Building a Business Intelligence System with the Pentaho BI Suite

J O H A N S Ö D E R L U N D

KTH Computer Science
and Communication

Master of Science Thesis
Stockholm, Sweden 2011

Building a Business Intelligence System with the Pentaho BI Suite

J O H A N S Ö D E R L U N D

Master's Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2011
Supervisor at CSC was Jens Lagergren
Examiner was Anders Lansner

TRITA-CSC-E 2011:106
ISRN-KTH/CSC/E--11/106--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Abstract

Virtual data is an ever increasing part of any large organization today. The data can be anything from transactional data coming from the day-to-day operation to customer and product information. Business Intelligence is an area that uses organizational data to perform analyses and create reports. The reports and analyses are great resources and can be used to give a better overview of the organization and help making better business decisions.

The purpose of the thesis was to investigate the possibility of creating a fully functional Business Intelligence-system by only using intelligent tools and not writing any program code at all.

A Business Intelligence-system consists of several parts and for each there is a wide variety of tools available both free and for a cost. The Pentaho BI Suite was chosen to build the system with. It is a complete BI suite that contains tools for all parts of a BI-system. The suite is available both as a commercial enterprise edition and a free community edition. The free community edition was used.

The investigation covered five issues in depth:

- Was it possible at all to create a BI-system without writing any code?
- What knowledge and skills was required to set up the system?
- What knowledge and skills was required for an end user?
- What were the major challenges in designing and implementing the system?
- How was the performance of the system and how could it be improved?

At the end of the investigation a fully functional and reliable BI system had been created. The system could successfully be used to do complex analyses and create informative reports. These could then in turn supply decision makers in an organization with facts to help them make better business decisions.

Referat

Konstruktion av ett Business Intelligence-system med Pentaho BI Suite.

Virtuell data är något som allt mer ökar i mängd hos stora organisationer och företag idag. Data kan finnas i en mängd olika system. Exempel är transaktionssystem, kund- och produktdatabaser eller system som täcker andra delar av affärsverksamheten. Business Intelligence är ett område som använder sig av tillgänglig data i en organisation för att med hjälp av den utföra avancerade analyser och skapa tillförlitliga rapporter om verksamheten. Analyserna och rapporterna kan sedan användas av beslutsfattare inom organisationen som stöd när affärsbeslut tas.

Detta examensarbete hade som mål att skapa ett business intelligence-system med hjälp av Pentaho BI suite – en paketlösning för Business Intelligence byggd på öppen källkod. Idén var att undersöka om det var möjligt att konstruera ett fungerande BI-system endast med hjälp av smarta verktyg. Helt enkelt skulle inte en enda rad programkod behöva skrivas. Fem frågeställningar undersöktes:

- Är det möjligt att skapa ett BI-system helt utan att skriva egen programkod?
- Vad behövs det för kompetens och kunskap för att utforma och konstruera systemet?
- Vad krävs av en slutanvändare för att kunna använda sytemet?
- Vilka stora utmaningar finns då systemet skall utformas och konstrueras?
- Vilken prestenda skulle systemet få och på vilka sätt kunde den förbättras?

Slutresultatet var ett funktionsdugligt och stabilt BI-system skapat med de verktyg som fanns i Pentaho BI suite. Systemet kunde användas för att göra komplexa analyser och skapa informativa rapporter. Dessa kunde i sin tur ha använts som stöd för affärsbeslut och för att ge tydliga överblickar över affärsverksamheten.

Contents

Abbreviations and technical terms

Abbreviation	Technical Term
BI	Business Intelligence
PQAT	Product Quality Assessment Tool
DSS	Decision Support System
ETL	Extraction, Transformation and Loading
DW	Data Warehouse
OLAP	Online Analytical Processing
ROLAP	Relational Online Analytical Processing
MOLAP	Multidimensional Online Analytical Processing
PDI	Pentaho Data Integration
SW	Schema Workbench
PAD	Pentaho Aggregation Designer
PRD	Pentaho Report Designer
DBMS	Database Management System
RDBMS	Relational Database Management System
CDC	Changed Data Capture
MDX	Multidimensional Expressions
SQL	Structured Query Language
XML	Extensible Markup Language
WAQR	Web Ad Hoc Query and Reporting Client
WYSIWYG	What You See Is What You Get

Introduction

Every business organization has some kind of information about itself and their operation. Data about products, employees, customers, sales and resources is today necessary to run a successful business [1]. External data - such as market trends and demographics - is also an important asset for an enterprise.

Today most data, if not all, is stored electronically. The most common choice is to store it in databases, but other solutions, such as storing it in plain files, also exists. While the operational data is crucial for an enterprise to function since it supports the day-to-day operation it can also be used for analysis. Analyzing operational and external data is a great help for managers, executives and others whose position makes their decisions affect the business. Being able to answer questions about the business such as "How much does a product sell in a certain area?" or "How much did our sales campaign affect our profit?" is important when making business decisions.

It is safe to assume that organizations have always looked to their own data for help in making important business decisions, but without a system to efficiently store and analyze the data this is not an easy task. A computer-based system is required to do complex analyses of the data. Decision Support Systems (DSS) has been around since the 1960s[2]. A DSS calculates answers according to one or more defined models. This makes it a good tool in cases where only a model is sufficient but many business decisions requires analyses made only on the organizations business data.

The ability to efficiently store and analyze business data is today a necessity for every large organization[3]. The data available has continuously increased since companies first started storing their business data electronically. One factor is that keeping historical data is important for analyses [4], so historical data is not often removed. Another factor is that more and more data becomes available with modern technological advancement. Furthermore people in charge making decisions want to be able to ask very complex questions very fast, probably asking more than just one question at a time. To meet these demands Business Intelligence (BI) emerged as an offspring of the DSSs [5].

The term Business Intelligence was proposed in 1989 by Howard Dresner, he described it as *"a set of concepts and methods for improving the decision process, using support systems based on facts"*[6].

Business Intelligence is today commonly seen as a term that refers to computer-based methods, tools and approaches used to analyze and interpret business data [7].

Background

Omicron Ceti AB is a consulting firm based in Kista, Sweden. Approximately five years ago they developed a quality analysis tool for Ericsson AB. The system is called Product Quality Assessment Tool (PQAT) and is still used by Ericsson today. The purpose of PQAT is to provide an accurate analysis of returns and repairs on Ericsson's products based on their product and return data repositories. PQAT is a home grown system made entirely by Omicron for this specific purpose. Omicron is now interested in looking at other solutions for future business opportunities. What is important to mention here is that PQAT, and other product quality systems can be implemented as a BI system (though a more narrow one since it only handles product quality). This is good since the purpose of a BI system is to give accurate and fast answers to various analyses, which is also the goal for a product quality system.

Why is it similar to a BI system then? The input to the system is a lot of data from many different operational system concerning everything about a company's products. This data is then loaded, cleansed and stored in a repository used only by the product quality system. The data is then used to give accurate analyses - in the form of chart, graphs and numbers - to people making decisions. As will be shown in the *Concepts* chapter of the thesis these functions is exactly what makes up a BI system.

Purpose

The purpose of the thesis was to investigate if a quality analysis tool could be made without traditional programming and instead make use of already existing BI-tools. The following issues were addressed in the thesis:

- Was it possible at all to create a BI-system without writing any code?
- What knowledge and skills were required to set up the system?
- What knowledge and skills were required for an end user?
- What were the major challenges in designing and implementing the system?
- How was the performance of the system and how could it be improved?

The end goal of the BI-system was to present three business values in various analyses: The amount of units that has been shipped, the amount of units that has been returned and the error percentage of the units (returned units / shipped units).

There are numerous BI-tools available today, both free open source solutions and commercial counterparts. The Pentaho BI suite was used to create the BI-system. It is an open source system with both a free community edition and an enterprise edition. Being open source makes it good for this thesis's purpose, since it uses a lot of the most popular freely available BI-tools. It it also a complete BI suite meaning it contains tools for all the parts in a BI system.

Delimitation

The purpose of the thesis is not to compare the Pentaho BI suite to other BI-solutions that exists. Neither was any other open source solutions tried as an alternative, since that would greatly increase the workload. The thesis does not explain in detail all the different methods and approaches that exists when creating a BI system, this is a whole thesis or book in itself and many have been written regarding that subject (e.g. [8][9]). An overview will be given of how a BI-system works and what parts it contains. This is required to understand the methodology choices, design discussions and system implementations discussed later on in the thesis.

Theory

This section will give an overview of how a BI-system works and what the different components are. The operation of a BI-system can be viewed as a continuous process which starts in the organizations operational systems and ends with reports and analyses. The process consists of several parts and can be defined as follows:

Source systems → ETL → DW → Analysis and Reporting tools → End user environments

Figure 3-1 shows a good overview of the process. Lets explain briefly how the parts connect to each other and their purpose.

The source systems is were data can be gathered from and consists of the organization operational systems and external data sources. A central point of BI is to gather all available data that is of interest and store it in a centralized repository. The repository is called a *Data Warehouse (DW)*. The basic idea of a DW is not just to have all data in a single storage point but also to have it error free, conformed and named and formatted so that it is easy to understand for a business user. Data has to be gathered, transformed and loaded into the DW from the source systems and this is what the *ETL* (short for Extraction, Transformation and Loading) does. When all data is available in the DW analyses and reports can be made with it. For this purpose there are analysis and reporting tools. The most common analytical approaches used in BI are *Online Analytical Processing (OLAP)* and *Data Mining*. OLAP is a method of translating relational data models into multidimensional models which gives it the ability to swiftly answer multidimensional analytical queries. Data Mining is an approach of analyzing data to find hidden patterns in it. The end user environments are the graphical interfaces that present the reports and analysis result to the end users, which are mostly decision makers or other business users. This is how a BI-system works with the different parts working together to create the information chain from the source systems to the end user.

This was a short overview of a BI-system and the sections of this chapter will cover each part of the BI-system in depth.

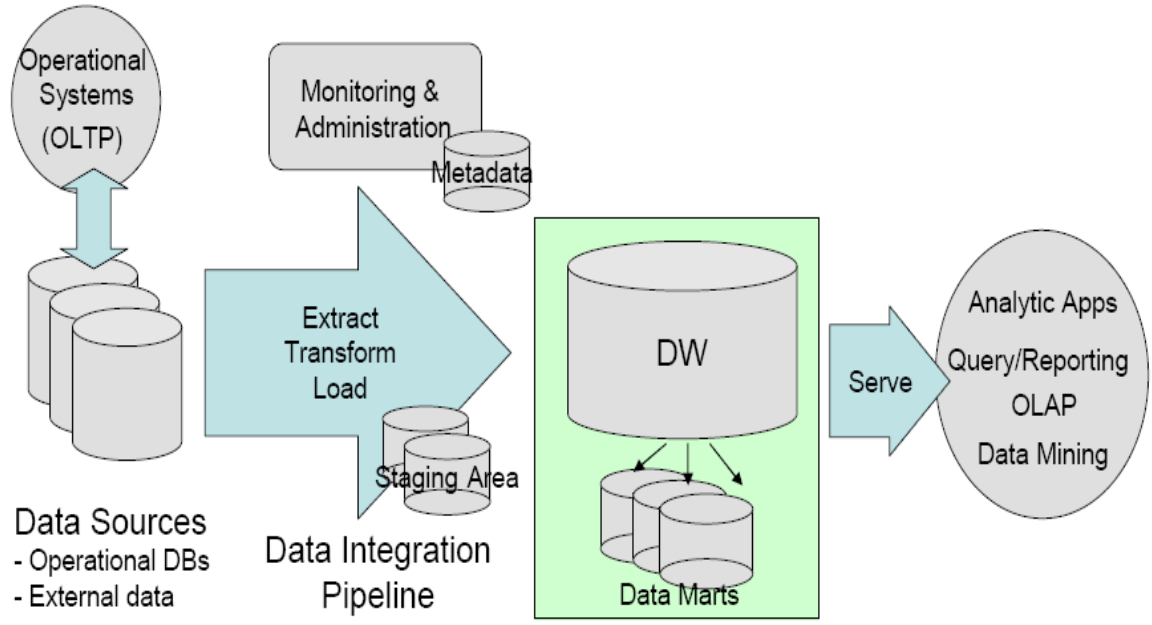

Figure 3-1: Overview of a BI system [P1]

The Data warehouse

A Data Warehouse (DW) is a storage of the organizations data apart from the operational systems where data can be accessed for quick analyses.

The purpose of a DW

A good question here is why a DW is necessary. Can't the data just be accessed directly from the operational system? To understand why a DW is important one can look at what problems arise if a DW is not present when a BI system is deployed and used.

Assume that a company wants to use their data from their operational systems for analyses. This is a very big step towards better business decisions for the company but putting the workload directly on the operational systems has many drawbacks:

- Many organizations have their data in more than just one repository, and to be able to get a correct analysis all relevant data has to be available. This means that many analyses require data from many different systems. The storage method can also differ from repository to repository and can be anything from storage in plain files to a RDBM or other database storage solutions [4]**.** This creates a serious problem when collecting the data since the analysis tools would need to get the data from many different systems that may store the data very differently from each other. Another problem is that the performance of storage systems can vary a lot, which can result in one system being a bottleneck in every analysis its data is used in.
- The data needs to be available instantly when a user is doing an analysis. Having data accessed and collected from the operational system every time a query is made from the analysis tools would put extra pressure on those systems and severely impact their

performance. Slowing down the operational systems used all the time in the day-to-day operation is not good for an enterprise. The operational systems are also not designed to handle the complex queries coming from the analysis tools. These queries require big amounts of data and operational systems are optimized for data transactions, not large retrievals[10].

- Operational systems are often not interested in storing historical data since it is not needed to run the day-to-day operation. Therefor many analyses would give incorrect or insufficient representations of the business due to the lack of historical data.
- The data gathered from the operational systems can often be named in a non-descriptive way and many times only makes sense to the designers and users of those systems. For example a column of data named "r_prod_q" would make no sense to someone who doesn't know in advance what the column is for. Business users will not be able to understand these definitions of the data without having to ask those who designed the systems.

The DW is designed to take care of all of these problems. It is designed to handle retrieval of data efficiently and can in theory store infinite amounts of data, thereby supporting the storage of all present as well as all historical data from all of the operational systems. The idea is to collect all data from all of the organization's systems - those where it has been deemed necessary to collect data from - and store it in the DW. The data is conformed to formatting and naming standards created by the organization and clear and uniformed definitions of what different data represents exists. This provides a sole centralized storage for all data that can be used in analyses. The data is also presented in a standardized and understandable way to an end user.

Designing a DW – Inmon vs Kimball

There are two major schools concerning the design of a data warehouse. One was described by Bill Inmon in 1990 when he published his book *Building the Data Warehouse* [8] . Ralph Kimball had another approach in mind and published his own book *The Data Warehouse Lifecycle Toolkit* [9] in 1996. The major differences in these two teachings can be summarized as follows:

Data marts: While Inmon saw the data warehouse as a monolithic storage for all the data that an enterprise has Kimball suggested the use of data marts. A data mart is a projection of part of the data that exists. The idea is to have one data mart per business function. For example one data mart could be production while another would be sales. This creates an easy way to select and administer what data different parts of an enterprise needs and should have access to. This approach also makes use of *conformed dimensions*, which means that several data marts can use the same dimension such as a production and sales data mart would both need a date dimension. Instead of having two date dimensions they can use the same shared date dimension.

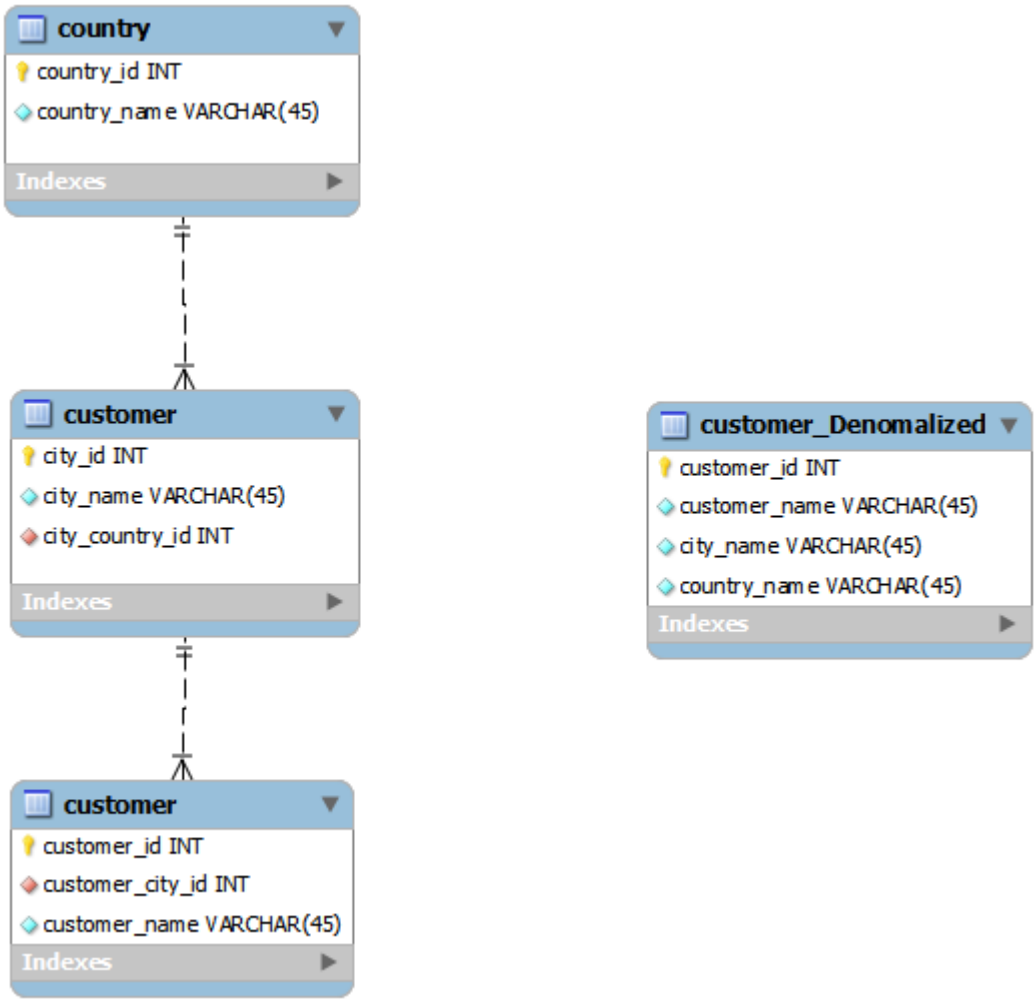

Figure 3-2: Inmon's approach on the left and Kimball's on the right

Normalized vs dimensional modeling: Whether or not to have data marts is not the only thing Inmon and Kimball disagrees on. Another difference is how the data warehouse should be physically structured. In Inmon's case the data warehouse should be a standard normalized database. Kimball introduced the idea of using de-normalized dimension tables, making dimension tables hold every information needed to describe the content in them. For an example in a normalized approach a dimension table holding information about customers would have extra tables describing attributes such as city and country. In Kimball's approach all of these tables would be merged in one customer dimension table making it withhold every information about customers. This will of course lead to redundancy in information since for example a factory dimension table would also need to contain information about a factory's location.

This thesis will use Kimball's approach when designing a DW - why this approach is chosen will be explained in the methodology section – and thus that approach will be explained in more detail in the following sections.

Dimensional modeling

The idea of dimensional modeling is to store data in a multidimensional model instead of a traditional normalized one often used in transactional operational systems. The dimensional model consists of two types of tables, *facts tables* and *dimension tables* [9].

The dimension tables holds pure information about different areas, such as products, customers or time.

Fact tables contains *measures* (also called *facts* or *metrics*) and keys to dimension tables.

The measures are business values that the organization wants to measure, for example revenue or number of units sold. The dimension tables defines how these values should be measured. To make the explanation easier lets assume that we have a company called *Computer Store*.

Computer store sells computers and at the moment has one store. The business has just started and they only sells three products – three different laptops. The store has successfully sold three products since the start. *Figure 3-3* shows Computer Stores transactional database that keeps track of their sales.

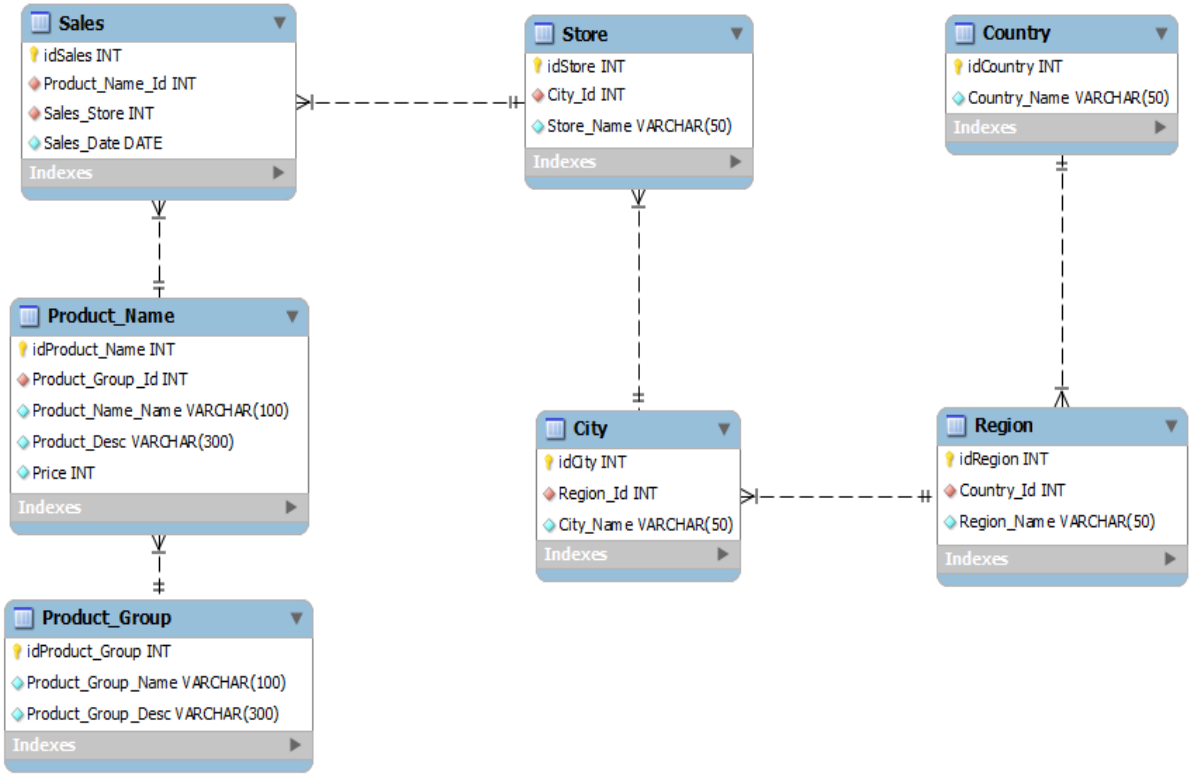

Figure 3-3: Computer Store's operational database

Now Computer Store wants to create a data warehouse to use for future analyses. The measure they are interested in, as a start, is the revenue for sales. With a measure specified the next step is to look at what dimensions are avaiable.

The transactional database contains information about products, stores and sales. This can be translated into three dimensions:

- A product dimension with information about the products they are selling.
- A store dimension with information about their stores.
- A date dimension to identify when sales have occurred.

Figure 3-4 shows a DW using these three dimensions to measure revenue.

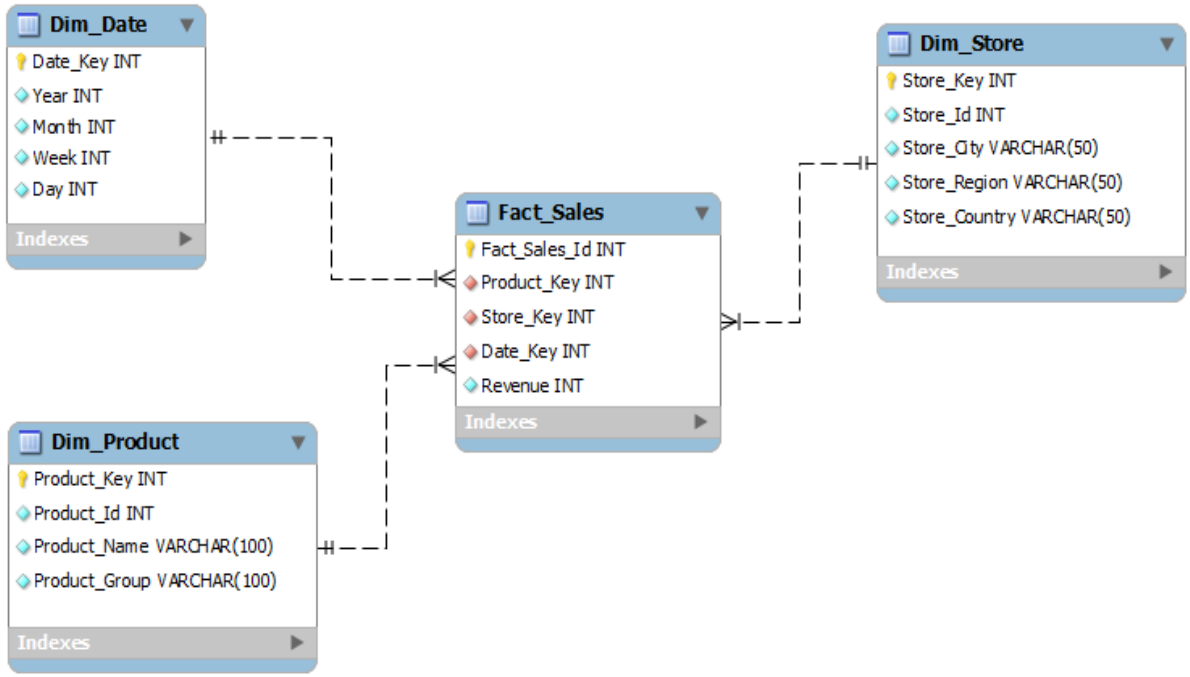

Figure 3-4: Computer Store's DW

The dimension tables are date, store and product. The fact table is made up of five columns: *Revenue, Units_Sold, Date_Id, Store_Id* and *Product_Id*. *Revenue* is a measure and gives the total revenue. The other three columns holds keys to rows in the dimension tables.

The value of *Units_Sold* is determined by the composition of the dimension keys. Lets do a brief example to give a clearer image of how the data is represented in the DW. Assume that we have two stores (store1 and store2), two products (product1 and product2) and three sales registered in the operational database. Two of the sales (sale1 and sale2) were made at store1 for product2 and one sale (sale3) was made at store2 for product1. Sale1 occurred 2011-01-07, sale 2 occurred 2011-01-10 and sale3 occurred 2011-01-11. This data would then be present in the operational database. The DW is at this point empty and data gets copied to it with the ETL tool. *Tables 3-1, 3-2* and *3-3* shows how the data would look like in the DW after the transfer. The date dimension tables data is not interesting since it just contains auto-generated dates. Lets say it has dates covering the whole of 2011.

Store_Key	Store_Id	Store_City	Store_Region	Store_Country
1	1	Stockholm	SL	Sweden
2	2	Stockholm	SL	Sweden

Table 3-1: The data in the store dimension table (dim_store)

Product_Key	Product_Id	Product_Name	Product_Group
1	1	Dell 2100	PC
2	2	iPad	MAC

Table 3-2: The data in the product dimension table (dim_product)

Fact_Sales_Id	Product_Key	Store_Key	Date_Key	Revenue
1	2	1	20110107	600
2	2	1	20110110	600
3	1	2	20110111	1200

Table 3-2: The data in the fact table (fact_sales)

Now for example if someone at computer store wanted to know how much the total revenue for store1 was they would just filter the fact table so that only rows where the Product_Key were equal to 1 would be selected and then sum the revenue field. Another example is the total revenue for a specific product from a specific store. That would be achieved by a filter using both the product key column and the store key column, which is very easy to do. This shows the potential of a good DW - important questions about the business can be answered with little effort.

Star and snowflake schema

There are two different models that a data warehouse can be modeled after: S*tar schema* and S*nowflake schema*. The common identifier for both models is the fact table which in both models holds the measures and keys to the dimension tables. What differs the two models from each other is that in a star schema every dimension table is completely denormalized and covers a whole area of the business, the Computer Store's DW model *(Figure 3-4)* is an example of a star schema. A snowflake schema on the other hand have different degrees of normalized dimensions split into multiple related tables. *Figure 3-5* shows Computer Store's DW modeled as a snowflake schema.

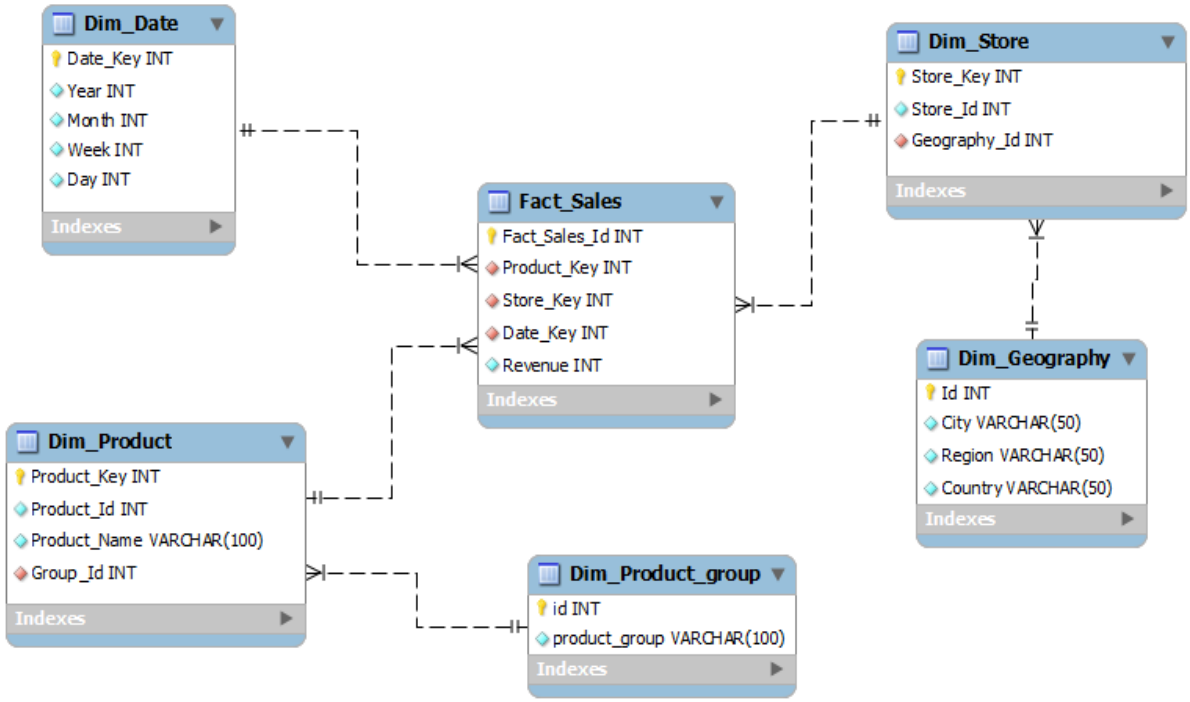

Figure 3-5: Computer Store DW with a snowflake schema

Granularity

Granularity is important to mention when talking about data warehouses because it determines how big the fact table will be [12]. The granularity of a fact table is the lowest atomic level of which a fact is defined [13]. The grain of a measure is a combination of the granularity of the dimension tables. Let's take the example of the revenue measure with the date and product dimensions again. The grain of the revenue measure will be the grain of the date and product dimension combined. So if the lowest atomic level of the date dimension is day and for the product dimension it is product subgroup then the granularity of revenue will be revenue by day by product subgroup.

ETL

The acronym ETL (short for Extraction, Transformation and Loading) is the part of the BI system that handles the initial data coming from the different source systems. It consists of three steps (hence its name).

Extraction

The actual data transaction from the source systems (often operational systems such as RDBMS used in the day-to-day operation) to the DW. A very important part of the extraction is to identify which data has been changed, inserted or deleted in the source system since the last data update, this process is called *Changed Data Capture (CDC)* [14]. Why is it so important? Without CDC the ETL would have to load all data from the source systems every

time data is to be updated. This is suboptimal and would take too much time as well as create unnecessary pressure on the source systems.

Transformation

There are several problems when dealing with the data from the source systems. A lot of systems have humans inserting the data, and it is no secret that humans make mistakes resulting in erroneous data. Duplication of data is another problem, for example different databases can have the same product in their data thus the product is defined twice and redundant information exists. Formatting of data can also differ from system to system creating more problems. For example a date can be formatted in many ways: yyyy-mm-dd, dd/mm-yyyy, mm/dd-yyyy are just three examples.

Another issue that needs to be addressed is the naming of the data. Many times it is only clear to the designer or the current users what the data represents. For example a column named prod_m marking the model of a product could be renamed to product_model. This is also a goal of the DW - to present data in a way that makes it easy to understand.

All of these problems can be addressed directly in the ETL. The goal of the transformation part is to cleanse, conform and alter data so that it is coherent, clean and fits in the data model used in the DW.

Loading

This is the actual loading of the data into the DW. Dimension keys needs to be looked up every time a fact row is inserted. Thus loading data into the fact table is a resource consuming process.

Staging area

A staging area is an alternative to transferring the data directly from the source systems to the DW. The staging area can be seen as a temporary storage between the source system and DW and the data is copied straight as it is from the source systems to the staging area. This approach puts the least pressure possible on the source systems since the data is only extracted before it is put in the staging area. Less pressure on the source systems is always a desirable since they are often critical to the organizations operation. When the data has been copied to the staging area more major operations can be made, such as transforming the data or analyzing it to create meta-data about erroneous entries.

OLAP and analysis

When it comes to creating reports based on the company data the DW is a fantastic source for that purpose. But when doing complex analyses it a DW starts to lack in performance. After all it is just a flat relational database, even if it is modeled in a multidimensional fashion.

Online Analytical Processing (OLAP) is a category of database processing used for multidimensional analyses [15]. This gives OLAP the capabilities to do complex analytical and ad-hoc queries very swiftly.

OLAP systems are categorized depending on how the underlying data is stored and traditionally [16] there are three variants:

Multidimensional, MOLAP – Data is stored in an optimized multidimensional array storage.

Relational, ROLAP – Data is stored in a relational database. New tables are created to hold aggregated values. Defined by schema files written in *Extensible Markup Language (XML)*.

Hybrid – Data is divided between relational and specialized storage.

The whole idea of an OLAP system is to create *OLAP cubes*. The cubes are what is used when querying the system. Without a cube there is nothing that can be queried.

Cubes, dimensions and measures

It was mentioned earlier that a star schema is a good choice for a DW since it translates well into an OLAP cube. This is true since an OLAP cube is a collection of *dimensions* which defines values of *measures*. The dimensions creates a multidimensional structure of cells in which each cell holds values of one or more measures. The measures are the things you want to report on such as revenue or number of sold units. Here it becomes clear that a star schema in fact can be translated almost directly into an OLAP cube with the dimensions tables being the dimensions and the fact table measures being the cubes measures. What is important to mention though is that the *cube measures* are aggregated values and not single atomic facts. How the values are aggregated depends on how the measure is defined. Common predefined aggregation functions are *count, sum* and *average*. Users can also define their own aggregation functions and thereby tailor the measures in any way they want.

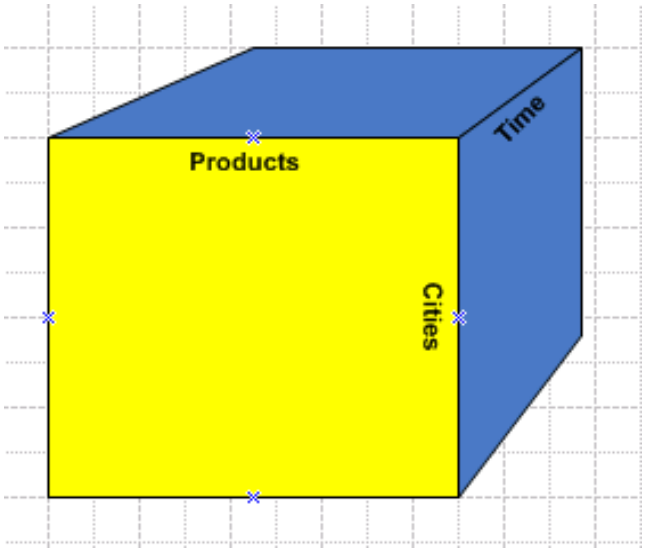

Figure 3-5: An OLAP cube

A cube consists of dimensions and measures, but it is also important to mention that the dimensions has hierarchies. A hierarchy in a dimension describes how the data should be ordered by granularity. Every hierarchy has levels, the lowest level is the highest granularity.

A dimension must contain at least one hierarchy but can have several. Here is an example to make it more clear:

A date dimension exists in a cube. The dimension has two hierarchies. One that describes months, and one that describes weeks. The levels for the month hierarchy are in order of granularity from lowest to highest: Year - Month - Day of Month.

And for the week hierarchy it looks like this: Year – Week – Day of week.

This means of course that the database table the data is gathered from must have fields with information about the year, month, week, day of month and day of week.

With these hierarchies defined the OLAP server knows how to order the data in the cube.

Data from OLAP cubes are retrieved with a querying language called MDX, abbreviation for *Multidimensional Expressions*. It was originally developed by Microsoft and is today a de facto standard for querying OLAP databases.

Further information about MDX and its syntax can be found here:

http://www.iccube.com/support/documentation/mdx_tutorial/gentle_introduction.html

http://en.wikipedia.org/wiki/Multidimensional_Expressions

Reporting and analysis tools

There are many reporting and analysis tools out there but they all have common characteristics. The purpose of the reporting tools is to be able to connect them to a database, preferably a DW, and retrieve wanted data to structure a report with. Often a storage function is provided to store already made reports and use them again as the data is updated. Other features that are often present are scheduling of report creation and the ability to create graphs.

The analysis tools often has to do with visualization of OLAP cubes and data mining. The concept of data mining will not be discussed but further reading can be found here:

http://en.wikipedia.org/wiki/Data_mining

It is important to remember that a BI system is not a magic box solving everything that's missing in an organization. In order to make a BI system worthwhile, the people using it must know the important questions and how to ask them. Selecting the data used in the BI system is also very critical since it defines the answers. A lack of data, or absence of vital data can result in faulty analyses that are believed to be correct. If business decisions are based on these faulty analyses it has done more harm than good and the whole purpose of the BI-system has been lost.

Pentaho BI suite overview

The Pentaho BI suite is made up of several tools that can be linked together to form a complete BI system. This chapter will give an overview of the tools available since most of them will be used to build the BI system.

Data Integration

The tool available for ETL in the Pentaho BI suite is *Pentaho Data Integration (PDI)*, also codenamed as Kettle. PDI has a graphical user interface where users can put together everything from simple to complex ETL solutions. PDI has the following building blocks:

Step: A step has a data input and a data output and data is streamed through the step. Every step has a function and when data is streamed through it it can be altered and/or filtered or just scanned to match patterns. There are many different steps available in PDI and for many users they are enough to do the trick. If they are not enough there are script steps available where either SQL or JavaScript can be used.

Transformation: A transformation is a container for steps. When a transformation is done and all the steps are linked together and has a proper input and output the user can run the transformation. Error logging is available to help the user see what went wrong if the transformation fails. What is important to mention here is that a transformation can be seen as a big stream. It has a start (input to file, table etc) and an end (output to file, table etc) and in between all of the data that has been fetched or created on the way is streamed through the whole transformation in the order it was brought into the transformation.

Job: A job consists of one or more transformations that will be run in order depending of how the job is set up. Data will not flow from one transformation to another, they will simply be executed in order. Jobs can be used to schedule tasks such as running all update transformations for dimension tables in a data warehouse.

Analysis

The OLAP server found in the Penatho BI suite is called Mondrian. Mondrian uses ROLAP technology and translates MDX queries to SQL used on a multidimensional model. Mondrian has caching and buffering capabilities which optimizes performance. It also has support for security roles that limits what data users have access to. The BI suite also contains *Mondrian* S*chema Workbench* (SW) and *Pentaho Aggregation Designer* (PAD). The workbench is a graphical tool for creating cube, a schema file is generated according to how the cube is designed in the GUI. The aggregation designer is used to add aggregate tables to speed up the cube performance. Tables can be added manually or the tool can suggest tables that it thinks would improve performance.

Reporting and presentation

There are two reporting tools available in the Pentaho BI suite. One is the web-based *Web Ad Hoc Query and Reporting Client (WAQR)*. WAQR offers generation of reports though it can

only use data predefined by meta-data models. Although the WAQR is easy and quick to use the data is limited to grouped lists and it does not support charts or graphs. In that aspect the other reporting tool, *Pentaho Report Designer (PDR)*, is much more powerful. With PDR a user can customize their report in almost any way and it has support for charts and graphs. PDR is not a *WYSIWYG (What You See Is What You Get)* editor so the user will have to work in a design environment that doesn't look like the final layout of the report. This requires some learning and understanding of the editor from the user.

The BI-server

The BI-server is a web server that can be accessed via a web interface. The server also has a solution repository to which solutions from the different tools can be published and accessed. In PDI this means transformations and jobs and from SW cube schemas. The repository can also contain reports and dashboards created in the web interface. The reports and dashboards are created with the WAQR mentioned in 4.3. The web interface also offers users to view pivot tables of OLAP cubes submitted to the repository from SW. The option of running pure MDX queries against cubes is also available. The BI server uses a preconfigured tomcat instance. Tomcat is an open source Java Servlet container developed by The Apache Software Foundation. To be able to access the BI-server login is required and it has full support for defining users and user roles.

By default the solutions repository, user and authentication data is stored in a preconfigured HSLQDB (which is a Relational DBMS)**.** The server can be reconfigured to use a user defined DBMS instead.

Method

This section will contain information about design choices and methods concerning the BI system that will be created with the Pentaho BI suite.

System machine

The BI-system was run on a laptop computer with the following specifications:

Processor: Intel Core 2 Duo T5850 (2.16 GHz)

System memory: 3.00 GB DDR2

OS: Windows 7 64-bit version.

Four different *Database Management Systems (DBMSs)* were tried as platforms for the data warehouse: MySQL, PostgreSQL, LucidDB and MonetDB. All of these are free open source DBMS with MySQL and PostgresSQL being the most well known. Both LucidDB and MonetDB are developed with a focus on data warehousing and business intelligence.

As for trying to increase the performance of the system no improvements were made directly in the OS. The improvements that were made during the work were done in the tools from the Pentaho BI suite and in the DBMSs.

Data generation

Since the data used in PQAT were not public information test data had to be created. The data model resembled but was not an exact copy of the one found in PQAT.

The data was placed in an operational database. Of course the operational database would not have any real users since it was solely created for the purpose of the thesis.

The database was designed according to what can be expected of a production database and the information available about PQATs database structure. The database contained information about products, customers, factories and error types. The database is shown below in *Figure 5-1*.

The last_modified field found in every table was used to imply when a row was last changed. This field was required for the CDC to work in the ETL, more on that in the *Implementation* chapter.

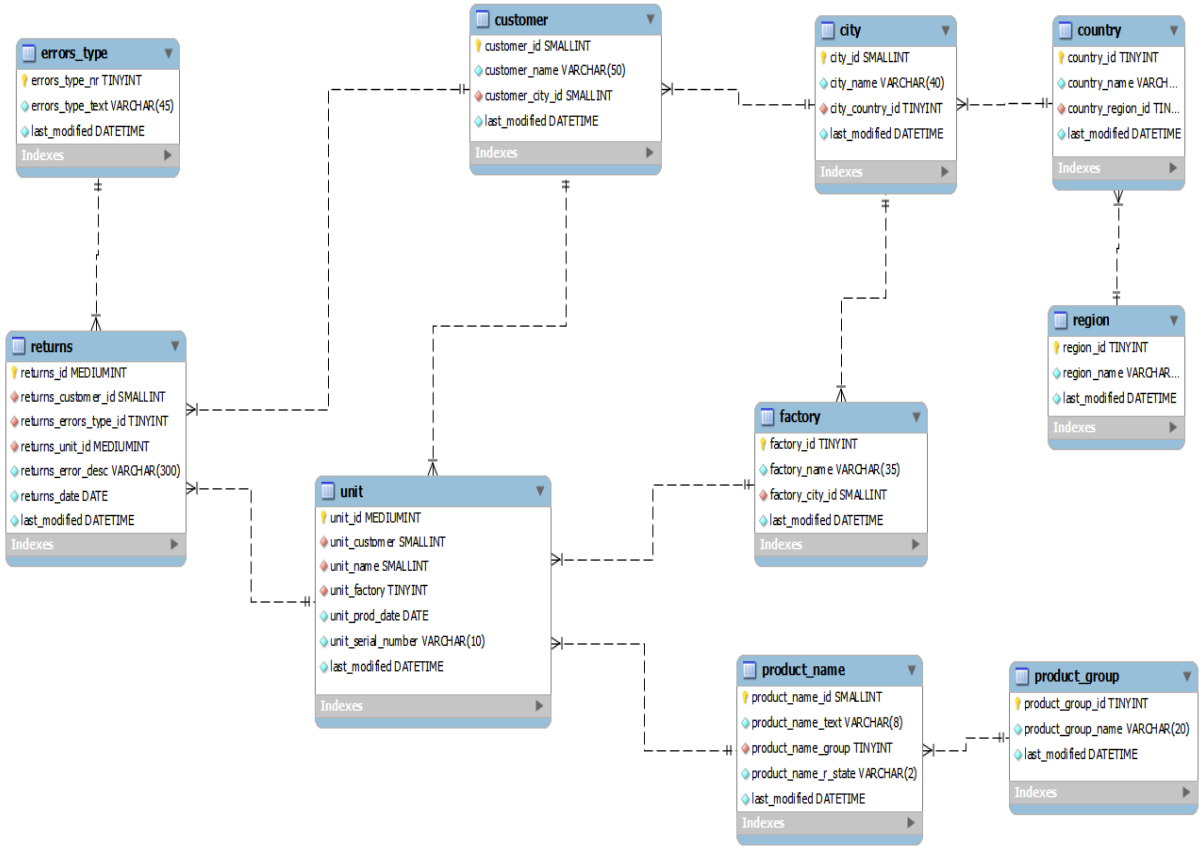

Figure 5-1: The operational database

Several copies of the database were created. Each with a different amount of produced units. The amount of returns was set to 10% of the amount of units in all databases. The high percent was chosen to make the analyses easier to perform since one of the quality measures we were interested in was the percent of units returned.

The purpose of having different amounts of units was to be able to evaluate the performance of the ETL and the OLAP cube. The purpose of having copies of the same database but with different amounts of units was because it proved easier than having to add or erase data every time another quantity of units was to be tested.

The six copies of the database that were created and the amount of units and returns inserted are shown in *Table 5-1* below.

Name	#Units	#Returns (10% of the units)
Prototype05	5000000 (5 million)	500000 (0.5 million)
Prototype10	10000000 (10 million)	1000000 (1 million)
Prototype20	20000000 (20 million)	2000000 (2 million)
Prototype30	30000000 (30 million)	3000000 (3 million)
Prototype40	40000000 (40 million)	4000000 (4 million)
Prototype50	50000000 (50 million)	5000000 (5 million)

Table 5-1: The various copies of the production database

Designing the Data Warehouse

The DW design were created using Kimball's approach. This was because the BI system were covering data for a certain part of an enterprise – a production environment. This made it ideal to design the DW as one data mart. Since data marts according to Kimball should represent exactly one part of a business operation one data mart was sufficient to cover a production system. Another reason for choosing Kimball's approach was the denormalized design suggested by Kimball. Having a normalized DW, as Inmon suggested, were not ideal since the focus of the DW was use it for analysis and build OLAP cubes from it. This made a denormalized design the better choice since was going to yield better performance to the cube. The ETL were also going to be faster with a denormalized design since key lookups necessary for inserting data into the DW would be faster with less tables to perform the lookups in.

The next design choice was wether to use a star schema or a snowflake schema. A star schema was chosen for the same reasons Kimball's denormalized approach was chosen. Less key lookups for faster ETL and easier translation to an OLAP cube. In Kimball's opinion snowflaking should only be used in special circumstances[9]. One could argue that for example geographic data for customers and factories should be put in it's own table. This would make the two dimension tables which has geographic data (the customer table and factory table) smaller and updates to geographic data would only occur in one place. This creates dependencies for the two dimension tables to a table holding geographic information and would slow down both the ETL and analyses and thus that design was not used.

The next step was to determine how many dimension tables there should be and how they would look. The following areas was determined to be of importance to analyses: product information, customer information, factory information, error information and time information.

After the dimensions had been decided the facts was the next step. The quality measures - facts - wanted were as mentioned earlier: number of produced units, number of returned units

and the error percentage. These three facts will be referenced to as *units*, *returns* and *ep* in the remainder of the text.

With the decision of using Kimball's method for modeling and having identified the dimensions and facts the DW was ready to be implemented.

It was decided that the database model for the DW were going to be done in *MySQL Workbench*.

The DW also needed a working CDC-solution. The CDC was going to be cross-implemented between the ETL and the DW with both of the systems being able to communicate data updates.

Building the ETL with PDI

The whole ETL-process was designed and implemented with the PDI. The major design choices were whether to use a staging area or not and how the CDC and data cleansing should be constructed.

The data transfer from the operational database to the DW were done with one transformation per dimension table. The fact table needed two transformations to load all data correctly, more on this later on.

Using a staging area was not necessary in the BI system since the operational databases were not used by anyone else and extra pressure on them from the ETL tool was not a problem.

A CDC-solution on the other hand was necessary to make the loading of data selective. The CDC-solution were implemented in the PDI transformations.

Notes on data cleansing

Data cleansing is really part of a broader topic – *data quality* – which in turn is a part of the subject *data governance*. It is up for major discussion whether or not data cleansing should take place in the ETL or not. Arguably the data should be cleansed and conformed at the root in the source systems. Having perfect data in the source systems is a very admirable goal, but in reality this is very hard. Many systems have manual inputs and as we all know people make mistakes. It is always easy to blame the human factor when looking for scapegoats but automatic systems can also create incorrect or false data due to various reasons such as malfunctions or bugs. This means that having correct data at the root is very tough work and not many organizations can manage that. So data cleansing is an important part of the ETL if an organization wants to be sure that the data used for analyses is correct and can be trusted. In the BI system built for this thesis no data cleansing was done since all of the data in the operational systems was automatically generated or predefined and thus could be considered safe. For the purpose of totality, challenges involving design and implementation of data cleansing will be discussed later.

OLAP schema and cube

The analysis part of the system consisted of cubes run on the Mondrian OLAP server. The cube schema were created with *Schema Workbench. Pentaho Aggregation Designer* were used to improve the performance of the cube. Both *Schema Workbench* and *Pentaho Aggregation Designer* are tools found in the Pentaho BI Suite.

Since the DW were designed as a star schema the cube looked very much the same as the DW itself. The measures in the cube were the same as the facts in the DW and the dimensions in the cube were the same as the dimension tables in the DW. This was one of the reasons why a star schema was used in the first place, a star schema translates almost seamlessly into an OLAP cube.

To test if the cube worked satisfactory and to test its performance, twelve analysis queries of varying complexity were made. These were run against the cube to measure the time it took to get an answer. The test cases can be viewed in Appendix A. A java program were written to query the cube, this also made it very easy to measure the response times.

The system as a whole

When the system had gotten its final design and been implemented each of the test databases (with their varying number of produced units) were used to test the system. Starting with moving data with the ETL from the operational system to the data warehouse, followed by creating OLAP cubes with Mondrian in order to answer a variety of analysis queries. The performance of both the ETL and Mondrian were measured in time.

Implementation

This chapter will go through how the different parts of the project were designed and implemented starting from the generation of the test data and ending with how the OLAP cubes were made. Every time a tool from the Pentaho BI suite has been used a brief explanation of how the tool works will be given so that the reader can follow the design process easier.

Data generation

The region, countries, cities and factories was manually written into script files and inserted into the DB, as were the product groups and error types. The customers were generated with the help of the open source program Spawner Data Generator.

For the generation of produced units for the test databases a java program was written. It used JDBC, the java API for database connection, to connect to the operational DB. The program goes through the customer table and generates between 1-10 orders from that customer. Each order is for a specific product and between 10 and 5000 units are ordered. Each unit gets a unique randomized serial number 10 characters long, allowed characters are A-Z and 0-9. After that a starting production date in the time-span 2005-01-01 to 2010-12-31 is generated. This is when the first unit in the order is produced, the rest of the units are randomly given a production date in the time-span starting date to (starting date + 100). All of the units in each order are considered to be produced from the same factory, which is also randomly picked for each order.

The same program also populates the returned table by selecting a specified number of units from the unit table and inserting them in the returned table. The return date is set to the production date + (1-1095) days, 1095 days being roughly three years.

Data warehouse construction

In the design decisions five dimension tables were identified. The following list shows how the dimension tables was implemented and describes their data and level of granularity:

Product – Holds information about the product. It was determined that the granularity here should be at the product name level. Data will be gathered from the *product_group*, *product_name* and *unit* tables in the operational database.

Customer – Holds information about all customers that have bought products. This table holds the customer name as well as geographic information about the customer. This results in a granularity at the customer name level. Data will be gathered from the *customer*, *city*, *country* and *continent* tables in the operational database.

Factory – Holds information about the factories that produces units. Same information as the customer dimension. Factory name and geographic location. Geographic information here is important to be able to see how much certain regions have produced. Data will be gathered from the *factory*, *city*, *country* and *continent* tables in the operational database.

Error – Contains all of the possible errors a unit can be tagged with. Contains the error number and the textual name of the error. Data will be gathered from the *errors_type* table in the operational database.

Date - The data in this table will represent time with a granularity of unique dates. The data will be generated with PDI and then inserted into the table. A span of 10 years will be sufficient to cover the production and return dates used in the generated test data in the operational databases. The dates will range from 2005-01-01 to 2015-01-01.

Every dimension table except the date dimension used auto-generated primary keys, such keys are called *surrogate keys*. The date dimension made use of a *natural* primary key in the form YYYYMMDD representing the date the row holds information about. Since dates are unique this creates a great primary key and as we will see later the use of this natural key will help speed up the ETL significantly. With the dimension tables ready it was time to decide on the fact table designs.

Five columns in the fact table were automatically determined by the dimensions tables and would hold foreign keys relating to them. What was left to add was what the fact or facts should be.. This means that there has to be a way to count the number of produced unit and the number of returned units. The error percentage can be derived from those two values by dividing returned units with produced units. Though before any decisions involving the fact table design was made another problem was noticed: there was something missing in the dimensions to get the proper grain for the quality measure.

The granularity of the units was at this point only at the unit name level as derived from the product dimension. Having the accuracy of the quality measures as high as possible is vital since that is the core purpose of the system. So the higher the granularity of the product dimension the more accurate the quality measure would be. The highest granularity possible for products are unique produced units. The problem that arises if we put unique units as the granularity in the product dimension is that it will be as long as the fact table in regard of the number of rows. This is not the purpose of a dimension table, they are supposed to be as short as possible. And having it as long as the fact table would create an enormous join each time filtering with the product dimension was required.

Instead of putting the unique units in the product dimension a single column dimension was added to the fact table. This single column dimension held the serial numbers for all units ever produced. This solution did keep the product dimension fairly short with the granularity at product name and the fact table became as long the amount of produced units.

With this solution all of the needed dimensions with correct granularity were in place and it was time to decide how the facts were to be represented.

There were actually two solutions tried, one was later discarded due to the second one being better. But let's take a look at both of them to see why one was chosen and the other one discarded.

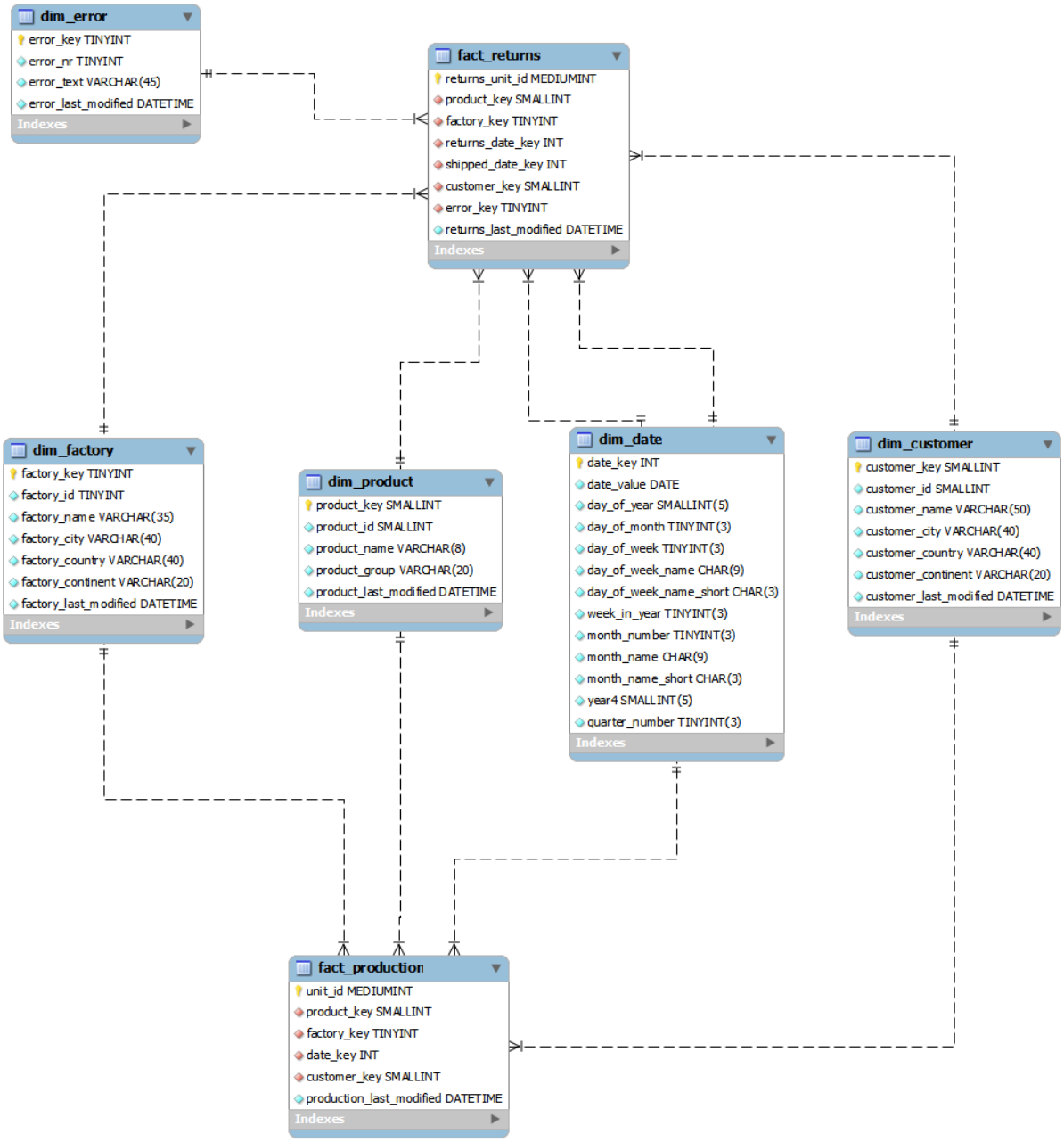

Figure 6-1: The first DW design

The first design used two fact tables, one holding produced units and the customer who bought them and one holding all the units that had been returned. This meant two data marts with five shared dimensions. The *units* fact was calculated by counting the number of rows in the unit_fact table since every row represents a unique unit. The *returns* fact was calculated in the same way from the fact_returned table. As mentioned earlier the third fact, the *ep*, could be derived from the two previous ones so all facts were now available as ready to be used in a cube. The design can be viewed in *Figure 6-1*.

At first this design looked good but it proved to be difficult when the cubes got taken into consideration. Since there were two fact tables, there also had to be two cubes - one for each fact table. To be able to get the most important quality measure (error percentage - total/returns) the two cubes created from each fact table had to be joined as a virtual cube. Or

the data - number of units and number of returns - had to be retrieved from both cubes each time and then divided. Creating a virtual cube or having two cubes at all felt too complex considering that the underlying DW really wasn't that advanced in its design. The design was remade.

Instead a single fact table was used. The improved design can be seen in *Figure 6-2*.

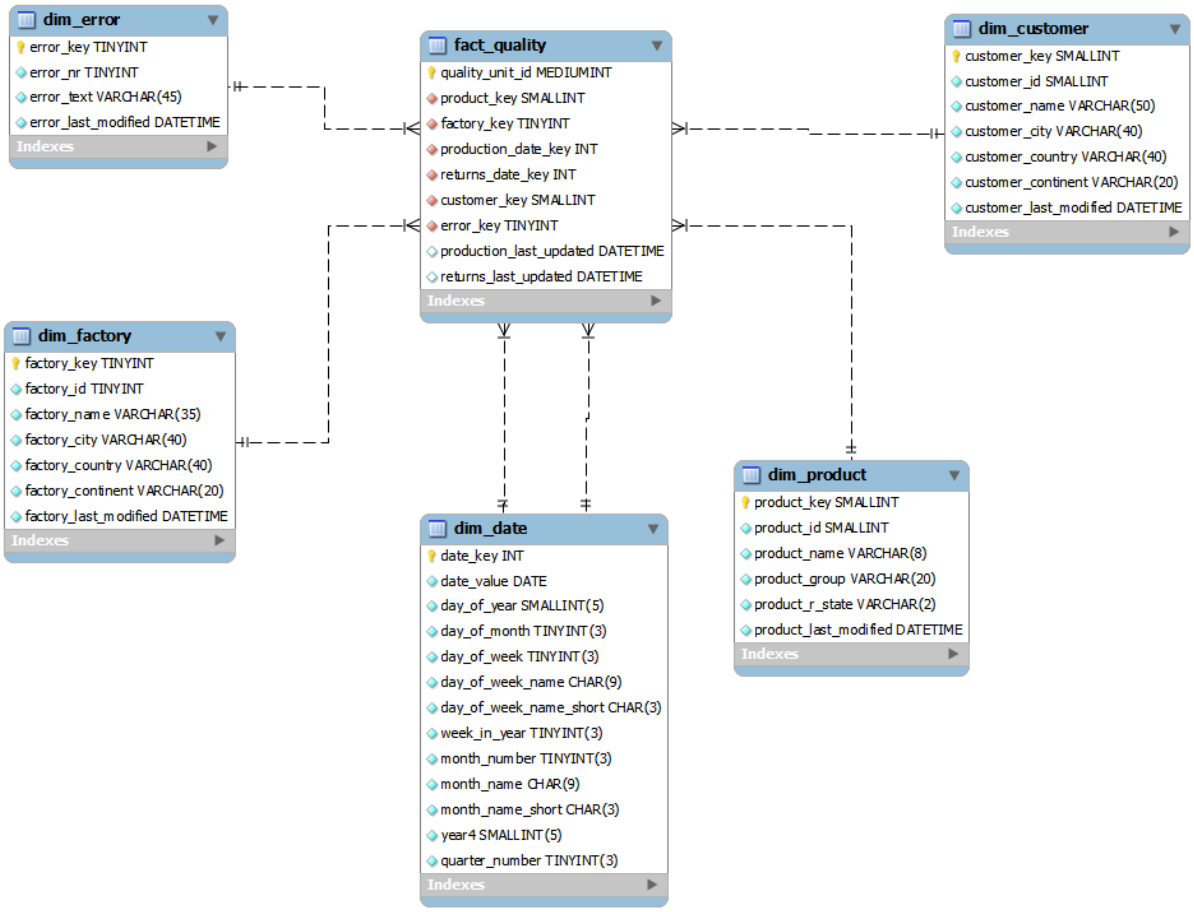

Figure 6-2: The second and final DW design

The fact table now contained key columns to all dimensions tables (two for the date dimension), the two *last_updated* columns from the previous fact tables, and two new columns. The two new columns, *quantity* and *returned* was the new fact columns. Quantity was always set to 1 and returned was set to 0 or 1 depending if the unit had been returned or not. Now the *sum* function could be used on those two columns to get the desired facts *units* and *returns*. This design was mainly done to counter some issues when building the OLAP cube later on but that will be discussed in the OLAP section of this chapter.

With this new design a new problem arose. If a unit had not been returned the keys columns *return_date_key* and *error_key* would be null. Null values in the fact table are not desirable because it causes inconsistency in data. Once again design ideas from Kimball were taken. Kimball states that instead of having null values in dimension key columns they should point to a specific row in the dimension table[17].

This lead to the insertion of a dummy row in both the *dim_error* and the *dim_date* table with fixed primary keys. All units that had not been returned referred to these rows in their *return_date_key* and *error_key* columns.

Earlier designs of the fact table contained the serial numbers as identifiers for the units, this was changed to the *unit_id* field from the operational database instead since it takes up less space and is easier to do lookups on. And the serial numbers aren't really needed when doing the analyses.

To be able to handle the CDC it was decided that every table in the DW should have a column called last_modified. The column type was set to *timestamp*, which contains a date and a time of day. The *last_modified* columns purpose is to be updated each time new rows are inserted och rows are updated in the table. By having this column and a counterpart in the operational system the ETL can take the maximum of the last_modified column in the DW, which would result in the latest time stamp, and then only select rows from the operational system that succeeds that time stamp. With this a functional CDC can be implemented in the ETL.

It is however important to note here that in many cases of building a BI system the designer does not have total control over the source systems, the more common case is that they have no control over them at all. This means that if some of the source systems does not have any indication of when information was last changed constructing a CDC would be a lot more complicated. The CDC for a source system without that information would have to record it's own information of what data has been gathered on previous data extractions. One solution to accomplish this has been discussed earlier and that is to use a staging area.

One last thing to comment on in the final fact table implementation is the necessity of two last_modified columns. The production_last_modified is pretty straight forward, it tells the CDC when the unit was either inserted or when its information was last updated. The problem with this is that if there were only this last_updated field no information about when a unit was returned would exist. Of course if a unit was returned the production_last_modified field could be updated to the time when the return was recorded but this would mean that the ETL wouldn't know if the time stamp signifies an information update or that the unit has been returned. This would create extra work for the ETL since it would need to compare operational data with its own to see which change has occurred. So instead of forcing this fail-safe it was decided that adding the returned_last_updated field was better. This column tells the ETL when information about a return was last made.

ETL jobs and transformations

As mentioned before the PDI works with *jobs*, *transformations* and *steps*. It was decided that the extraction and loading of the data from the operational database to the DW would be done with one transformation per dimension table. The fact table loading consisted of two transformations, but let's start with the dimension table transformations.

Loading the dimension tables

Figure 6-3 shows the PDI job that loaded all data from the operational database to the corresponding dimension tables in the DW.

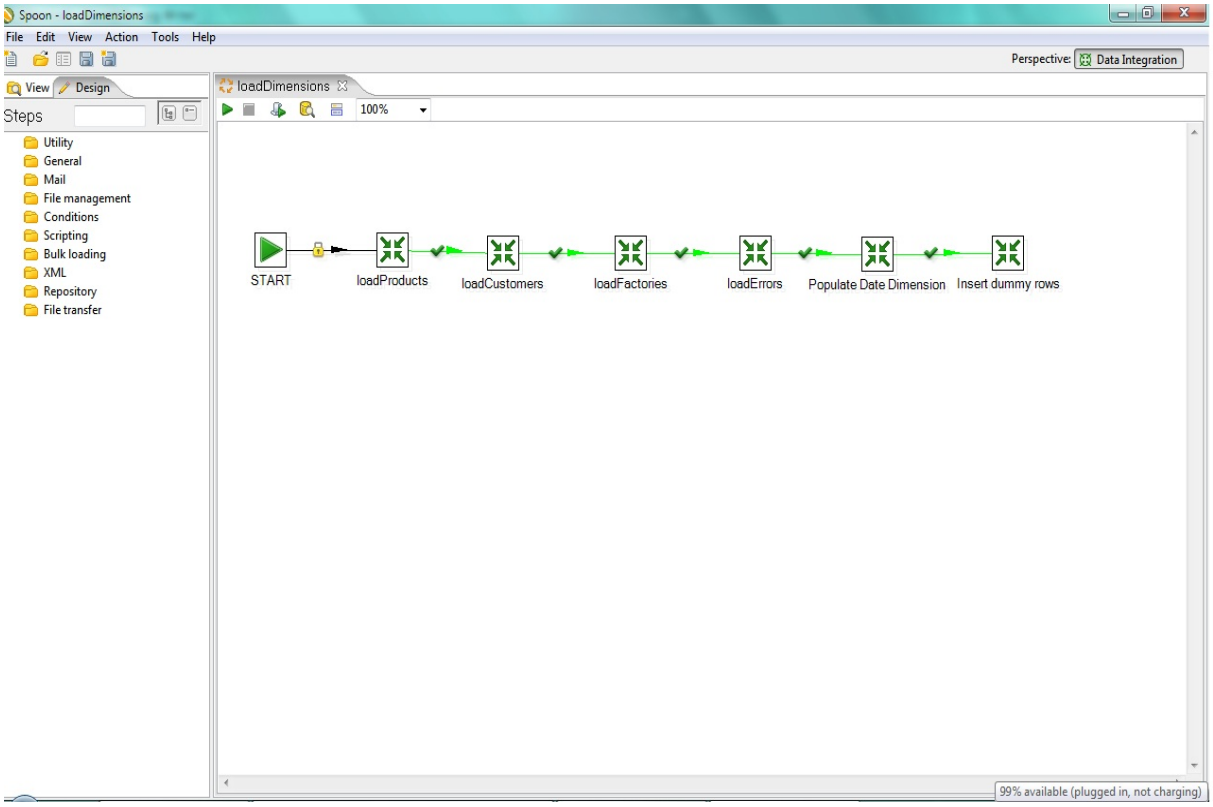

Figure 6-3: The dimensions load job

It consisted of six transformations – five for the five dimension tables and one for inserting the dummy rows into the date and error dimensions. The transformations for loading the dimension tables - *loadProducts*, *loadCustomers*, *loadFactories* and *loadErrors* - all looked very similar and to not get over-explicit a closer look will only be taken at the *loadCustomer* transformation.

The loadCustomers transformations loaded all of the customer data into the customer dimensions table. The transformation design is shown in *Figure 6-4*.

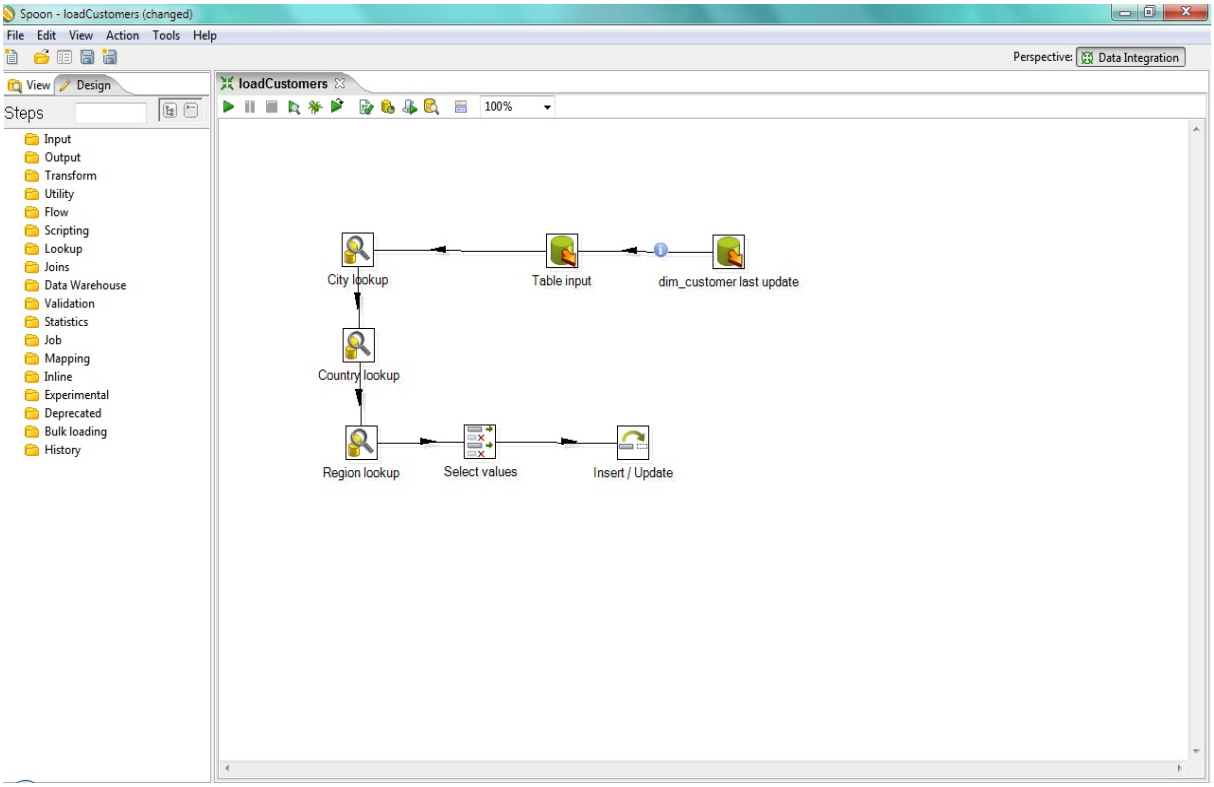

Figure 6-4: The customers load transformation

The first step called “dim_customer last updated” is the CDC solution used and will be discussed later.

The next step is of course to get the data from the operational database. This were done with the *Table input* step. A database connection must be defined as the input source for this step. To query the database SQL queries can be written manually in the step or a database explorer can be used to select which table to import data from. The step will pass along the fetched rows from the table as its output. The table input step in *loadCustomers* connected to the prototype DB and selected all of the fields from the customer table.

The geographic data about the customers (continent, country and city) didn't reside in the customer table but in its own tables. This data also had to be gathered for the customer dimension table since it was totally denormalized and held all available information about a customer. The customer table in the prototype DB contained foreign keys with information about which city the customer were based in. The city table in turn contained foreign keys to which country the city was located in. And the country table contained foreign keys as to which continents the countries belonged to. This means that the information needed to be fetched from the different tables with the use of the foreign keys. For this the *Database Lookup* step was used in the *loadCustomers* transformation. The lookup step requires as input a table name, a key column in the table and a value from the stream in PDI to do the matching against. A user can then choose which columns to fetch from the table if a lookup is successful. In *loadCustomers* it was done, in order, as follows:

City lookup: Fetch city name and country id from city table, match on city id from stream with city id in table.

Country lookup: Fetch country name and continent id from country table, match on country id from stream with country id in table.

Continent lookup: Fetch continent name from continent table, match on continent id from stream with continent id in table.

After the the lookups all of the necessary data were available. The *Select values* step were used to filter out unwanted data in the stream, such as the foreign keys. They were not part of the data wanted in the customer dimension table and thus were discarded after use.

The last step were to inset the data into the customer dimension table. This were done with the *Insert / Update* step. This step needs a database connection, a table and a key column. The key column is used to match with a field from the stream. If a match is found that row in the table is updated. If no match is found a new row is inserted. Which columns that should be updated if a match is found can also be specified. In *loadCustomers* the customer dimension table from the DW were the table and the *customer_id* field were used to match with the *customer_id* field from the stream, which originated from the customer table in the operational DB. All of the columns in the dimension table were set to be updated if a match were found, except the customer_id field since that were used for the matching and should not be changed.

This is how the loading of the customer dimension table was implemented in PDI. As mentioned before all of the dimension table loading transformations were very similar and they all consisted of a table input, some table lookups and an insert / update output. All of them also had the first step which has not yet been explained - the CDC step. In the loadCustomers transformation it was called “dim_customer last updated”. It was a table input step which contained manually written SQL:

```
SELECT  COALESCE( MAX(customer_last_modified),'1970-01-01 00:00:00') AS
max_dim_customer_last_update FROM dim_customer
```

The MAX function returns the greatest value from the *customer_last_modified* field in the customer table from the DW. Since it was a time stamp field the greatest value would be the latest time stamp. The COALESCE function returns the first non null value from its parameters. This means that if there was any data at all in the customer dimension table the resulting value would always be the greatest value of the *customer_last_modified* column. If the customer table was empty then the '1970-01-01 00:00:00' time stamp would be returned.

The value from the SQL query was forwarded to the *table input* step used to fetch data from the operational database. In loadCustomers the SQL for fetching data from the operational database looked like this:

```
SELECT customer_id, customer_name, customer_city_id, last_modified

FROM customer

WHERE last_modified > ?
```

Everything except the WHERE statement were auto-generated with the database explorer. The “?” in the WHERE statement was the input from the previous step – the time stamp from the query in the *dim_customer last updated* step. This meant that every time the *loadCustomers*

transformation was run the latest time stamp in the dimension table was fetched and only the rows from the operational database that had been changed or inserted after that time stamp were fetched. This meant a working CDC had been implemented.

The date dimension transformation differs from the others because it was not fetching data from any database. Instead dates were generated in PDI and inserted into the date dimension table.

The date transformation is shown in *Figure 6-5*. A description of the most important steps will be made to clarify how the transformation is done.

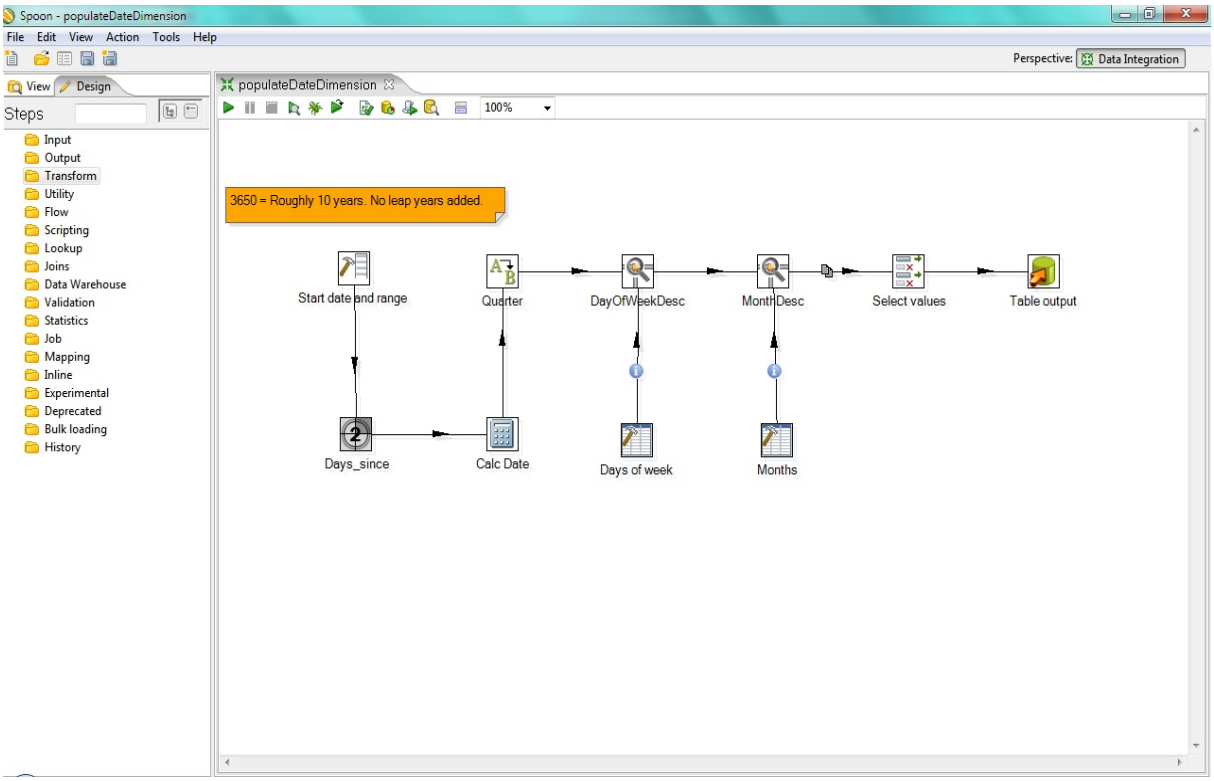

Figure 6-5: The date generation transformation

Start date and range: This step takes a limit parameter and column data. The limit parameter determines how many rows that will be created, this will be the total number of dates generated. Each row will have one column named *start_day* which is the first date in the wanted date range. Since a full range of dates from 2005-01-01 to 2015-01-01 (10 years) was the goal for the date dimension the start date was set to 2005-01-01. The limit was set to 3652 which is 10 years * 365 days + the two extra days in February from the leap year of 2008 and 2012 which is the leap years in that time period.

Days_since: This is just a sequence that starts at 0 and gets incremented by one each time. This means that it will simulate the days that has passed since the start date each time a new start_day row is created and passed along in the stream.

Calc date: The most important step in the transformation. This step is a *Calculator* step. The calculator step consists of functions, parameters and results. A function is chosen, and there a wide variety of numeric, string and date functions available. Some functions require

parameters and they can either be written directly in the step or taken from the steam. All of the functions returns a result, which is passed along in the stream. The step in this particular transformation makes use of some date functions, the most important one being *Date A + B days* which takes as parameters a date A and an integer B. The date A is the start_day column from the steam and the integer B is the value from the *Days_since* step. This results in every date between 2005-01-01 to (2005-01-01 + 3652) days to be created. This function also handles leap years.

The rest of the steps were there to determine which quarter a date belongs to and what month name and weekday name a date had. The last step were a *Table output* step that put the generated dates into the date dimension table.

The last transformation in the dimensions job were the *insert dummy rows* transformation and was really simple. It was just static hard coded information inserted into the date dimension and the errors dimension.

So that was how the transformations to populate the dimension tables in the DW was designed. Now a closer look will be taken at how the fact table was populated.

Loading the fact table

The job for populating the fact table was divided into two transformations. The first one, called *populateProductionFacts*, inserted the data about every produced unit. The second one, called *populateReturnFacts*, updated the columns implying the return of a unit for units that had been returned. *populateProductionFacts* is shown in *Figure 6-6*.

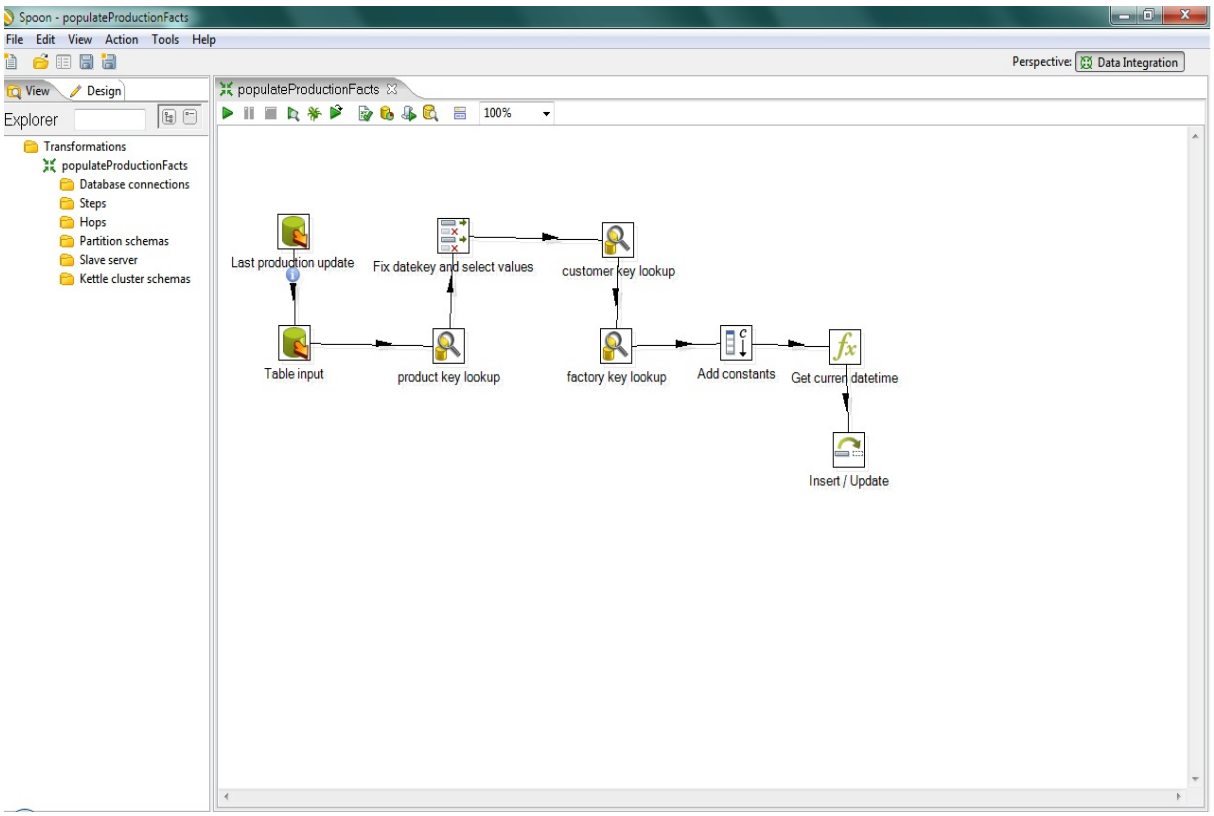

Figure 6-6: The units load transformation

The transformation started as all of the other transformations with the CDC table input step, followed by the operational database table input. After that there were four keys that needed to be looked up, one for each dimension table. Each produced unit were going to be a row in the fact table and every unit row contained keys to corresponding rows in every dimension table. The first design of this transformation had a table lookup for the date dimension as well. But since the date dimension makes use of smart primary keys (yyyyMMdd) the key can be derived from the production date and a table lookup is skipped. This made a vast improvement to the run time of the transformation since the table dimension is the biggest of the four. So the *Fix datekey and select values* step filters out needed values in the stream and converts the production date format (yyyy/MM/dd hh:mm:ss) to a date key format (yyyyMMdd).
The *Add constants* step added the key values in the columns *returns_date key* and *error_key* for the dummy rows in the date dimension and error dimension. If the unit had been returned these values was going to be changed but that is what the *populateReturnFacts* transformation was for. There was no need to set the *returned* column to a constant value since it was set to default to 0 in the database. The *Get current datetime* step executeed the NOW() function in PDI which returns the systems current date and time. This value was then used for the column *production_last_updated*.

The last transformation was *populateReturnFacts* and is shown in *Figure 6-7*.

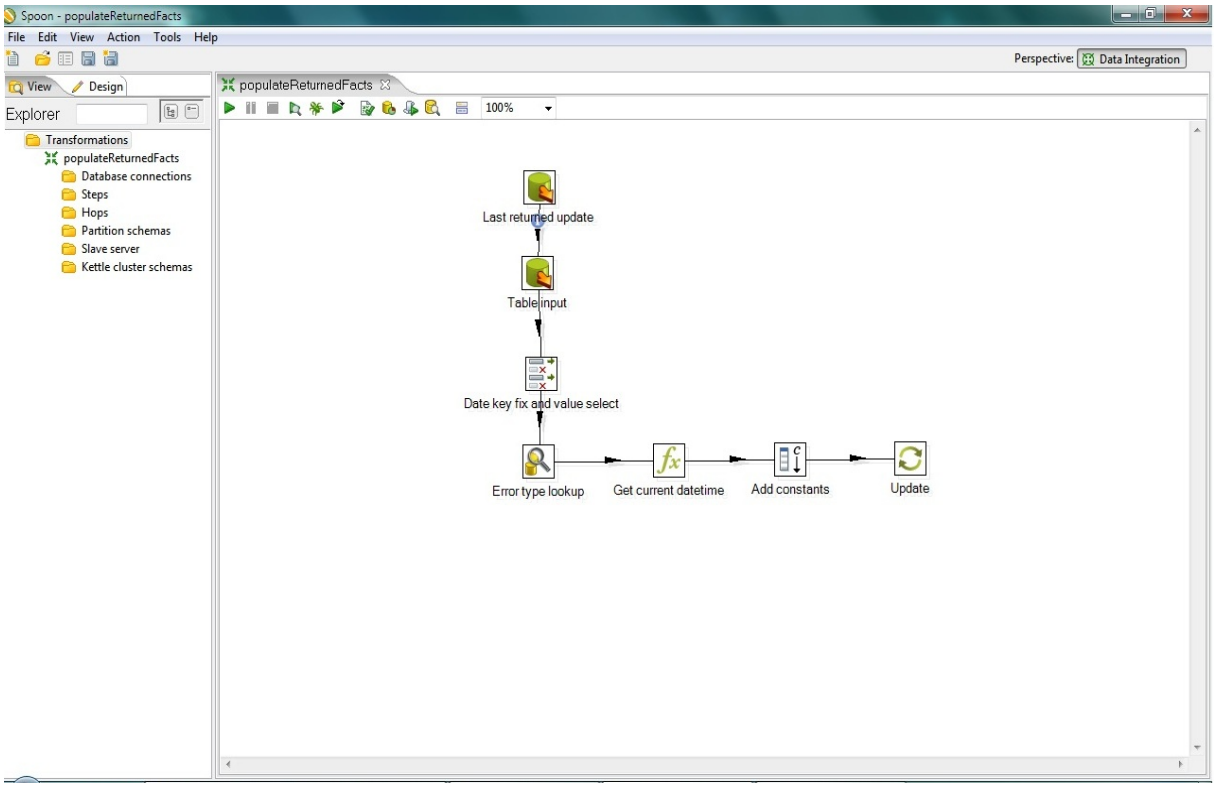

Figure 6-7: The returns load transformation

It started with the CDC and a table input from the returns table in the operational database. Then the date key for the *returns_date_key* column was formatted correctly by altering the *returns_date* field from the *return* table. After that a table lookup was made to get the key for

the error type. The *Get current date time* step was the same as in *populateProductionFacts*, but this time the date was intended for the *returns_last_updated* column. The *Add constants* step set *returned* to 1 and finally an *Update* step were used to alter the rows with units that had been returned. The update step was used instead of an insert / update step since nothing was being inserted, rows were just being updated.

That concludes the design of the ETL in PDI. No code was needed to design the ETL at all. Some manual SQL were written but that was only to do the CDC step and to generate the dummy rows for the error and date dimension tables.

OLAP with Schema Workbench and Pentaho Aggregation Designer

With the DW and ETL all done the next step was to design a cube. The cube was the most important part for the end result since all analyses made was going to be queries run against it.

A cube is described by an XML-file which describes the dimensions, measures and hierarchies in the cube. As mentioned previously *Schema Workbench* (SW) was used to construct the cube. What SW really does is that it gives the user a graphical interface in which the cube can be designed. The schema file is automatically generated according to the users inputs in the GUI. This means that no knowledge about the rules and standards of schema files are necessary. All the user needs to know is that a cube consists of dimensions, measures and hierarchies and how these building blocks are connected. With that knowledge it is really easy to design a cube in the SW. *Figure 6-8* shows how SW looks. In the figure the final cube design for the prototype cube is also loaded in SW for the purpose of having some content. So lets go through how the prototype cube was designed with SW and also look at what design decisions were made during the process.

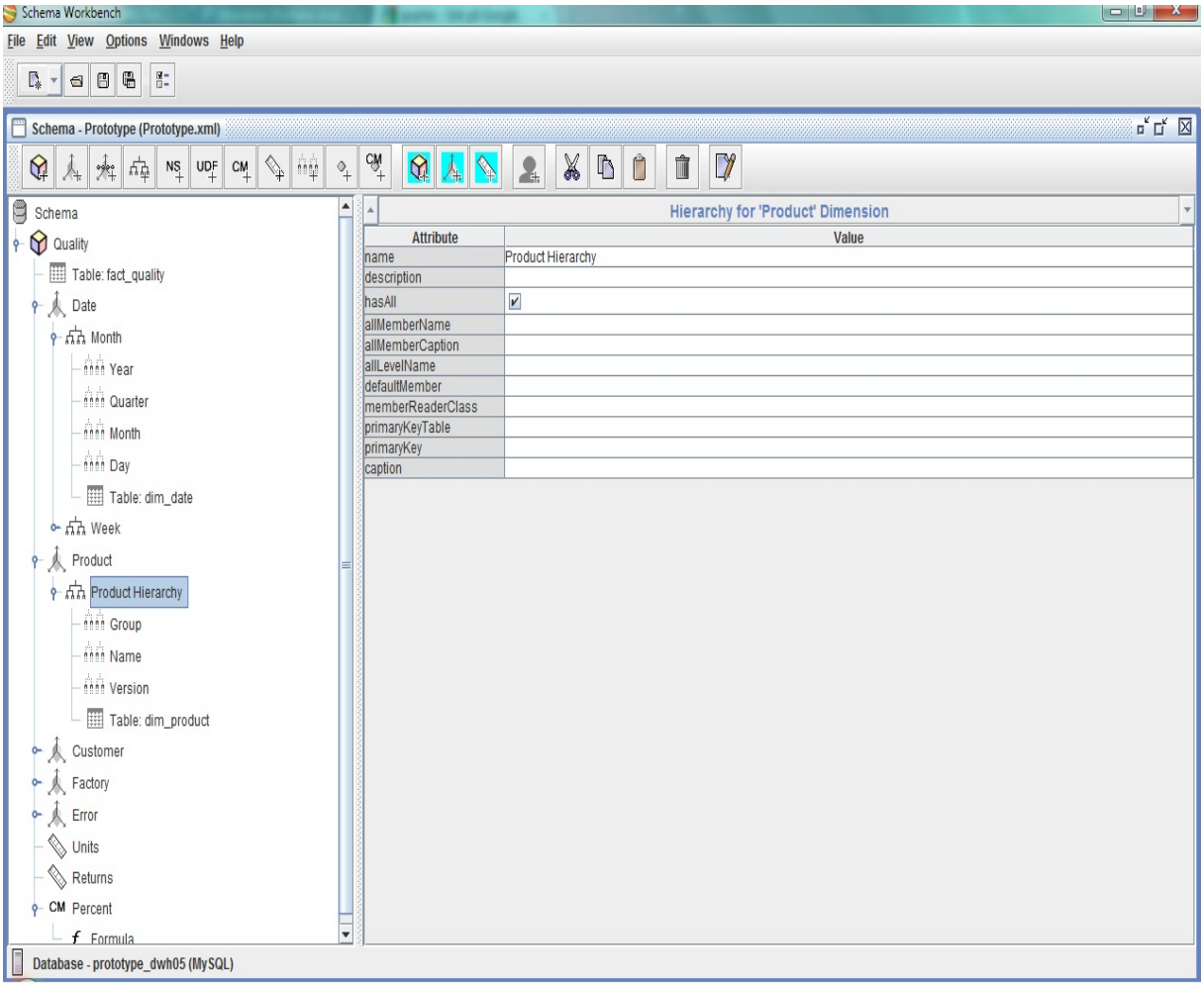

Figure 6-8: The Schema Workbench

The SW consists of a toolbar for commonly used functions and two content panes. The left pane is the *object explorer* which shows the contents of a Mondrian schema. The content is the different *objects* that exists in a schema such as cubes and their corresponding dimensions and measures. The right pane is the *options-pane* and shows the parameters available for an object when it is select in the object explorer. All terms used above should not be considered official ones instilled by Pentaho Corporation - they are defined by the author of the thesis.

The first thing to do in SW is to specify a database connection. SW connects to the database and then the database can be used to build a cube. To build the cube there are a variety of functions available in the SW.

The functions that were used in SW to create the cube was the following:

Add cube – This adds an empty cube the the XML schema. Has parameters for meta data such as name and description but none of the parameters are required to create a cube. But it needs dimensions and measures added to it. For measures to be added a fact table must first be specified. This is done by selecting a table from the database connection. After that measures can be added with the *add measure* function. If that doesn't cover the intended measures a user has in mind *calculated member* can be used instead, more on these later.

Add dimension – Adds a dimension to a cube. The dimension itself has, just as the cube, parameters for meta data but instead of dimensions it requires at least one hierarchy to be specified and for the hierarchy to be able to have levels a database table is also needed. After a table is specified from the database connection level can be added. The levels must point to columns in the added dimension table.

Add measure - Adds a measure to the cube. The measure is defined by using predefined functions (count, distinct count, sum, min, max and avg) on one column in the fact table.

Add calculated member – A measure that is calculated by a user defined formula. The formula is specified in MDX and the data is gathered from the cube.

With the functions at hand the Mondrian schema for the prototype system could then be built. It had been decided earlier that one cube was enough so a cube was added to the schema with the *add cube* function. After that the five dimensions were added – Date, Product, Error, Customer and Factory. Each of the dimensions had one hierarchy added to them except Date, which got two. Again the star schema design of the DW shined through here as each dimension table could be directly connected to the dimensions. Adding levels was also very straight forward since the DW had been design with great care and with cubes in mind. The levels in the dimension hierarchies looked exactly like the internal ordering in the dimension tables. The following list shows how the dimensions and hierarchies turned out:

Date dimension

Hierarchy name: Months

Levels: Year, Quarter, Month, Day of month

Hierarchy: Weeks

Levels: Year, Quarter, Week, Day of week

Product dimension

Hierarchy name: Product Hierarchy

Levels: Product Group, Product Name, Version nr

Error dimension

Hierarchy name: Error Hierarchy

Levels: Error nr

Customer dimension

Hierarchy name: Customer Hierarchy

Levels: Continent, Country, City, Customer Name

Factory dimension

Hierarchy name: Factory Hierarchy

Levels: Continent, Country, City, Factory Name

With the dimensions done it was time to add the measures. Lets look at how they were implemented. Three measures were wanted – the number of units, the number of returned units and the error percentage for the units. It was decided to call the measures *Units*, *Returns* and *Percent*. As discussed in the data warehouse section of this chapter the warehouse was later redesign to solve some issues with the cube design. At first the cube measures were calculated in the following manner:

Units – Really very straightforward, the count function was used on the unit_id field in the fact table. Since unit_id was a column dimension it would always be present in a result even if none of the other dimensions were used in the query. So counting the number of unit_id rows in a result set gives the amount of units.

Returns – This was added as a *calculated member* and the formula counted the number of rows in the result set where the error_key field was not equal to 255. If you recall the row in the dim_error table was the dummy row inserted that non returned units could reference to in the error_key field in the fact table. So counting every rows that had a key not equal to 255 would result in the amount of returns in the result set.

Percent – This was also a calculated member and the formula was simply:

[Measures].[Error] / [Measures].[Units][1]

15[Measures].[Measure name] is how you select a measure from a cube in MDX.

After some testing and consideration this felt like the wrong approach. Doing the calculation of the unit amounts and returned amounts each time a question was asked to the cube was both performance and time consuming. The final approach was to add the two additional fields to the fact table that was in the final DW design - quantity and returned. This gave the opportunity to calculate the measures like this:

Units – Still a normal measure but instead of counting the unit_id field the sum function is used on the quantity field.

Returns – Same solution as for the units but the sum function is used on the returned field.

The calculated member to give the Percent measure was kept since it's just a simple division, but the members of the division became a lot faster to calculate.

This concludes this section and also the chapter. The whole system had now been designed and implemented and was ready to be tested. The big questions now was if it was going to work, how good it would work and what the performance would be like.

Results

Running the system

The system was started and the ETL ran very smoothly without any errors with the MySQL and PostgresSQL DBMSs. Both LucidDB and MonetDB encountered problems in the ETL. The problems was encountered in the *bulk loading* steps available in PDI for LucidDB and MonetDB. A standard *table output* step was tried instead but that did not work either. The *bulk loading* steps was recently moved by Pentaho Corporation from the experimental status to stable status and little documentation was available. After several different approaches in solving this a solution was not found and both DBMSs were discarded due to time constraints.

After the DW had been populated with the data from the source system a cube was created and the queries made against it got correct result without any incidents.

The BI-server was also tested. The default HSQLDB storage for the BI-servers data was changed to the MySQL server running the prototype databases. Finding how to do this and were to make changes took some time and the following forum guide found in the official Pentaho forums were used:

http://forums.pentaho.com/showthread.php?66901-Pentaho-BI-Server-3.5-MySQL-PostgreSQL-and-Oracle-for-Windows-amp-Linux-Tutorial

After following the guide the BI-servers databases were successfully run from the MySQL server. Accessing the solution repository from both PDI and SW worked as intended and transformations, jobs and schemas were successfully uploaded and downloaded directly in the tools. The pivot tables for OLAP cubes were tested and worked as intended. Reports were created and submitted to the solutions repository without any difficulties. In other word the BI-server worked as described in the information from Pentaho Corporation.

Performance and performance tuning

The ETL

At first the ETL was just run with the default properties on every step in the transformations. As mentioned in 7.1 this worked fine and no problems were detected so the question now was if the time run times could be lowered. The focus will be on improving the fact table loading, which includes the *populateProductionFacts* and *load populateReturnFacts* transformations. The loading of the dimension tables was so fast that it really didn't need any tweaking (longest run time was below 10 seconds) and the tweaking done with the fact table loadings can be applied to those too if one chooses to.

A number of tuning options were identified in PDI. First of all the commit size in the *table output* step could be altered. The default value here was 100. This means that every time the stream has delivered 100 rows to the output step a those rows are inserted into the database. The fact table transformation for units were run with different commit sizes to see what commit size yielded the best result. The amount of data was first set to 500.000 and then 5.000.000 rows. The results are shown in the *Table 7-1* below.

Commit size (#)	PostgreSQL 50K rows (s)	MySQL 50K rows (s)	PostgreSQL 5M rows (s)	MySQL 5M rows (s)
1 - 10	921	672	9486	6897
100	484	365	5013	3589
1000	487	390	5117	3613
10000	500	402	5218	3746
100000	500	404	5388	3823

Table 7-1: Commit size run times

As the results show the best value for the commit size was the default value of 100.

The next performance adjustment to try was preloading of the tables used for key lookups in the transformation. The lookup steps had a checkbox that if marked would preload the whole table used for the lookup when the transformation started. This proved to be an improvement to the transformation run time and it was decided to use it in the final implementation. It is important to note though that if preloading a a table is used there has to bee enough system memory available to load the table into. So very large dimension tables will use a lot of memory when using this approach and it needs to be taken into consideration when designing transformations. The dimension tables used for the prototype system was fairly small and they were all able to be preloaded without taking up too much memory. The exception though was that the dim_date dimension table was not preloaded since it was too big to preload. This leads to the third and last performance adjustment for the fact table loading. Since the date dimension used a *natural key* (yyyyMMdd) as primary key the key is always known. This means that the key can be generated directly in the ETL transformation. As explained in *Section 6.3.2*. the date keys were generated from the production date and the return date fields gathered from the operational database. Generating date keys instead of doing lookups in the date dimension proved to be the best improvement to the fact table loading. Below is a short list of test times with and without the adjustments when loading the fact table.

Run times for *populateProductionFacts* with 5 million units

Preloading and generation of date keys: 59 minutes

No preloading, generation of date keys: 109 minutes

No preloading, no generation of date keys: 201 minutes

Run times for *populateReturnFacts* with 2 million returned units

Preloading and generation of date keys: 25 minutes

No preloading, generation of date keys: 30 minutes

No preloading, no generation of date keys: 59 minutes

It can be derived from the results that the date key generation was a significant improvement to the run time. The preloading of lookup tables was also an improvement but didn't have the same impact on performance as the key generation had.

With the adjustments made to the fact table transformations they were ready to be tested. The transformations were run on all of the source databases. This meant load sizes of 5, 10, 20, 30 ,40 and 50 million rows for the units transformation and 0.5, 1, 2, 3, 4 and 5 million rows for the returns transformation. Both DBMSs were also tested. The results are shown below both as a table in *Table 7-2* and as a graph in *Figure 7-1*.

Load units to fact table (Seconds) PostgreSQL	**Load units to fact table (Seconds) MySQL**	**Load returns to fact table (Seconds) PostgreSQL**	**Load returns to fact table (Seconds) MySQL**
5013	3589	578	432
10418	7296	1205	1028
21693	14847	2578	2130
31374	23294	3708	3278
40811	31641	4987	4848
55447	40583	6431	6209

Table 7-2: Run times for loading the fact table with the ETL tool

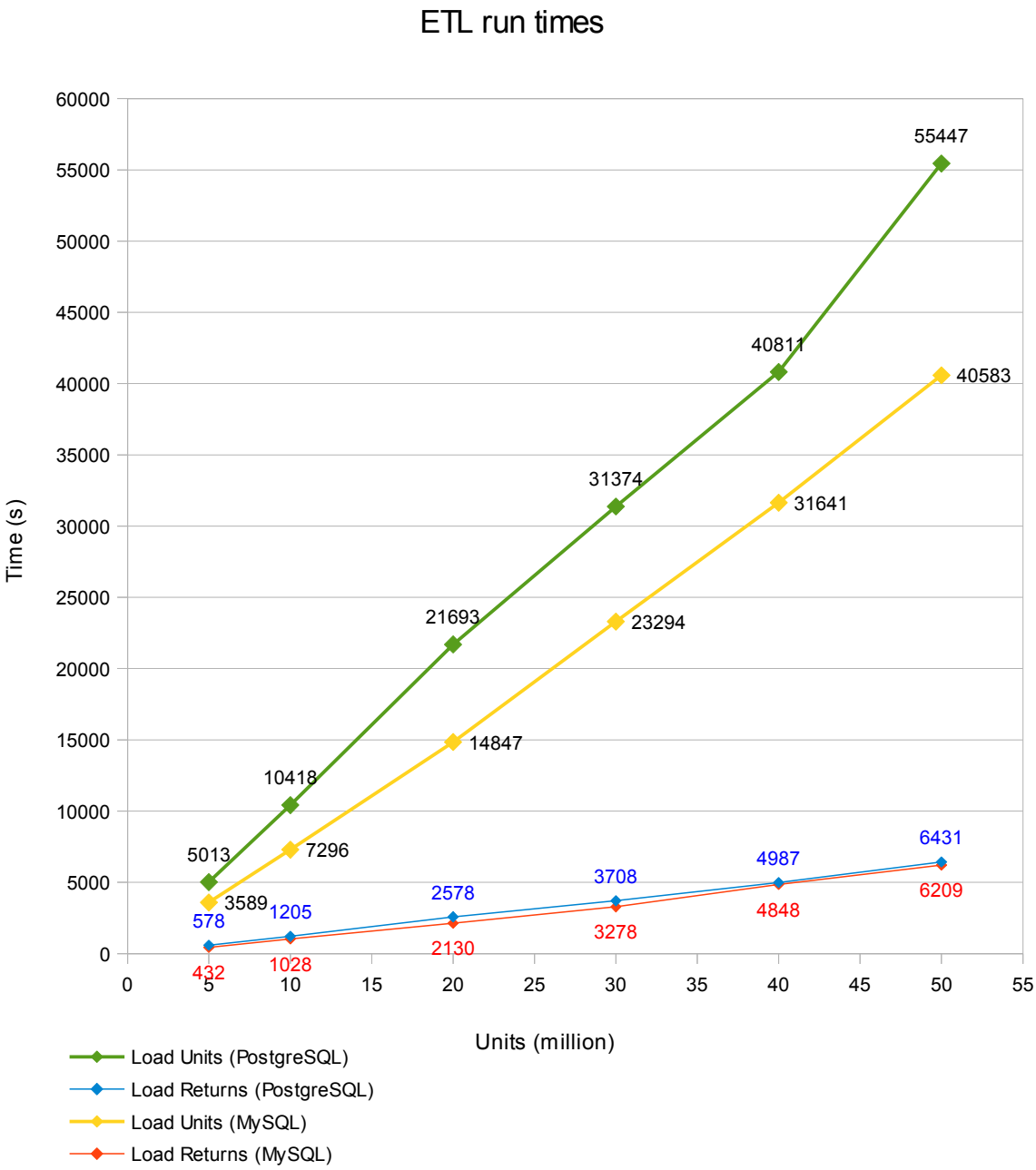

Figure 7-1

As was expected the longest run times occurred when the highest quantity of rows, 50 million, were loaded. Loading of 50 million units from the operational database to the data warehouse using the MySQL databases took 11 hours and 16 minutes. With PostgreSQL it took 15 hours and 24 minutes. Loading 5 million returned units took 1 hour and 43 minutes with MySQL and 1 hours and 47 minutes with PostgreSQL.

Mondrian

To test Mondrian properly a simple java program was created to start the server an run a client towards it. A cube was created from the schema designed in SW and then the twelve test queries found in Appendix A was run against it. This was done for all of the DW sizes and for both of the DBMSs used. Every test was timed and the results can be seen in Appendix B.

One way to improve query results is to use aggregated tables. An aggregate table holds measures that are calculated before hand. What the values for the measures are is decided by the aggregation levels. The aggregation levels are levels from the dimension hierarchies and an aggregation table can have one level from each existing dimension. For example if a common question for the prototype system is "How many returns is there in a specific month?". An aggregate table for this type of question would need aggregation on the level month in the date dimension. To accomplish this a table would need to be created in the DW to hold the data and then aggregation must be defined in the schema file for the cube in order to let Mondrian know it exists. The table created in the DW would look like *Table 7-1* below.

dim_date_year4	dim_date_quarter_number	dim_date_month_number	fact_quality_units	fact_quality_returns	fact_quality_fact_count
2005	1	1	3456	274	3456
2005	1	2	2876	145	2876
...	...	...	...	...	...
2008	3	7	3298	254	3298
...	...	...	...	...	...

Table 7-1: An aggregation table example

The first three columns are the *year4*, *quarter_number*, *month_number* columns from the date dimension and is the aggregation level we wanted. The next two columns are calculated values for the measures *units* and *returns*. Since the measure *percent* is a calculated member it is not present in the aggregation table. The *fact_quality_fact_count* column is so the cube knows how many rows has been aggregated. In our case this value will be the same as the unit measure since that measure is just a sum of the quantity column from the fact table.

So lets say this aggregation had been implemented and someone wanted an answer to the question "How many units were returned in July 2008?". Mondrian would then see that an aggregation table existed with the aggregation on the level month for the date dimension and get the value from the aggregate table. This means that the result takes as long as it takes the database to get the correct row (year=2008, month=7) from the aggregated table.

Aggregate tables that was specifically matched to the twelve test queries were made with *Pentaho Aggregation Designer*. The aggregations made are shown in a table in Appendix C.

The run times of the test queries with the cube now using the new aggregate tables is shown in Appendix B. Almost all of the queries got instantaneous results. This is really what to be expected since the work load has shifted from calculating measures by joining dimensions tables with each other to just fetching the right row in an aggregation table. Still it proves the power of using cubes with aggregation tables and if you know what the most common queries are aggregation tables can be used to improve the time it takes to run analyses.

Discussion and conclusions

This chapter will address the issues raised in the beginning of the thesis and at the same time discuss what the investigation resulted in and what conclusion can be drawn from the work.

Was it possible at all to create a BI-system without writing any code?

The simple answer to this question is yes, it is possible to set up a BI system without even having to bother about how things work in the background. As this thesis has proven with the use of the Pentaho BI suite a whole BI system was set up and used without any major problems. Tools were available for every part in the system. The tools did a great job of giving the user an intuitively graphical interface to work in. And working in the tools led to well functioning parts which as a whole made up the BI system. None of the tools needed any knowledge about programming or program code to be used properly.

What knowledge and skills were required to set up the system?

This is were it gets more interesting. First of all setting up a BI system requires knowledge about what BI is and how it works. Many major design choices needs to be made and the better the understanding of BI is the better the choices will get and the system will be more tailored to a specific organizations needs. Knowledge and insight in BI is the first step to create a well working BI system.

That knowledge involves being able to identify what the organization wants to do with a BI system. Which reports are the organization interested in? What are the analyses required to create those reports and is there data available in the organization to support those analyses?

Identifying what data the organization has and what it can be used to answer is crucial when creating a BI system. Making analyses on insufficient data or even the wrong data is a major error. And if decision makers take the results as true and decisions are made based on them that could lead to a disaster. This means that creating a BI system also requires insight in what data is available to be used in the system and what type of analyses can be done with the data.

The DW is a database and has to be created and maintained in a DBMS. This requires knowledge about databases, SQL and at least one reliable DMBS. The ETL also has some section were knowledge of SQL is vital. Apart from knowing the ins and outs of designing a BI system as a whole designing the DW is the second most important part, why will be discussed in the performance question.

When it comes to OLAP cubes SW and PAD does most of the work for the user but knowledge about how cubes are modeled is required to be able to create them in SW.

Here is a list of what is required to setup and manage the different parts in the system:

System as a whole:

- Knowledge and insight in BI theory. The better knowledge the better the design of the system will be.
- Database knowledge

Data warehouse:

- Database knowledge
- Being able to handle at least one DBMS to create the DW in.

ETL:

- Small SQL knowledge

Schema Workbench and Pentaho Aggregation Designer:

- Knowledge about cube structures and aggregation tables

Reporting tools:

- To be able to define data sets to be used in reports SQL is required if data is gathered directly from the DW.
- If data is gathered from an OLAP cube MDX knowledge is required.

What knowledge and skills were required for an end user?

Training in how to use the reporting tools and the ability to process manuals. To make it even easier meta data can be specified to data sets and the end user can be presented with the meta data when selecting data to reports. For example a manager of the system can specify a data set called “Shipped units per year” which collects number of shipped units per year from a cube with MDX. The end user then only has to select “Shipped units per year” an add it to his report to get the numbers wanted.

What were the major challenges in designing and implementing the system?

Almost all of the major challenges lies before the system is even implemented. A BI-system is not a magic box solving any unanswered business question an organization has. There must first be a clear and sound definition of what the purpose of the system is. The output of a BI-system is what comes out at the end of the process: business analyses and business reports. What do the organization want to analyze and what reports are they interested in having? A lot of thought should be put into this. Second the data requirements for making the analyses and reports must be identified.

And when the required data is known there must be a proper analysis of data available. The organization must be sure that the data will be available to the system. This includes

supplying the correct data as well as the required quantity of data. Addressing these issues is the first major challenge.

The next challenge lies in designing the system. A BI-system consists of several parts and it is important that each part is designed properly. Starting with the ETL there are three topics to cover: Data cleansing, CDC and staging areas.

Lets begin with data cleansing, which is probably one of the parts in a BI system that organizations puts the most work hours into[14]. Data cleansing is extremely hard to do and the ideal scenario would be that the source systems contained correct and properly formatted data to begin with. This is as discussed in 3.2 usually not the case. Data cleansing can be made in PDI with various available tools. The big question here is whether to put a big effort in getting data as clean as possible or be satisfied with more or less incorrect data slipping into the DW. Having extensive data cleansing in the ETL will of course slow it down. A recommendation here is to run the system without accepting any incorrect data at all. Instead information - meta-data - about the erroneous data is logged and stored. The logged meta-data can then be analyzed to give a good view of which source systems contains incorrect data and where it is located. Then the organization have the choice to address the issue directly in the source systems. Of course simpler data cleansing mechanics should be present to correct minor problems such as simpler formatting errors or misspellings. In the end it is all a question of how much time - and in the end money, it is worth to spend on having correct conformed data in the system.

Having a staging area or not is really a question of how much pressure the organization wants to put on the source systems when extracting data. The more complex the ETL process is the heavier the burden will be on the source systems. The amount of data to be extracted will determine the length of the extraction, thus determine the time frame in which the ETL will affect performance. The number of available CDC solutions is closely connected to the staging area. Since a CDC solution can be put in the source systems, in the staging area, in the DW or split amongst any combination of the three not having a staging area will limit the options. Using time stamps,sequences or triggers to determine when data was last changed are the best choices, but where to put the information is a bigger challenge. Having the information in the source system is a good choice since the time stamp or sequences used can just be compared to the ones in the staging area or DW. But putting this kind of information in the source systems is not always possible, or worth the work. Many transactional systems have auto generated-data which means that the system or software that generates the data must be changed in order for it to account for chronology. In some cases this might prove to be very simple but in other cases it might mean a lot of work and in large organizations with many different source systems, all with their own auto-generating software, the amount of work increases greatly. The alternative is to use a log-based solution, to let the DBMS's in the source systems log every data change and then the logs are used to look up what data has changed. Choosing CDC-solution requires some serious thought and can even be implemented differently for each source system if it is necessary. A staging area can be of great help but is not necessary to solve the issue of CDC.

The biggest advantage is the difference in workload the ETL can put on the source system and the staging area. Since all of the data is extracted to the staging area the issue of putting extra pressure on the source systems will only have to do with the extraction process. More performance-demanding operations, such as analyzing the data or transforming it, can be done when it has been transferred to the staging area. The staging area will not have the same restrictions on the workload since the ETL is the only process that will access it.

How was the performance of the system and how could it be improved?

The performance of the system can be viewed in C*hapter 7*. Considering the machine the system was created and run on, these values could be vastly improved on a machine dedicated to running a BI-system (e.g. a dedicated server machine with OS and DBMS properly configured to boost the performance of that particular system). The longest run time for the ETL was to populate the fact table and if we add the two run times for the unit transformations and return transformation we land at a total of 12 h 59 min 49 s for 50 million rows. Take into consideration that 50 million rows is a very big load and this was to initialize the system. Future ETL loadings would only need to transfer data about newly added rows or row updates. With a dedicated server machine with tailored configurations the ETL jobs could easily be run over night when most operational systems has their lowest work load.

It is very important to discuss how the design of the DW and the configuration of the DBMS impacts system performance.

The performance of the system relies heavily on the DW performance. The ETL loads data into the DW, cubes are created from the DW and reports uses data either directly from the DW of from the cubes. How the DW is modeled will mostly affect the OLAP cubes. If the DW is designed with a cube in mind as was did in this thesis it will be very easy to design cubes since the structure is already laid out. Although the DW model needs much consideration and needs to be tailored according to the goals for the particular BI system this is not all. The DBMS itself must be tailored to support the system as best as possible if good performance is a goal. No in-depth explanation of how to set up a DBMS for a BI system will be given but some major things will be pointed out:

- Since most analyses uses joins on large database tables a large temporary table size is a huge improvement to cube performances.
- The DW will have many inserts each time the ETL is run. Having good indexes will help in cube queries but slow down inserts. Priority considerations must be taken here.

The cube performance without the use of aggregate tables was on the other hand a slight disappointment. One can argue that the test queries run against the cube maybe isn't the most common queries that will be used in reports but that is no excuse for the most complex query with 50 million rows in the DW taking 1 h 25 min for the MySQL DW. PostgreSQL fared much better with the same query taking roughly 19 min. Some effort were put into trying to configure both DBMSs to support the cube analyses better and as seen in the results this was most successful with PostgreSQL. If PostgresSQL is better suited for DWs and OLAP cubes is a matter for a more thorough analysis. Too little data is available to draw any major

conclusions about this in the thesis. It was very unfortunate that the DW and cube (or the ETL for that matter) could not be tried with LucidDB and MonetDB. Since the goal both these DBMSs is to be used for data warehousing and business intelligence the performance would have probably been much better with them than with MySQL and PostgreSQL. Although the analysis run times were disappointing this was as said without the aggregated tables. With the aggregated tables the run times were below one second on almost all of the test queries. The most complex one took 1-2 minutes depending on DBMS. So, with a system with proper aggregated tables the analysis speed is astonishingly fast. Again it is important to note that the configuration and optimization of the DBMS has a huge impact on the performance of t he system.

Final thoughts

The Pentaho BI suite is a very strong alternative to creating the system with your own code. With a dedicated server machine to run the system on and a well optimized, properly configured DBMS, the result is a very powerful system that can be a great assistance to decision makers in an organization.

References

[1] Borking, K., Danielsson, M., Ekenberg, L., Larsson, A., Idefeldt, J. *Bortom Business Intelligence*. Second edition. Elanders Sverige AB, Sweden, 2010.

[2] Power, D.J. *A Brief History of Decision Support Systems*. DSSResources.COM, World Wide Web, http://DSSResources.COM/history/dsshistory.html, version 2.8, May 31, 2003.

[3] Rivest S., et al. *SOLAP technology: Merging business intelligence with geospatial technology for interactive spatio-temporal exploration and analysis of data*. ISPRS Journal of Photogrammetry & Remote Sensing, issue 60, p. 17–33, 2005.

[4] Bouman, R., van Dongen, J. *Pentaho Solutions - Business Intelligence and Data Warehousing with Pentaho and MySQL*. Wiley Publishing, Inc., Indianapolis, USA, 2009.

[5] Azevedo, A., Santos, M.F., *Business intelligence: State of the art, trends, and open issues.* 1st International Conference on Knowledge Management and Information Sharing, p. 296-300, Funcahl, Madeira, 2009.

[6] Velicanu, M., Matei, G. *A Few Implementation Solutions for Business Intelligence*. Informatica Economică, issue 3, p. 138-146, 2008.

[7] Dayal, U., Castellanos, M., Simitsis, A., & Wilkinson, K. *Data integration flows for Business Intelligence*. 12th International Conference on Extending Database Technology: Advances in Database Technology (p. 1-11). St. Petersburg, Russia, 2009.

[8] Inmon, W. H., *Building the Data Warehouse*, First Edition. QED, Wellesley, MA, 1990.

[9] Kimball, R., et al. *The Data Warehouse Lifecycle Toolkit.* Wiley Publishing, Inc, New York, USA, 1998.

[10] Rivest, S., et al. *SOLAP technology: Merging business intelligence with geospatial technology for interactive spatio-temporal exploration and analysis of data.* ISPRS Journal of Photogrammetry & Remote Sensing, issue 60, p. 17-33, 2005.

[11] Levene, M., Loizou, G. *Why is the Snowflake Schema a Good Data Warehouse Design?*. Information Systems, Volume 28, Issue 3, 2003.

[12] Utley, C. *Designing the Star Schema Database*. CIOBriefings.COM, World Wide Web, http://www.ciobriefings.com/Publications/WhitePapers/DesigningtheStarSchemaDatabase/tabid/101/Default.aspx , version 1.1, July 17, 2008.

[13] IBM Corporation, *IBM Informix Database Design and Implementation Guide*, IBM.COM, World Wide Web, http://publib.boulder.ibm.com/infocenter/idshelp/v10/index.jsp?topic=/com.ibm.ddi.doc/ddi228.htm, November 2, 2005.

[14] Casters, M., Bouman, R., van Dongen, J. *Pentaho Kettle Solutions – Building Open Source ETL Solutions with Pentaho Data Integration.* Wiley Publishing, Inc., Indianapolis, USA, 2010.

[15] Codd, E.F., Codd, S.B., Salley, C.T. *Providing OLAP to User-Analysts – An IT Mandate*. E. F. Codd & Associates, 1993.

[16] Microsoft Corporation, *SQL Server Analysis Services - Multidimensional Data*, MICROSOFT.COM, World Wide Web, http://msdn.microsoft.com/en-us/library/ms174915.aspx, January 1, 2011

[17] Thornthwaite, W. *Kimball Design Tip #43: Dealing With Nulls In The Dimensional Model*, Kimball Group, World Wide Web, http://www.kimballgroup.com/html/designtipsPDF/DesignTips2003/KimballDT43DealingWith.pdf?TrkID=DT128_DT43, February 6, 2003.

Figure references:

[P1] Dayal, U., Castellanos, M., Simitsis, A., & Wilkinson, K. *Data integration flows for Business Intelligence*. 12th International Conference on Extending Database Technology: Advances in Database Technology (p. 1-11). St. Petersburg, Russia, 2009.

Appendix A

This appendix lists the twelve test queries that were run to test the performance of the created OLAP cube.

Test case 1:

Question: Quality measure per product group per factory/total and all years per quarter

Filters: Year, product group, factory name

Measure: Percent

Aggregations: Production Date.Year.Quarter, Product.Group, Factory.Name

Test case 2:

Question: Quality measure per product group per factory/specific time frame

Filters: Specific year, month, product group, factory name

Measure: Percent

Aggregations: Production Date.Year.Month, Product.Group, Factory.Name

Test case 3:

Question: Shipped units/country, total and all years per quarter

Filters: Year and quarter, customer country

Measure: Units

Aggregations: Customer.Country, Production Date.Year.Quarter

Test case 4:

Question: Shipped units/city, total and all years per quarter

Filters: Year and quarter, customer city

Measure: Units

Aggregations: Customer.City, Production Date.Year.Quarter

Test case 5:

Question: Error percent per product name and it's versions/total and years and months

Filters: Year and month, product name and version

Measure: Percent

Aggregations: Production Date.Year.Month, Product.R-state

Test case 6:

Question: Error percent per product name and it's versions/specific time frame

Filters: Specific year, quarter and month, product name and version

Measure: Percent

Aggregations: Production Date.Year.Quarter.Month, Product.R-state -- SAME AS 5.

Test case 7:

Question: Error type percentages on product group/total, all years per quarter

Filters: Year and quarter, error, specific product group

Measure: Percent

Aggregation: Error.ErrorType, Production Date.Year.Quarter, Product.Group

Test case 8:

Question: Error type percentages on product group/specific time frame

Filters: Specific year, quarter and month, error, specific product group

Measure: Percent

Aggregation: Error.ErrorType, Production Date.Year.Quarter.Month, Product.Group (Same as Test case 7)

Test case 9:

Question: Error type percentages for specific product name/total and year and quarters

Filters: Year and quarter, error, specific product name

Measure: Percent

Aggregation: Error.ErrorType, Production Date.Year.Quarter, Product.Name

Test case 10:

Question: Error type percentages for all product names and r-states per factory/total and years per quarter

Filters: Year and quarter, error, product name and version, factory name

Measure: Percent

Aggregation: Error.ErrorType, Production Date.Year.Quarter, Product.R-state, Factory.Name

Test case 11:

Question: Error percent for all product names per customer/Years and quarters and total

Filters: Year and quarter, product name, customer name

Measure: Percent

Aggregation: Production Date.Year.Quarter, Product.Name, Customer.Name

Test case 12:

Question: Error percent for all customers/Years and quarters and total

Filters: Year and quarter, customer name

Measure: Percent

Agrregation: Production Date.Year.Quarter, Customer.Name

Appendix B

This appendix shows the run times for the test cases in Appendix A run against the OLAP cube. **All cell values are in seconds**.

MySQL – No aggregation tables used:

	Number of units (million)					
	5	**10**	**20**	**30**	**40**	**50**
Run time for T1	194	465	985	1572	2093	3269
Run time for T2	113	287	647	1009	1462	1717
Run time for T3	56	105	216	322	452	1454
Run time for T4	57	110	229	341	475	1499
Run time for T5	65	119	252	357	494	1622
Run time for T6	30	57	119	172	237	312
Run time for T7	122	236	572	788	1159	811
Run time for T8	77	145	388	432	600	525
Run time for T9	97	194	489	556	809	675
Run time for T10	486	950	1831	2719	4420	5096
Run time for T11	222	423	422	622	773	2561
Run time for T12	53	105	204	332	435	1043

PostgreSQL – No aggregation tables used:

	Number of units (million)					
	5	**10**	**20**	**30**	**40**	**50**
Run time for T1	36	106	207	755	1018	1455
Run time for T2	11	43	57	317	433	498
Run time for T3	13	35	54	145	206	237
Run time for T4	14	38	59	151	215	246
Run time for T5	21	45	74	171	232	268
Run time for T6	2	4	8	21	35	41
Run time for T7	6	18	30	153	304	232
Run time for T8	0	0	0	1	1	2
Run time for T9	0	0	0	0	0	0
Run time for T10	153	292	549	752	971	1129
Run time for T11	151	197	288	347	408	472
Run time for T12	10	22	47	76	97	115

MySQL – Aggregation tables used:

	Number of units (million)					
	5	**10**	**20**	**30**	**40**	**50**
Run time for T1	0	0	0	0	0	0
Run time for T2	0	0	0	0	0	0
Run time for T3	0	0	0	0	0	0
Run time for T4	0	0	0	0	0	0
Run time for T5	5	7	7	8	9	9
Run time for T6	0	0	0	0	0	0
Run time for T7	0	0	0	0	0	0
Run time for T8	0	0	0	0	0	0
Run time for T9	0	0	0	0	0	0
Run time for T10	73	101	120	128	128	128
Run time for T11	122	120	119	121	120	121
Run time for T12	0	0	0	0	0	0

PostgreSQL – Aggregation tables used:

	Number of units (million)					
	5	**10**	**20**	**30**	**40**	**50**
Run time for T1	0	0	0	0	0	0
Run time for T2	0	0	0	0	0	0
Run time for T3	0	0	0	0	0	0
Run time for T4	0	0	0	0	0	0
Run time for T5	1	2	2	4	5	5
Run time for T6	0	0	0	0	0	0
Run time for T7	0	0	0	0	0	0
Run time for T8	0	0	0	0	0	0
Run time for T9	0	0	0	0	0	0
Run time for T10	23	50	55	55	56	56
Run time for T11	52	52	53	52	51	52
Run time for T12	0	0	0	0	0	0

Appendix C

This table shows the aggregation tables made with Aggregation Desginer. Every rows is on aggregation table and the columns indicates at what level the dimensions were aggregated in that table. The syntax is Hierarchy(Level). If there is only one hierarchy present in the dimension - which is the case of all but the date dimension - the hierarchy name will be omitted. The (All) level means that no aggregation for that dimension has been done.

	Production date	**Product**	**Customer**	**Error**	**Factory**
1	Month(Quarter)	(Group)	(Group)	(All)	(All)
2	Month(Month)	(Group)	(Group)	(All)	(All)
3	Month(Quarter)	(All)	(All)	(Country)	(All)
4	Month(Quarter)	(All)	(All)	(City)	(All)
5	Month(Month)	(Version)	(Version)	(All)	(All)
6	Month(Quarter)	(Group)	(Group)	(All)	(Type)
7	Month(Month)	(Group)	(All)	(Type)	(All)
8	Month(Quarter)	(Name)	(All)	(Type)	(All)
9	Month(Quarter)	(Version)	(All)	(All)	(Name)
10	Month(Month)	(All)	(Name)	(All)	(All)
11	Month(Quarter)	(All)	(Name)	(All)	(All)

TRITA-CSC-E 2011:106
ISRN-KTH/CSC/E--11/106-SE
ISSN-1653-5715

www.kth.se