INTRODUCTION TO THE

SQLBI METHODOLOGY

Draft 1.0 – September 20, 2008

Alberto Ferrari (alberto.ferrari@sqlbi.eu)

Marco Russo (marco.russo@sqlbi.eu)

Introduction

This paper is an introduction to a methodology for the implementation of advanced BI solutions using Microsoft SQL Server, SQL Server Analysis Services and – more generally – the Microsoft suite for Business Intelligence.

It is difficult to clearly describe what the paper is about. It is much easier to say what it is not about and what you are supposed to know in order to get the most out of this paper.

- It does not describe what SSAS is, but sometimes it will try to show how it internally works. We will extensively use SSAS but will never spend time describing what you will do with a single click of the mouse. You are supposed to have a medium knowledge of the usage of SSAS and the ability to navigate easily through all its windows and properties. Instead, sometimes it will be necessary to go deep into its calculation technique in order to understand why a specific way leads to better results when compared to another one.
- It is not a paper about SQL Server Integration Services. Sometimes you will see screenshots of SSIS packages briefly described in order to make concepts clearer, but we will not spend time describing each feature of SSIS.
- It does not describe SQL at all. The book is full of SQL statements that you will need to immediately understand because we will provide no description of the SQL Syntax. For a BI analyst, SQL should be easy to read as English is.
- It is not an introductory or generic paper on BI solutions. You are supposed to be a BI analyst who knows the basis of BI and wants to discover something more about using the full power of SSAS to give better solution to your customers.
- It is not a Kimball versus Inmon book. You should already know both methodologies. When needed, we will point out some consideration about which one is better in particular cases, but we will not spend too many words on it.
- It is not a bible about BI with SSAS. Anything said here is our personal opinion derived from our personal experience. Over the years, this experience lead to successful implementation of several BI solutions, but we always had to tune up the concepts of principle stated here with the real problems we had to face with the customer. Whenever you will not agree with what we say, always remember that this is our vision of a BI solution, never the one and only truth.
- It does not cover only multidimensional design. It covers topics from the design of the initial database up to the complete design of the final OLAP cubes viewed by the user. We will go into some details when needed and leave some others when we feel that they are not so important.

If forced to give a definition of what this paper is about, we can say that this paper will provide you some new tools of the trade that – in our opinion – are very interesting for a good BI analyst. You may or may not decide to try to use them in your projects but, knowing them, you will be a better BI analyst.

The world of business intelligence is very challenging. Users require advanced analysis, BI solutions grow very fast and quickly pervade the whole corporation. The volume of data that BI solutions need to manage is huge and always growing.

We tried to collect in this book our personal experience in the development of BI solutions. Every time a new challenge rises, we always try to look at it as an opportunity to create new patterns that will be useful in the future. When we solved the first challenge, we take some time to better study it and develop a methodology that will make it easier to solve similar situations in the future.

The result of this work is a collection of considerations that finally leads to a model of implementing BI solutions. From the OLTP database to the final report required from the user the way is very long. We are trying to provide you some hints and insights that – over the time – lead us to smooth this way. We really hope that the same hints will be useful for you.

The reference for this book is not that of data warehouses of multi terabyte. We built several data warehouses in the size order of hundred gigabytes and we are well aware that – for multi terabyte databases – some of our solutions would be critical. Nevertheless, we know there are many small and medium sized companies that need great data warehouses to grow up and be able to understand better their business. In these cases, you can consider developing some of the solutions we adopted.

What we are trying to explain in this paper is a way of thinking to BI solutions with the mind completely open. Sometimes we will use a tool for what it was not designed for, just because we discovered that – looking at it from another perspective – it could make something great.

People in the BI community need to share experience in order to let others discover more and more ways to achieve the great results BI is producing over these years. This is our personal contribution to this wonderful world of research activity.

We hope you will enjoy reading as much as we enjoyed writing and we really hope that, at some instant during the reading, you will close it and think: “that is interesting, I can try it with my customer”.

Our best reward for this work will then be your thoughts in that one instant.

Architecture of a BI solution

The goal of the chapter is to provide a solid basis for discussions. In this chapter we will recall standard terminology used in BI solutions and describe the main problems that we have to face when designing a BI solution.

Please note that we are talking of a data warehouse solution, not a simple database nor a specific cube.

It is misleading to think that a data warehouse solution consists of only the final database used by the customer; we will speak about a "Solution" as we are interested in talking about the whole process of building that final database.

During the descriptions we will sometime stop to think more about a specific definition. Remember: the goal of the paper is not that of describing a methodology. We want to explain what consideration will lead to the final architecture we use. So be prepared to read a simple definition and then a long set of considerations that will change it. At the end, it will be clear why the definition is or is not correct and you will be able to have your own opinion about it.

The real goal of the paper is to let you follow the reasons that lead us to the final solution, not just of describing it.

Classification of BI solutions

In the whole paper, we will sometime refer to the size of the BI solution and give some advice for small or medium BI solution compared to big ones, stating that a specific hint may or may not be viable for that specific size of the solution.

We want to define a classification of solutions based on their size and complexity. It is clear that size leads to complexity: even a simple solution made up of a few dimensions will become complex if it will have to handle several billions rows in the fact table. Normally the processing of data warehouses happens during the night, in order to have all data available for the users in the morning. This may not be your specific case but, when we speak of something happening overnight, we think to something happening during the ETL phase of the data warehouse creation.

- **Small BI solution**

  We can rebuild a small BI solution each time because the entire ETL phase will consume a few hours of computation and does not need to keep complex history of changes in the dimensions.

  If you are developing a small BI solution, you normally will recreate the whole database each time the ETL phase starts, producing a fresh database each night. This simple method of creating the data warehouse is very convenient, when applicable, because it will highly reduce the complexity of handling the database. We can apply any change without worrying about the old data just because there is no old data (where old data means data already processed and imported in the BI solution).

  Even if this approach might seem strange, we find that many small or medium size companies have data warehouses whose elaboration will last no more than six, seven hours of computation. Using this pattern leads to simple solutions very easy to handle and with a very high return of investment.

  We will sometime refer to small BI solution as "one shot solutions" because they can be built with "one shot" of ETL computation.

- **Medium BI solutions**

  If we need to trace history of changes in the dimensions, (i.e. you have some SCDs in the solution) the one shot solution seems to be no more a viable way. This is partially true.

  If the complete solution can be rebuilt nighttime, we will always try to keep it as a one shot solution, computing it each night. If, for some reason, we need to maintain the history of changes of some specific dimensions, we think that it is easier to store the different version of those dimensions in a persistent database, reloading all the facts and dimensions each night.

  Doing this, we maintain the advantages of the one shot solution adding a slight complexity for the storage of different versions of the dimensions, but we still have full control on any database change since we can rebuild the database during one pass of computation.

- **Large BI solutions**

  When the size of the fact tables becomes so big that "one shot solution" is not a viable one, then the only choice is that of loading it incrementally each night, applying changes to the dimensions and adding new facts to it.

  When a solution is very large, the overall complexity raises of some levels. We cannot rebuild the database easily, nor add attributes smoothly to the solution. Any change will require a high level of attention because all operations will consume CPUs and disks to a high degree.

  Moreover, some of the techniques described in this book would lead to poor performance because the size of the tables is so high that you eventually need to use simpler techniques to handle them.

Our experience is that the majority of solutions in the market are made of small or medium sized solutions. Whenever we start a new BI solution, we always need to understand in which scenario we are because the architectural choices will highly depend on the size of the BI solution.

Nevertheless, even for large BI solutions, we will be able to apply several of these techniques to smaller fact tables. It is important to remember that we can always create a smaller fact table from a big one changing the granularity of some dimensions (time is one of the best candidates in these situations).

Actors

Before starting to dive into technical details, let us spend some words to describe who the actors that work around a BI solution are.

User / Customer

In the whole paper, we will refer to the "user" as somebody who wants to analyze information using a BI solution. This kind of user is not the only one, there may be several different users for a BI solution, but the analyst is the most challenging one and the most interested in the advanced topics described in this paper.

The user is normally a DSS (Decision Support System) analyst who is interested in making the best with the data that he has to handle every day. He knows that data is available through his IT infrastructure and wants an easy and effective means to analyze and aggregate it.

In our experience, the user is somebody that is very Excel addicted. In the past, he used Excel to compute huge spreadsheets. He is sometimes a guru of Excel and he knows how to format data and present them in a fashionable way.

Before the advent of BI, he gathered data from reports, queries and any kind of source. Our goal, as BI analysts, is that of providing him with the data needed for reporting, taking them from the BI solution we are building. The data has to be fast and accurate.

As our user is very Excel addicted, we will try to make him use only Excel or any kind of very simple client. We have always to remember that the user does not know (and should not learn) MDX nor any kind of advanced query technique. In our opinion, a BI solution is a good one if the user is completely satisfied by using only Excel or any other basic client for each of his/her queries.

If we can express a complex query only using MDX, probably we could improve the dimensional design to simplify its usage. Of course, this is our opinion, but we believe this so much that it will permeate the whole methodology.

BI ANALYST

Anybody who is reading this paper should be a BI analyst. A BI analyst is the professional who handles the complexities of building a BI solution.

Beware that the term "analyst" in the BI world is much different from the same term in the software world. In the world of software, you normally find the figure of the "analyst" as one opposed to that of the "programmer". We often think to the analyst as somebody who speaks with the customers and write a huge documentation about the requirements of specific software, translating the user language into a more technical one. A plethora of programmers will write the software, testers and users will test it and eventually the software will start to work. This is the well-known life cycle of software.

A BI Solution is not software. The lifecycle of a BI solution is completely different from that of the software. In the BI world, the analyst is often the same person who will develop or at least actively participate to the production of the solution for the customer. He needs to know very well all the pro and cons of each single decision to migrate data from the OLTP system to the data warehouse.

There are many good reasons for this particular mood of the BI analyst, the first and most important one is that a BI solution is completely based and driven by data, not by user requirements. The needs of the customers are different depending on the specific kind of data that they want to analyze. Moreover, the particular tools that will be used both at the client and server side will definitely influence the decisions. A good BI analyst needs to consider all these information.

Furthermore, it is very rare the case in which a BI solution starts when the complete requirements have been collected and – when it is the case – it will very often fail because it has become too complex. User requirements change very quickly. When the users start to look at the data, they always discover something interesting to analyze that was not so clear at the beginning. This is not a failure of the analyst; it is a rule in BI solution development.

A good BI analyst need to change very often the design of the solution in order to produce a good solution for the users and need to handle very rapidly any change of idea. BI users do not change their requirements because of the weather. The real situation is that they did not perfectly know them at first as they do not know what they will need tomorrow. This is the way a DSS analyst look at the world: discover new information to dive in day by day.

In our opinion, a good BI analyst is somebody who is always ready to change his way in order to follow the user needs. A BI analyst is addicted to satisfy the always-changing user requirements. A BI analyst is definitely different from a software analyst.

As we are BI analysts, the reader of the paper should be a BI analyst; we will often refer the BI analyst as "we", meaning both you and us. So, when you read "From our point of view" we really mean "from the BI analyst point of view".

We hope that it will be both yours and ours.

MICROSOFT BI SUITE

This paper is about solutions created with Microsoft SQL Server suite. We will not provide any description of the core functionalities of SQL Server. We will simply use them.

In this chapter, we want to recall the software that you need to know and their usage for the purpose of this paper.

- **SQL Server**

  We will use SQL Server as the database engine that handles all databases in a BI solution. Since this paper is on dimensional modeling, it is not mandatory to use SQL Server as the database. Nevertheless, we will describe some hints that are specific to the SQL implementation of Microsoft SQL Server but the concepts described in the paper are still valid for any database engine.

- **SQL Server Integration Services (SSIS)**

  We will build the whole ETL pipeline using SSIS. The examples and several optimization considerations rely on the engine of SSIS. There is no specific need to adopt SSIS as the ETL engine even if all the considerations about the ETL will rely on SSIS.

- **SQL Server Analysis Services (SSAS)**

  SSAS is the multidimensional modeling engine extensively used and described throughout the whole paper. All the examples, considerations, optimizations and modeling technique described in this paper will work only on SSAS.

  The goal of the paper is not that of providing a generic multidimensional modeling course but that of describing advanced dimensional modeling with SSAS.

  We will not cover all the aspects and functionalities of SSAS. Our experience and the models we will provide do not cover:

  - Real time analysis
  - ROLAP

  The models described in this paper cover the more classical use of SSAS for the analysis of complex data warehouses in order to model the business aspects of a customer.

- **Excel**

  We will use Excel as the main client for OLAP cubes.

- **ProClarity**

  Several customers use Proclarity as a convenient BI client. Sometimes we will use the desktop edition of ProClarity to illustrate particular situations even if – for all the examples – we will try to use Excel.

ARCHITECTURE

There are many description of what a BI solution is. We do not want to go into deep or formal definition here but just recall what the objects that a BI solution should handle are.

The reader should already have a good knowledge of what a Kimball solution is. In this chapter, we only want to give names to objects in order to be able to write of them.

During the description, we will make some consideration that will be an introduction of the main arguments of the whole methodology.

COMPONENTS OF A BI SOLUTION

In this section, we want to define the main components that are normally involved in the building of a Data Warehouse solution.

We will refer to the whole schema of processing as the "ETL Pipeline". When we speak about the ETL pipeline, we refer to the whole process that starts from the OLTP system and finally produces the OLAP cubes and reports used by our customer.

The goal of this chapter is to build the ETL pipeline incrementally. At the end, we will have a clearer view of what components constitute this pipeline.

Source OLTP database

The source OLTP (On Line Transactional Processing) is the main source for a data warehouse. It can be a legacy system, CRM, accounting manager or – in general – any kind of database that users adopted in order to manage their business.

Sometimes the OLTP may consist of simple flat files generated by processes running on the host and that we will read during the ETL phase. In such a case, the OLTP is not a real database but we can still transform it importing the flat files into a simple SQL database. Therefore, regardless of the specific media used to read the OLTP, we will refer to it as a database.

Let us view some of the most important characteristics of the OLTP system

- The OLTP is normally complex software that handles information and transactions but, from our point of view, the OLTP is simply a database.

  We do not normally communicate in any way with the software that populates the data we want to analyze, our job is that of loading the data coming from the OLTP, integrate it into the data warehouse and the let the users navigate into the data.

  As the data resides into the database, we think to the OLTP as being composed of only the OLTP database.

- We cannot make any assumption on the OLTP structure.

  Somebody else has defined the OLTP system and probably is currently maintaining it. Our task, as BI analyst, is to reveal interesting information hidden in the OLTP database. The option of changing anything is normally prohibited, thus we have to take the OLTP system "as is" without the conviction that it could be made better.

- The OLTP contains several data that are not conformed to the general rules.

  Normally in the OLTP system, you will find historical data that is not correct. This is a rule, not an exception. A system that runs from years has very often data that is not correct and never will be.

  The BI solution need to make the cleansing of the data and the rearrangement of information. Usually, it is too much expensive to perform these corrections in the old data of the OLTP system.

- The OLTP has often a poor documentation.

  In our experience, very often the OLTP has a poor documentation. The first (and hard) work of the BI analyst is that of creating a good documentation of the system, validate the data and check it for any inconsistency.

Sometimes we work directly with the OLTP system in order to read data but more frequently, and depending on the size of the Data Warehouse, we try to make a SQL Server database that contains a mirror of the same data.

The advantage is intriguing: being able to do something prohibited like to alter the database creating indexes, views and any other kind of tool that will help us to work smoothly. This need will lead us to define a new object in the BI solution that we call "Mirror OLTP" i.e. a mirror copy of the OLTP database.

From the logical point of view, there are no differences between the OLTP database and the mirror. From the technical point of view, the difference is dramatic, as we will have full control of the mirror database.

Clearly, creating a mirror comes at a price: the time to construct the OLTP mirror. Sometimes we cannot create this mirror database because there is not enough time to query the OLTP and duplicate its data into a mirror. In this case, you will have another option, described later: the OLTP metadata DB.

In our experience, the creation of an OLTP Mirror has always been a successful strategy for several reasons:

- We use the "real" OLTP for a small amount of time during mirroring. Then we free it, so it can do any kind of database maintenance needed.
- The ability to modify the keys, relations and views of the mirror lead to simpler ETL processes that are cheaper both to build and to maintain over time
- The ability to build particular indexes in OLTP Mirror could improve performance of other steps of ETL processes, without worrying about index maintenance on the original OLTP
- The mirror is normally much smaller than the complete OLTP system as we do not have to load all the tables of the OLTP and, for each table, we can decide to mirror only a subset of the columns

The following picture shows the situation:

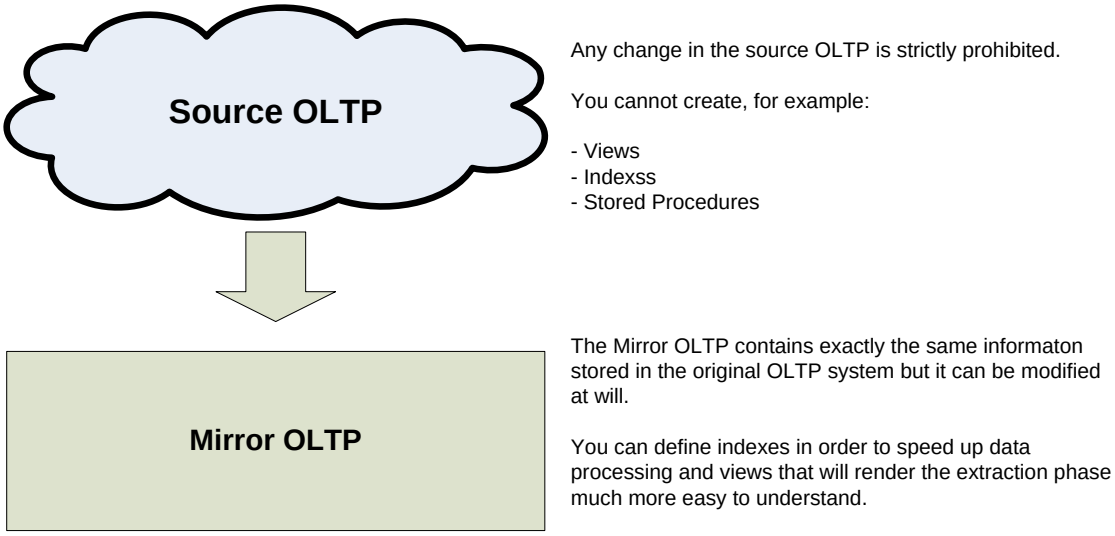

Figure 1 – (IMG 0014) Source and Mirror OLTP

The ability to define simple views in the Mirror OLTP leads to a much easier implementation of the subsequent ETL phase and should not be underestimated.

The structure of the OLTP, especially when based on very old DB2 databases, is very difficult to read and understand. Hiding this level of detail using views leads to a much easier comprehension of the whole process, from both the user and the analyst point of view.

Even when we cannot do the mirroring of the OLTP database due to timing reasons, it may be interesting to define an intermediate database where you can define these views. We call this database the *OLTP metadata DB*. The database – in this specific situation – will not contain any data but it will contain very useful metadata about how we read the information in the OLTP database and what is the source of any further defined object.

Moreover, by making this information available in a database, you will be able to share them with any other developer who can read a DB, reducing the need for verbose documentation.

In the following picture, you can see the difference between the generation of a mirror OLTP and an OLTP metadata database:

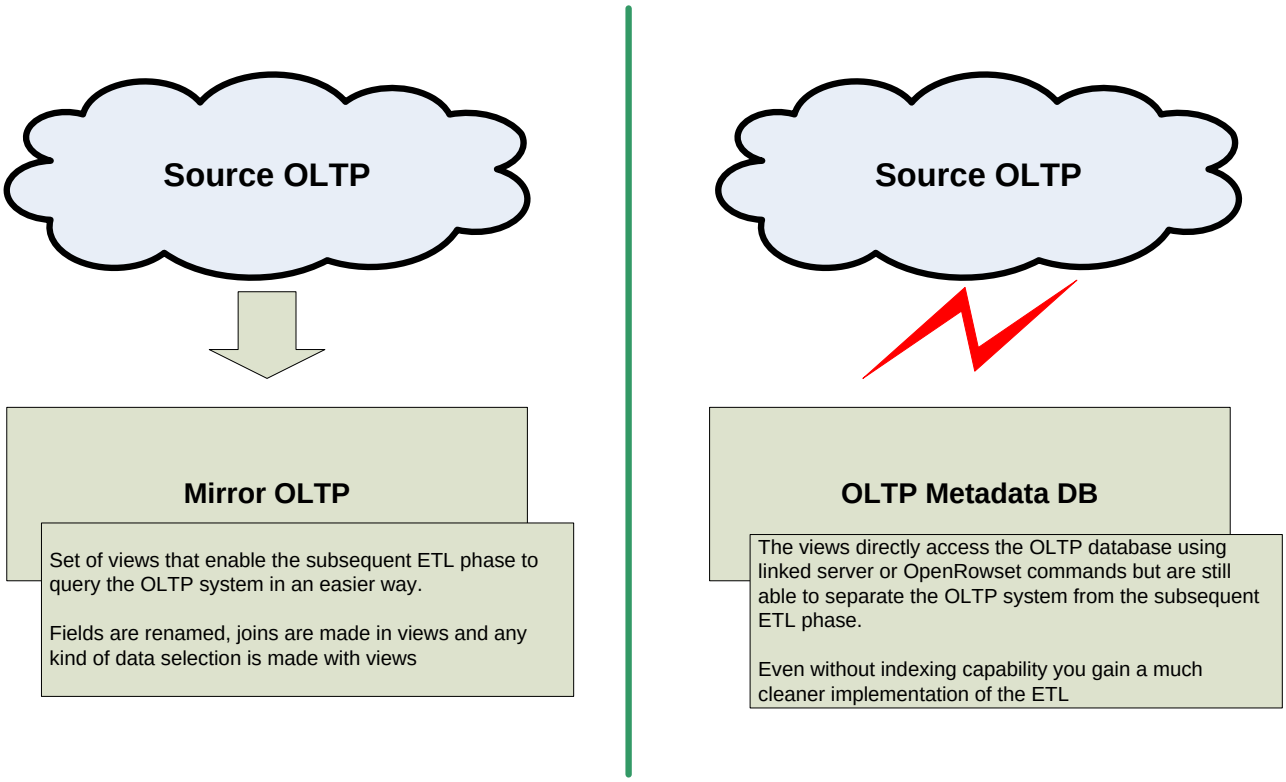

Figure 2 – (IMG 0015) Mirror and Metadata OLTP

As another big advantage in the creation of a mirror or metadata OLTP database, consider the great functionality of SQL 2005 schemas. If you associate each table or view from the OLTP into its appropriate subject area, you can use reasonably chosen schemas to separate the source OLTP in subject areas.

This will lead to a very easy way to understand database and will give you the great opportunity of viewing exactly which data from the OLTP participates to which subject area in the final data warehouse.

During the whole paper, when appropriate, we will use the AdventureWorks database as the OLTP system. Sometimes AdventureWorks will not have the complexity needed for the description; in that case, we will provide you some sample database to use in order to understand the problem.

The reason for this choice is that this database is a good example of an OLTP system, it is quite complex, the reader is probably familiar with the tables and concepts inside it and all examples will be easily executed on the reader's computer without any modification.

As AdventureWorks is already a SQL database, we will not have the necessity of mirroring it. However, in order to get a cleaner view of the queries, we will generate a metadata DB that will make the necessary queries to the OLTP database. This will result in a more readable ETL phase.

Please note that AdventureWorks has a very clean structure. It works on SQL 2005 using all the features that are available in this DB. In the real world, things are much different. For example, you might have a DB2 database with field and table names so cryptic that the ability to rename them is welcome.

Configuration Database

The OLTP system contains the main data that is useful to create the Data Warehouse. There are – however – several kinds of data that are not contained in the OLTP system. It would be impossible to describe them all because they are very customer specific. For example consider these:

- **Budgeting**: Small companies do budgeting with Excel spreadsheet and not with an OLTP system. We will need to read those files into the configuration database and then use the data to generate the specific data marts.

- **Meaning of codes**: sometimes customers give to specific fields different meanings, then they "translate" values from the OLTP system into more meaningful values (i.e. "code 01" in the category field means "corporate customer"). This information is really configuration and should be stored in the configuration database.
- **Code remapping**: you normally have one OLTP system but… what happens when there is more than one? We will need to match information coming from one system against the other ones. We will need tables that let us make these translations easily. Moreover, the users should be able to update these remaps.

In our experience we found than each data warehouse we had to build has several of these sources of information. They may be in very different format but we normally refer to them as "Configuration" as their reason of being is to store "configuration information".

You may wonder why there is the need to store the configurations into a database that is different from that of the data mart or the staging one. The reason is very simple and is that configuration data need to be persistent and the users might change them. We will rebuild the DataMart each night while the staging area is a technical area and we do not want users to mess with it.

Moreover, there is a problem with keys: in the data marts, you normally use surrogate keys for any kind of relation, while the OLTP system has its own set of keys (we refer to them as natural keys, they are also known as application keys). In the configuration database, you always have natural keys.

Let us see how the introduction of the configuration database changed the picture of the data flow:

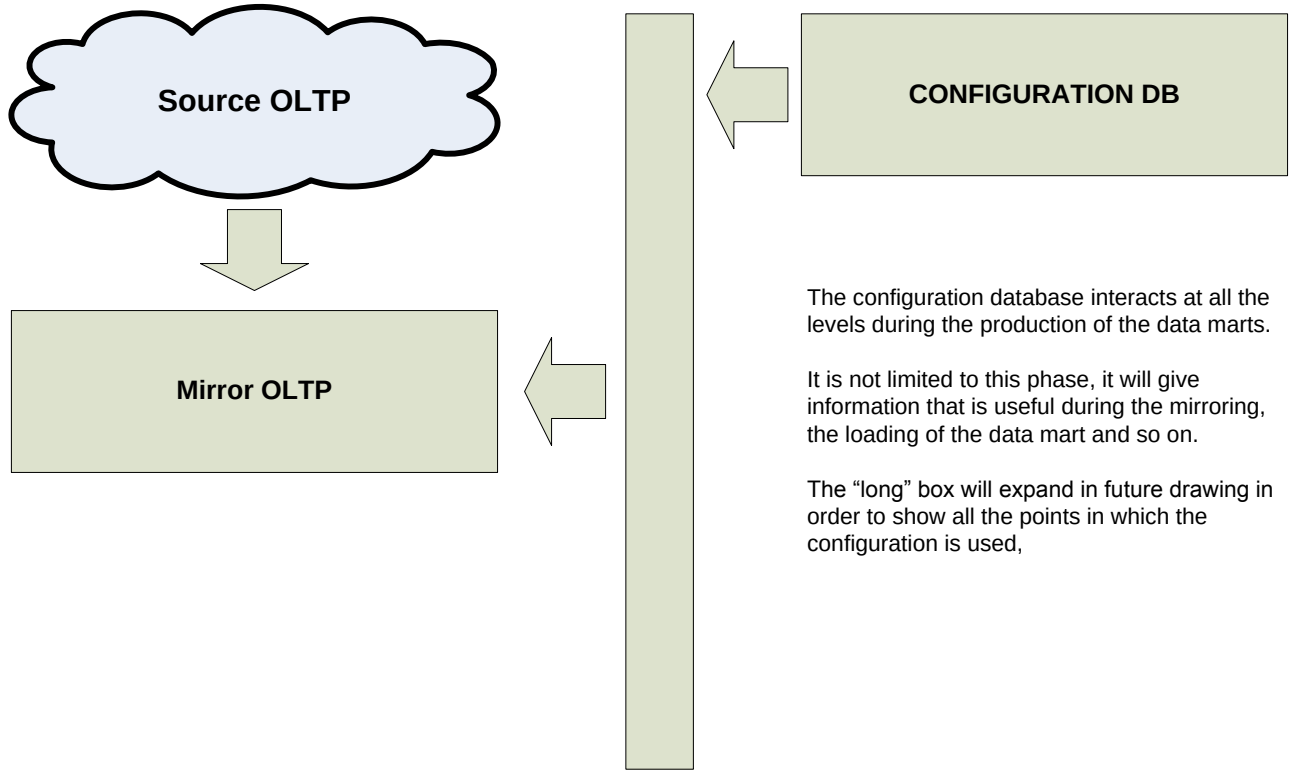

Figure 3 – (IMG 0016) Configuration Database

In the design of a clean data warehouse, the configuration database is mandatory and we should reserve enough time to design it correctly. Moreover, the end user should have a simple interface in order to update the configuration database. You can use Access forms, a simple web interface or a client/server application, there are many choices here. The important thing is that the user should be able to update any configuration by its own, without ever require a technician to do it.

The configuration database is normally stored in a simple SQL Server database.

Staging Area

During the ETL phase, we will need many temporary tables to go on with the processing. These tables are useless at the end of the process as the user will never see them, but are very useful to be able to carry on the ETL process.

The Staging area is where the ETL stores this kind of information. We need to separate the staging area from any other database simply because the staging area is a strictly technical domain. We often delete, add or modify tables following the life of the ETL packages. The ETL programmers need a place where they know nobody else will ever question what is happening: this is the staging area.

One question that arises quite often is "can we store persistent data in the staging area?" In our opinion the answer is categorically no. The staging database is a temporary store for tables and structures that are useful for the ETL phase. The data in the staging area does not persist over time simply because it is useless to do it.

If you have any kind of persistent data then this is configuration data or data warehouse data. It does not matter from where this data comes, if it is persistent then it is not staging data.

The matter is much more philosophical than technical but it is still very important. If we need to persist some data into the staging area, then they probably are:

- **Historical information**, e.g. some data needed to categorize dimensions. This is data warehouse data and we need to treat it in its own way. If you leave this information in the staging area and do not expose them into the data warehouse then you are missing the capability, in the future, to explain "how" we generated some data.

  The user will never modify this information so it is not configuration data. However, given that it is not even staging data, its place is that of the data warehouse.

- **Configuration values,** e.g. values useful for building some dimensions. These are definitely configuration data; the user must be able to update them.

- **Bookkeeping information,** e.g. days before we will delete data from the data warehouse. These also are configuration data and should never be stored in the staging area.

The complete picture of the ETL phase is becoming more complex with the introduction of the staging area:

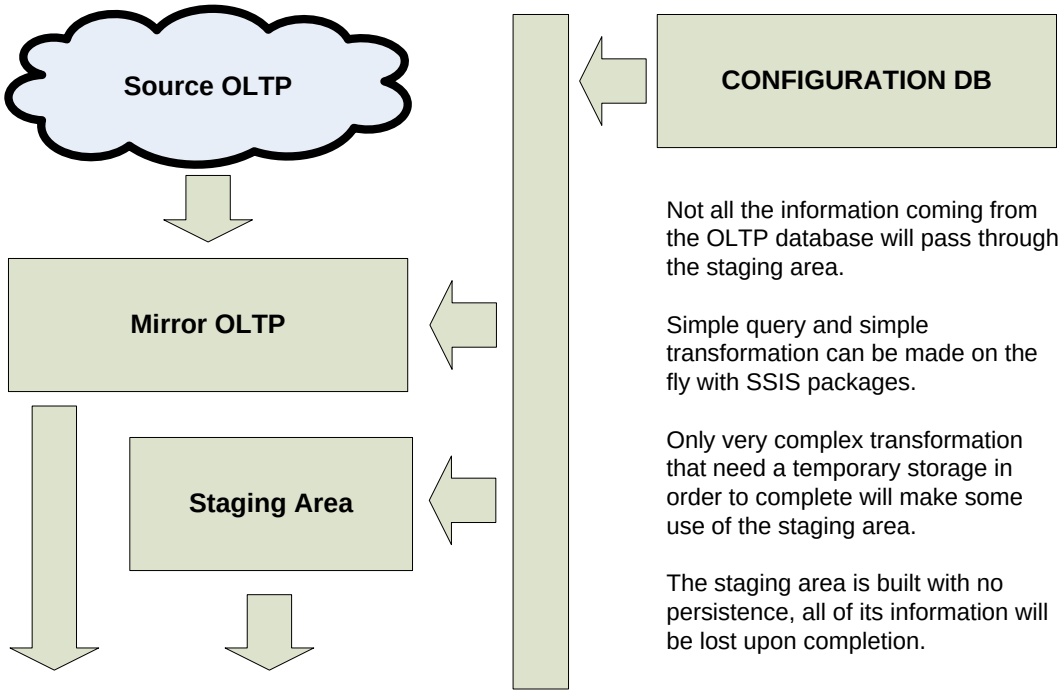

Figure 4 – (IMG 0017) Staging area

As a rule of thumb, we build the ETL phase in a very simple way: it must work with a completely new staging database without requiring any kind of data to be present. Over time this technique lead to a very clean design of the whole ETL phase, even if at first it may seem useless.

Data Warehouse

The data warehouse is the database that will contain all the tables, views, procedures and code that our customers will use for their daily reporting activities.

We will arrange the information in the data warehouse so that they are easy and fast to query. While the OLTP database is designed with updates in mind, the data warehouse is designed thinking to queries, not updates.

As with other concepts in the BI world, there is still no full agreement on what a Data Warehouse is. According to Kimball, the data warehouse is the union of all the data marts. according to Inmon, the data warehouse is a relational model of the corporate data model. In our vision there cannot be one clear definition of a data warehouse, the content of the data mart highly depend upon the complexities of the specific BI solution we are building for the customer.

Sometimes the Kimball definition is correct: after the analysis process, we end up with several data marts and we can unify them under the bus structure defined by Kimball.

Sometimes we will need to build a relational model in order to coordinate the building of several data marts because we discovered that single parochial data marts do not coexist very well under the bus structure. In this case, we will start upgrading our data mart to a structure that is more Inmon like.

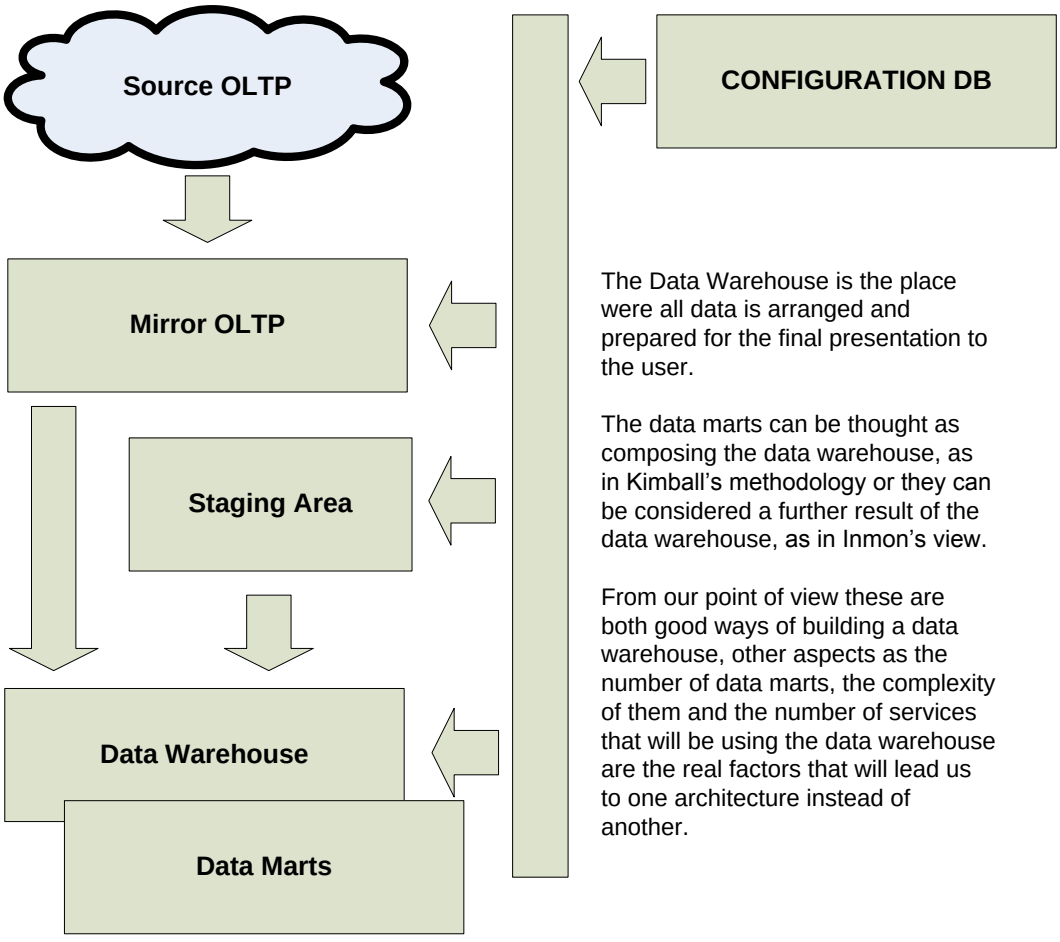

Figure 5 – (IMG 0018) ETL up to the data warehouse

We will return later on the distinction between a data warehouse and a data mart. Up to now, what is important to note is that the configuration database is still operating during the construction of the data

warehouse and the data marts. This will explain why we put so much stress on the necessity of a specific configuration database. Its range of interaction during the whole process of ETL is so wide that it is one of the most important databases in the whole project.

Moreover, from this stage, we will start having data marts as the output of the last box, regardless of the fact that they are a part or a product of the data warehouse.

Data Marts

One question is still open: what is a data mart? The term is vague, ask 100 persons what a data mart is and you will get 100 different answers, more or less all correct, but no one will be "the correct one".

We do not want to give a definitive answer to the question; it would be useless and generate many philosophical discussions. Nevertheless, it is important to define what a data mart is for what concern this paper.

We think to data marts as a set of star schemas and conformed dimensions related to a specific departmental view of data or – in some cases – a specific subject area. Kimball's definition is somehow generic: "a logical subset of the complete data warehouse". Moreover, Kimball's definition is not based on the structure of the data warehouse but instead on the departmental view of it. He says that we should consider data mart as a project by itself. However, this concerns people and money, not the internal structure of the data warehouse.

Let us try to analyze what does "logical subset" mean. We can interpret it at least in two ways:

- A subset that is not a silly one. This is clear and (hopefully) easy to accomplish for a good BI analyst, if we give to "logical" its grammatical sense. We can take for sure that Kimball meant at least this.
- A subset that is not identical to the physical model of the database. This interpretation is very clear and may be the correct one but it is an interpretation. We are taking the mean of "logical" in contrast with "physical" and we are deciding that there is the possibility to build a physical model different from the logical one.

Even if the only "safe" interpretation is the first one, we grant for good the second one. We define a data mart as an autonomous subset of the complete data warehouse that is functionally defined by facts and dimensions. We also state that the structure of a data mart might be different from that of the data warehouse. The data mart is composed of fact and dimensions that will be derived from the data warehouse but might be slightly different from the original ones.

We will see that the distinction between logical and physical visions is very important in our model of data warehouse. At the end of the process, we will have data marts that can show to the users some data but rely on physical structures that can be quite different.

OLAP Cubes

We will use the single data marts from the data warehouse to generate OLAP cubes via SSAS. The user finally uses OLAP cubes in order to bring its data to Excel.

First, we need to define what a cube is. When we speak about a cube in this chapter, we do not want to stress the technical side of the cube but the functional one. A functional cube is an SSAS project that might contain several cubes and dimensions but share the same dimensional structure between them all. On the other side, you might have several projects that rely on the same relational database but expose different dimensional structures.

We might have introduced another term to define a functional cube but, from the context, it will always be clear when we will speak about "functional" cubes and technical ones and, for the sake of simplicity, we decided not to introduce another complex term.

In this chapter, specifically, we will always refer to cubes as functional ones because we are not speaking about technical stuff but stress the methodology point of view.

There are several questions about how we will relate data marts to cubes. Let us see some of the most important:

- **Is the relation between a data mart and a cube one to one?**

  The answer, usually, is no. We will use one single fact table in several ways to produce different cubes. We will use main and conformed dimensions in almost any cube while we will use specific dimensions in only one particular cube.

- **Is a cube originating from more than one data mart?**

  In this case the answer is often no. Even if a single data mart will generate more than one cube, a specific cube will contain only the information coming from one data mart.

  Conformed and shared dimensions are an exception but, as we are sharing them, we cannot think to them as belonging to one specific data mart.

- **Is the usage of all dimensions the same in all cubes?**

  Unfortunately, the answer to this question is no. Main shared dimensions have the same usage over the whole corporation structure. However, there are very often small or rarely used dimensions that we will use in different ways.

  It is always possible to start a project and have all dimensions behave the same but, as users start to request updates, through the time you always end up with different usages. This is not an analysis error. This is something we have to face, as it is inherent to a BI solution that we always need to tailor to the users, not the analyst.

  Think – for example – at the date dimension. It will normally have a range of several years at the date granularity. Nevertheless, the user – in order to get data faster –soon or later will request you to reduce the range of the date dimension to only the dates available for a certain cube. You cannot remove irrelevant dates from the table, so you will end up with different views that will generate different date dimensions, at least one for each cube, sometimes more than one. All these dimensions will share the same structure but will get their data from different sources.

  Junk dimensions have similar issues, but we will return on them later.

When you build a small/medium sized data warehouse, you will end up with at least five different cubes that will share most dimensions and have their own. As the relationship between data marts and cubes is not one to one, we will have to put some intermediate structure between, in order to be able to decouple these structures. Even if SSAS does not enforce this behavior, we will do it later.

Reports

Reports are simply another way of getting data out of the data warehouse. You can build them on top of the OLAP cube by writing MDX queries or you can use SQL statements to query directly the data marts.

We need to check some rules before building a report:

- **A report does not need SQL or complex MDX queries**

  The best solution to develop it is not to do it. If no MDX is required then the users should be able to build the report with Excel or any other OLAP client. The best solution you can give them is the instructions on how to do it. They will be happier as they can change the appearance or the content of the report by their own without requiring any further assistance.

- **A report requires SQL statements**

  You can try to understand better if something is missing from the OLAP cube. You may build the same report by adding some degenerate dimensions or some attributes to the existing dimensions, and this would be the best solution. If it is still not feasible, then querying the SQL database is surely an option with Reporting Services.

- **A report requires complex MDX code**

  You can investigate to discover if some calculated members are missing from the solution or if the data model lacks of some feature that would make the user able to build its own report. If the user needs to write code to get – for example – a distinct count of a specific set, you may want to include the set and the distinct count measure in the cube, in order to let him use these values in reports. In specific cases, the best choice will be that of building the reports with Reporting Services, but in our experience we found very few of these cases where this was really necessary.

Anyway, always remember that users that are able to build reports by themselves will be happier and will have a much deeper understanding of the BI infrastructure. This will lead to a better cooperation between the analyst and the users in the future.

Client Tools

Up to now, we have discussed of databases. What we need to keep in mind is the fact that a database is a repository. The customer will not see the databases. He will use a client tool to navigate the data and all his requests will start from their experience with that tool.

If we want to satisfy our customer, we need to be very aware of this fact: the customer client tool will need to use smoothly our data warehouse. It would be useless to build a complex infrastructure that is not queryable at all because of limitations in the client tool.

As we will see later, there are some idiosyncrasies when using specific client tools. The BI analyst needs to know how a client will interoperate with the BI solution he is designing, otherwise the final product will lack the simplicity and integration that the ultimate goal of the solution itself.

Operational Data Store

In the Inmon's view of a data warehouse there is a data structure called the ODS (Operational Data Store). ODS exists to answer queries in a very fast way and contains pre-calculated measurements, often called profiles. The contents of the ODS will vary from solution to solution. Nevertheless, if the data warehouse you are building involves other systems that will expose information from the data warehouse, the building of the ODS can be a lifesaver.

The ODS contains data structures that are ready to be queried by other subsystems. These subsystems are made of software, not humans. Users will interact with that software and get their answers, users will never have a direct access to the ODS. As the query that these systems will do are predictable (in contrast with user defined queries, where you do not know exactly what the user is searching for), you can compute them once and let them available in the ODS.

The ODS gathers its information from both the data warehouse and the final cubes. The cubes are typically faster in answering questions when compared with the data warehouse, simply because OLAP cubes may have already computed the values needed by the ODS using aggregations.

The ODS is not very common in small data warehouses, but we included it in order to give a clearer idea of what a complex data warehouse may be.

ODS is not always built as a separate system. If we build reports with Reporting Services and use its scheduling and caching features, we are building a sort of ODS. In fact, we are clearly creating a cache of pre-computed data that will be made easily available to the users without the need of on line computation.

We do not want to go into a deeper description of what can be done with an ODS. If you need it in your BI solution, you can find description that is much more detailed in specific books. From our point of view, it is important to consider that an ODS might be present in the final architecture of the BI solution. The reason for which it is important to consider it is that ODS is a VIP user of the whole BI solution. If it is present, then it needs to be served well because its results will be necessary for the correct operation of other pieces of software.

Complete architecture

In this picture, we show the summary of our view of the architecture of BI solution. It will merge all pieces described up to now into a final vision that shows all the items that we will need to study carefully during the process of building a BI solution.

As you can easily see, when we think about a BI solution we think to something that is much more complex than a single database or OLAP cube.

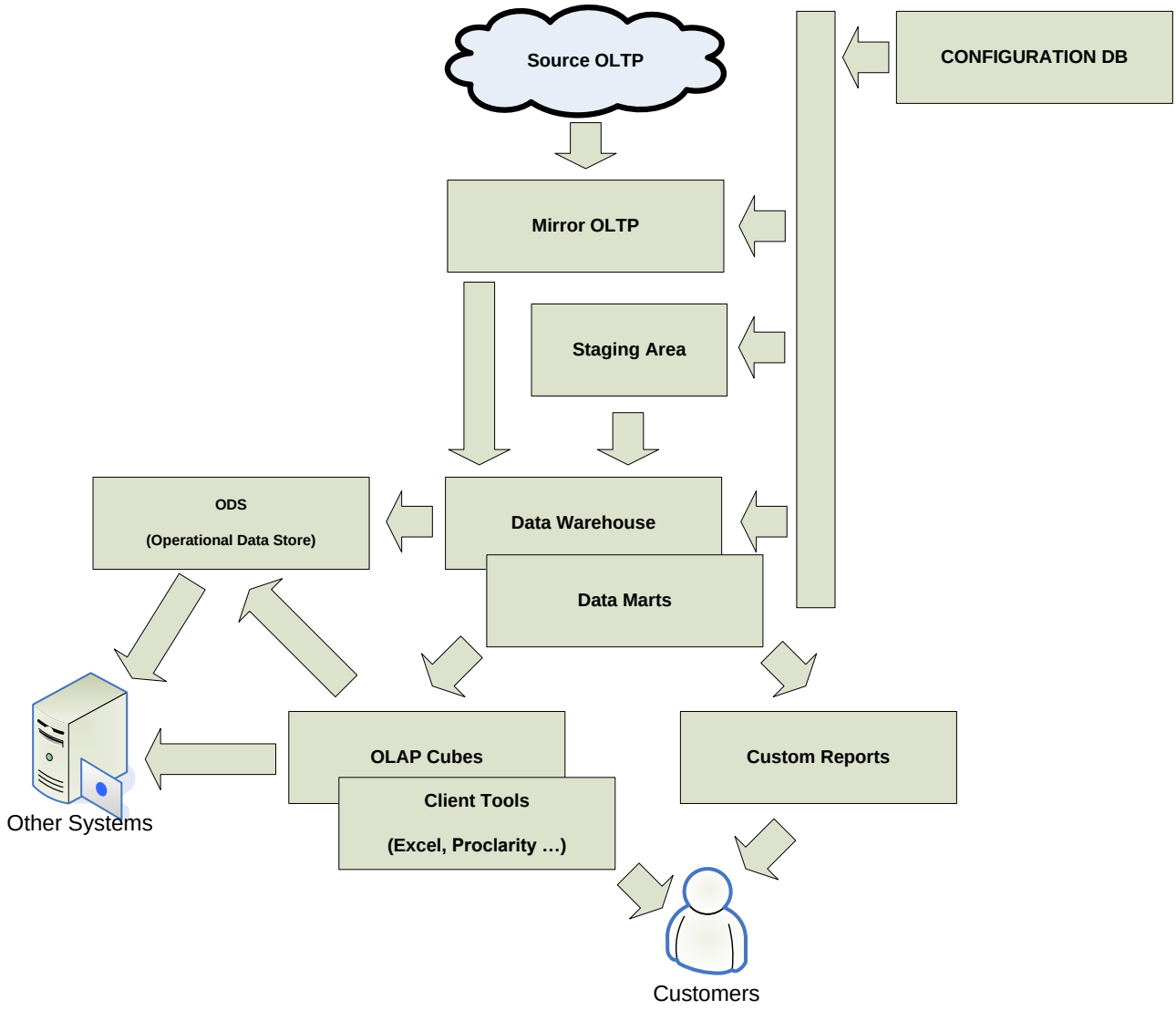

Figure 6 – (IMG 0019) Complete ETL pipeline

Let us review it in more detail:

- **Source OLTP**, when possible, will be mirrored in the Mirror OLTP or, if not feasible, will pass through a Metadata OLTP database.
- **Mirror OLTP** and **Configuration database** both contribute in the population of the staging area, where data will be transformed to enter the data warehouse. These databases will start the real ETL pipeline.
- The **staging area** lives by itself. During staging processing we will cleanse and reorganize the information in order to meet the characteristics needed by the data warehouse. Data can or cannot pass through the staging: this depends on the specific architecture you have designed.

- The **data warehouse** receives data directly from the Mirror OLTP and/or from the staging area during the ETL phase.

  We can consider the data marts as constituting the data warehouse as in Kimball's bus architecture or as being derived from the data warehouse through another ETL as in Inmon's view.

  In our opinion both these views can coexist, there is no real need to consider the two architectures as being completely different. There may be cases where we do not need to build any relational data warehouse but, if necessary, we can consider it as being the foundation for fact tables and dimensions. The BI analyst should take the final decision and, in a well-designed architecture, that decision can change over time to accommodate a more complex structure.

- Fact tables and dimensions are arranged in **data marts**.

  A data mart can contain global dimensions and locally created dimensions. There is absolutely no need to consider the dimensions strictly as tables and a relation one to one between tables and dimensions.

  In the data mart area, you can rearrange dimensions in order to get the best result for a specific data mart. What is important is that each data mart relies on fact and dimensions previously defined.

- Each data mart will produce some **OLAP cubes** where each cube can contain one or more measure group related to data mart specific dimensions.

- We use the data marts to produce **detailed reports** that expose specific information in a more detailed form than the one exposed by OLAP cubes.

- If present, the **ODS** will gather data from the data warehouse, the data marts and from the OLAP cubes.

  It will pre-compute profiles for other subsystem. As these subsystems have a predictable behavior, the construction of profiles will give answers very fast.

  No user will ever interact with the ODS, the ODS exists to serve only software.

- The **users interacts** with OLAP cubes and/or reports and gets the data out of the data warehouse using custom tools for the OLAP cubes and (usually) a simple web browser for custom reports

De-coupling single steps

As you have seen from the construction of the final picture of a BI solution, we need to carry on several steps before completing the whole way from OLTP to OLAP cubes.

We need to link each step with the subsequent by some kind of dialogue. In order to define a dialogue, we need to define the protocol to use and then clearly describe what kind of data can be moved from one step to the other. This is what we call de-coupling the steps. Declaring clearly which communication media to use between different systems is a widely approved technique and – in our vision – each step represents a different system.

The need for de-coupling rises for at least two very different reasons:

- **De-couple to let independent development**

  If two systems are de-coupled then you can apply changes to one of them without having to worry about the interference with the others. All what you need to do is to be sure that the dialogue is still intact and has the same meaning as before. When you guarantee that, you are free to adopt any change.

  As changes occur very frequently in the BI world, we need a robust system that will let us apply those changes rapidly and in a safe way.

- **De-couple to create documentation**

When you have to make a change, you want to be sure that everything will continue to work smoothly after the update. If you define the communication media that links together the two steps, you will provide the documentation needed to make people work safer.

This documentation should be available at a glance to the person who will make the change. Therefore, putting it into a very complex Word document is never a good idea. Database administrators and SSIS package writers want to get the updated information.

These two reasons will lead us to decide where to put this de-coupling means. It is not time to go into deeper details about that. What is important now is to understand that the existence of different steps involves the necessity of de-coupling. Moreover, this de-couple operation needs to be the most evident one.

The careful readers will already have noticed de-coupling in the definition of the views in the OLTP mirror database, They are probably thinking that we are going to use a similar approach to de-coupling between the subsequent steps. Since they are careful readers, we can assure them that they are right even if – in the subsequent steps – we will have more choices for de-coupling. None of these choices will be easy and intuitive as views.

THE KIMBALL METHODOLOGY

In this chapter, we will make a short description of the Kimball methodology adopted all around the world as the most effective way of building a BI solution. Kimball describes it as the "Bus Architecture".

Kimball's Bus Architecture is not the only paradigm used to create a BI solution, there are many others and we do not want to start a religious war about it. We believe that the Kimball's approach to the definition of a BI solution is very effective even if, in some situation, it lacks some power.

Anyway, in the entire paper, we will refer at facts, dimensions, junk dimension and all the terminology defined in Kimball's architecture. Thus, we are going to review them in a deeper detail.

In our vision of a BI solution, the Kimball methodology is always present even if, in some cases, we take some freedom to adopt it in a slight different way than Kimball design.

Fact and Dimensions

The core of the Kimball methodology is that of separating the OLAP database in two distinct type of entities:

- **Dimension**: a dimension is one analytical object in the BI space. A dimension can be the list of products or customers, the time space or any other entity used to analyze numbers. Dimensions have attributes. An attribute of a product may be its color, its manufacturer or its weight. An attribute of a date may be simply its weekday or its month. We will see that the correct determination of attributes for dimension is not so easy but it is one of the most challenging aspects of dimensional modeling.

  Dimensions have both natural and surrogate keys. The natural key is the original product code, customer id or real date. The surrogate key is a new integer number used in the data marts as a key that joins facts to dimensions.

  A dimension might reference other dimensions and/or might correlate to other dimensions even if the main purpose of dimensions is that of joining to facts.

- **Fact**: a fact is something that happened and/or has been measured. A fact may be the sale of a single product to a single customer or the total amount of sales of a specific item during a month. From the BI analyst point of view, a Fact is a number that the user would like to aggregate in several forms in order to generate its reports.

We normally relate a fact to several dimensions, but we do not relate facts in any way with other facts. We will see that there will be the need to relate facts with other facts, but we will do this for technical reasons, not functional ones.

Facts definitely have relations with dimensions via the surrogate key. This is one of the foundations of Kimball's methodology. Surrogate keys are useful in many cases but will bring us some headaches, as we will see in the next chapters.

From the perspective of this paper, it is not interesting to go deeper into a formal definition of what is a measure or a dimension, the importance of defining granularity and so on. We assume that the reader is already familiar with all these concepts.

Star Schema, Snowflake schema

When you define dimensions and create joins between fact and dimension tables, you end up with a star schema. At the center, there is always the fact table. As the fact table is directly related to the dimensions, if you place its dimensions around you will get the familiar "star" we are used to work with.

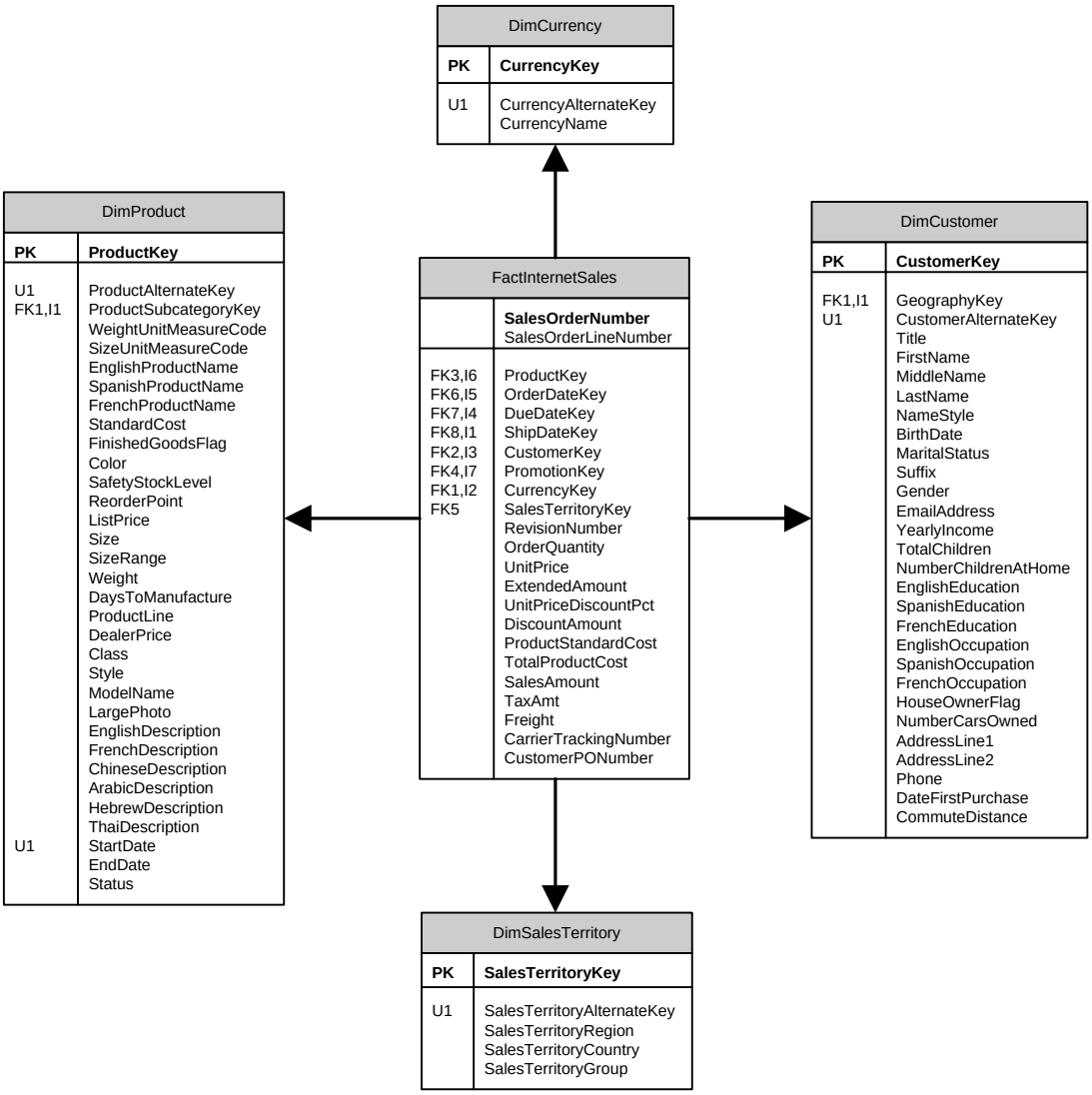

Figure 7 – (IMG 0020) Example of a star schema

Looking at this picture, you can easily understand that a Customer bought a Product with a specific Currency. The sale is pertinent to a specific Territory. Star schema has the considerable characteristic that they are easily understandable by anybody at first glance.

Nevertheless, it is not always easy to generate star schemas, sometimes the BI analyst need to (or is lead to from inexperience) end up with a more complex schema that resembles that of the relational models. Look at the same structure when you add the Geography dimension:

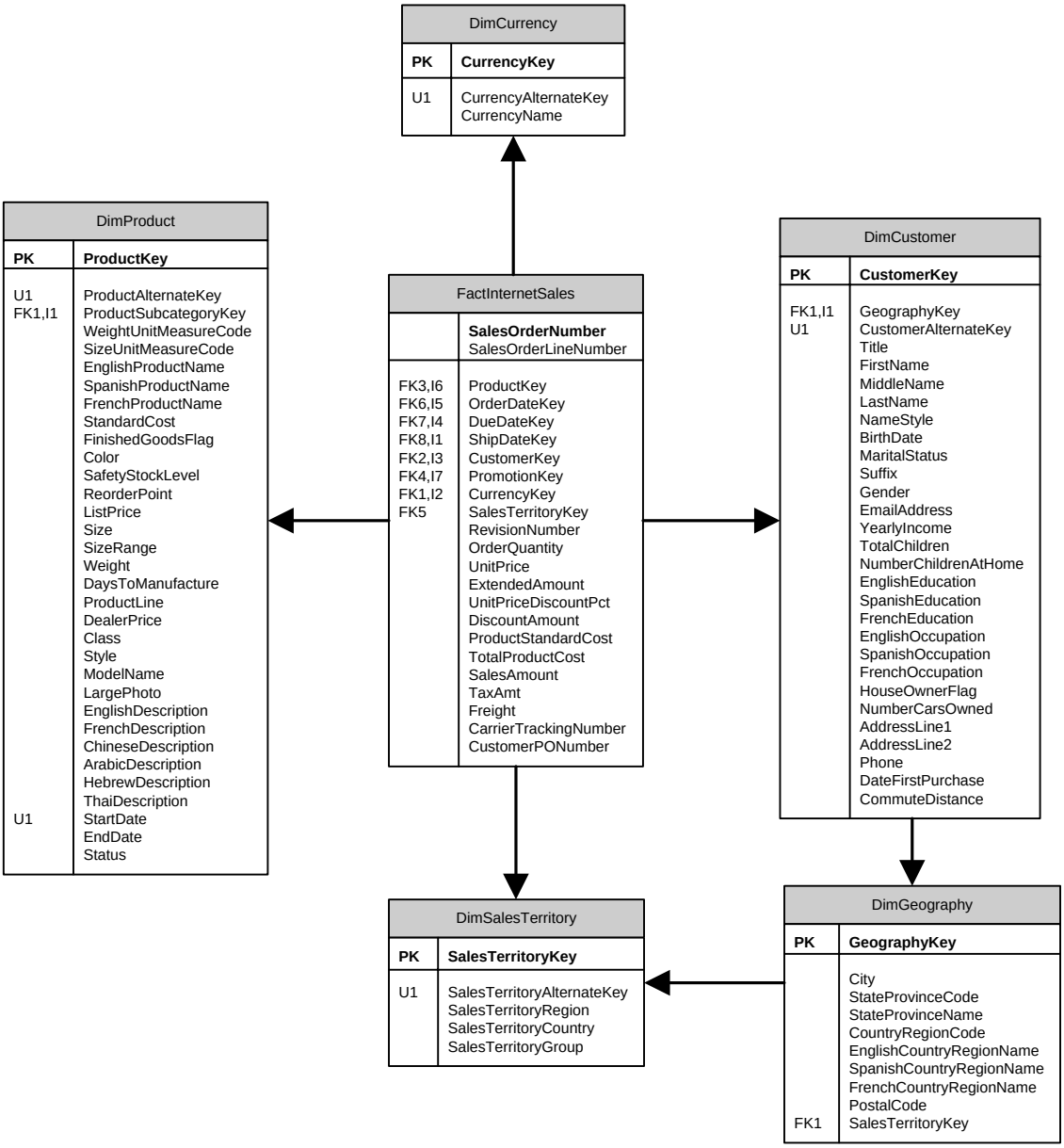

Figure 8 – (IMG 0021) Example of a snowflake schema

This is as a "snowflake" schema. Think to some more tables like DimGeography appearing in the diagram and you will easily see a snowflake instead of the previous star.

The snowflake schema is nothing but a star schema complicated by the presence of intermediate tables and joins between dimensions. The problem with snowflakes is that reading them at first glance is not so easy. Try to answer these simple two questions:

- Is the Geography dimension reachable from the FactInternetSales?
- What does the SalesTerritoryKey in FactInternetSales mean?
    - It is a denormalization of the more complex relation through DimCustomer
    - It is a key computed during ETL

The answers are easy:

- DimGeography is not related to FactInternetSales, even if it would have been possible from a technical point of view, there should be a functional reason not to do it
- DimSalesTerritory is not the territory of the customer but the territory of the order, computed during the ETL phase.

The problem is that – in order to answer these questions – you will have to search in the documentation or in the projects, otherwise you have to start diving into the ETL code to discover the exact meaning of the fields.

The simplicity of the star schema is lost when you switch from a star schema to a snowflake one. Nevertheless, sometimes snowflakes are necessary, but it is very important that – when a snowflake starts to appear in your project – you explain how to read the arrows and what the fields mean.

Once we discovered the meaning of DimGeography (it contains attributes of the territory of the order), a much better solution would have been that of adding to the cube model an "OrderGeography" dimension derived from DimSalesTerritory and DimGeography.

Junk Dimension

At the end of the dimensional modeling process, you often end up with some attributes that do not belong to any specific dimension. Normally these attributes have a very limited range of values (they normally have three or four values each, sometimes more) and they seem not to be so important to be considered dimensions.

Obviously, we do not even think to the option of removing this information from the BI solution.

We have two choices:

- Create a very simple dimension for each of these attributes, elevating their rank from attribute to dimension and still maintaining the neatness of the whole project. This will rapidly lead to a number of dimensions in the final solution that is too high. The user will dislike it, because the cube has become too complex.
- Merge all these attributes in a "Junk Dimension". A junk dimension is simply a dimension that merges together attributes that do not have another placement and share the characteristic of having only a few distinct values each.

Junk dimensions are very useful for performance reasons, because you will limit the number of fields in the table of facts. Nevertheless, a BI analyst should not abuse of junk dimensions. Let us see in more details the pro and cons of junk dimensions.

The main reasons for the use of a junk dimension are:

- If you join several small dimensions into a single junk, you will save some fields in the fact table. For several millions row this can be a significant improvement in space and time for the cube process.
- Reducing the number of dimensions will lead the OLAP engine to perform better during the aggregation design evaluation and cube process, bringing better performance to both the server farm and the end user experience.
- The end user will never like a cube with 30 or more dimensions. It is difficult to use and to navigate. Reducing the number of dimensions will make the BI solution more acceptable.

However, there is one big disadvantage in using a junk dimension: whenever you join distinct values into a junk dimension, you are clearly stating that these values will **never** have the rank of a specific dimension.

We will discuss in more details the drawbacks of a poorly designed junk dimension in the following chapter about "updating facts and dimensions".

Degenerate dimensions

Degenerate dimensions appears when you have values you do not want to drop from the fact table but that do not relate to any dimension.

The typical case of degenerate dimension is the transaction number for a point of sale data mart. The transaction number may be useful for several reasons, for example to compute the "total sold in one transaction" measure. Moreover, it might be useful to go back to the OLTP database to gather other information. Even if it is often a requested feature, users should not navigate sales data using a transaction number. If ever needed, there will be a specific report that will show the content of a specific transaction.

Degenerate dimensions are very useful for reporting. They are a kind of information not normally exposed in cubes but that can be useful for specific reporting.

Degenerate dimensions often have almost the same cardinality of the fact table. If they do not have it, then we might better represent them using standard dimensions.

Slowly changing dimension Type I, II, III

Dimensions change over time. A customer changes his address, a product may change its price or other characteristics and – in general – any attribute of a dimension might change its value. Some of these changes are useful to track, while most of them are useless.

In BI development, managing several years of historical depth, you need to keep track of the changes the user wants to analyze.

Changes do not happen very often. If they do, then we might better represent the attribute using a separate dimension. If the changes rarely appear, then SCD (Slowly Changing Dimensions) are the solution we need to model them.

SCDs come in three flavors:

- Type 1: We maintain only the last value of each attribute. If a customer changes its address, then the previous one is lost and all the previous facts will be shown as if the customer always had at the same address.
- Type 2: We create a new record whenever a change happens. All previous facts will still be linked to the old record. Thus, the old facts will be linked to the old address and the new facts will be linked to the new address.
- Type 3: If what we want is simply to get some hints about the "last old value" of a specific attribute of a dimension, we can add a field to the dimension table in order to save just the "last value of the attribute" before updating it. In the real world, this type of dimension is used very rarely and is of no interest in this paper.

The type of the dimension is not the same in the whole project. We will normally end up with several dimensions of type 1 and occasionally with a couple of dimensions of type 2.

Moreover, there may be the need to handle the same dimension with different slowly changing types in different cubes. The handling of changes will inevitably make the queries more complex and different users might be interested in different details of even the same dimension.

- Type 1 dimensions are very easy to handle and to manage: each time we detect a change, we apply it to the dimension in the data mart and that is all the work we need to do.
- Type 2 dimensions are easy too; when we detect a change, we invalidate the old record by setting its "end date of validity" and insert a fresh new record with the new values. As all the new data will refer to the new status of the record, it is simple to use for lookup only the records that are still valid and we will have only one valid record for each dimension value.

Even if they look quite simple, type 2 dimension hide many problems in the real world, as we will see in the next chapter about "updating facts and dimensions".

Bridge Tables (or Factless fact tables)

We use "bridge tables" and "factless fact tables" terms to identify the same kind of tables. We prefer the term "bridge tables", because "factless fact tables" can be misleading.

Let us discover something more about these tables:

- Factless fact tables are fact tables, not dimensions. You cannot expect a factless fact table to hold any attribute and you should expect to relate it to some dimension, as fact tables are.
- Factless fact tables are factless, so they do not contain any fact. This is the genesis of their strange name.

Therefore, factless fact tables are tables that are related to other dimensions but do not contain any fact.

If we focus on the usage of factless fact tables, we see that they represent many to many relationships between dimensions. They have to be related to dimensions (and so they should be fact tables) and, as a relation contains no attribute, they do not contain facts (and so are factless fact tables).

With the usage in mind, a better name for factless fact table is that of "bridge tables". This name is not our creation, many people use "bridge tables" instead of "factless fact table" and we happily adopt this term. Moreover, it is not true that bridge tables do not contain facts. They may define facts as any other fact table does, even if it is not so evident at first glance. We will spend several chapters on bridge tables, but this is not the right time yet.

From now on, we will never use the "factless" name but we will use "bridge" instead, because this is the correct name for their meaning into the model. There is plenty of work in this paper about many to many relationships. Thus, we will use many bridge tables to represent different aspects of dimensional modeling.

If you prefer the "factless" name, then you will have to keep in mind that – in our terminology – "bridge" means "factless fact table".

Now, let us see an example of a bridge table. Consider the following situation in an OLTP database

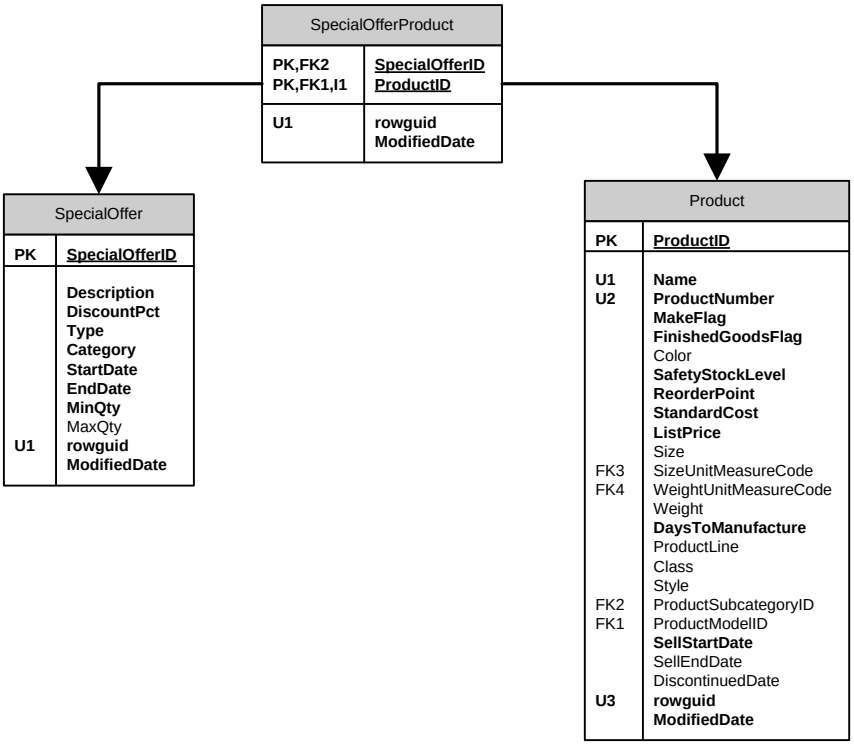

Figure 9 – (IMG 0022) Example of bridge table

In a certain period of time, each product can be sold in a special offer. The bridge table (SpecialOfferProduct) tells us which products was on special offers, while the special offer tells us information about time, discounts and so on.

The standard way of handling this situation is to de-normalize the special offer information in a specific dimension directly linked to the fact table, in order to be able to check whether a specific sale was under special offer or not. In this way, you can use the fact table to hold both the facts and the bridge. Nevertheless, leaving it into the model as a bridge table leads to some very interesting kind of analysis.

It is questionable and very interesting whether we can make the relationship only using fact tables (i.e. storing for each sale all three data: product, sale and special offer) or we should do it using a bridge table. While the fact storage of the relationship is certainly correct, we need to think carefully because, using only that model, all the relationships that did not generate any sale will be lost.

If a a specific special offer is not used in product sales, we will not see any relationship between the special offer and the product, exactly like the product had never been related to a special offer. This is because the fact table does not contain any row that define a relationship between the special offer and the product. This situation may lead to a very big confusion and/or wrong reports. We always need to remember that the absence of a fact may be relevant as its presence is. Moreover, sometimes the absence is more relevant than the presence.

We will use bridge tables to model many to many relationships that do not strictly depend on a fact that define the relationship. The relationships modeled by many to many relationships are not bound to any fact table and exists regardless of any fact table.

This is the real power of bridge tables. As we have bigger power, we have bigger responsibilities because bridge tables will sometimes cause us some headache.

Snapshot vs. transaction fact tables

Now that we have defined what a fact table is, let us go deeper and define the difference between transaction fact tables and snapshots.

A transaction fact table records an event and, with that event, some measurements that can be done and analyzed. When we detect a sale, the recording of it is a row in the transaction fact table, along with all its related information.

A snapshot is the recording of a specific situation. If you record in a fact table the total amount of sales for each specific product, you are not recording an event but a specific situation.

The total amount of sales of a product is pertinent to each product in a period. In this sense, it is not a fact but a situation related to facts. Another kind of snapshot is useful when you want to measure something not directly related to any other fact. If you introduce a ranking of your customer based on sales, payments and other characteristics, you may want to snapshot these data in order to detect the evolution of ranking related to specific marketing operations.

We record snapshots mainly for two reasons:

1. Have aggregated values at hand in order to speed up the execution of queries
2. Be able to compare the situation in the past with the current one or – in general – the evolution of a specific situation in time

The second goal is much harder to achieve. When the user wants to look at the evolution of a specific measure over time, you need to spend some time in order to understand his requirements. Let us see this with an example

A user has a ranking method that classify his customers based on

- Total amount of sales
- Payment speed

- Number of different products bought

The ranking assigns values from "A" to "E" depending on the goodness of the customer. The user wants to analyze the evolution of ranking over time because he wants to verify that the marketing is making a good job in elevating the quality of customers.

When a user asks to "analyze the evolution", we must pay attention to this question: "Do we want to verify how globally the ranking are changing or do we want to verify if the same customers are improving their quality"? There is a big difference in the modeling of these two situations. Going deep in these differences is out of scope in this paper. However, you can take a look to "The Many-to-Many Revolution" paper published on our web site to look at some examples of these models.

Updating facts and dimension

In a theoretical world, data that is stored in the data warehouse do not change. We have read several times in several books that we must build data warehouses thinking to insert operation, not updates. Data comes from the OLTP, is cleaned and stored in the data warehouse until the end of time and will not change because that was the situation at the time of insertion.

Nevertheless, the real world is much different from the theoretical one.

In our experience, updates in the data warehouse appear frequently and are at least of two different kinds:

- Structural updates: the user needs more information. He simply forgot to tell us that there is a measure he wants to use or, more often, changes in the market or in his consciousness of BI makes the need for more data necessary.
- Data updates: we need to update data that were already read in the past because it is wrong. We need to delete the old data and enter the new one, as the old information would lead to confusion. There are many reasons why bad data comes to the data warehouse. The sad reality is that bad data happens and we need to manage it gracefully.

Now, how do these kinds of updates interact with facts and dimensions? Let us recall briefly what the physical distinctions between facts and dimensions are:

- Dimensions are normally small table, less than 1 million rows, very frequently much less. They are stored as simple tables.
- Facts are huge tables, they normally have up to hundreds of millions rows, often they are stored using partitions and represent the real time consuming operation in the building of the data warehouse.

Structural updates on dimensions are very easy to make. Simply update the table with the new metadata and, at the next run of your new ETL package, the dimension will reflect the new values. If the user decides that he wants to analyze data based on a new attribute of the customer, then the new attribute will be readily available to all of his customers. Moreover, if the attribute is not present for some customers, then it will get a default value with no problems. After all, updating one million rows is not a difficult task for SQL Server.

On the other side, structural updates are a huge problem on fact tables. The problem is not that of altering the metadata, but determining and assigning a default value to all the huge amount of rows that are already stored in the data warehouse.

Fact tables are good for insertions. However, creating a new field with a default value would result in an UPDATE command sent to SQL server that will probably run for hours and kneel down your database server.

Moreover, if you do not have a simple default value to assign, then you will need to compute the new value for each single row in the fact table, issuing updates to the fact table. Again, this operation will last indefinitely.

We have seen in our experience that it is often better to reload the entire fact table instead of making any update on it. Of course, in order to reload the fact table, you need to have all of your source data at hand and this is not always an easy task. We will return on reloading the data marts later.

Data updates are a bigger problem, both on facts and dimensions. Data update in dimensions might have the same problems as adding a new field: the number of rows that we need to update is so high that simple SQL commands will be a problem. Clearly, this is true only if you need to do a massive update. Updating a small number of rows is not a problem at all.

Data updates on dimensions may be a problem when:

- The data update changes in same way the meaning of the dimension. If what you change is only an attribute of a dimension then everything will work fine. Nevertheless, suppose that the attribute you are changing is the code of a customer or of a product. We know that codes should never change but… sometimes they do. Moreover, in some situations, the dimension might not have any specific code. Sometimes the set of attributes completely defines a dimension, as it happens with range dimensions (a dimension used to create cluster of rows depending, for example, from the total sold and that may divide sales in LOW, MEDIUM and HIGH). In this case, changing an attribute will change the meaning of the dimension and will lead to very complex situations. We will discuss some examples of those situations in the chapter "Multidimensional Modeling Toolkit", because they are too complex to handle here.
- We need to update data because wrong information entered in the data warehouse. This situation is very annoying. Suppose to have an SCD of type 2 and that a record entered the data warehouse with wrong attributes. In this situation, we would have created a new record and linked all the facts received after that to the new (and wrong) record. Recovering this situation requires us to issue very precise UPDATE instructions to SQL Server and to re-compute all the fact tables rows that depend – for any reason – from the wrong record (we might have fact tables with aggregates to speed up some computations).

  Moreover, bad data in dimensions is not very easy to catch, sometimes several days – if not months – pass before someone (in the worst case the customer) discover that something went wrong. We will not be able to discover bad data, because from our point of view bad data looks the same as good one.

  There is no good recipe to avoid bad data. When it happens, we need to be ready to lose some time to recover from the error. Clearly, "one shot" databases are not prone to this situation because, if bad data is corrected, the entire data warehouse will be reloaded from scratch and the situation will be restored in a snap. This is one of the most interesting point in the "one shot" data warehouses and one of the main reason for which we always choose "one shot" technique when it is possible.

Natural and Surrogate keys

In Kimball's view of a data mart, all the natural keys should be represented with a surrogate key that is a simple counter and has no meaning at all. This will lead to a complete freedom in the data marts to add or redefine natural keys meanings. Moreover, the usage of the smallest integer type that can hold the data will lead to a smaller fact table.

All this is very good and advisable. Nevertheless, there are situations in which the use of surrogate keys should be more relaxed or – from another point of view – there is a real point into making the surrogate keys meaningful instead of meaningless. We will discuss it later in a more technical chapter but it may be interesting to have a look at these situations here.

- **Date**: we can use a meaningless key as a surrogate key. However, will we have any advantage in doing so? In our opinion, the best representation of date surrogate key is an integer in the form YYYYMMDD, so 20080109 is January 9 of 2008.

All invalid values may be easily represented with negative numbers, so -1 may be the unknown date, -2 may be the empty date and so on. You have plenty of space for all the dummy dates you will ever need.

Moreover, the date dimension will never change. You might add some attributes to a date dimension but, surely, you will never add a new day to the calendar.

Finally, one very important reason to use meaningful keys for the date dimension is that we normally use dates in partitioning functions with SQL Server 2005. If we end up using surrogate and meaningless keys for the date dimension, we will lose this very important functionality for large fact tables.

- **Ranges**: Suppose we want a dimension that will rank the sales based on the amount of the specific sale. We want to analyze information based on a range, not on each single amount.

  If we define a RangeAmount dimension and an ID_RangeAmount key in the fact table, this will solve our modeling problems. However, what will happen when the customer will want to change the ranges? We will have to re-compute the whole fact table because the ID_RangeAmount key will become useless.

  On the other side, if you decide that 100$ will be the granularity of the range dimension, you can use FLOOR (Amount / 100) as ID_RangeAmount and, in doing so, you will be able to update the attributes of the RangeAmount dimension that will lead to hierarchies without updating the fact table.

  Even in this case, we will use a surrogate key that has a semantic defined.

- **Excel and Proclarity** will save queries maintaining the ID's of the dimensions when we go at the granularity level. This means that if –for any reason – we will change this ID in the future (for example, because we have a "one shot" BI solution), the saved query will no longer work.

  This situation will be explained and solved later but, just to give you an idea, there are situations (for instance, parent/child hierarchies) where this problem and the auto-referencing nature of the dimension will lead us to a very annoying problem. In these conditions, we will not be able to use surrogate keys to relate facts and dimensions and we will be forced to use natural keys instead.

- **Junk dimensions**: junk dimensions, when defined at the conformed dimension level, lead to problems when there is the need to update data. We believe that junk dimensions are a logical kind of dimension and should not be confused with the physical layer. The technique that we will define for junk dimensions will make use of multiple surrogate keys that are a sort of mix between natural and surrogate keys. In this case, we will not deviate from the standard for technical problems but because of our methodological approach, which prefers the ease of change instead of the need to maintain physical and logical model too strictly tied together.

Therefore, the conclusion is that surrogate keys are very useful and we are not saying that there is something wrong with them but, in some defined situations, we will deviate from the standard of surrogate keys and use different form of keys, depending on the situation we will face each time. We will try to be very clear on what these situations are. We consider them as exceptions to the general rule of using meaningless surrogate keys for dimensions.

WHY AND WHEN WE USE KIMBALL METHODOLOGY

Kimball methodology of dividing the BI world between fact and dimensions is very effective and leads to a complete solution in a very small amount of time. Moreover, Kimball's technique has a lot of documentation and you can find an answer to nearly every question you can have. In our experience, you can start from scratch and give the user a first feedback on his/her data in a matter of days. After such prototype, you start the normal BI solution lifecycle.

The resulting data marts are very easy to query from both developers and end user tools. The direct relationships between facts and dimensions grant anyone the ability to construct very simple queries, most of the time without having a look at the metadata documentation.

This is certainly correct for simple star schemas. When you start using snowflake, bridge tables and/or other advanced techniques, then the simplicity is somehow lost and the complexity of queries starts to rise. Nevertheless, it is always much easier then querying the original OLTP database.

In our experience, Kimball's methodology is great for the first steps of the introduction of BI to a customer. It is perfect when the complexity of the data warehouse is not too big and where the BI infrastructure handles data coming from a limited number of sources. When the data warehouse is getting complex, then it will be dangerous to force your way of thinking only in the Kimball direction.

A good old dictum is: "there's nothing that cannot be done by means of adding one more level of indirection". In the BI world, this is still true: when things are getting too complex, it is time to introduce a new level of indirection. Using Inmon's approach to the data warehouse creates this new level of indirection that will help us solve several problems, at the price of a data warehouse structure that becomes much more complex.

The Inmon Methodology

Inmon's vision of a data warehouse is very different from that of Kimball. If we want to synthesize this in a few words we can say that Inmon's structure is based on a complex corporate relational database while Kimball's one is based on a dimensional and fact view of the same corporate data.

Inmon states that the creation of a relational database with a slight normalization need to be the foundation for data marts, so we will not create the data marts directly from the OLTP system through a staging area. Instead, we will create them from the relational architecture of the corporate data. We call a data warehouse built in such a way an "Architected Environment".

We are not going to describe the full process of building such architecture because

- It is very well documented in Inmon's books
- It relies on the very well known theory of relational database, that the reader knows well
- Our goal is not that of create the data warehouse but to describe a general architecture and give some new tools of the trade to the BI analyst

There is nothing wrong with Inmon's view of a data warehouse. More, we do believe that the capability to create an architected environment as a foundation for data marts is a great achievement, because it gives to the BI analyst a place where he can make the consolidation of information easily. Nevertheless, as it is a great achievement, it will take time and money. It will take so much time and so much money that – as a first step in the building of a BI solution – it may appear unfeasible.

We should stress one other point. We do not see a real separation between Inmon's view of a data warehouse and Kimball's bus architecture. Surfing on the web, the choice between which paradigms to use is much more similar to a war of religion than to a technical issue. Nevertheless, we do believe that a good BI analyst should know well both methodologies and be able to choose which one to adopt, changing his (religious?) belief to the real needs of the end user. There is no space for solution prêt-a-porter in the BI world, we always need to deeply understand our user's expectations and spending capabilities to give him the best solution for his money and time.

What is more interesting is the fact that we believe that both methodologies can happily coexist in a single data warehouse. Kimball's solution is very fast to achieve. We can start with a pure Kimball data warehouse, moving data directly from the OLTP to facts and dimensions in the data marts. When the going will get though (i.e. more cubes will appear and dimensions will start to spread) we will add one level of indirection adding the data warehouse level.

One of the final goals of the data warehouse is the production of data marts made up of dimensions and facts. Thus, we are not going to waste any work, we are adding more horsepower to our BI solution by opening a new level (the data warehouse) where we can coordinate the cleansing of data.

What – in our opinion – we should not do is ignore that a data warehouse can be built using Inmon's methodology. When we start to have computer services that need data from the data warehouse, when the sources of information are spreading, when the number of facts and dimensions in the cubes are augmenting, Inmon's structure is really a thoroughbred. When needed, we ride it.

We borrowed from Inmon's architecture the concept of Operational Data Store (ODS). This is a storage system that holds quickly retrievable profiles of data. Those profiles can serve other software systems in the solution.

The point is that when other pieces of software relies on data that can be generated by your BI solution, the construction of an ODS is useful. ODS can pre-compute data from OLAP cubes, data marts, relational data warehouse. The pre-computed data will be available so quickly that they can generate immediate synthetic views of the status of a customer, a product, a specific period.

Giving you user's IT infrastructure the capability to use an ODS is a great achievement, because it will lead to BI data available all over the intra network.

Again, the building of an ODS is not strictly related to the presence of an architected environment, we can build an ODS directly on top of data marts and OLAP cubes. ODS is one of the many tools of the trade that we can bring with us when it will be the time to build a BI solution.

WHY AND WHEN WE USE INMON METHODOLOGY

Our standard approach with a new user is that of using the Kimball methodology. When the first cubes start to work and BI starts to spread over the customer's IT infrastructure, we make ourselves several questions:

- Is there the need of an ODS?
- Is there any software that will query our data warehouse in a predictable way?
- Are all the data marts independent or they can deduce some of them from a more general view of data?
- Are all the data marts showing comparable data?
- How many updates to our data marts are required if changes happen in the data or in the structure of data coming from the OLTP?
- Is there a need to have an historical depth and detail that we cannot easily represent with facts and dimensions?

We need to answer these questions each time we add a new feature to the system. When we feel that the complexity of the entire BI solution is getting too high, we start to gradually integrate Inmon's structure into the solution and lead it to a more complex data warehouse.

Of course, as we are aware of this possible transition since the beginning of the data warehouse construction, we create the whole structure of the ETL pipeline so that this change will be feasible. For this reason, we try to develop the cleanest architecture.

Anyway, even if we can use Inmon's methodology at the data warehouse level, the final structure is always that of separate data marts for the department level of information. Therefore, the data warehouse is an intermediate step where we can consolidate and cleanse data in order to feed data marts and outer systems.

BUILDING THE ARCHITECTURE

In the previous chapter, we made several considerations about the architecture of a BI solution. As it is an introductory chapter, we left some technical details. Now, in this chapter, we will recall the same consideration but go deeper up to a complete construction of the infrastructure of a BI solution.

This chapter mixes different technologies because we believe that – in order to take the correct decision in design – you need to understand and cover all the technologies that play around in the project. In this chapter, you will find many details. They are not useless; always remember that devil lives in details.

At the end of this chapter, our vision of the infrastructure of a BI solution based on the BI suite from Microsoft will be complete and clear.

DESIGNING RELATIONAL DATA

In this section, we provide information about the relational side of a BI solution. We define the number and the types of databases that we will use, and some hints about how to transfer successfully data from one database to the other.

Before diving into the technical details of each single database, we will start with some considerations about the usage we are doing of some of the basic techniques available in Microsoft SQL Server to organize our data.

Usage of schemas

We need to divide the data through the whole ETL pipeline into subject areas. The meaning of a subject area really depends on the needs of the customer. Typical subject areas include:

- Sales
- Accounts
- Warehouses
- Suppliers
- Personnel and staff management

Clearly, this list is far from being complete and is different for every business. We will need to define subject areas along with the customer in order to detect the correct ones.

SQL Server 2005 provides schemas to arrange tables and – in our experience – the usage of schemas to assign database objects to subject areas leads to a very clear database structure.

We will use schemas through the whole ETL pipeline because very often a specific table in the OLTP system already belongs to a specific subject area. Using the subject area separation from the beginning will let us easily understand whether the usage of a table is correct or not.

Some tables will eventually have no place at all in subject areas but you might always define a “COMMON” subject area that we will use to group all the tables that have no other place. This too leads to the comprehension of the usage you are doing of each table through the entire ETL pipeline.

As a rule of thumb if a table changes subject area during the ETL pipeline, then something is going wrong with the design and we have to take a deeper look at this in order to have a chance to correct it.

Usage of views

As you may remember, in a previous chapter we outlined the need to de-couple each step of the ETL pipeline. We do not want to recall all the good reasons for doing it because we have already seen that this de-coupling is strictly mandatory. Now we want to define the technical means used to obtain this de-coupling.

You have several options in order to make this operation but, in our experience, the creation of SQL views is the best choice. Here are some reasons:

- **Views are stored where we need them.**

  When you need to read the specification of an interface, you want to do it quickly. Views are stored in the database that will provide data to the next step (so the views in the OLTP mirror provide data to the staging and to the data warehouse). They are exactly where you want them to be. If you need to update the OLTP mirror, you want to be sure that all views are still intact after you have done the update.

  Think – for example – at the feeding of the cubes. You may be tempted to use the data source views available in SSAS to define the views that will feed your cube structure. In doing so, you are hiding these views from the database administrator that could update the data marts.

  It is not easy to have a tour into all the projects to see what the usage of a specific field is. If you use views to declare which fields you are using and for which cube, then it will be easy for everyone to have a deeper understanding of the whole structure. Moreover, there are tools on the market that can analyze dependencies between table and views. It is not easy to do the same if views are stored out of the database from which they extract data.

- **Views can easily change names to columns.**

  In the database, we might have SoldQty as a field of a table. This is good because it is incisive and does not contain useless spaces. In the cube, we want to show it as "Quantity Sold" simply because our user wants a more descriptive name.

  Views are a very useful means to change names when needed. Moreover, with views we are publicly declaring the change of the name so that everybody will easily understand that a specific field with a name in one level is – in reality – a field that might have another name in the previous one.

  Moreover, the OLTP database often has very cryptic column names. The ability to change the name from "AAGE200F.DSCAG" to "Sales.SalesPerson.Name" is very valuable. Of course, you may object that they represent the same value, but they do it in the same way as machine code represents the same algorithm as a C# program. "More readable" will always translate directly into fewer errors to find.

  It is clear that we should avoid the practice of changing names at each single level. As always, having the opportunity to do something does not mean that we need to do it.

- **Views can make simple computation**

  What can we do if we need - for some MDX expression - the value of "Qty * Price" in the cube in order to use it? We can compute it directly into the data source view but, as before, we are hiding a computation into a specific project and other people will not be able to see it at a glance. Using a view, we are declaring the computation that we are doing.

  This is certainly true for simple computation. On the other side, if a view is doing too complex computation, then we are probably missing some transformation code into some of our ETL packages.

  We should avoid the practice of hiding too much code into a view because it often rises from a lack in the analysis of the ETL flow. Moreover, this computation will waste time when we will execute the view. Putting the code into the ETL phase will consume CPU only once, when we first compute the value. From then it will always be available in a snap.

- **Views are made of plain text**

This is great as it is simple. We can easily search for all the occurrences of a specific column, table or any kind of value using a simple text editor. We do not need any development tool nor need we to dive into unreadable XML code to have a clear view of the usage of a specific value.

If we need to update a view, we can do it without even opening BIDS. That means that nearly everybody can do it.

Moreover, as views are simple text, source control system will handle them very easily. We can check who updated what in order to solve a problem or answer to a question. For example, an Integration Services package is saved as an XML file (the file has a .DTSX extension). Two versions of a package are comparable using diffgrams. However, it is not easy to understand the differences reading an XML file, just as it is not easy to edit a DTSX file without using the graphical editor embedded BIDS. Therefore, it is not easy to look at the differences in SQL code embedded into two versions of the same package.

- **Everyone who has the rights to do it can update views**

  A view can be updated very quickly as it does not require any kind of processing, just ALTER it and the work is done. You do not need to use an UPDATE statement to make simple (and possibly temporary) changes to the data.

- **Views can reduce the number of columns**

  There is really no need to show to a cube or to an ETL package more than what it strictly needs. Moreover, showing more fields will only lead to confusion, because there are probably some technical columns that you might want to keep private, in order to be able to change their behavior in the future without having to recompile or revisit all of your projects.

  A view can filter the number of columns providing you the ability to hide what you do not want others to see.

- **Views can provide default values when needed**

  When you have a NULLable column that contains NULL values, you can easily assign to it default values using views. It is very questionable why you should have a NULLable column in a data warehouse, but sometimes it happens.

- **Views can show a real star schema even if the relational model is more complex**

  We will return on this topic later when we will describe in more detail how to build the interface between the data marts and the cubes. What is important is that sometimes you end up with a relational design that is not a perfectly designed star schema, for example because you had the need to store several keys into dimensions. A reason for that could be the need for some data during the ETL phase or the participation of a dimension in a more complex design when compared to what was needed to a specific cube.

  By removing useless columns, by creating joins when it is necessary and in general by designing the appropriate queries, you can expose to the SSAS solution a real star schema, even when the relational model has a more complex structure.

  There is no gain in showing to SSAS the real relational structure of your data marts. Views can give a significant help in hiding what is not necessary and in restructuring the data marts. We will provide some examples of that in a further chapter.

- **Views are database objects**

  As views are database objects they inherit two great properties:

  - We can configure views for security. Nobody that does not have the right to see some data will be able to gather them fraudulently.

- Views can have a schema. As we are using schemas for the definition of subject areas, we will be able to assign views to subject areas. This simple fact will lead to a very clean project where each object belongs to the subject area that is relevant for it.

We are not saying that views are the only way you can use to obtain all of these advantages, at each single level of the ETL pipeline you can obtain the same using different means. What makes views our best friends is the fact that we can use them through the entire ETL pipeline and provide at each level a uniform way of describing the communication between one level and the subsequent. We believe that no other means is able to work through all the ETL levels in such a uniform way.

Of course, there are some exceptions to these rules and – in the following chapter – we will see that sometimes SQL code is better than views. As a rule, you can think that each communication layer between one level and the subsequent ones in the ETL pipeline is composed with views.

Now a little anticipation: the usage of schemas to describe subject areas that we presented in the previous chapter stops at end of the data mart level, when we define views to provide data to cubes and reports. Here the distinction between subject areas starts to be inconsistent because a cube will quite always gather data from different subject areas in order to provide its results. The views that will provide data to the cubes will not belong to a specific subject area but to schemas constructed with the name of the cube to declare that the view is relevant to a specific cube or report.

Therefore, if we have a cube that presents sales at a monthly level we probably would call it MonthlySales. The name of the view that provides the date dimension to this cube should be CubeMonthlySales.Date. Doing so, we are clearly stating who will use the data.

The careful reader should remember that – in our vision – the source of the Date dimension in the monthly sales could be different to the source of the Date dimension for sales management. The reason is that the number of different values in each of those dimension is varying depending on the age and granularity of the facts being presented. Different views can provide the same data with different boundaries and/or granularity.

It is not an error to define more views for the same dimension, even if it seems cumbersome, because we do not know what changes we will do in the future to that specific dimension as it is seen by that specific cube. Using views, we gain the freedom to make any change smoothly. Moreover, we can have an immediate look at what usage that specific cube is doing of the database simply looking at the views that exists in its specific schema.

Mirror OLTP

We have already discussed the advantages of creating a mirror database for the OLTP system. In this chapter, we will look at some details about its creation.

The Mirror OLTP database is a copy of the relevant data needed from the OLTP database to process the data warehouse. The Mirror does not need to hold a copy of all the tables from the OLTP database. It contains only the tables that we need to feed the ETL pipeline.

The same happens for columns. OLTP databases often contain a lot of technical column and/or information that are not useful for the data warehouse. These data are not useless, they are useful during OLTP processing but, from the point of view of the data warehouse, we can ignore them.

To ignore a column or a table we simply do not create it into the mirror database. Therefore, the mirror database is – from the metadata point of view – a subset of the OLTP database both in number of tables and of columns in each table.

The BI analyst need to define which tables and columns should be included. The worst choice is to ignore the problem and produce a mirror of the whole database. Not only this decision will lead to poor performance, the worst of it all is that we are not declaring which data is useful and which one is not.

If we will ever create this situation then we will be bounding the OLTP database with its mirror in such a tight way that any kind of update to the OLTP system will inevitably lead to updates in the process of mirroring and, sooner than you may think, it will become a nightmare.

If from the metadata point of view the OLTP mirror is a subset of the OLTP database, the same may or may not be true for data. You may have several situations:

- **The data warehouse and OLTP are so small that you have time to rebuild everything during nighttime.**

  In this case the OLTP mirror must contain all the data coming from the OLTP database as all the data has to be rebuilt every time.

  Even if this situation may seem fancy, there are many small companies where the size of the whole database is so small that the servers will be able to rebuild a complete data warehouse in a few hours. This will lead to a very fast development of the data warehouse and will give us the ability to accept any change very quickly, because we do not have the need to worry about historical data.

- **The data warehouse and OLTP are small, but you have to handle historical changes.**

  Even in this case the OLTP mirror can contain the whole subset of the OLTP database. It is not useful to optimize the process of mirroring if the server is not doing something else. Thus, you can take a full snapshot and then extract from there the relevant updates.

- **The OLTP is too big to have an option on mirroring it.**

  In this case, we can decide to mirror only a subset of the data. We can take only the last year of data and work on it, ignoring anything that has happened before.

  Obviously, if we will ever need to gather more information from the OLTP system, we may need to mirror a major portion of it. As we need to make seldom this operation, we can take the time needed to do it.

  The same consideration applies when we create just the OLTP metadata database. Building the views that will show the information from the OLTP, we can make constraints with specific WHERE conditions that will make only a portion of the whole database visible to the ETL pipeline.

Once we have defined how the OLTP mirror is structured, we have to fill it with data. The golden rule during the mirroring process is to not make any change on data coming from the OLTP original database. If we have to change names to columns, data types or any other property of the database, then we are already entering into the ETL phase. Mirroring is not part of the ETL. We use it as a simple and convenient step to optimize resources.

Filling the OLTP mirror is just a matter of writing SELECT statements that will take data from the OLTP database and fill tables with the same name into the mirror OLTP database.

In this situation, we have two options:

- **Use an SSIS package**

  We can easily build a package with several data flow tasks that will make the mirror. They will execute in parallel letting us cut the total execution time to that of the biggest table. This is the most common technique but it is far from being the best.

  Because we need to do no processing at all during the mirroring process, all the task need to go in parallel. If at some point we are putting a link between two tasks, then we will know that something is wrong with our mirror algorithm.

- **Use a custom tool**

  The maximum degree of parallelism that we can obtain with SSIS is that of the table level. There are custom tools (you can find a free one at www.sqlbi.eu/SqlBulkTool.aspx, written and extensively

used by the authors) that will let us use the partitioning features of SQL Server 2005 to raise the parallelism to the partition level. This means that a table of ten million of rows might be loaded as ten partitions of one million rows each, all in parallel. Moreover, as we have written this tool with mirroring in mind, it will handle any change in metadata without user intervention for reconfiguration.

The advantage in terms of mirroring speed is tremendous. It will let us build the OLTP mirror in minutes instead of hours, making it much easier to adopt the mirror OLTP option.

In both of these situations, we have to check that indexes, constraints and any other option that will slow down the process of mirroring are disabled. At the end of the mirroring process, we can create indexes and stored procedure to make the process of reading data from the mirror faster.

After the data is loaded into the mirror, our work is still not finished. We have to write the queries that will read data from the OLTP mirror and feed the subsequent phases of the ETL processing. If we are choosing the option of creating only the metadata OLTP database then we will skip all the steps before and we will start our work from here.

Supposing that the next process of the ETL phase is made of SSIS packages, we need to write a view for any interaction that the SSIS package will have with the OLTP mirror or metadata database. This normally means to create a view for any Source that we have in the ETL package[1].

In this picture you will see the complete design for the OLTP mirror database:

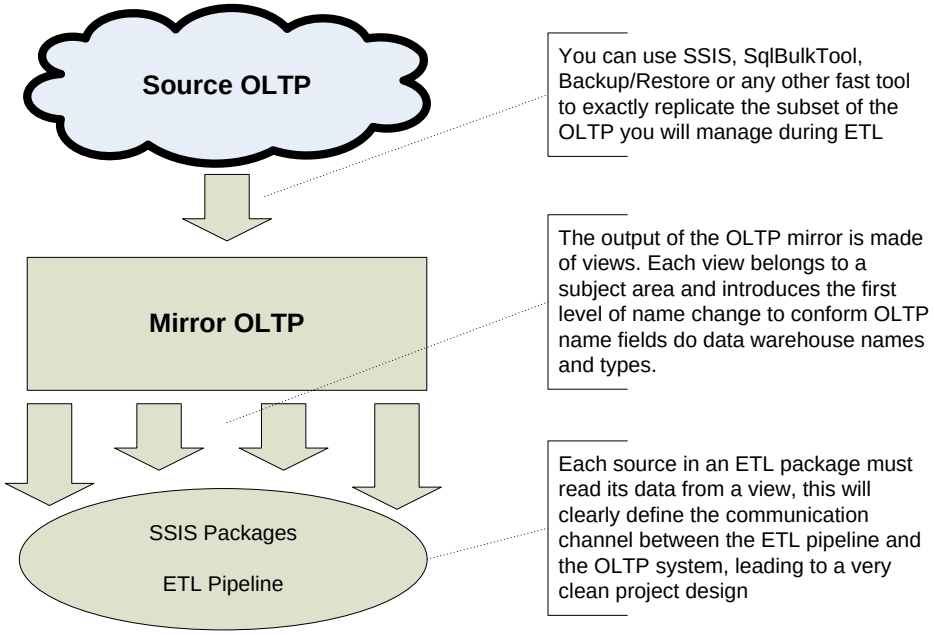

Figure 10 – (IMG 0023) Mirror OLTP detailed

It might seem cumbersome to do all this works because we can simply write SQL queries directly into the source components of SSIS. Once again, remember that documentation of the communication is a very important task for a BI analyst; it will save us a lot of time in the future when we will have to investigate about the consequences of a change in the OLTP database. Moreover, the views can have a schema and, as we are using schemas to identify subject areas, this is the first point in which you are declaring not only the usage of information coming from the OLTP but also the subject area into which each object will go.

One last point: if we de-couple the SSIS package from the database using views, we are giving to some others the opportunity to fine-tune the queries adding indexes and working on benchmarks while we are still working on the SSIS package. All that the database administrator will need to know is which views to

[1] After having done all these queries, we may be tempted to use the "Table or view" option in SSIS sources. This would be a very bad idea. We always use SQL Command with SELECT statements that gather data from the views. The performances of SQL commands is much better than that of tables or views due to the internal operations that SSIS does to get data from the source, especially when using an OleDB connection.

optimize. Our package will then use the optimized queries as it used the slow ones, without any kind of change.

Of course, we can obtain more or less the same indirection level by means of using the Data Source View object in SSIS. Doing so, we are hiding the information about the interaction between our package and the database inside of the package. To force the other members of our team to know how to use SSIS is not a good option. Views are universally understandable in the database world and, as a common denominator of every database professional, are a very good language of interaction.

If the OLTP database is a SQL 2005 database, we have another very interesting option to consider. If – before starting the mirroring process – we create a SNAPSHOT of the entire database, we are freeing our entire process from potential LOCK contention and – most important – we are viewing a consistent state of the database during the whole process of mirroring, even if some users are still interacting with the database.

As snapshots are very fast both to create and to drop (at the end of the mirroring process), we consider it as a primary option when we have to mirror a SQL Server database. Of course, other vendors might provide similar mechanism and, if appropriate, you can consider using this option if you are able to do it.

One last option, for SQL server databases, is that of doing a backup/restore. This may be interesting from the performance point of view but, in doing so, we are missing the important topic of declaring the subset of the OLTP database that you will use. We do not believe that this is a real option and we use it only when it is the only viable option for doing a fast mirror.

Staging

The staging area is the first place where the ETL process starts. In our vision, the staging area is the place where temporary tables needed by ETL packages will reside. No persistent data should reside in the staging area. Persistent data should reside in the configuration database, when needed.

As this is our vision of the staging area, it will be very difficult to describe in detail how to design its content. Moreover, the design of the staging is somehow an oxymoron because staging is, by definition, a temporary and technical place.

The ability to use SSIS for complex data flow tasks has lead several programmers to avoid the creation of temporary tables. They prefer to make very intricate data flows that perform the same operation without the need to have temporary tables.

This is very good from the performance point of view, running fast is always preferable when compared to walking slow. Nevertheless, there is always a price to pay and we have to be careful when it will be the time to decide whether to use temporary tables or not.

Temporary tables are a fundamental part of the debugging of the ETL phase. When we detect an error, we will often see it in the OLAP cube. More often, the user will tell us that a specific number is incorrect when compared with his sources of data. What we need to do is to go back in the ETL pipeline in order to detect where we are computing that specific value and check the code.

If we do not have temporary tables, we will not be able to verify the output of each step of the pipeline in order to find the wrong point in our project. We can only look at the boxes and arrows inside the data flow and think to what is happening there. On the other side, temporary tables will let us check the outputs of each step until we find the wrong one. We can correct it and – having the intermediate tables – we can run only that particular step in order to check if the correction is now producing the right results.

This seems to be a “debugging tip” and – in fact – it is. However, what makes it useful is the fact that BI solutions are very dynamic by nature. We will change them several times, even when they have been in production for a long time. A small change can cause wrong results. We will need to be able to debug it very quickly during the whole lifetime of the solution.

There will be situation where we will save the temporary results in private and technical cubes, just to be able to look at them with Excel and navigate the data in an easy way. Clearly, we will hide these private

cubes to the end user. Nevertheless, they will provide an invaluable source of information about what happened during the ETL phase and they will save us several days of troubles in debugging the code.

This does not mean that any processing has to be stored on temporary tables. Only when the processing is complex, when we feel that we will have to update the code quite often, then we can decide to store intermediate results in temporary tables. Otherwise, the usage of temporary tables will slow down the whole ETL phase without giving us any advantage.

We think to a BI solution as something that is always under development and we will make ourselves ready to make a session of update/deploy/debug in any moment.

Data warehouse

The data warehouse level is the most important one in the whole project but, from the point of view of this chapter, it is the less interesting. You will find many books that describe the details about how to build a data warehouse and we are not going to replicate any information here.

The philosophy we are following will guide the choices:

- If we are building a Kimball data warehouse, we can follow Kimball's directions to build fact and dimensions according to his methodology. We will end up with a data warehouse composed of data marts, so our work is finished when we have built the single star schemas.
- On the other side, if our choice is that of an Inmon data warehouse, then we have to follow his directions to design a relational database that will hold the data warehouse. Moreover, we need some ETL code to produce the departmental data marts with their star schemas.

In both case, at the end of the data warehouse level, we would think to be ready to expose the data marts to the SSAS cube. Surprisingly, the answer is no. In our vision, there is still one very important step that needs to be completed in order to have a good design.

Either we chose Kimball or Inmon, we have some ETL that produced a database that should look like a star schema. This is not yet a data mart; it is a database that will finally produce a data mart.

Data Mart

We defined data marts as a "logical subset of the data warehouse". The question still open is "what do we mean by logical?" We use the term logical in contrast with physical, where we mean by physical level the real database design that is holding our data.

A very good example of this is in the AdventureWorks BI solution. If you look at the Product dimension, you will see that it contains data coming from three tables:

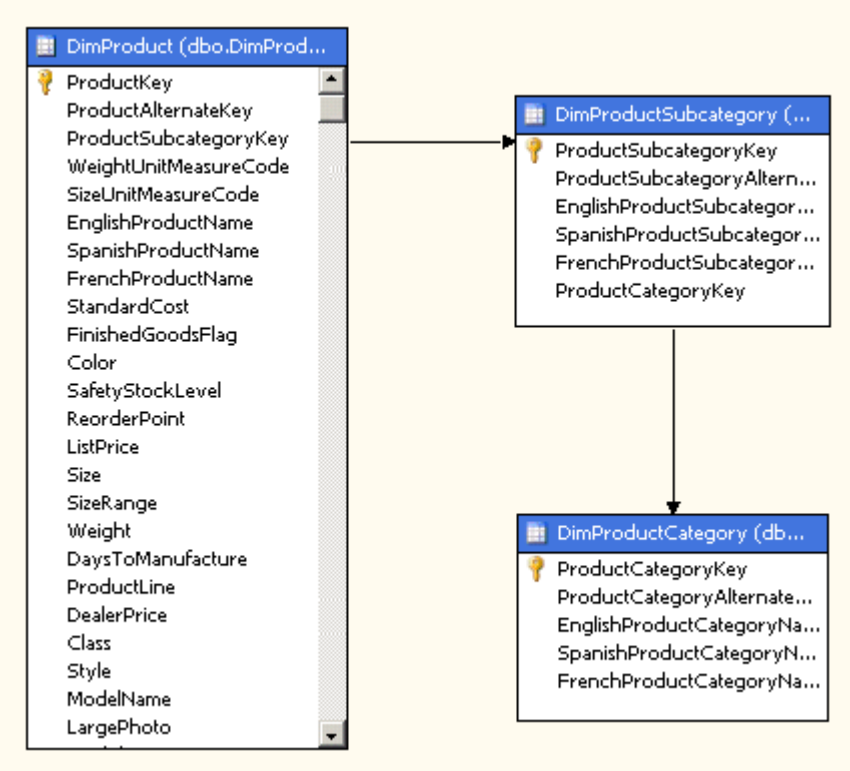

Figure 11 – (IMG 0024) Product dimension in AdventureWorks

At the database level, these three tables resemble the structure of three separate dimensions. At the end of the ETL pipeline, the same three tables will feed the cubes to produce the dimensions. Here we will lose two dimensions and see only category and subcategory as attributes of the dimension products. Therefore, we have three tables but the user will see only one dimension.

The problem is that if the user has no way of querying the category and subcategory as separate dimensions, then they are not dimension at all and they should be modeled as attributes of the product dimension.

Clearly, the analyst has thought – in this specific situation – that in the future these attributes may become dimensions by their own when, for example, he will introduce a budgeting that will make extensive use of these two dimensions. In that situation, it will be easy to separate the product from its categories because the relational model is already prepared to accept this change.

There are absolutely no problem at all with the relational design. What is not clear is the fact that the data mart is composed of only one dimension (product) and the other two dimensions are technical ones that should not be exposed outside of the relational world.

Therefore, from the logical point of view, we only have one dimension. From the physical one point of view, we have three tables. This is the case where our definition of data marts starts to make sense. Even if the physical layer has three separate dimensions (which is very good) the logical one (the data mart, for instance) will expose only one dimension.

If we use SSAS to make this join between dimensions (using reference dimensions or the DSV, for example), we are hiding the important information about the composition of the data mart into the SSAS project. This will lead to two problems:

- If the dimension is used in two distinct cubes, then it will have to be replicated exactly in the same way in order to give a consistent view of the dimensional structure of the BI solution
- If a technician does not have access to the SSAS project, he will never see this structure and he will spend time trying to understand which relationship exists between the cube and the relational world.

On the other side, if we create specific views that will make the join, we will get several advantages:

- The declaration of the data mart structure resides into the database. Anybody will easily check it because it is declared in the structure of the database.
- The SSAS cube will see only one dimension, which is exactly what it will show to the user.
- The surrogate keys (ProductCategoryKey and ProductSubcategoryKey) will disappear. They are not useful in any way, they are not exposed to the user nor used in any way into the cube. Clearly, this situation might change in the future. When, for example, we will add budgeting to the solution, the keys will start to make sense. At that point, we can revive the dimension and expose the keys. This is the power of having a physical layer different from the logical one; we can change the solution for our needs.
- We will see a star schema and will never be tempted to think that there is a snowflake. In fact, there is no snowflake at all.

There is another big advantage in building the data mart with views: views are very easy to update and change by need. If – for any reason – we will need to add another table to this structure and create or change attributes to the product dimension, it will be very easy to change the data mart view without the need to update the SSAS solution.

In case we have designed an Inmon data warehouse, we will have exactly the same problems to solve. It is very common to have some architectural considerations that lead us to create a different logical and physical structure. Views are the means that will let us derive the logical from the physical one and – in our experience – are real lifesavers, as all the indirection levels are.

OLAP cubes

We have already outlined that OLAP cubes have their source in the data mart. This is not always true. Sometimes the OLAP cubes will have their structure reside in the data mart but they might have the need of loading data that is slightly different from what is stored in the data warehouse.

Think, for example, to the date dimension. Date dimension usually contains a huge period in order to be able to fulfill all the date need of all the data marts. Clearly, we are thinking that we have only one dimension and one table that will contain dates.

If the range of the dates for the whole data warehouse starts from January 1 of 1950, there is absolutely no need to show the date starting so far in – for example – a financial cube that shows the sales and contains data from January 12 of 2005. In doing so, we are forcing the user to scroll down to the current year each time he wants to select a specific date from the dimension.

It would be much better to have the sales cube showing only a subset of the date dimension by limiting its range to only the period that is currently useful for the fact table.

In order to obtain such a feature, we need another set of views that we call cube views. Cube views are useful only for one cube and are very specific as their output is exactly what the OLAP cube needs, nothing else.

Cube views should maintain the same structure of the dimensions and fact tables they rely on, making only aesthetic changes and/or range delimitations. The reason is that cube views should not carry on any ETL process. Their only reason of being is that of feeding a cube with data already computed in the data warehouse.

Nevertheless, even if they should not be powerful views, they are useful because they will declare, in the database, exactly what we will use in a specific cube. This information is very important to track cube dependencies.

Once more, we are not using a feature of SSAS (named queries in the data source view) in order to share the information about how we will gather data for the cube with people that has potentially no access to

the SSAS solution. You should have understood that – in our vision – information sharing between people working on the project is very important.

Configuration database

The configuration database is the place where you store all the configuration tables of your solution.

There are no generic rules about this database because it is very customer specific. It is advisable to use the same subject area separation used throughout all the other databases by using database schemas but, upon that, all the other configurations depend on the specific business area on which we are working.

One important thing to remember is that we normally use the "simple recovery mode" for all the data warehouse databases, as it is normally useless to get full recovery of a database used for ETL. The configuration database is different, because it is a user updatable database and, for this reason, we should always use the full recoverable mode on the configuration database.

Log database

The log database is one of the most difficult to project. Sometimes the log database is underestimated and we will end up without logging information that clearly states who did what and when, which processes run and which did not. Sometimes the log database is overestimated and becomes a project by itself, requiring a specific cube to navigate it. Both these situations are extreme and we should carefully avoid them.

In this chapter, we will spend some words about what – in our opinion – to log and when, trying to give some hints and tips in order to avoid falling in one of the situations we have described.

- The log database should be independent on the other databases

  This may seems easy but, in reality, it is not. The keys and values in the log database should not refer to any other database. This is important because all the other databases will change over time while the log database may last for a long period.

  Moreover, the log is usually related to errors i.e. rows that will not be inserted into the standard database for any reason. By viewing the log database, we should be able to gather all the information needed to solve the specific problem we are facing.

- The log database should clear by itself automatically

  It is useless to maintain a long period of detailed logging. After one year the detailed log information are no more useful.

In our experience, the log is very useful to catch errors in the ETL phase, where one or more of the packages might fail and compute erratic values. Sometimes, the problem arises several days after it happened, because the user did not query the cube in that period or the data affected by the problem were not in queries. This should not happen if we designed a correct error system. The user will have an evidence of whether everything went ok or not.

The importance of auditing tables is somehow underestimated. In our opinion, auditing tables should reside in the data mart, might be part of the cubes and hidden to the user but always accessible to the BI manager in order to detect what happened during both the ETL process and the cube process.

However, a good question is: "what is the difference between log and audit?"

Log is a technical stuff. It contains information about which package started and ended correctly or with errors. It will tell you how many rows the ETL read and how many of them were correct or not. Information like these is of no use to any OLAP user.

On the other side, auditing contains information that may be useful to the users. Information like the last day that has been loaded, the exact number of shops that sent their sale information and when. These information are useful because an user may be interested in analyze data but might be induced to believe

that he is looking at a complete set of information while – for some reason – some data is not completely updated.

Auditing is normally a subset of logging. However, while log data resides in the log database, auditing resides in the data warehouse and will have its fact tables (and potentially dimensions) in order to expose data as any other cube does.

Disk optimization

Now that we have defined the technical stuff about the various databases, we can spend some words on how to obtain the maximum ETL and cube process speed of our BI solution.

We are not going deep into technical details about how to optimize a disk subsystem; this will be the work of specialists. Instead, supposing to have more than one disk system, we want to point out some considerations about where to store each database in order to get the most from the server.

The rules are indeed very easy:

- Try not to read and write on the same disk
- Use fastest disks for the task and database that will need disk speed
- Use largest and slowest disk for the storage of persistent data

How can you reach such results? The answer is to analyze the ETL pipeline, understanding which databases we read and which we write during a specific phase of the ETL pipeline. After we will have created these couples of databases, we will be able to verify that they do not reside on the same disk.

In the picture, each box represents a database and contains a number that represents the disk into which the database can be stored.

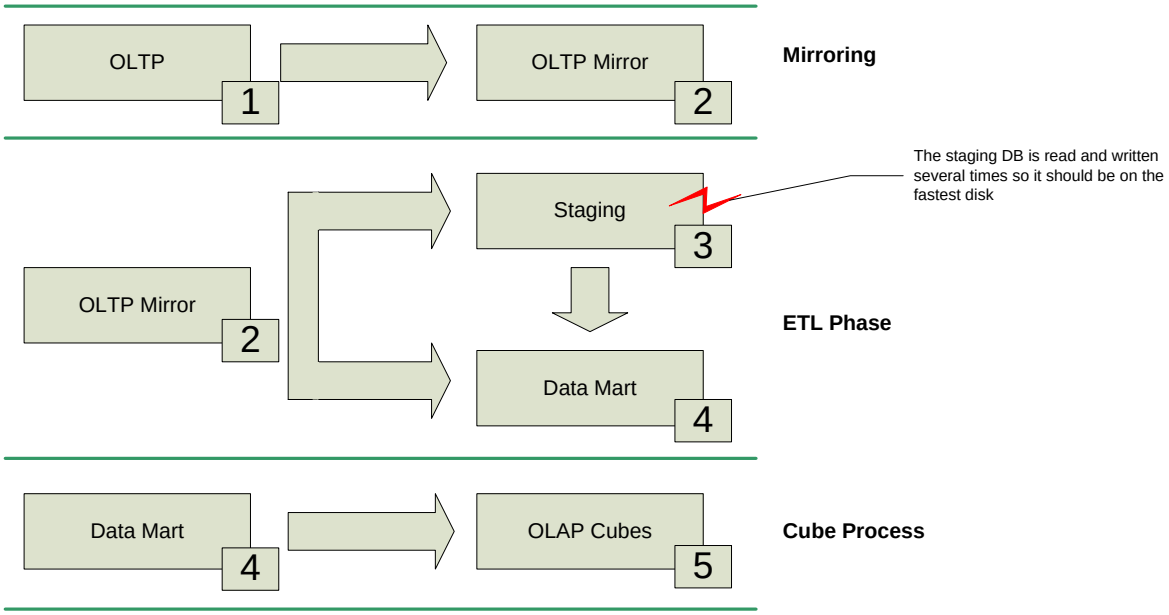

Figure 12 – (IMG 0025) Disk usage

Clearly if the OLTP system resides on a remote server, the numbers will go from 2 to 5 instead of 1 to 5. You will notice that the configuration database is missing. This is because we will read and potentially write the configuration database during the whole ETL pipeline. Adding it would have made the picture less clear. A good place for the Configuration database is on disk number 2, where the OLTP Mirror database resides.

The Log database is missing too, but it should have a very limited amount of traffic and it is not a crucial decision where to store it. We can move it wherever we want.

The best situation is when we have enough disks to move each database to its storage system. This will lead to optimal performances. However, when you build a small or medium data warehouse, chances are that you do not have a very high number of disks. In that case, you can try to optimize your performance trying to avoid that each single step (mirroring, ETL and cube process) read and write on the same disk.

Please note that the Staging database is the most crucial one. It will contain temporary tables that we will write and read during the ETL phase. We will read, update, rewrite and index each table during the ETL. If we have limited very fast disks, their best usage is that of the Staging database.

If we have fewer disks then we will have to decide which databases to store on the same disk. Using the picture is very easy to detect that – having only four disks – the OLAP cubes and OLTP mirror databases can reside on the same disk without degrading performances, because the OLTP mirror is sleepy during the processing of OLAP cubes. Using these disks for writing the data of OLAP cubes will cause no degradation of performance. If – on the other side – we place both the data mart and OLAP cubes on the same disk, probably we will be in a situation where we read from and write to the same disk, lowering in this way the overall performance of the process.

As we said before, if the fast disks we are using for staging are big enough to contain the OLAP cubes, then we can process them on these disks. During processing of the OLAP cubes, SSAS will read and write huge amount of data to the disks. You can consider making the process cube on the fastest disks (number 3) and then, when processing is finished, move the data to a bigger disk where the cubes can be stored safely, freeing the fastest disks for some other purpose.

Always remember that these are not fixed and immutable rules. Your specific implementation of the architecture will probably be constrained by other considerations. You – as the BI analyst – are the one and only person that will be able to take these decisions in a correct way.