Business Intelligence: Multidimensional Data Analysis

Per Westerlund

August 20, 2008
Master Thesis in Computing Science
30 ECTS Credits

Abstract

The relational database model is probably the most frequently used database model today. It has its strengths, but it doesn't perform very well with complex queries and analysis of very large sets of data. As computers have grown more potent, resulting in the possibility to store very large data volumes, the need for efficient analysis and processing of such data sets has emerged. The concept of *Online Analytical Processing (OLAP)* was developed to meet this need. The main OLAP component is the *data cube*, which is a multidimensional database model that with various techniques has accomplished an incredible speed-up of analysing and processing large data sets. A concept that is advancing in modern computing industry is *Business Intelligence (BI)*, which is fully dependent upon OLAP cubes. The term refers to a set of tools used for multidimensional data analysis, with the main purpose to facilitate decision making.

This thesis looks into the concept of BI, focusing on the OLAP technology and date cubes. Two different approaches to cubes are examined and compared; *Multidimensional Online Analytical Processing (MOLAP)* and *Relational Online Analytical Processing (ROLAP)*. As a practical part of the thesis, a BI project was implemented for the consulting company Sogeti Sverige AB. The aim of the project was to implement a prototype for easy access to, and visualisation of their internal economical data. There was no easy way for the consultants to view their reported data, such as how many hours they have been working every week, so the prototype was intended to propose a possible method. Finally, a performance study was conducted, including a small scale experiment comparing the performance of ROLAP, MOLAP and querying against the data warehouse. The results of the experiment indicates that ROLAP is generally the better choice for data cubing.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First of all, I would like to thank my two supervisors: Michael Minock at Umeå University, Department of Computing Science; and Tomas Agerberg at Sogeti.

I would like to express my gratitude to Sogeti for giving me the opportunity to do my master thesis there, and also to all employees who have made my time at the office a pleasant stay. I would specifically like to thank, in no particular order, Ulf Smedberg, Ulf Dageryd, Slim Thomsson, Jonas Osterman, Glenn Reian and Erik Lindholm for helping me out with practical details during my work.

Thanks also to Marie Nordqvist for providing and explaining the Agresso data that I have based my practical work on.

Finally I would like to thank Emil Ernerfeldt, Mattias Linde and Malin Pehrsson for taking the time to proofread my report and for giving me valuable feedback. Thanks also to David Jonsson for giving me some ideas about the report layout.

1

Introduction

This chapter introduces the thesis and Sogeti Sverige AB, the consulting company where it was conducted. A background to the project is given and the problem specification is stated as well as the project goals and the tools used during the process. The chapter ends with an outline of the report.

Business Intelligence (BI) is a concept for analysing collected data with the purpose to help decision making units get a better comprehensive knowledge of a corporation's operations, and thereby make better business decisions. It is a very popular concept in the software industry today and consulting companies all over the world have realized the need for these services. BI is a type of *Decision Support System (DSS)*, even though this term often has a broader meaning.

One of the consulting companies that are offering BI services is Sogeti Sverige AB, and this master thesis was conducted at their local office in Umeå. Sogeti Sverige AB is a part of the Sogeti Group with headquarters in Paris, France, and Capgemini S.A. is the owner of the company group. Sogeti offers IT Solutions in several sectors, for example Energy, Finance & Insurance, Forest & Paper, Transport & Logistics and Telecom.

1.1 Business Intelligence

The BI concept can be roughly decomposed into three parts: collecting data, analysing and reporting.

Collecting data for a BI application is done by building a data warehouse where data from multiple heterogeneous data sources is stored. Typical data sources are relational databases, plain text files and spread sheets. Transferring the data from the data sources to the data warehouse is often referred to as the *Extract, Transform and Load (ETL)* process. The data is extracted from the sources, transformed to fit, and finally the data is loaded into the warehouse. The ETL process often brings issues with data consistency between data sources; the same data can have a different structure, or the same data can be found in several data sources without coinciding. In order to load it into the data warehouse the data has to be consistent, and the process to accomplish this is called *data cleaning*.

A common tool for analysing the data is the *data cube*, which is a multidimensional data structure built upon the data warehouse. The cube is basically used to group data by several dimensions and selecting a subset of interest. This data can be analysed with tools for *data mining*, which is a concept for finding trends and patterns in the data.

The concept of data mining is outside the scope of this thesis and will not be discussed any further.

Finally, reporting is an important issue with BI. Reporting is done by generating different kinds of reports which often consists of *pivot tables* and diagrams. Reports can be uploaded to a report server from which the end user can access them, or the end user can connect directly to the data source and create *ad hoc* reports, i.e. based on the structure of the data cube the user can create a custom report by selecting interesting data and decide which dimensions to use for organizing it.

When analysing data with BI, a common way of organizing the data is to define *key performance indicators (KPIs)*, which are metrics used to measure progress towards organizational goals. Typical KPIs are revenue and gross operational profit.

1.2 Background

Sogeti is internally using the business and economy system *Agresso* which holds data about each employee, the teams they are organized in, which offices the employees are located at, projects the company is currently running, the clients who have ordered them and so on. There are also economical information such as costs and revenues, and temporal data that describes when the projects are running and how many hours a week each employee has been working on a certain project.

Agresso is used by the administration and decision making units of the company, and the single employee can access some of this data through charts describing the monthly result of the company. A project was initiated to make this data available to the employees at a more detailed level, but since it had low priority, it remained in it's early start-up phase until it was adopted as an MT project.

1.3 Problem Statement

Since Agresso is accessible only by the administration and decision making units of the organisation, the single employee can not even access the data that concerns himself. The aim of the project was to implement a prototype to make this possible. All employees should of course not have access to all data since that would be a threat to the personal integrity; an ordinary consultant should for example not be able to see how many hours another consultant have spent on a certain work related activity. Thus, what data to make available has to be carefully chosen.

The purpose of the project was to make relevant information available to all employees, by choosing adequate KPIs and letting the employees access those. The preferred way to present this information was by making it available on the company's intranet.

1.4 Goals

The following is a list of the project goals, each one described in more detail below.

- Identify relevant data
- Build a data warehouse and a data cube
- Present the data to the end user

- Automate the process of updating the warehouse and the cube

Identify relevant data

The first goal of the project was to find out what information could be useful, and what information should be accessible to the single employee. When analysing this kind of data for trends, KPIs are more useful than raw data, so the information should be presented as KPIs. There are lots of possible KPIs to use, therefore a few had to be chosen for the prototype.

Build a data warehouse and a data cube

The second goal was to design and implement a data warehouse and a data cube for the Agresso data to be stored. The data warehouse model had to be a robust model based on the indata structure, designed as a basis for building the data cube. With this cube it should be possible to browse, sort and group the data based on selected criteria.

Present the data to the end user

The cube data had to be visualized to the end user in some way. Since the data has a natural time dimension, some kind of graph would probably be appropriate. The third goal was to examine different ways of visualizing the data, and to choose a suitable option for the prototype. Preferably, the prototype should be available on Sogeti's intranet.

Automate the process of updating the warehouse and the cube

The process of adding new data and updating the cube should preferably be completely automatic. The fourth goal was to examine how this could be accomplished and integrate this functionality in the prototype.

1.5 Development environment

The tools available for the project was Sybase PowerDesigner, Microsoft SQL Server, Microsoft.NET and Dundas Chart. A large part of the practical part of the project was to learn these tools, doing this was done by reading books [9, 11, 13] and forums, and by doing several tutorials and reading articles on Microsoft Software Development Network[1]. The employees at Sogeti has also been a great knowledge base.

PowerDesigner is an easy-to-use graphical tool for database design. With this tool, a conceptual model can be developed using a drag-and-drop interface, after which the actual database can be generated as SQL queries.

Microsoft SQL Server is not only a relational database engine, it also contains three other important parts used for BI development; Integration Services, Analysis Services and Reporting Services.

Integration Services is a set of tools used mainly for managing the ETL process, but it is also usable for scripting and scheduling all kinds of database tasks. With this tool it is possible to automate processes by scheduling tasks that are to be performed regularly.

[1]See *http://www.msdn.com*

Analysis Services is the tool used for building data cubes. It contains tools for processing and deploying the cube as well as designing the cube structure, dimensions, aggregates and other cube related entities.

Reporting Services is used for making reports for the end user, as opposed to the tasks performed by Analysis Services and Integration Services, which are intended for system administrators and developers. Reporting Services provides a reporting server to publish the reports and tools for designing them.

1.6 Report outline

The rest of the report is structured as follows:

Chapter 2. The Relational Database Model explains the basic theory and concepts of relational databases, which are very central to data cubes and Business Intelligence.

Chapter 3. Online Analytical Processing describes the notion of Data Cubes. This chapter describes how cubes are designed and used, and it also has a part covering cube performance.

Chapter 4. Accomplishments describes the accomplishments of the whole project. This includes the implemented prototype, how the work was done and to what extent the goals were reached, and how the performance tests were made.

Chapter 5. Conclusions sums up the work and discusses what conclusions can be reached.

Appendix A. Performance Test Details contains details of the performance tests, such as the exact queries used and the test result figures.

Appendix B. Glossary contains a list of important terms and abbreviations used in the report with a short description of each.

2

The Relational Database Model

This chapter explains the basic theory behind the relational database model. Basic concepts, including Codd's first three normal forms, are defined, described and exemplified.

The relational database model was formulated in a paper by Edgar Frank Codd in 1970 [3]. The purpose of the model is to store data in a way that is guaranteed to always keep the data consistent even in constantly changing databases that are accessed by many users or applications simultaneously. The relational model is a central part of *Online Transaction Processing (OLTP)*, which is what Codd called the whole concept of managing databases in a relational manner. Software that is used to handle the database is referred to as *Database Management Systems (DBMS)*, and the term *Relational Database Management Systems (RDBMS)* is used to indicate that it is a relational database system. The adjective *relational* refers to the models fundamental principle that the data is represented by mathematical relations, which are implemented as tables.

2.1 Basic definitions

Definition. A *domain* D is a set of atomic values. [7]

Atomic in this context means that each value in the domain is indivisible as far as the relational model is concerned. Domains are often specified with a domain name and a data type for the values contained in the domain.

Definition. A *relation schema* R, denoted by $R(A_1, A_2, \ldots, A_n)$, is made up of a relation name R and a list of attributes $A_1, A_2, \ldots, A_n$. Each *attribute* A_i is the name of a role played by some domain D in the relation schema R. D is called the *domain* of A_i and is denoted by *dom(A_i)*. [7]

Table 2.1 is an example of a relation represented as a table. The relation is an excerpt from a database containing clothes sold by some company. The company has several retailers that sell the products to the customers, and the table contains information about what products are available, in which colours they are available, which retailer is selling the product and to what price. According to the above definition, the attributes of the example relation are: Retailer, Product, Colour and Price (€).

Example of related domains are:

- **Retailer_names:** The set of character strings representing all retailer names.

- **Clothing_products:** The set of character strings representing all product names.
- **Product_colours:** The set of character strings representing all colours.
- **Product_prices:** The set of values that represents possible prices of products; i.e. all real numbers.

Retailer	Product	Colour	Price(€)
Imaginary Clothing	Jeans	Blue	40
Imaginary Clothing	Socks	White	10
Imaginary Clothing	T-shirt	Black	15
Gloves and T-shirts Ltd.	T-shirt	Black	12
Gloves and T-shirts Ltd.	Gloves	White	12

Table 2.1: A simple example of a relation represented as a database table.

Definition. A *relation* (or *relation state*) r of the relation schema $R(A_1, A_2, \ldots, A_n)$, also denoted by $r(R)$, is a set of n*-tuples* $r = \{t_1, t_2, \ldots, t_m\}$. Each n-tuple t is an ordered list of n values $t = (v_1, v_2, \ldots, v_n)$, where each value $v_i, 1 \leq i \leq n$, is an element of $\text{dom}(A_i)$ or is a special *null* value. The i^{th} value in tuple t, which corresponds to the attribute A_i, is referred to as $t[A_i]$ (or $t[i]$ if we use the positional notation). [7]

This means that, in the relational model, every table represents a relation r, and each n-tuple t is represented as a row in the table.

Definition. A *functional dependency*, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a *constraint* on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$. [7]

The meaning of this definition is that if attribute X functionally determines attribute Y, then all tuples that have the same value for X, must also have the same value for Y. Observe that this definition does *not* imply $Y \rightarrow X$. In Table 2.1, the price of the product is determined by the product type and which retailer that sells it, hence $\{Retailer, Product\} \rightarrow Price$. This means that for a certain product sold by a certain retailer, there is only one possible price. Since the opposite is not true, knowing the price doesn't necessarily mean that we can determine the product or the retailer; the product that costs 12 € could be either the gloves or the t-shirt sold by *Gloves and T-shirts Ltd.*

Definition. A *superkey* of a relation schema $R = \{A_1, A_2, \ldots, A_n\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. A *key* K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more. [7]

Hence, there can be only one row in the database table that has the exact same values for all the attributes in the superkey set. Another conclusion of the definition above is that every key is a superkey, but there are superkeys that are not keys. Adding an additional attribute to a key set disqualifies it as a key, but it's still a superkey. At the other hand, removing one attribute from a key set disqualifies it as both a superkey and a key. Since keys and superkeys are sets, they can be made out of single or multiple attributes. In the latter case the key is called a *composite key.*

Definition. If a relation schema has more than one key, each is called a *candidate key*. One of the candidate keys is arbitrarily designated to be the *primary key*, and the others are called *secondary keys*. [7]

Definition. An attribute of relation schema R is called a *prime attribute* of R if it is a member of *some candidate key* of R. An attribute is called *nonprime* if it is not a prime attribute - that is, if it is not a member of any candidate key. [7]

In a database system, the primary key of each relation is explicitly defined when the table is created. Primary keys are by convention underlined when describing relations schematically, and this convention is followed in the examples below. Two underlined attributes indicates that they both contribute to the key, i.e. it is a composite key.

Definition. A set of attributes FK in relation schema R_1 is a *foreign key* of R_1 that references relation R_2 if it satisfies both: (a) the attributes in FK have the same domain(s) as the primary key attributes PK of R_2; and (b) a value of FK in a tuple t_1 of the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state $r_2(R_2)$ or is *null*.

Having some attributes in relation R_1 forming a foreign key that references relation R_2 prohibits data to be inserted into R_1 if the foreign key attribute values are not found in R_2. This is a very important referential integrity constraint that avoids references to non-existing data.

2.2 Normalization

The most important concept of relational databases is *normalization*. Normalization is done by adding constraints to how data can be stored in the database, which implies restrictions to the way data can be inserted, updated and deleted. The main reasons for normalizing database design is to minimize information redundancy and to reduce disk space required to store the database. Redundancy is a problem because it opens up the possibility to make the database inconsistent. Codd talked about *update anomalies*, which is classified into three categories; *insertion anomalies*, *deletion anomalies* and *modification anomalies* [7].

Insertion anomalies occur when a new entry that is not consistent with existing entries is inserted into the database. For example, if a new product sold by *Imaginary Clothing* is added in Table 2.1, then the name of the company must be spelled the exact same way as all the other entries in the database that contain the company name, otherwise we have two different entries for the same information.

If we assume that Table 2.1 only contains products that currently are available, then what happens when a product, for example gloves, run out of stock? The gloves entry will be removed from the table, and we have lost the information that the gloves are white and that they can be bought from *Gloves and T-shirts Ltd.* We have even lost the information that gloves is a product! This is an example of a deletion anomaly.

Finally, suppose that the retailer *Gloves and T-shirts Ltd.* decides to start selling jeans too, and therefore they change name to include the new product. All entries in the database that contain the retailer name must now be updated, or else the database will have two different names for the same retailer. This is an example of modification anomalies.

These issues may seem insignificant when looking at small examples such as the one above, but in a large and complex database that is constantly updated it is very important that the data is consistent. A normalized database makes sure that every piece of information is stored in only one place, thus modification of the data only has to be done once. Storing the same data in one place instead of once per entry will also save a lot of storage space.

Normalization is accomplished by designing the database according to the *normal forms*. Below, Codd's first three normal forms will be described. Those are the most common normal forms used when designing databases, but there are several other normal forms that are not mentioned here. Each one of the three first normal forms implies the preceding one, i.e. 3NF implies 2NF, and 2NF implies 1NF.

Definition. A relation schema R is in *first normal form (1NF)* if, for all attributes A_i, the domain D of A_i only contains atomic values, and for all n-tuples t, all values $t[A_i]$ are single values from D.

Retailer	**Product**	**Colour**	**Price(€)**
Imaginary Clothing	Jeans	Blue	40
Imaginary Clothing	Socks	White, Blue	10
Imaginary Clothing	T-shirt	Black	15
Gloves and T-shirts Ltd.	T-shirt	Black	12
Gloves and T-shirts Ltd.	Gloves	White	12

Table 2.2: An example of a violation to 1NF, information about the blue and the white socks sold by Imaginary Clothing must be separated into two different rows.

Retailer	**Product Id**	**Product Name**	**Colour**	**Price(€)**
Imaginary Clothing	1	Jeans	Blue	40
Imaginary Clothing	2	Socks	White	10
Imaginary Clothing	3	Socks	Blue	10
Imaginary Clothing	4	T-shirt	Black	15
Gloves and T-shirts Ltd.	4	T-shirt	Black	12
Gloves and T-shirts Ltd.	5	Gloves	White	12

Table 2.3: Example of how the relation in Table 2.2 could be changed into 1NF.

1NF simply means that in every row of a table, there can only be one value per attribute. Table 2.2 exemplifies a violation of the first normal form by having two different colours of socks in the same row. To turn this table into 1NF, every colour of the same product has to be stored in a separate row. Table 2.3 illustrates how the table could be modified to not violate 1NF.

As an additional modification that has to do with convenience and not with 1NF, the attribute *Product* has been split into two: *Product Id* and *Product Name*. Every product has been given a product id, and using this id we can determine the product name and the colour. *Product Id* and *Retailer* together have been chosen to be the primary key.

Definition. A relation schema R is in *second normal form (2NF)* if every nonprime attribute A in R is not partially dependent on *any* key of R. [7]

2NF means that every attribute that is not contained in the key set must provide a fact about the key, the whole key and nothing but the key. For example, if the key is a composite key, then no attribute is allowed to provide a fact about only one of the prime attributes. Table 2.3 violates 2NF because the product name and the colour is determined only by product id, but the primary key for the table is a composition of product id and retailer. Table 2.4 (a) and (b) illustrates a possible decomposition of the table that doesn't violate 2NF. Product Id has become the primary key in Table 2.4 (b), and in Table 2.4 (a) it is part of the primary key, but also a foreign key that references (b). Observe also how this restructuring removed the redundancy of the product names.

Retailer	**Product Id**	**Price (€)**
Imaginary Clothing	1	40
Imaginary Clothing	2	10
Imaginary Clothing	3	10
Imaginary Clothing	4	15
Gloves and T-shirts Ltd.	4	12
Gloves and T-shirts Ltd.	5	12

Table 2.4: (a) A decomposition into 2NF of Table 2.3.

Product Id	**Product Name**	**Colour**
1	Jeans	Blue
2	Socks	White
3	Socks	Blue
4	T-shirt	Black
5	Gloves	White

Table 2.4: (b) A decomposition into 2NF of Table 2.3.

Definition. A relation schema R is in *third normal form (3NF)* if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R. [7]

The third normal form implies that there can be no transitional dependencies, that is if A alone is the key attribute, $A \rightarrow B$ and $A \rightarrow C$, then $B \nrightarrow C$ must hold.

Figure 2.1 illustrates a schematic view of the functional dependencies allowed by the different normal forms. Key is a single key attribute, prime1 and prime2 is together a composite key. A and B are attributes that are determined by the key attribute(s), and the arrows indicates functional dependencies. The figure also illustrates, just as stated above, that all relations fulfilling the requirements for 3NF also fulfils the requirements for 2NF and 1NF.

2.3 Indexing

An important issue of relational databases is how to optimize the database for querying. The most common method is to use indexing, that is having an index structure containing pointers to the actual data in order to optimize search operations. A good index structure can increase the performance of a search significantly. The index structure that is the primary choice for most relational database systems today is the B^+*-tree* [5].

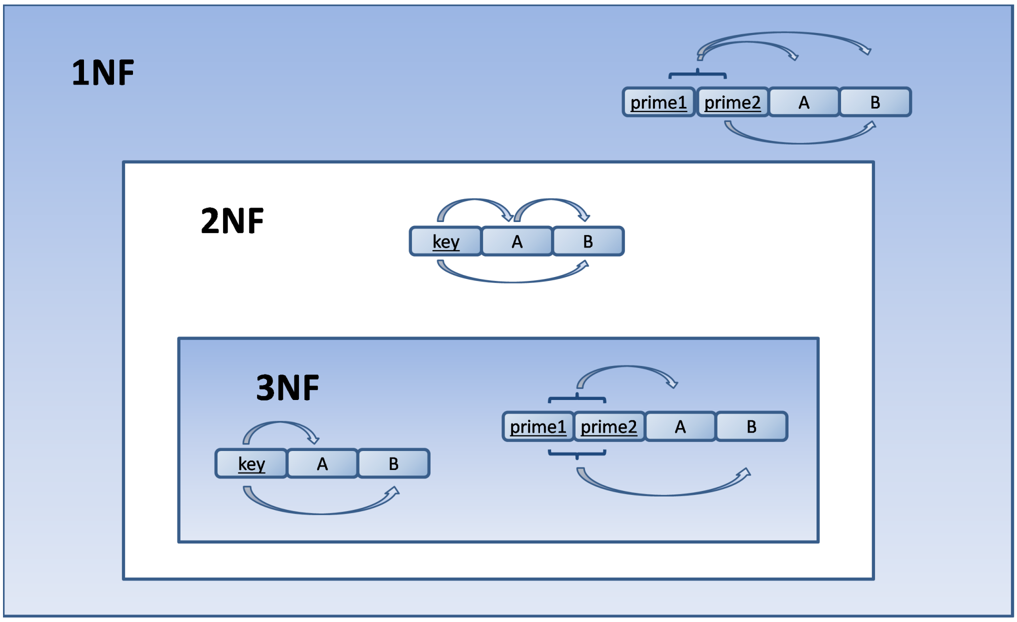

Figure 2.1: Overview of what dependencies are allowed by the first three normal forms. A and B are non-prime attributes, prime1 and prime2 are prime attributes that makes a composite primary key, and key is a single primary key. Functional dependencies between attributes are represented as arrows.

The B^+-tree is a tree structure where every node consists of a maximum of d keys and $d + 1$ pointers. The number d is fixed for a certain tree, but the exact number of keys in each node can differ. Each pointer is referring to another tree node, or to the indexed data if the node is a leaf node. The leaf nodes are also linked together with pointers in order to get a fast sequential access. Figure 2.2 illustrates an example, the actual data is marked with a dashed border.

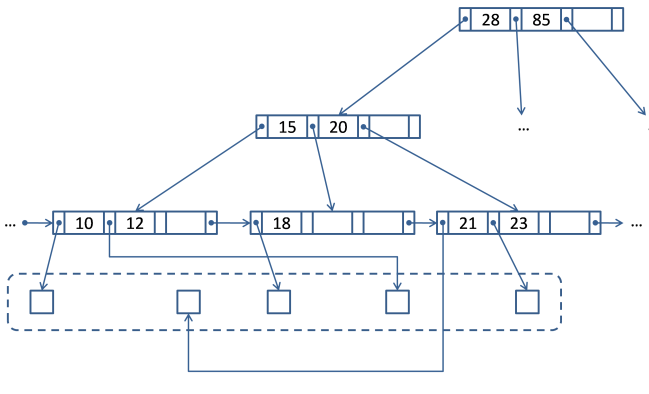

Figure 2.2: An example of a B^+-tree, the data that the leaf nodes point to is marked with a dashed border.

To find a specific value in the database, we have to find its key in the tree. Assume we want to find the value that corresponds to the key X. The search starts in the root node, and every key in the current node is compared to X until a key that is equal to or greater than X is found, or until there are no more keys in the node. When this happens, the associated pointer is followed to the next node, and the procedure is repeated. The search stops when the leaf level is reached and if the key is found there, the data can be retrieved by following the correct pointer.

The B^+-tree is always kept balanced, i.e. every leaf node in the tree has the same depth. This is accomplished by changing the structure of the tree during insertion and deletion, but the details of these algorithms are not discussed here.

The B^+-tree is an advantageous index structure for databases because of its performance. Maintenance of the structure demands some extra time because of the restructuring of the tree when inserting and deleting records, but this loss is small compared to the gain of searching the tree. The fact that both random and sequential access is possible is another reason why this type of indexing is so common.

3

Online Analytical Processing

This chapter describes the concept of data cubes and data warehousing in the context of Online Analytical Processing (OLAP). The principles of data cubes are explained as well as how they are designed and what they are used for. The relationship between data cubes, data warehouses and relational databases is also examined.

The strength of OLTP databases is that they can perform large amounts of small transactions, keeping the database available and the data consistent at all time. The normalization discussed in Chapter 2 helps keeping the data consistent, but it also introduces a higher degree of complexity to the database, which causes huge databases to perform poorly when it comes to composite aggregation operations. In the context of business it is desirable to have historical data covering years of transactions, which results in a vast amount of database records to be analyzed. It is not very difficult to realize that performance issues will arise when processing analytical queries that requires complex joining on such databases. Another issue with doing analysis with OLTP is that it require rather complex queries, specially composed for each request, in order to get the desired result.

3.1 OLAP and Data Warehousing

In order to handle the above issues, the concept of *Online Analytical Processing (OLAP)* has been proposed and widely discussed through the years and many papers have been written on the subject.

OLTP is often used to handle large amounts of short and repetitive transactions in a constant flow, such as bank transactions or order entries. The database systems are designed to keep the data consistent and to maximize transaction throughput. OLAP databases are at the other hand used to store historical data over a long period of time, often collected from several data sources, and the size of a typical OLAP database is often orders of magnitude larger than that of an ordinary OLTP database. OLAP databases are not updated constantly, but they are loaded on a regular basis such as every night, every week-end or at the end of the month. This leads to few and large transactions, and query response time is more important than transaction throughput since querying is the main usage of an OLAP database [2].

The core of the OLAP technology is the *data cube*, which is a multidimensional database model. The model consists of *dimensions* and numeric metrics which are referred to as *measures*. The measures are numerical data such as revenue, cost, sales and budget. Those are dependent upon the dimensions, which are used to group the data similar to the *group by* operator in relational databases. Typical dimensions are time, location and product, and they are often organized in *hierarchies.* A hierarchy is a structure that defines levels of *granularity* of a dimension and the relationship between those levels. A time dimension can for example have hours as the finest granularity, and higher up the hierarchy can contain days, months and years. When a cube is queried for a certain measure, ranges of one or several dimensions can be selected to filter the data.

The data cube is based on a *data warehouse*, which is a central data storage possibly loaded with data from multiple sources. Data warehouses tend to be very large in size, and the design process is a quite complex and time demanding task. Some companies could settle with a *data mart* instead, which is a data warehouse restricted to a departmental subset of the whole data set. The data warehouse is usually implemented as a relational database with tables grouped into two categories; dimension tables and fact tables.

A *dimension table* is a table containing data that defines the dimension. A time dimension for example could contain dates, names of the days of the week, week numbers, months names, month numbers and year. A *fact table* contains the measures, that is aggregatable data that can be counted, summed, multiplied, etc. Fact tables also contain references (*foreign keys*) to the dimension tables in the cube so the facts can be grouped by the dimensional data.

A data warehouse is generally structured as a *star schema* or a *snowflake schema.* Figure 3.1 illustrates an example of a data warehouse with the star schema structure. As seen in the figure, a star schema has a fact table in the middle and all dimension tables are referenced from this table. With a little imagination the setup could be thought of as star shaped.

In a star schema, the dimension tables do not have references to other dimension tables. If they do, the structure is called a snowflake schema instead. A star schema generally violates the 3NF by having the dimension tables being several tables joined together, which is often preferred because of the performance loss that 3NF causes when the data sets are very large. If it for some reason is desirable to keep the data warehouse in 3NF the snowflake schema can be used. Figure 3.2 illustrates an example of how Figure 3.1 could look like if it were to be structured as a snowflake schema.

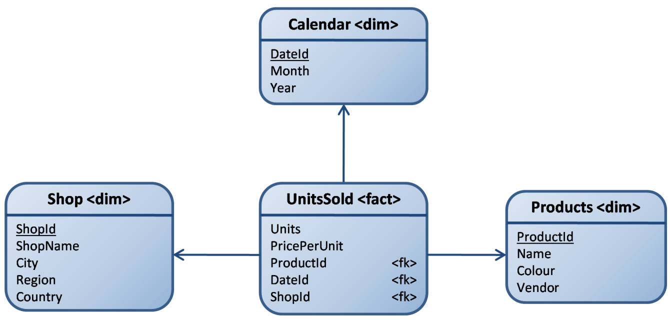

Figure 3.1: Example of a star schema. The foreign keys are marked with <*fk*>.

In 1993, Codd et al. published a white paper [4] where the need for OLAP services

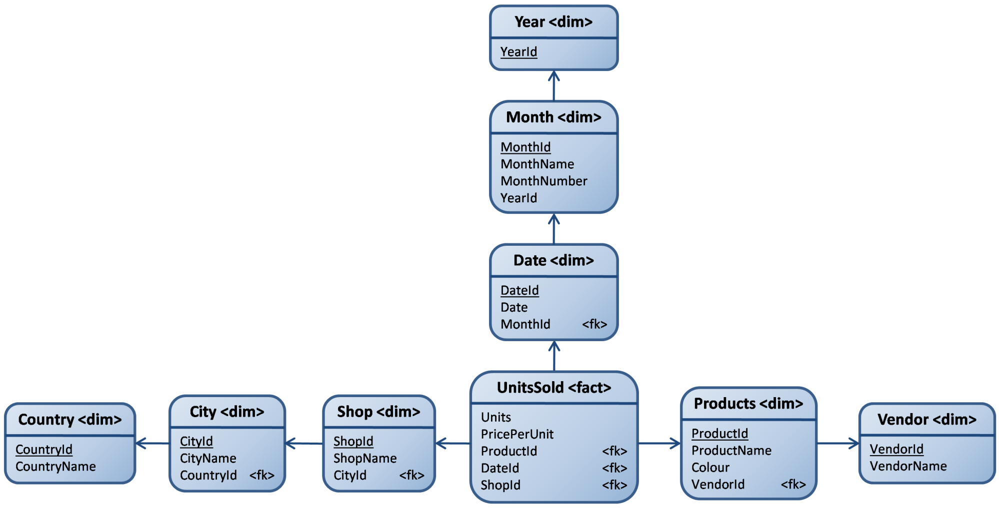

Figure 3.2: Example of how Figure 3.1 could be organized as a snowflake schema. The foreign keys are marked with <*fk*>.

is discussed, and the requirements for OLAP are summarized in the following twelve rules:

1. **Multidimensional Conceptual View**
   The end user's view of the OLAP model must be multidimensional since the analysis to be done is multidimensional by nature. Note that having a multidimensional view doesn't necessarily mean that the data is actually stored multidimensionally.

2. **Transparency**
   The OLAP architecture should be transparent, i.e. facts such as if there is a client-server architecture behind the application, or at what layer the OLAP functionality lies, should be transparent to the user.

3. **Accessibility**
   The OLAP tool must be able to map its own logical schema on data collected from different, heterogeneous, relational and non-relational data sources. The end user should not be concerned with where the data comes from, or how it is physically stored; the data should appear as a single, consistent view.

4. **Consistent Reporting Performance**
   As the number of dimensions and the size of the database increases, the user should not notice any significant reporting performance degradation. In cases where performance becomes an issue, the OLAP tool could propose alternative strategies such as presenting the information in some other way.

5. **Client-Server Architecture**
   It is mandatory for OLAP tools to be capable of operating in a client-server environment since a data cube often is a central storage accessed remotely by client applications.

6. **Generic Dimensionality**
   Every data dimension must be equivalent in structure and operational capabilities. There can be additional features granted to specific dimensions, but the basic structure must be the same for all dimensions, and thus additional features must be possible to grant any dimension.

7. **Dynamic Sparse Matrix Handling**
   A typical data cube is very sparse. Therefore, the OLAP tool should handle sparse matrices in a dynamic way to avoid letting the cube size grow unnecessary. There is no need to calculate aggregations for each possible cell in a cube if only a small fraction of the cells actually contains data.

8. **Multi-User Support**
   OLAP tools must provide concurrent access to the data and the OLAP model, in a way that preserves integrity and security.

9. **Unrestricted Cross-dimensional Operations**
   An OLAP tool must be able to do calculations and other operations across dimensions without requiring the user to explicitly define the actual calculation. The tool should provide some language for the user to utilize in order to express the desired operations.

10. **Intuitive Data Manipulation**
    Drill-downs, roll-ups and other operations that lie in the nature of dimension hierarchies[1] should be accessible with ease via direct manipulation of the displayed data, and should not require unnecessary user interface operations such as menu navigation.

11. **Flexible Reporting**
    The reporting should be flexible in the sense that rows, columns and page headers in the resulting report must be able to contain any number of dimensions from the data model, and that each dimension chosen must be able to display its members and the relation to them, e.g. by indentation.

12. **Unlimited Dimensions and Aggregation Levels**
    In the article, Codd et al. states that a serious OLAP tool must be able to handle at least fifteen and preferably twenty dimensions within the same model. Maybe this rule should be extended to allow an unlimited amount of dimensions, just as the name of the rule implies. In either case, it is also stated that each dimension must allow for an unlimited amount of user defined aggregation levels within the dimension hierarchy.

The paper clearly states that OLAP should not be implemented as a new database technology since the relational database model is 'the most appropriate technology for enterprise databases'. It also states that the relational model never was intended to offer the services of OLAP tools, but that such services should be provided by separate end-user tools that complements the RDBMS holding the data.

[1] Codd et al. use the term *consolidation path* in their paper, but the term *dimension hierarchy* is more common in recent research and literature.

3.2 Cube Architectures

Even though the basic cube principles are the same, the cube can be implemented in different ways. Different vendors advocate different architectures, and some offers the possibility to choose an architecture for each cube that is created.

There are two main architectures that traditionally have been discussed in the field of database research; *Multidimensional OLAP (MOLAP)* and *Relational OLAP (ROLAP)*.

Multidimensional Online Analytical Processing is based upon the philosophy that since the cube is multidimensional in its nature, the data should be stored multidimensionally. Thus, the data is copied from the data warehouse to the cube storage and aggregations of different combinations of dimensions are pre-calculated and stored in the cube in an array based data structure. This means that the query response time is very short since no calculations has to be done at the time a query is executed. At the other hand, the loading of the cube is an expensive process because of all the calculations that have to be done, and therefore the data cube is scheduled to be loaded when it is unlikely to be accessed, on regular intervals such as once every week-end or every night.

A problem that has to be considered when working with MOLAP is *data explosion*. This is a phenomena that occurs when aggregations of all combinations of dimensions are to be calculated and stored physically. For each dimension that is added to the cube, the number of aggregations that is to be calculated increases exponentially.

Relational Online Analytical Processing is, just as the name suggests, based on the relational model. The main idea here is that it is better to read data from the data warehouse directly, than to use another kind of storage for the cube. As is the case with MOLAP, data can be aggregated and pre-calculated in ROLAP too, using *materialized views*, i.e. storing the aggregations physically in database tables. A ROLAP architecture is more flexible since it can pre-calculate some of the aggregations, but leave others to be calculated on request.

Over the years, there has been a great debate in the research field whether OLAP should be implemented as MOLAP or ROLAP. The debate has however faded, and in the last decade most researchers seem to argue that ROLAP is superior to MOLAP, among others a white paper from MicroStrategy [10]. The arguments are that ROLAP perform almost as good as MOLAP when there are few dimensions, and when there are too many dimensions MOLAP can't handle it because of the data explosion. Already with about 10-20 dimensions the number of calculations becomes tremendously large in consequence of the exponential growth. Another important aspect is the greater flexibility of ROLAP; all aggregations don't need to be calculated beforehand, they can be calculated on demand as well.

3.3 Relational Data Cubes

One way to implement relational data cubes was proposed by Gray et al. 1997 [8]. In this paper the *cube operator* - which is an n-dimensional generalization of the well-known *group by* - and the closely related *rollup* operator is described in detail. The cube and the rollup operator has been implemented in several relational database engines to provide data cube operability.

Consider the example figures in Table 3.1. A retailer company has sold jeans and gloves in two colours; black and blue. The total amount of items sold during 2007 and 2008 is summed separately for each year and colour in the *Sales* column. Now, a manager would probably want to look at the figures in the form of sums of the total

Product	Year	Colour	Sales
Jeans	2007	Black	231
Jeans	2007	Blue	193
Jeans	2008	Black	205
Jeans	2008	Blue	236
Gloves	2007	Black	198
Gloves	2007	Blue	262
Gloves	2008	Black	168
Gloves	2008	Blue	154

Table 3.1: Example table of sales for some company selling clothing products.

sales during one year, the total amount of jeans sold during 2007, or maybe the amount of blue gloves sold during 2008. One intuitive way to arrange the data for this purpose is to use a *pivot table*. In this kind of table, data is structured with both row and column labels and is also summed along its dimensions in order to get a clear view of the data set. Table 3.2 is an example of how Table 3.1 could be arranged as a pivot table.

	2007		2007	2008		2008	
	Black	Blue	Total	Black	Blue	Total	Grand Total
Jeans	231	193	424	205	236	441	865
Gloves	198	262	460	168	154	322	782
Grand Total	429	455	884	373	390	763	1647

Table 3.2: Example of a pivot table.

The cube operator expresses the pivot table data set in a relational database context. This operator is basically a set of *group by* clauses put together with a union operation. It groups the data by the given dimensions (attributes), in this case *Product*, *Year* and *Colour*, and aggregations are made for all possible combinations of dimensions. Table 3.3 illustrates the resulting data set when the cube operator is used combined with the *SUM* operator.

The special *ALL* value has been introduced to indicate that a certain record contains an aggregation over the attribute that has this value. The *NULL* value can be used instead of the *ALL* value in order not to manipulate the SQL language.

The cube operator results in a lot of aggregations, but all aggregated combinations are seldom desired and therefore the rollup operator was also proposed. This operator works the same way as the cube operator, but with the difference that new aggregations are calculated only from already calculated aggregations. Table 3.4 illustrates the results from a rollup operation.

Algorithms have been developed in order to improve the performance of the cube operator and a few of those are compared by Agarwal et al. [1]. All of the algorithms compared utilize the fact that an aggregation using a *group by* operation (below simply called a *group by*) in general does not have to be computed from the actual relation, but can be computed from the result of another *group by*. Therefore only one *group by* really has to be computed from the relation, namely the one with the finest granularity, and all the others can be computed from this result.

Figure 3.3 illustrates how *group by* results can be used to compute other aggregations.

Product	Year	Colour	Sales	Product	Year	Colour	Sales
Jeans	2007	Black	231	Gloves	2008	ALL	322
Jeans	2007	Blue	193	Gloves	ALL	Black	366
Jeans	2007	ALL	424	Gloves	ALL	Blue	416
Jeans	2008	Black	205	Gloves	ALL	ALL	782
Jeans	2008	Blue	236	ALL	2007	Black	429
Jeans	2008	ALL	441	ALL	2007	Blue	455
Jeans	ALL	Black	436	ALL	2007	ALL	884
Jeans	ALL	Blue	429	ALL	2008	Black	373
Jeans	ALL	ALL	865	ALL	2008	Blue	390
Gloves	2007	Black	198	ALL	2008	ALL	763
Gloves	2007	Blue	262	ALL	ALL	Black	802
Gloves	2007	ALL	460	ALL	ALL	Blue	845
Gloves	2008	Black	168	ALL	ALL	ALL	1647
Gloves	2008	Blue	154				

Table 3.3: The resulting set of the cube operator aggregating the sum operator, applied to Table 3.1.

Product	Year	Colour	Sales	Product	Year	Colour	Sales
Jeans	2007	Black	231	Gloves	2007	Black	198
Jeans	2007	Blue	193	Gloves	2007	Blue	262
Jeans	2007	ALL	424	Gloves	2007	ALL	460
Jeans	2008	Black	205	Gloves	2008	Black	168
Jeans	2008	Blue	236	Gloves	2008	Blue	154
Jeans	2008	ALL	441	Gloves	2008	ALL	322
Jeans	ALL	ALL	865	Gloves	ALL	ALL	782
ALL	ALL	ALL	1647				

Table 3.4: The resulting set of the rollup operator when it was applied on Table 3.1.

A, B, C and D are attributes in the relation, and every node in the graph represents the resulting set of a group by on the attributes in the node. The ALL attribute in the top node indicates the empty group by. The bottom node contains the attributes $\{A, B, C, D\}$, which means the node represents the result of a group by on these four attributes. Using this result, group by operations on $\{A, B, C\}$, $\{A, B, D\}$, $\{A, C, D\}$ and $\{B, C, D\}$ can be computed. Following the graph upwards, we see that the result of a group by on $\{A, B, C\}$ can be used to compute the results of group by operations on $\{A, B\}$, $\{A, C\}$ and $\{B, C\}$, etc.

This technique improves performance by minimizing disk I/O, but also by minimizing the number of aggregations by not actually aggregating all possible combinations that is the result of the cube operator. The differences between the algorithms is basically that they use different strategies for selecting which results to use for calculating new results, or in other words: they choose different paths in a graph such as Figure 3.3.

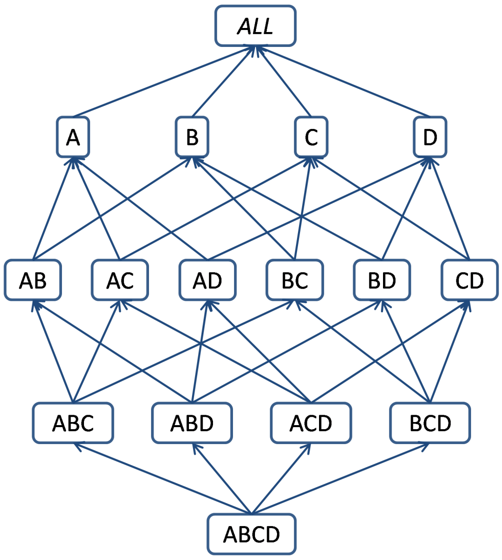

Figure 3.3: Graph describing how group by operations can be calculated using earlier calculated group by operations instead of the data source.

3.4 Performance

As mentioned earlier, performance is one of the big issues when handling these amounts of data. There are however methods developed for improving the performance of data cubes, both for MOLAP and ROLAP.

3.4.1 Clustering

When talking about database performance, the disk I/O is a problematic bottleneck since disk access results in a high latency compared to memory access. A solution widely used for minimizing disk access is *clustering* [7]. The basic principle of clustering is to organize the disk structure of the data in a way that makes sure that records that are expected to be queried together are physically stored together. This means that large blocks of data can be read once, instead of accessing the disk once for each record which would be the case if the data was spread out.

3.4.2 Bitmap Indexing

A common way to index relational databases is to use a B^+-tree for searching (see Chapter 2), and using *Row Identifiers (RIDs)* in the leaf nodes to specify the physical disk address where the row data is written. If the data is known to only be read, like in an OLAP database, then the query performance can be essentially improved by using a *Bitmap Index* [12]. If the data is not read-only as is the case in a typical OLTP database, then the bitmap index is not a very good idea to use since its maintenance cost is too high for insert and delete operations.

The basic idea of a bitmap index is that for each possible value in the column that is to be indexed, a bitmap is created with one bit for each row in the table. In a certain bitmap, all bits that correspond to the represented value is set to '1' and all other bits are set to '0'.

Product	**Colour**	**B_1**	**B_2**	**B_3**
Jeans	Blue	1	0	0
Socks	Black	0	0	1
T-shirt	Black	0	0	1
T-shirt	Blue	1	0	0
Gloves	White	0	1	0

Table 3.5: An example of bitmap indexing.

Table 3.5 illustrates the idea of a bitmap index; it contains a few products and their colours, where the colour column is the one to be indexed. For every possible colour, a bitmap B_i is created that keeps track of all tuples in the relation that has that colour as the value, e.g. all tuples with the attribute value 'Blue' is represented by B_1='10010'.

Provided that the *cardinality* of a column isn't too high, i.e. the number of possible values for that column isn't too large, there are two improvements introduced by bitmap indices. The first is that the disk size required to store the indices are reduced compared to using B^+-trees. In the above example, three bits are required for each value, compared to the size of a disk address when using RIDs. The second improvement is performance. The bitwise operators *AND*, *OR* and *NOT* are very fast, and using those it is possible to execute queries with complex *WHERE* clauses in a very efficient way. Aggregation operations like *SUM* and *AVG* is also affected by the choice of index.

It doesn't take long to realize that a bitmap index isn't of much use if the value to be indexed is an integer which can have several thousand possible values. Instead, a variant called *bit-sliced index* can be used in such cases to preserve the benefit of bitmap indices. We assume that the target column is populated by integer values represented by N bits. The bit-slice index assign one bitmap for each of the N bits, instead of one bitmap for each possible value. Table 3.6 illustrates the idea; the amount of sold items for each product is represented by its binary encoding with column B_3 representing the least significant and column B_1 representing the most significant bit. Reading these binary codes column by column gives the bitmaps in the index, e.g. B_1 = '1100'.

Product	**Amount sold**	**B_1**	**B_2**	**B_3**
Jeans	5	1	0	1
Socks	7	1	1	1
T-shirt	3	0	1	1
Gloves	2	0	1	0

Table 3.6: An example of bit-sliced indexing.

Using this index method, a great query performance increase will will be gained when it comes to aggregating.

If a relation scheme has a high cardinality, the size of the bitmap index will grow large rather quickly because every possible value will give rise to another bitmap, with as many bits as there are tuples in the relation. In these cases there are efficient compression techniques that reduce the bitmap sizes.

Two common compression schemes that have these properties are the byte-aligned bitmap code (BBC) and the word-aligned hybrid code (WAH), examined thoroughly by Wu, Otoo and Shoshani [14, 15]. The BBC is based on byte operations and the WAH is a variant of BBC that is based on words which helps word-based CPUs handle this scheme more efficiently. Basically, both methods work by finding sequences of consecutive bits that have the same value, and replacing these with the bit value and a counter that keeps track of how many bits there are in the sequence. Wu et al. have shown that these compression techniques result in indices that in worst case are as large as the corresponding B^+-tree, but often much smaller. Besides, many logical operations can be performed on the data without decompressing it.

3.4.3 Chunks

Another technique that aims at speeding up OLAP queries was proposed by Deshpande, Ramasamy, Naughton and Shukla in 1998 [6]. The idea is to cache the results of requested queries, but in *chunks* instead of the whole result. In a multidimensional environment, different queries requests results along different dimensions, and often the result sets intersects, resulting in common subsets. Using traditional query caching, these common subsets can not be cached and reused other than if the whole query is passed again. Using chunks, smaller parts of each query is cached, and when a new query is requested the caching engine finds which chunks exist in the cache, and send those to the query along with newly calculated chunks that don't exist in the cache. With this technique, query results can be reused in a more efficient fashion, which means that frequently requested data can be stored in chunks in the memory and doesn't have to be read from disk for every query.

A further optimization proposed by Deshpande et al. is to organize the backend file system by chunks, either by implementing support for chunk files in the DBMS, or by letting an existing RDBMS index the files by a chunk attribute that is added to the tables. This is a form of clustering that is shown to be efficient for multidimensional data.

4

Accomplishments

This chapter describes the implementation process step by step, how the work was done and the problems that occurred during the development. It describes the different parts of the implementation and the last part of the chapter describes the performance study that was conducted.

4.1 Overview

The first goal of the project was to identify relevant data. In order to do this, an understanding of the source data, how it was structured and what it was used for was essential, which in turn requires an overall understanding of the company's organisation. This was acquired through discussions with economic administration employees, and these discussions were the basis for learning about the KPIs used by the company and for understanding the layout of the Excel spreadsheet containing the source data. When this was done, a few KPIs of interest were chosen as focus during the project; *Utilization Rate, Vacation Excluded (URVE)* and *Bonus Quotient*.

URVE is calculated by dividing billable hours by the total number of hours worked, not counting hours of vacation (vacation hours are inserted into the database, but as the name of the KPI suggests, vacation is excluded). Billable hours are hours that the client can be charged with, in contrast to non-billable hours which are for example hours spent on internal education or parental leave. URVE was chosen because it represents something that is very important to every consulting company: the number of billable hours compared to the total number of hours that the employees has worked. This is strongly connected to the company's profitability since billable hours are paid by the clients while all other hours are paid by the company itself.

Bonus Quotient is the number of bonus generating hours during one month divided by possible hours the very same month. Bonus generating hours is a set of hour types that generate bonus, mainly billable hours. This second KPI was chosen because it is interesting to the consultants. Being able to access this figure gives an indication how much bonus they will acquire.

The next goal was to build a data warehouse and a data cube. Both were developed in parallel because the cube is based on the data warehouse and a change in the latter implies a restructure of the cube. An existing but incomplete database model was given in the beginning of the project. This first draft was an internal experiment which was made by a small project group at Sogeti, but it was abandoned due to lack of time until it became an MT project.

The third goal was to present the data to the end user. There are plenty of ways to access a data cube and look at the data, so three methods were examined within the limits of the project. These methods are: using Reporting Services, connecting directly to the cube via Microsoft Excel, and finally implementing a .NET application for data visualization using Dundas Chart.

The last goal was to automate the process of updating the warehouse and the cube. In order to fill the data warehouse and the cube with data, several queries to convert the data were written. These queries were executed through scripting in Integration Services. Cube loading and processing tasks were also incorporated in these scripts, which were written with the purpose to avoid manual handling of the cube update process as much as possible. However, the indata was received as Excel spreadsheets via e-mail, and even though the Integration Services script handled the update process it still had to be started manually.

4.2 Implementation

Figure 4.1 shows a rough overview of the components used in the system. The different parts will be described in more details below.

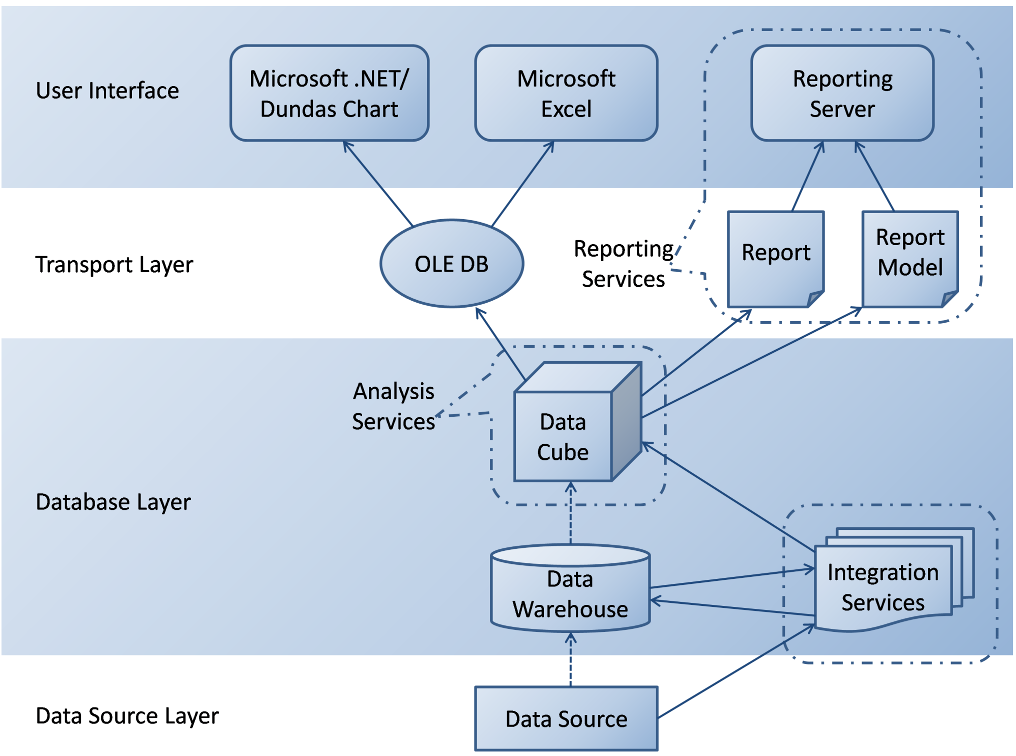

Figure 4.1: Overview of the system.

4.2.1 Data Storage

The origin of the data was the economy system Agresso. However, Agresso itself wasn't directly accessible during the development process, so the data source used was an Excel spreadsheet containing an excerpt of the data at a given time. The data in the spreadsheet was imported into a temporary table in the database with columns corresponding to the ones in the Excel file. A set of SQL queries were written to extract the data from the temporary table, transform and insert it into the data warehouse.

The data warehouse model was developed using Sybase PowerDesigner, which is an easy-to-use graphical design tool for designing databases. This tool was used to create a conceptual model of the data warehouse and from there generate the actual database in terms of SQL queries. The first model of the database was completely normalized, which turned out not to be useful at all. A normalized data warehouse makes it difficult to build the cube, and the cube operations will perform quite poorly because of the additional *joins* that the normalization introduces. A better way to design a data warehouse is to build it as a star schema. The model implemented in this project was structured close to a star schema, but for practical reasons some of the tables are somewhat normalized. Figure 4.2 illustrates the final data warehouse model.

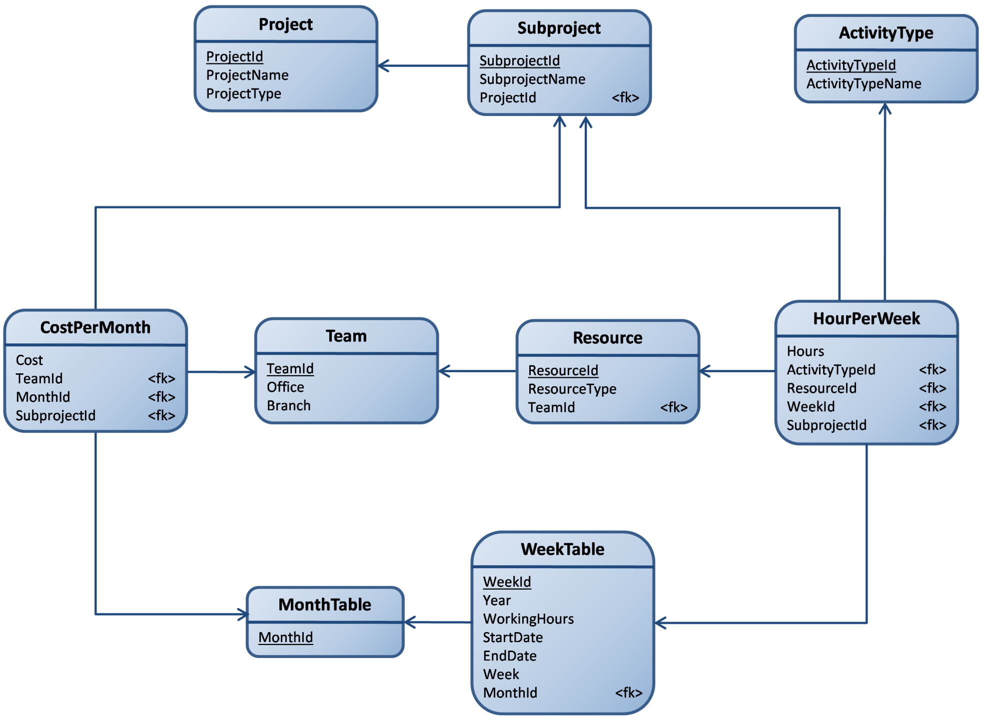

Figure 4.2: The data warehouse schema.

The data cube was built using Analysis Services. Before building the actual cube, a data source and a data source view have to be defined. The data source in this case was the data warehouse, and the data source view contains most of the tables as is. The KPIs chosen for the prototype were calculated by creating a named query each.

4.2.2 Presentation

Excel was examined as an alternative for reporting, see Figure 4.3 for a screenshot of an Excel connection to the cube. To the right the dimensions and measures of interest can be selected, and the pivot table to the left displays the result. This is generally a very flexible way do to reports; the user can select exactly the data to be displayed, and what dimensions to group the data by. As seen in the screenshot, drill-down and roll-up operations are done by clicking on the small boxes with plus and minus signs. Of course, it is also possible to create diagrams of the data in traditional Excel fashion.

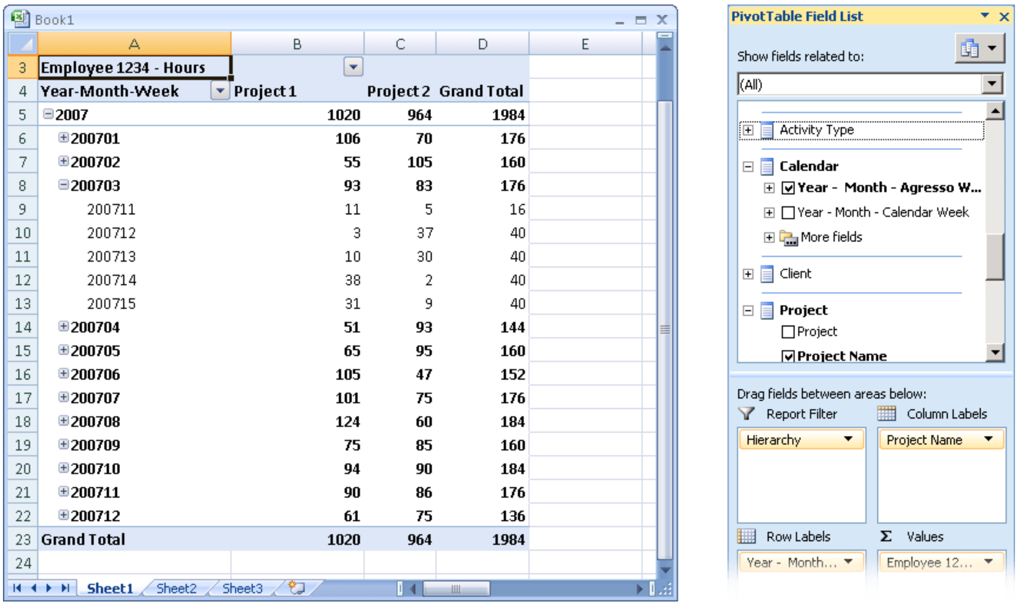

Figure 4.3: Screenshot of reporting using Excel.

Another option investigated was Reporting Services which offers similar possibilities. Reports are created that contains pivot tables and diagrams of data selected by the author of the report, and here the drill-down and roll-up operations can be performed too. Reports can also be linked together via hyperlinks that directs the end user to another report which gives the reporting approach additional potential. The reports are updated on a report server from where the end user can download it. However, the Excel approach is more flexible since it enables the end user to create custom reports.

Since the goal was to make information accessible on an individual level, there was no need to have the possibility to browse the whole cube. On the contrary, that would make the user interface more complicated (the cube has to be browsed to find the current user) and it also introduces security issues; only the sub cube that contains information about the connected user should be accessible. These were the main reasons why Excel and Reporting Services were excluded from the list of options.

Instead, a prototype application were developed in .NET that visualizes the data read from the cube, see Figure 4.4 for a screenshot. This application requires the name and employee number of the user currently logged in from Windows Active Directory. Filtering the cube data by this information, data is received from the cube and visualized as a graph. The user can specify which KPIs to display and the time period for which to display the data.

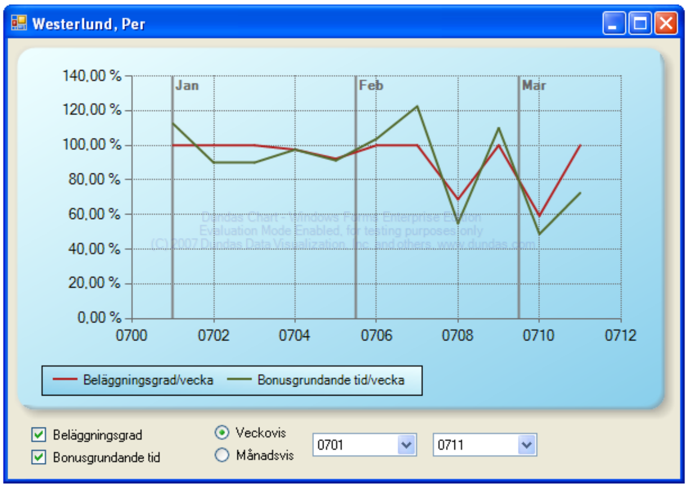

Figure 4.4: Screenshot of the .NET application developed to present the data.

4.3 Performance

In order to measure and compare the performance, test data were generated and loaded into the data warehouse and the cube. Query performance was tested for MDX queries against the cube when it was configured for ROLAP and MOLAP respectively. Performance was also measured for SQL queries run against the data warehouse directly, without any optimizing techniques to see what difference it makes. A set of five test queries were written in MDX, and a set of five corresponding SQL queries that gives the same information were written for the data warehouse. The MDX and the SQL queries didn't produce the exact same result depending on architecture differences. MDX is by nature multidimensional and the result set can be compared to a spreadsheet with column and row labels, e.g. hours worked on rows and resource id on columns. The SQL queries, at the other hand, produce tables which only has column labels. However, the information that can be interpreted from the data is the same.

The actual query performance were measured with the *SQL Server Profiler*, a tool used to log all events that occur during any operation on the database, with timestamp and duration. The test was made by doing three implementations: (a) a test data generator; (b) a test executioner that loads the data warehouse and the cubes, and then executes the test queries; and finally (c) a test result parser for the trace log files that were produced by the profiler.

When implementing the test data generator, the intention was to make the test data look like real data, but without too much effort put on the small details. Given a number of employees, a data file of the same format as the authentic data sheets were to be generated. Every employee in the test data set work 40 hours a week, every week of the year, distributed over randomly selected activity types and projects.

The first version of the generator wrote Excel files as output because the source data were provided in this format. When the implementation was done and tested, the problem with this approach was soon clear; when generating large sets of data, Excel couldn't handle it. Due to Microsoft's specification[1], Excel 2003 has a limit of 65 536 rows per spreadsheet. It seemed like the next version, Excel 2007, could handle about 1 000 000 rows, which would possibly be sufficient for the performance tests. However, this was not an option since Integration Services 2005 - which was the tool used for importing Excel data into the database - doesn't support the 2007 file format.

Another solution to the problem could be storing several sheets in one file, but this idea didn't seem very flexible and was not tested. Instead, the generator was rewritten to output plain text files. When the implementation was ready, 9 data sets of different sizes were generated.

The test executioner used the same script that was written to import the real data. Every data set was imported into the data warehouse one by one, and for each data set, both a MOLAP and a ROLAP cube was loaded with the data. After the data set was loaded into the cubes, each MDX query was executed 10 times on each cube and the SQL queries were executed 10 times each against the data warehouse.

Finally the test result parser was given the trace log files that were produced by the profiler. In these log files, timestamps and durations of every operation was written. The log files were parsed and the average query duration for each combination of cube, query and data set (or query and data set for the non-cube tests) was calculated.

Details on the performance test such as the exact queries, the sizes of the data files, average durations, etc. can be found in Appendix A.

[1]See *http://office.microsoft.com/en-us/excel/HP051992911033.aspx* for details

5

Conclusions

This chapter presents the conclusions reached while working on the project. The goals of the practical part, and how they were met in the resulting implementation is discussed as well as the conclusions from the more theoretical part.

The need for multidimensional data analysis as a support for business decisions has emerged during the last decades. The well accepted OLTP technology was not designed for these tasks and therefore, the OLAP technology was developed as a solution. But do we really need a technology like OLAP? Even though OLTP wasn't designed for these tasks, isn't it possible to exploit the all-purpose relational database technology? The short answer is that it is quite possible to implement data cubes directly in a relational database engine without modifications, but it has its limitations.

5.1 Why OLAP?

One way to do cubing without OLAP is of course to write SQL queries that extracts the result sets that is desired, and that contains the same data that would result from equivalent OLAP operations. This was done as a comparison during the performance tests of the project, see Appendix A for details. There are however some major drawbacks with this approach.

First of all, the performance would be unacceptable when the database is very large with many relations involved, for example a database of a complex organisation that hold many years of historical data. The joins and aggregations required would slow down query response time enormously, and this can clearly be seen in the test results. However, the tests were performed with realtime calculations, and the query response time could be optimized with materialized views or maybe some explicit indexing. At the other hand, the queries were executed against the star schema shaped data warehouse, which is filled with redundant data to minimize the number of joins required. To do the same queries against a normalized data source would deteriorate performance. Anyhow, since the OLAP tools are specifically developed for these kind of queries, they are of course optimized for short query response times. Some of these optimizations take advantage of the read-mostly nature of OLAP models and can hardly be found in an all-purpose relational database engine.

Second, the reporting would be limited. A great advantage of OLAP tools is that the user view is multidimensional and the reporting is very flexible. The cube operator proposed by Gray et al. is helpful to avoid the complex queries required to do all necessary aggregations, but it is still presented in relational form which in this context is a limitation. OLAP is very flexible with both column and row labels, and even if it is not so common, reporting in more than two dimensions is fully possible. Adding to this the drill-down and roll-up operations makes these kind of tools superior to relational databases when it comes to analyzing data.

Of course, the need for multidimensional data analysis for a smaller organisation with a limited database may not require all the extensive capacity of OLAP tools, which often are expensive even though there are open source alternatives for BI solutions as well[1]. During the performance test, the largest data set tested was a 2 GB raw data file imported to the data warehouse. This file contains weekly hour reports for 20 000 employees during one year, about 9 000 000 rows of data. This is quite a lot of data, and the slowest query executed had a duration of about 25 seconds.

As an intellectual experiment we assume that a small to medium sized, local clothing store wants to analyze their sales, and they sell about 1 product every minute in average as long as they are not closed. This will result in 480 transactions a day, and if we assume there are 50 weeks in a year, compensating for holidays, and each week containing 6 working days, that will sum up to about 150 000 transactions during one year. This is not even close to the test data set mentioned above, and thus querying this data set without any OLAP tools or specific optimizations would not be a problem.

As stated above, OLAP is a requirement for large scale data analysis. Most recent research papers concerning data cubes and data warehouses seem to agree that ROLAP is the most suitable architecture for OLAP, and basically that was what Codd et al. stated in their white paper back in 1993. The benefit that MOLAP brings is better performance, but the question is if it is worth reduced flexibility. ROLAP is more flexible than MOLAP in the sense that not all cells in the data cube has to be pre-aggregated, this is also an important aspect of the scalability of OLAP applications. When the number of dimensions increases the data explosion problem grows, primarily with MOLAP, and at some point it becomes overwhelming.

However, there are researchers who still advocate MOLAP as the superior technology and they are supported by some vendors, one of them being Microsoft, offering the user a choice between the two architectures in Analysis Services. Even if both options are available, the MOLAP is strongly recommended because of its increased performance [9].

The performance tests conducted during this project indicate that MOLAP indeed has a better performance with Analysis Services, which is quite expected since aggregating data is a demanding task. However, I did expect that the difference between MOLAP and ROLAP would be greater. It would be interesting to do tests on more complex models of larger scale, especially models with many dimensions. If the performance gain coming from using MOLAP isn't very large, then there is really no reason to chose MOLAP. It also seems that there are many solutions for optimizing ROLAP performance, and by using these, ROLAP can probably compete with MOLAP when it comes to performance issues.

5.2 Restrictions and Limitations

The implementation was done in two different versions; one as an ASP.NET web page, and one as a .NET Windows application. The Windows application seemed to work

[1] Pentaho BI Suite is an example of a completely free, open source solution for BI, *http://www.pentaho.com/*

properly, but the ASP.NET web page could only be run in the development environment since a problem with the Active Directory (AD) emerged. When the application was launched on the intranet, the communication with the AD didn't work and the application couldn't receive the employee number needed to filter the data from Analysis Services. One of the project goals was to make the information available on Sogeti's intranet, but since there wasn't enough time to fix the above problem, only the Windows application was made available for download on the intranet.

In this implementation, the .NET application has full access to the data cube without restrictions, and this could be considered a potential security risk. It is possible to set security in the cube, but it would be quite complicated because there are lots of users that got to have access to the cube, and the information about them is stored in the Windows Active Directory (AD). Setting the security in the cube for each user would require some kind of connection between the AD and the cube, which most likely is possible, but there was not enough time to investigate this option and therefore the whole cube was made accessible with just one login, giving the responsibility of managing security to the application prototype.

5.3 Future work

During this project I didn't have direct access to Agresso, instead I received excerpts of the database stored as Excel spreadsheets. These spreadsheets were manually extracted from Agresso and then sent by e-mail. With these prerequisites, an automation of the updating process is complicated to implement. It is probably possible to trigger the loading process when an e-mail with new data arrives, but it requires the Excel spreadsheet to have an exact structure in order to import the data correctly, and the email has to be sent regularly when new data is available. Another option is that somebody manually starts the loading process every month, and makes sure that the data structure is correct. This is how the loading process was done during the development of the prototype. The best option for a running system would however be direct access to the data source.

The management of Sogeti Sverige AB has shown interest in the project, and by the time my work was finished, a large scale project was initiated at a national level of the company to implement similar functionality on the intranet.

References

[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 506–521, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[2] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[3] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970.

[4] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP to User-Analysts: An IT Mandate. 1993.

[5] D. Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[6] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching Multidimensional Queries Using Chunks. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 259–270, New York, NY, USA, 1998. ACM.

[7] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2004.

[8] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[9] R. Jacobson and S. Misner. *SQL Server 2005 Analysis Services Step by Step*. Microsoft Press, Washington, 2006.

[10] MicroStrategy. The Case for Relational OLAP. 1995 . https://www.microstrategy.com/Solutions/5Styles/olap_analysis.asp.

[11] S. Misner. *SQL Server 2005 Reporting Services Step by Step*. Microsoft Press, Washington, 2006.

[12] P. E. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In Joan Peckham, editor, *SIGMOD Conference*, pages 38–49. ACM Press, 1997.

[13] P. Turley, J. Kasprzak, S. Cameron, S. Iizuka, and P. Guzman. *SQL Server 2005 Integration Services Step by Step*. Microsoft Press, Washington, 2007.

[14] K. Wu, E. J. Otoo, and A. Shoshani. A Performance Comparison of Bitmap Indexes. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pages 559–561, New York, NY, USA, 2001. ACM.

[15] K. Wu, E. J. Otoo, and A. Shoshani. On the Performance of Bitmap Indices for High Cardinality Attributes. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 24–35. Morgan Kaufmann, 2004.

Appendix A

Performance Test Details

The performance tests were made with generated test data. 9 test data file were generated based on a given number of employees, and for each employee, hour data were generated for each working day during one year. The data files were imported separately into a data warehouse, and for each data set a MOLAP and a ROLAP cube were loaded.

To do the test, five different MDX queries and five corresponding SQL queries were written. The SQL queries return the same result as the MDX queries, but the result is not exactly the same due to fundamental differences in the database models; the MDX queries on the cubes returns a two-dimensional result set with both row and column labels while the SQL queries on the data warehouse returns ordinary relations with only column labels. However, the meaning of the resulting data is the same.

For every data set, the five MDX queries were run 10 times each on the two cubes, and the SQL queries were run 10 times on the data warehouse. SQL Server Profiler was used to measure the duration of the queries, and the average durations were calculated and are listed below.

The computer used for the performance testing was a virtual server running Windows Server 2003 R2, Standard edition. According to the Windows system, the virtual machine was running on an Intel Pentium 4 2,8 GHz processor with 640 MB of RAM.

Table A.1 describes the 9 data sets that were generated. The number of employees in each data set and the number of rows generated is provided.

Data Set #	**Employees**	**Rows Generated**
1	100	46 042
2	1 000	442 230
3	2 000	888 841
4	5 000	2 218 487
5	7 500	3 342 743
6	10 000	4 468 858
7	13 000	5 811 294
8	16 500	7 351 367
9	20 000	9 268 325

Table A.1: Test data sets used during performance testing

The disk size of each data file and the corresponding databases (MOLAP cube, ROLAP cube and data warehouse) is described in Table A.2. Note that the data stored in the data warehouse is often redundant due to its non-normalized structure. Project names and activity type names are examples of string data that is redundant and thereby occupy much more space than required. This is probably the main reason why the cubes are far more compact than the data warehouse.

Data Set #	Data File	DW	ROLAP Cube	MOLAP Cube
1	10 285	18 560	2 233	2 740
2	98 817	148 544	2 235	5 191
3	198 591	296 021	2 256	9 831
4	495 728	746 581	2 254	21 434
5	746 897	1 106 176	2 275	35 246
6	998 519	1 477 888	2 259	45 511
7	1 298 472	1 921 408	2 302	66 795
8	1 642 247	2 428 864	2 309	89 130
9	2 070 770	3 069 760	2 304	112 842

Table A.2: Sizes in kB of the test data files, the data warehouse (DW) and the data cubes.

Table A.3 and Table A.4 list the query durations measured for the five MDX queries on the ROLAP and the MOLAP cube. The execution of queries 2 and 5 on data sets 5 to 9 (the entries marked with -) gave rise to the following error during execution:

```
Memory error: While attempting to store a string, a string was found that
was larger than the page size selected. The operation cannot be completed.
```

The queries were sent to the cube engine from a .NET application that was developed for testing the performance. The problem seemed to arise when the result set was too large, it couldn't be sent back to the application due to limitations on the page size. The result of this was that the durations from these erroneous queries were a lot shorter than they were expected to be, considering the durations of queries with no errors. The erroneous durations have been removed and are represented by zero in Figure A.1.

Data Set #	Query 1	Query 2	Query 3	Query 4	Query 5
1	541	234	170	363	466
2	1 145	1 024	196	319	3 294
3	1 351	1 926	217	458	5 083
4	2 175	18 375	313	773	24 308
5	1 682	-	562	588	-
6	1 879	-	516	616	-
7	2 417	-	564	874	-
8	2 808	-	712	696	-
9	2 248	-	829	954	-

Table A.3: Average query durations for the ROLAP cube (μs).

The SQL queries were executed against the data warehouse in two modes; cached and not cached. These results are listed in Table A.5 and Table A.6. As can be seen, the problem with large result sets was not an issue when querying the data warehouse.

Figure A.1 visualizes durations for the five MDX queries as well as the five SQL queries. The reason for separating the data series and for putting the MDX and the SQL query durations in different graphs becomes quite clear when looking at the magnitude;

Data Set #	Query 1	Query 2	Query 3	Query 4	Query 5
1	25	31	21	30	122
2	22	456	30	24	1 258
3	91	628	51	33	3 486
4	89	18 125	65	36	21 470
5	92	-	97	35	-
6	119	-	142	34	-
7	134	-	140	89	-
8	113	-	184	30	-
9	265	-	255	38	-

Table A.4: Average query durations for the MOLAP cube (μs).

Data Set #	Query 1	Query 2	Query 3	Query 4	Query 5
1	247 039	170 586	54 348	169 028	227 137
2	575 582	926 258	201 807	519 521	1 395 206
3	1 002 062	1 457 348	225 316	804 587	2 801 574
4	2 200 032	4 981 525	594 383	1 803 914	6 982 024
5	2 390 159	5 158 413	520 625	2 077 508	8 684 045
6	2 714 874	7 180 407	1 017 350	2 038 647	12 259 188
7	2 769 751	9 286 215	858 818	1 976 891	15 485 503
8	3 035 573	10 824 070	1 359 450	2 423 688	21 212 824
9	2 965 990	16 376 624	1 692 932	2 563 666	23 808 536

Table A.5: Average query durations for the data warehouse, not cached (μs).

Data Set #	Query 1	Query 2	Query 3	Query 4	Query 5
1	19 531	69 822	19 742	10 779	109 996
2	13 607	327 378	107 630	7 177	854 689
3	16 327	863 099	68 164	5 501	2 184 572
4	283 171	1 908 062	128 297	19 742	3 590 072
5	216 904	3 308 682	164 555	14 388	6 036 926
6	251 473	4 868 164	236 568	15 805	8 132 516
7	310 104	5 493 966	280 917	21 800	11 884 609
8	288 938	7 544 958	373 537	20 092	15 237 895
9	269 348	9 906 479	405 336	20 542	19 961 023

Table A.6: Average query durations for the data warehouse, cached (μs).

the SQL queries, even when they are cached, are often more than 1000 times slower than the MDX queries. Comparing them in the same diagram would not make any sense.

There are some irregularities in the data series of the ROLAP queries and the data warehouse series that wasn't cached. These irregularities might be explained by the fact that the server used for the tests was a virtual server running on a machine hosting several virtual servers. Thus all servers shared the same hardware, and activity in other servers could affect the test result by slowing down the test server for some of the queries.

Finally, Table A.7 lists the queries used for testing. All queries are listed in MDX as well as SQL, and a short description explains the meaning of the queries.

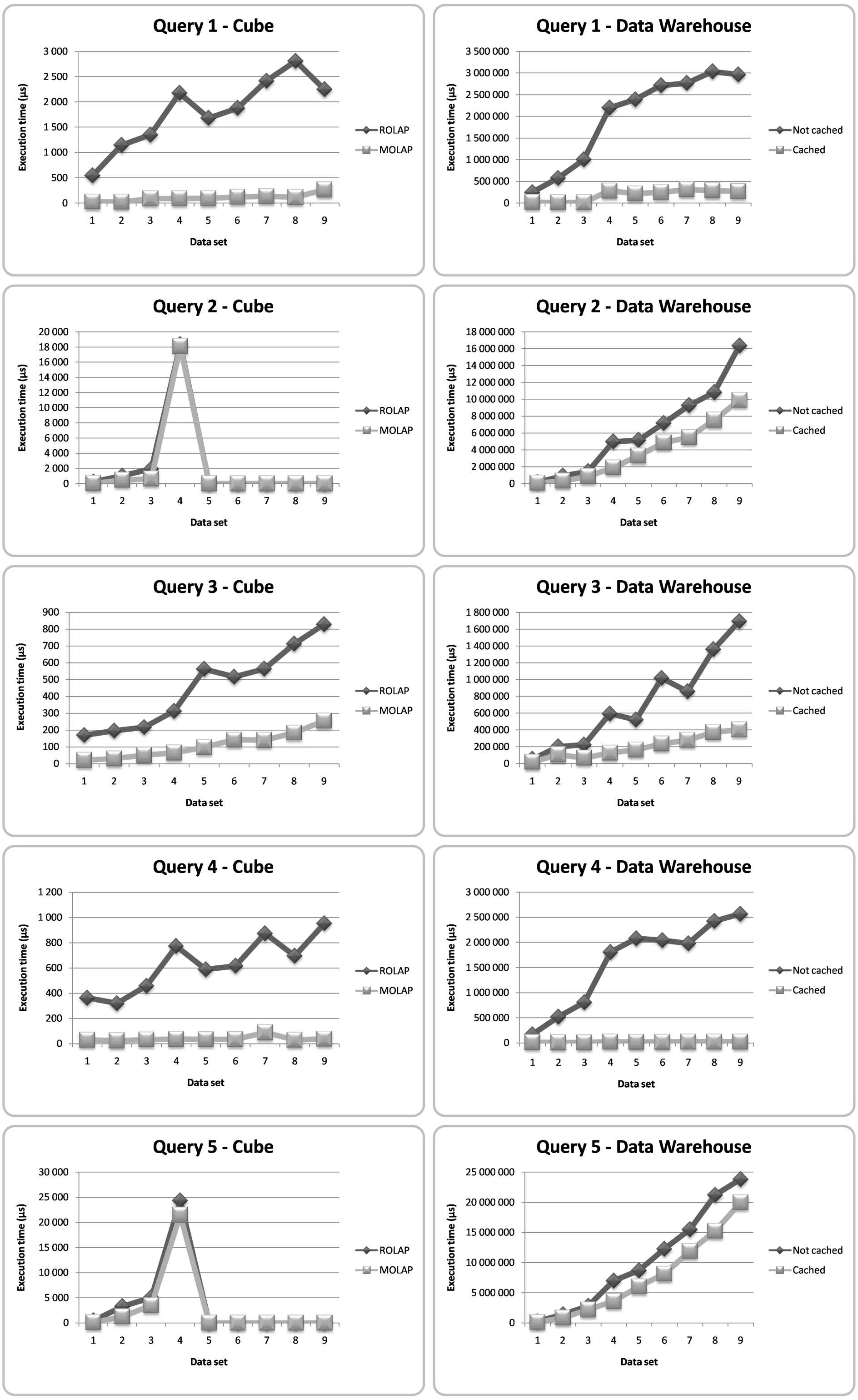

Figure A.1: Diagrams visualizing query durations for both the cube and the data warehouse.

	MDX	SQL	Meaning
4	SELECT {[Resource].[Resource Id].[1]} ON COLUMNS, {[Calendar].[Month].[200701]: [Calendar].[Month].[200712]} ON ROWS FROM [Kobi] WHERE [Bonus];	SELECT WeekTable.MonthId, SUM(BHours) / SUM(WeekTable.WorkingHours) FROM Bonus, WeekTable WHERE Bonus.WeekId = WeekTable.WeekId AND Bonus.ResourceId = 1 AND WeekTable.MonthId >= 200701 AND WeekTable.MonthId < 200801 GROUP BY ResourceId, WeekTable.MonthId;	'For each month during 2007, select bonus quotient for employee with id 1.'
5	SELECT {[Resource].[Resource Id].members} ON COLUMNS, {[Calendar].[Month].[200701]: [Calendar].[Month].[200712]} ON ROWS FROM [Kobi] WHERE [Bonus];	SELECT ResourceId, MonthId, SUM(BHours) / SUM(WeekTable.WorkingHours) FROM Bonus, WeekTable WHERE Bonus.WeekId = WeekTableWeekId GROUP BY ResourceId, MonthId;	'Select bonus quotient for every employee each month during 2007.'

Table A.7: MDX and their corresponding SQL queries used during the performance testing (continued).

	MDX	SQL	Meaning
1	SELECT {[Resource].[Resource Id].[1]} ON COLUMNS, {[Calendar].[Calendar Week].[0701]: [Calendar].[Calendar Week].[0752]} ON ROWS FROM [Kobi] WHERE [Urve];	SELECT WeekId, AVG(Urve) FROM (SELECT WeekTable.Week AS WeekId, Urve.Urve FROM Urve, WeekTable WHERE WeekTable.WeekId < 200801 AND WeekTable.WeekId >= 200701 AND ResourceId = 1 AND Urve.WeekId = WeekTable.WeekId) AS Foo GROUP BY WeekId;	'For each week during 2007, select Urve for the employee with id 1.'
2	SELECT {[Resource].[Resource Id].members} ON COLUMNS, {[Calendar].[Month].[200701]} ON ROWS FROM [Kobi] WHERE [Urve];	SELECT ResourceId, AVG(Urve) AS Urve FROM (SELECT ResourceId, Urve, WeekTable.WeekId FROM Urve, WeekTable WHERE WeekTable.WeekId = Urve.WeekId AND WeekTable.MonthId = 200701) AS Foo GROUP BY ResourceId;	'Select Urve for all employees during January 2007.'
3	SELECT {[Activity Type].[Activity Type].members} ON COLUMNS, {[Calendar].[Month].[200703]} ON ROWS FROM [Kobi] WHERE [Measures].[Hours];	SELECT ActivityType.ActivityTypeId, SUM(Hours) FROM HoursPerWeek, ActivityType, WeekTable WHERE HoursPerWeek.ActivityTypeId = ActivityType.ActivityTypeId AND WeekTable.WeekId = HoursPerWeek.WeekId AND WeekTable.MonthId = 200703 GROUP BY ActivityType.ActivityTypeId;	'Select the total number of hours for each activity type during March 2007.'

Table A.7: MDX and their corresponding SQL queries used during the performance testing.

Appendix B

Glossary

1NF
Abbreviation of *First Normal Form.*

2NF
Abbreviation of *Second Normal Form.*

3NF
Abbreviation of *Third Normal Form.*

– A –

Aggregation
Applying an operator, for example *sum* or *average*, on all values in a data set is called aggregating. Aggregations are often applied on a data set in combination with the *group by* operator.

Agresso
Agresso is a commercial business and economy system used by Sogeti.

Analysis Services
A tool for building data cubes and analysing data, a component of Microsoft SQL Server.

– B –

BI
Abbreviation of *Business Intelligence.* Term referring to a technology used to gather and analyze data in order to find what factors are affecting a business.

– C –

Candidate Key
Any *key* of a relational schema.

Cardinality
The cardinality of an *attribute* is the number of values in the *domain* associated with that specific attribute. For example, a relation with the attribute gender would have a cardinality of 2 since possible values are male and female.

Composite Key
A *key* that consists of multiple attributes.

– D –

Data Cube
A database model used for analysing data multidimensionally.

Data Mining
Term describing tools used to find trends and patterns in large data sets.

Data Warehouse
A central data storage used as a base for building *data cubes* and do *data mining.* The data in a data warehouse is often collected from multiple, heterogeneous data sources.

DBMS
Abbreviation of *DataBase Management System.* Software that manages databases and transactions.

Dimension Hierarchy
A division in *granularity* levels of a *dimension.* For example, a typical time dimension hierarchy contains years, months and days.

Dimension Table
A database table that contains dimension data, i.e. data that is used to group *measures.*

Domain
The set of values that can be applied to a certain *attribute.*

Drill-down
Operation on a data cube that traverses a *dimension hierarchy* down towards a finer *granularity.*

DSS
Abbreviation of *Decision Support System.* A computer system designed to help making decisions based on given facts.

– E –

ETL
Abbreviation of *Extract, Transform and Load.* Term describing the process of collecting data from multiple data sources and load it into a *data warehouse.*

– F –

Fact Table
A database table that contains *measures* and references to *dimension tables.*

Foreign key
A *foreign key* in a relation is a set of attributes that references a primary key in another relation; that is, the values of the foreign key attributes must be found in the referenced table.

– G –

Granularity
Level of detail of data in the *data cube.* The data in the *data warehouse* determines the finest level of granularity of the cube, *aggregations* on this data creates higher levels of granularity.

– I –

Integration Services
Tool used to schedule data transfer tasks, for example importing data into a data warehouse. This tool is a component of Microsoft SQL Server.

– K –

Key
A minimal *super key*, meaning that if one *attribute* is removed, the set is no longer a super key.

KPI
Abbreviation of *Key Performance Indicator*. A figure to measure efficiency in a business, helpful when making business decisions.

– M –

MDX
Abbreviation of *Multidimensional Expression*. Query language used for querying data cubes. Syntactically similar to *SQL*.

Measure
A numerical value in a *data cube* that can be used in *aggregations*.

MOLAP
Abbreviation of *Multidimensional Online Analytical Processing*. *OLAP* architecture that is based on multidimensional matrices.

– N –

Named Query
Database views are called Named Queries in Microsoft Analysis Services.

Normalization
A concept for designing databases. A normalized database has a set of rules, or constraints, that determine how data can be inserted, updated or deleted.

– O –

OLAP
Abbreviation of *Online Analytical Processing*. Term describing tools for storing and analysing data multidimensionally.

OLTP
Abbreviation of *Online Transaction Processing*. Term describing tools for handling relational databases and transactions.

– P –

Primary Key
The primary key of a relation is one of the relations *keys*, arbitrarily chosen.

Prime Attribute
An *attribute* of a relation schema that is member of some *candidate key* of the relation.

– Q –

Query
A text string sent to a *DBMS* in order to modify or read from a database, for example update, receive or insert data.

– R –

RDBMS
Abbreviation of *Relational DataBase Management System*. Software that manages relational databases and transactions.

ROLAP
Abbreviation of *Relational Online Analytical Processing*. *OLAP* architecture that is based on a relational database.

Roll-up
Operation on a data cube that traverses a *dimension hierarchy* up towards a coarser *granularity*.

– S –

Secondary Key
All *keys* in a relation that is not chosen to be the *primary key*.

Snowflake Schema
A way to structure *data warehouses*. The snowflake schema is a *star schema* that has been *normalized*.

Star Schema
A way to structure *data warehouses* with a *fact table* in the middle, referencing surrounding *dimension tables*.

Super Key
A set of *attributes* that functionally determines all other attributes in a *relation*.

SQL
Abbreviation of *Structured Query Language*. Language used for querying relational databases.

– T –

Tuple
An ordered list of values, which are elements of a specific *domain*.

– U –

URVE
Abbreviation of *Utilisation Rate Vacation Excluded*. One of the *KPIs* used during the project.