Department of Computer Science
Aalborg University

Multi-Dimensional Classification

Data Mining using Data Cubes

Master Thesis
by
Peter Jensen

Aalborg Universitet · Fredrik Bajers Vej 7E · 9220 Aalborg Øst · Tlf. 96 35 80 80 · Telefax 98 15 98 89

Department of Computer Science

Aalborg University

Title:
Multi-Dimensional Classification
Data Mining using Data Cubes

Project:
DAT6, Spring 2003

Project Group:
E1−119b

Group Members
Peter Jensen

Supervisor:
Finn Verner Jensen

Number of Copies: 5

Number of Pages: 112

Number of Appendices: 2

Abstract:
This thesis deals with the use of data mining on data warehouse structured data, also known as multi-dimensional data.

The theory regarding data warehouses is investigated with the purpose of understanding the structure of data in these. Then a data set, dealing with the sales of products, and the payments of the customers, is analysed. There are to goals for this analysis, one is to create a multi-dimensional design, the other, and more important, is to get experience in creating such designs, to understand the structure of the designs better.

Then it is tried to analyse the multi-dimensional data using a traditional data mining tool, Clementine. The aim of this analysis is to discover weaknesses in traditional data mining tools when dealing with multi-dimensional data. We then propose a way to analyse multi-dimensional data in general, and we propose changes to decision tree induction algorithms such that they utilise the multi-dimensional structure better.

Finally, we evaluate the proposed way to analyse multi-dimensional data using a prototype of a graphical interface, and we analyse some of the proposed changed to decision tree induction using the data set we have been working with.

Preface

This master thesis is the documentation of the work done by Peter Jensen during the DAT6 semester, spring 2003. The project is written under the research unit *Decision Support Systems* at the Department of Computer Science at Aalborg University.

The work is partly continued from my previous work, which is presented in [Jen01].

This thesis investigates the possibilities and problems in merging data mining and data warehousing.

Throughout the report, references to source material are shown in brackets and refer to the bibliography on page 79. Attributes are typeset with *italic font*, and relational tables are typeset with **bold font**.

The author would like to thank Professor Finn Verner Jensen for his great supervision and inspiration during the project.

Aalborg, 15th of August, 2003.

Peter Jensen

Contents

Introduction

Today the use of data mining is becoming more widespread, one of the reasons is that companies are focusing more on their data and the use of this data. The analysis of this data can be aided by data mining, however, a crucial part of data mining is having the right data, and this data must be of sufficient quality. Thus, as experience has often shown, the major part of data mining projects is the gathering of data and cleaning of data.

Currently, many companies are beginning to create what is known as data warehouses, which is basically centralised storage of all data related to a company. The data in data warehouses is not simply stored as the pieces of data, it was when it was spread out over the entire company. Instead, it is integrated in the data warehouse, that is, data from multiple sources is cleaned and defined using a common view of the entire company organisation. The effect of this approach, is data of high quality and a description of the available data, since it is impossible to define a common view of all data relating to the company without such a description. Moreover, the storage of this data is done in such a way, that the performance when analysing it is improved compared to ordinary storage methods. This improvement is achieved by calculating summary information.

We want to investigate the connection between data warehousing and data mining, since it seems natural to use the data of better quality, which is stored in the data warehouse. Furthermore, the data has extra structural information concerning the domain it represents, thus we would like to investigate how this added information can be used to improve data mining.

A significant part of this investigation is getting an understanding of what a data warehouse is, and more importantly, how one models data in them. This is very important to understand the structure of data, since it is more complex than the traditionally view, cases consisting of attributes. Thus, we use a real-world data set relating to a small "business" and try modeling this data in a data warehouse. We then want to proceed by testing how a standard data mining tool copes with the analysis of data warehouse structured data. Based on this test we hope to find some limitations and possible solutions to them. Finally, we want to investigate how a specific data mining algorithm can take advantage of the data warehouse.

The reader is assumed to be familiar with the data mining conceps, that are described in [Jen01].

Summary

This report investigates the possibilities that lie in using data warehouses for data mining. The structure of data warehouses is examined, and one of the current data mining tools is tested on these, with the aim of discovering problems and areas which can be improved. Based on this we propose a view on how multi-dimensional data mining should be handled, and we propose improvements to decision tree induction algorithms. An application is created to test the proposed user interface and one of the improvements to decision tree induction.

In chapter 1 basic database technology is introduced. This consists of the relational data model, keys, joins, and aggregate functions.

In chapter 2 data warehousing is introduced. First the purpose of data warehouses is described, then the structure of data warehouses is examined. OLAP is introduced and the various storage models are discussed.

In chapter 3 a relational database describing a small business, with regard to sales of products and customer payments, is analysed and transformed into a small data warehouse to gain more knowledge about how data warehouses are constructed.

In chapter 4 we examine related work and describe the problems, we have discovered when trying to use Clementine for data mining on the multi-dimensional data. We then proceed to describe concept hierarchies and propose a view on how data mining should be done in multi-dimensional databases.

In chapter 5 we first analyse the decision tree induction algorithm, and based on this analysis we find the general points at which it can be modified. Using this knowledge we propose several modifications, which we see as possible improvements that can be achieved by using the extra structure of the dimensional model.

In chapter 6 we describe the application we have created, and the experience we have had during testing of it. We also present test results with regard to the performance of one of the improvements to the decision tree induction algorithm.

Finally, we conclude on the project and suggest future work in chapter 7.

Chapter 1

Fundamental Relational Database Technology

In this chapter we introduce the concepts from database technology, which are necessary to understand remaining chapters. If the reader is familiar with relational database concepts, this chapter can easily be skipped. The chapter is based on [SKS02], where it covers Entity-Relationship modelling, relational algebra, and SQL in a practical manner, that is, we will not introduce these separately, instead the topics within each of these areas have been combined.

We first introduced the most basic definitions concerning databases and their usage, then we introduce keys, as a means of identifying a row of data. Next we consider how data is organised in the database and how this affects query performance. Finally, we introduce joins, as a way of combining several entities inside the database, and a special type of functions, called aggregate functions, which are used to summarise data.

1.1 Basic Definitions

A *database* is a collection of interrelated data. A *database management system* (DBMS) is software that manages one or more databases. There exists different kinds of approaches to organising and managing a database, however, we will only deal with the *relational database*, in which data is organised as a collection of tables.

A *table* consists of *attributes* and *rows*, also called variables and tuples. In Table 1.1 an example can be seen, where *ID, name, country, postal code, city* and *employed* are the attributes, with the rows of data listed below.

ID	name	initials	country	postal code	city	employed	tax
3	John Dove	JD	England	LE3 1TZ	Leicester	True	33.3
7	Alice Jensen	AJ	Denmark	9000	Aalborg	False	NULL
19	Jan Hansen	JH	Denmark	9000	Aalborg	True	61.1

Table 1.1: Example of a table.

attribute	data type
ID	integer
name	char(30)
initials	char(5)
country	char(30)
postal code	char(10)
city	char(30)
employed	boolean
tax	float

Table 1.2: Schema for table in Table 1.1.

In a relational database, each table has a schema associated with it, which describes properties of the table. The most important property, is the data type of each attribute, other properties which, will be described later, are various kinds of keys and indices, restrictions on the values an attribute can attain, restrictions on combinations of attribute values, and so forth. See Table 1.2 for an example of a schema.

The available data types depends on which DBMS is used, however, a common subset of data types is defined by the SQL standard. The most important of these types are:

char(n) or character(n): Fixed-length character string of length n.

varchar(n) or character varying(n): A variable-length character string of maximal length n.

int or integer A finite subset of the integers, the minimal and maximal numbers that can be represented depends on DBMS. However, a 32-bit representation is guaranteed, which gives the range $-2^{31} - 1$ to 2^{31}.

float(n): A floating-point number with a precision of at least n digits. Additionally, most DMBSs support *float* (stored in 32 bits) and *double* (stored in 64 bits) floating-point numbers, with the same precision as the same data types in the C programming language, although, these can have different names. In Oracle they are called float (32 bit) and double (64 bit), while in Microsoft SQL Server they are called real (32 bit) and float (64 bit).

date: A calendar date containing year, month, and day of the month.

time: The time of day, with at least hour, minute, and second. Sometimes finer granularity is available, as well as time zone information.

datetime: A combination of date and time.

Sometimes, it is advantageous to include attributes in a table, which are calculated using other attributes within the same row, these are called *derived attributes*. For instance, in the previous example, tax was stored as a percentage, if we instead wanted it as a number, we could define a new attribute *taxnum* as

$$taxnum \overset{def}{=} \frac{tax}{100}.$$

Some DBMSs support derived attributes directly, that is, one can specify a formula for an attribute, then this attribute is calculated when a row has been fetched. Other DBMSs does not support derived attributes directly, however, these support the use of automatic functions, which are run every time a row is inserted or modified. Thus a derived attribute can be stored as a normal attribute, and letting the DBMS handle the calculation of it.

Furthermore, a special value called *NULL* is defined, which corresponds to an unknown value. When dealing with data containing NULLs one must be very careful with the queries that are used, since these unknown values easily cause unwanted effects. For instance, given two boolean attributes a and b, consider the expression $a \vee (b \wedge NULL)$. If a is true then the expression evaluates to true, since the result of $b \wedge NULL$ does not influence the result. If both a and b are false, then it evaluates to false, since the NULL value does not influence the result. However, if a is false and b is true, then the result depends on the NULL value, which is unknown, so the result is NULL. since the NULL in the expression is an unknown value that can be either true or false. Likewise a comparison NULL = NULL, evaluates to NULL, since both NULLs represent some unknown value, and these unknown values are not necessarily equal.

1.2 Keys

In this section we introduce superkeys and candidate keys with the aim of defining primary and foreign keys.

In general keys are used to describe a combination of attributes within a table, which uniquely identifies a row. These keys will later be crucial when dealing with indices.

Let $A_1, A_2, \ldots, A_n$ be the attributes of a table. Then a subset of these attributes, $B_1, B_2, \ldots, B_m$ is called a *superkey* if they uniquely determine a row in the table, that is, no two rows can exist in the table with the exact same values for $B_1, B_2, \ldots, B_m$. In Table 1.1 the attributes *ID, name, country* form a superkey (since it is assumed that no two rows have the same *ID* value. However, it does not seem sensible to use this key when the attribute *ID* alone identifies a row uniquely. Thus, a *candidate key* is defined as a minimal superkey. In the same example, the attribute *ID* is a candidate key, furthermore, the attribute *initials* might be a candidate key if they are assigned uniquely to each person and a person only occurs once in the table.

A *primary key* is defined as a candidate key, which is chosen by the database designer as the primary means of identifying a row. Only one primary key can be assigned to a table, however, any number of candidate keys can be assigned (to ensure that data conforms to the uniqueness restriction given by a candidate key).

In a database relationships between tables are specified by using the primary key of a table as attributes in another table. If we consider the table in Table 1.1 again, then it would be possible to split this single table into two tables, one containing data related to a person (Table 1.3), and the other containing data related to postal codes and city names (Table 1.4). The primary key for the person table is still the attribute *ID*, while the primary key chosen for the city table consists of the attributes *country* and *postal code*. By using the primary key of the city names table in the person table, a relationship is formed, such that when one wants the city name related to a person, it can looked up in the city names table.

In a table, a *foreign key* is a set of attributes, which form a primary key in another table. For

ID	name	initials	country	postal code	employed	tax
3	John Dove	JD	England	LE3 1TZ	True	33.3
7	Alice Jensen	AJ	Denmark	9000	False	NULL
19	Jan Hansen	JH	Denmark	9000	True	61.1

Table 1.3: Person table.

country	postal code	city
England	LE3 1TZ	Leicester
Denmark	9000	Aalborg

Table 1.4: City table.

instance, in Table 1.3 the attributes *country* and *postal code* is a foreign key, since they are the primary key of the city names table. Foreign keys are used in *foreign key constraints*, which pose restrictions on the values the attributes of a foreign key can attain in a row. Basically they are used to ensure that a reference is present in the table being referenced. For example, a foreign key constraint would normally be attached to Table 1.3, which ensures that the *country* and *postal code* values used in the table exist in the city names table. Thus it would not be possible to add a row with *country* = Denmark and *postal code* = 9220 to the table, without adding the appropriate data to the city names table. Likewise it would not be possible to delete data from the city names table, if the *country* and *postal code* values are present in the person table. This is also known as *referential integrity*.

1.3 Data Organisation and Query Performance

In this section we briefly cover how the table data is organised with the aim of describing certain types of query performance indicators. The discussion is very simplified, since we only wish to be able to roughly classify query performance.

Generally the data in a table is stored in a number of *blocks* on a non-volatile medium. A block usually contain a subset of the rows stored in a table, within the block the rows can either be ordered according to some key or they may be stored unordered.

If the data in Table 1.3 is stored unordered in two blocks, and we wish to find the person with *ID* 5, then we are forced to scan the blocks until we find the row with *ID* = 5. This type of access to data is called a *full scan*, and it has the worst performance, provided the query is searching for a subset of rows. On the other hand, if the rows within each block were ordered according to the *ID*, then it would only be necessary to scan the blocks until the queried *ID* was found, and fetch all the rows with this *ID*. However, some additional data is needed to optimise the query even further, since it is not known which block the correct rows are in. This is accomplished with an index, containing a search key to block and row position mapping. A *primary index* (also called a *clustered index*) is an index whose search key defines the ordering of rows within blocks. Suppose a primary index was created for the previous example, with the attribute *ID* as search key. If we want the rows with *ID* being 5, then we simply find the

p.ID	p.name	p.init	p.country	p.code	p.emp	p.tax	c.country	c.code	c.city
3	John Dove	JD	England	LE3 1TZ	True	33.3	England	LE3 1TZ	Leicester
3	John Dove	JD	England	LE3 1TZ	True	33.3	Denmark	9000	Aalborg
7	Alice Jensen	AJ	Denmark	9000	False	NULL	England	LE3 1TZ	Leicester
7	Alice Jensen	AJ	Denmark	9000	False	NULL	Denmark	9000	Aalborg
19	Jan Hansen	JH	Denmark	9000	True	61.1	England	LE3 1TZ	Leicester
19	Jan Hansen	JH	Denmark	9000	True	61.1	Denmark	9000	Aalborg

Table 1.5: Example of a cartesian product of person table (p) and city table (c).

value 5 in the index, and fetch the correct rows in the correct block. However, if we want to find all rows with *postal code* 9000, this index will not help, and we must use a full scan. Thus *secondary indices* can be created, which are the same as primary indices, with the exception that the rows in the blocks are not ordered according to the index.

It should be noted that the primary index does not have to use the primary key as search key, and a table can have secondary indices defined, without having a primary index. The use of a primary index is mainly to improve full scans so the data is organised in the order most commonly used during a full scan.

1.4 Joins

Until now, we have only examined a single table at a time, however, when dealing with data in databases it is virtually always necessary to query more than one table to get the wanted result. If we consider the tables in Tables 1.3 and 1.4, and we wish to retrieve the person information as well as the city name, we either have to retrieve the person information first, and then do a lookup in the city table for each person to find the city name, or we have to let the DBMS combine the two tables and return the result. Clearly, the first solution is tedious and inefficient, which is why *joins* have been introduced to combine information from multiple tables.

The *cartesian product* of two tables, t_1 and t_2, is defined as having all attributes from t_1 and t_2, and it contains every combination of rows from t_1 and t_2. Thus, the cartesian product of the person and city tables is the table shown in Table 1.5. Usually there exists a relationship between the tables, which are being combined, if this is the case then we restrict the resulting rows to the rows that adhere to this relationship. For instance, when combining the person and city tables, we want the *country* and *postal code* in the person table to be equal to the same attributes in the city table. This is also known as a *natural join*, that is, attributes with the same name in each table are required to have the same value in each row.

A more general type of join is the *inner join* which specifies exactly how the restriction is to be made using some predicate, this is often used if the attributes have different names in the tables being joined. Additionally, different types of *outer joins* exist, which deal with including data that does not exist in both tables being joined. If for example the (Denmark, 9000, Aalborg) row did not exist in the city table, an inner join would not return rows referring to (Denmark,9000). This is a very simplified explanation, however, the types of joins are not crucial for understanding the work presented in this report.

1.5 Aggregate Functions

The term *aggregate function* means some function used in a database for summarisation. In general, these functions work on a collection of values and return a single value. Sometimes they are simply referred to as *aggregates*. Examples of these, are count, max, min, and sum, which returns the number of rows, the minimal value, the maximal value and the sum of values for a collection of values, respectively. In practice, one specifies the table (which can be a result of a query), and which attribute to use, then the values of this attribute is passed to the aggregate function for each row in the table. For instance, calculating sum(ID) in Table 1.1, would return the value 29, whereas count(ID) would return the value 3.

At times one does not wish to calculate the aggregate function on an entire table, but instead wish to partition the table and calculate the aggregate function on each partition. Returning to the Table 1.1, if we are interested in getting the number of persons registered for each country present in the table, then we would partition the table by country and apply the count aggregate function on the partitions. This partitioning is commonly referred to as *group by*, due to its syntax in the SQL language[1].

The remaining part of this section is based on [AAD$^+$96]. Aggregate functions can be divided into three categories, depending on how the calculation of the function on a multiset can be distributed across disjoint subsets of this multiset. Let the multiset $v = x_1, x_2, \ldots, x_n$ be the values we wish to calculate the aggregate function, F on. Then divide v into the disjoint subsets $v_1 = x_1, x_2, \ldots, x_{n_1}$, $v_2 = x_{n_1+1}, x_{n_1+2}, \ldots, x_{n_2}$, …, $v_m = x_{n_{m-1}+1}, x_{n_{m-1}+2}, \ldots, x_n$. Then F is said to be *distributive* if there exists a function G, such that

$$F(v) = G(\{F(v_1), F(v_2), \ldots, F(v_m)\})$$

. In other words, if the input values to the aggregate function can be partitioned into disjoint subsets, which can be aggregated seperately, and these then can be combined to the aggregate value of the whole multiset, then the aggregate function is distributive.

Examples of distributive functions are min, max, and sum, the function G used to combine these aggregate functions is the aggregate function itself. For instance, the sum function is first applied to the subsets, and then the sum function is used again on the value sum of each subset. The count aggregate function is also distributive, however, its combining function, G, is the sum function.

An aggregate function that can be obtained by using an algebraic function with a finite amount of parameters, each of which are obtained using a distributive aggregate function, is called an *algebraic* aggregate function. An example of this type of aggregate functions is the average function, which can be obtained by $\frac{sum}{count}$.

An aggregate function which is neither distributive, nor algebraic is called a *holistic* aggregate function. The median function is an example of a holistic aggregate function.

[1]select country, count(*) from persons group by country

Chapter 2

Data Warehousing

In this chapter we introduce the concept of data warehousing, with the focus on dimensional modelling. The chapter is mainly based on [Inm02] and [Kim96], with the use of some summarised information from [HK01].

2.1 The Data Warehouse

In traditional databases the focus has been on processing transactions, that is, it is more concerned with running a business. However, more companies are beginning to see the value of being able to analyse their business. For this purpose the traditional databases are often unsuitable, since they usually do not track changes over time. For instance, when an order has been processed, it may be removed from the database, or when a customer changes address the old address is overwritten, so it is impossible to do proper analysis over time.

To improve the analysis of a business, a new kind of database has been created, the *data warehouse*. The changes are not in how the DBMS process data, but rather in the way data is entered and organised in the database.

The term data warehouse was coined by Bill Inmon in 1990, and he defined it as "A warehouse is a subject-oriented, integrated, time-variant and non-volatile collection of data in support of management's decision making process" [Ree].

This definition has not changed since, and a data warehouse is still defined as above in [Inm02], where the four terms are explained as follows:

subject-oriented refers to data being organised around the major subjects within the company, instead of their application areas. For instance an insurance company might have its traditional database organised by the types of insurances it deals with. Whereas its subject areas would be customer, policy, and claim.

integrated refers to data being integrated from multiple sources. It is the most important aspect of the data warehouse, and also the most time consuming. This is due to data not simply being transferred into the data warehouse when it is received from multiple sources, instead it is integrated, which means that different parts of an organisation has to agree on all terms used in the data being integrated. Additionally, data may reside in many different formats, and different values may refer to the same thing. For instance, the gender of a person may be described as “m/f”, “male/female”, or “0/1”. Thus, a large part of the creation of a data warehouse, is defining attributes and finding ways to integrate attributes coming from different sources.

time-variant refers to all units of data being tagged with a time stamp, or using some other approach to indicate in which time frame a unit is active. For instance, the address of a customer has a time frame in which it is valid, thereby a change of address is only valid from the time it is registered, and older data is not affected by the change.

non-volatile refers to the way data is loaded and accessed in the data warehouse. In a traditional database, individual rows are often changed. However, in the data warehouse, a large amount of data is loaded, and then it is not modified again, it is only accessed for analysis.

In [Kim96] Ralph Kimball states the following goals for a data warehouse:

1. The data warehouse provides access to corporate or organizational data.
2. The data in a data warehouse is consistent.
3. The data in a data warehouse can be separated and combined by means of every possible measure in the business.
4. The data warehouse is not just data, but also a set of tools to query, analyze, and present information.
5. The data warehouse is the place where we publish used data.
6. The quality of the data in the data warehouse is a driver of business reengineering.

The term *data warehousing* is defined as the process of building a data warehouse. One of the main design decisions when building a data warehouse is determining the granularity of data. That is, the highest level of detail which can be queried in the warehouse. For example, if a company registers all their sales transactions, these may be represented using such high granularity that each transaction is represented in the data warehouse, or they may be summarised to each hour of the day and stored at this granularity level in the data warehouse. Which level is chosen depends on the amount of data and the type of analysis that is to be done on the data.

One of the most important parts of the data warehouse is the *metadata repository* (usually referred to simply as metadata), which stores data about all the attributes, their interpretation, and their relationship. The reason for its importance is due to the importance of data integration. By storing detailed information about how an attribute is interpreted, it is easier to get a common view of the data within a company.

It should be noted that, depending on the size of the data warehouse, the analysis itself may not be performed using the data warehouse. Instead it can be performed in *data marts*, which are smaller databases that extract part of a data warehouse. However, we are not going to distinguish between these two types of databases, so in the remainder of the report we will simply refer to the data mart as a data warehouse.

2.2 Dimensional Modelling

A *dimensional model* is represented as an n-dimensional data cube, with a set of dimensions and a central subject, which depends on these dimensions.

If we consider a sales database, which registers every sale in a store, then the sales could be represented in a cube. In this case, sales would be the central subject, and date, product, or other attributes, which the sale depend on would be grouped into dimensions. Related attributes should be part of the same dimension.

2.2.1 Facts and Measures

A *fact* is usually a representation of some event in the domain of the business. To each fact a number of *measures* are attached, these describe some measurable values concerning the fact. For instance, if we are modelling the sales of a company, each sale could be represented as a fact, and the measures, would be the price of the product, the quantity sold, and so forth.

A measure is categorised according to its additivity. An *additive* measure can be added over all dimensions, a *semi-additive* measure can be added over some dimensions, and *non-additive* measure cannot be added over any dimensions. Additionally, an aggregation function is attached to each measure, so the measures are aggregated correctly across dimensions. For instance, the price or quantity of products would be added together with the sum function, whereas a measure storing the average price, would use the average function.

There do exist other types of facts, which deal with so-called "snapshots", that is, they do not model an event, instead they register the current state of the business in some way[PJ01]. For example, when the current inventory of a shop is registered, this would be represented as a snapshot fact. However, these kinds of facts are rare in comparison to event facts, and difficult to model, so we restrict the type of facts we consider to event facts.

2.2.2 Dimensions

A *dimension* consists of entities, which participate in each fact, but whose attributes do not change with each fact. Returning to the sales example, a customer is part of every sale, but the attributes describing the customer do not change with every sale, thus the attributes of the customer are modelled as a dimension, instead of as measures attached to the fact.

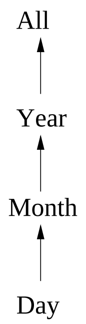

Figure 2.1: Concept hierarchy for simple date dimension.

Within each dimension the attributes are ordered in a hierarchy, which is called a *concept hierarchy*[1]. This hierarchy is created such that it represents generalisation and specialisation of attributes. It always has a unique top node labelled *All* which corresponds to all entities in the dimension, and a unique bottom node, which correspond to the highest degree of detail the dimension registers. For instance, a date dimension consisting of the attributes *Year*, *Month*, and *Day*, would be represented as shown in Figure 2.1. In this representation All corresponds to all data in the dimension, and *Day* to the level of granularity. *Year* is the data grouped by the value of year, *Month* is the data grouped by the value of year and the value of Year. That is, we speak about Januar 2002, and so on, so all the data which correspond to a month in 2002 can be generalised to the year 2002.

Sometimes, there exists attributes, which cannot be generalised or specialised to other attributes, in this case, they are simply placed between the All level and the level of granularity. For instance the attribute *Weekday* could be added to the previous concept hierarchy, without it being part of months or years, as we only want the attribute to represent a weekday over all months and year. That is, we want to be able to analyse all Mondays, and not just the Mondays within a given year. In this case, the attribute would be added as shown in Figure 2.2.

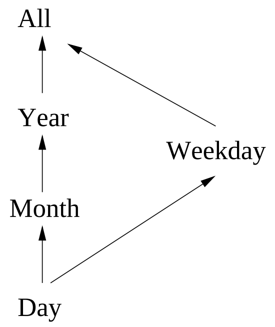

Figure 2.2: Concept hierarchy for more complex date dimension.

2.2.3 The Data Cube

After introducing the previous concepts, we can now show an example of a data cube, consisting of facts and dimensions.

Consider the sales example, with a fact corresponding to every sale, which contains a single measure *number_sold*. Additionally, a customer, product, and time dimension exist. This could be represented as shown in Figure 2.3. In this example, the numbers in each box indicate the number of sold products grouped by customer, product and year. Notice that the year-axis is denoted with "date", since it corresponds to a single attribute from the date dimension, one could have chosen to partition the data by months or some other date-related attribute.

[1]Technically, it should be called a concept lattice, however, most literature use the word hierarchy, so we will also call it a hierarchy.

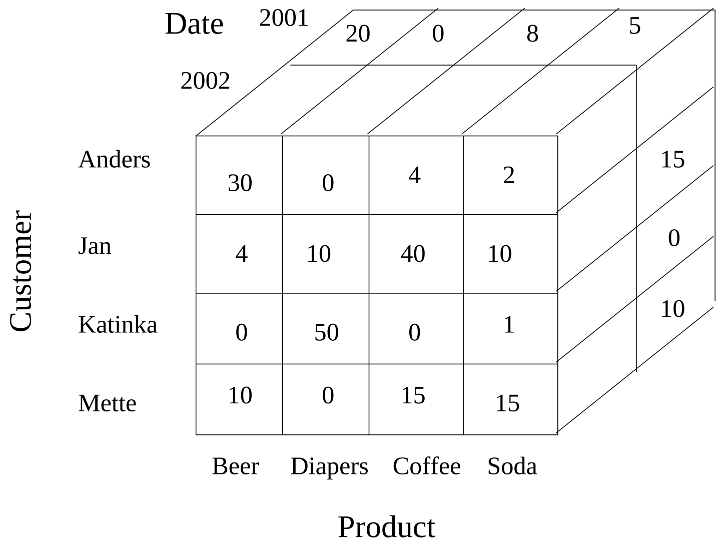

Figure 2.3: Example of a cube.

2.3 OLAP

The cubes previously described, support a special kind of analysis, referred to as *On-Line Analytical Processing (OLAP)*[2]. This kind of analysis uses special operations on the data cubes, which makes it easier to analyse summarised data. First the user is able to select a number of dimensions which is to be displayed in cube or tabular form. Then the following operations exist and can be used on the dimensions:

Roll-up: Generalise the current level of a dimension, that is, climb up the concept hierarchy, for instance from *Month* to *Year*. When the All level is reached, it corresponds to removing the dimension from the cube, since it does not filter the data anymore.

Drill-down: Specialise the current level of a dimension, that is, go down the concept hierarchy, for instance from *Month* to *Day*.

Slice and Dice: The slice operation performs selection on a single dimension, for instance limiting data by "*Year* = 2002", or "*Month* = January or *Month* = August". The dice operation is a generalisation of the slice operation, which performs the selection on two or more dimensions.

Pivot: A visual operation, which rotates the axes of data, to get a different point of view.

Some OLAP systems have additional functionality to combine two cubes, which is called *drill-across*.

These operations are either available in a visual OLAP query tool, or available in a specialised query language.

[2]OLAP is often compared to OLTP, which means On-Line Transactional Processing, which is the traditional way of using databases.

2.4 Storage Models for Cubes

The introduced multi-dimensional model can either be implemented using special data structures, or it can be modelled in a relational database. We first introduce the relational models and then discuss the different types of architectures.

The most common way to model a multi-dimensional model in a relational database is using a *star-schema*. This schema consists of a table for each dimension (the dimension tables), and a table for the facts (the fact table). Then each primary key from the dimension tables are added as foreign keys to the fact table, and no other relationships are added. An example of a star schema is shown in Figure 2.4.

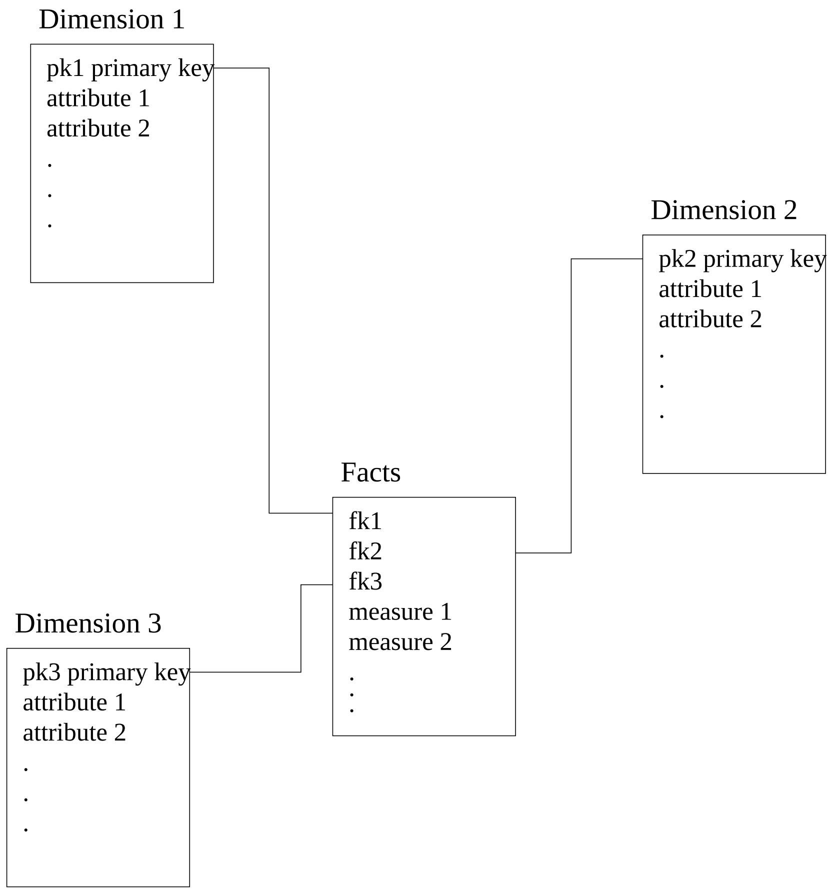

Figure 2.4: Star schema.

The approach used in the star schema results in redundant data in more complex dimensions, so these dimensions can be normalised. That is, some of the dimensions are split into smaller tables each with their own primary key and some attributes, and the dimension table then references these tables. This is known as a *snowflake schema*, an example is shown in Figure 2.5.

Both [Kim96] and [IRBS99] strongly discourage the use of snowflake schemas. The reasoning is that the dimension tables are extremely small compared to the fact table, and browsing

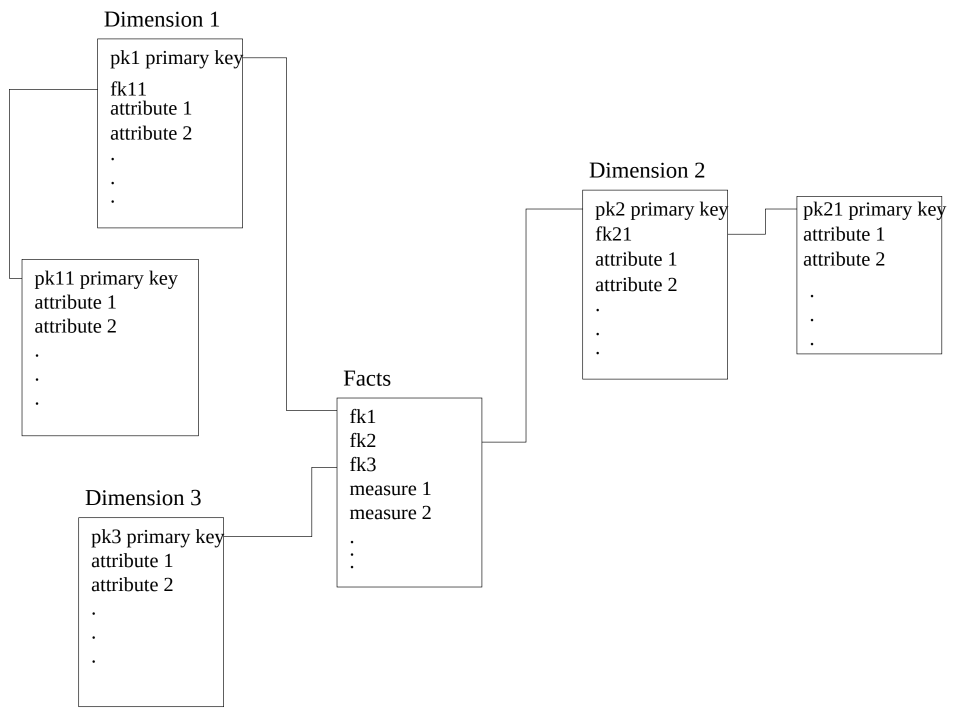

Figure 2.5: Snowflake schema.

of dimensional data is one of the most common activities. So the space savings due to normalisation of the dimension tables are insignificant compared to the performance penalties of performing several extra joins to get the dimensional data. Thus we restrict our future relational representations of multi-dimensional data to star schemas only.

The servers or DBMSs, which store the dimensional data can be categorised into three categories:

Multi-dimensional OLAP (MOLAP): All data is stored in special data structures, specially suited for dimensional data. These utilise special structures for the aggregates, which improves performance, however, they do not scale well with extremely large amounts of data, compared to mature relational DBMSs.

Relational OLAP (ROLAP): All data is stored in a relational database using a star schema, snowflake schema or some other model. These scale very well, due to the maturity of the current DMBSs. However, they are not as efficient in using aggregates, and can have problems with chosing the right join order for dimension tables and fact tables. These problems are being addressed by the major DBMS manufacturers and most DBMSs today can be optimised for star schemas.

Hybrid OLAP (HOLAP): A combination of relational and multi-dimensional databases, where the large amounts of fact data is stored in the relational database, while the aggregates are stored in a multi-dimensional database.

2.5 Pre-computed Aggregates

The main reason behind quick processing of OLAP queries, is the use of *pre-computed aggregates*, that is, aggregate values for certain levels of dimensions and combinations of dimensions are computed when data is loaded into the data warehouse, instead of at the time the query is requested.

It should be noted, that it is usually impossible to compute all combinations of dimensions and dimension levels. With the assumption that only one attribute is included from each dimension in the aggregates, the amount of different dimension/attribute combinations in an n-dimensional cube would be:

$$\prod_{i=1}^{n} L_i$$

where L_i is the number of levels within dimension i, including the All-level (since this corresponds to leaving out the dimension).

Instead the aggregates that improves performance most are computed. This improvement is based both on statistics collected about which type of queries the users are requesting, and based on the reduction in rows that must be fetched to calculate an aggregate. Consider an example, where the fact table consists of 3 years of data, with a granularity of one minute. If we have a date dimension, consisting of the attributes *Year, Month, Day, DayHour,* and *DayMinute*, which are linearly ordered in a concept hierarchy. Then, assuming a fact is recorded every minute during these 3 years, we have the following amount of distinct values for each attribute:

Attribute	Distinct values
Year	3
Month	36
Day	1095
DayHour	26280
DayMin	1576800

If we want to calculate an aggregate value, without any pre-computed aggregates, then it is necessary to calculate it based on the 1.6 million rows every time. Assuming that the aggregate function is distributive. Then, if for instance, an aggregate had been calculated for *DayHour*, this aggregate could also be used to calculate both *Year, Month,* and *Day*, reducing the amount of rows needed for the calculation from 1.6 million to just 26.280.

So, generally, the aggregates are pre-computed for the attributes in the lowest levels of the concept hierarchies. Then, depending on the distinct values present at the other levels, and the number of users using the dimension, aggregates at higher levels can also be pre-computed.

2.6 Loading Data into the Data Warehouse

A large part of maintaining the data warehouse is concerned with loading data into the data warehouse. When the data warehouse is initially created data is loaded into it, however, after this initial load, additional data is loaded at regular intervals.

There are a number of steps, which are performed during each loading of data, these are commonly referred to as *ETL*, which is an acronym for Extract, Transform, and Load. Extract is the process of gathering data from a single source or multiple sources, both databases, files, and other types of external sources. Transform is the transformation, cleaning, and integration of data to the structure specified by the data warehouse. This can be as simple as mapping between two sets of values, to complex calculations involving many sources. Load is the process of storing the transformed data into the data warehouse, calculating aggregates and any other maintenance that must be done to the data warehouse.

Chapter 3

TREO Data

In this chapter we introduce a database, which is being used by a club at the university to register sales and payment status for members.

First we describe the overall purpose of the club, then we examine the quality of data, and clean the data based on the quality observations. Next we analyse which data cubes can be created using the available data, based on this analysis, the design for two cubes is created. Finally we perform the needed data transformations to make the data suitable for the cubes, the most complex part, of this process, is the derivation of historical data, since the database does not include this explicitly.

The original database was received in a Microsoft Access database. This data has been imported into a Microsoft SQL Server database, and all database related operations have been performed with SQL Server and Analysis Services, which is the data cube/OLAP software included with SQL Server. The only exception is the creation of boxplots, which has been done using R 1.6.2. This is a statistical program, which is described, and freely available for download, at http://www.r-project.org.

3.1 Description of the Analysis Domain

At Aalborg University a club exists, called the F-club. This club is for employees and students affiliated with the computer science department and the math department. The club consists of many smaller clubs with a specific purpose. One of these is the TREO, which is responsible for selling food, drinks, and various other products to members of the F-club at low prices.

This is done by having a common refrigerator with the products, which the TREO orders at various distributors. The members of the F-club can then purchase the products they want, however, they are responsible for paying the products themselves.

In the early days this was done by using a “stregsystem”, which was large paper sheets, where members would have a place for their username. When they fetched something from the refrigerator, they would set a mark at their username, thereby accumulating marks. Then when a member had accumulated a certain amount of marks he or she would pay to the TREO for the purchases.

At some point in time it was decided to create an electronic system for handling the purchases.

It would work the same way as mentioned above, but instead of using paper and pen, the member would register electronically which product he or she purchased when fetching a product from the refrigerator. Additionally the payments paid by the members would be registered in the system, so it was possible to see exactly how much money a member owed. Thus it would be possible to enforce limits on how large a debt a member was allowed to have.

The system started online in the fall of 1996, and from this point on a transition to the new system was gradually undertaken. The system has two thresholds for the total debt of the member. A warning threshold, which means a member has to pay some of his or her debt within 14 days, or the member will be blocked, that is, unable to register new purchases. And a blocked threshold, which blocks the member instantly when that threshold is reached. These warning and blocked thresholds were 150,00 dkr and 250,00 dkr, respectively, in the beginning. Later they have been changed to 0,00 dkr and -50,00 dkr, respectively. A special feature of the system, is the so-called "multi-buy", which makes it possible to easily purchase larger quantities of the same product (however, due to software bugs in the TREO software or the web browser running the system, this feature is not that commonly used).

Later, a special arrangement with the computer science department has been made, that is, the department pays the coffee its employees drink. This has been implemented as a special product with cost 0,00 dkr. The amount of free coffee purchased is then handed over to the department, which pays the TREO for the coffee. However, these department payments are not registered in the database.

When a member wants to pay his or her debt, or make sure there is enough money on the account, the member has to pay money to the TREO. This is presently done on fridays between 12:00 and 12:30, however, it is also possible to pay via bank or giro. Sometimes, the nice people in the TREO will even accept money outside this time frame. Whether the payment has always been done on fridays or not, we are not completely sure about, however, it should be possible to infer this from the data. Some irregularities are probably present, since payments are also received during the registration of new members at the beginning of a study year, which does not necessarily occur on a friday.

From the above description of the domain of analysis, a number of observations are important.

- The club is not trying to make a profit

- The purchase is done based on trust, the purchasing member is responsible for paying by himself.

- There exists thresholds for when a member is warned about being locked out from the system until payment is received, and another threshold that immediately blocks the member.

- Some blocked members might take advantage on the trust factor, and accumulate purchases until they have paid and are allowed to use the system again, then purchasing their accumulated purchases

Column	Data Type
employee_type	integer
description	varchar(20)
free_coffee	boolean
Total rows: 3	

Table 3.1: Schema for **Employee_type** table.

employee_type	description	free_coffee
0	Studerende	0
1	Institut 16	1
2	Matematik	0

Table 3.2: Data in **Employee_type** table.

3.2 Overview of Tables and Identification of Primary Keys

The first important task is to get an overview of what data is available, and discover how the different parts of the data is related. This is accomplished by examining the layout (schemas) of the tables in the database, thereby describing the attributes and by studying the actual data to gain insights about the attributes and tables.

During this examination we will also look for attribute combinations, which can be used as primary keys for the tables in the database, since it is lacking these, causing significant performance degradation.

3.2.1 Employee_type Table

The **Employee_type** table, shown in Tables 3.1 and 3.2 contains the categories of employee types who are using the system.

The main purpose of registering employee type is to be able to decide which members are able to receive free coffee. Currently there are three types: Students, employees at the computer science department (registered as "Institut 16" in the data), and employees at the math department. Among these only the employees at the computer science department receives free coffee.

The natural primary key for this table is the attribute *employee_type*.

3.2.2 Members Table

The **Member** table, shown in Table 3.3 is used to register information about the members of the TREO.

user_id identifies a single person as a member of the TREO, thus giving access to buying products if the member is marked as active (which is controlled by the *active* attribute). The

Column	Data Type
user_id	integer
active	boolean
aargang	integer
debt	double
board_debt	double
advance	double
last_warned	integer
first_warning	integer
undos	integer
total_undos	integer
employee	integer
Total rows: 1745	

Table 3.3: Schema for **Members** table.

values for *user_id* are in the set $\{1 \dots 1774\}$, with a total of 1745 values, which are all unique. *active* is either 0 for false or 1 for true.

aargang is the year the member started studying or working at the university. If an employee has studied at the university before becoming an employee, this attribute will be set to the time at which the member started studying. The values for this attribute are in the set $\{0, 1, 1975 \dots 2002\}$, the values from 1975 and up represent a year, however, more analysis must be done for the members having *aargang* 0 or 1.

board_debt, *debt*, and *advance* are numbers pertaining to the payments done by the members. *board_debt* is debt registered in the old non-electronic system, this value is being reduced first when members pay their debt. *debt* is the debt from purchases using the electronic system, whenever a product is purchased this value is increased by the price of the product. *advance* is the amount of money due for the member. When a payment is paid by a member the following procedure is performed

1) Reduce *board_debt* until it is 0 or the entire payment is used

2) Reduce *debt* until it is 0 or the entire payment is used

3) Add the remaining payment to *advance*.

Usually we are only interested in the total amount of money the member owes or has due. Thus we define this as a member's balance, which is defined as follows,

$$balance \overset{def}{=} advance - board\_debt - debt.$$

last_warned and *first_warning* are used for sending warnings to members that owe money, which exceeds a threshold set by the TREO. In the present data *first_warning* is an integer representing a date, the so-called *epoch*, which is the number of seconds since the 1st of January, 1970 at 0:00:00[Gro]. Examination of the data shows that 268 members have a non-zero *first_warning*, the earliest is 1997-02-06 and the latest 2003-01-20. *last_warned* does not seem to be used, since it is 0 for all members.

Column	Data Type
user_id	integer
subscriber_since	datetime
Total rows: 29	

Table 3.4: Schema for **Coffee** table.

Column	Data Type
date	integer
Total rows: 1	

Table 3.5: Schema for **Paid_ansat_kaffe**.

undos and *total_undos* are related to the possibility of cancelling a sale. *undos* is the number of cancellations done since the last payment, *total_undos* is the number of cancellations done for the entire time the member has been registered in the TREO. The reason for this design is that a member is only allowed to cancel 5 purchases since the last payment. However, examination of the data reveals that 293 members have a value larger than 0 for at least one of these attributes, of these members, only 2 members have different values for *undos* and *total_undos*. This does not seem consistent with the assumed design of these attributes, so these two attributes will be combined to one. This is accomplished by retaining *undos* and removing *total_undos* from the database. In the two cases where the attributes differ, *undos* is assigned the maximal value of *undos* and *total_undos*. The attained values of the new attribute are between 0 and 5. It is not registered when the undos were done, however, cross-checking with the sales table shows that 13 of the members with $undos > 0$ have not purchased anything using the electronic system.

employee is an attribute describing the employment status of the member, it is an integer which references the **Employee_type** table.

user_id is chosen as the primary key for this table, since it identifies each member uniquely.

3.2.3 Coffee Table

user_id is a unique identifier for the member registered in the system (identified in the **Members** table), and as such the only candidate for a primary key.

subscriber_since is the date from which the member is registered as receiving free coffee. The data type of *subscriber_since* is datetime, however, only the date part is used, which is evident from all time values being 0:00:00. The values for the dates are between 1999-11-02 and 2002-09-06.

3.2.4 Paid_ansat_kaffe Table

The **Paid_ansat_kaffe** table, shown in Table 3.5, contains the date of the last payment of free coffee by the computer science department. The money transactions for free coffee are

Column	Data Type
user_id	integer
date	integer
amount	double
Total rows: 5046	

Table 3.6: Schema for **Payments** table.

Column	Data Type
product_id	integer
name	varchar(20)
price	double
active	boolean
Total rows: 47	

Table 3.7: Schema for **Products** table.

not part of the system, since free coffee is registered as a product with cost 0,00, and the payments done by the department are not registered. Thus this table is removed from the database, since it does not contribute with any useful information. However, note that it is only the money transactions, and not the actual purchasing transactions, that are missing, so it is still possible to investigate, for instance, how much coffee is purchased by members.

3.2.5 Payments Table

The **Payments** table, shown in Table 3.6, contains all payments, done by members, which have been registered by the system.

user_id is an integer from the **Members** table. *date* is a date in epoch format, and *amount* is the amount of kroner paid to the TREO.

user_id and *date* are chosen as primary key, since these are unique for all payments.

3.2.6 Products Table

This table, shown in Table 3.7, describes all the products offered by the system (now and in the past).

product_id is a unique identifier for the product and an obvious primary key. All integers in the interval $[1, 47]$ are used. *name* is the name of the product which is shown to the members when buying products. *price* is the current price of the product and *active* is a flag to determine whether the product is currently being sold or not.

Examination of the products reveals a product named “Fake” which is an entry that is not being used and never has been, so it is removed from the **Product** table.

Column	Data Type
product_id	integer
price	double
date_start	integer
Total rows: 226	

Table 3.8: Schema for **Prices** table.

product_id	date_start	price
11	1996-11-14 11:39:29	7,00
11	1998-09-23 15:00:11	8,00
11	2001-05-09 11:31:44	9,00
11	2002-11-15 13:20:35	10,50
11	2002-11-15 13:23:39	10,50
11	2002-11-15 13:24:10	10,50
11	2002-11-15 14:01:20	10,50

Table 3.9: Example of redundant price changes in the **Prices** table.

3.2.7 Prices Table

The **Prices** table, shown in Table 3.8, contains historical data for the prices of products sold by the TREO. *product_id* is an integer from the **Products** table, *price* is the price of the product. *date_start* is the date from which the price is active, this date is stored in epoch format. Investigation of the values shows that the table contains redundant information, due to it containing new prices that are the same as the old prices. In the example in Table 3.9 the last four changes to a price of 10,50 can be removed. In general it is possible to order the price changes by date for each product and then only keep the first price change if several equal price changes are detected.

product_id and *date_start* are chosen as primary key, since they are unique for all rows.

Column	Data Type
user_id	integer
product_id	integer
date	integer
price	double
paid_for	boolean
Total rows: 170467	

Table 3.10: Schema for **Sales** table.

3.2.8 Sales Table

The **Sales** table, shown in Table 3.10, describes a single sale of a product. *user_id* is a reference to the member purchasing the product, *product_id* is a reference to the product being purchased at the date of the *date* attribute, which is also stored as an epoch value. *price* is the price of the product, and *paid_for* is a special flag used by the system, which will be removed since it is only used for internal "bookkeeping" purposes[1].

When a member of the system performs a multi-buy of n products, then n equal rows are inserted into the **Sales** table. This poses a problem, since there does not exist any attribute combination, which is unique for all rows, thereby a primary key cannot be created, which reduces performance (especially since it is the largest table). This problem can be solved in two ways, either by adding a unique transaction identifier attribute or by merging transactions that are inserted by multi-buy. The problem with the first solution is that artificial transactions are created since products purchased during a multi-buy ought to be a single transaction. Thus, we choose the latter solution, which we will return to later in section 3.6.1.

3.3 Identifying Relationships

In the original database there does not exist any explicit relationships between the tables, that is, it does not contain foreign key constraints. However, implicitly a number of relationships are present in the form of attributes having similar names across tables as described previously.

In Figure 3.1 the tables are shown with the relationships which have been identified during the examination of the tables.

Next, the data is verified to conform to the presented relationships. For each of the relationships shown in Figure 3.1 it is ensured that a valid foreign key exists in the referenced table. For instance, the **Payments** table references the **Members** table, this means that every *user_id* in the **Payments** table must exist in the **Members** table.

These checks only present two cases with an invalid reference. Both are in the **Sales** table, where a member is registered with *user_id* 0, this member does not exist. To solve this problem both these cases are removed since the number of cases is insignificant compared to the amount of data.

3.4 Data Cleansing: Outlier Detection and Correction

We now turn to a deeper investigation of the more complicated tables and attributes of the data, with the objective of finding incorrect and extreme values that may be incorrect.

The tables **Coffee**, **Employee_type**, and **Products** contain so few rows that their values can be inspected manually, and no apparent invalid or extreme values are found.

The remaining data checks are done using queries, the precise queries being used can be found in Appendix B.2.

[1]Actually, it makes it possible to delete old transactions, this feature has, luckily, not been used.

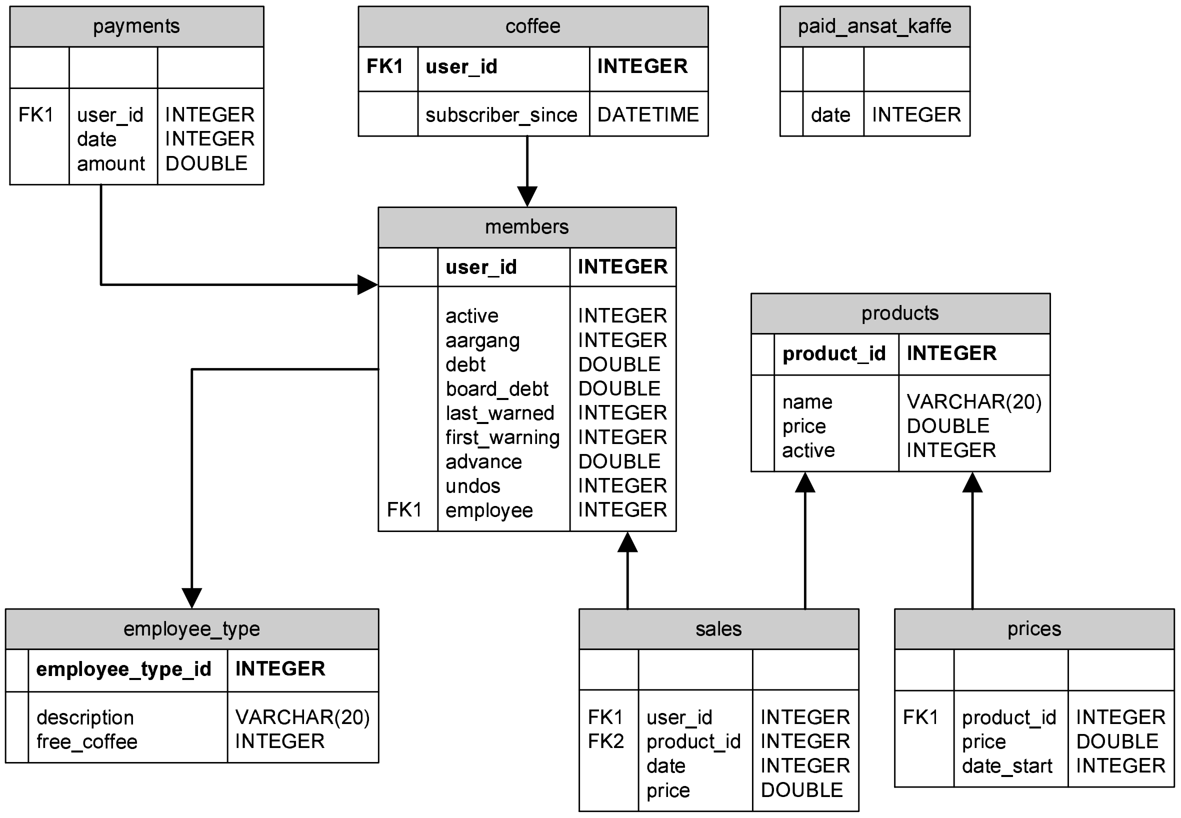

Figure 3.1: Database layout for the enhanced TREO database, with relationships added and irrelevant attributes removed.

3.4.1 Members Table

First the *user_ id* is checked to be positive, unique and non-null, these constraints are fulfilled. Next *active* is checked to be either 0 (false) or 1 (true) which is the case for all members. Then *aargang* is inspected by grouping the members by *aargang* and checking the number of members at each *aargang*. The result is shown in Table 3.11. The first problem is the inconsistency in the specification of the year, the value 1 is probably a mistyped 2001, however, 0 can be both an unknown value and a mistyped 2000. Inspection of the other years also reveal that the number of members in 2000 is very low compared to the surrounding years. To determine the members, with *aargang* equal to 0, which should be changed to *aargang* 2000, we use the sales data. We first determine at which date the first purchase was done by someone with aargang 1999, then we change the members with *aargang* 0 that have not purchased products before this date, and have done some kind of purchase. This procedure results in 18 members being changed.

The same procedure is done for members with *aargang* 1, all four of these members are changed to *aargang* 2001.

Finally *board_ debt*, *debt*, and *advance* are inspected using boxplots. These represent the balance status for each member. The boxplots are shown in Figure 3.2. The most extreme values are 5000dkr, however this is *advance* and *debt* for the same member, so it cancels out.

aargang	members
0	52
1	4
1975	9
1976	8
1977	13
1978	9
1979	16
1980	8
1981	20
1982	22
1983	36
1984	73
1985	86
1986	53
1987	46
1988	59
1989	66
1990	67
1991	90
1992	75
1993	63
1994	93
1995	77
1996	77
1997	95
1998	80
1999	129
2000	58
2001	100
2002	161

Table 3.11: Aargang

aargang	members (before)	members (after)
0	52	34
1	4	0
2000	58	76
2001	100	104

Table 3.12: Aargang: Members before and after adjustment

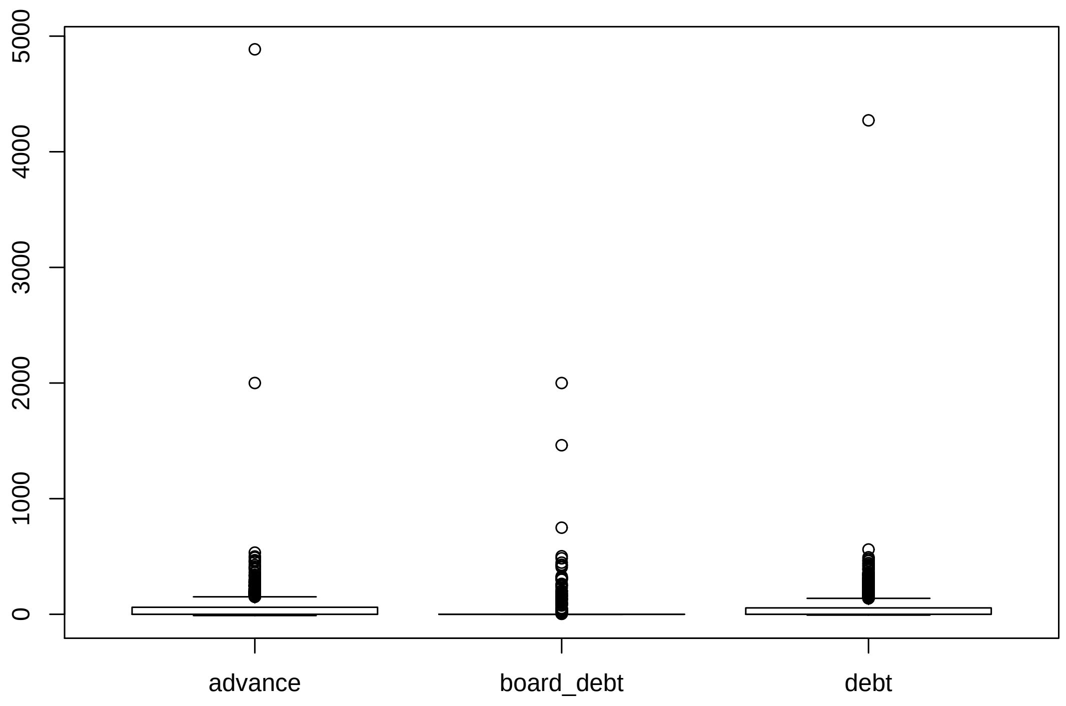

Figure 3.2: Boxplots for *advance*, *board_ debt*, and *debt* in the **Members** table

Likewise a value of 2000dkr is found for *board_ debt* and *advance* of another member, which also cancels out. Thus no obvious invalid values are found.

The remaining attributes for this table have been verified previously.

3.4.2 Payments Table

user_ids are already checked to be in the member table and the dates are within their proper range. However, as described in section 3.1, we are not completely sure about when the payments have been done in the past. For many years it has been done on fridays, and to check whether it always has been friday, the number of payments by year and weekday are shown in Table 3.13. This table shows that most payments have always been done on fridays. However, a significant number of payments are also done on other days, this might be explained by holidays, or special payments due to new members joining the TREO at when a new semester begins.

The inspection of the paid amount is done with a boxplot, shown in Figure 3.3, this reveals one very extreme value (22805 dkr versus 5000 and 4000). Looking at the minimum amount of payments, there are many very small payments (even a payment of 0,00), however, this can

Day \ Year	1996	1997	1998	1999	2000	2001	2002	2003	Total
Monday	1	6	17	126	27	47	101	5	330
Tuesday	0	41	26	50	32	17	101	0	267
Wednesday	0	17	20	80	54	99	207	3	480
Thursday	0	3	52	20	27	30	31	6	169
Friday	80	578	444	348	541	632	1031	22	3676
Saturday	0	0	1	72	2	13	9	0	97
Sunday	0	0	1	18	4	2	1	1	27
Total	81	645	561	714	687	840	1481	37	5046

Table 3.13: Payments by year and weekday.

user_id	date	amount
553	1997-04-18 11:12:27	22805,00
991	2002-08-08 10:54:20	5000,00
1	2000-10-06 11:01:53	4000,00
1	2000-10-06 11:00:27	4000,00
1	1999-11-19 12:04:35	3000,00
1	2001-09-07 11:47:12	2500,00
1	1997-11-14 13:11:00	2014,25
1	1997-11-14 13:09:44	2014,25
1347	2001-03-02 12:48:32	2000,00

Table 3.14: Payments $\geq$ 2000dkr

be explained by people leaving the club, who then pay their remaining debt.

Due to the very large payments found in the boxplots, we analyse the biggest payments further. The payments with an amount greater than or equal to 2000dkr are shown in Table 3.14. It can be seen that big payments are usually being done by the member with *user_ id* 1, however, even for this member there is an odd pattern in the payments, with equal amounts being done within minutes. This seems quite suspicious, since people usually only perform one payment on the same day, so it is likely a double entry of the amount. To investigate this matter further we find all the payments that are done by the same user on the same day. These are listed in Table 3.15. There is a total of 37 events, all of which are two payments done on the same day by the same member. It seems that there are two different kinds of events, one where the same amount is paid twice, and another where a small amount and a large amount is paid. The first could be explained by the abovementioned double entry of the payment, the latter by a mistyped amount, followed by a correction amount[2], however it seems odd that all amounts are postive in this case.

The problems, which are outlined above are partially solved, by removing one of the payments, in all registered double payments. The events with small and large values paid on the same day are not corrected, since they are assumed to be correct.

[2]inferred from the observation that people usually pay in some multiply of 100dkr

date	user_id	amount
1996-11-29 11:48:02	878	161,00
1996-11-29 11:48:22	878	8,0
1997-11-14 13:09:44	1	2014,25
1997-11-14 13:11:00	1	2014,25
1998-02-06 12:20:35	1125	200,00
1998-02-06 12:20:48	1125	200,00
1998-05-15 11:25:56	1	1500,00
1998-05-15 11:34:11	1	1,00
1998-09-04 11:16:53	1046	153,25
1998-09-04 11:18:51	1046	80,00
1998-10-21 09:08:02	1011	280,00
1998-10-21 09:09:31	1011	220,00
1998-10-23 10:47:23	1064	40,00
1998-10-23 10:47:51	1064	360,00
1998-10-26 11:08:20	1	1486,00
1998-10-26 11:09:20	1	1486,00
1999-01-27 13:24:55	1085	211,75
1999-01-27 13:25:10	1085	10,00
1999-03-12 11:34:24	1178	0,00
1999-03-12 11:34:47	1178	300,00
1999-04-29 10:44:13	615	250,00
1999-04-29 10:44:18	615	250,00
1999-06-29 12:23:21	1153	229,00
1999-06-29 12:23:27	1153	220,00
1999-10-04 13:32:12	1270	35,00
1999-10-04 13:32:35	1270	315,00
1999-10-04 13:40:00	1235	30,00
1999-10-04 13:40:20	1235	270,00
1999-10-05 10:26:31	1237	20,00
1999-10-05 10:46:58	1237	30,00
1999-10-29 10:28:47	1282	188,00
1999-10-29 13:33:19	1282	110,00
1999-11-12 11:56:39	1099	100,00
1999-11-12 11:57:38	1099	100,00
1999-11-12 11:59:17	1318	150,00
1999-11-12 11:59:37	1318	150,00
1999-11-19 11:58:39	1077	380,00
1999-11-19 12:01:54	1077	90,00
2000-02-25 13:09:31	1103	200,00
2000-02-25 13:09:32	1103	200,00
2000-06-21 13:03:24	677	200,00
2000-06-21 13:03:48	677	200,00
2000-10-06 11:00:27	1	4000,00
2000-10-06 11:01:53	1	4000,00
2000-10-27 11:27:29	1237	198,00
2000-10-27 13:14:03	1237	105,00
2001-03-09 08:10:25	1323	24,00
2001-03-09 12:10:10	1323	100,00
2001-05-02 12:58:52	1320	200,00
2001-05-02 13:02:07	1320	200,00
2001-07-13 14:04:46	1127	200,00
2001-07-13 14:04:50	1127	200,00
2001-09-07 11:39:44	1442	20,00
2001-09-07 11:39:54	1442	180,00
2001-11-23 12:20:01	1532	20,00
2001-11-23 12:20:18	1532	180,00
2001-12-21 11:52:56	1484	100,00
2001-12-21 11:53:07	1484	25,00
2001-12-21 11:56:45	930	175,00
2001-12-21 11:57:12	930	175,00
2002-02-06 11:19:55	1083	70,25
2002-02-06 11:20:13	1083	0,50
2002-04-12 11:20:21	1349	300,00
2002-04-12 11:22:10	1349	50,00
2002-05-03 11:13:33	1563	20,00
2002-05-03 11:13:40	1563	180,00
2002-06-07 10:47:52	1	150,00
2002-06-07 10:48:22	1	1350,00
2002-09-13 11:17:57	1438	100,00
2002-09-13 11:19:14	1438	100,00
2002-09-20 11:36:46	1744	50,00
2002-09-20 11:37:46	1744	50,00
2002-12-06 12:26:36	962	100,00
2002-12-06 12:28:45	962	100,00
# different members: 32		
# different days: 34		

Table 3.15: Multiple payments on same day by same member.

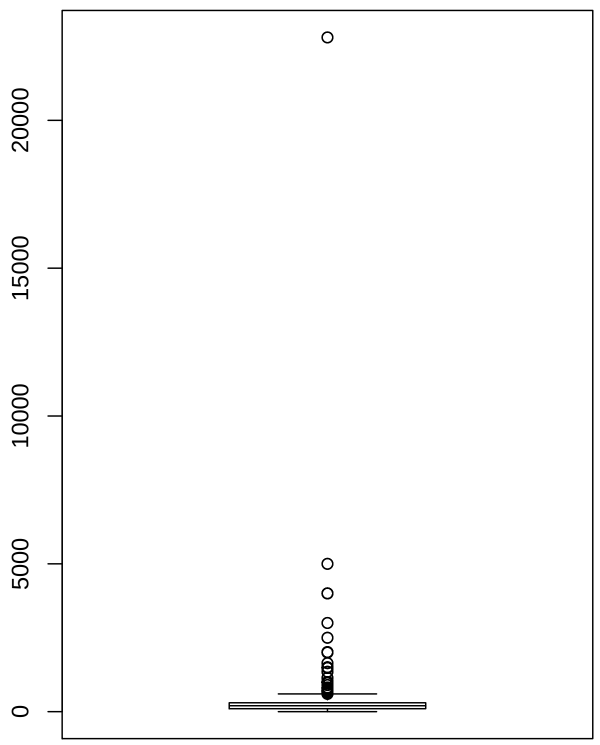

Figure 3.3: Boxplot of *amount* in the **Payments** table

3.4.3 Prices Table

product_ id and dates are valid. Investigation of the prices show no extreme values (when the type of product is taken into consideration). However, one product has a negative price, which is adjusted to a positive price within hours. The sales transactions show that no member has purchased the product to a negative price, so the price adjustment has most likely been during a test period or been a quickly corrected error, thus the negative price is removed from the table.

3.4.4 Sales Table

The *user_ id* and *product_ id* were previously checked and two transactions with invalid *user_ id* were removed. The dates are checked to be in their proper range, which they are. Finally, the *price* attribute is checked to be equal to the price found in the **Prices** table. This is done by looking up the price at the latest date in the **Prices** table, that is before the date in the

Sales table. For example, a product purchased at 1999-12-24 17:23:29 the largest date which is before the purchasing date is found in the **Prices** table, and the price change at this date is compared to the one in the **Sales** table. This check did not discover any problems.

3.5 Preliminary Design of Data Cubes

After the current investigation of data, there are two main subjects which can be modelled in a data cube. The first is the amount of sales, which most tables are related to. The second is members and their balance, including warnings given to them.

Whether these two subjects should be modelled using one cube or two cubes is not obvious. The balance and warning history of members could be seen as a dimension, with the sales data being the facts, or a smaller cube centered around the member balance could be created. Since we are not sure which approach is the best, we begin by creating a cube for sales with member as a dimension. Then, if it is needed, we can create a separate cube using this dimension later, if this is needed.

With the aim of describing the selected two subjects, most tables seem to be able to add information to the subjects. However, **paid_ansat_kaffe** does not contribute with any information.

3.5.1 Identify Grain and Dimensions

The first important choice that must be made is how fine-grained the sales data is modelled. Either every transaction can be modelled, or it can be summarised to some level, for instance, sales during a minute or an hour. Since the amount of data is small, we choose to model sales at the transaction level to preserve all information in the data.

During the examination of the sales table it was found that it references the members and products tables. Thus these tables are candidates for dimensions, which also seems sensible.

Furthermore, the date and time of the sale should be represented as a dimension. In general there are two ways to represent the date and time. Either as one dimension, or splitting it into a date dimension and a time dimension. The difference between the two solutions is the level of detail one wishes to be able to support, and the types of summarations that are deemed useful. Using one dimension makes it possible to summarise data by for instance 4th of April at 14:00. However, in the available data which represents about 6 years of sales this would not amount to many sales, furthermore, when analysing the data, it would be too detailed. Thus it is chosen to split the date and time data into separate date and time dimensions.

The **Payments** and **Prices** tables do not seem likely as dimensions, however, the data stored in them may be used to add derived attributes to the already mentioned dimensions.

3.5.2 Concept Hierarchies

Next we determine the concept hierarchies within the dimensions. In the illustrations for these hierarchies we do not show hierarchies with only one attribute. So, within each dimension the attributes are the ones shown in the concept hierarchy illustration and any attribute shown in the related table in Figure 3.1 which are not included in the depicted hierarchy.

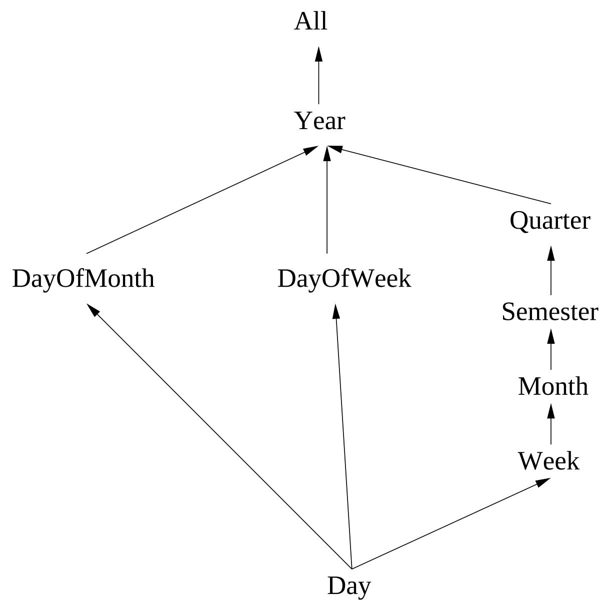

Figure 3.4: Concept hierarchy for date dimension.

Date Dimension

This dimension mainly consists of a year split into common calender terms, as shown in Figure 3.4. Furthermore, an attribute called *term* has been added, which describes the current term (or special event) at the university. More precisely the months are assigned a value in the set {Exams, Fall, Spring, Summer break} by the following mapping.

Semester	Months
Exams	January, June
Fall	September, Oktober, November, December
Spring	February, March, April, May
Summer break	July, August

Additionally, the day of week, month, and year are added.

Time Dimension

The time dimension consists of hour and minutes. Furthermore, a more coarse-grained attribute has been added called *timeofday*. This attribute maps the hours of the day into natural intervals in the set {night, morning, noon, afternoon, evening} as shown below.

timeofday	hour
night	1..5
morning	6..10
noon	11..13
afternoon	14..17
evening	18..23

The entire hierarchy is presented in Figure 3.5.

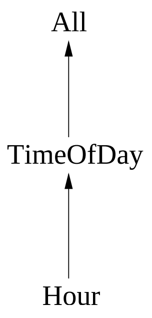

Figure 3.5: Concept hierarchy for time dimension.

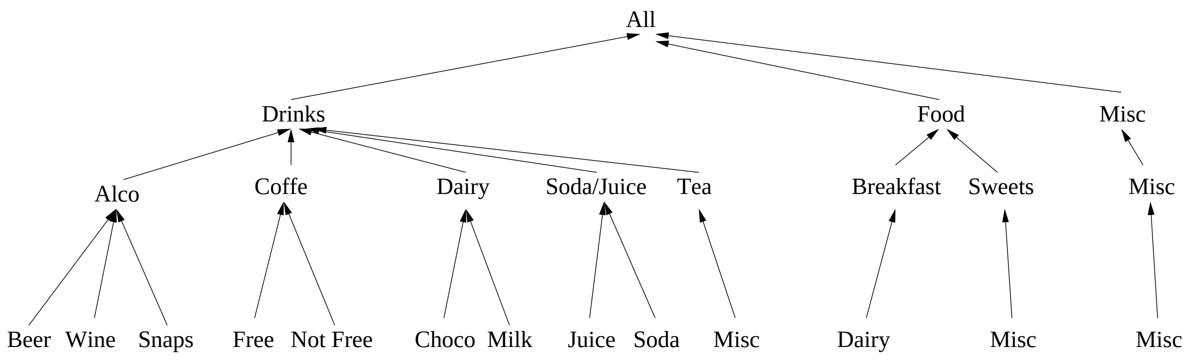

Figure 3.6: Concept hierarchy for product dimension.

Member Dimension

All attributes in the member dimension are single-attribute hierarchies.

Product Dimension

By examining the available products in the system, we have tried creating a logical hierarchy of related products, which is shown in Figure 3.6. Notice, that this hierarchy must be constructed using additional attributes in the product dimension. This can either be accomplished by adding all the values shown in the figure as boolean attributes, and setting these attributes accordingly for each product. However, a better solution exists, which is creating an attribute for each level of the hierarchy, and then use the values in the hierarchy is values for these attributes. This approach only requires three new attributes, and they form a linear hierarchy. The added attributes and values are:

MainClass	{ Drinks, Food, Misc }
Class	{ Alco, Coffee, Dairy, Soda/Juice, Breakfast, Sweets, Misc,}
SubClass	{ Beer, Wine, Snaps, Free, Not Free, Choco, Milk, Juice, Soda, Dairy, Misc }

With these attributes the concept hierarchy for the product dimensions can be represented using attributes as the one shown in Figure 3.7.

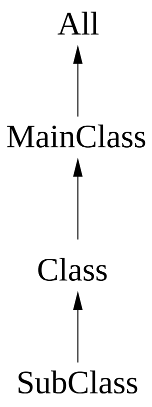

Figure 3.7: Concept hierarchy for product dimension using attributes.

3.5.3 Identify Measures

The measures available in the sales table is the price of the sold product. A new attribute called *n_units* is added, which is one for every sale, since only one product is registered for every sale. However, this attribute is useful when summarising data.

3.6 Data Transformation

We now turn to the transformation of data, that is, taking the data present in the source database from the TREO, and transforming this into structures that are easier to incorporate into a cube.

We have chosen to first create a star schema representing the cubes, then later use the OLAP tools to create cubes with aggregates. The reason for this choice, is that it becomes possible to analyse cube-structured data easily in the database using standard tools, instead of using specialised OLAP tools to access the cubes. However, it also means that aggregates are not being used. If aggregates are to be used then the real data cubes must be used using the specialised tools available.

First, we perform the simpler transformation tasks, then we create the tables to support the date and time dimensions. Next, we calculate the balance of members at the time of their purchase. Finally, we compute the historical data for members and create special attributes for sales summaries with regard to the number of active members.

3.6.1 Simple Data Transformation

The system registering purchases has a special feature called multi-buy, which facilitates buying larger amounts of a single product. The system does not register this event in any special way, instead, it simply inserts the correct amount of transactions into the **Sales** table. However, when doing this the transactions get the same timestamp. Thus the layout of sales data can be simplified by combining these artifical transactions into a single transaction. This is accomplished by using the *n_units* attribute to count the number of units sold,

attribute	type	description
balance	double	*advance − board_debt − debt*
first_purchase	int	epoch of first sales transaction
last_purchase	int	epoch of last sales transaction
first_payment	int	epoch of first payment
last_payment	int	epoch of last payment
never_used_system	boolean	1 if first_purchase and last_purchase are both NULL 0 otherwise

Table 3.16: Added attributes to the **Members** table

and adding the attributes *unit_price*, and *total_price* to the **Sales** table (or rather, a new copy of the **Sales** table, called **Sales_cmp**). Then the old sales data is inserted into the new table, with transactions having the same *user_id*, *product_id*, and *date* being merged. *n_units* is the number of products purchased, *unit_price* is the price for a single product, and $total\_price = n\_units \cdot unit\_price$.

The **Members** table is updated to contain a new attribute called *balance* which is given as the money due for the member minus the total debt of the member. Furthermore, when doing analysis of the members we assume this will be done in connection with payments and sales. To ease this analysis we add a number of attributes to the member table with regard to sales and payments. The added information is the first and last date a payment was made, and likewise for sales. Furthermore, a boolean attribute is added, to indicate whether any sales are registered at all for a member. The most important purpose of this attribute is to be able to filter out members who have not used the electronic system, since in many circumstances these members will not be able to contribute to the analysis of the sales data. The mentioned added attributes are shown in Table 3.16.

Finally the **Prices** table is reduced by removing price changes, which do not alter the price, that is, if the previous price for a product is equal to the new price in the table.

3.6.2 Date and Time Support Tables

Most of the analysis of the data relies on date and time values in some way. These are represented in data cubes as dimensions, and have to be improved considerably with derived attributes, instead of the usage of the epoch time format.

In section 3.5.1 we discussed the possible ways of designing the date and time dimensions and decided using two separate tables. Thus we create two tables, one for the date and another for the time dimensions. Additionally we create a table to map between epoch values and identifiers of the date and time dimension. Apart from this we try to derive as many attributes as possible for the date and time dimension tables.

This transformation can be done in various ways, either by using the data and time functions present in the database system, or by external programs. If the necessary functions are available in the database system, we presume these will give the least amount of problems, so this approach is chosen.

First all dates in the existing tables are extracted and converted into the SQL type datetime.

attribute	type
epoch	int
ts	datetime
dateid	int
timeid	int

Table 3.17: **Epoch_mapping** table

These values are then stored in the **Epoch_mapping table**, with epoch values as primary key and a datetime attribute. The schema for the this table is shown in Table 3.17.

Next, the two dimension tables are created, one for the date dimension, called **Datedim**, and another for the time dimension, called **Timedim**. The **Datedim** table is populated with all days between the first and last occurring in the system. That is, even days where there does not occur any events are added. The reason for this approach, is that most days are present in the system, and that some of the data processing, described later, becomes easier when all days are present. The **Timedim** table is only populated with values from the **Epoch_mapping** table, since the granularity of the time data would make a complete table a lot larger than desirable.

The general procedure done for each date value is as follows.

```
1: Discard time data
2: Remove duplicates (at selected data granularity)
3: Order by date
4: Insert into Datedim table
5: Add references from Datedim to Epoch_mapping
```

Since we are not going to add data to the cubes after they have been created, we chose to order the dates, to optimise the presumed most common form of full scan of the table, by increasing date value.

The same procedure is used for the time dimension, where the granularity is kept as it is (seconds), though only minutes are represented explicitly in the time table. The reason for this approach is that it does not remove the possibility of retrieving the ordering of the sales transactions.

The layout of the date and time dimension tables is shown in Tables 3.18 and 3.19. The derived attributes were described during the discussion of the date and time dimension in section 3.5.1.

3.6.3 Calculate Balance Attribute for Sales

An important aspect related to the sales transactions that is not available in the database is the balance of a member at the time of purchase, since this is crucial in deciding whether a member can purchase a product or not. However, it is possible to calculate the balance at the time of purchase if one combines the sales and payments data. That is, if we want to calculate the balance of member m at time t, it can be done as follows. Let ta be the set of all times registered in the system (whereby $0 \notin ta$), then $tp_t \overset{def}{=} \{x \in at \mid \text{ payment done by } m \text{ at time } x \wedge x < t\}$,

attribute	type	values
dateid	int	$\{1, 2, \ldots\}$
year	int	$\{1996, 1997, \ldots, 2003\}$
semester	varchar(50)	{ Spring, Fall, Summer break, Exam }
quarter	int	$\{1, 2, 3, 4\}$
month	int	$\{1, 2, \ldots, 12\}$
month_name	varchar(10)	{ January, February, ..., December }
week	int	$\{1, 2, \ldots, 53\}$
weekday	int	$\{1, 2, \ldots, 7\}$
weekday_name	varchar(10)	{ Monday, Tuesday, ..., Sunday }
dayofmonth	int	$\{1, 2, \ldots, 31\}$
dayofyear	int	$\{1, 2, \ldots, 366\}$
studyyear	int	$\{1996, 1997, \ldots, 2002\}$
studysemester	varchar(3)	{ E96, F96, E97, ..., E02 }
date	datetime	

Table 3.18: **Datedim** table

attribute	type	values
timeid	int	$\{1, 2, \ldots\}$
timeofday	varchar(20)	{ night, morning, noon, afternoon, evening }
hour	int	$\{0, 1, \ldots, 23\}$
mins	int	$\{0, 1, \ldots, 59\}$
time	datetime	

Table 3.19: **Timedim** table

and $ts_t \overset{def}{=} \{x \in at \mid \text{ purchase done by } m \text{ at time } x \wedge x < t\}$.

$$p_0 \overset{def}{=} 0, \quad s_0 \overset{def}{=} 0$$

$$p_i \overset{def}{=} \text{amount paid at time } i \text{ by member } m$$

$$s_i \overset{def}{=} \text{price paid at time } i \text{ by member } m$$

$$balance_{mt} = \sum_{i=0}^{n} p_i - \sum_{i=0}^{o} s_i$$

$$\text{where } n \overset{def}{=} \max\{0\} \cup tp, o \overset{def}{=} \max\{0\} \cup st_t$$

Notice that since we use the sales data before time t the balance will be before the current purchase. To emphasize this we do not add an attribute called *balance*, instead we call it *balance_before*, and additionally we add an attribute called *balance_after*, which is defined as *balance_before* + *total_price*. These attributes ought to be calculated correctly, however, due to the way the system was introduced, and the lack of registration of data, there are a number problems with the approach.

1) Members who used the old non-electronic system had an initial balance, which is not registered.

2) When a member undoes a purchase, it is only registered that that an undo-action has been performed. Neither the date of the undo-action, nor the product or price is registered. Thus artificially increasing the debt of members, who have used the undo feature.

3) Erroneous payment entries, which have been manually corrected, are not registered completely, thereby making our calculation wrong for these members.

1) can be amended by using the balance information from the **Members** table. Basically, a correction factor is added to the calculated balance. This factor is derived from the difference between a members balance calculated at his last purchase, and the *balance* in the **Members** table. However, this factor will also be affected by 2) and 3), thus not necessarily calculated correctly. This means, that if a member has only used the electronic system, but has used the undo feature or has had a misregistered payment, then the member will get a correction factor for 1) that he should not have. To solve this problem partly we only apply the correction factor to members with *aargang* 1996 and previous years, since these have been members during the transition period. The remaining members are assumed to only have used the electronic system, thus any differences must be due to 2) or 3). Both the uncorrected and corrected balances are added to the **Sales** table, the uncorrected attributes are prefixed with "unc" as shown in Table 3.20

The correction factor is stored in a table called **Member_correction**, which is shown in Table 3.21. For every member registered with *aargang* 1996 or less, the correction factor is calculated and added to **Member_correction**. This is calculated as

$$correction \overset{def}{=} balance - (\text{sum of payments} - \text{sum of purchases}),$$

where *balance* is from the **Members**, and the two sums are from the **Payments** and **Sales_cmp** tables, respectively. All the remaining members are added with a correction value of 0.

attribute	type
balance_before	double
balance_after	double
unc_balance_before	double
unc_balance_after	double

Table 3.20: Calculated and derived attributes for **Sales** table

attribute	type
user_id	int
correction	double

Table 3.21: Schema for **Member_correction** table

3.6.4 Historical Member Data

Based on the **Members** table we can find the balance of all members, and then derive whether they are warned or blocked from using the system, by comparing their balance to the thresholds set by the TREO for these events. However, this is only possible for the day where the data in the database was extracted from the purchasing system, since there does not exist historical data for any of the attributes in the **Members** table. With a small exception with regard to *first_warning* since it gives the date at which the member was last warned due to a low balance. However, even this attribute only contains information about the last time a warning was issued, there is no information about earlier warnings. This is a major problem if one wants to analyse buying patterns, since it is crucial to have the member balance, and member status, that is, whether the member is warned, blocked, or neither warned nor blocked.

It would be possible to use the balance described in the previous section, however, the granularity is too fine and it would be difficult to work with. Instead we have chosen to create a table containing historical information for each member at each day present in the database. Even if we had chosen a higher granularity it would still not account for the imprecise calculation of balance as described in the previous section. Choosing an even coarser granularity would not be advisable, since the way warnings and payments interact means that most members probably just have a warning between 1 and 7 days. If, for instance, a week had been chosen instead, many warnings would not even be detected.

For each member and day the balance is calculated. Based on this calculation and the date, a number of related attributes are added, shown in Table 3.22. *warned* and *blocked* are boolean attributes, which determine the status of the member. Additionally, the number of days the member has been in a certain state is registered, as well as the number of days till a member is blocked (if the member has gotten a warning that is).

Before the balance for a member can be calculated, it must be defined what balance is for a member on a specific day. The problem is that a member's balance may change during the day, so a consistent way of determining the balance must be found. Two main possibilities are either choosing a time of day, where the balance current balance is chosen, or taking the average or mean of balance values for the given day. We do not think think that the choice

attribute	type	description
user_id	int	
dateid	int	
balance	double	calculated balance at end of day
active	boolean	member has done purchases on this day
warned	boolean	member has been warned
days_warned	int	days member has been warned
days_till_block	int	days until member is automatically blocked
blocked	boolean	member has been blocked
days_blocked	int	days member has been blocked

Table 3.22: **Member_day** table

will result in much difference, so we chose the simplest, that is, a specific time. To keep the calculations simple we use the latest time possible on the day, which means that all payments and sales for a given day is included in the members balance.

The actual procedure for calculating the balance is done for each member, who has used the electronic system, as follows:
(**Member_day**.$balance_d$ refers to the *balance* in the **Member_day** table on day d)

```
1:  $s_d \leftarrow$ First date of purchase
2:  $p_d \leftarrow$ First date of payments
3:  $first\_date \leftarrow \min\{s_d, p_d\}$
4:  **Member_day**.$balance_d \leftarrow 0 \quad \forall d < first\_date$
5:  $balance \leftarrow 0.$
6:  **for all** $day \geq first\_date$ **do**
7:      $amount_{sales} \leftarrow \sum_{i=1}^{n}$ price of purchase$_i$ on $day$
8:      $amount_{payments} \leftarrow \sum_{i=1}^{o}$ amount of payment $i$ on $day$
9:      $balance \leftarrow balance + amount_{payments} - amount_{sales}$
10:     **Member_day**.$balance_{day} \leftarrow balance$
11: **end for**
```

After the balance has been calculated for all members, who have used the electronic system at some point in time, the attributes related to warning and blocked status are calculated[3]. This is performed by scanning the days from the earliest to the latest in time order, comparing the balance to the warning and blocked limits. At the same time, data is processed with respect to number of days warned or blocked, and the number of days until being blocked.

```
1: $w_{ts} \leftarrow$ warning threshold
2: $b_{ts} \leftarrow$ blocked threshold
3: $warned \leftarrow 0$ {Is the member warned?}
4: $blocked \leftarrow 0$ {Is the member blocked?}
5: $days_{warned} \leftarrow 0$ {Number of consecutive days the member has been warned}
6: $days_{blocked} \leftarrow 0$ {Number of consecutive days the member has been blocked}
7: $days_{till\_block} \leftarrow 0$ {Number of days until member is blocked}
```

[3]Actually it is combined procedure to save time, however, it can be viewed as two separate to simplify the algorithms

```
 8: for all $day$ in Member_day do
 9:   if $md\_balance_{day} < w_{ts}$ then
10:     if $warned = 0 \wedge blocked = 0$ then
11:       $warned \leftarrow 1$
12:       $days_{till\_block} \leftarrow 14$
13:       $days_{warned} \leftarrow 0$
14:     end if
15:     if $warned = 1$ then
16:       $days_{warned} \leftarrow days_{warned} + 1$
17:       $days_{till\_block} \leftarrow days_{till\_block} - 1$
18:     end if
19:     if $blocked = 1$ then
20:       $days_{blocked} \leftarrow days_{blocked} + 1$
21:     end if
22:     if $blocked = 0 \wedge (days_{warned} = 14 \vee \textbf{Member\_day}.balance_{day} < b_{ts})$ then
23:       $warned \leftarrow 0$
24:       $blocked \leftarrow 1$
25:       $days_{blocked} \leftarrow 0$
26:       $days_{warned} \leftarrow 0$
27:       $days_{till\_block} \leftarrow 0$
28:     end if
29:   else {Member's balance is below warning threshold}
30:     $warned \leftarrow 0$
31:     $blocked \leftarrow 0$
32:     $days_{blocked} \leftarrow 0$
33:     $days_{warned} \leftarrow 0$
34:     $days_{till\_block} \leftarrow 0$
35:   end if
36:   Update $warned$, $blocked$, $days\_warned$, $days\_blocked$, and $days\_till\_block$ in $\textbf{Member\_day}_{day}$
      according to appropriate variables.
37: end for
```

The mentioned procedures are used twice, first for creating a table, **Member_day_unc**, which uses uncorrected balances, and a second time to create the **Member_day** table, which uses the previously calculated corrections. This correction is used at the first balance value having a value different from 0 (that is, in step 3 of the balance calculation algorithm, the balance is set to the correction value, not 0).

3.6.5 Sales Data Summaration

In the same manner as we created day summaries for members, we also want to create these for sales. The purpose of this approach is to create summaries, which are easier to use for analysis. For instance, when analysing the amount of sales during a certain period, we do not believe precise member information is required, that is, data about which members did the purchasing. Instead we want to have data about the number of active members, which is the amount of distinct members that have purchased items during the period.

The summaries described above are created by creating a table, which contains a row for the cartesian product of all days and products. That is, every row describes the sales of a single product on a given day. The information registered is the number of units sold, the total price of the units, and many different ways of counting the distinct number of active members. The problem with active members, is that the information cannot easily be aggregated. If we have the active members for each day, then it is not possible to determine the active members during a week only based on the counts for each day, since we cannot determine duplicate members across the days. Thus, we create many different types of aggregates for this information. Note, that this information is equal for all products on a given day.

In addition to the sales summaries and active members, we would also like to have information about the price changes. This is achieved by adding an attribute describing whether a price change has occured on a given day for a specific product, and an attribute with the amount the price has increased (which will be negative if the price is decreased). Furthermore, it is registered how many days the products has not had any price changes.

A summary of the attributes can be seen in Table 3.23.

attribute	type	description
product_id	int	
dateid	int	
n_units	int	number of product units sold
total_price	double	total price for sold product units
price_change	boolean	was price changed for current product on current day?
price_increase	double	amount price was increased (negative for reductions)
active_members_day	int	active members during day
active_members_week	int	active members during calendar week
active_members_month	int	active members during calendar month
active_members_semester	int	active members during season/semester
active_members_year	int	active members during calendar year
active_members_ssemester	int	active members during study semester
active_members_syear	int	active members during study year
active_members_rweek	int	active members for current day $\pm$ 3 days
active_members_rmonth	int	active members for current day $\pm$ 15 days

Table 3.23: **Sales_day** table

3.7 Data Mining Directions

In this section we analyse which patterns or knowledge the club could be interested in finding, and we analyse how well the data supports the suggested data mining directions.

3.7.1 Possible Data Mining Scenarios

Case a) Identify members that cheat with payments.

Due to the way the shop is run there are no measures to control whether customers pay for their items or not. Thus, it is unlikely that it will be possible to detect customers that cheat frequently. However, there is another kind of cheating, which could also be due to members forgetting to pay. This occurs when members are registered as owing money to the club, and therefore are not allowed to purchase items. This combined with the fact that payments are only done on fridays, means that some members will purchases items without paying until they have paid money to the club again – or maybe not paying for the products in the period. It would be interesting to detect this behaviour if possible.

Case b) Identify patterns leading to missing bottles.

When buying soda water, members are expected to return the bottles they have bought the drink in. Currently, a lot of bottles are disappearing, which results in the club losing money. It would be advantageous to find patterns leading to this behaviour, thereby making it possible to determine measures for avoiding these patterns.

Case c) Ability to predict the sales of (certain) products.

Some of the products sold by the club, have a very limited lifetime, so naturally these items should not be ordered in excessive quantitities. Thus it would be nice to be able to predict the future sales of the these products.

Case d) Analyse effects of price changes.

This option is relevant for many shops, however, it is not important for this club, since the club has very low prices, due to it not trying to make profit.

3.7.2 Data Requirements and Availability

Case a) The data contains detailed information pertaining to sales and payments, furthermore the balance of each member at the date of the snapshot is available. To perform the analysis described in this case, we assume clustering would be able to detect some patterns, which can be further analysed. In order to improve the clustering, a number of extra attributes might be needed.

We believe the balance at the time of purchase is important. Furthermore, the days until next warning and days since last warning are important, since these give information about when the possibility for purchasing items will be blocked. However, this also means that in the current database, the attributes we believe are most important, have been calculated since the original database did not containt enough historical.

Case b) With the current data it will be imposible to analyse this case. An important attribute with respect to this case is the number of disappearing bottles, and this number is not being registered.

Case c) The available data is sufficient for this case, since the focus already is on the sales of products. The initial analysis can be done by summarising the sales for a given product over a period, looking for patterns in this summary. The sales will be increasing, since the number of active members has been increasing since the start of the club, so it will be necessary to take this into account. The first idea to solve this is by looking at the sale per active member. Doing so, should make it possible to analyse how the sales are over the weeks of a term or year.

Other relevant information that can be considered is active members that are in excessive "blocked from buying" states, which can be due to them not paying their debt and simply buying using the account of other members.

After considering summarised data, it might be possible to use classification to analyse it even further.

Case d) This analysis should be based on the results in case c). There exists historical data regarding the price changes. It would be possible to normalise using active members and a week profile developed in case c), and then analyse these sales with respect to price changes.

Chapter 4

Multi-dimensional Data Mining

In this chapter we first describe some of the related work we are aware of. Next we present some problems we have encountered when trying to use standard tools to data mine multi-dimensional data. Then we investigate concept hierarchies more closely, since these are the most important description of structure within the data warehouse with regard to data mining. Finally, we describe our view on how data mining could be done on multi-dimensional data and how meta data can be used.

4.1 Related Work

In this section we briefly introduce the work, we are aware of, which has been done with within the area of data mining multi-dimensional data.

A short overview of data cubes and data mining, with the focus on rules, can be found in [Pal00].

The main part of the work with regard to mining of multi-dimensional data has been done by Professor Jiawei Han, and his colleagues and students at Simon Fraser University. An introduction to the work can be found in “OLAP and Data Mining”[Han98]. The main contributions are presented in the book “Data Mining: Concepts and Techniques” [HK01], which introduces data mining and data warehousing, and deals with the integration of these areas. However, the discussions dealing with this integration are very broad, and only go into detail with regard to mining association rules. Some of the material in the book is based on several PhD and master theses, these contain a more detailed discussion of the topics. Concept hierarchies, automatic generation of concept hierarchies, and multi-level rules are thoroughly analysed in [Fu96]. In [Lu97] different types of concept hierarchies are investigated and a special representation of concept hierarchies is proposed to optimise the performance of the roll-up and drill-down OLAP operations. [MWG$^+$97] briefly considers decision tree induction in relation to concept hierarchies. [Tar98] deals with special types of data cubes created for the purpose of data mining. In [Pin01] multi-dimensional sequential pattern mining is investigated, which is followed up in [PHP$^+$01].

Finally, a more detailed discussion of the integration of multi-dimensional data and data mining can be found in [Che01].

At Microsoft Research there has also been done research on the issues of combining multi-dimensional data and data mining, as well as relational data and data mining. The focus of this research is more database-centered. In [BCF99] classification is considered in combination with SQL databases. [NBCF01] deals with the integration of data mining and SQL databases, with the objective of introducing the interface called OLE-DB for Data Mining, which is a common interface programmers can use to utilise data mining in databases. Lastly, the efficiency of querying parts of data mining models has been investigated in [CS02].

4.2 Clementine Experience

We were curious about how the current data mining tools would handle multi-dimensional data. So we tried finding an approach that would allow us to work with the data using a common tool. Due to our good experience with Clementine in general, we chose this as the data mining tool we would test.

The first major problem, is that Clementine cannot access data cubes directly, since its data import capabilities are restricted to file access and relational database access. Thus we use the cubes, which are stored in the relational database using a star schema. Hereby it is possible to access the cube data, by performing a join between all dimension tables and a fact table.

It later became apparent, that using database access was quite slow compared to using formatted text files, so we chose to export the data from the database to a text file instead, using the abovementioned join.

Even with these changes, there were still some inconvenient tasks which had to be done every time the attributes in the text file were changed. The main problem was that there did not exist a way to specify the type of each attribute. This is not a major problem if you only use the tool occasionally, however, if it is used frequently, then it would certainly be nice if one could specify options about every attribute directly in the database once and for all.

The final inconvenience, we encountered, was the lack of dimension and concept hierarchy information, which of course is to be expected when the tool has not been designed for such structural information. However, when there are many dependent attributes present in the data being analysed, then it becomes increasingly difficult to manage the attributes used in training data mining models. When the input and output attributes are chosen, then one must be aware of the depedencies among attributes. If two input attributes are dependent, then it depends on the classification type, whether this causes problems or not, however, it will slow down the algorithm. If an attribute, which is closely related to the output attribute, is used as input attribute, this attribute may determine the output attribute completely, or at least improve the results artificially.

With these problems in mind, we now turn to analysing how multi-dimensional data mining can be done, and how the user interface can be improved.

4.3 Concept Hierarchies

This section is based on [HK01] and [Lu97].

A concept hierarchy specifies a mapping of data, from a set of low-level concepts to higher-level concepts. In Figure 4.1 a concept hierarchy is shown for the days in years the 2000 and 2001. At the lowest level, each day is represented, above these, the month corresponding to a given day is shown. These months are again mapped to the year the month occurs in.

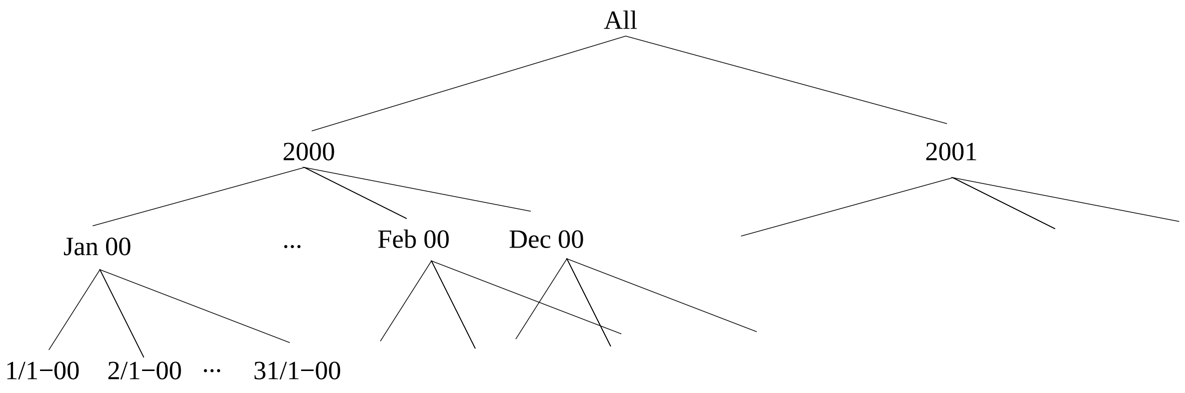

Figure 4.1: Instance-defined concept hierarchy.

The main purpose of concept hierarchies is to support specialisation and generalisation. That is, if the data is viewed at “day”-level, by generalisation it can be viewed at month, year, or all level. In the same manner specialisation is supported by the concept hierarchy.

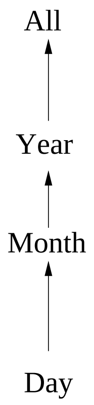

Figure 4.2: Schema-defined concept hierarchy.

The hierarchy in Figure 4.1 is called an instance-defined concept hierarchy because the hierarchy is based on the actual values in the data. Another approach is to define the hierarchy based on attributes in a database schema, which is known as a schema-defined concept hierarchy. Figure 4.2 shows the equivalent schema-defined concept hierarchy for Figure 4.1.

The precise definition of a concept hierarchy is:

Definition 4.1 (Concept Hierarchy)
A concept hierarchy is a partially ordered set $(H, \succ)$, where H is a finite set of concepts and $\succ$ is a partial order on H. □

A concept hierarchy is also called a taxonomy, is-a hierarchy, or a structured attribute.

person	*age*
1	38
2	5
3	19
4	44
5	80

Table 4.1: Original age data.

Schema- and instance-based concept hierarchies can be defined as follows.

Definition 4.2 (Schema-based Concept Hierarchy)
Let A be the attributes of a dimension in a data cube. Then a schema-based concept hierarchy is a partially ordered set $(H, \succ)$, where H is a finite set of concepts, $H \subseteq A$, and $\succ$ is a partial order on H. $\Box$

Definition 4.3 (Instance-based Concept Hierarchy)
Let A be the attributes of a dimension in a data cube, let $Val(a), a \in A$ be the values attribute a_i attains in a data set D. Then an instance-based concept hierarchy is a partially ordered set $(H, \succ)$, where H is a finite set of concepts, $H \subseteq \cup_{a \in A} Val(a)$, and $\succ$ is a partial order on H. $\Box$

Instance-based concept hierarchies are also called set-grouping hierarchies.

Whether one should use a schema-based or instance-based concept hierarchy, is usually not obvious, and in many situations both can be used. Consider a discrete attribute called *age*, which registers a persons age in whole years. If we want to generalise the age ranges, we can both use schema- and instance-based concept hierarchies.

The original data is shown in Table 4.1. Now, we want to split the age into the ranges 0-10, 11-20, ...,91-, and young (0 - 20), adult (21 - 40), middle aged (41 - 60), and old (61 -). If we want to use a schema-based concept hierarchy, this can be done by adding two new attributes, one for the numeric ranges, and another for the more general textual ranges. The resulting data can be seen in Table 4.2 and the associated schema-based concept hierarchy to this table is shown in Figure 4.3. The same can be accomplished with the original data and an instance-based hierarchy, as the one shown in Figure 4.4.

In general an instance-based concept hierarchy can always be represented as a schema-based concept hierarchy. However, a schema-based concept hierarchy can only be represented as an instance-based concept hierarchy if it is a total order. If it is not, the schema-hierarchy can be split into smaller hierarchies which have a total order, then these can be transformed to instance-based concept hierarchies.

Two other types of concept hierarchies are defined in [HK01] and [Lu97]. The *operational concept hierarchy* is an instance-based concept hierarchy, which is generated by a set of operations on data. This could for instance be some discretisation procedure used on continuos attributes, or for instance clustering could be used. The other type, is the *rule-based concept hierarchy*, which is a concept hierarchy where the generalisation of a concept has a rule attached, which

person	*age*	*age_ numeric*	*age_ textual*
1	38	31 - 40	adult
2	5	0 - 10	young
3	19	11 - 20	young
4	44	41 - 50	middle aged
5	80	71 - 80	old

Table 4.2: Schema-based age data.

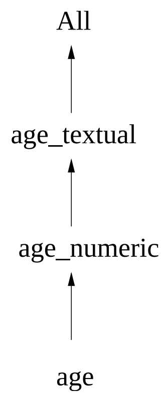

Figure 4.3: Schema-based concept hierarchy for *age*, *age_ numeric*, and *age_ textual* attributes.

is evaluated using any data available in the database. If we consider the age example again, a rule-based concept hierarchy could, for example, use data regarding the century in which the person lived to determine the age description. Neither the operational, nor the rule-based concept hierarchy will be used later in this report, since they are variations of the schema- and instance- based concept hierarchies.

4.4 Data Mining in Data Cubes

Normally when some form of data mining is performed, data consists of a number of cases, each with a value for a number of attributes. However, when a data cube is used, the data has a more complicated structure, thus it is necessary to analyse this structure, and determine how data mining can be performed using it. **[mere intro...+ eksempel]**

4.4.1 Data Cube Structure

As previously described in chapter 2, a data cube consists of a number of dimensions, and a number of measures related to the dimensions. Using a star schema approach, this results in a database table for each dimension and a fact table, which stores the measures and a reference to all the dimension tables. Additionally, each dimension has a schema-defined concept hierarchy, which supports the generalisation and specialisation operations on the cube. The entire structure is shown in Figure 4.5.

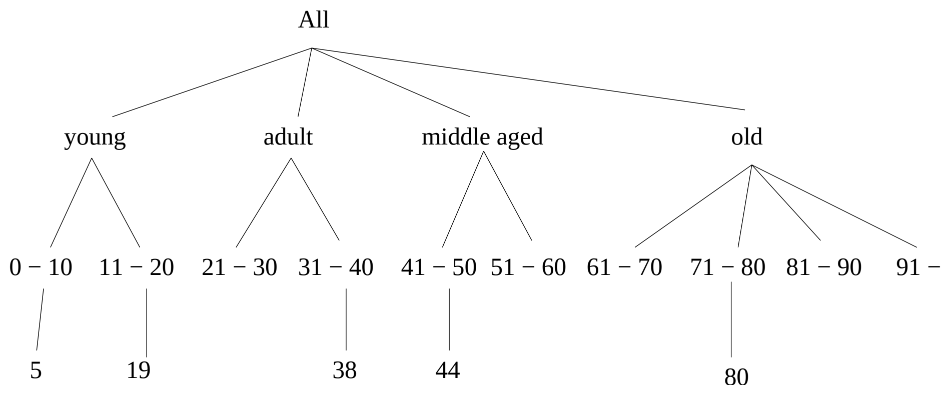

Figure 4.4: Instance-based concept hierarchy for age attribute.

We are going to make an assumption about the concept hierarchies describing the dimensions. The assumption is that the concept hierarchies contain a top node, labelled "All", which is the most general description of data (all data), and a bottom node, corresponding to the granularity unit chosen for the dimension, which is the most specific description of data (a single case, at the granularity unit). Furthermore, all paths in the concept hierarchy are from the bottom node to the top node.[1]

Based on the way dimensions are used during the data mining phase, one can split it into two different kinds, intra-dimensional and inter-dimensional data mining. *Intra-dimensional data mining* only uses one dimension and the measures, or only the dimension. *Inter-dimensional data mining* is data mining using more than one dimension. One possible use for intra-dimensional data mining could be determining or improving the concept hierarchies for complex dimensions. However, in the following we focus on inter-dimensional data mining.

4.4.2 How Should Facts be Weighted?

An important step before being able to data mine a cube, is to find ways to extract information about the transactions which have resulted in the cube at hand. The two main objectives that we see, are:

1) Find a relation between the dimensions and the facts, such that it can be determined when a transaction has taken place or not.

2) If possible, find a way to determine how many events in the domain being analysed have resulted in a single fact row.

Usually 1) can be achieved by storing the number of transactions in the original data. Then the sum aggregate operation can be used to find the number of transactions when generalising dimensions. Thus, a transaction has taken place when the value of the transaction-count-measure is greater than zero.

If each transaction in the original data corresponds to an event in the domain which is being analysed, then the mentioned transaction-count-measure will also give the number of trans-

[1]This assumption only holds when data is only stored at a single unit of granularity.

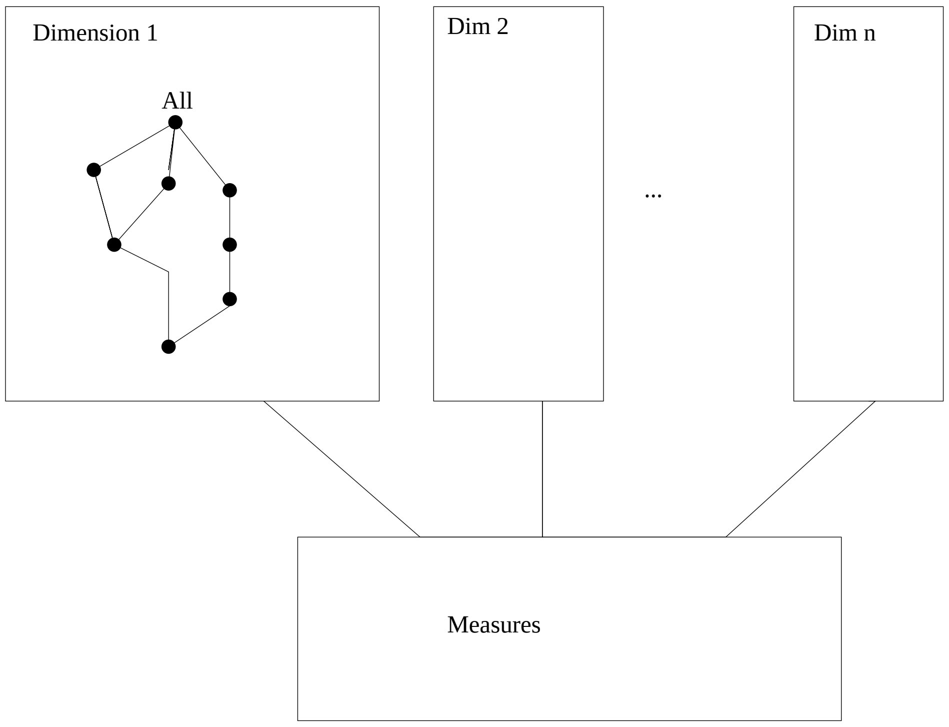

Figure 4.5: Structure of a data cube.

actions. However, if the original data compresses these events or handles them in other ways, it may be necessary to use other kinds of aggregation functions.

For instance, in our handling of the TREO data, the sales data is compressed, such that when multiple items of the same kind have been purchased at the same time, it is only stored as a single transaction. As described in 3.6.1, the compressed sales data contains an attribute, which stores the amount of items purchased, this corresponds to the events in the analysis domain when analysing how much has been sold. However, if it is the number of people using the system at a given time that is being analysed, then it would be another matter, since one person buying multiple items should only count as a single event. In the first case, it would be possible to use the items sold as an indicator of the number of events in the domain, and using the sum aggregate function would result in the correct number of transactions when analysing the data cube. In the second case, the same attribute can be used, however, it should use the count aggregate function instead.

Even in the simple case where only a single dimension is being analysed, one must be careful with which weight is attached to each row in the dimension. Consider a customer-dimension, which contains the address of the customer, including the city and postal code. Then assume we want to determine how likely it is that the mapping between city and postal code is correct. Which weight should be attached to each customer in the customer dimension? It depends entirely on how the mapping between city and postal code has been verified. If it is verified when the customer data is entered into the customer dimension, then each row in the dimension should have equal weight. However, if the customer data is used for billing information, residing in a fact table, then the mapping between city and postal code may be verified every time the customer occurs in the fact table. Thus, the weight should be the number of occurences in the fact table in this case.

As can be seen from the above paragraphs, there is no single answer to how the number of events in the domain of analysis can be found, since it depends on how these events are registered. Thus it can only be said that this must be analysed before data mining can be performed in a data cube, and it would be nice if the data mining tool supported this weighting of rows depending on various criteria.

4.4.3 Attribute Definition and Selection

An important part of the data mining task is to select the relevant attributes to perform the data mining on. If an attribute is not selected its information is lost. At the same time, the worst factor in the complexity of data mining algorithms is the number of attributes in the data set.

The main difference between data mining traditional case-based data and data cubes, is the difference in their underlying attribute structure. Thus, to use the classic algorithms, we first have to map the attributes in the data cubes to case-based attributes.

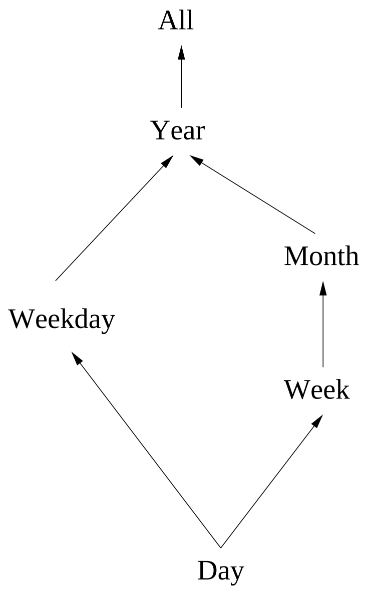

Figure 4.6: Schema-defined concept hierarchy for date dimension.

Before we continue with how it is possible to map dimension-attributes to case-based attributes, we must look closer at the schema-defined concept hierarchies. Consider the concept hierarchy in Figure 4.6, if the current level of generalisation is the *Day* level, and we want to generalise this level, there are two possibilities, either *Weekday* or *Week*. However, when we use the concept hierarchies for analysis, it is preferable to have a unique way of specialising and generalising. We do this by splitting the hierarchy based on the paths which exist from the bottom vertice to top vertice. The result is shown in Figure 4.7.

With this in mind, we see a number of ways to perform the mapping from dimension-based

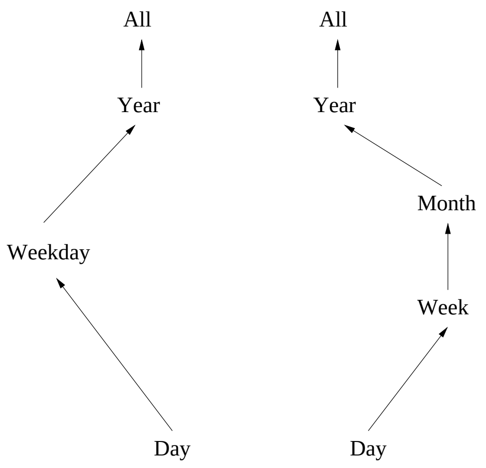

Figure 4.7: Top-bottom-paths of date concept hierarchy.

to case-based attributes:

1) Map all attributes from the dimensions to a case-based attribute.

2) Map all top-bottom-paths in the concept hierarchy to a case-based attribute, and select a generalisation level in this path.

3) Map all top-bottom-paths in the concept hierarchy to a case-based attribute, and run the data mining algorithm at each generalisation level in this path.

The first approach suffers from a number of problems. First of all, it results in a large number of attributes, which significantly increases the complexity of the data mining algorithms. Furthermore, there is a chance of getting trivial results, which could be read from the actual concept hierarchies. For instance, consider a date hierarchy All $>$ Year $>$ Quarter $>$ Month $>$ Day, a rule of the form Month $=$ January $\Rightarrow$ Quarter $=$ Q1 is not interesting.

The second approach solves the problems described above, since dimension-attributes which are related in the concept hierarchy are not present in the data mining data at the same time, however, the user has to specify the generalisation level for a potentially large number of top-bottom-paths. Furthermore, there might be some interactions between different levels in the top-bottom-path, which cannot be detected when using this approach, depending on the data mining algoritm being used.

The third approach is a variation of the second approach, instead of having the user specify each generalisation level, all of them are tried automatically. However, this easily becomes an intractable task.

Approach 2) and 3) could be combined, such that the user specifies the initial levels, and nearby levels are tried automatically.

However, none of these approaches are perfect, thus it might be better to modify the existing data mining algoritms. For instance, such that each top-bottom-path corresponds to an attribute, and the algorithm then use generalisation and specialisation operators on these.

4.4.4 Mining the Data

After the attributes have been processed as discussed in the previous section, these can be represented as traditional case-based data. Thus, it is possible to analyse them using the current data mining tools, which to do not support data mining on cube data. However, it must be considered how the input to the data mining tools should be generated. If we have a list of attribute values, these could correspond to a large number of transactions. There are two ways of solving this problem:

1) Duplicate the attribute values until they correspond to the correct amount of transactions.

2) Use a "weight" attribute, which specifies how many transactions the other attribute values represent.

Clearly the second approach is preferred, since it can reduce the amount of data considerably. However, both the data mining tool and the used algorithm must support the use of a weight attribute, which is often not the case.

4.5 Using Meta Data

As we described in chapter 2, meta data is an important part of a data warehouse. We believe that the use of meta data should also be extended to data mining purposes, that is, using meta data to support the data mining tools.

The main advantage, we anticipate, of this meta data, is to ease the description of attributes during data mining. Instead of having to specify that a variable is continous or discrete, this information can be store as meta data. Even more detailed groupings of attribute types can be defined. Another possibility is to store data which only changes during the loading process of the data warehouse, for instance the number of distinct values an attribute attains. This is common information used to decide whether a variable is too specific to be included in a data mining task, and the global storage of it, would improve performance. Depending on the actual systems being used, it could also be possible to store concept hierarchies, attribute storage type, and measure aggregation types as meta data. This information may also be available directly from the DBMS, however, most ways to determine this information differs from DBMS to DBMS. Thus, a common storage of this information could make the data mining tools more portable.

Chapter 5

Cube-based Decision Trees

In this chapter we first introduce the decision tree induction algorithm, in a very general form. Then we consider at which points of the algorithm, it can be modified in general. Finally, we consider each of these modification points with respect to data cubes and determine the kinds of improvements which can be obtained.

5.1 The Generald Decision Tree Induction Algorithm

The following algorithm is based on the basic algorithms shown in [HK01] and [Dun03], then extended to make it as general as possible, without introducing splits on more than one attribute.

Generate_tree function(D,A)
Input: D (training data), A (candidate-list of possible attributes)
Output: Decision Tree

```
1: if all samples in $D$ belong to same class, $C$ then
2:     return leaf node, labelled with class $C$
3: end if
4: if $A = \emptyset$ then
5:     return leaf node, labelled with appropriate class in $D$
6: end if
7: for all $a \in A$ do
8:     determine best way to split of attribute $a$, resulting in splits $s_{a1}, s_{a2}, \ldots, s_{an}$, each $s_{ai}$
       with a predicate $p_{ai}$
9:     calculate split-measure for $a$ using best split
10: end for
11: Choose $r$, attribute with best split-measure
12: Create node $N$, label it with $r$
13: for all $s_{ri}$ do
14:     add arc from $N$, label it with predicate $p_{ri}$
15:     $D' \leftarrow \{d \in D | p_{ai}(d) \text{ true } \}$
16:     $A' \subseteq A$
```

```
17:     if stop criterion reached then
18:         add leaf node with appropriate class in D
19:     else
20:         attach tree returned by Generate_tree($D'$,$A'$)
21:     end if
22: end for
```

Notice that step 16 is usually $A \leftarrow A \backslash \{r\}$, however the attribute need not be removed, since another type of split could be performed on the same attribute later.

When the algorithm stops adding splits, either due to lack of attributes to split on, or due to a stop criterion being reached, an appropriate class is chosen from the remaining data at the current part of the tree. This is usually the most common class in the part of the considered data, but need not be. We are not going to consider this decision a way of modifying the algorithm, since it only affects the class chosen when the tree is not grown any deeper, which is not the part of the algorithm we want to deal with.

Furthermore, we are not going to deal with the stop criterion being used, since it is difficult to estimate the effects of it in a general setting, where the pruning phase is not in place.

The remaining possible ways of improving the decision tree induction algorithm, that we see as possible are:

1) attribute-selection

2) constraints on available split attributes

3) split-measure

4) constraints on available split points/method of selecting split points

5) split-point-measure

6) pruning/post-processing of tree

Another aspect that must be considered is how the algorithm is improved, we classify the improvements into three categories:

1) Complexity reduction

2) Explainability improvement (more intuitive and simpler trees)

3) Classification accuracy improvement

Where 1) improves on the runtime of the algorithm, 2) deals with how the user perceives the resulting tree, and 3) deals with the objective quality of the resulting tree.

5.2 Multi-dimensional Improvements

First, each kind of improvement is considered isolated with regard to data cubes.

In the discussion that follows, a common example will be used. It consists of two dimensions, a date dimension consisting of the attributes Year, Quarter, Month, Week, Weekday, Date, and a location dimension, consisting of the attributes, Country, Region, City, Street, Address. For each dimension a schema-based concept hierarchy is created, these are shown in Figure 5.1.

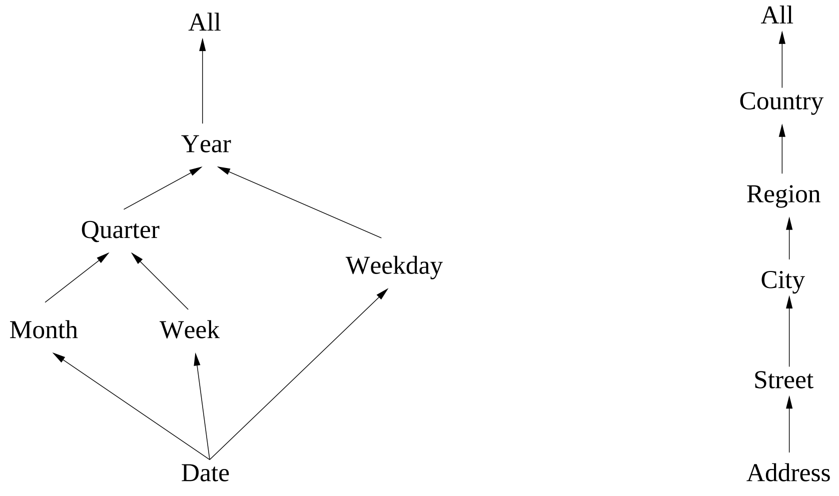

Figure 5.1: Concept hierarchies for date and location dimension.

CH_d and CH_l are the concept hierarchies for the date and location dimensions, respectively. Furthermore, the following total ordered schema-based concept hierarchies are defined:
TCH_{d1}: All $>$ Year $>$ Quarter $>$ Month $>$ Date.
TCH_{d2}: All $>$ Year $>$ Quarter $>$ Week $>$ Date.
TCH_{d3}: All $>$ Year $>$ Weekday $>$ Date.
$TCH_l = CH_l$ (since it is already a total ordering).

The values of the different attributes is assumed to be the following:
Year: 2001, 2002, 2003.
Quarter: Q1, ..., Q4.
Month: Jan, ..., Dec.
Week: 1, ..., 52.
Weekday: Mon, ..., Sun.
Date: 1/1-2001, ..., 31/12-2003.

Country: Denmark (DK), Germany (GE)
Region: Northern Jutland (NJ), Central Jutland (CJ), Southern Jutland (SJ), the islands (I), Northern Germany (NG), Southern Germany (SG).
City/Street/Address: Too many to list.

Note that the abbreviations shown in parentheses are used in figures to make them more compact.

During the next sections it will sometimes be necessary to differentiate between the possible predicates used when defining the splits of an attribute. The types of predicates we consider are $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, and $\in$. Where $\in$ refers to a branch that is followed when an attribute is in a set of elements. However, we would like to simplify these to only two kinds of splits, thus, we introduce equality- and membership-splits.

A branch which is followed with predicate $=$, that is, when the attribute is equal to a specific value, is referred to as an equality-branch. An attribute where all branches are of the equality-branch type is referred to as an equality-split attribute. A membership-branch, and membership-split attribute are defined likewise for the $\in$ predicate. Unless otherwise mentioned all predicates, except $=$, are treated as the membership predicate. If for instance the predicate is "< 7", all values less than 7 can be found and grouped into the set S, then it is enough to test for "$\in S$".

5.2.1 Constraints on Split Attributes

Suppose that all the attributes of each concept hierarchy are viewed as possible attributes to include in the decision tree induction algorithm. This results in attributes being included that are generalisations or specialisations of other attributes (provided the concept hierarchies are complex enough).

Under certain circumstances it is possible to reduce the amount of attributes that are candidates for a split. If the complexity of deciding which attributes can be skipped is less than the complexity of including the attributes, then it is possible to reduce the overall complexity of the algorithm.

First we must analyse under which circumstances an attribute can be ignored, then we need to analyse the complexity of deciding whether an attribute can be ignored or not.

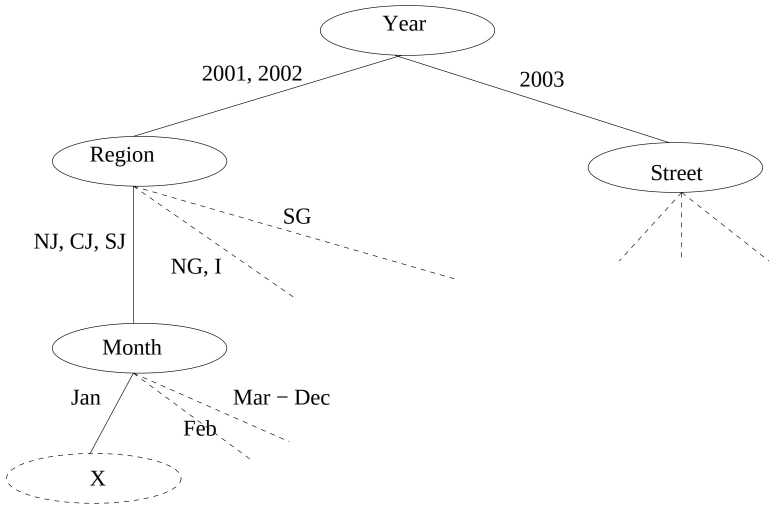

Figure 5.2: Available split attributes, example.

When deciding the split attribute at X in the decision tree shown in Figure 5.2, some attributes are not necessary to consider. These are Quarter and Year, since the split on Month implies that all data at this point are in Q1 2001. Also, Country is irrelevant since the regions NJ, CJ, and SJ reside in the same country.

In general it can be seen that if an equality-split is performed on attribute $A \in TCH_i$, then all attributes in TCH_i, which are more general than A can be discarded. If a membership-split is performed on attribute $B \in TCH_j$, then the situation is more complicated. All attributes in TCH_j, which are more general than B are candidates for being excluded. However, the only attributes that can be excluded, are the ones that have same value for all data under consideration at the split-attribute.

Clearly, the situation where a previous equality-split has been performed can be used to optimise the decision tree induction, since the decision of which attributes can be disregarded only relies on the information on concept hierarchies.

However, when dealing with membership-splits it becomes more complicated, since the values of data has to be investigated. One possibility may be to store instance-based concept hierarchies, and use these for lookup of information. Part of the instance-based concept hierarchy for the location dimension is shown in Figure 5.3. To decide whether an attribute can be disregarded, first all values that are part of the membership-split must be found in the concept hierarchy, then their parents in the tree must be found in the tree. If they have a common parent, then the attribute corresponding to investigated level in the tree, and all its parents can be disregarded. If they do not have a common parent, then one can investigate the parents of the parents, and so on.

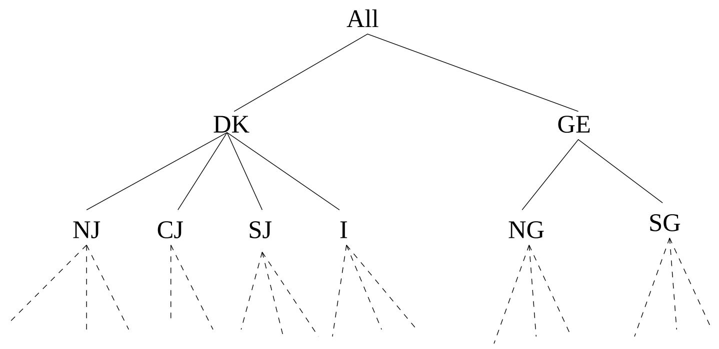

Figure 5.3: Partial instance-based concept hierarchy for location dimension.

This would most likely reduce the complexity, since these concept hierarchies are only related to a single dimension, and dimensions contain far less data to consider than a dimension joined with the fact table.

Another possibility may be to divide the attribute-selection into a two-level process, where an attribute from each concept hierarchy is tested first. Based on this test the best, or the n best concept hierarchies are found. Within these concept hierarchies all attributes are tested. Thus, the complexity is reduced if there is a significant amount of non-single-attribute concept hierarchies. However, how this approach would influence the quality of the found decision trees is not easily predicted.

5.2.2 Split-measure

The split measure itself could also be modified to use information from the dimensions and concept hierarchies. We do not see any immediate use, however, it would be possible to reward or punish attributes depending on whether they are from the same dimension or not. Likewise, an attribute could be rewarded/punished depending on how general or specific it is.

At the moment we do not see a possible use for this, since it is not clear whether it is advantageous to have many attributes from the same dimension or not, however, it could be left to the user as an expert option to tune the mining algorithm.

With regard to rewarding an attribute for being general, this may avoid overfitting, but this is only speculation, and it depends on the split measure being note.

5.2.3 Constraints on Available Split Points and Split Point Measure

If all attributes from the dimensions and concept hierarchies are used, then it does not seem possible to improve the chosen split points based on the extra information from the concept hierarchies.

So suppose that each total ordered concept hierarchy is viewed as a possible attribute, named TCH_i as described above, and that to each such attribute a property is attached, which describes the generalisation level that the attribute is at, denoted by $L = level - name$. For instance, the attribute TCH_{d1}, could have $L = Month$, meaning that the attribute is considered at the Month-level.

Then we propose the idea of using the attribute values at a certain generalisation level as possible split points. More precisely, consider the attribute TCH_{d1}. Based on the example, the possible ways of splitting this value, which should be considered, are the following:
Year-level: { 2001, 2002, 2003 }
Quarter-level: { Q1, Q2, Q3, Q4 }
Month-level: { Jan, Feb, ..., Dec }
Date-level: { 1/1-2001, 2/1-2001, ..., 31/12-2003 }

After each set of split points have been evaluated, either the most general is chosen among the ones with best split-point measure or if the split-point measure does not reward more general levels compared to more specific levels, then the choice made should be based both on the split-point measure and the level of generality. This is due to specific split points being able to classify the training data more precisely, thus reducing explainability.

See Figure 5.4 for an example of such a decision tree.

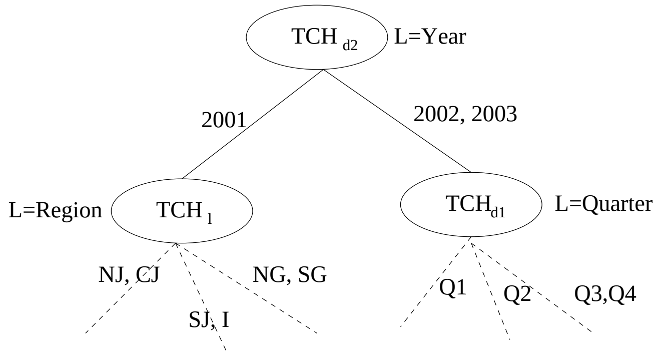

Figure 5.4: Decision tree with constrained split points.

The purpose of choosing these split points, is to improve explainability. For instance having the split points Q1, Q2 compared to the ranges [Jan,Feb,Mar] and [Apr,May,Jun], seems like a more compact description of the same concept. The improvement is even greater if it is compared to actual dates as split points. Figure 5.5 shows an example of a decision tree with the same classification abilities as the one in Figure 5.4, but with a less general split-point/attribute selection.

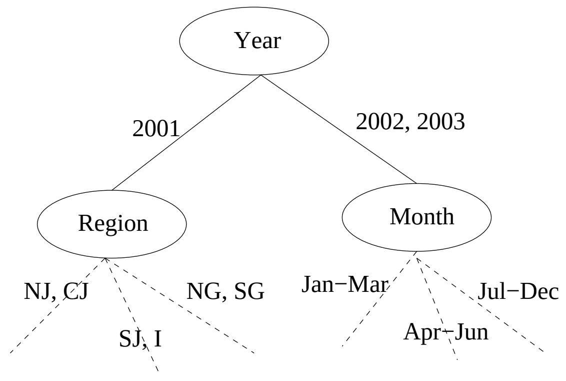

Figure 5.5: Decision tree without constrained split points.

Another possibility may be to not restrict the level at the node level, but instead allow different generalisation levels at each branch.

Finally it may be possible to use the aggregates stored in the cube when calculating these split points, and thereby reducing the complexity of the algorithm.

5.2.4 Pruning/Post-processing of Tree

The last possible modification is to generate a decision tree, and then modify the tree before or after the pruning stage, with the objective of increasing its explainability.

First we must establish when an attribute and its split points can be generalised in an existing decision tree without altering the classification accuracy. This is easier to see using an example, Figure 5.6 shows an instance-based hierarchy. Before an attribute at Month level, can be generalised to Quarter level, the split points it has, may only be the ones that divide the quarters. That is, every split on the Month attribute must contain a complete quarter or a number of complete quarters. If this is fulfilled, then the split represent exactly the same data, due to the relation between Month and Quarter. So, a split involving only January and February, cannot be generalised to Q1, since this could include more data (data with Month = March). Likewise, a split involving July to September, cannot be generalised to Q3, since this could exclude data for the month June.

Suppose that the attributes used in the decision tree induction algorithm is all attributes available in the dimensions. This could result in a tree of the form shown in Figure 5.7.

Consider the leftmost Region attribute, which has a split on { NJ, CJ, SJ, I } and { NG, SG }. If we use the instance-based concept hierarchy for the location dimension, shown in Figure 5.3, it can be seen that the two groups of data can be generalised to the Country level. That is, the region split attribute can be replaced by a Country split attribute, with the branches, DK, and GE. Thus a simpler split involving less values is achieved, without altering the classification accuracy.

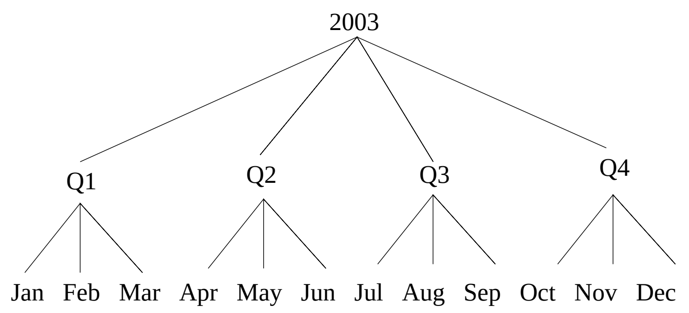

Figure 5.6: Partial instance-based concept hierarchy for TCH_{d1}.

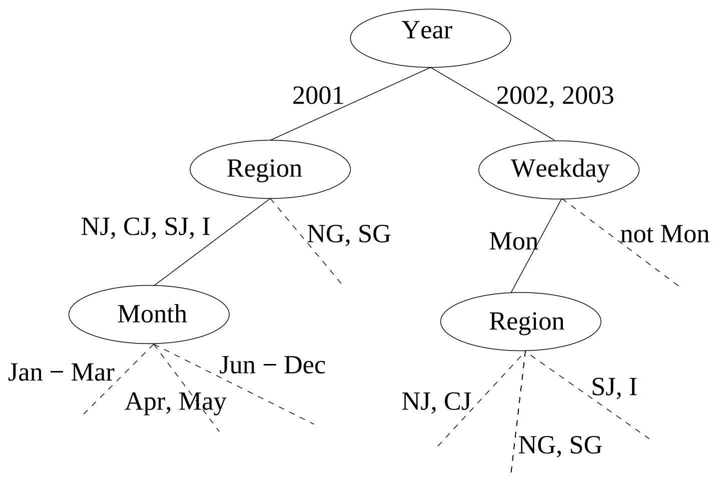

Figure 5.7: Decision tree before/after pruning.

However, the above scenario will not occur that often if there are many possible values for an attribute, since it requires the split attribute to split on very rare sets (the sets defined by the instance-based concept hierarchy).

A more likely situation is presented in the Month attribute, where the months are almost split according to the Quarter level. It would be possible to use Quarter as a split attribute, using Q1, Q2, Q3-Q4 as the split conditions, however, this would place June in the wrong branch. If Month was chosen over Quarter due to better classification accuracy, this will lead to a reduction in classification accuracy. Whether the tree with improved explainability and reduced classification accuracy is better than the original tree depends on how much the explainability is improved and how much the accuracy is reduced. So, some measure is needed to compare these factors, then a decision based on this measure can be made.

Another situation, illustrated by the rightmost Region attribute, is splits lining up to different generalisation levels. This region attribute is split into { NJ, CJ }, { NG, SG }, and { SJ, I }. { NG, SG } can be generalised to GE, since all regions for Country = GE are included in this split. However, { NJ, CJ }, cannot be generalised to DK, since this would include more regions than defined by { NJ, CJ }. The specific problem here, is that part of the branches can be generalised to a higher level, while others cannot. A possible solution would be to allow a split on more than one attribute, thus allowing branches at different generalisation levels. That is, a node called "Region/Country", with splits that specify which attribute they use for the split. This solution could also be used for the Month attribute, by generalising Jan-Mar to Q1, without altering the rest of the splits.

In Figure 5.8 the tree from Figure 5.7, can be seen after the discussed simplification steps.

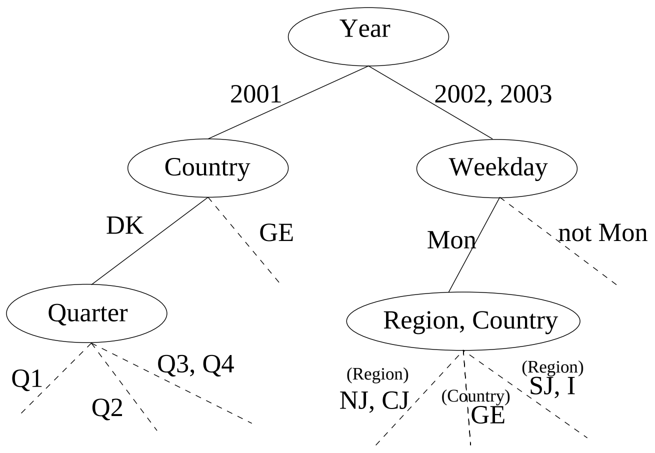

Figure 5.8: Decision tree after simplification.

5.2.5 Interactive Decision Tree Induction

A final idea we propose is to view each total ordered concept hierarchies as an attribute with a level attached. Then begin the decision tree induction by setting the level of each attribute, either using some heuristic algorithm or by using attribute-oriented induction. Then the decision tree is induced and presented to the user. At this point, the user should have the choice of changing the level, associated with each attribute, to investigate the consequences of this change. Depending on how the levels and attributes are selected, it may only be necessary to induce the the subtree with root in the node whose level is changed. This could result in a significant reduction in complexity depending on where the node is situated in the tree. Also, if the original induced tree did not chose the optimal level of the attribute that is to be changed, then it may be necessary to move the node up in the tree, resulting in more calculations.

Chapter 6

Evaluation of Some Proposed Ideas

In this chapter we analyse the experiments we have conducted based on the previous analysis. We have created a prototype implementation of some of our ideas. The application has been programmed in C# and connects to Microsoft SQL Server for database access[1]. The choice of language is mainly due to its easy database access and due the author's curiosity about the abilities of this new language. The source code for the application is available at "http://www.cs.auc.dk/∼peterj/mdm/"[2].

6.1 Multi-dimensional Data Mining User Interface

Based on the problems described in section 4.2, and the discussions in section 4.4, we have experimented with creating a user interface for multi-dimensional data mining.

6.1.1 Attribute View and Selection

We have tried the two views described previously, that is, either viewing all attributes as attributes, or viewing concept hierarchies as attributes with a level. None of these two approaches is consistently significantly better than the other. It mostly depends on the type of concept hierarchy and the number of attributes in the concept hierarchy. If the concept hierarchy consists of highly related attributes choosing the level is easiest, and it does not seem to be too limiting to exclude the remaining attributes. This is also true if it is a small concept hierarchy, since the number of attributes being excluded is also small. However, when dealing with large or less related attributes, then it is not enough to simply be able to select one of them. Our conclusion is that, both views of data should be supported, and selectable by the user.

With regard to attribute selection and deselection, we have found that choosing or removing attributes from an entire dimension or concept hierarchy can be useful, though it should still be possible to select or deselect single attributes. This is especially true with regard to the

[1]We are in the process of modifying the database access, so it will be possible to use it with Microsoft Access, whereby it will be easier for other people to test, and it will also be easier to supply other people with test databases. However, this conversion has not been completed yet

[2]An executable version and test data will appear when the Microsoft Access database access is functional

selection of a target attribute, since it becomes possible to deselect all attributes chosen as input attributes from a dimension or concept hierarchy when a target attribute is chosen inside this dimension or concept hierarchy. We have tried using a selection hierarchy consisting of dimension, concept hierarchy, total ordered concept hierarchy, and attribute. However, this becomes too complex, so instead we suggest that the user can choose between using either concept hierarchies or total ordered concept hierarchies.

In summary:

- Both the "attribute as selection unit view", and the "concept hierarchy as attribute with level" view should be supported, and user-selectable.
- Selecting or deselecting attributes as input attributes should be possible at the dimension, concept hierarchy, and attribute level.
- When a target attribute is chosen, the user interface should provide the option of automatically removing all input attributes in the dimension or concept hierarchy the target attribute is part of.
- Using both concept hierarchies and total order concept hierarchies at the same time will most likely make the user interface too complex.

6.1.2 Data Mining Mode

With regard to the actual data mining (and in part to the available dimensions and measure attributes), we suggest that three different types of data mining is supported. These are:

- Intra-dimensional.
- Fact-weighted intra-dimensional.
- Inter-dimension.

The first type is the simplest and performs data mining within a single dimension. The second uses a dimension table joined to the fact table, thus weighting the dimension rows after their number of occurrences in the fact table. This thereby includes the measure attributes as possible input attributes. It could also be possible to use a measure attribute to determine the weight each fact should have. The third type is what we expect to be the most common type of data mining in multi-dimensional data, since it uses more than a single dimension. When this type of mining is used, it should be possible to specify some expression using the measure attributes, which evaluates to true when a transaction occurs in the fact table. This would most commonly be a measure which should be non-zero. Additionally a measure-attribute should be selectable as weight, like the *n_ units* measure, which was described during the TREO data analysis.

We have implemented the above data mining methods, and found them sufficient for our purposes, however, they have only been used on a single data cube, so we may very well have missed some other necessary functionality.

6.1.3 Meta Data

All information regarding attributes, concept hierarchies, dimensions and the fact table have been stored as meta data. This has proved useful, in comparison to storing the information statically, since it becomes very easy to adjust, for instance, concept hierarchies on-the-fly. Additionally, it is very convenient not to be required to specify the types of attributes, since this information is stored in the database.

6.1.4 Simplifications

With the current prototype, a few simplifications have been made. The first simplification is that there can only exist one cube in a database, this is to simplify the design. One lesson learned is that when one has to manage dimensions, concept hierarchies, and attributes, it becomes a lot more complex than the traditional rows and attributes.

The second simplification is that the cube functionality of the database is not used. This is due to portability and lack of time.

6.2 Decision Tree Induction Modifications

In this section we describe the prototype implementation of one of the decision tree improvements we proposed earlier. This implementation as been included in the abovementioned application.

6.2.1 Basic Algorithm

We have implemented the C4.5 algorithm, which only uses equality-splits, and we have chosen to simplify it by only allowing discrete attributes.

The algorithm has been implemented with a split-measure based on the following definitions, from [Mit97] and [Jen01]:

$$Entropy(S) = \sum_{i=1}^{c} -p_i \log_2 p_i$$

where S is a collection of rows, with c classes, p_i the proportion of rows in S belonging to class i, $0 \log_2 0$ defined to be 0.

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

where $Values(A)$ is the set of possible values for attribute A, and S_v the subset of S where A has value v.

$$SplitInformation(S, A) = -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

where S_i is the subset obtained from S when partitioning S by the c-valued attribute A.

With the actual split-measure being:

$$GainRation(S, A) = \frac{Gain(S, A)}{SplitInformation(S, A)}$$

Usually these calculations are done in main memory, by starting with the entire database and then partitioning it as the decision tree is induced. However, this approach cannot be used with large databases, since they cannot fit into main memory. Thus we have tried implementing it without holding the data in main memory, instead we query the database. This approach has the added benefit of giving us a hint about the type of queries the database must answer during standard decision tree induction, so we can analyse whether a data cube would improve these queries. We have not implemented any pruning strategies, since we want to keep the focus on how the splits are performed during the tree induction phase. If pruning is used, it is difficult to distinguish between effects of the split of attributes and the final pruning of the tree.

6.2.2 Tests

There are two of the proposed algorithms we would like to test, one is the use of specific split intervals based on the available data in a concept hierarchy. However, this approach is not interesting in comparison to a traditional C4.5 algorithm using only equality-splits, since the result would be equal to simply using all the attributes from the input concept hierarchies and then induce a decision tree using these.

Instead we focus on the algorithm, which use representatives from concept hierarchies to test the goodness of the concept hierarchy, and then only test all the attributes in the best concept hierarchy plus all attributes in single-attribute concept hierarchies. We are interested in discovering how large a reduction in attribute goodness tests this approach gives, and how the approach affects the prediction abilities of the decision tree.

We have chosen to only use a subset of the data, which is available in the cube to speed up the calculations. The data has been divided into 3 subsets, a training set consisting of 34K rows, and an evaluation set and a performance set, both consisting of 17K rows. The evaluation set is to be used by the induction algorithm during for instance pruning. While the performance set is never used by the induction algorithm, it is only used for testing the performance of the final classifier on unseen data. We saw two choices for dividing the data, one selecting rows at random, and another, where the rows are selected based on the date, such that the training set has the oldest data, the evaluation set some newer data, and the performance set has the newest data. The latter is more difficult for the classifier, so this approach is chosen since we are interested in detecting even small changes between the two algorithms.

A final choice regarding the tests, is the selection of a representative from the concept hierarchies. We have chosen to use the most specific attribute, which is not a primary key.

We have chosen a set of specific cases, where the input attributes has been chosen based on the size of the concept hierarchies, and not on whether it would give some interesting knowledge about the data.

6.2.3 Results

In the following we present 5 cases, which have been used to examine how the base decision tree (c4.5) behaves, compared to the modified algorithm (mod), where a representative from concept hierarchies is used. *Attr* is the number of attributes whose split-measure has been calculated, *Train* is the fraction of correctly classified training cases, and *Performance* is the fraction of correctly classified performance cases. We have chosen not to show the actual decision trees, most of the trees using c4.5 are very complex due to the lack of pruning.

Case 1: Input Attributes:
(Memberdim) active, aargang, semester, undos.
(Timedim) timeofday, hour.

Target: (Memberdim) free_coffee.

	Attr	Train	Performance
C4.5	322	0,9211	0,9222
Mod	114	0,9161	0,9188

The performance difference is very small in this case, while the modified algorithm has done significantly less calculations. By inspection of the trees it has been seen that the C4.5 tree is very complex due to it using the *hour* attribute, which the modified algorithm does not use.

Case 2:

Input Attributes:
(Memberdim) active, aargang, semester, undos.
(Productdim) name, active, MainClass, Class, SubClass.

Target: (Memberdim) free_coffee.

	Attr	Train	Performance
C4.5	541	0,9779	0,9773
Mod	437	0,9779	0,9773

The trees in this case are equal, however, the modified algorithm used slightly less calculations.

Case 3:

Input Attributes:
(Memberdim) active, semester, free_coffee, undos.
(Datedim) year, semester, quarter, month, week.

Target: (Memberdim) aargang.

	Attr	Train	Performance
C4.5	677	0,9087	0,8635
Mod	69	0,8931	0,8688
Mod2	76	0,8931	0,8688

In this case the difference between the trees is very large, this is due to C4.5 including the *month* attribute fairly close to the leaves of the tree in several branches. We have tried to modify the representative used in the modified algorithm to see whether it would choose more attributes. The new algorithm (mod2) does a test on the top and bottom nodes of the hierarchy (excluding the All node, and any primary key). However, this does not change the resulting tree.

6.2.4 Conclusion

It is difficult to make any sound conclusion on these limited tests, but they do show that there can be a reduction in the number of calculations done. However, the resulting decision trees are also different. In this limited test the differences have not had any impact on the precision of the classifier. More tests are clearly needed, and with the use of a pruning phase it would be possible to judge the “Performance” values better.

Chapter 7

Conclusion and Future Work

In this chapter we conclude on the work we have done, then we consider some of the future work which is possible.

7.1 Conclusion

In this thesis we have introduced data warehousing and the dimensional model. We have described the general features of these, that is, their common use, design and possibilities. Then we have analysed the available data in a traditional relational database, which describes sales transactions and customer payment transactions. It was found that there was not registered enough data to make a proper analysis, mainly due to the database was not designed to store historical information about the customers. Nonetheless, we used this data and analysis of data to construct a small data warehouse using the dimensional model.

Then we tried analysing the data warehouse using a traditional data mining application, with the main objective of discovering how a traditional tool would handle data warehouse structured data. Several problems were found, and we have suggested solutions to these.

Next, we have analysed how decision tree induction can be improved when the dimensional model is used. We have proposed several possible improvements.

Finally, we have performed an evaluation of our proposed solution to the general data mining task, and some of the specific improvements to the decision tree induction algorithm. The evaluation of the way data mining can be done in general when the data is multi-dimensional, has been performed by creating an application with the proposed functionality. However, we can only document our personal experience with this application, since no formal tests have been done on its interface. The evaluation of the improvements to decision trees has been done with respect to lowering the amount of attributes that must be tested for split-abilities. It was found that our suggested method of testing representatives of each total ordered concept hierarchy, resulted in a lower amount of tests without reducing the quality of the decision trees significantly. However, these results were only obtained on our test data, which did not contain many concept hierarchies, so the results are not conclusive.

7.2 Future Work

An experience from this project, which did not surprise us, was how time consuming the preparation of data is. It has often been stated in the literature, that the data cleaning phase is the phase which takes most time. One way to solve this problem, would be to create a data warehouse repository, that is, a collection of data warehouse databases, which can be used for research purposes, like the UCI Machine Learning Repository[1]. Due to the different nature of data warehouse data, in comparison to the traditional case-based datasets, a number of problems exist with creating such a repository:

- The data warehouse cannot be stored as a single text file. Furthermore, the storage method should not be DBMS specific, since this would complicate the use of the data warehouses, or may even make it impossible for some users to use the data. A solution may be to store the data warehouse as a star schema, and store each dimension, and the fact table, as separate text files.
- The business domain of a data warehouse is usually complex, so it is difficult to understand what the data represents. This can be solved by making sure the business domain, structure of data warehouse, and attributes of the entire data warehouse are described properly.
- The size of data is usually large, which can cause problems both for the people offering the data, and the people trying to access the data. We do not see any way to avoid this if the research is to be done on a realistic data warehouse.

In the design of our data warehouse, we encountered problems with dynamic dimensions, like the balance of a customer, and various attributes related to this balance. How this is normally modelled and how that influences on the data mining task would be another subject that can be examined.

More generally, it would be nice to have statistics on the data warehouses that are in use today. For instance, the typical size of dimensions, and information about the size of concept hierarchies. With this information it would be easier to analyse the performance of algorithms, compared to a vague guess on how large these normally are.

Another area, which it would be interesting to investigate, is how the user interface to the data mining tools can be modified to accommodate multi-dimensional data sources. We have done some preliminary considerations on this, however, the evaluation thereof is not objective, since we naturally made the user interface like we would prefer. Thus it would be interesting to examine what experienced data miners would like from their user interface in this regard.

A related issue, is the use of meta data for the data mining applications. It would be very advantageous to have a standard for defining this meta data, which could be used by any data mining tool.

We have proposed a number of improvements with regard to decision tree induction, however, we have only evaluated one of them, and even this evaluation should be performed on a more complicated database. Likewise, the remaining improvements should be evaluated. We have only considered decision trees, but there are many other algorithms, which could be

[1]http://www.ics.uci.edu/∼mlearn/MLRepository.html

improved to utilise multi-dimensional data. For instance, the Naive Bayes classifier should be investigated, especially due to its assumption of independence between attributes, which definitely does not hold for attributes belonging to the same concept hierarchy.

Bibliography

[AAD$^+$96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 1996 Int. Conf. Very Large Data Bases*, pages 506–521, September 1996.

[BCF99] J. Bernhard, S. Chaudhuri, and U.. Fayyad. Scalable Classification over SQL Databases. In *Proc. of 15th International Conference on Data Engineering*, 1999. Sydney, Australia.

[Che01] Zhengxin Chen. *Data Mining and Uncertain Reasoning: An Integrated Approach*. John Wiley & Sons, Inc., 2001.

[CS02] S. Chaudhuri and S. Sarawagi. Effecient Evaluation of Queries with Mining Predicates. In *Proc. of 18th International Conference on Data Engineering*, 2002. San Jose, USA.

[Dun03] Margaret H. Dunham. *Data Mining Introductory and Advanced Topics*. Pearson Education, Inc., 2003. ISBN 0-13-088892-3.

[Fu96] Yongjian Fu. *Discovery of Multi-Level Rules from Large Databases*. PhD thesis, Simon Fraser University, 1996.

[Gro] The Open Group. The Open Group Base Specification Issue 6, IEEE Std 1003.1, 2003 Edition.
http://www.opengroup.org/onlinepubs/007904975/basedefs/xbd_chap03.html.

[Han98] Jiawei Han. Towards on-line analytical mining in large databases. *ACM SIGMOD Record*, 27(1):97–107, March 1998. ISBN 0163-5808.

[HK01] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2001. ISBN 1-55860-489-8.

[Inm02] W. H. Inmon. *Building the Data Warehouse*. Wiley Computer Publishing, 3rd edition, 2002. ISBN 0-471-08130-2.

[IRBS99] W. H. Inmon, Ken Rudin, Christopher K. Buss, and Ryan Sousa. *Data Warehouse Performance*. Wiley Computer Publishing, 1999. ISBN 0-471-29808-5.

[Jen01] Peter Jensen. Knowledge Discovery and Tests of Available Tools. Technical report, Aalborg University, 2001.

[Kim96] Ralph Kimball. *The Data Warehouse Toolkit.* John Wiley & Sons, Inc, 1996. ISBN 0-471-15337-0.

[Lu97] Yijun Lu. Concept Hierarchy in Data Mining: Specification, Generation and Implementation. Master's thesis, Simon Fraser University, December 1997.

[Mit97] Tom M. Mitchell. *Machine Learning.* WCB/McGraw-Hill, 1997. ISBN 0-07-042807-7.

[MWG+97] Kamber. M, L. Winstone, W. Gong, S. Cheng, and J. Han. Generalization and decision tree induction: Efficient classification in data mining. In *Proc. of 1997 Int. Workshop Research Issues on Data Engineering (RIDE'97)*, pages 111–120, April 1997.

[NBCF01] A. Netz, J. Bernhardt, S. Chaudhuri, and U. Fayyad. Integrating Data Mining with SQL Databases: OLD DB for Data Mining. In *Proc. of 17th International Conference on Data Engineering*, 2001. Heidelberg, Germany.

[Pal00] Themistoklis Palpanas. Knowledge discovery in data warehouses. *ACM SIGMOD Record*, 29(3), September 2000.

[PHP+01] Helen Pinto, Jiawei Han, Jian Pei, Ke Wang, Qiming Chen, and Umeshwar Dayal. Multi-dimensional sequential pattern mining. In *Proceedings of the tenth international conference on Information and Knowledge Management,Atlanta, Georgia, USA*, 2001. ISBN 1-58113-436-3.

[Pin01] Helen Pinto. Multi-dimensional Sequential Pattern Mining. Master's thesis, Simon Fraser University, April 2001.

[PJ01] Torben B. Pedersen and Christian S. Jensen. Multidimensional database technology. *IEEE Computer*, 34:40–46, 2001. ISSN 0018-9162.

[Ree] Michael Reed. A Definition of Data Warehousing. http://www.intranetjournal.com/features/datawarehousing.html.

[SKS02] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill Higher Education, 4th edition, 2002. ISBN 0-07-112268-0.

[Tar98] Yin Jenny Tarn. Datacube: Its Implementation and Application in OLAP Mining. Master's thesis, Simon Fraser University, September 1998.

Appendix A

Microsoft SQL Server

This appendix describes some of the non-standard SQL functions and operations we have used in our handling of data.

A.1 Functions

This section is based on the help files which are included in the Microsoft SQL Server package. These are also available at http://msdn.microsoft.com (search for "Transact-SQL Reference").

A.1.1 CAST

(data type given in *date_type*) CAST(*expression* AS *data_type*)

expression Any valid expression.

date_type The data type which the *expression* should be converted to.

Example:

```
> select cast('2003-02-03' as datetime)
2003-02-03 00:00:00.000
```

A.1.2 CONVERT

(data type given in *date_type*) CONVERT(*date_type*[(length)], *expression*[,*style*])

date_type The data type which the *expression* should be converted to.

length Optional parameter indicating the length of the data type (if applicable).

expression Any valid expression.

style Optional parameter specifying the style of the date format when expression is of type datetime or smalldatetime. Likewise the style of the string when converting data of type float, real, money, or smallmoney[1].

There are a number of styles which can be used for the date format, however, we will only describe the ones used in our data analysis scripts. They are listed below:

Style	Format
108	hh:nn:ss
112	yyyymmdd

Where y indicates year, m the month, d the day, h the hour, n the minute, and s the second. The number of occurences of these letters specify the number of characters used in the date format.

For instance, converting the date 2003-02-03 17:23 to a string using style 108 will result in 17:23:00, whereas using style 112 will result in 20030203.

Example:

```
> select convert(nvarchar,'2003-02-03 17:23',112)
2003-02-03 17:23
> select convert(nvarchar,cast('2003-02-03 17:23' as datetime),112)
20030203
```

The first example shows how not to convert a date when you specify it as a character string. Since the expression is already a string, it is simply returned as it is, since the target data type is a string. In the second example the date in character format is first converted to datetime data type, and then CONVERT is used to convert it to string using the specified style.

A.1.3 DATEADD

datetime|smalldatetime DATEADD(*datepart*,*number*,*date*)

datepart The part of the date that a value is added to. The value of this parameter must be in the set { Year, quarter, Month, dayofyear, Day, Week, Hour, minute, second, millisecond } or one of the abbreviated forms of the values in the set[2].

number The value added to *date* using *datepart*, if it is not an integer, the value is rounded down.

date The date that *value* is added to. The parameter must be of type datetime or smalldatetime, or a string in date format.

The return date type of the function depends on the date type of the *date* parameter. It returns in smalldatetime if *date* has type smalldatetime, otherwise the return type is datetime.

Example:

[1]these will not be described

[2]Not described here since we will use the complete word to increase readability

```
> select dateadd(second,900000011,'1970-01-01')
1998-07-09 16:00:11.000
```

Which incidently corresponds the epoch date value 900000011 converted to a readable date value.

A.1.4 DATEDIFF

integer DATEDIFF(*datepart*,*startdate*,*enddate*)

datepart The part of the date which the result value is returned in. The possible values are described in section A.1.3. Note that this does not mean that the difference is only done for the datepart, it is done for the complete date and then returned using datepart. For instance, the difference between 1999-02-03 and 2003-02-03, when Month is selected as datepart, is not 0, it is 48.

datestart The beginning date for the comparison, this must either be of type datetime or smalldatetime, or a string in date format.

dateend The ending date for the comparison, specified as *datestart*.

The difference is calculated by subtracting *startdate* from *enddate*, and the result is returned as a signed integer, which means that if *enddate* < *startdate*, then a negative value is returned. Example:

```
> select datediff(Month,'2003-02-03','1999-02-03')
-48
```

A.1.5 DATEPART

integer DATEPART(*datepart*,*date*)

datepart The part of the date that is to be returned, specified as described in section A.1.3.

date The date to extract the part of date from.

Example:

```
> select datepart(quarter,'2003-02-03')
1
```

A.1.6 DATENAME

nvarchar DATENAME(*datepart*,*date*)

datepart The part of the date that is to be returned, specified as described in section A.1.3.

date The date to extract the part of date from.

This function is equivalent to DATEPART, except the value is returned in character format. If *datepart* is Month, then a value in the set { January, February, . . . , December } will be returned. If *datepart* is weekday, then a value in the set { Monday, Tuesday, . . . , Sunday } will be returned. Otherwise the value returned corresponds to the value returned from DATEPART, with the exception of the date type.

Example:

```
> select datepart(Month,'2003-02-03')
February
```

A.1.7 SET DATEFIRST

SET DATEFIRST *number*

This is technically not a function, however... This function is used to define which weekday is the first in a week, it is defined from the following table:

Value	First weekday
1	Monday
2	Tuesday
⋮	⋮
7	Sunday

The value of DATEFIRST affects the values returned by DATEPART and DATENAME with respect to weekday.

Appendix B

Data Preprocessing, SQL statements

B.1 Table Definitions

In this appendix all the SQL commands we have used on the original database are documented.

B.1.1 Original Tables

```
CREATE TABLE [dbo].[coffee] (
[user_id] [int] NOT NULL ,
[subscriber_since] [smalldatetime] NULL ,
[date] [int] NULL
)

CREATE TABLE [dbo].[employee_type] (
[employee_type_id] [int] NOT NULL ,
[description] [nvarchar] (20) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
[free_coffee] [int] NULL
)

CREATE TABLE [dbo].[members] (
[user_id] [int] NOT NULL ,
[active] [int] NOT NULL ,
[aargang] [int] NOT NULL ,
[debt] [float] NOT NULL ,
[board_debt] [float] NOT NULL ,
[last_warned] [int] NOT NULL ,
[first_warning] [int] NOT NULL ,
[advance] [float] NOT NULL ,
[undos] [int] NOT NULL ,
[total_undos] [int] NOT NULL ,
[employee] [int] NOT NULL ,
[balance] AS ([advance] - [debt] - [board_debt]) ,
[first_payment] [int] NULL ,
[last_payment] [int] NULL ,
[first_purchase] [int] NULL ,
[last_purchase] [int] NULL ,
[never_used_system] [tinyint] NULL
)

CREATE TABLE [dbo].[paid_ansat_kaffe] (
[date] [int] NULL
)

CREATE TABLE [dbo].[payments] (
[user_id] [int] NOT NULL ,
[date] [int] NOT NULL ,
[amount] [float] NOT NULL
)

CREATE TABLE [dbo].[prices] (
[product_id] [int] NOT NULL ,
[price] [float] NOT NULL ,
[date_start] [int] NOT NULL
)

CREATE TABLE [dbo].[products] (
[product_id] [int] NOT NULL ,
[name] [nvarchar] (20) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
[price] [float] NULL ,
[active] [int] NULL
)

CREATE TABLE [dbo].[sales] (
[user_id] [int] NOT NULL ,
[product_id] [int] NULL ,
[date] [int] NULL ,
[price] [float] NULL ,
[paid_for] [int] NULL ,
[money] [float] NULL
)
```

B.1.2 Dimensional and Helper Tables

```
CREATE TABLE [dbo].[datedim] (
[dateid] [int] NOT NULL ,
[year] [int] NOT NULL ,
[semester] [nvarchar] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[quarter] [int] NOT NULL ,
[month] [int] NOT NULL ,
[month_name] [nvarchar] (10) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[week] [int] NOT NULL ,
[weekday] [int] NOT NULL ,
[weekday_name] [nvarchar] (10) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[dayofmonth] [int] NOT NULL ,
[dayofyear] [int] NOT NULL ,
[date] [smalldatetime] NOT NULL ,
[studyyear] [int] NOT NULL ,
[studysemester] [nvarchar] (3) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL
)

CREATE TABLE [dbo].[epoch_mapping] (
[epoch] [int] NOT NULL ,
[ts] [datetime] NOT NULL ,
[dateid] [int] NULL ,
[timeid] [int] NULL
)

CREATE TABLE [dbo].[member_correction] (
[user_id] [int] NOT NULL ,
[correction] [float] NOT NULL
)

CREATE TABLE [dbo].[member_day] (
[user_id] [int] NOT NULL ,
[dateid] [int] NOT NULL ,
[balance] [float] NOT NULL ,
[active] [tinyint] NULL ,
[warned] [tinyint] NOT NULL ,
[days_warned] [smallint] NOT NULL ,
[blocked] [tinyint] NOT NULL ,
[days_blocked] [smallint] NOT NULL ,
[days_till_block] [smallint] NOT NULL
)

CREATE TABLE [dbo].[member_day_unc] (
[user_id] [int] NOT NULL ,
[dateid] [int] NOT NULL ,
[balance] [float] NOT NULL ,
[active] [tinyint] NULL ,
[warned] [tinyint] NOT NULL ,
[days_warned] [smallint] NOT NULL ,
[blocked] [tinyint] NOT NULL ,
[days_blocked] [smallint] NOT NULL ,
[days_till_block] [smallint] NOT NULL
)
```

```
CREATE TABLE [dbo].[memberdim] (
[memberid] [int] NOT NULL ,
[active] [int] NOT NULL ,
[aargang] [int] NOT NULL ,
[semester] [int] NOT NULL ,
[employee_type] [nvarchar] (20) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[free_coffee] [nvarchar] (1) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[advance] [float] NOT NULL ,
[debt] [float] NOT NULL ,
[board_debt] [float] NOT NULL ,
[balance] AS ([advance] - [debt] - [board_debt]) ,
[undos] [int] NOT NULL ,
[never_used_system] [tinyint] NOT NULL
)

CREATE TABLE [dbo].[productdim] (
[productid] [int] NOT NULL ,
[original_id] [int] NOT NULL ,
[name] [nvarchar] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[price] [float] NOT NULL ,
[active] [int] NOT NULL ,
[MainClass] [nvarchar] (20) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[Class] [nvarchar] (20) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
[SubClass] [nvarchar] (20) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL
)

CREATE TABLE [dbo].[sales_cmp] (
[user_id] [int] NOT NULL ,
[product_id] [int] NOT NULL ,
[date] [int] NOT NULL ,
[n_units] [int] NULL ,
[total_price] [float] NULL ,
[unit_price] [float] NULL ,
[balance] [float] NULL ,
[balance_after] AS ([balance] - [total_price]) ,
[dateid] [int] NULL ,
[timeid] [int] NULL ,
[balance_uncorrected] [float] NULL ,
[balance_after_uncorrected] AS ([balance_uncorrected] - [total_price])
)

CREATE TABLE [dbo].[sales_day] (
[dateid] [int] NOT NULL ,
[product_id] [int] NOT NULL ,
[n_units] [int] NOT NULL ,
[total_price] [int] NOT NULL ,
[active_members_day] [int] NULL ,
[active_members_week] [int] NULL ,
[active_members_month] [int] NULL ,
[active_members_semester] [int] NULL ,
[active_members_year] [int] NULL ,
[active_members_rweek] [int] NULL ,
[active_members_rmonth] [int] NULL ,
[active_members_syear] [int] NULL ,
[active_members_ssemester] [int] NULL
)

CREATE TABLE [dbo].[timedim] (
[timeid] [int] IDENTITY (1, 1) NOT NULL ,
[timeofday] [nvarchar] (20) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
[mins] [int] NULL ,
[hour] [int] NULL ,
[time] [datetime] NOT NULL
)
```

B.2 Misc. Checks and Fixes

```
-- check undos
select count(1) from members where total\_undos > 0 or undos > 0;
select count(1) from members where total\_undos != undos;

-- changes to Members
declare @min_date integer
set @min_date = (select min(date) from sales where user_id in
                         (select user_id from members where aargang=1999))
select distinct user_id from sales
where user_id in (select user_id from members where aargang=0)
and date < @min_date

declare @min_date integer
set @min_date = (select min(date) from sales where user_id in
                          (select user_id from members where aargang=1999))
update members set aargang=2000 where aargang=0 and members.never_used_system=0
and user_id not in (
select distinct user_id from sales
where date < @min_date
and user_id in (select user_id from members where aargang=0)
)
update members set aargang=2001 where aargang=1

update members set first_payment=agg.min,last_payment=agg.max
from (select user_id,max(date) as max, min(date) as min from payments p
group by user_id) agg where agg.user_id=members.user_id
```

```
update members set first_purchase=agg.min,last_purchase=agg.max
from (select user_id,max(date) as max, min(date) as min from sales_cmp s
group by user_id) agg where agg.user_id=members.user_id

select aargang,state,count(1) as count from (
select m.user_id, m.balance - p.sum + s.sum as diff,m.aargang,
       case when (m.balance - p.sum + s.sum > -20
            and m.balance - p.sum + s.sum < 20) then 'ok' else 'err'
       end as state
from (select user_id,sum(amount) as sum from payments group by user_id) p,
     (select user_id,sum(total_price) as sum from sales_cmp
      group by user_id) s, members m
where p.user_id=m.user_id and s.user_id=m.user_id
) total
group by aargang,state
order by aargang

update members set never_used_system=1
update members set never_used_system=0
where user_id in (select user_id from sales)
```

B.3 Data Transformation

B.3.1 General Functions

```
drop function fn_replace_if_null
go
create function fn_replace_if_null(@value float,@replace_value float = 0)
returns float as
begin
    if @value is null return @replace_value
    return @value
end
go

drop function fn_min
go
create function fn_min(@val1 int, @val2 int) returns int as
begin
if (@val1 > @val2) return @val2
return @val1
end
go

drop function fn_max
go
```

```
create function fn_max(@val1 int, @val2 int) returns int as
begin
if (@val1 < @val2) return @val2
return @val1
end
go
```

B.3.2 Epoch and Date Handling Functions

```
drop function fn_epoch_to_datetime
go
create function fn_epoch_to_datetime(@epoch int) returns datetime as
begin
    return (dateadd(second,@epoch,'1970-01-01'))
end
go

drop function fn_datetime_to_epoch
go
create function fn_datetime_to_epoch(@dt datetime) returns int as
begin
    return (datediff(second,'1970-01-01',@dt))
end
go

drop function fn_datetime_to_semester
go
create function fn_datetime_to_semester(@dt datetime) returns nvarchar(12) as
begin
    declare @month int
    set @month = datepart(month,@dt)
    return case
               when @month between 2 and 5 then 'spring'
               when @month between 9 and 12 then 'fall'
               when @month = 1 or @month = 6 then 'exam'
               when @month = 7 or @month = 8 then 'summer break'
               else 'Undefined'
           end
end
go

drop function fn_datetime_to_timeofday
go
create function fn_datetime_to_timeofday(@dt datetime) returns nvarchar(12) as
begin
```

```
    declare @hour int
    set @hour = datepart(hour,@dt)
    return case
                when @hour between 0 and 5 then 'night'
                when @hour between 6 and 10 then 'morning'
                when @hour between 11 and 13 then 'noon'
                when @hour between 14 and 17 then 'afternoon'
                when @hour between 18 and 23 then 'evening'
                else 'Undefined'
           end
end
go

drop function date_to_studyyear
go
create function date_to_studyyear(@dt datetime) returns int as
begin
    return case
           when datepart(month,@dt) between 9 and 12 then datepart(year,@dt)
           else datepart(year,@dt) - 1
           end
end
go

drop function date_to_studysemester
go
create function date_to_studysemester(@dt datetime) returns nchar(3) as
begin
    declare @res_sem nchar(1),
            @res_year nchar(2),
            @month int,
            @year int
    if @dt is null return null

    set @year=datepart(year,@dt)
    set @month=datepart(month,@dt)

    set @res_sem='F'
    if @month between 9 and 12 or @month=1 set @res_sem='E'

    if (@month=1) set @year = @year - 1
    set @res_year=substring(cast(@year as nchar(4)),3,2)

    return @res_sem + @res_year
end
go
```

```
drop procedure p_fill_dates
go
create procedure p_fill_dates(@from datetime, @to datetime) as
begin
    set nocount on
    declare @date datetime,
            @id int
    set @date = (convert(nvarchar,@from,112))
    set @id = 1
    while @date <= @to
    begin
        insert into datedim(dateid,date,year,quarter,month,month_name,week,
                          weekday,weekday_name,dayofyear,dayofmonth,semester,
                          studyyear,studysemester)
        values (@id,@date,datepart(year,@date),datepart(quarter,@date),
            datepart(month,@date),datename(month,@date),datepart(week,@date),
            datepart(weekday,@date),datename(weekday,@date),
            datepart(dayofyear,@date),
            substring(convert(nvarchar,@date,3),1,2),
            dbo.fn_datetime_to_semester(@date),
            dbo.date_to_studyyear(@date),dbo.date_to_studysemester(@date))
        set @date = dateadd(day,1,@date)
        set @id = (@id + 1)
    end
end
go
```

B.3.3 Member Functions

```
DROP PROC sp_foreach_member_do
GO
DROP PROC p_foreach_member_do
GO
CREATE PROC p_foreach_member_do(@all_users int = 1,@proc nvarchar(100),
                                @params nvarchar(100) = '') as
BEGIN
DECLARE @cmd nvarchar(255)
DECLARE @user_id integer
IF @all_users = 0
BEGIN
  DECLARE uids CURSOR LOCAL STATIC FOR
  SELECT DISTINCT user_id FROM members
  WHERE never_used_system = 0
  ORDER BY user_id
END
ELSE
```

```
BEGIN
  DECLARE uids CURSOR LOCAL STATIC FOR
  SELECT DISTINCT user_id FROM members
  ORDER BY user_id
END

OPEN uids
FETCH NEXT FROM uids INTO @user_id
WHILE @@FETCH_STATUS = 0
BEGIN
    SET @cmd = @proc + ' ' + CAST(@user_id as nvarchar(10))
    IF (@params IS NOT NULL AND @params != '') SET @cmd = @cmd + ' ,' + @params
    EXEC (@cmd)
    FETCH NEXT FROM uids INTO @user_id
END

CLOSE uids
DEALLOCATE uids
END


-- balance = debt + board_debt - advance

DROP FUNCTION acc_payments
GO
CREATE  FUNCTION acc_payments (@uid integer,@at integer) RETURNS float AS
BEGIN
DECLARE @res float
SET @res = (select sum(amount) from payments where user_id = @uid
            and date < @at)
if @res is null return 0
return @res
END
GO

DROP FUNCTION acc_sales
GO
CREATE  FUNCTION acc_sales (@uid integer,@at integer) RETURNS float AS
BEGIN
DECLARE @res double float
SET @res = (select sum(total_price) from sales_cmp
            where user_id = @uid and date < @at)
IF @res is null return (0)
return (@res)
END
GO
```

```
DROP FUNCTION curr_money
GO
CREATE FUNCTION curr_money (@uid integer,@at integer) RETURNS float AS
BEGIN
return (select dbo.acc_payments(@uid,@at) - dbo.acc_sales(@uid,@at))
END
GO
```

B.3.4 Transformation

```
drop procedure p_create_epoch_mapping
go
create procedure p_create_epoch_mapping as
begin
    declare @min_date datetime,
            @max_date datetime
    truncate table epoch_mapping
    insert into epoch_mapping(epoch,ts)
    select distinct date,dbo.fn_epoch_to_datetime(date)
    from (
        select date from coffee
        union
        select date from paid_ansat_kaffe
        union
        select last_warned from members where last_warned > 0
        union
        select first_warning from members where first_warning > 0
        union
        select date from payments
        union
        select date_start from prices
        union
        select date from sales
    ) total order by date

    set @min_date = (select min(ts) from epoch_mapping)
    set @max_date = (select max(ts) from epoch_mapping)
    truncate table datedim
    exec dbo.p_fill_dates @min_date,@max_date

    truncate table timedim
    insert into timedim(time)
    select distinct convert(nvarchar,dateadd(second,epoch,'1970-01-01'),108)
        as tim from epoch_mapping order by tim
    update epoch_mapping set dateid=dd.dateid
    from datedim dd
    where convert(nvarchar,epoch_mapping.ts,112) = convert(nvarchar,dd.date,112)
    update epoch_mapping set timeid=td.timeid
    from timedim td
    where convert(nvarchar,epoch_mapping.ts,108) = convert(nvarchar,td.time,108)

    -- fill in derived time dimension attributes
    set datefirst 1 -- monday is represented by 1, and so forth
    update timedim set hour=datepart(hour,time),mins=datepart(minute,time),
        timeofday=dbo.fn_datetime_to_timeofday(time)
end
go

exec dbo.p_create_epoch_mapping

insert into sales_cmp(user_id,product_id,date,n_units,unit_price,total_price)
select user_id,product_id,date,count(price),sum(price), avg(price)
from sales group by date,user_id,product_id order by date

update sales_cmp set dateid=e.dateid,timeid=e.timeid
from epoch_mapping e where sales_cmp.date=e.epoch

update sales_cmp set balance=dbo.curr_money(user_id,date)
```

B.3.5 Historical Member Data

```
drop proc p_update_md
go
create proc p_update_md(@user_id int,@use_correction int = 0) as
begin
    declare @first_date int,
            @first_dateid int,
            @table nvarchar(15),
            @query nvarchar(255)

    set @first_date = (select dbo.fn_min(first_payment,first_purchase)
                       from members where user_id=@user_id)
    set @first_dateid = (select dateid from epoch_mapping
                         where epoch=@first_date)

    if @use_correction = 0 set @table = 'member_day_unc'
    else set @table = 'member_day'

    set @query = 'insert into ' + @table
         + '(user_id,dateid,balance,active,warned,blocked,days_warned,'
         + 'days_blocked,days_till_block) select '
         + cast(@user_id as nvarchar)
```

```
     + ',dateid,0,0,0,0,0,0,0 from datedim where dateid < '
     + cast(@first_dateid as nvarchar)
exec(@query)

declare @balance float,
        @day_payments float,
        @day_purchases float,
        @day_dateid int,
        @day_active int,
        -- warn/block limit variables
        @wlim int,
        @blim int,
        @change_dateid int,
        @in_warn int,
        @in_block int,
        @days_warned int,
        @days_blocked int,
        @days_till_block int
set @wlim = -150
set @blim = -250
set @change_dateid = 1776

declare days cursor local static for
select dateid from datedim where dateid >= @first_dateid order by dateid

if @use_correction = 0 set @balance = 0
else set @balance = (select correction from member_correction
                     where user_id=@user_id)
set @in_warn = 0
set @in_block = 0
set @days_warned = 0
set @days_blocked = 0
set @days_till_block = 0

open days
fetch next from days into @day_dateid
while @@FETCH_STATUS = 0
begin
    -- calculate balance
    set @day_payments = (select sum(amount)
                         from payments p, epoch_mapping em
                         where p.user_id=@user_id and em.epoch=p.date
                         and em.dateid=@day_dateid)
    set @day_purchases = (select sum(total_price) from sales_cmp
                          where user_id=@user_id and dateid=@day_dateid)
    set @day_active = (select 1 where exists (select 1 from sales_cmp
                          where user_id=@user_id and dateid=@day_dateid))
```

```
if @day_active is null set @day_active = 0
else set @day_active = 1
if @day_payments is null set @day_payments = 0
if @day_purchases is null set @day_purchases = 0
set @balance = @balance + @day_payments - @day_purchases

-- general b/w
if @day_dateid >= @change_dateid
begin
    set @wlim = 0
    set @blim = -50
end

-- calculate b/w

if @balance < @wlim
begin
    if @in_warn = 0 and @in_block = 0
    begin
        set @in_warn = 1
        set @days_till_block = 14
        set @days_warned = 0
    end

    if @in_warn = 1
    begin
        set @days_warned = @days_warned + 1
        set @days_till_block = @days_till_block - 1
    end

    if @in_block = 1
    begin
        set @days_blocked = @days_blocked + 1
    end

    if @in_block = 0 and (@days_warned = 14 or (@balance < @blim))
    begin
        set @in_warn = 0
        set @in_block = 1
        set @days_blocked = 0
        set @days_till_block = 0
        set @days_warned = 0
    end
end
else
begin
    set @in_warn = 0
```

```
            set @in_block = 0
            set @days_blocked = 0
            set @days_warned = 0
            set @days_till_block = 0
        end

        -- generating a single dynamic query would look simpler, but for
        -- performance reasons we duplicate the inserts for each table
        if @use_correction = 0
        begin
            insert into member_day_unc(user_id,dateid,balance,active,
                                       warned,blocked,
                                       days_warned,days_blocked,days_till_block)
             values(@user_id,@day_dateid,@balance,@day_active,
                    @in_warn,@in_block,
                    @days_warned,@days_blocked,@days_till_block)
        end
        else
        begin
            insert into member_day(user_id,dateid,balance,active,
                                       warned,blocked,
                                       days_warned,days_blocked,days_till_block)
             values(@user_id,@day_dateid,@balance,@day_active,
                    @in_warn,@in_block,
                    @days_warned,@days_blocked,@days_till_block)
        end

        fetch next from days into @day_dateid
    end

    close days
    deallocate days
end
go

drop proc p_update_correction
go
create proc p_update_correction as
begin
    declare @last_aargang int
    set @last_aargang = 1996

    truncate table member_correction

    insert into member_correction(user_id,correction)
    select user_id,0 from members
    /* We do not process non-purchasing customers, so skip this part
    update member_correction set correction=m.balance - p.sum
    from members m,
         (select user_id,sum(amount) as sum from payments group by user_id) p
    where m.user_id=member_correction.user_id
    and p.user_id=member_correction.user_id
    and m.aargang <= @last_aargang
    and not exists (select 1 from sales_cmp
    where user_id=member_correction.user_id)
    */

    update member_correction set correction=m.balance + s.sum
    from members m,
         (select user_id,sum(total_price) as sum from sales_cmp
          group by user_id) s
    where m.user_id=member_correction.user_id
    and s.user_id=member_correction.user_id
    and m.aargang <= @last_aargang
    and not exists (select 1 from payments
                    where user_id=member_correction.user_id)

    update member_correction
    set correction=m.balance - (dbo.fn_replace_if_null(p.sum,0)
                                - dbo.fn_replace_if_null(s.sum,0))
    from (select user_id,sum(amount) as sum from payments group by user_id) p,
         (select user_id,sum(total_price) as sum from sales_cmp
          group by user_id) s,
         members m
    where p.user_id=member_correction.user_id
    and s.user_id=member_correction.user_id
    and m.user_id=member_correction.user_id
    and m.never_used_system=0 and m.aargang <= @last_aargang
end

drop procedure p_complete_md_update
go
create procedure p_complete_md_update as
begin
    set nocount on
    truncate table member_day_unc
    truncate table member_day
    exec p_foreach_member_do 0,'p_update_md','0'
    exec p_update_correction
    exec p_foreach_member_do 0,'p_update_md','1'
end
go
```

```
exec p_complete_md_update
go

update sales_cmp set balance_uncorrected=balance
update sales_cmp set balance=balance_uncorrected + mc.correction
from member_correction mc
where mc.user_id=sales_cmp.user_id
```

B.3.6 Historical Sales Data

```
insert into sales_day(dateid,product_id,n_units,total_price)
select dateid,product_id,sum(n_units),sum(total_price)
from sales_cmp
group by dateid,product_id
order by dateid

-- calculate active members...
update sales_day set active_members_day=agg.cnt from
(select count(distinct user_id) as cnt,dateid from sales_cmp
 group by dateid) agg
where sales_day.dateid=agg.dateid

update sales_day set active_members_week=agg.cnt
from datedim dd,
     (select count(distinct user_id) as cnt,dd.year,dd.month,dd.week
      from sales_cmp s, datedim dd where s.dateid=dd.dateid
      group by dd.year,dd.month,dd.week) agg
where dd.dateid=sales_day.dateid and dd.year=agg.year and dd.month=agg.month
and dd.week=agg.week

update sales_day set active_members_month=agg.cnt
from datedim dd,
     (select count(distinct user_id) as cnt,dd.year,dd.month
      from sales_cmp s, datedim dd where s.dateid=dd.dateid
      group by dd.year,dd.month) agg
where dd.dateid=sales_day.dateid and dd.year=agg.year and dd.month=agg.month

update sales_day set active_members_year=agg.cnt
from datedim dd,
     (select count(distinct user_id) as cnt,dd.year
      from sales_cmp s, datedim dd where s.dateid=dd.dateid
      group by dd.year) agg
where dd.dateid=sales_day.dateid and dd.year=agg.year

update sales_day set active_members_semester=agg.cnt
from datedim dd,
     (select count(distinct user_id) as cnt,dd.year,dd.semester
      from sales_cmp s, datedim dd where s.dateid=dd.dateid
      group by dd.year,dd.semester) agg
where dd.dateid=sales_day.dateid and dd.year=agg.year
and dd.semester=agg.semester

update sales_day set active_members_ssemester=agg.cnt
from datedim dd,
     (select count(distinct user_id) as cnt,dd.studysemester
      from sales_cmp s, datedim dd where s.dateid=dd.dateid
      group by dd.studysemester) agg
where dd.dateid=sales_day.dateid and dd.studysemester=agg.studysemester

update sales_day set active_members_syear=agg.cnt
from datedim dd,
     (select count(distinct user_id) as cnt,dd.studyyear
      from sales_cmp s, datedim dd where s.dateid=dd.dateid
      group by dd.studyyear) agg
where dd.dateid=sales_day.dateid and dd.studyyear=agg.studyyear

update sales_day set active_members_rweek=agg.cnt
from (select dd2.dateid,count(distinct user_id) as cnt
      from sales_cmp s, datedim dd, datedim dd2
      where s.dateid=dd.dateid and
      dd.date between dateadd(day,-3,dd2.date) and dateadd(day,3,dd2.date)
      group by dd2.dateid
) agg
where sales_day.dateid=agg.dateid

update sales_day set active_members_rmonth=agg.cnt
from (select dd2.dateid,count(distinct user_id) as cnt
      from sales_cmp s, datedim dd, datedim dd2
      where s.dateid=dd.dateid and
      dd.date between dateadd(day,-15,dd2.date) and dateadd(day,15,dd2.date)
      group by dd2.dateid
) agg
where sales_day.dateid=agg.dateid
```