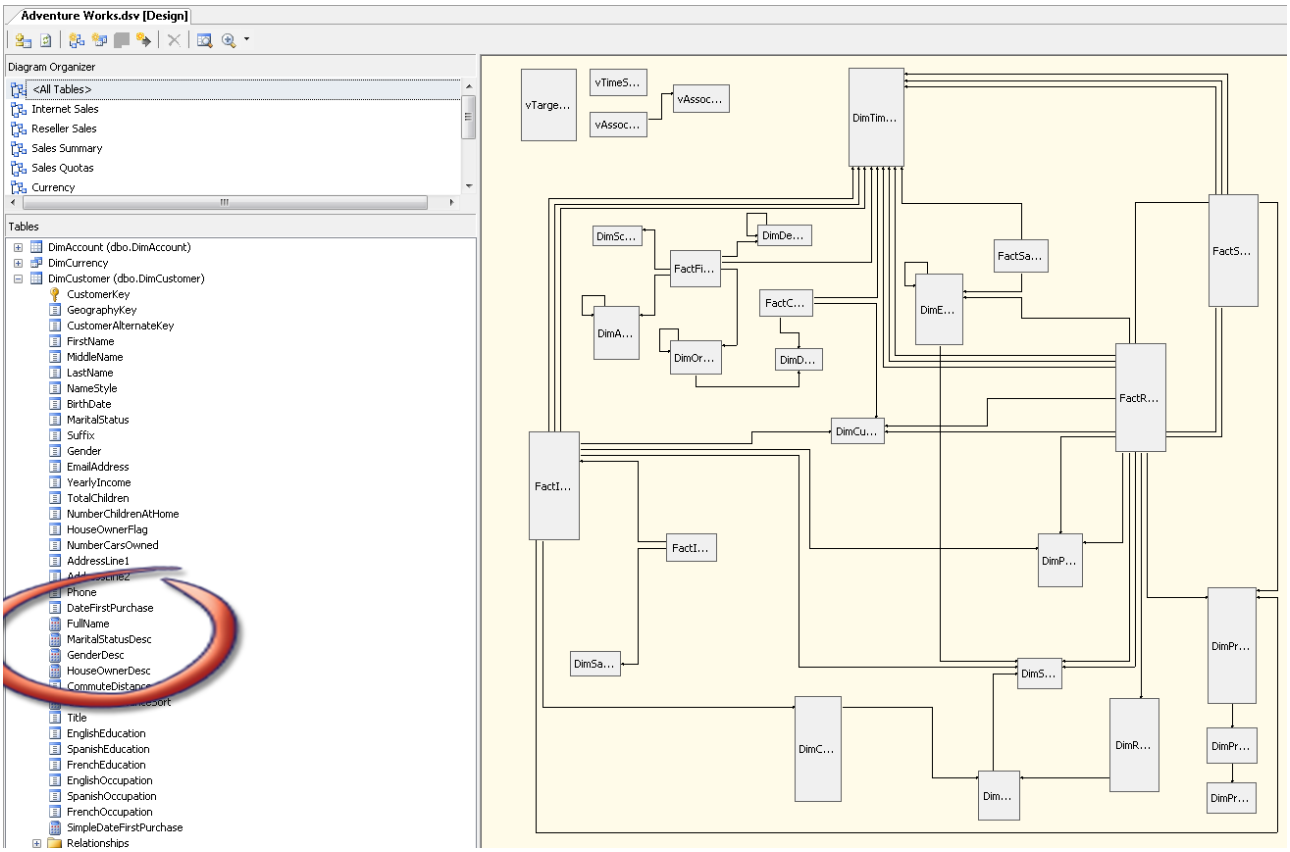
We do not want to bore the reader with too many details about these queries. We have already made the hard work of understanding each query and define the relationships between dimensions. The real point to get here is that if we want to understand what are the relationship among dimensions and OLTP tables, we need to open each data flow task and spend a few hours (or days?) to document everything.

If the BI analyst of Adventure Works had written a good documentation, then our life would be much easier but we would still have to handle the classical problem of "out of date" documentation. In the IT world we are all well aware that if documentation is hand-written, it will inevitably be out of date in a few months after the first release.

The first lesson here is that SSIS hides documentation in every SQL statement used into sources.

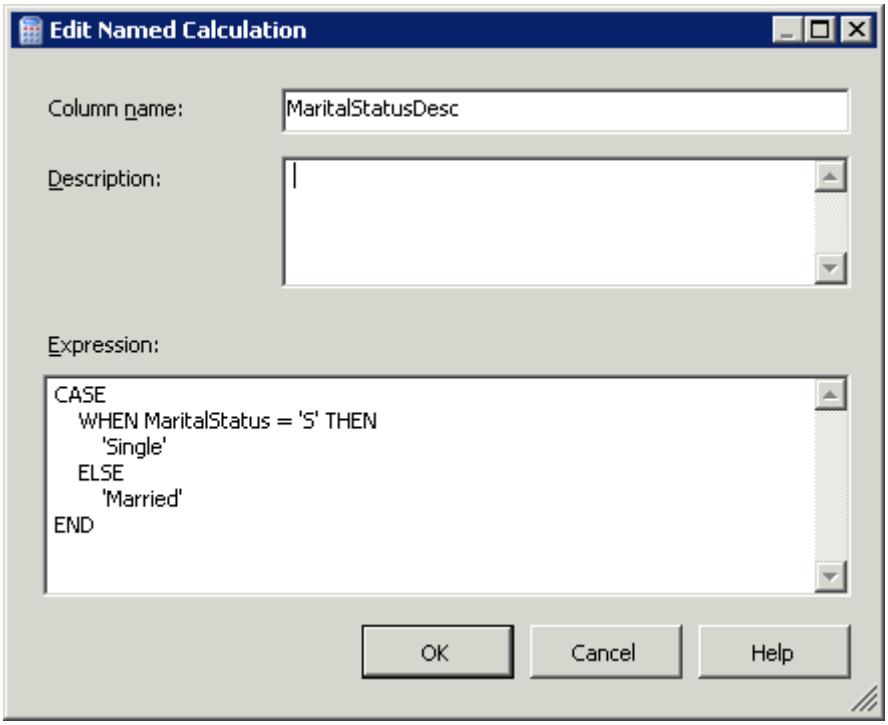
Analysis of SSAS data source views

It is now time to dive into the data source view of the SSAS solution in search of hidden computations. It is not necessary to analyze in deep detail the SSAS solution, because we are just searching for parts of the ETL process hidden in the data source view. It is sufficient to open the project data source view and analyze all the tables searching for computed fields. BIDS makes the process easier as it shows computed fields with a different icon. In the picture, you can see some of the computed fields in the Dim Customer table:



IMG (0119): Computation hidden in the data source view

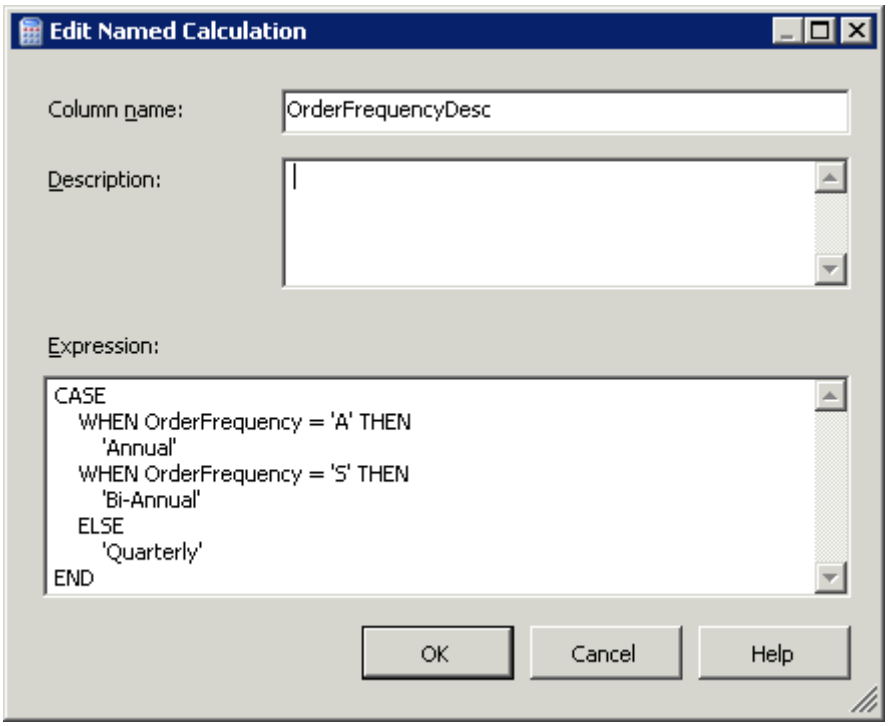
Opening up, for example, the MaritalStatusDesc field, we will find this:



IMG (0120): Marital Status Description

We do not want to hide this translation of Single/Married into the SSAS project; we want to make it explicit in the data warehouse providing a remap table that will handle it. Please note that, from the functional point of view, we are not changing anything. We are only moving computations and concepts where they belong to, trying to make all the computation explicit.

There are many other fields like this and they have more or less the same structure. Among all, a very interesting one is in the Dim Reseller table. If we look at the OrderFrequency we will find:



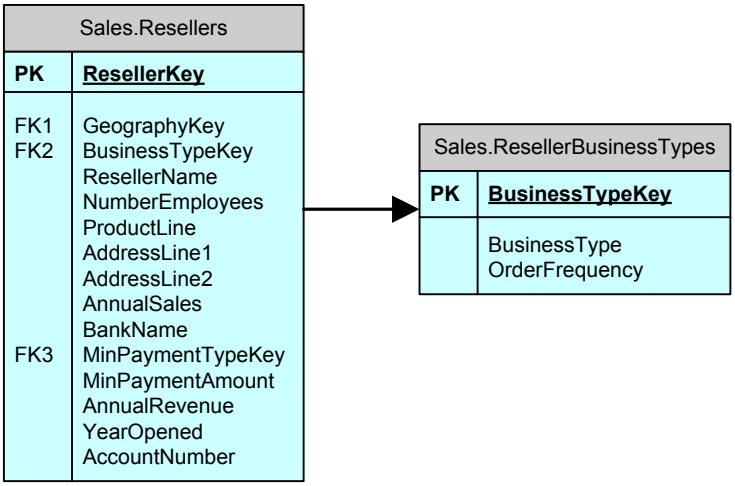
IMG (0121): Order Frequency Description

Therefore, it seems like all other remap codes. However, when we look at the source query in the SSIS package that produces the OrderFrequency code, we will find this interesting computation:

```
...
CASE Survey.ref.value (
        'declare default element namespace
        "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey";
        (BusinessType)',
        'varchar(10)')
    WHEN 'BM' THEN 'Value Added Reseller' --Bike Manufacturer
    WHEN 'BS' THEN 'Specialty Bike Shop'
    WHEN 'OS' THEN 'Warehouse'
END AS BusinessType,
...
CASE Survey.ref.value (
        'declare default element namespace
        "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey";
        (BusinessType)',
        'varchar(10)')
    WHEN 'BM' THEN 'S' --Semi-Annual
    WHEN 'BS' THEN 'A' --Annual
    WHEN 'OS' THEN 'Q' --Quarterly
END AS OrderFrequency,
...
```

The Order Frequency is computed using exactly the same XML field used to compute the BusinessType code. Therefore, the interesting fact here is that there is a strict dependency between the business type and the order frequency. Both are computed as codes and then remapped to description but – in fact – they represent two different names for the same concept. This is a structural fact that is not explicit. Therefore, we will make it clearer building a BusinessType table that will provide both the business type description and the order frequency one.

As we can see, the process of search for hidden computation let us discover structural information deeply hidden in the solution. Worse of all is the fact that the project itself does not provide a clear understanding of those structural relationships and it is failing to self document. In order to better understand what we mean, the following picture shows an extract of the final data warehouse structure, where the relationship between Order Frequency, Business Type and Reseller is made evident by the database structure:



IMG (0122): Business Type Table

As we have seen, the process of rebuilding the metadata of the data warehouse proceeds with the analysis of both the SSAS data source view and the SSIS packages that does the ETL. Normally we use the data source view to build a list of "interesting topics" to analyze and the SSIS package to get deeper details about how these topics are computed.

If we had more than one cube, we would spend much more time doing this analysis because each cube might hide different computations based on the same column and all these hidden computation must be made evident in a clean data warehouse design. The same applies for any report generated with Reporting Services of any query that produces analysis for the user. All these queries should gather their information from one single repository, in order to have a coherent environment that produces the user results.

It is not important to go in deeper detail with this specific solution, what we want to stress is the fact that no computation should ever be hidden in the data source view. The metadata structure of the data warehouse should be clear at the database level. Moreover, the data source view is an interface between the database and the OLAP cubes and should never contain any ETL step.

USAGE OF SCHEMAS

Let us recall briefly the usage of schemas in the different databases:

- **OLTP Mirror**: it will contain an OLTP schema for the table mirrored from the OLTP and a schema for each subject area of the data warehouse, where we put the views that gather data from the OLTP tables and produce entities in the subject area. The "Customers" view that will create the "Sales.Customers" entity is held in the "Sales" schema that contains the "Sales" subject area.
- **Data warehouse**: it will contain a schema for each subject area where we put entities and a schema for each subsystem that will be generated from the data warehouse. Subsystems are data marts but also reports, analytical tools or any other subsystem that gathers data from the data warehouse. The "Dim_Customers" view that generates the "Customer" dimension in the sales cube of the Sales data mart will be held in the "SalesDataMart" schema, which contains all the views required from the Sales data mart.
- **Data Mart**: it is a single database that will contain more than one data mart. There will be a schema for each data mart, containing the tables with facts and dimensions that are pertinent to that data mart. Moreover, it will contain a schema for each cube that is generated from the data marts. The Customers view that extract customer attributes from the Sales.Dim_Customer dimension will be contained in the "CubeSales" schema.

- **Configuration**: the configuration database is a bit different from all the others, because it interacts with all the different steps of the ETL. It provides information to both the data warehouse, the data marts and potentially to the cube generation. Moreover, it is a common practice to put the connection strings in specific tables of the configuration database. The rule of thumb is that the only "hardwired" connection should be to the configuration database. All other information should be gathered from there. For this reason, the configuration database contains some schemas that are inherent to subject areas of the data warehouse (those are the "main" subject areas in our model) and one schema for each ETL step that will get data from there. An example may be useful: if during the ETL of the data warehouse from the OLTP we need the tax percentage to compute the amount of taxes for one sale, then we will create a table in the configuration database named "Sales.Parameters" that will contain all the different parameters of the sales subject area. Then, we will create a view "DataWareHouse.Sales_OrderParameters" that will take values from the "Sales.Parameters" table and provide them to the data warehouse ETL, in the subject area Sales for the computation of orders. Even if it might seems difficult at first, this naming technique lead to a very easy comprehension of all the different uses of the configuration database, as we will show later.

More generally, each database contains the schemas used by itself and the schemas used by the next step in the ETL phase. It gathers data from the previous steps of ETL from views that have the same name and schema of its entities; then, it produces views that have same name and schema of the next step.

This usage of schemas and names will lead to a very clean architecture that makes it easy to follow the flow of data.

DATA WAREHOUSE DATABASE

The AdventureWorks solution has been designed as a pure Kimball solution. It starts from the OLTP and generates the dimensional structure needed for the cube. We want to build an Inmon database that will store all the information for the data warehouse and then generate the dimensional structure of the cube starting from the data warehouse.

In the data warehouse, there are no facts and no dimensions. The data warehouse level is based on the good old entity/relationship model. Moreover, the data warehouse does not contain surrogate keys, which will be added at the data mart level.

Let us start with the definition of schemas. The schemas in the data warehouse level are:

- **Financial**: it holds all the entities that are related to the financial aspect of the data warehouse like accounts, scenarios and so on.
- **Common**: this schema contains common used entities like date, currencies, etc.
- **Sales**: in this schema we will include all the entities related to sales like customers, resellers, employees and territory
- **Products**: this schema holds the definition of products.

A problem we have to face is about the multi-language support that is needed by the solution. If we were going to introduce multi language support at the data warehouse level, we would end up with several problems:

- Each description in a different language would waste space in the data warehouse.
- The addition of a new language would require a complex operation in order to fill in all the description for the old rows. Moreover, language support is an "interface" topic. It does not regard the structural design of the data warehouse.

- Any change in the configuration tables with the translation would require an update of the data warehouse.

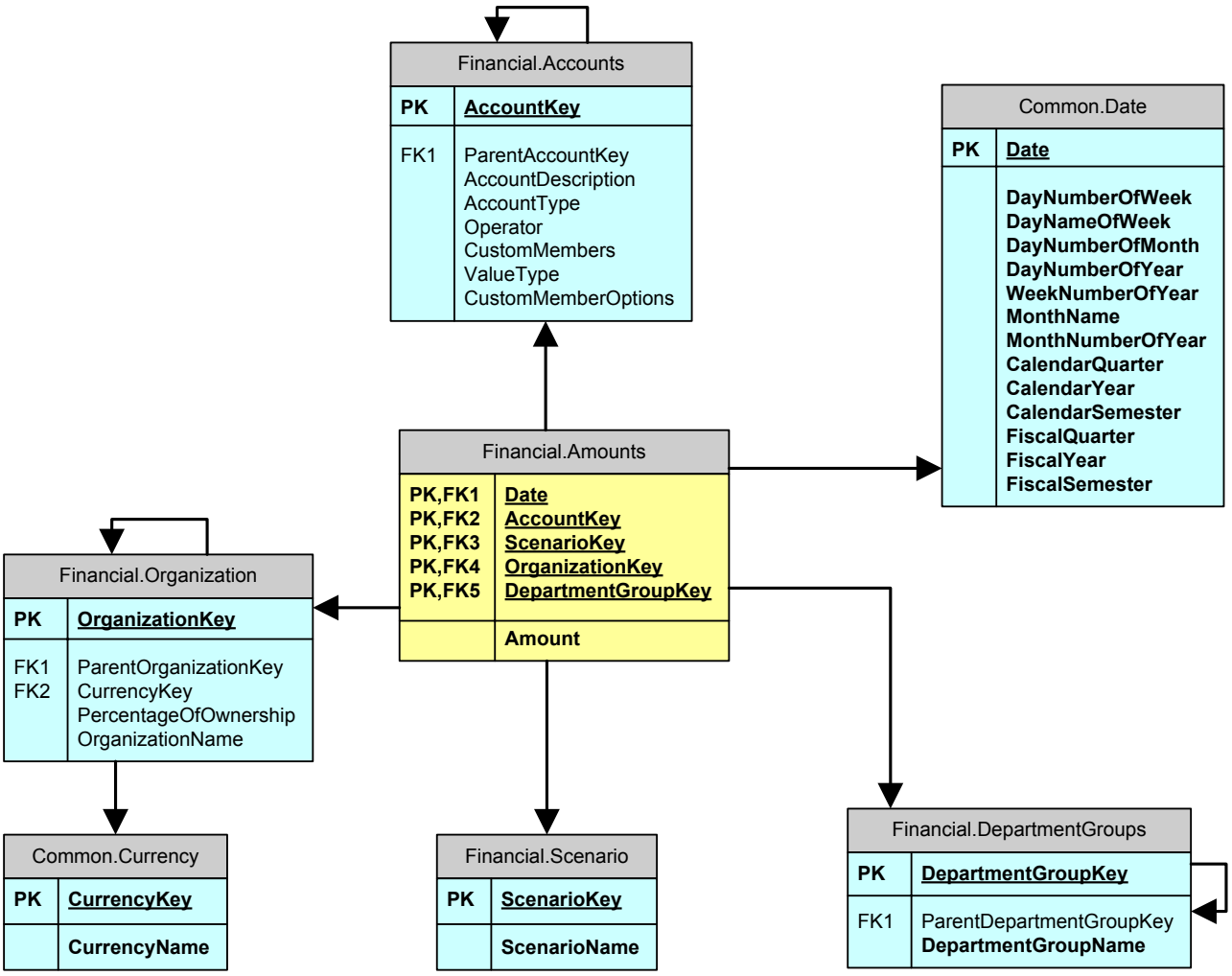
For these reasons, the data warehouse should only contain English description (or your native language, if the DWH does not need localization). Multi language support will be added at the data mart level, where we will need to manage user interface topics.

Moreover, the data warehouse will contain several small tables that will provide the data mart will full description of all the technical fields. Think, for example, to the MaritalStatus flag in the Employee's dimension. Even if we can store the flag in a single character field, all the data marts will provide a more complete description, mapping "S" to "Single" and "M" to "Married". For this reason, we are introducing the mapping tables that will then be used by the data mart views. All these tables will be part of the "Remap" schema and will assure us that we are making a consistent usage of descriptions throughout the whole data warehouse.

Let us briefly review the various subject areas at the data warehouse level.

Financial Subject Area

The financial subject area maintains entity related to the financial life of AdventureWorks. Most of the tables are not loaded from the OLTP database; they come into the data warehouse from CSV files. The nice aspect of having a data warehouse level is that we handle external data in a graceful way. Since the data is into the data warehouse, we do not need to worry about "where" the data comes, we can use it in a coherent way as long as it resides in the data warehouse.

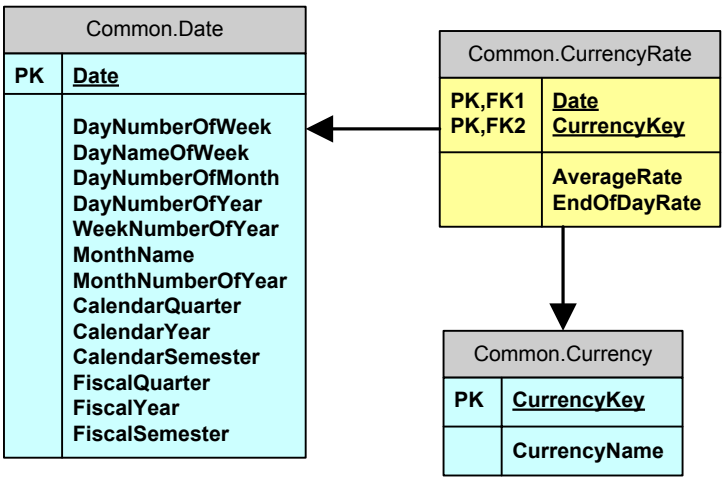


IMG (0126): Financial Subject Area

Even if this structure resembles that of a star schema, it is not. In the data warehouse design we do not need to create star schemas, we are free to design the relational model to fit our needs of clarity and completeness. The star schemas will be derived from this structure and will reside in the data mart level.

Common Subject Area

The "Common" subject area is used to group the structures that are commonly used in the data warehouse. There is nothing special to say about this subject area, it exists in almost any data warehouse we have ever designed and contains, at least, the "Date" entity.

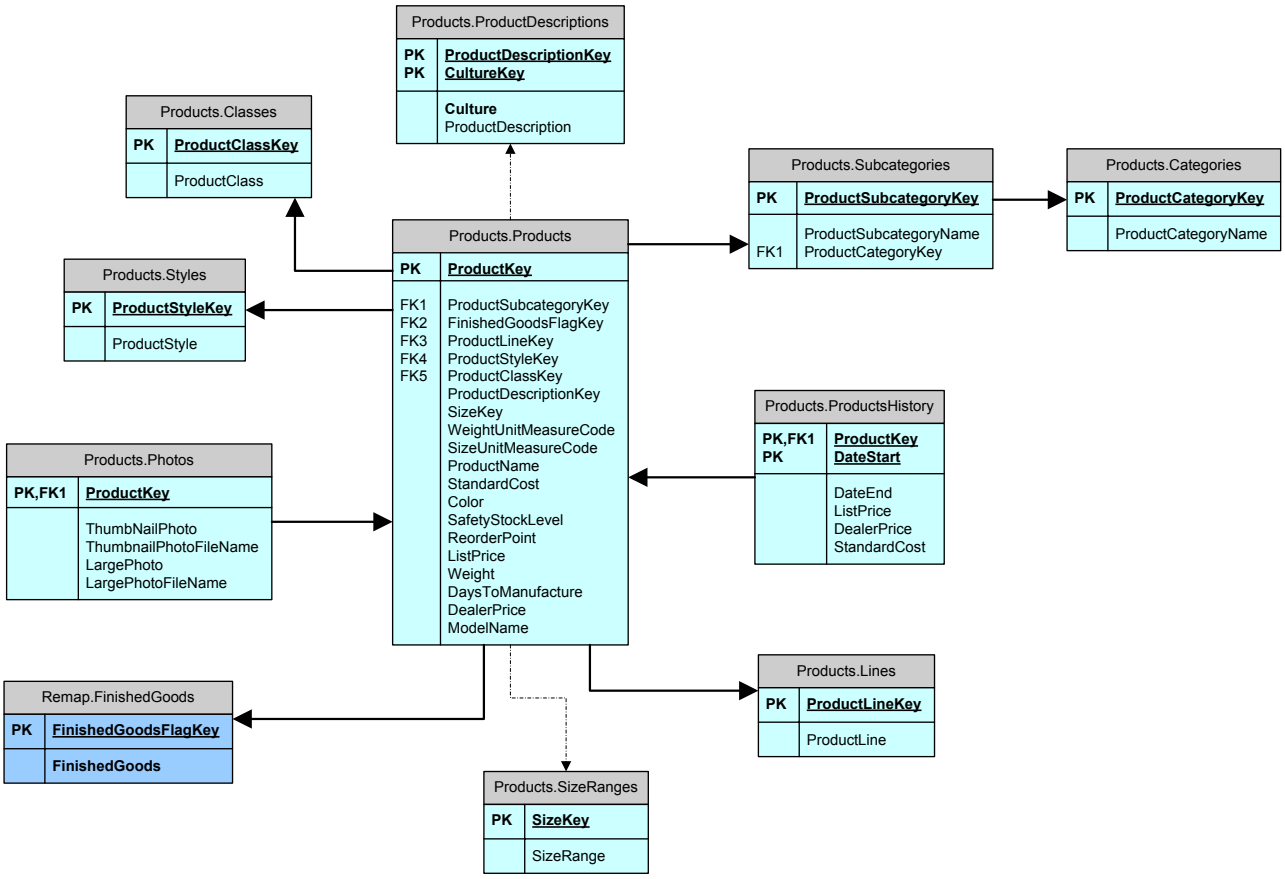


IMG (0127): Common Subject Area

In this case, we added the currency and currency rate entities to the common subject areas. Even if the currency rate is used only in the sales cube, we want to look forward and think that, in the future, several other cubes might use the currency rate entity.

Products Subject Area

This is a quite complex subject area. We use it to describe what a product is.



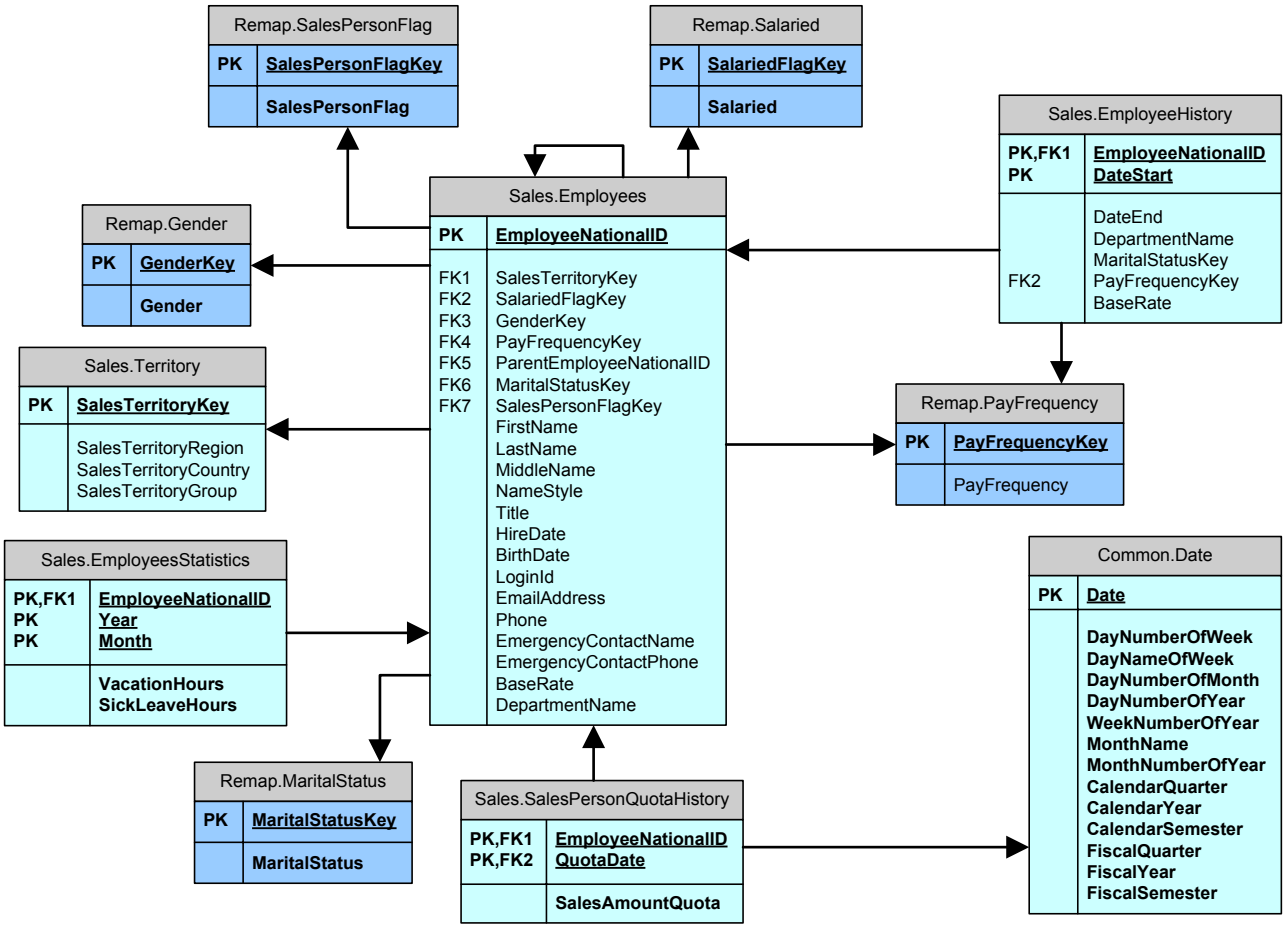
IMG (0128): Products Subject Area

There are several aspects to note about this subject area:

- **Several entities exist here even if they have no corresponding table in the OLTP database**. Look for example at Products.Styles and Product.Classes. These tables do not exist in the OLTP database, because the description of the codes was hardwired in the data source view of the SSAS solution. The data warehouse takes care of this aspect by creating the table and letting us navigate easily the real structure of the data warehouse.
- **Some relations are not enforced with foreign keys**. The link between Products and SizeRanges, for example, is shown with a dotted arrow. This means that there is no "real" relationship between products and size ranges. This kind of relationship should be represented with a LEFT OUTER JOIN in any query because it is acceptable to have a ProductSize value that has no corresponding value in the SizeRanges table. Even if it might be strange in a relational model, these kind of relationships are indeed very common in the data warehouse solution.
- **There is no need to perform a full denormalization process**. Each product has a subcategory that – in turn – has its category. If we were in the data mart world we would have defined category and subcategory at the product level but – in the data warehouse world – there is plenty of space for normalization, if it helps in the structure. The goal of the data warehouse is not to provide a fully de-normalized structure that will be analyzed by automated tools, the goal is to provide the foundation for any kind of analysis and, for this reason, only a slight denormalization process happens here.
- **Products is not an SCD**. It could not be an SCD simply because it is not a dimension. In the data warehouse world, there is no space for a concept like "dimension". Everything is an entity. For this reason, we keep the history of products in a separate table and we will generate an SCD from these two tables only if it will be necessary. Moreover, we might have a cube that needs SCD handling and one which does not. The data warehouse needs to be able to feed both.

SALES SUBJECT AREA

The sales subject area is – naturally – the most complex one. We divided it into two pictures because the employees part of it is quite complex by itself.

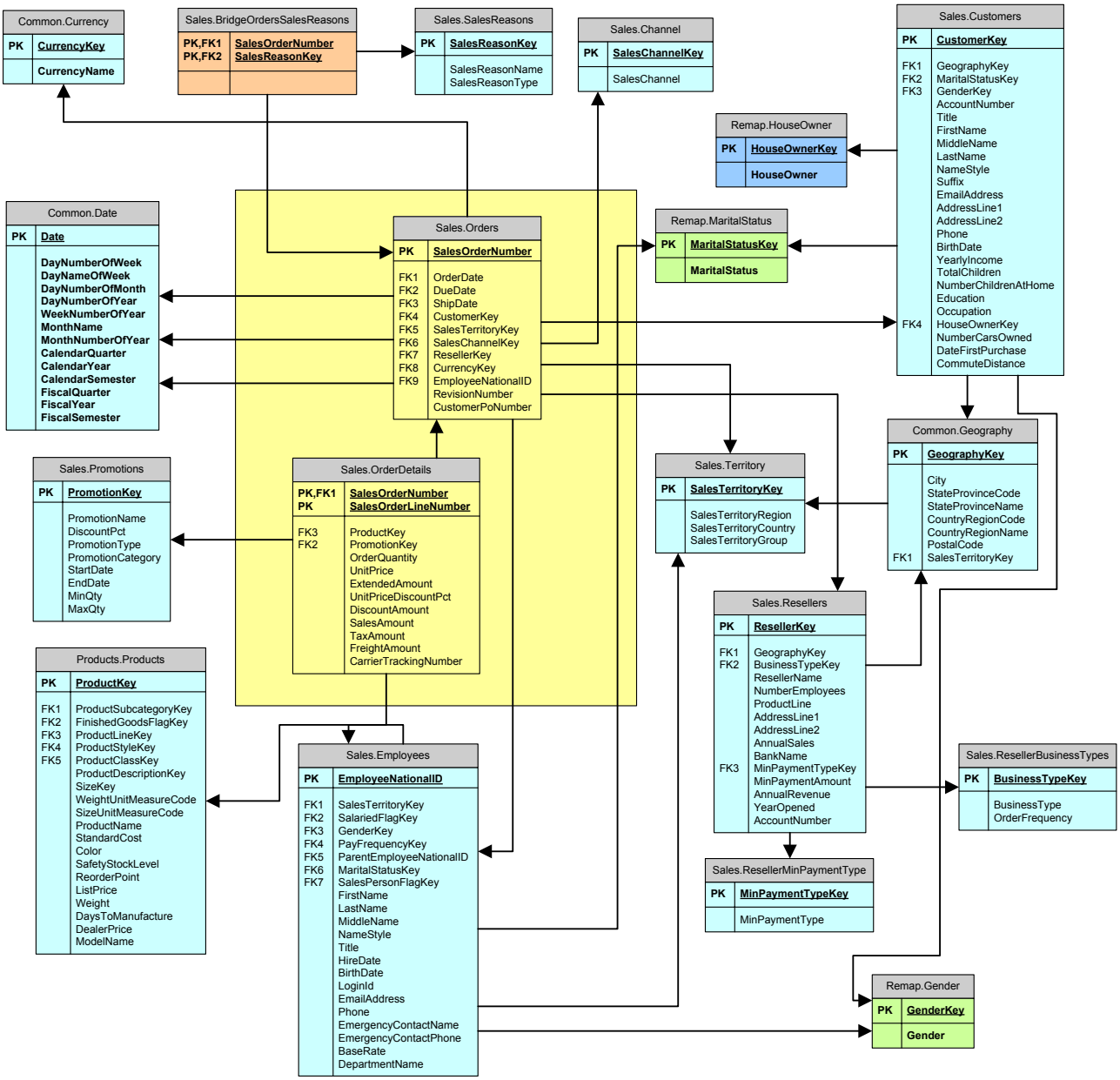


IMG (0130): Sales.Employees Subject Area

The important aspects to note here are:

- **Vacation hours and sick leave hours were stored in the dimension of the original BI solution**. We moved them to a new entity in order to be able to differentiate between different periods. If the data mart needs to expose the last value, it will gather the information easily but, from the data warehouse point of view, this entity has an historical depth that cannot be lost.
- **The OLTP maintains different tables for the historical values** of department, payfrequency and base rate. We merged them into a single historical table. Moreover, as we might be interested in the variations of the marital status, we added the MaritalStatusKey to the model. Please note that this is a very simple example of a much powerful feature of having a data warehouse. Even if the OLTP does not provide us a history of an attribute, we can easily add it to the data warehouse gaining the opportunity to use it to create an SCD later at the data mart level. As we stated before, we might decide to use or not to use a specific historical table in the creation of a single data mart. The data warehouse will hold the complete history and we will be able to decide later whether to use it or not.

Now it is time to show the complete Sales subject area:



IMG (0131): Sales Subject Area

The structure is quite complex but still much easier to understand than the original OLTP database.

- The difference between sales order headers and details has been maintained in order to make it easier to follow the relationships between entities. Moreover, in the data warehouse world, there is no need to de-normalize everything and so we do not do it.
- Several remap tables have been added to have a coherent way of describing codes in the OLTP database (S for Single, M for Married and so on).
- The ResellerBusinessTypes table has been created to show the relationship between business type and order frequency, which was hidden into the queries of the ETL process.

Please take some time to understand the whole structure. In the specific case of Adventure Works the data model of the data warehouse is very simple, but it is absolutely necessary that the BI analyst perfectly knows the structure of the data model, in order to master it and to be able to decide "where" he can search for something or where he should add a new entity.

USEFULNESS OF THE DATA WAREHOUSE LEVEL

Looking at the previous pictures, we might wonder if the data warehouse level is useful or not because, at first glance, it shows a level of complexity that is similar to that of the OLTP database. The answer is a definitive "yes" for those reasons:

- The design of the data warehouse is with analysis in mind, where the OLTP design has been done with "operation" in mind.
- The complexity is much lower when compared to the OLTP database. A slight level of de-normalization has been added and several small tables were added but, from the analytical point of view, the navigation of this model is much easier than the original OLTP database.
- The model shows arranged corporate data model, not the OLTP one. It does not force us to follow difficult links between technical entities, as it is the case in the OLTP database. All the "technical stuff" about the operational usage of the OLTP database disappears in the data warehouse.
- The data warehouse level gracefully integrates data coming from different sources, giving us a clear view of what data is available for analysis, no matter of its origin (CSV files, OLTP databases, EXCEL spreadsheets or anything else).
- The data in the data warehouse is granted to be clean and well organized. Because it is loaded by an ETL phase, we can take any necessary step to cleanse and integrate all the sources of data into a single view of the whole data model.

One last point stands for all. When we will see the queries that originate from the data warehouse and that are used to feed the data mart level, we will see that they are straightforward to write and to understand. This is because all the problems of converting the OLTP tables into a more coherent and analytical view of data are handled during the data warehouse loading phase. The technique is always that of "divide et impera": by adding a level between the OLTP and the data marts, we are separating a complex process into two easier ones, gaining a lot in terms of readability and maintenance requirements.

OLTP MIRROR DATABASE

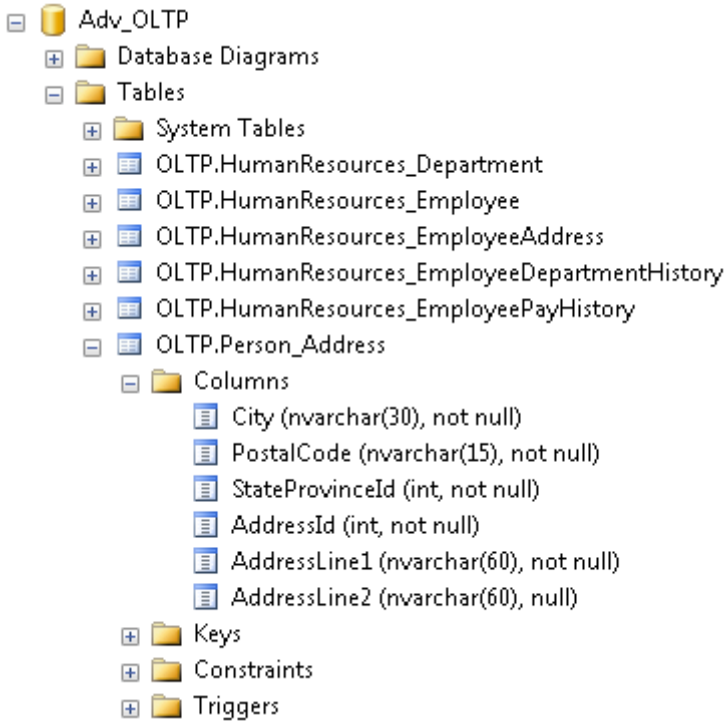
In the case of Adventure Works, the OLTP mirror database can be designed easily because we already have, in the original package, all the queries that need to execute in order to gather data from the OLTP. In a real solution, the work is harder because it requires detecting all the useful tables and moving them into the OLTP mirror database, removing later the useless columns.

In order to build the OLTP Mirror database, we analyze all the queries in the ETL package and create new tables in a database called Adv_OLTP that contains the useful tables and columns from AdventureWorks. The column names and types are identical to the original OLTP ones. This is the OLTP Mirror Database discussed in the previous chapters. It contains an exact copy of all the columns and tables needed for the ETL process. As we have already stated, it will not contain the entire OLTP database but only what is relevant for the ETL process.

Since we want to use schemas for subject areas in the OLTP mirror database, we move all the tables into a single schema called "OLTP" adopting a naming convention to maintain the old schema. The original table Sales.Orders, for example, has been moved to OLTP.Sales_Orders. In this way, we will be free to use the Sales schema with the data warehouse meaning without having to worry about schema usage in the original OLTP database.

The first good and very important result is that, if we want to see what is used in the OLTP database, we can just browse the Adv_OLTP database with SQL Server Management Studio and have a look at the columns in each table.

In the following picture, we show the OLTP.Person_Address table:



IMG (0125): Adv_OLTP useful to see the columns usage of the OLTP database

We can easily see that it contains fewer columns than the original table and we can be sure that all of those columns are necessary in the ETL phase. Moreover, if a table is not present in the OLTP schema, we can grant for sure that the OLTP table is useless in the ETL phase of the data warehouse construction.

If, for any reason, a new field will be necessary in the future, it would be enough to add it to the OLTP mirror database to show the evidence of this new requirement. No hand-made documentation is requested for this task. No hand-made document exists, it would be always out of date.

We might want to create a better documentation for the OLTP database. In that case, a good solution is to make a reverse engineer of the OLTP mirror database with a modeling tool and document all the columns and tables in that tool. We use Microsoft Office Visio (Visio hereinafter) as a modeling tool in our example. After the first reverse engineer, we can use Visio to make any update to the OLTP database and use Visio reports to document the OLTP mirror database.

OLTP Mirror Loading

After the definition of the OLTP Mirror tables, we need a method to load it from the original AdventureWorks database.

We could create an SSIS package with several data flow tasks that SELECT data from the original database and put them into the OLTP Mirror. This technique has several drawbacks. First, the fact that we have an SSIS package to maintain. Moreover, the degree of parallelism obtained is very small.

Instead, we used SqlBulkTool with a proper XML configuration file that eases the process of mirroring. You can find source code and full description of this tool at www.sqlbi.eu/SqlBulkTool.aspx. We built this tool just for this purpose.

Using this tool, we only need to write an XML configuration files that looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Configuration
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="SqlBulkToolConfig.xsd"
```

```
    MaxNumberOfThreads="15" >
    <DataBases>
        <DataBase
            Name = "OLTP Mirror"
            Run = "true"
            DataBaseClass="SqlServer"
            SourceConnectionString="Data Source=localhost;Integrated Security=True;Initial
Catalog=AdventureWorks"
            DestConnectionString="Data Source=localhost;Integrated Security=True;Initial Catalog=Adv_OLTP">
            <Tables>
                <Table
                        SourceTableName="HumanResources.Department"
                        DestTableSchema="OLTP"
                        DestTableName="HumanResources_Department"
                        TruncateBeforeLoad ="true"/>
                <Table
                        SourceTableName="HumanResources.Employee"
                        DestTableSchema="OLTP"
                        DestTableName="HumanResources_Employee"
                        TruncateBeforeLoad ="true"/>

                … … …

                <Table
                        SourceTableName="Sales.Store"
                        DestTableSchema="OLTP"
                        DestTableName="Sales_Store"
                        TruncateBeforeLoad ="true"/>
            </Tables>
        </DataBase>
    </DataBases>
</Configuration>
```

Having the source and destination connection strings and each table definition, the tools analyzes the metadata of the tables and generates proper SQL statements that produce the mirror. As the tool adapts itself to the database metadata, we will not need to change anything in the configuration file when we will add or remove columns to the mirror tables. Moreover, because the tool is intended to produce a mirror at the fastest speed, we can rely on its design and forget about any SSIS configuration to maximize mirroring speed.

We do not want to make any advertising about SqlBulkTool, at least because it is free and provided with full source code! The point here is that the BI analyst should not use SSIS only because SSIS exists. There are cases where a bit of coding is the best technique to gather the best results, both in terms of speed and design. In these cases, the choice of the best tool is very important to the success of the full BI solution.

Moreover, there are some very specific situations where we will not be able to create the OLTP mirror database. In these cases, we will skip this step and, instead, produce a database composed only of views that gathers data from the OLTP and exposes them to the ETL phase.

OLTP Mirror Views

The next task of the OLTP Mirror database is to feed the ETL phase. In order to do it, we write a set of views that introduce field renames and – optionally – some very simple computation to feed the ETL phase. Those views will have the same schema and name of the tables in the destination data warehouse.

Some words of caution: OLTP Mirror is the source of the ETL and not the first step in the ETL phase. This consideration is very important because, sometimes, we will be tempted to introduce some kind of computation in the OLTP views, as this might seem to ease the process of the ETL. In our opinion, this is wrong. The OLTP views should not contain any kind of computation.

A good example of it is the Sales.Products view that retrieves the products from the OLTP for the creation of the Product entity in the data warehouse:

```
CREATE VIEW Products.Products AS
    SELECT
        ProductKey              = p.ProductNumber,
```

```
        ProductSubcategoryKey   = p.ProductSubcategoryID,
        FinishedGoodsFlagKey    = p.FinishedGoodsFlag,
        ProductLineKey          = p.ProductLine,
        ProductStyleKey         = p.Style,
        ProductClassKey         = p.Class,
        ProductDescriptionKey   = p.ProductModelID,
        SizeKey                 = p.Size,
        WeightUnitMeasureCode   = p.WeightUnitMeasureCode,
        SizeUnitMeasureCode     = p.SizeUnitMeasureCode,
        ProductName             = p.Name,
        StandardCost            = p.StandardCost,
        Color                   = p.Color,
        SafetyStockLevel        = p.SafetyStockLevel,
        ReorderPoint            = p.ReorderPoint,
        ListPrice               = p.ListPrice,
        Weight                  = CONVERT (FLOAT, p.Weight),
        DaysToManufacture       = p.DaysToManufacture,
        DealerPrice             = p.ListPrice * 0.60,
        ModelName               = pm.Name
    FROM OLTP.Production_Product p
        LEFT OUTER JOIN OLTP.Production_ProductModel pm
            ON p.ProductModelID = pm.ProductModelID
```

The view computes DealerPrice using a constant (0.60) and produces the dealer price, assuming that the dealer price is roughly 60% of the list price. This kind of computation is required in the ETL phase but it should not be done by this query. As it is a part of the ETL, its place is the SSIS package that implements the ETL phase. Hiding this information in the source query will create confusion when we want to understand which fields the ETL process computes and which ones are derived from the OLTP.

The right way to compute the Dealer Price is to gather the list price from the OLTP and then computing the value in a “derived column” transformation. In this way, the computation is done in its proper place and any SSIS programmer will easily understand how it is computed.

A good question at this time is “What about JOINs?” At first glance, it seems that a JOIN, as a structural change in the table definition, should not exist in the OLTP for the same reasons. Again, this is not true. Any metadata change such as column rename, column type changes, JOINS and WHERE conditions will find its perfect place in the OLTP queries.

The reasons of being of the OLTP queries are:

- Make the OLTP database easier to manage from the ETL packages
- Introduce a light level of denormalization, removing useless tables that can be better represented by columns
- Rename OLTP columns in order to make them more suitable for the data warehouse

As always, these rules are not absolute. Sometimes a light level of computation is welcome in the OLTP queries. We have always to remember that building a data warehouse solution is very similar to writing source code: clarity is the final goal and no rule will ever lead us to success, only our brain will do it.

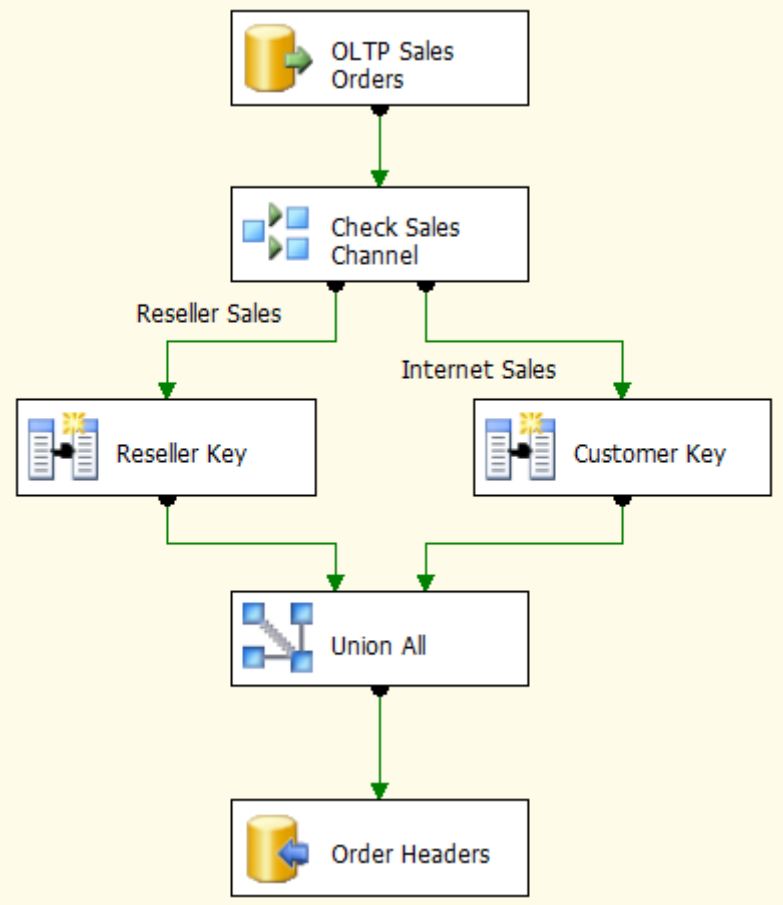
Beware that, even if JOINs are acceptable in the OLTP queries, they must be local to the OLTP mirror database. There is no reason to make a join between different databases. If this is needed, we will handle it with SSIS using lookups. We must always adhere to the concepts of locality. A query must be understandable in the environment where it resides. If we need to look into another database (the configuration, for example), then we are forcing the reader of the query to understand the ETL logic and we know that there is a lack of analysis that needs to be filled.

As another example, we can consider the OLTP query for the Sales.Orders. The sales table contains the ID of the customer. The customer can be an internet final customer or a reseller, because the OLTP stores resellers in the customer table. As the data warehouse will contain two different structures for resellers and final users, the single customer ID will be split in two different IDs: one for resellers and one for final customers.

We could write the query like this:

```
SELECT
    SalesOrderNumber    = soh.SalesOrderNumber,
    OrderDate           = soh.OrderDate,
    DueDate             = soh.DueDate,
    ShipDate            = soh.ShipDate,
    SalesChannelKey     = c.CustomerType,
    CustomerKey         = CASE WHEN CustomerType = 'I' THEN c.CustomerID ELSE NULL END,
    ResellerKey         = CASE WHEN CustomerType = 'S' THEN c.CustomerID ELSE NULL END,
    CustomerType        = CustomerType,
    SalesTerritoryKey   = soh.TerritoryId,
    CurrencyKey         = COALESCE (cr.ToCurrencyCode, 'USD'),
    EmployeeNationalID  = e.NationalIDNumber,
    RevisionNumber      = soh.RevisionNumber,
    CustomerPONumber    = soh.PurchaseOrderNumber
FROM
    OLTP.Sales_SalesOrderHeader soh
        INNER JOIN OLTP.Sales_Customer c
            ON soh.CustomerID = c.CustomerID
        LEFT OUTER JOIN OLTP.Sales_CurrencyRate cr
            ON soh.CurrencyRateID = cr.CurrencyRateID
        LEFT OUTER JOIN OLTP.HumanResources_Employee e
            ON soh.SalesPersonID = e.EmployeeID
```

Again, we are hiding an important aspect of the ETL phase in an OLTP query. We are assigning a value to CustomerKey or to ResellerKey to divide the input in two streams that represent different orders. A better solution is to let the query return the CustomerID column and then write the code that separates between internet and reseller orders in the SSIS package:



IMG (0118): Reseller and Internet orders

Why is this decision so important? Think at "who" will look at each part of the ETL phase. A database administrator might be interested in looking at the views in order to understand "which fields from the OLTP are used in the ETL". He is not interested in the transformation of the columns. His interest is only on which ones are used and which ones are not. On the other side, an SSIS programmer might be interested in

looking at how the columns are transformed during the ETL phase. In this case, the SSIS programmer is not interested in anything but the transformation phase. The real origin of the data is not relevant to him. As we are good BI analysts, our goal is to make information available where it is needed. Therefore, we will move column selection in the OLTP views and column transformation in the SSIS package.

The decision about where to put each step of the computation is not easy, even if it is clear that a well designed ETL system will be very easy to read. The ability to produce a good ETL system is a fact of experience. We can only provide some examples and some guidelines, but the final decision is always up to the BI analyst. What we want to stress is that these decisions cannot be undertaken, because they are very important, as they will highly change the quality of the final result.

CONFIGURATION DATABASE

When the BI solution needs to handle different languages, the configuration database has a schema named "Translation" that holds all the translations needed. There are several tables there, but we will not discuss them beacuse they are very easy.

CONFIGURATION FOR THE SALES SUBJECT AREA

Since we have several values that are used to perform both the data warehouse and the data mart ETL phases and that are all relevant to the Sales subject area, we defined a table (Sales.Parameters) that holds these values.

The table is very simple and it represents a convenient place where to put configuration values that the users might need to change over time.

Name	Value	Description
DealerPercentage	0.70	Contains the percentage of the list prices that represents the dealer price
TaxPercentage	0.08	Contains the percentage of taxes for each sale
FreightPercentage	0.025	Contains the percentage of the amount that is spent on shipment

Following our method, we do want to declare in the database the usage of those values, because somebody will surely need to read them in the future. For this purpose, we define views that expose values in the table in a more clean way:

```
CREATE VIEW DataWareHouse.Products_Parameters AS
SELECT
    DealerPercentage = (SELECT
                            CAST (ParameterValue AS FLOAT)
                        FROM
                            Sales.Parameters
                        WHERE
                            ParameterName = 'DealerPercentage')
```

As usual, this view has two purposes:

- It declares the usage of the different values in the table to anybody who can open the database
- It declares that there is a dependency between the data warehouse (in the subject area of products) and the Sales.Parameter table. If we want to know more about this dependency, we will have to look at the ETL code that loads the Products subject area.

CSV FILES CONVERTED TO TABLES

The original solution had several CSV files that were loaded directly in the data marts. As CSV files are hard to read and understand, we moved all the CSV files into tables in the configuration database.

Because this is a demo, the tables are loaded during the database creation. In a real world solution, we would have two possible situations:

- The CSV files are provided from an external system. In this case, there would be a specific ETL process that loads these files in the data warehouse.
- The CSV files are edited by the user. In this case, there would be some kind of software that allows the user interacting with the configuration database and editing the information.

In our example the second solution is the correct one: the values are edited in the configuration database and then moved into the data warehouse, where they will be available to any subsystem that uses them.

DATA WAREHOUSE ETL PHASE

The data warehouse maintains a coherent view of analytical data taken from the OLTP database and needs to be loaded from the OLTP database. For this reason, we write an SSIS package that loads data from the OLTP and the Configuration, producing the data warehouse.

The data warehouse ETL phase will start with the views that gather data from the OLTP and move all information in the data warehouse.

It is very hard, if not impossible, to give generic advices about how to perform this step. Each OLTP system has its problems to solve. Moreover, the same OLTP system might contain data that is user-specific. Therefore, we can say that each customer has its own peculiarities.

Instead of giving general rules, we will spend some words on the decision we took about Adventure Works.

PAY ATTENTION TO COMPILED PLANS FOR VIEWS

Before diving into the project, we want to spend just a couple of words about a very important aspect of the whole methodology concerning the usage of VIEWS with SSIS.

Views are normally crated on an empty database and our programming model is full of views. We need to be careful to the execution plan that SQL Server will use for our views. If we SELECT from a very big table and do several JOINs, then it might be the case that SQL Server stores an execution plan that – at some point in time – will become inefficient.

The general rules of SQL Server for the recompilation of the execution plans are valid for OLTP databases. From the BI point of view, it is always better to pay the price of a full recompilation of both VIEWs and Stored Procedure before each run, in order to be always sure to get the best plan for our current data.

For this reason, in all the Sources of our SSIS packages, long lasting views are expressed with the OPTION(RECOMPILE) flag, like:

```
SELECT
    *
FROM
    Sales.Resellers
OPTION (RECOMPILE)
```

CURRENT / HISTORICAL VALUES

In the OLTP system of Adventure Works there are two different modes of mantaining historical attributes. Let us take, as an example, the Employees:

- In Employees the columns “Pay frequency”, “rate” and “department” are stored in the historical table only, where both date of start and end of validity are stored too.
- In Products the standard cost is kept both in the historical table (which holds all previous values) and in the current table of products, where the current value is stored.

In the data warehouse, we need to maintain a coherent way of storing information. We decided to keep the current value in the primary table and the historical values in a separate table, for both products and employees. This will make it easier, for subsystems that do not need SCD handling, to recover the actual information about an entity.

Moreover, for the employees there are two different historical tables in the OLTP:

- EmployeePayHistory, which maintains the history of payfrequency and rate
- DepartmentHistory, which maintains the history of departments

To make the structure simpler, we merged both of them into a single historical employees table that contains both the information, merging the two histories into a single one. The query to do it is complex but the final structure is cleaner.

XML DATA TYPES

In the OLTP system, several values are stored in XML fields. These fields are hard to query and do not give good performance. For this reason, we decided to remove all XML fields, substituting them with normal columns of the proper type.

In this way, the OLTP view will need to handle XML remapping. After the data warehouse loading, all values will be exposed through standard SQL columns, making it easier to build queries against the data warehouse.Data Marts design

As we have seen from the data warehouse structure, the financial data mart is very independent from the sales one. In the original solution, both structures reside in the same cube. In our solution, we will build a different cube for financial analysis.

By making two distinct cubes, we are making the user’s life easier. He will only see the dimensions and fact tables related to one specific cube. It is useless, for example, to show the Accounts dimension in the sales cube, because this dimension is not related to any sale fact table.

If this was a real world solution, we might think to maintain the two cubes in two separate databases. We think at it because the BI solution will grow up in the future. Maintaining two separate databases for the two cubes could lead to some optimization in future implementations. We can imagine two separate groups of BI analysts carrying on the work on both cubes in parallel, being sure that they do not interfere each other. The choice between a single data mart database and separate ones is not an easy one. We should think carefully at all the implications of the choice and finally make it. Nevertheless, as both databases rely on the same data warehouse structure, we will always be able to separate them in the future when it will eventually be needed.

The data marts are pure Kimball data marts. We introduce surrogate keys, fact tables and dimensions. Up to now, we never minded if an entity will be a dimension or a fact table, because in the data warehouse world there is no such concept as a dimension or a fact table. On the other side, in the data mart world, all entities will become fact tables, dimensions or bridge tables and there will be no space for a generic “entity” item.

Moreover, it might be the case that the same entity in the data warehouse will produce more than one single entity in the data mart. Data marts exist to make the cube creation straightforward. Thus, we will cover all the aspects of the final cube production in the data mart.

Before going on with the description, we need to stress a concept: data marts are not cubes. We may decide to do both cube implementations and/or reporting on data marts. Moreover, a single data mart might provide information to more than one single cube.

Think, for example, at the date dimension. It will be used both in the financial cube as in the sales one. We will build a single date dimension in the data mart, but we already know that we might need to create two distinct SSAS date dimensions for the two different cubes. We will not do it in our example because we are producing a single solution. However, in case we would decide in the future to create two separate projects for the two cubes, we would need to feed the two dimensions from a single table.

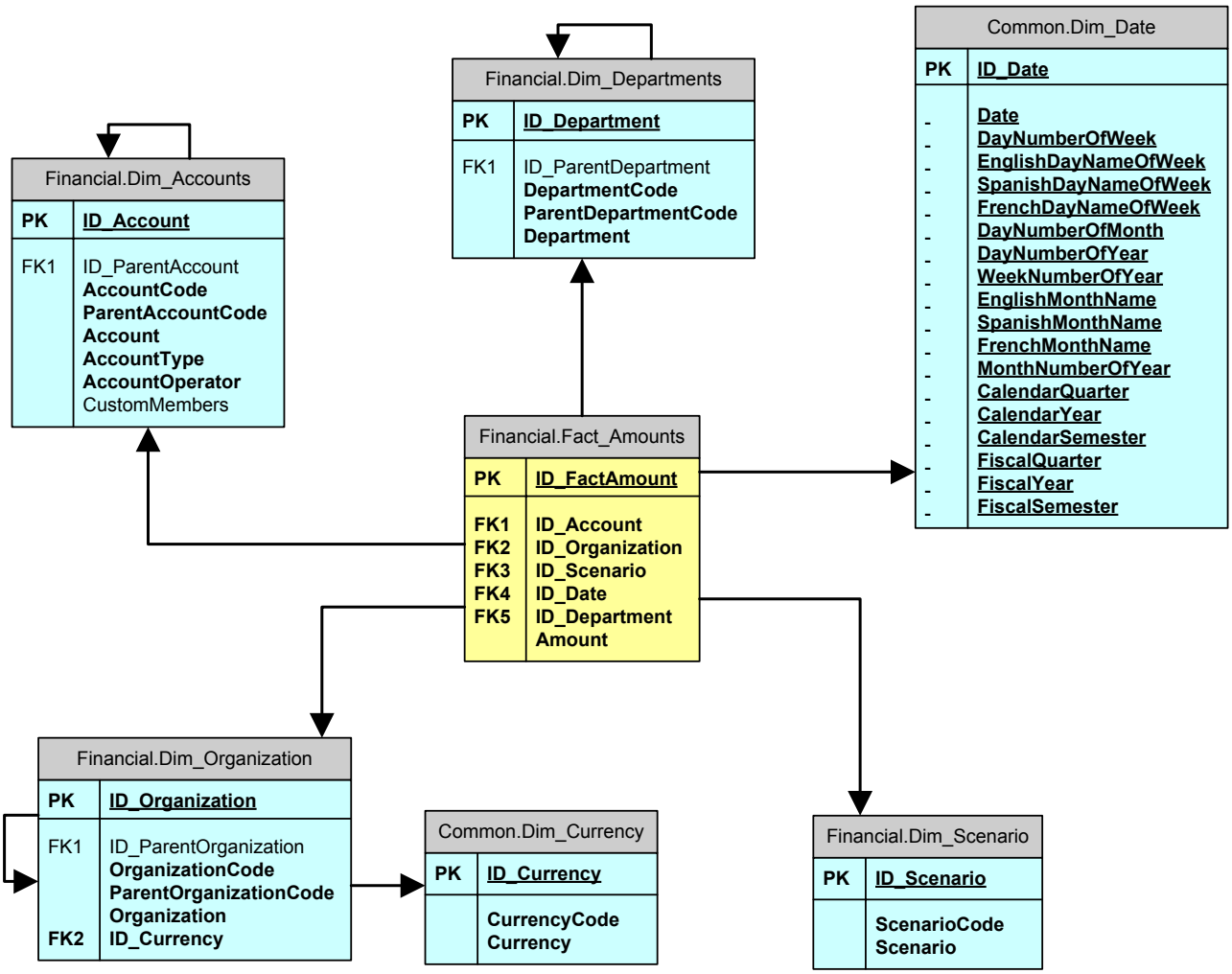
A better example is that of the currency dimension. We will use the currency as a dimension in the sales cube, but we will use it as an attribute for the Organization dimension in the financial cube. In order to avoid duplication of ETL code, we will generate a single dimension table, but we will show to the SSAS solution the currency as attribute or dimension, depending on our needs. Clearly, we will use views to create the final dimensions and fact tables consumed by the SSAS solution.

Now it is time to have a closer look at the data mart. Here we define three schemas:

- Common: to hold the common entities like the date dimension.
- Financial: to hold facts and dimensions of the financial cube.
- Sales: to hold facts and dimensions of the sales cube.

We can see that we are using schemas with a different point of view, if compared to schemas in the data warehouse. In the data warehouse, schemas were used to separate between subject areas. Here in the data mart area we use schemas to identify different data marts. Later, we will use schemas to define different cubes.

In the following picture, we can examine the financial data mart.



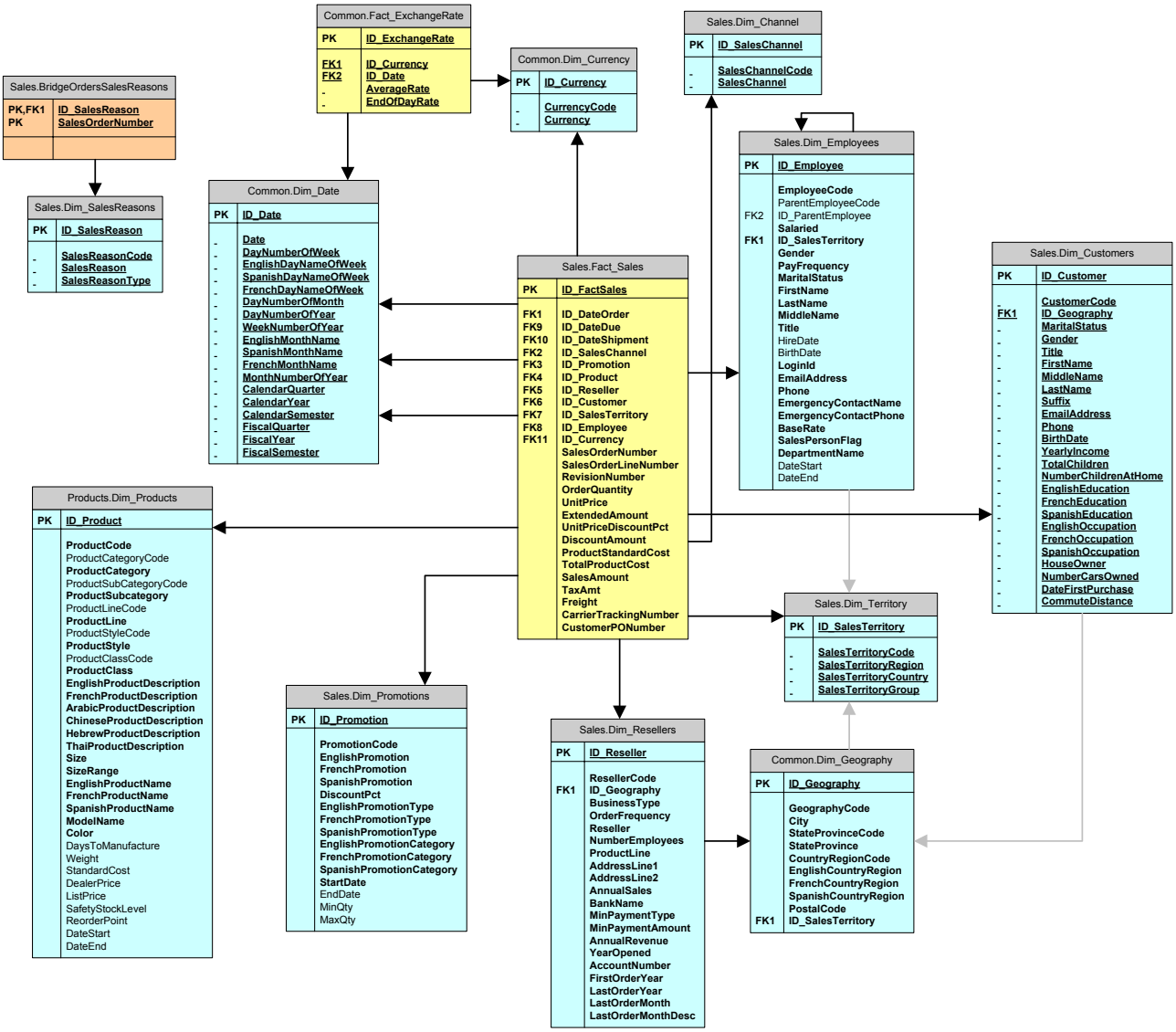
IMG (0133): Financial Data Mart

Now that we are in the data mart area, we start speaking about facts and dimensions. Moreover, we need to use surrogate keys instead of the natural ones used in the data warehouse. The reason is very simple: in the data marts we will define slowly changing dimension and the natural keys would not be enough to define a correct key into the tables.

All surrogate keys will start with “ID” prefix to identify them directly. The old keys have been renamed as “codes”, because they are no longer keys but simply identify original values.

The structure is very similar to that of a star schema even if, for technical reasons, we decide not to de-normalize completely the currency into the organizations. We could have done it but, as we have a currency dimension that is needed alone by the sales cube, we decided to leave it as a separate dimension in the Common schema. Later, when we will build the cube views, we will remove this inconsistency and will show a completely formed star schema.

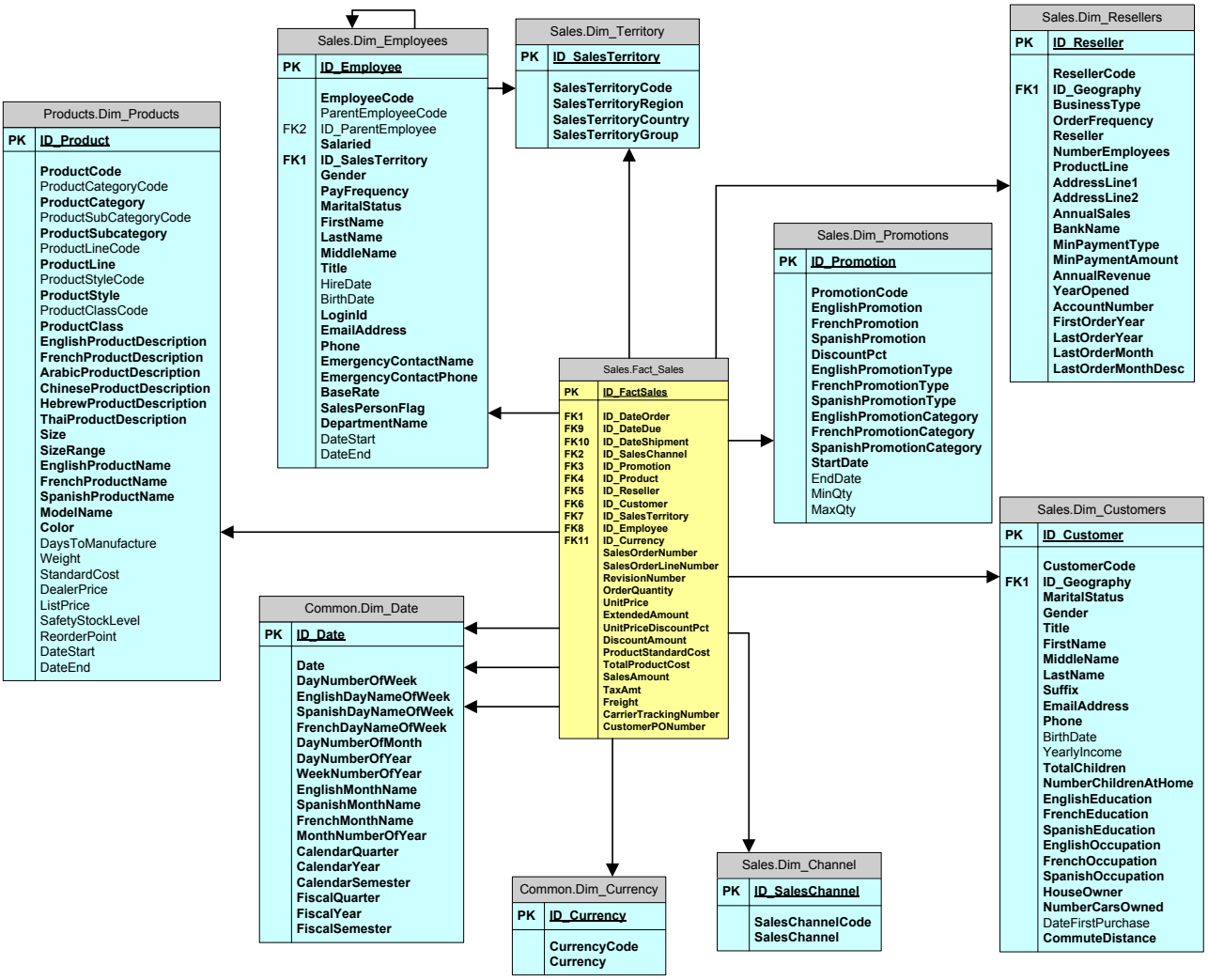
The sales picture is much more complex:



IMG (0134): Sales Data Mart

We have removed all the remapping tables and de-normalized everything in order to show the star schema that is necessary to the SSAS solution.

The structure looks complex only because there are two fact dimensions (Exchange Rate and Sales) and a bridge (Orders Sales Reasons). However, if we look carefully, we will easily discover the star schemas. In order to see them it is enough to build separate diagrams, showing only one fact table for each different diagram.



IMG (0135): Sales Data Mart - Fact Sales

If we look at the fact sales table with its entire referenced tables, we can easily recognize the familiar star structure of the data marts. The only exception is in Dim_Territory, which is referenced both from the employees dimension and from the fact table. This is very similar to Dim_Currency. We will remove the relationship in the cube views, where we will translate the relationship between employees and territory into attributes of the employees, thus clearing the inconsistence.

Data Mart Views

The data flows from the data warehouse into the data marts and it does it with an SSIS package that gathers its sources from views. We call them "Data Mart Views". We want to check some best practices about them.

In order to do that, let us examine in detail the "Dim_Resellers" view, which computes the dimension of resellers from various tables in the data warehouse:

```
CREATE VIEW SalesDataMart.Dim_Resellers AS
    SELECT
        ResellerCode        = ResellerKey,
        Reseller            = COALESCE (ResellerName,           'N/A'),
        GeographyCode       = GeographyKey,
        BusinessType        = COALESCE (b.BusinessType,         'N/A'),
        OrderFrequency      = COALESCE (b.OrderFrequency,       'N/A'),
        NumberEmployees     = COALESCE (NumberEmployees,        'N/A'),
        ProductLine         = COALESCE (ProductLine,            'N/A'),
        AddressLine1        = COALESCE (AddressLine1,           'N/A'),
        AddressLine2        = COALESCE (AddressLine2,           'N/A'),
```

```
        AnnualSales         = COALESCE (AnnualSales,            0),
        BankName            = COALESCE (BankName,               'N/A'),
        MinPaymentType      = COALESCE (p.MinPaymentType,       'N/A'),
        MinPaymentAmount    = COALESCE (MinPaymentAmount,       0),
        AnnualRevenue       = COALESCE (AnnualRevenue,          0),
        YearOpened          = COALESCE (YearOpened,             'N/A'),
        AccountNumber       = COALESCE (AccountNumber,          'N/A'),
        FirstOrderYear      = COALESCE ((SELECT
                                             YEAR (MIN (O.OrderDate))
                                         FROM Sales.Orders O
                                         WHERE O.ResellerKey = R.ResellerKey),
                                        0),
        LastOrderYear       = COALESCE ((SELECT
                                             YEAR (MAX (O.OrderDate))
                                         FROM Sales.Orders O
                                         WHERE O.ResellerKey = R.ResellerKey),
                                        0),
        LastOrderMonth      = COALESCE ((SELECT
                                             MONTH (MAX (O.OrderDate))
                                         FROM Sales.Orders O
                                         WHERE O.ResellerKey = R.ResellerKey), 0)
    FROM
        Sales.Resellers r
        LEFT OUTER JOIN Sales.ResellerBusinessTypes b
            ON b.BusinessTypeKey = r.BusinessTypeKey
        LEFT OUTER JOIN Sales.ResellerMinPaymentType p
            ON p.MinPaymentTypeKey = r.MinPaymentTypeKey

    UNION ALL

    SELECT
        ResellerCode        = CAST ('N/A'           AS NVARCHAR (15)),
        Reseller            = CAST ('N/A'           AS NVARCHAR (50)),
        GeographyCode       = CAST (-1              AS INT),
        BusinessType        = CAST ('N/A'           AS NVARCHAR (20)),
        OrderFrequency      = CAST ('N/A'           AS NVARCHAR (20)),
        NumberEmployees     = CAST (0               AS INT),
        ProductLine         = CAST ('N/A'           AS NVARCHAR (50)),
        AddressLine1        = CAST ('N/A'           AS NVARCHAR (60)),
        AddressLine2        = CAST ('N/A'           AS NVARCHAR (60)),
        AnnualSales         = CAST (0               AS MONEY),
        BankName            = CAST ('N/A'           AS NVARCHAR (50)),
        MinPaymentType      = CAST ('N/A'           AS NVARCHAR (20)),
        MinPaymentAmount    = CAST (0               AS MONEY),
        AnnualRevenue       = CAST (0               AS MONEY),
        YearOpened          = CAST (0               AS INT),
        AccountNumber       = CAST ('N/A'           AS NVARCHAR (20)),
        FirstOrderYear      = CAST (0               AS INT),
        LastOrderYear       = CAST (0               AS INT),
        LastOrderMonth      = CAST (0               AS INT)
```

There are several topics to cover about this view:

- The view is composed of a UNION ALL: the original source and the dummy row. All the dimensions should contain at least a dummy row where we will map the unmatched references. The technique of adding the dummy row directly inside of the view leads to some advantages:
    - If we ever change the view (i.e. adding a column to it), we will get an error if we forget to add the same column to the dummy record. This will make it easier to maintain the project over time.
    - By looking at the view, it is very easy to understand and/or review the values of the dummy record. Moreover, all the COALESCE in the main SELECT should be identical to the values of the dummy record. This can be easily checked and/or corrected too.
- The view contains several JOINs. This is acceptable as the JOINs are following foreign keys declared at the data warehouse level. Since we are querying the same database following original keys, JOINs are accepted and will make the ETL process easier. Moreover, JOINs will declare dependencies between dimensions and data warehouse entities, letting us to document those dependencies in an easy way.

- The main query contains some sub queries. These are accepted too, because they are computing MAX and MIN values from a related table. The data warehouse does not need to maintain these values over time. They will be computed when needed by each data mart view, since we cannot imagine the same values will be eventually useful to other data marts. Moreover, as the value they are computing comes from the data warehouse, any other data mart might be able to compute it in the same way, if needed.

We want to spend some more words on JOINs. In this first query, JOINs are used to remap keys to their complete description, In this way, the number of rows returned by the query is exactly the same as the number of rows in the Reseller Entity. This happens because Dim_Reseller is not a slowly changing dimension. A different situation happens with the Product dimensions in the view, of which we want to show only a fragment:

```
CREATE VIEW SalesDataMart.Dim_Products AS
    SELECT
        ProductCode                 = P.ProductKey,

… … …
        DateStart                   = ph.DateStart,
        DateEnd                     = ph.DateEnd,
        StandardCost                = ph.StandardCost,
        ListPrice                   = ph.ListPrice,
        DealerPrice                 = ph.DealerPrice,

… … …

        SafetyStockLevel            = p.SafetyStockLevel,
        DaysToManufacture           = p.DaysToManufacture
    FROM
        Products.Products p

… … …


            LEFT OUTER JOIN Products.ProductsHistory ph
                ON ph.ProductKey = P.ProductKey
```

In this case, since the product dimension is an SCD, we rebuild the complete history of it using the ProductHistory entity from the data warehouse. The LEFT OUTER JOIN will return more rows than the ones present in the Products table. This is one of the most powerful capabilities of the usage of a data warehouse to build data marts. From the same entity we can build SCD o standard dimensions, depending on the specific usage that we want to do of the entity in the data marts.

Even in this case, as JOINs are following foreign keys, they can (and should) be used in views, because they clearly declare the flow of dependencies.

Apart of complex selection, data mart views should not carry on any ETL step, as it was the case of data warehouse views. The ETL, if needed, should be part of the SSIS package, where programmers will search if they want to understand the ETL phase.

In the building of the data mart, we will use several views from the configuration database too. In this specific situation, we will search for translations of terms in the translation schema of the configuration database. In a real world situation, we might have more complex interactions between the configuration database and the data marts. Nevertheless, using views, all these interactions will be automatically documented.

DATA MART ETL PHASE

The ETL from data warehouse to data marts gathers its data from data mart views in the data warehouse and:

- Transform entities in the data warehouse in fact tables, dimensions and bridge tables. All the objects that reside in the data mart need to be pure Kimball objects; there is no space for ambiguity.
- Transform historical tables form the data warehouse in potential slowly changing dimensions. This is not mandatory. We can build both a standard dimension and/or a slowly changing dimension from the same entity in the data warehouse. The choice is up to the user and, thanks to the existence of the data warehouse, can be easily changed in the future.
- Add surrogate keys to all the dimensions. We need to do it because the keys of the data warehouse (which are natural keys) are no longer unique, due to SCD handling.
- May rename some columns in order to make their name more "user friendly", where we mean by "user" the end user.
- Computes some columns that can make easier the usage of the data marts. If, for example, the user wants to browse resellers using the "month of the last order", we will compute the value and its description during the data mart ETL phase. There is no reason at all to maintain this information in the data warehouse, because it can be easily gathered using views or SQL statements. Because the information needs to be present in the reseller dimension, we add it to the dimension and compute it from the orders table of the data warehouse.

There is only one thing that we cannot do in the data mart ETL phase: gather data from sources different from the data warehouse.

Data marts are a derivation of the data warehouse, they will always contain less information than the data warehouse. They will never expose more information than those present in the data warehouse. They might change the granularity of the facts, they may filter some dimension removing useless rows and/or columns, but they cannot add any source of information.

The reason is very simple: if some information comes from external sources directly into the data marts, then no other data mart will ever be built using the same information. It will be a complete failure of the BI solution if a user will ever show a report containing information that are not available and might be in contrast with reports produced by other people using the same data warehouse.

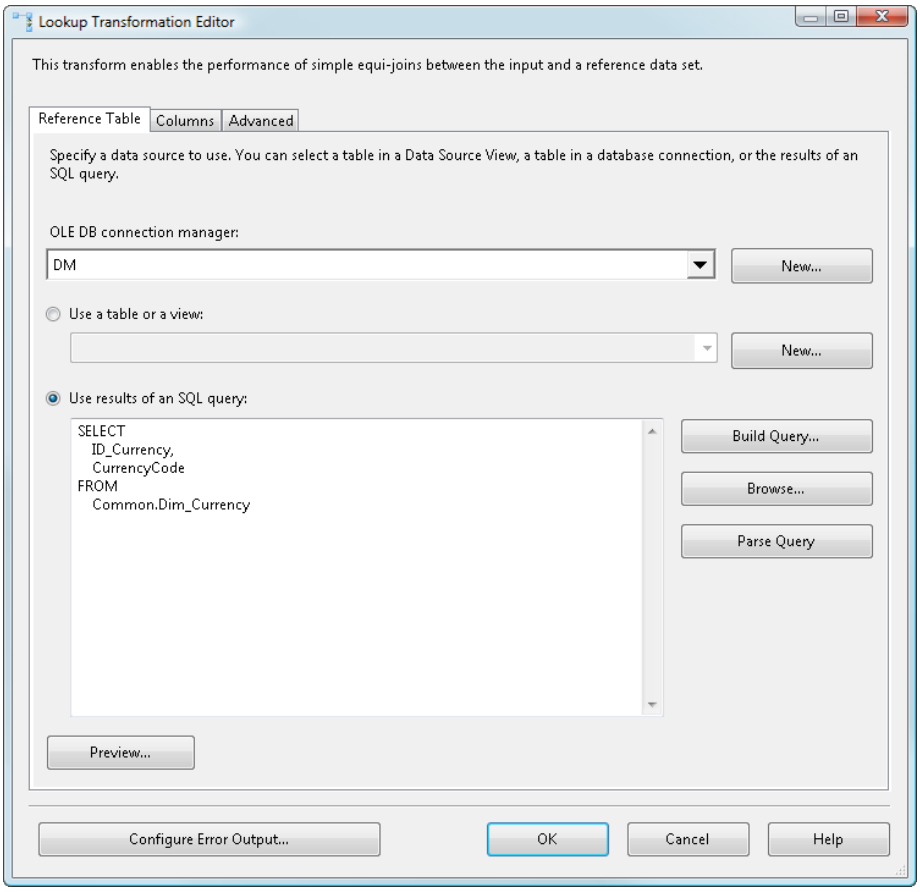
If a datum is shown in a report, then it must come from the data warehouse. If somebody shows the sales in a year, then he must have caught that information from the one and only source of information that contains the yearly sales: this is the data warehouse. If this condition is not enforced, then the data warehouse will be useless, as a central database holding the one and only truth.

Sooner or later, the user will ask to integrate some information in the cubes and we might think that the easiest way of adding them is to integrate them in a specific data mart, simply because we are told that this information is "only used by users of this cube". Beware of that: it is never true. Users will show their reports in meetings where they are used to discuss the future of the company. It is crucial that the same information is available to all users. They might decide to ignore it but, if somebody ever uses that information, they must be able to add it to their specific cubes.

SURROGATE KEYS HANDLING

Even if the reader may think that we are fan of views, in our model there is a specific point where we do not use views but prefer direct SQL coding. This place is the Lookup component, whenever it is needed only to resolve natural keys into surrogate ones.

If we look at the Fact Currency Rate data flow in the data mart ETL phase, we will find the lookup of the currency that looks like this:



IMG (0136): Lookup of surrogate keys do not use views

In the query, we do not use any view, we directly access the Common.Dim_Currency table and use its columns to build the lookup table.

The reason for which we do not use views in this case is very simple: there is no need to do it. Views are extensively used in our model to declare dependencies between different phases and/or entities of the ETL process. However, for surrogate keys, this dependency is already very clear by the fact that the Fact Exchange Rate table has a foreign key that points to the Dim Currency table. It is so evident that there is a dependency between these two items that it would be academic to enforce it with a view.

The lesson here is very simple: views are a powerful means of documenting dependencies but we use them only when we think that there is the need to do it.

DUMMY VALUES HANDLING

Whenever an incorrect lookup is found during the ETL of fact tables and/or dimensions, we need to assign an "unknown" value to that link. Even if NULLable columns are acceptable in the data warehouse, they are not in the data marts.

In order to make it easier to get these dummy values from the data mart, we normally build a view that gather all these values from the data mart and returns them all to the caller. It is up to the caller the decision about which of these columns to take or not.

```
CREATE VIEW GetDummyValues AS
    SELECT
        ID_Account_Dummy        = (SELECT ID_Account        FROM Financial.Dim_Accounts     WHERE AccountCode           = -1),
        ID_Organization_Dummy   = (SELECT ID_Organization   FROM Financial.Dim_Organization WHERE OrganizationCode      = -1),
        ID_Department_Dummy     = (SELECT ID_Department     FROM Financial.Dim_Departments  WHERE DepartmentCode        = -1),
        ID_Currency_Dummy       = (SELECT ID_Currency       FROM Common.Dim_Currency        WHERE CurrencyCode          = 'N/A'),
        ID_Date_Dummy           = (SELECT ID_Date           FROM Common.Dim_Date            WHERE ID_Date               = -1),
        ID_Scenario_Dummy       = (SELECT ID_Scenario       FROM Financial.Dim_Scenario     WHERE ScenarioCode          = 0),
        ID_SalesChannel_Dummy   = (SELECT ID_SalesChannel   FROM Sales.Dim_Channel          WHERE SalesChannelCode      = '?'),
        ID_SalesReason_Dummy    = (SELECT ID_SalesReason    FROM Sales.Dim_SalesReasons     WHERE SalesReasonCode       = -1),
        ID_Promotion_Dummy      = (SELECT ID_Promotion      FROM Sales.Dim_Promotions       WHERE PromotionCode         = -1),
        ID_SalesTerritory_Dummy = (SELECT ID_SalesTerritory FROM Sales.Dim_Territory        WHERE SalesTerritoryCode    = -1),
```

```
        ID_Employee_Dummy       = (SELECT ID_Employee       FROM Sales.Dim_Employees        WHERE EmployeeCode          = 'N/A'),
        ID_Geography_Dummy      = (SELECT ID_Geography      FROM Common.Dim_Geography       WHERE GeographyCode         = -1),
        ID_Customer_Dummy       = (SELECT ID_Customer       FROM Sales.Dim_Customers        WHERE CustomerCode          = 'N/A'),
        ID_Reseller_Dummy       = (SELECT ID_Reseller       FROM Sales.Dim_Resellers        WHERE ResellerCode          = 'N/A'),
        ID_Product_Dummy        = (SELECT ID_Product        FROM Products.Dim_Products      WHERE ProductCode           = 'N/A')
```

This “tip” might be very useful in ETL packages because it gives the programmer several advantages:

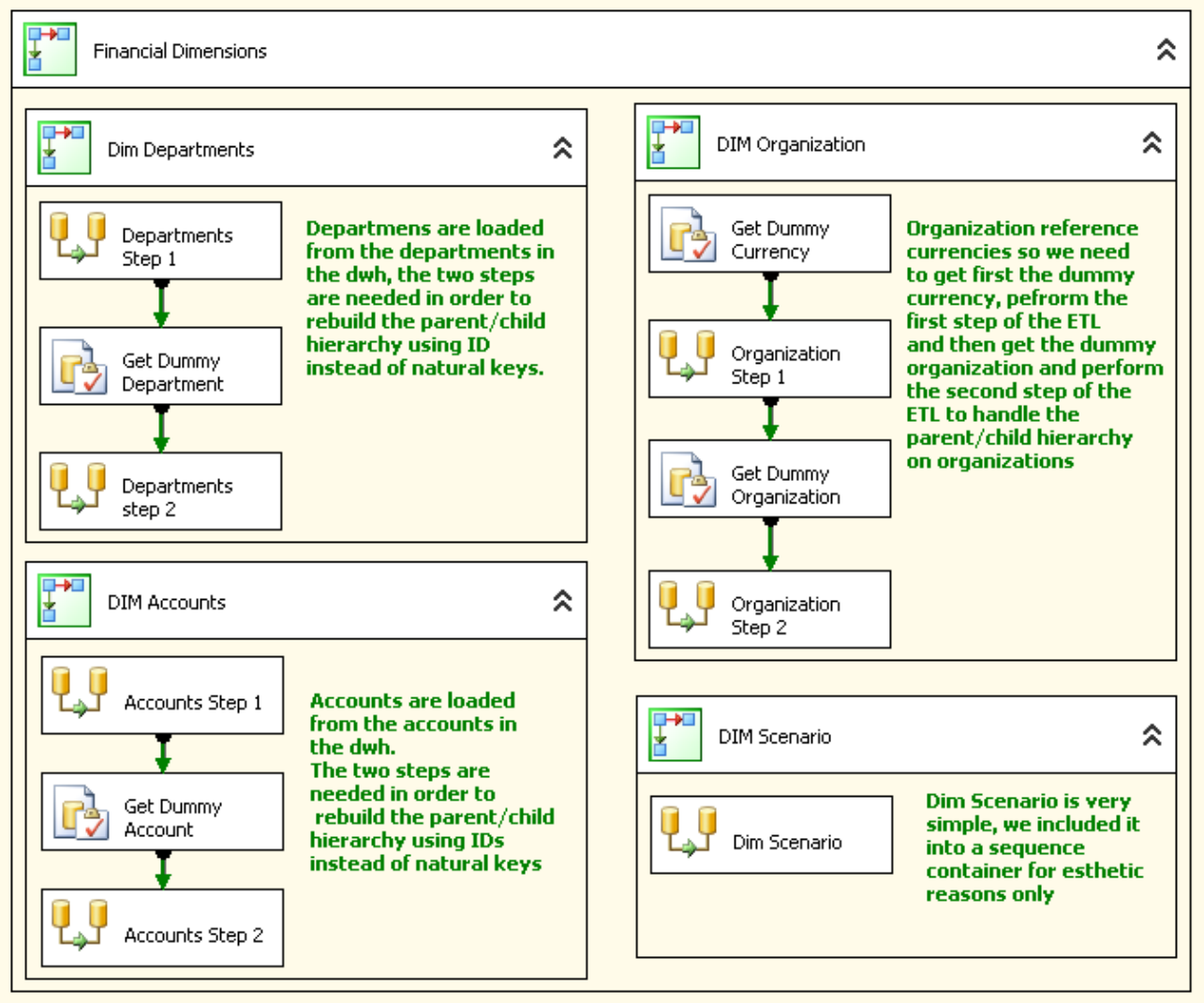
- The place where to declare a dummy value is only one; no SQL code will be sprinkled over the packages to get dummy values.
- If the programmer wants to check if all dummy values are defined he can simply look at a simple view
- The query to get dummy values is always “SELECT * FROM GetDummyValues”; it is up to the single task to decide what to get, adding rows to the resultset of the task.

As this query is very fast, it is possible to call it several times, whenever a dummy value is required, without worrying about performances.

Dummy values are stored in variables and, at the end of each flow, a derived column transformation handles the assignment of dummy values to all the columns that need a default value.

These variables are always local to the sequence container that loads a specific dimension/fact table. This makes it very easy to move sequence container from one place to another inside of the package, because each container is – in some way – self-contained. Using this technique, the SSIS programmer need to worry about real dependencies between facts and dimensions and does not need to handle dependencies generated by variable usage.

As usual, a picture is worth a thousand words:



IMG (0140): Sample usage of GetDummyValues

Using this technique, we call the view five times in this sequence and each time we take only the needed dummies. For example, look at the Dim Organization sequence container: during the first step, we need the dummy value for currencies, because organizations refer to currency and, in case of failure, the task need to use the dummy currency. Since no dummy organization might exist before the end of step 1, we need to load the dummy organization after step 1. However, this is easy, we just use the same code as before changing the result set. Moreover, all the variables used by the containers are local and the SSIS programmer is free to move each container elsewhere being sure that he is working in a safe mode.

CUBE IMPLEMENTATION

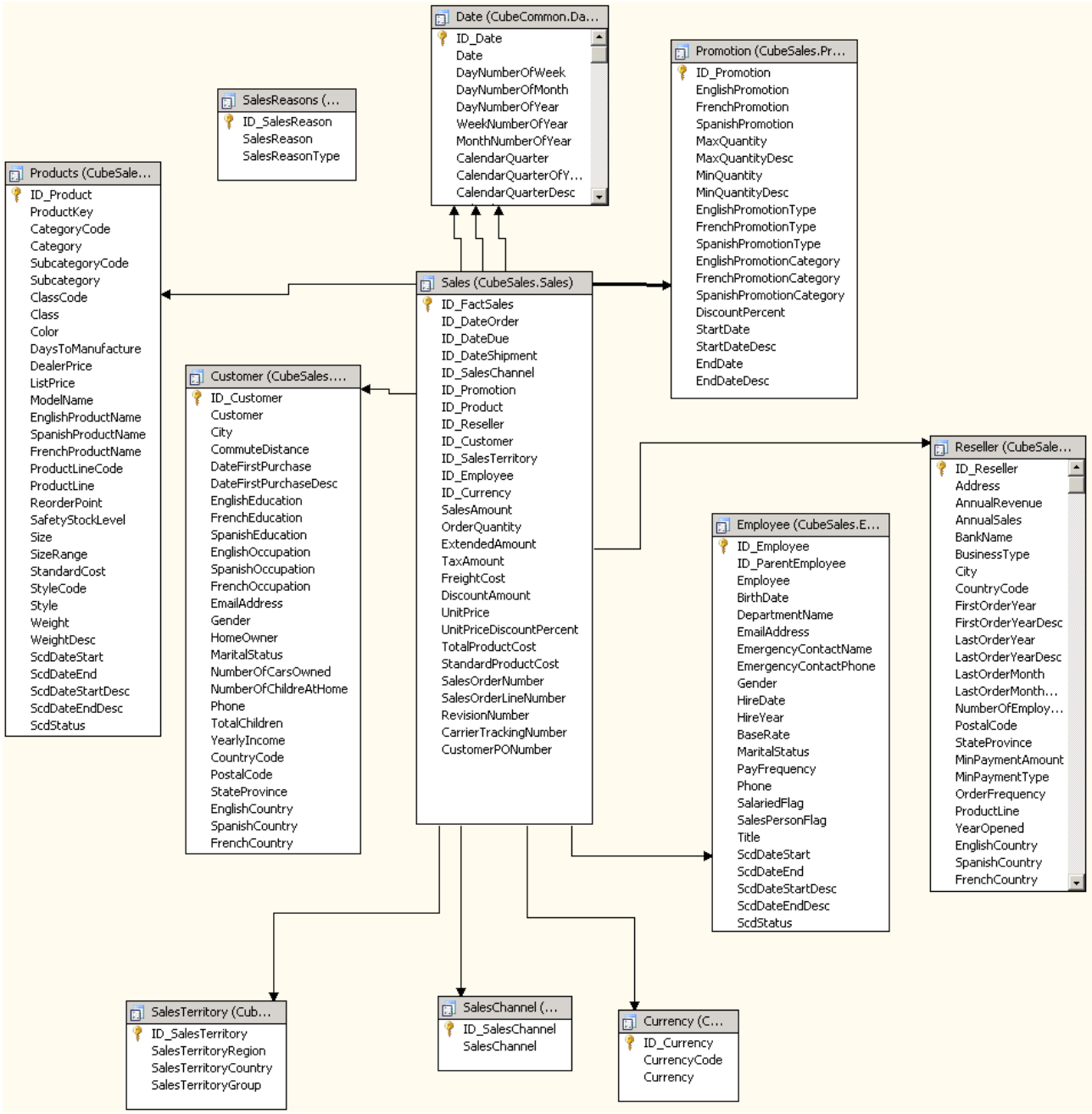
The final step to produce the results for our customer is that of producing the fact and dimensions that will be shown to the user.

DATA SOURCE VIEW

The first step in the creation of an SSAS solution is to build a data source view. The data source view will use only the views defined in the data mart database. Views are designed to show star schemas to the solution.

The careful reader should remember that, in the sales cube, the Geography dimension seemed to be used as a snowflake with the Customer dimension. The snowflake exists only at the database level and for performance reasons. From the cube point of view, there is no snowflake at all. In fact, the CubeSales.Customers view will resolve it performing a join between the two dimensions. This view exposes to the cube a simple Customer dimension where the columns coming from the Dim_Geography table becomes regular attributes.

In the following picture you can see that the structure shown in the data source view is a perfect star schema. There are no relationships between Geography and Customers. Moreover, Geography is not present in the data source view because there are no fact tables using it.

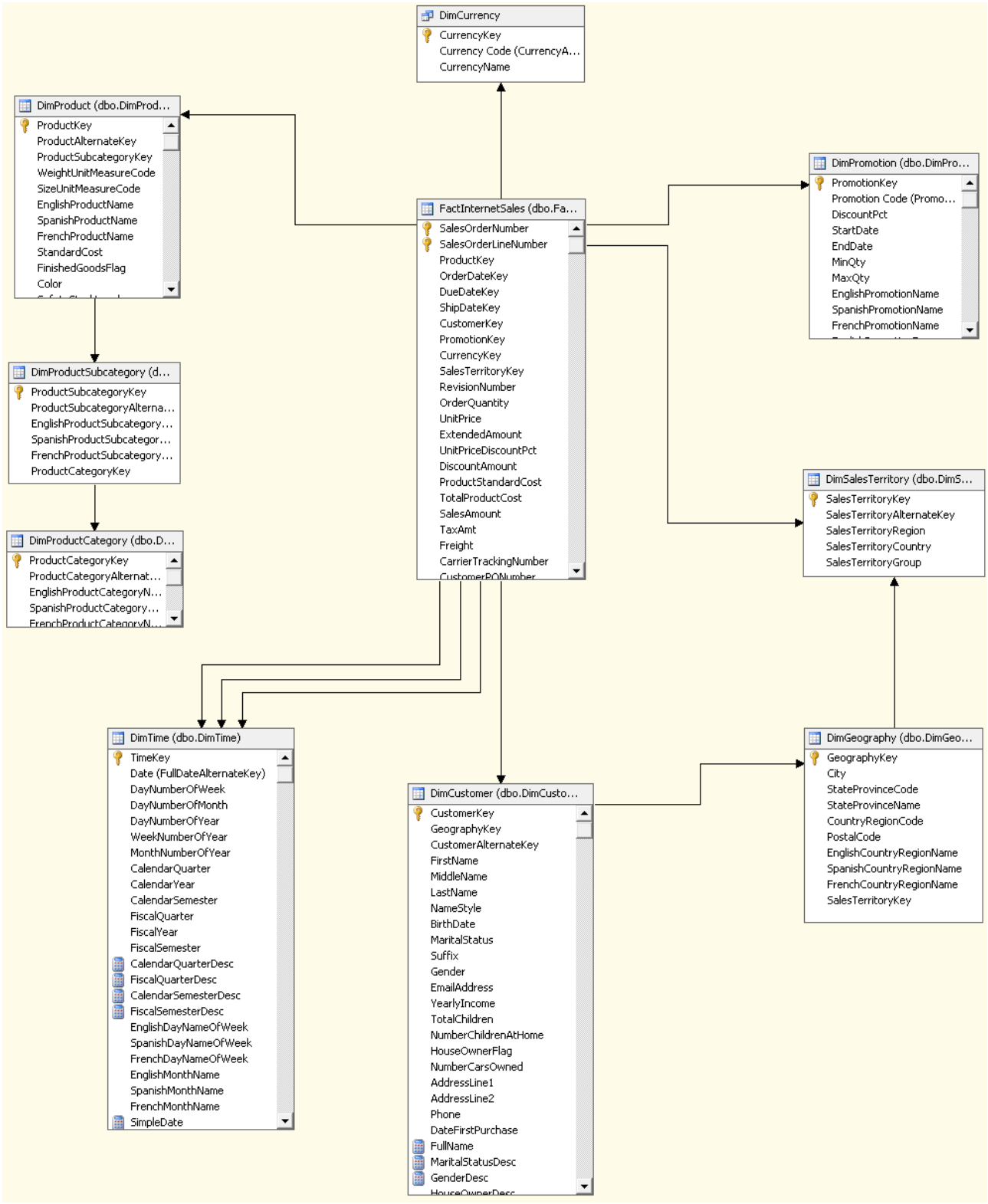


IMG (0137): Fact Sales Data Source View

The main goal of the data source view is to let the SSAS analyst have a clear view of the data structure in the data mart that he needs to use to build the cube. For this reason, he will only see the meaningful relationships between facts and dimensions and does not have to worry about strange and/or technical

items. The same applies for columns: if a column is not used in the SSAS solution, the view will hide it in order to make the full diagram easier to understand.

To appreciate the differences in design, compare the previous picture with the data source view for "Internet Sales" in the original SSAS solution:



IMG (0138): Original Internet Sales Data Source View

There are several problems in this data source view:

- The relationships between product, product subcategory and product category are too complex, involving three tables. Moreover, it is not evident, from the data source view, which columns of the various tables should become attributes of the product dimension and which should not.
- The fact table can reach Sales Territory following two different paths:
    - Directly with SalesTerritoryKey
    - Indirectly, through Dim Customer and then Dim Geography

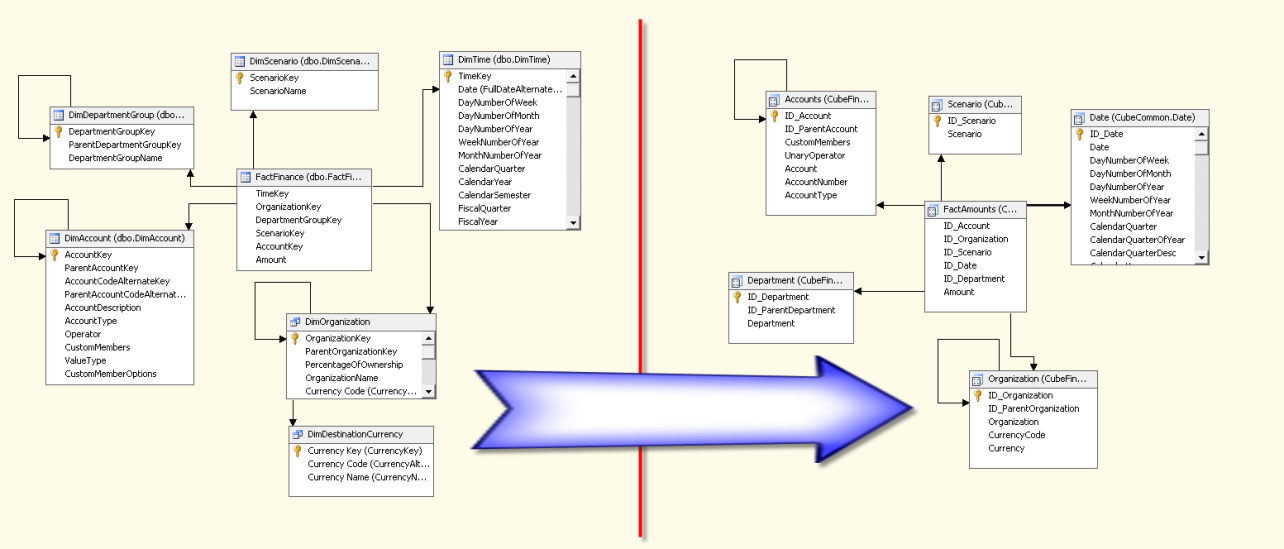
  Therefore, we have at least three different options to expose Dim Territoy to the user:

    1. Two Dim Territory role dimensions, one for the fact table and one related to customers
    2. Move Dim Territory attributes in customers and expose only the territory related to the fact table
    3. Create a Geography dimension that exposes Dim Territory columns as attributes and also relates this dimension to customers.

  The third solution is the one implemented in the Adventure Works cube but, by a first look at the data source view, it is not clear what the analyst should do. The existence of more than one option probably leads to confusion. Worse, it is not evident in the data source view when the cube is first developed and it will never be evident when we will look at it in the future. In order to understand the usage of Dim Geography in the original cube, the authors had to spend some time diving into the dimension usage of the cube to double check why Dim Geography was there.

The usage of views solves all these problems at the database level. Views show a clear structure that leads to no doubt. Clearly, because we are not using tables, the BIDS wizard will not recognize any relationships inferring them from the foreign keys and the SSAS analyst needs to rebuild relationships by itself. Even if this leads to some work during the first phase of cube development, this effort will be compensated in the future, when the cube will be updated or analyzed by someone else.

Another example can be that of the financial data source view, where we show both diagrams side by side.



IMG (0139): Financial Data Source View

The main difference is in the Organization dimension. The two tables (dim organization and dim destination currency) have been substituted with only one view (Organization). Using views, it is clear from the data source view that there will be no currency dimension in the financial cube. The code and name of the currency dimension will become attributes of the organization dimension.

Sales Channel

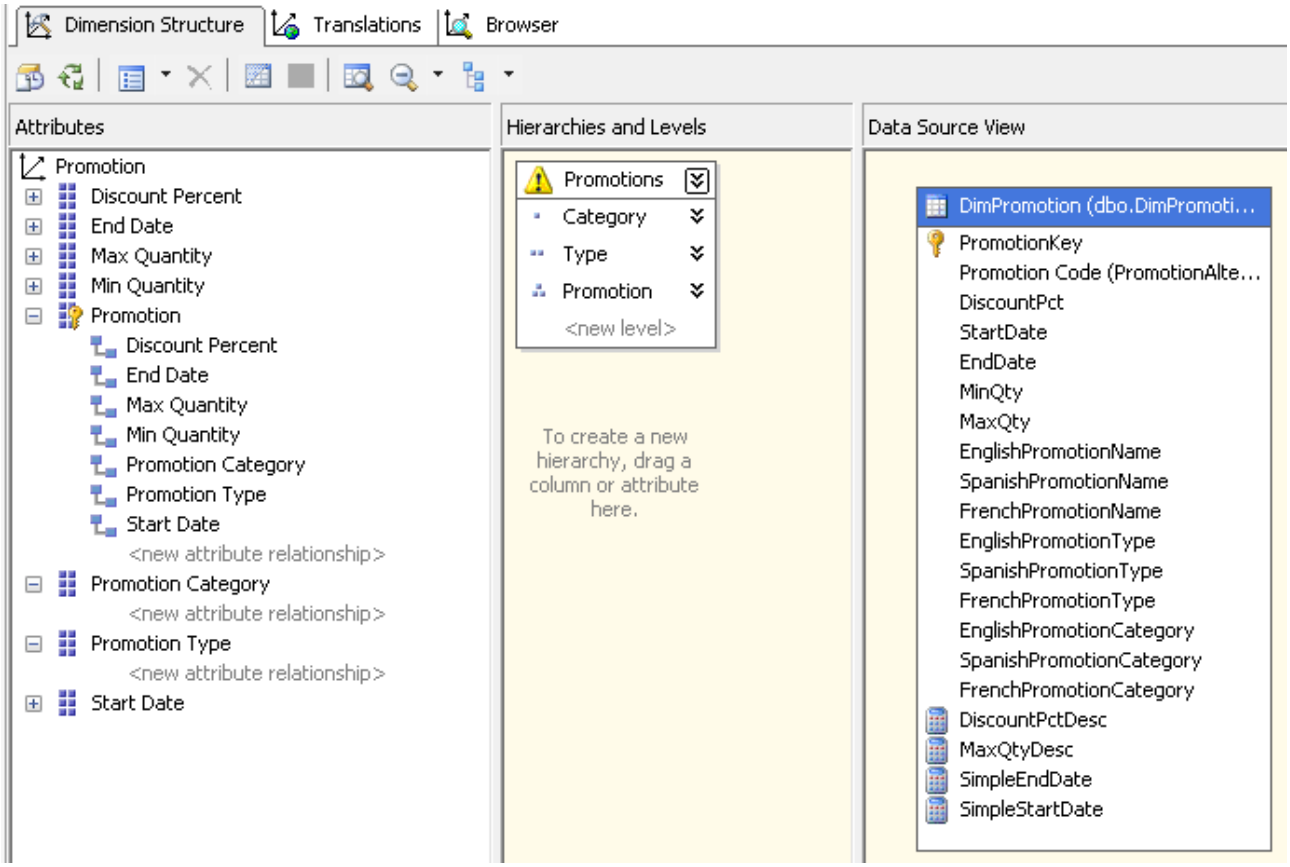
In the original solution, sales were divided between internet and reseller. This leads to a proliferation of fact tables and to a very complex structure of the dimension usage panel of the solution.

In order to make the cubes easier for the final user, we decided to merge the sales and add a dimension to separate them. We called this dimension "Sales Channel"; it contains only two rows: "INTERNET" or "RESELLER".

Whenever possible, these kinds of simplification lead to a simpler cube structure and, in the BI world, simpler normally means "more valuable", because users will accept more easily the introduction of the BI solution in their analysis world.

Promotions

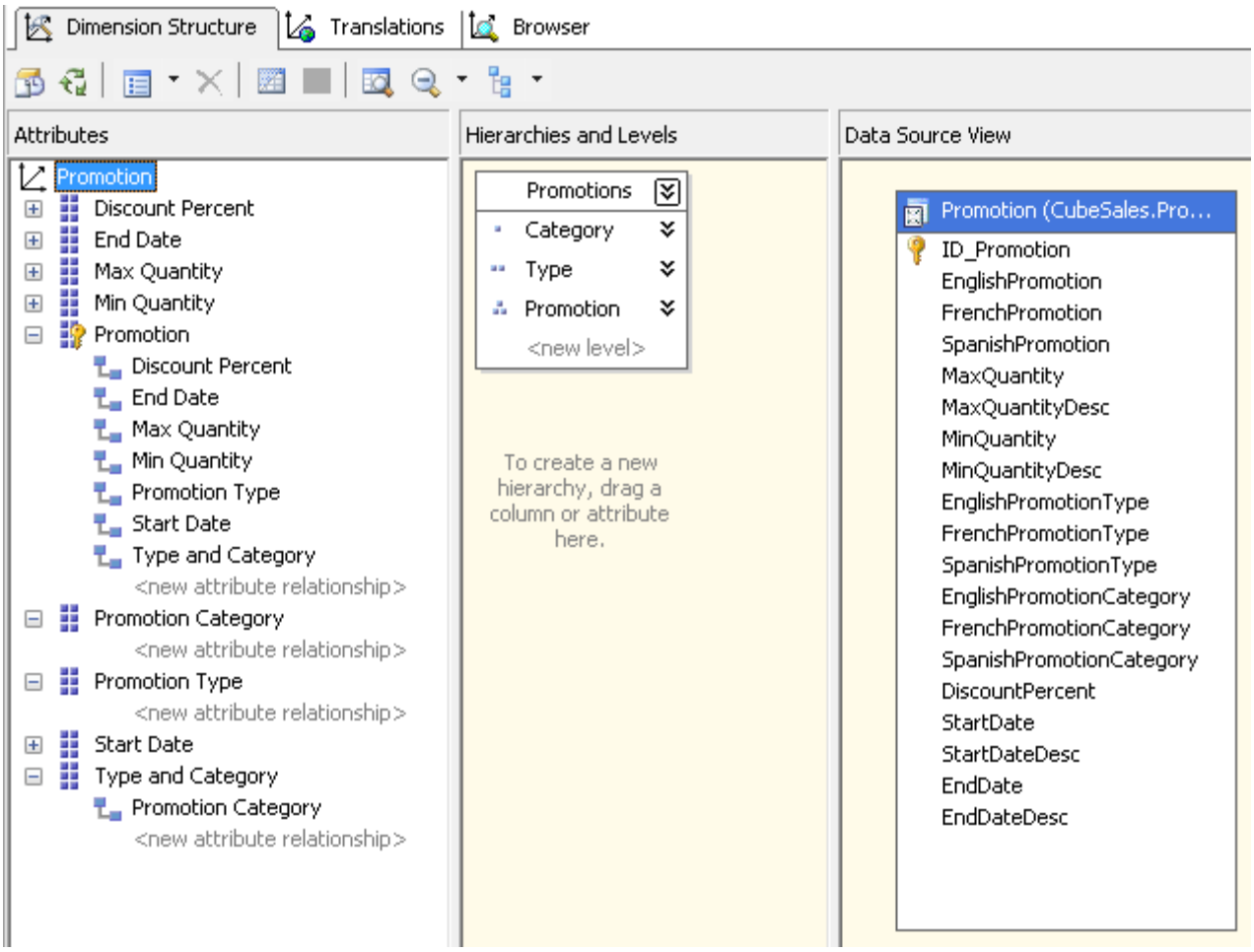
The promotion dimension is straightforward; the only problem is that the original one has a hierarchy that is not natural. The problem is that there should be no relationship between the type of the promotion and its category. Even if an analysis of the data shows that the relationship seems to exists, we suppose that the user decided that the relationship is not guaranteed to be true.



IMG (0123): Original Promotion Dimension

As we do not want to include any non-natural hierarchy in the solution, we create a new attribute (called Type and Category) that contains both the type and the category in its key column. Doing this, we created a new attribute that guarantees that there exists the required relationship between "Category" and "Type and Category".

The new attribute will have the Promotion Type as its name column. It will use a key with a higher granularity than its description. This is the required behavior in order to guarantee the correctness of the relationship.



IMG (0124): New Promotion Dimension

We will hide the technical attribute (the user will be able to directly browse only the Promotion Type and Promotion Category attributes) but we will use it to define the hierarchy, renaming it to “Type”.

When the user will browse the cube using the hierarchy, he will see the “Promotion Type” but – in reality – he will be browsing the “Type and Category” attribute.

When we complete the work applying the correct attribute relationships, the hierarchy becomes natural and the user will appreciate a huge increment in the browsing speed.

In the example of AdventureWorks this technique is not really useful, due to the very small amount of data that is processed by the cube. However, the rule of thumb is to avoid any non natural hierarchy in the whole solution. Following this rule will let us make the cube grow without suffering performance problems.

DOCUMENTATION OF THE WHOLE PROJECT

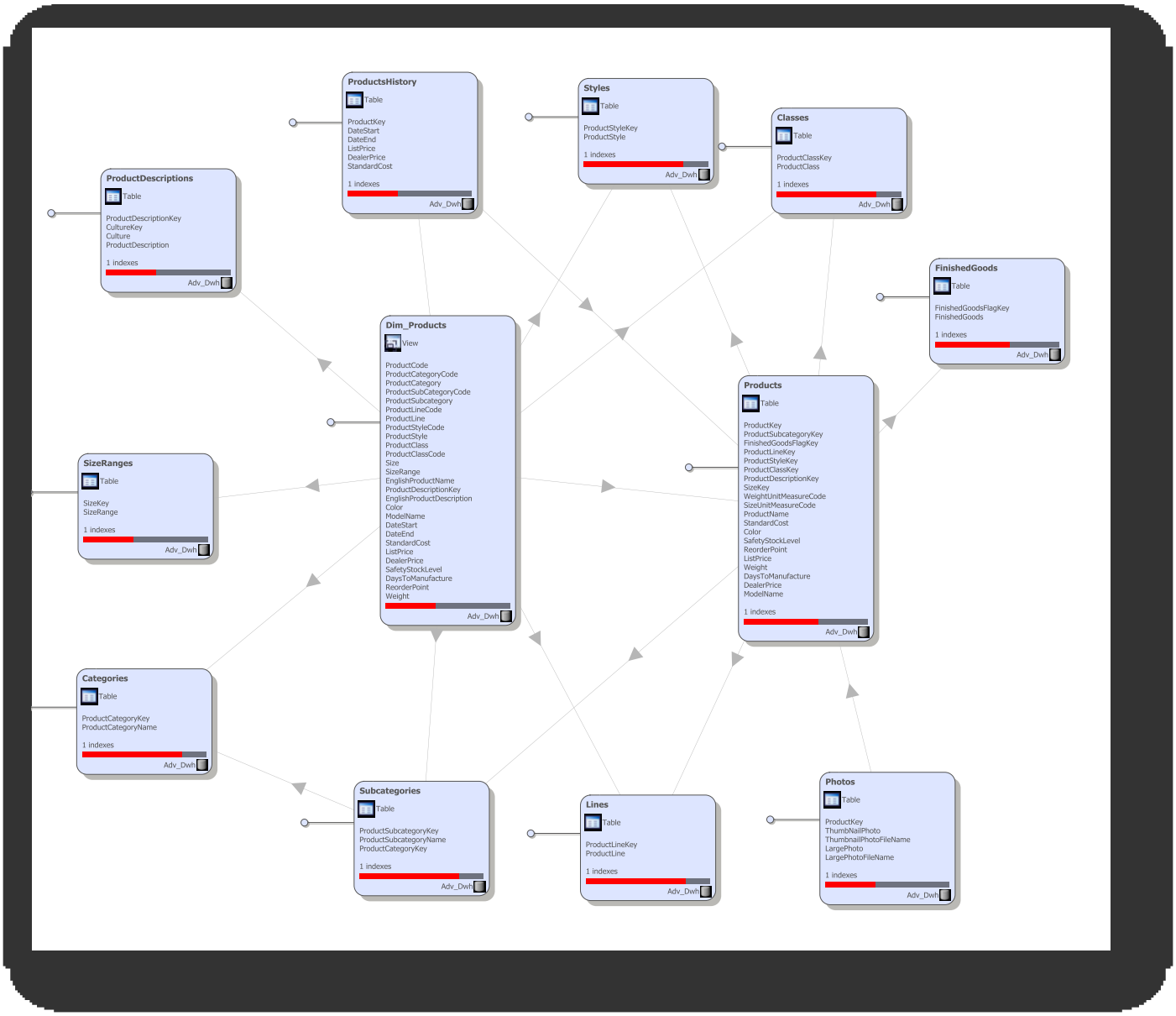
If the reader has been patient enough to reach this chapter, he will be rewarded with the best capability offered by our methodology: documentation.

We have stressed throughout the whole chapter the importance of using views to make interfaces between levels and the need to document with views each step of ETL phase, from the OLTP up to the final cube production. The main reason for doing so is that views can be analyzed by many third part tools that can in turn generate automatic documentation for a database, in the form of HTML and/or help files. This auto generated documentation process relies on the database structure so we know that anything written at the database level can be easily documented.

We decided to use two products so famous that they do not need any advertising. We are pretty sure that similar products from other software house will produce similar output. The point here is not to choose product A instead of product B but to understand that, using views to declare table dependencies, you are making the dependency tracking possible.

As you surely know well by now, in the Data Warehouse database we will have a view called SalesDataMart.Dim_Products that produces the input for the Dim_Products dimension in the SalesDataMart data mart.

If we ask RedGate SQL Dependency Tracker to analyze this view, we will get this chart in a snap, just the time to select the item to analyze:



IMG: SQL Dependency Tracker analysis of Dim_Products

It is clear for anybody, whether a technician or not, how to read this chart and the kind of dependency it shows. Moreover, as this chart can be generated in a few clicks and its sources is exactly the same source of the ETL process (i.e. the view) we can be pretty confident that the documentation will always be aligned with the source.

The other product used is Apex SqlDoc. SqlDoc has the capability to analyze a whole database and generate a help file that contains the documentation for the database[1]. Using SqlDoc we can analyze the same view and get two different pieces of information:

Top Custom Text

View: Adv_Dwh.SalesDataMart.Dim_Products

Collapse All

Statistics

Objects that depend on SalesDataMart.Dim_Products

Objects that SalesDataMart.Dim_Products depends on

Object Name	Object Type	Dep Level
Products.Products	Table	1
Products.SubCategories	Table	1
Products.Categories	Table	1
Products.Lines	Table	1
Products.Styles	Table	1
Products.Classes	Table	1
Products.SizeRanges	Table	1
Products.ProductDescriptions	Table	1
Products.ProductsHistory	Table	1
Remap.FinishedGoods	Table	2

Total 10 object(s)

Column Level Dependencies

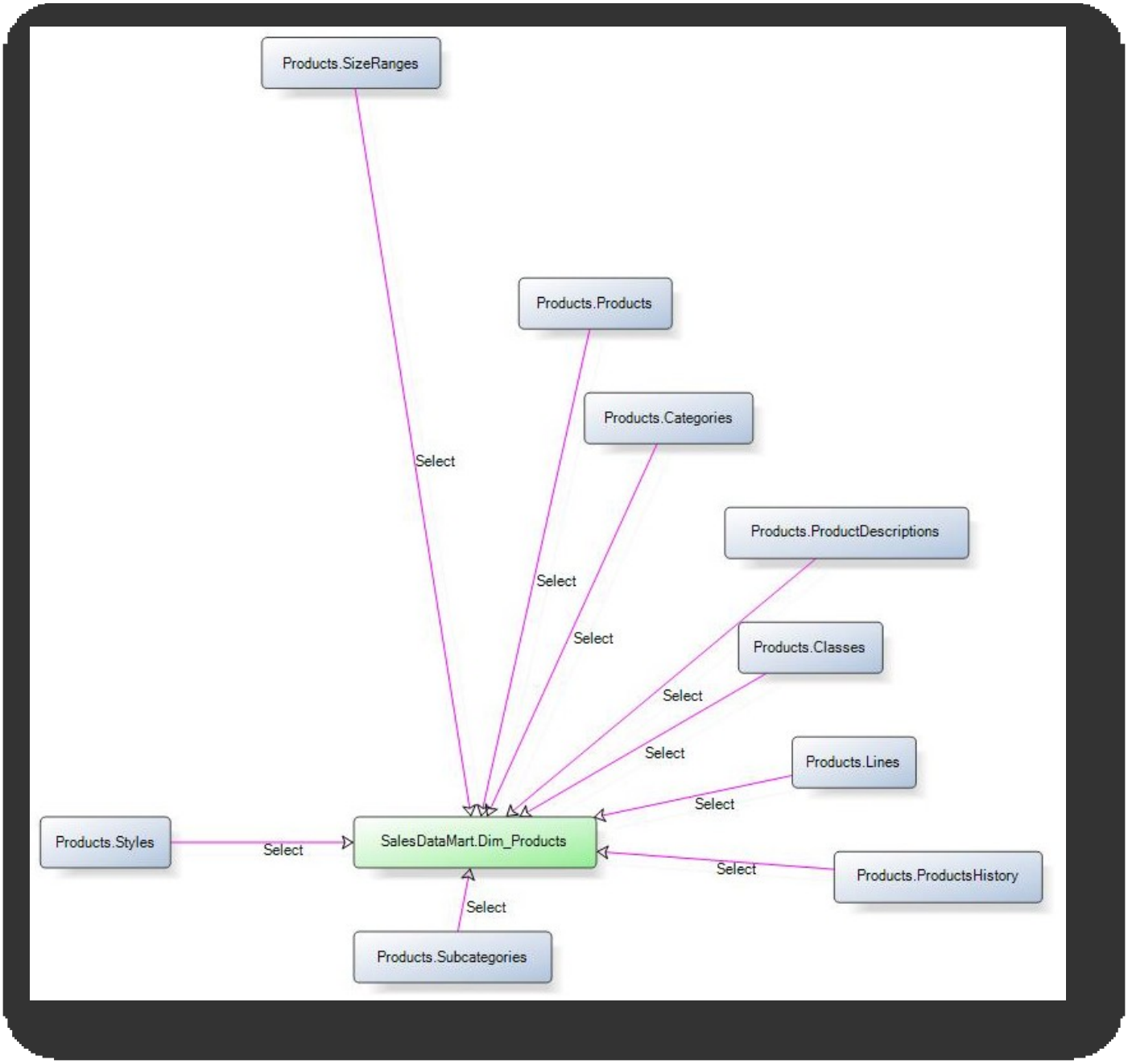
Object Name	Column	Object Type
Products.Products	ProductKey	Table
Products.Products	ProductSubCategoryKey	Table
Products.Products	ProductLineKey	Table
Products.Products	ProductStyleKey	Table
Products.Products	ProductClassKey	Table
Products.Products	SizeKey	Table
Products.Products	ProductDescriptionKey	Table
Products.Products	Color	Table
Products.Products	ModelName	Table
Products.Products	SafetyStockLevel	Table
Products.Products	DaysToManufacture	Table
Products.Products	ReorderPoint	Table
Products.Products	Weight	Table
Products.SubCategories	ProductCategoryKey	Table
Products.SubCategories	ProductSubcategoryName	Table
Products.SubCategories	ProductSubcategoryKey	Table
Products.Categories	ProductCategoryName	Table
Products.Categories	ProductCategoryKey	Table
Products.Lines	ProductLine	Table
Products.Lines	ProductLineKey	Table
Products.Styles	ProductStyle	Table
Products.Styles	ProductStyleKey	Table
Products.Classes	ProductClass	Table
Products.Classes	ProductClassKey	Table
Products.SizeRanges	SizeRange	Table
Products.SizeRanges	SizeKey	Table
Products.ProductDescriptions	ProductDescription	Table
Products.ProductDescriptions	ProductDescriptionKey	Table
Products.ProductDescriptions	CultureKey	Table
Products.ProductsHistory	DateStart	Table
Products.ProductsHistory	DateEnd	Table
Products.ProductsHistory	StandardCost	Table
Products.ProductsHistory	ListPrice	Table

IMG: Apex SQL Doc Analysis of Dim_Products (Textual analysis)

In the first image we can see that the tool is able to get single column dependencies, letting us document from where each single column of the view comes from.

The next picture shows something similar to the graphical representation of Dependency Tracker:

[1] This feature is covered by RedGate products too but we have chosen to show Dependency Tracker from RedGate, which produces a great graphical result and SqlDoc from Apex, which produces a great Help file. Please remember that our goal is not to that of advertize any product but just to show how to obtain good results.



IMG: Apex SQL Doc Analysis of Dim_Products (Graphical Analysis)

It is clear that these charts are very useful to add to any documentation we will ever produce for the BI solution. It will also help the reader to understand the flow of data of our ETL procedure.

So, even if we cannot say that using views we can totally avoid the need for documentation of the BI solution, we can say for sure that the usage of views, with the aid of some third party tool that analyzes them, will produce a good level of automated documentation. Moreover, the charts produced by automatic tools will be of great value when added to the standard documentation of the BI solution.