Design and implementation of a database client application for inserting, modifying, presentation and export of bitemporal personal data

Diploma Thesis

Tobias Schlaginhaufen
Zürich, Switzerland
Student ID: 99-905-721

Professor: Prof. Klaus R. Dittrich
Supervisor: Boris Glavic
Date of Submission: April 30, 2007
Abstract

There exists only little support for temporal data in conventional relational databases, although many applications require temporal or even bitemporal storage of data. This thesis describes the implementation of a bitemporal database and a pursuant client application, for managing study subjects of a longitudinal etiological study of adjustment and mental health.

Designing a bitemporal database on top of a relational database model involves the dilemma of using the well-established storage organization and query evaluation techniques of relational databases and accept a certain redundant storage of data or time-normalizing the data model and avoid redundancy but incur a degenerated relational model, which is complex, difficult to handle and may degrad the performance of a relation database system. We present an approach striking the balance between these two extremes.

Temporale Daten werden von konventionellen relationalen Datenbanksystemen noch kaum unterstützt, obwohl viele Anwendungen die temporale oder bitemporale Speicherung von Daten benötigen. Im Rahmen dieser Diplomarbeit wird eine bitemporale Datenbank und eine dazugehörige Client-Anwendung erstellt, um Probanden einer ätiologische Langzeitstudie zur psychischen Gesundheit zu verwalten.

Das Entwicklung einer bitemporalen Datenbank auf einem relationalen Datenmodell bringt das Dilemma mit sich, entweder die etablierte Speicherorganisation und Anfragetechniken von relationalen Datenbankystemen zu nutzen und gewisse Redundanz bei der Speicherung in Kauf zu nehmen, oder das Datenmodell temporal zu normalisieren, Redundanz zu vermeiden, jedoch ein degeneriertes Relationenmodell in Kauf zu nehmen, welches komplex zu handhaben ist und die Performance-Eigenschaften von relationalen Datenbanksystemen schlecht ausnützt. Wir präsentieren einen Ansatz, der für die gegebene Anwendung einen Mittelweg zwischen diesen beiden Extremen findet.
Acknowledgements

Acknowledgements...
Contents

1 Introduction 4
  1.1 Structure of Thesis .................................... 6

2 sesam 8
  2.1 sesamDB .................................................. 9
  2.2 Subject Management .................................... 10
    2.2.1 Traceability ........................................ 10
    2.2.2 Time-varying information ......................... 10

3 Temporal databases 12
  3.1 Application .............................................. 12
  3.2 Definitions and Taxonomy ............................... 14
    3.2.1 Time-varying and static attributes ................ 14
    3.2.2 Concepts of time ................................... 14
      3.2.2.1 Valid Time ..................................... 15
      3.2.2.2 Transaction Time ............................... 15
      3.2.2.3 User-Defined Time ................................ 16
    3.2.3 Kinds of databases .................................. 16
      3.2.3.1 Static Database ................................ 16
      3.2.3.2 Static Rollback Database ...................... 17
      3.2.3.3 Historical Database ............................ 17
      3.2.3.4 Temporal database ............................... 17
    3.2.4 Intervals in temporal databases .................... 18
    3.2.5 Attribute versus tuple timestamping ............... 18
    3.2.6 Synchronous attributes ............................. 19
    3.2.7 Time dependence .................................... 19
    3.2.8 Time normalization .................................. 20
  3.3 Research status .......................................... 21
    3.3.1 Temporal Query Languages .......................... 22
      3.3.1.1 TempSQL ........................................ 22
      3.3.1.2 TSQL ........................................... 22
## 3.3.1.3 HSQL

23

3.3.1.4 TSQL2

23

3.3.2 Temporal joins

24

3.3.3 Temporal indexing techniques

25

### 3.4 Approaches

25

#### 3.4.1 Concepts of “now” and “infinity”

26

#### 3.4.2 Temporal Data Models

26

##### 3.4.2.1 Trade-off between query performance, complexity and redundancy

27

#### 3.4.3 Separation of history relation

28

## 4 Our approach

29

### 4.1 Our “now” concept

29

### 4.2 Keys

30

### 4.3 Time normalization

31

#### 4.3.1 Valid time domain

31

#### 4.3.2 Transaction time domain

32

#### 4.3.3 Bitemporal domain

33

### 4.4 Intermediate layer

34

### 4.5 Interval overlaps

35

#### 4.5.1 The “main address” problem

37

## 5 Implementation

39

### 5.1 Requirements

39

#### 5.1.1 Given systems

40

#### 5.1.2 Client requirements

40

#### 5.1.3 Database requirements

41

### 5.2 Architecture

41

### 5.3 Database

42

#### 5.3.1 Common conventions

44

#### 5.3.2 Static relations

45

#### 5.3.3 Time-varying relations

47

#### 5.3.4 Views

48

##### 5.3.4.1 The “reintegration” view

48

##### 5.3.4.2 History views

50

#### 5.3.5 Temporal Constraints

51

### 5.4 Java Client

54

#### 5.4.1 Database interface

54

##### 5.4.1.1 Hibernate and the persistency classes

54

##### 5.4.1.2 Relationships with Hibernate

55

#### 5.4.2 GUI classes

56
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.2.1 GUI composites</td>
<td>57</td>
</tr>
<tr>
<td>5.4.2.2 Edit dialogs</td>
<td>58</td>
</tr>
<tr>
<td>5.4.2.3 Search dialogs</td>
<td>58</td>
</tr>
<tr>
<td>5.4.2.4 History dialogs</td>
<td>59</td>
</tr>
<tr>
<td>5.5 Tests</td>
<td>59</td>
</tr>
<tr>
<td>6 Summary and Conclusions</td>
<td>61</td>
</tr>
<tr>
<td>6.1 Summary</td>
<td>61</td>
</tr>
<tr>
<td>6.2 Conclusions</td>
<td>61</td>
</tr>
<tr>
<td>6.3 Outlook</td>
<td>61</td>
</tr>
<tr>
<td>6.3.1 Use of supporting tools and frameworks</td>
<td>61</td>
</tr>
<tr>
<td>6.3.2 Additional functionality</td>
<td>62</td>
</tr>
<tr>
<td>A Database schema</td>
<td>63</td>
</tr>
<tr>
<td>B Java class hierarchy</td>
<td>67</td>
</tr>
<tr>
<td>B.1 Package Hierarchies</td>
<td>68</td>
</tr>
<tr>
<td>B.2 Class Hierarchy</td>
<td>68</td>
</tr>
<tr>
<td>B.3 Interface Hierarchy</td>
<td>72</td>
</tr>
<tr>
<td>List of Figures</td>
<td>73</td>
</tr>
<tr>
<td>List of Tables</td>
<td>75</td>
</tr>
<tr>
<td>Bibliography</td>
<td>76</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In this thesis we develop a bitemporal database for a special application domain, namely the subject management for a longitudinal psychology study.

The aim of databases is storing facts of a mini-world in an information system. A lot of facts are not static but they change over time. Facts are not valid forever but have a they are valid in the real world for a certain time period. The frequency of changes ranges from years to milliseconds or even beyond, if we include scientific applications. A name of a person changes very seldom but a share price changes several times during a minute.

A wide range of database applications are temporal in nature and manage time-varying data. Databases storing only the current facts about the mini-world are called snapshot databases while databases which store the course of facts, as they change over time, are called temporal databases.

For temporal facts stored in databases, it is symptomatic that facts in the real world and facts in the database are usually not synchronized. In most applications the facts in the database are updated after the fact has changed in the real world. However facts that are predictable can be updated in the database before they take the value in the real world. Hence a temporal database maintains past, present, and future data [Tansel et al., 1993].

Databases aiming to model two temporal dimensions are called bitemporal databases. One dimension referring to the validity of facts in the real world
### Table 1.1: Bitemporal database example (following [Wikipedia, 2005])

<table>
<thead>
<tr>
<th>Date</th>
<th>What happened in the real world</th>
<th>Database action</th>
<th>What the database shows</th>
<th>VT Begin</th>
<th>VT End</th>
<th>TT Begin</th>
<th>TT End</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 Jan 80</td>
<td>John is born</td>
<td>Nothing</td>
<td>John is not in the database</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 Jan 00</td>
<td>John moves to Basel</td>
<td>Nothing</td>
<td>John is not in the database</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25 Feb 00</td>
<td>John is entered into the database</td>
<td>John is inserted into the database</td>
<td>John lives in Basel</td>
<td>01 Jan 00</td>
<td>infinity</td>
<td>25 Feb 00</td>
<td>now</td>
</tr>
<tr>
<td>20 May 01</td>
<td>John reports that he is going to move to Zürich on 01 Aug 01</td>
<td>Old information about John living in Basel is updated</td>
<td>John lives in Basel</td>
<td>01 Jan 00</td>
<td>infinity</td>
<td>31 Jul 01</td>
<td>20 May 01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New entry for Zürich is inserted</td>
<td>John lives in Zürich</td>
<td>01 Aug 01</td>
<td>infinity</td>
<td>05 Jun 02</td>
<td>now</td>
</tr>
<tr>
<td>01 Aug 01</td>
<td>John moves to Zürich</td>
<td>Nothing</td>
<td>the same as on 20 May 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 Apr 02</td>
<td>John moves to Geneva</td>
<td>Nothing</td>
<td>the same as on 20 May 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>05 Jun 02</td>
<td>John reports that he moved to Geneva on 01 Apr 02</td>
<td>Old information about John living in Zürich is updated</td>
<td>John lives in Zürich</td>
<td>01 Apr 02</td>
<td>infinity</td>
<td>05 Jun 02</td>
<td>now</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New entry for Geneva is inserted</td>
<td>John lives in Geneva</td>
<td>01 Apr 02</td>
<td>infinity</td>
<td>05 Jun 02</td>
<td>now</td>
</tr>
</tbody>
</table>

is called *valid time*, and the other dimension referring to the insert, update or delete time of a fact in the database is called *transaction time*.

The example in table 1.1 shall illustrate the *two* temporal dimensions of some facts about the life of John. Information about John’s domicile have a *valid time* (*VT*) in the real world, namely starting on the day John moves in a new home, and ending on the day John moves out. Consider a database (e.g. the one of the registration office) aiming to store the information about John’s domicile. It is very likely that John does not always report his move to the authorities exactly on the same date he actually moves, since he is probably very busy on that day packing all his belongings and carrying his furniture to the new place. Therefore he will report his new address either some days before of after the move. In order to track the dates an information was entered, modified or deleted in the database there is a second time domain called *transaction time* (*TT*).
CHAPTER 1. INTRODUCTION

Figure 1.1 shows the bitemporal dimensions, valid time and transaction time, of three facts stored in the database. Each version of a fact stored in a bitemporal database can be viewed as a bounded plane in a 2-dimensional space, more precisely it is a plane at least bounded by $TT_{\text{begin}}$ in the transaction time dimension. Optionally it is bounded in the valid time dimension, if the beginning or end of the validity in the real world is known. A plane is bounded by $TT_{\text{end}}$, in case the tuple has become historic.

There are two version about the fact “John lives in Basel” and “John lives in Zürich”. A historic and a currently valid on in each case. There is only one currently valid version about the fact “John lives in Geneva”.

The points of update (the little circles) lie below the 45-line, if the tuple on the database is updated after the validity in the real world started. It is above, if the database is updated before the validity started in the real world. If it lies exactly on the line, the valid time begin and transaction time begin are simultaneous.

1.1 Structure of Thesis

This thesis is structured as follows:

Chapter 2 introduces the background project giving rise to this thesis, the Swiss Etiological Study of Adjustment and Mental Health sesam and the sub-project sesamDB responsible to provide several database applications to the researchers of the sesam study.

The following chapter is about temporal databases. After bringing in definitions and the taxonomy of temporal databases, there is an overview of the research in this area. Different approaches for modelling temporal databases are explained thereafter.

Our approach to model a bitemporal database for the given application is explicate in chapter 4.

A further chapter describes the implementation of database and the software client developed within the scope of this thesis.
Resultant issues will be discussed after this and out-coming conclusions about this practical application of bitemporal databases will complete this exposition.
Chapter 2

sesam

According to World Health Organisation estimates, depression will be the second most important cause of premature death and health impairment by 2020. The sesam research project aims to uncover the complex factors that influence the development of mental health over a person’s lifetime.[SNF, 2006]

sesam stands for Swiss Etiological Study of Adjustment and Mental Health and is a project sponsored by the Swiss National Science Foundation.

The goal of sesam is to

• identify constitutional and protective factors,

• understand combinations in the context of life that conflict with a healthy mental development,

• gain basic knowledge for the development of prevention, treatment and coping strategies.

In this long-term study, 3000 children will be studied, together with their
parents and grandparents, from the 20th week of pregnancy to early adulthood. [SNF, 2006]

The core study analyses the psychological, biological, genetic and social factors that contribute to the mental health and influence a healthy development of a human being. [sesam, 2006]

Embedded in the core study there are 12 sub-studies...

- Substudy B explores whether familial risk conditions can be positively influenced for supporting the healthy development of children.
- Substudy C
- Substudy D
- Substudy E
- Substudy F
- Substudy G
- Substudy H
- Substudy I
- Substudy J
- Substudy K
- Substudy L
- Substudy M

2.1 **sesamDB**

The *sesam* study will generate a large amount of data like questionnaires, biological analysis, genetic data, multimedia content and sequence data. [Glavic et al., 2006] The time span of the *sesam* project and thus of the collection of
data is more than 20 years. The sub-project \textit{sesamDB} is responsible to design and implement a database and client applications which are necessary to manage the scientific and administrative data of \textit{sesam} [Dittrich and Glavic, 2006]. The database and the application developed in the scope of this thesis is part of \textit{sesamDB}.

\section*{2.2 Subject Management}

The task of this thesis is the design and implementation of a database client application for administrating study subjects, employees and other persons associated with \textit{sesam}, together with their addresses, e-mail addresses and phones. Relations among study subjects need to be represented in the database, as well as other facts about their study participation.

The database has to be bitemporal, which means it has to be capable to manage time-varying data and keep a history of all data manipulations.

\subsection*{2.2.1 Traceability}

The long duration of the project requires that all data manipulations are comprehensible and traceable., i.e. it has to be possible to retrace who inserted or updated data at what time. This is quite an important requirement since there will be a lot of personnel entering and modifying data over a period of 20 years. If an error in the data is found, a detailed update history helps a lot to investigate the source of the mistake. Additionally users should be able to comment every update they perform. If for example the person holding the role of the father changes, a comment about the circumstances would be valuable.

\subsection*{2.2.2 Time-varying information}

Since the study subjects of the \textit{sesam} project are watched for about 20 years, it is very likely they will change their address, their phone numbers or even
their names. The participation with the research project might be suspended for a certain time period and be resumed later.

Not just the children themselves but also their parents (biological and psychological) and grandparents participate in the study and are therefore stored in the database. The biological parents indeed never change but the family environment a child grows up in does.
Chapter 3

Temporal databases

A database that maintains past, present, and future data is called a *temporal database*. [Tansel et al., 1993]

While in conventional databases only a snapshot of the mini-world is stored, temporal databases aim to store facts as they change over time. Facts stored in a temporal database are associated with time, most prominently with one or more dimensions of *valid time* and *transaction time*. [Jensen, 2000](p 29).

Many applications require temporal data because not only current facts of the modelled mini-world are relevant but also the facts from the past and the future, as far they are predictable. Together with the validity of facts in the real world, which is modelled by recording the *valid time*, the history about the facts’ states in the database is recorded by registering the *transaction time*.

3.1 Application

The number of possible applications where temporal storing of data is useful or required is tremendous.

A lot of facts changes over time but it is not always required to store the changes of the data in the database. Consider a personal e-mail address database. In such an application it is normally not required to store the past states of information as people change their e-mail address from time to time.
CHAPTER 3. TEMPORAL DATABASES

An old e-mail address is not worth storing since it is probably disabled after a new e-mail address becomes valid. It is sufficient to store the currently valid address and always overwrite the old ones. Such a database stores a snapshot of the real world and features no temporal aspects.

In a more professional environment such a simple snapshot database usually does not meet the requirements. For instance for a database of a warehouse storing the information about the products at stock, it is probably required that data manipulations are traceable. If an employee changes the description of a product in the database, the supervisor wants to be able to understand who modified the dataset. Actually he wants to be able to look up the former description of the product. In case the employee entered a description to the wrong product by mistake, the supervisor can easily correct the error. Such a database features a rollback functionality which allows the users to look up former states of the database and reverse data manipulation if needed.

Real-world database applications are frequently faced with accountability and traceability requirements. Accountability implies one can find out who entered a certain data set. Traceability refers to that former states of data sets are kept and they can be reconstructed. This leads to transaction time databases that faithfully timestamp and retain all past database states. [Copeland, 1982].

However in a long-term research project like sesam it is not only nice to have but required to have a temporal database. For the researchers it is essential to know not only the current participation of a study subject in a certain study but also the participations in the past. One important part of the sesam project is the research of the influence of the family on the study subjects psyche. Since constellations of families change at times, relationships between the study subjects are typically time-varying. Additionally to the ability to store past, current and future data, a rollback mechanism is also required for the same reasons as in the warehouse database.
3.2 Definitions and Taxonomy

This section shall give an overview of terms and concepts in conjunction with temporal databases.

3.2.1 Time-varying and static attributes

In temporal databases it is all about storing time-varying attributes. In order to model a temporal database, a distinction between static and time-varying attributes is needed. Time-varying attributes are values that change over time, while static attributes never change their value in the real world. The only reason why a static value would have to be modified in a database, would be to correct it.

Consider modelling a subject management database consisting of entities and relationships. Entities, i.e. physically existing objects in the real world, typically consist of static and time-varying attributes, while relationships are typically time-varying as a whole. A study subject is an entity while roles like e.g. legal representative are relationships between different study subjects. The study subject’s date of birth is typically a static attribute, because it never changes in the real world. Family name is a time-varying attribute, since it can change in the real world due to marriage or other reasons. The relationship legal representative between two subjects is time-varying as a whole, since it has a certain start date and probably ends at some day.

3.2.2 Concepts of time

Different kinds of concepts of time are used in the field of temporal databases. This section shall give an overview of the used terms and why there are basically only three concepts of times necessary to deal with bitemporal information.

Researchers have been arguing that a single time attribute is insufficient, and that two time attributes are necessary to fully capture time-varying infor-
mation [Snodgrass and Ahn, 1985]. A third concept is used to store information about events in the real world.

For these three concepts of time exist different terms:

### 3.2.2.1 Valid Time

In this thesis we use the term *valid time* to describe the time period when a information was valid in the real world. Other terms used in the literature are for example *event time* and *effective time* [Snodgrass and Ahn, 1985].

The valid time of a fact is the collected times possibly spanning the past, present, and future when the fact is true in the mini-world. [Jensen et al., 1994]

For instance let’s recall the introductory example. In order to store the information “John lives in Basel from 01 Jan 2000 until 31 Jul 2001”, we store the information “John lives in Basel” with a valid time period “[01 Jan 2000, 31 Jul 2001]”. The valid time period can regard the past, present or future.

### 3.2.2.2 Transaction Time

On the other hand we use the term *transaction time* to describe when some data was represented in a database. Other terms used in the literature for this second concept of time are *physical time*, *registration time* and *data-valid-time* [Snodgrass and Ahn, 1985].

Let’s assume that the information “John lives in Basel from 01 Jan 2000 until 31 Jul 2001” was entered into the database on 07 Jan 2000. Thus we store the information “John lives in Basel” with a valid time period “[01 Jan 2000, 31 Jul 2001]” and with a transaction time “07 Jan 2000”. As long as the data set is not modified or deleted, the transaction time end has no determined value; its value can be seen as a variable. This special value is named “until changed” (UC) or quite simply “now” by many authors [Jensen,
2000]. The transaction time of current data sets comes up to a interval “[07 Jan 2000, now]”.

While the valid time concept is used to describe the validity period of a certain information in the real world, and the transaction time is an artificial time concept only existing in the database system.

3.2.2.3 User-Defined Time

The third concept is called user-defined time. This time concept is necessary for additional temporal information, not handled by transaction time or valid time, stored in the database [Snodgrass and Ahn, 1985].

Consider the example of a student database of a university. The entry and the leaving of a student can be seen as the begin and the end of the validity period of a certain student. The date when a certain student tuple was entered into the database is the transaction time associated with the tuple. But there are more time related attributes stored in this database. For example the date of birth of the students, the date of high school graduation, the dates of passed exams, etc. These kinds of time attributes are summarized in the concept of user-defined time.

3.2.3 Kinds of databases

Following [Snodgrass and Ahn, 1985] we distinguish between four distinct kinds of databases, according to the temporal aspects they support:

3.2.3.1 Static Database

A static, also known as conventional or snapshot database models the real world, as it changes dynamically at a particular point in time. The current state of the database is its current contents, which does not necessarily reflect the current status of the real world [Snodgrass and Ahn, 1985].

In such a database data is always overwritten as it takes new values. No historic data is kept.
3.2.3.2 Static Rollback Database

A *static rollback database* extends the static database by not overwriting the stored values as the real world changes, but keeping the old versions of the tuples. Also a deletion of a record is usually not done by physically overwriting a record but by marking a record as deleted. Thus all manipulations of data are recorded. For this approach a representation of *transaction time* is necessary to store the time when the information was stored in the database. It is not distinguishable whether a value has changed because the attribute of the real-world has changed, or whether the attribute was erroneous and was corrected. With this approach a history of the database activities, rather than the history of the real world is recorded [Snodgrass and Ahn, 1985].

Using a *conventional database* is becoming unpopular since for most applications it is necessary to be able to rollback or undo transactions. And since prices for data storage are falling, the ability to preserve past states of the data is worth the extra storage costs.

3.2.3.3 Historical Database

As values in the real world change over time, *historical databases* aim to store the history as it is best known. While *static rollback databases* record a sequence of static states, *historical databases* record a single historical state per relation. [Snodgrass and Ahn, 1985]. In other words each time-varying attribute is associated with a *valid time*.

There exist definitions calling this kind of database already *temporal database*. Apparently it meets the very broad definition "a database that maintains past, present, and future data is called a temporal database"[Tansel et al., 1993].

3.2.3.4 Temporal database

A database which combines the features of *static rollback* and *historical database* is called *temporal* or *bitemporal database*. The term bitemporal suggests that two concepts of time are represented in the database, namely *valid time* and *transaction time*. Time-varying facts are associated with a *valid time*, repre-
senting the time span they were valid in the real world. Both time-varying and static facts are associated with *transaction time*. *Transaction time begin* represents the point in time a fact adopted its state in the database. *Transaction time end* refers to the point in time a fact became obsolete and therefore was replaced by a new version or deleted.

### 3.2.4 Intervals in temporal databases

In temporal databases additional kinds of data constraints emerge concerning interval overlapping. Such constraints are normally not required in conventional databases though in spatial databases this is a topic as well.

One typical constraint is that the *valid time* periods of the versions of a certain entity must not overlap. For example the name attribute of a person is typically time-varying. But a the validity of a person’s name expires in the moment a new name becomes valid. Overlapping of the *valid time* interval of names for the same person should not be allowed.

A similar constraint could be, that a person can have several phones, but only one main phone at any point in time.

In order to handle overlapping of intervals one has to identify all possible constellations of overlapping. There are 13 situations two intervals in a 2-dimensional space can be arranged. Figure 4.4, in the next chapter, will present them.

### 3.2.5 Attribute versus tuple timestamping

In temporal data models, attribute values are associated to one ore more time attribute. We can distinguish between data models where each attribute is timestamped separately and models where a set of attributes, namely a tuple, is timestamped.[Jensen, 2000](chapter 6)

The advantage of *attribute timestamping* data models is that redundancy can be completely avoided. As attributes change their values independently from each other in the real world, so do their database values. But *attribute*
timestamping leads to a completely degenerated relation schema, since each table consists only of one column with user data and the associated time attributes.

Temporal database models using tuple timestamping use the well-established storage organization and query evaluation techniques of relational databases. But if only one attribute of a tuple changes its value, the whole tuple has to be copied inevitably leading to redundancy.

### 3.2.6 Synchronous attributes

The definition of synchronism is needed so that we can characterize relations and avoid redundancy [Jensen and Snodgrass, 1995].

A set of time-varying attributes in a given relation is called synchronous if every time-varying attribute can be uniformly associated with and be directly applied to the timestamp values in each tuple of the relation.[Tansel et al., 1993](p 94)

Jensen gives a more mathematical definition: Attributes are synchronous if their lifespans are identical when restricted to the intersection of their lifespans.[Jensen and Snodgrass, 1995]

Consider a company where an employee gets a raise in salary if and only if they get a promotion, in other words, and employees are never demoted. In this case, salary and position of an employee form a set of synchronous attributes. [Tansel et al., 1993](p 94).

### 3.2.7 Time dependence

Synchronism and time dependence are related terms needed to specify time normalisation, introduced in section 3.2.8.

Time dependencies occur when two or more temporally unrelated facts are mixed in one time-varying relation.[Tansel et al., 1993](p 96)
In other words, when non-synchronous attributes are maintained in the same relation, these attributes are mutually time dependent.

### 3.2.8 Time normalization

Navathe and Ahmed [Tansel et al., 1993](p 96) propose a time normalization. The idea underlying time normalization is the conceptual requirement that tuples are semantically independent of one another.

A relation is in time normal form (TNF) if and only if it is in Boyce-Codd-Normal Form and there exists no temporal dependencies among its non-key attributes. [Tansel et al., 1993](p 96)

Table 3.1 is in TNF, if salary and position are synchronous. In case salary and position are not synchronous there would be time dependence and table 3.1 would have to be normalized, resulting in two relations shown in table 3.2.

<table>
<thead>
<tr>
<th>EmpNo</th>
<th>Salary</th>
<th>Position</th>
<th>$T_{\text{beg}}$</th>
<th>$T_{\text{end}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>20K</td>
<td>Typist</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>33</td>
<td>25K</td>
<td>Secretary</td>
<td>24</td>
<td>35</td>
</tr>
<tr>
<td>45</td>
<td>27K</td>
<td>Jr Engineer</td>
<td>28</td>
<td>37</td>
</tr>
<tr>
<td>45</td>
<td>30K</td>
<td>Sr Engineer</td>
<td>37</td>
<td>42</td>
</tr>
</tbody>
</table>

Table 3.1: Synchronous time-varying attribute
3.3 Research status

Storing time varying information in databases has been done for decades and there has been research activity in this area for quite some time [Snodgrass, 1990]. Since 1998 temporal support is indeed a part of SQL standard. SQL3 supports a serie of temporal data types like \textsc{date}, \textsc{time}, \textsc{timestamp}. Further the notion of a table with temporal support is supported. A table can have \textit{valid-time} support, \textit{transaction-time} support, or both (bitemporal support) [Snodgrass, 1998]. Nevertheless neither the commercial database management systems (DBMS) such as \textit{Oracle}, \textit{Sybase}, \textit{Informix} and \textit{O2}[Steiner, 1998] nor the open-source projects \textit{PostgreSQL} and \textit{MySQL} have fully implemented this. The data types \textsc{date}, \textsc{time} and \textsc{timestamp} are widespread in most DBMS, but native (bi)temporal table support or a temporal query language is only implemented in some niche products such as \textit{TimeDB}[Schmidt, 2005].
3.3.1 Temporal Query Languages

A notable part of the research in temporal databases focuses on the development of temporal query languages.

While conventional SQL lacks from featuring support for temporal data, researchers proposed many temporal query languages, to simplify and optimize manipulation of temporal data. Temporal query languages reduce the amount of database code needed for temporal data queries by as much as a factor of three in comparison to the standard SQL query language [Jensen, 2000](preface)

We present a selection of query languages extending SQL:

3.3.1.1 TempSQL

TempSQL is an extension of SQL with much more expressive power than conventional SQL. For SELECT statements the additional expression WHILE simplifies temporal queries tremendously. [Tansel et al., 1993](page 38) A query “Give managers of John” is expressed in TempSQL as follows:

```
SELECT name, manager
WHILE emp.dept=manager.dept
FROM emp, management
WHERE name=John
```

The WHILE clause limits the time range of the result set to managers, who were in charge while John was actually working in their department. Managers running the departments John worked in before or after he worked there are excluded.

3.3.1.2 TSQL

TSQL is also a superset of SQL. [Tansel et al., 1993](page 99) TSQL has the additional constructs like the WHEN and TIME-SLICE clause. Using the WHEN clause comparison of time intervals becomes very easy. A query “Find the manager of employee 23 who immediately succeeded manager Jones” is expressed in TSQL as follows:

```
SELECT name, manager
WHILE emp.dept=manager.dept
FROM emp, management
WHERE name=John
```
CHAPTER 3. TEMPORAL DATABASES

SELECT B.mgr
FROM management A, management B
WHERE A.employeenum = B.employeenum
  AND A.employeenum = 23
  AND A.manager = 'Jones'
WHEN B INTERVAL FOLLOWS A INTERVAL

3.3.1.3 HSQL

The Historical Query Language supports real time\textsuperscript{1} databases. ([Tansel et al., 1993](page 110)

\textit{HSQL} supports the definition of historical relations.

```
CREATE STATE TABLE emp
  (ENO CHAR(10),
   SAL DECIMAL(5),
   UNIQUE (ENO))
WITH TIME GRANULARITY DATE
```

This statement creates two relations in the background, namely \textit{CURRENT-EMP}(ENO, SAL, FROM, TO) and \textit{HISTORY-EMP}(ENO, SAL, FROM, TO). Data manipulation operations are performed on the virtual relation \textit{EMP} and passed on to the \textit{CURRENT} respectively \textit{HISTORY} relations.

3.3.1.4 TSQL2

\textit{TSQL2}, is a temporal extension to the SQL-92 language standard. It comes from a different group of researchers than \textit{TSQL}. [Snodgrass et al., 1994],[Jensen, 2000](Part I+II)

For example creating a table with valid time timestamped tuples is done by adding an option \texttt{AS VALID STATE} to the table definition statement:

```
CREATE TABLE Emp
  (ID SURROGATE NOT NULL,
   NAME CHARACTER (30) NOT NULL,
   Salary DECIMAL (8,2))
AS VALID STATE;
```

Inserting a tuple looks as follows:

\textsuperscript{1}by real time the authors of \textit{HSQL} mean what we call valid time in this thesis.
And a query “Who had the same salary for the longest total time?” in TSQL2 looks as follows:

```sql
SELECT SNAPSHOT E2.NAME
FROM Emp(ID,Salary) as E1,
     Emp(ID,Name) as E2
     (SELECT MAX(CAST(VALID(E) AS INTERVAL DAY)
      FROM Emp(ID,Salary)(PERIOD) AS E) AS E3
WHERE E2.ID = E1.ID
```

The list TQuel, are extension to the relation data model.

HRDM (Historical Relational Data Model)

### 3.3.2 Temporal joins

Joins in temporal databases are an issue because of the following reasons: First, conventional techniques are designed for the evaluation of joins with equality predicates rather than the inequality predicates prevalent in valid time queries. Second, the presence of temporally varying data dramatically increases the size of a database.[Gao et al., 2003]

Researchers have proposed new join operators for temporal databases like temporal Cartesian product, time join, TE-join, natural time join, intersection join, union join, etc.

We present an example for an application of the temporal Cartesian product operator. Consider a database with two relations shown in table 3.3.
A query “Show the names of employees and managers where the employee worked for the company while the manager managed some department in the company” would lead to a complex query using conventional SQL. A join operator performing a temporal Cartesian product simplifies this task. The resulting relation is shown in table 3.4.

### 3.3.3 Temporal indexing techniques

Another subfield of temporal database research is about indexing techniques. Since temporal normalization leads to data models with many fragmented relations, database performance becomes an issue.

Like most indexing structures, the desirable properties of a temporal index include efficient usage of disk space and speedy evaluation of queries.[Ooi et al.,
1998]. Since bitemporal records can be viewed as bounded planes in a 2-dimensional space, indexing techniques from spatial databases can be adapted. [Jensen, 2000](chapter 36) proposes an extension of the \( R^*-tree \), called \( GR^-\text{Tree} \) supposed to be faster than the original \( R^*-tree \).

### 3.4 Approaches

This section introduces some approaches for designing temporal database models found in literature focusing on approaches that build bitemporal database models upon conventional relational databases.

#### 3.4.1 Concepts of “now” and “infinity”

In a temporal database we want to be able to store information like “Fact X is valid since ‘01 Jan 2005’ until now”. But “now” is not a constant value but rather an ever changing variable. “now” is expressed in SQL as \texttt{CURRENT_TIMESTAMP} within queries, but for storing the information behind “now” in the database, a concept is needed.

Some DBMS feature a database-resident variable, such as “now”, “until-changed” (UC), “@” and “-”. Time variables would be convenient but Jensen advises that they would lead to a new type of database, consisting of tuples with variables [Jensen, 2000].

Using a variable “now” is overly pessimistic and limits valid time to the past [Jensen, 2000]. We can not use “now” to store the information about the future like “Fact X is valid from ’01 Jan 2010’ until changed” until the year 2010 when it becomes to “Fact X is valid from ’01 Jan 2010’ until now”.

Instead of a variable there are special (non-variable) valid-time values, such as “forever”, “infinity” or “\( \infty \)” [Jensen, 2000]. With such an approach it does not matter whether a valid time is in the past or future. Nevertheless Jensen calls this technique overly optimistic, because the use of a constant \textit{forever} or any large date value is always inappropriate and does not model the real world correctly.
3.4.2 Temporal Data Models

We present some relational temporal data models and we confine ourselves to models supporting both *valid time* and *transaction time*, known as *bitemporal*.

One distinction can be made between models sticking to first normal form (1-NF) and models not in 1-NF. Related is the distinction between *tuple timestamping* and *attribute-value timestamping*. [Jensen, 2000](chapter 6)

Remaining in 1-NF means in this context means, a database is modelled as if it was a snapshot database, following the established normalization rule proposed by [Codd, 1970], and time attributes are added to the tuples. Those data models remaining within 1-NF with *tuple timestamping* potentially introduce redundancy because attribute values that change at different times are repeated in multiple tuples.

The non-1-NF models, usually with attribute timestamping avoid redundancy. But they are not capable of directly using relational storage structures or query evaluation techniques. The performance of a non-1-NF may degrade. [Jensen, 2000](chapter 6)

3.4.2.1 Trade-off between query performance, complexity and redundancy

Modelling a temporal database the same way one would model the equivalent snapshot database and adding time attributes to the tuples is one way to go. Completely decompose the attributes into many relations and timestamp each attribute seperatly is the other extreme.

Consider a database snapshot database consisting of one relation, a timeinvariang key, and 5 attributes (table 3.5):

<table>
<thead>
<tr>
<th>key</th>
<th>att1</th>
<th>att2</th>
<th>att3</th>
<th>att4</th>
<th>att5</th>
</tr>
</thead>
</table>

Table 3.5: Example: The snapshot equivalent
CHAPTER 3. TEMPORAL DATABASES

Making this database temporal means adding time attributes. One way to do this would be just adding the time attributes to the relation, in other words timestamping the tuples (table 3.6. Assuming all attributes are mutually non-synchronous, changing the value of one attribute would lead to a replication of a whole tuple and the values of the 4 attributes not changed would be saved manifold, resulting in redundancy.

Table 3.6: Example: Relation not in TNF

<table>
<thead>
<tr>
<th>key</th>
<th>att1</th>
<th>att2</th>
<th>att3</th>
<th>att4</th>
<th>att5</th>
<th>T&lt;sub&gt;begin&lt;/sub&gt;</th>
<th>T&lt;sub&gt;end&lt;/sub&gt;</th>
</tr>
</thead>
</table>

Time normalizing the given relation leads to 5 relations shown in 3.7. The space needed to store one version of a tuple is higher in fact, but as the attributes values change asynchronously over time, the storage space needed becomes less, because data is not stored redundantly, as in the non-TNF relation.

Table 3.7: Example: Time normalized

<table>
<thead>
<tr>
<th>key</th>
<th>att1</th>
<th>T&lt;sub&gt;begin&lt;/sub&gt;</th>
<th>T&lt;sub&gt;end&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>att2</td>
<td>T&lt;sub&gt;begin&lt;/sub&gt;</td>
<td>T&lt;sub&gt;end&lt;/sub&gt;</td>
</tr>
<tr>
<td>key</td>
<td>att3</td>
<td>T&lt;sub&gt;begin&lt;/sub&gt;</td>
<td>T&lt;sub&gt;end&lt;/sub&gt;</td>
</tr>
<tr>
<td>key</td>
<td>att4</td>
<td>T&lt;sub&gt;begin&lt;/sub&gt;</td>
<td>T&lt;sub&gt;end&lt;/sub&gt;</td>
</tr>
<tr>
<td>key</td>
<td>att5</td>
<td>T&lt;sub&gt;begin&lt;/sub&gt;</td>
<td>T&lt;sub&gt;end&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

3.4.3 Separation of history relation

The currently valid tuples and the historic tuple could theoretically be stored in the same table. In practice, the currently valid tuples are usually stored in a current table and the historical tuples are kept in separate table called audit or history table.

Referenz benötigt
+1/2
Seite

28
Chapter 4

Our approach

This chapter covers a detailed presentation of our approach of building a bitemporal database.

Our bitemporal data model is based on a conventional relational database. For the *sesam subject management database* the data volume, the number of tuples as well as the update frequency of the time-varying attributes are on a manageable scale. Therefore our approach focuses not primarily on scalability and performance issues, but on a wieldy complexity.

For instance we do not completely time-normalize the relations as proposed by Navathe and Ahmed [Tansel et al., 1993], but rather keep the attributes together in as few relations as possible.

Furthermore we introduce a layer hiding the partly time-normalized relations for the application by providing updatable views.

4.1 Our “now” concept

According to Jensen, there is no variable “now” in conventional databases, only in temporal databases [Jensen, 2000].

Since we build our approach on a conventional database we are interested in developing a concept without a variable.
In order to store information like "Fact X is valid since '01.01.2005' until now" or "Fact X was valid since ever until '01.01.2005' " we do not use a variable but we use the lower and upper boundaries of the value range of the timestamp data type of our DBMS. The upper boundary of timestamp is called infinity, the lower boundary -infinity.

Like in math, we call the interval \([-\infty, \infty]\) open. Let \(d_1\) and \(d_2\) be date values with condition \(-\infty < d_1 < d_2 < \infty\). Then the intervals \([-\infty, d]\) and \([d, \infty]\) are half-open, and \([d_1, d_2]\) is a closed interval.

For a fact associated with a half-open interval in the transaction time domain, this means it is a current fact. A closed interval in the transaction time domain implies that a fact is historic. There is no open interval in the transaction time domain, since there is always a lower bound, representing the date when a tuple was entered into the database.

Valid time intervals are either closed intervals, with a determined begin and end, or they come up to a half-open interval with only one determined boundary, and if neither the begin and end are known, it results in an open interval.

4.2 Keys

There are special issues concerning keys we have to deal with in a bitemporal database. There are two aspect about keys we do not have in a conventional database: First, in temporal databases an object of the real world is not represented by one object in the database, but by multiple versions of the object. Second, we assume all attributes are subject to modification, even the natural key of an object. Since each object is referenced by other tuples, namely its history tuples, updates of the key attributes would lead to a chain reaction of updating references.

For these reasons it is accurate to have an artificial key representing the objects of the real world we call static id, also known as timeinvariant key (TIK) [Jensen, 2000](chapter 29). There is an artificial key for each timevary-
ing version we call timevarying id. So the unique identifier for each currently valid version of an object is a key composed of its static id and none, one or several timevarying ids. The key of the historic objects are composed of the key of its current object plus the transaction time.

4.3 Time normalization

4.3.1 Valid time domain

For the domain of valid time, we follow the proposed time-normalization of Navathe and Ahmed [Tansel et al., 1993] (p 97) only in parts. As seen in section 3.4.2.1 in an application, were all attributes are asynchronous from each other, time-normalization leads to decomposition of the relations. Strictly following the rule of time-normalization would lead to a data model were the number of relations equals the number of attributes. Such a fragmented data model degrades performance and becomes confusingly complex.

In our approach relations are only split to a minimum extent, so the complexity of the data model remains low, and queries perform well. In most cases a relation is split into a static one, only with time-invariant attributes, and one or two time-varying relations.

Besides this model reduces at least a part of the redundancy, compared to just timestamping the snapshot model, it has the advantage, that static keys are in static relations, and therefore they are unique keys and they can be referenced by other relations.

Figure 4.1 illustrates an example of how an object (key, A, B, C) is split along the columns into a static and two time-varying relations for the valid time domain. Attribute A is a static one while B and C are time-varying. The different versions of the object can be assembled by joining together the tuples in the three relations with the same static id and overlapping valid time intervals.
CHAPTER 4. OUR APPROACH

4.3.2 Transaction time domain

In the domain of transaction time (TT), tuples are divided into current and historic data sets. We call a relation with currently valid tuples current relation, the one with the historic tuples history relation.

Each tuple’s TT<sub>begin</sub> value is a certain date value in the past. If the TT<sub>end</sub> is also a certain date value in the past, it means that such it is historic. Such a tuple belongs either to a deleted data set or it was overwritten. Storing current and historic tuples in the same relation is one possibility. But there are some reasons to separate them. Since current tuples are typically accessed much more frequently than historic tuples, it makes sense to keep them in a separate table for performance reasons. The number of historic tuples can be much higher than the number of current tuples.

Another point supporting a separation of current and historic data sets is that current tuples are referenced by other current tuples and therefore the key of an object should not change. As introduced in section 4.2 in our approach, current tuples always have a primary key (id), while the historic tuples’ primary key is composed of the primary key of the current relation plus the transaction time: (id, tt<sub>begin</sub>). In our application, references among historic tuples are usually not needed. Historic tuples are typically accessed via their current version.
CHAPTER 4. OUR APPROACH

Since in the current relation all tuple’s $TT_{end}$ value is equal to now, this attribute can be omitted.

One might argue that in the domain of valid time a separation of currently valid and formerly or future data sets would make sense for the same reason. However for the valid time domain there exist data sets that are already in the database but will become valid in the future. With such a model one would always have to check whether a data set becomes valid at this moment and if so, move it to the current relation. Such a permanent monitoring would be rather hard to handle. On the one hand implementing such a mechanism would be quite complex compared to our solution. On the other hand permanently monitoring a sizable part of the database would lead to performance issues.

4.3.3 Bitemporal domain

Combining the two approaches for valid time and transaction time leads to the picture illustrated in figure 4.3.

Each of the relations resulting from the time-normalization (black) features a history table (grey). One object is identified by its static primary key. The different versions have a key composed of the static key and the time-varying keys.

A relation of the snapshot model results in 2, 4, or 6 relations in the bitemporal model using our approach.
CHAPTER 4. OUR APPROACH

4.4 Intermediate layer

The main disadvantage of time-normalization is that the application programmer has to deal with a complex and fragmented data model. In order to reduce this drawback, our approach introduces a layer between the split relations and the application. This layer consists of updatable views. The application performs INSERT, UPDATE and DELETE operations on these views as if they were database tables.

An INSERT statement on such a view can have two different meanings. In one case, a new object is inserted into the database. Thus tuples have to be inserted into the static and time-varying relations. In the other case, if an existing primary key of the static relation is given, it is about inserting a new time-varying tuples. In this case, there is no insert into the static relation, but potentially an update on an existing tuple of the static relation.

Updating underlying relations by performing updates on a views is not
always possible or might not have a definite implication. For instance updating aggregated values is not possible or updating underlying relations without its key might cause nondistinctive behaviour. Therefore the views to be updated in our approach have no aggregations and must comprehend a unique key of the underlying relations.

With this layer, a data model is presented to the application programmer with the complexity level of the equivalent snapshot database. Indeed, joining together the decomposed relations could also be done on application level. But then every application accessing the database would have to implement the same functionality itself. We think this is a strong argument to implement this layer on the database.

4.5 Interval overlaps

In section 3.2.4 we introduced a new kind of constraints occurring in temporal databases. If valid time intervals of a certain object shall not overlap, a new kind of database constraint is needed.

Since the valid time interval boundaries are typically entered by the users, the database has to assure the compliance with the interval constraints.

One strategy could imply that inserts or updates leading to a constraint violation are refused by the DBMS. This would lead to a lot of manual work for the users since the validity intervals of existing data set would have to be adapted before.

The strategy in our approach assumes that the currently inserted or updated data set has priority and the other existing tuples have to make room for it.

Figure 4.4 lists all possible constellations of one existing interval $x$ and a new interval $y$ with priority.

Figure 4.4: Interval overlaps [Tansel et al., 1993](p 71)
In case (1) and (13) the existing interval is not affected by the insertion of y since the intervals do not overlap.

Whether case (2) and (12) require action depends on the convention whether the begin of one interval can be equal the end of another interval or not. If such a convention applies, case (2) and (12) represent an overlap, otherwise not. Our approach suggests a convention that the boundaries of two meeting intervals have to be distinct from each other. The end of the preceding interval needs to be smaller than the beginning of the succeeding. We define a minimum difference, which is the smallest possible interval length of the given data type. This interval is called “chronon”, and its actual length is depending on the data type of the used DBMS.

Thus in cases (2), (3) and (4) the interval x’s begin would have to be set to \( y_{\text{end}} + 1 \text{"chronon"} \). In case (8), (11) and (12) \( x_{\text{end}} \) has to be pruned to \( y_{\text{begin}} - 1 \text{"chronon"} \).

In a situation of type (9), interval x has to be split into two intervals. One with a \( x_{\text{begin}} \) unchanged but \( x_{\text{end}} \) set to \( y_{\text{begin}} - 1 \text{"chronon"} \). A second interval has \( x_{\text{begin}} := y_{\text{end}} + 1 \text{"chronon"} \) and a unchanged \( x_{\text{end}} \).

In cases (4), (5), (7), (8) the existing data set is replaced because the new one overlaps it completely.

Recapitulatory, the 13 cases of overlapping can be categorized into the following situations:

- (2), (3) and (4): \( x_{\text{begin}} \) is postponed
- (10), (11) and (12): \( x_{\text{end}} \) is prepone
- (9): \( x \) is split
- (4), (5), (7), (8): the old entry is replaced.

For our application we assume that the standard operation has the following pattern: For example, a person’s name’s validity end will usually be set to \(+\text{infinity}\). When a new name is entered, the old name’s validity ends just before the validity begin of the new name starts.
CHAPTER 4. OUR APPROACH

One can consider warning the user if a gap in validity results. For our example of person’s names this would be useful since having a time period when a person does not have a name would be an unwanted situation. This problem depends on the kind of information to be stored.

4.5.1 The “main address” problem

A very similar kind of constraint concerning interval overlaps is the following: Overlapping of the tuples in a relation shall be allowed, but for certain values, there shall be no overlapping.

Such a constraint in an application could be that a person can have several addresses whose validity can overlap arbitrarily. But at every point there can be only one main address.

One possibility is to model the relationship between persons and addresses with two relations, one for the main address and one for the others. Then the same rule for overlapping as described in the previous section could be applied for the main address relation and no special action would be needed for the other relation. But maintaining the same kind of relationship in two distinct relations has also drawbacks. If for example a person-address relationship already exists and later becomes a main address, the administration effort would be much higher than have all the relationships in one database table.

Therefore our approach suggests to maintain one table for person-address relationships. A constraint is needed which assures that if an address is set as the main address, the completely overlapped addresses need to be set to non-main, and the partially overlapped addresses have to be split into an interval when they are main and a non-main interval.

Figure 4.5 shall illustrate the problem of inserting a new main person-address relationship where already exist main addresses. The left side is the situation before inserting the new relation, while on the right side is the situation after inserting. The interval about to insert (dotted) partially overlaps an existing interval and completely overlaps a second one. The partially overlapped needs to be split while the completely overlapped is set to non-main.
Figure 4.5: The overlap issue with main addresses
Chapter 5

Implementation

This chapter describes the design and development of the bitemporal database and the Java client.

5.1 Requirements

The *sesam* study requires a software client with the capability to enter, view and modify data about persons, more precisely study subjects and employees. The information to be maintained about persons includes personal data like name, gender, date of birth, etc., postal addresses, e-mail addresses and phone numbers. For study subject further personal details about the persons are stored like languages, life dates and special annotations and announcements. Special annotations include notes about individual restrictions like disabilities. Annotation inform about special circumstances like for example the death of a family member.

Since a significant part of the *sesam* research includes the studies of parents and grandparents, relations among study subjects are represented in the database. Besides the mother, father, legal representative and grandparents of the study subject there are also relationships between the adults namely partner and spouse. Relationships might change over time. Further, the subjects’
participation status in the project itself, as in the sub-studies is maintained and typically changes during the more than twenty years the project lasts.

For employees hiring date, jobrole and the core studies they are working at are stored in the database.

5.1.1 Given systems

Since the application for subject management of sesam is part of a greater project including other databases connected to it, the project managers laid down to use one single database product to be used by all applications. There exist Database Management Systems designated for temporal and bitemporal data models, like for example TimeDB [Schmidt, 2005], but they are all niche products whose durability is not assessable. Moreover many of the standard features of a modern conventional DBMS are not available in these specialized systems. Therefore it was decided to use a widely established relational DMBS PostgreSQL.

The Database Management System as well as the Programming Language are already established within the sesamDB project. The DBMS is the open source database PostgreSQL (Version 8.x). The client application is build with Java and the graphical user interface is made with the Standard Widget Toolkit (SWT). The use of SWT makes it possible to run the application on different operating system because SWT accesses the user-interface facilities of the operating systems on which it is implemented [SWT, 2007].

5.1.2 Client requirements

The primary aim of the client application is to provide an intuitive graphical user interface which allows the users to enter, view and modify data about study subjects and employees. Besides accessing the current information there is also the possibility to view the history of data modifications. One can see who when updated a data set which helps if investigations are required.

The complexity of the bitemporal database should be hidden from the
CHAPTER 5. IMPLEMENTATION

client. Temporal operations like historization, and checking temporal interval constraint should be handled by the database.

5.1.3 Database requirements

All entities and relations time varying in the real world are modelled as such in the database. All data changes have to be traceable, which implies that audit tables are required for all database tables. This includes that the modifier is stored together with each tuple.

This section describes the requirements of the software. Some exemplary use cases are described detailed.

5.2 Architecture

Figure 5.1: The architecture

On the lowermost layer are the database tables. Some of the application logic, like e.g. the “historization” of data, is implemented with triggers. By having the historization functionality at this level, all applications writing to the database can benefit from it. They do not have to care about historization of their data manipulations.

In the case of non-split relations, the application performs its operation directly on the database tables (pictured by the right arrow). If bitemporal relations are split into multiple tables, they are joined together to views. Thus the fragmented relations are hidden to the application layer. These views are updatable and the application performs data manipulation operations on them (pictured by the left arrow). INSTEAD OF RULEs, as they are called in PostgreSQL, are responsible to remit the operations on the views to the tables. In other DBMS, e.g. Oracle this procedures are called INSTEAD OF TRIGGERs.

The advantage of having this layer of views is that applications do not have to maintain various relations. The consistency of the data spread over the various relations is assured by the database. Particulary with regard to
that several applications might access the database, a lot of implementation work only has to be done once.

The upper part of figure 5.1 illustrates the Java application. The persistency of Java objects is done using the Hibernate framework. Hibernate is widespread, easy to use and distributed under the General Public License [Hibernate, 2007]. A similar framework would have been iBATIS by the Apache Software Foundation. By means of both frameworks, plain Java Objects (POJOs) are mapped to their database equivalent via XML mappings. The persistency framework loads tuples from the database, creates a POJO with and save them.

The actual Java application is responsible to display the persistency objects on the (SWT) graphical user interface and remit data manipulations on them to the persistency framework.

5.3 Database

This section describes the modelling and implementation of the database.

When designing a conventional snapshot database one entity or relationship is normally modelled as one database relation.

In our bitemporal database, an entity that would be modelled as one relation in a snapshot database, is split into several relations and requires triggers for historization. In order to hide the complexity of the underlying database model, views are provided that join together the split relation.
Figure 5.2: Overview of database model

Figure 5.2 illustrates the collaboration of relations, views, rules and triggers that are necessary for a split time-varying entity Entity_X. Relations Entity_X_static and Entity_X_timevar store the current data while Entity_X static hist and Entity_X timevar hist store the historic data. Relations Entity_X static and Entity_X timevar are joined together and form view V_Entity_X. This view is accessed by the application and all kinds of data
manipulations can be performed on it as if it were a database table. Rules are converting the data manipulation commands and propagating them to the relations $\text{Entity}_X\text{.static}$ and $\text{Entity}_X\text{.timevar}$.

Each modification of tuples in relations $\text{Entity}_X\text{.static}$ and $\text{Entity}_X\text{.timevar}$ causes firing a historize trigger. If a tuple in one of the current relations is updated, it is copied to the dedicated history relation before the update is performed. In case of a deletion, a tuple is removed from the current relation to the history relation. A boolean attribute deleted of the history relation indicates whether an tuple was deleted.

Each current relation has a dedicated history relation. Each of this pair of relations is combined to a read-only view. $\text{Entity}_X\text{.static}$ and $\text{Entity}_X\text{.static_hist}$ form the view $\text{V}_\text{Entity}_X\text{.static_hist}$ in our illustration. $\text{V}_\text{Entity}_X\text{.static_hist}$ and $\text{V}_\text{Entity}_X\text{.timevar_hist}$ form the basis for $\text{V}_\text{Entity}_X\text{.hist}$.

The reason for these intermediate views where current and history data are combined is that in $\text{V}_\text{Entity}_X\text{.hist}$ current tuples are joined together with historic tuples. Consider an entity is split into static and timevar relations. An update on this entity might affect only the time varying part of the entity. Thus no history entry exists for the static part although it is part of a historic version of the entity.

The proceeding subsections explain the relations, views, rules and triggers more detailed.

### 5.3.1 Common conventions

The following common conventions affect all types of relations: For all relations exists an identification column id of type bigint. The values for id are generated by calling a sequence `nextval('seq_Entity_X_static_id')`. This practice of using a generated number as primary key is widespread. It is especially convenient when inserting tuples into multiple related tables, because one knows the id of a tuple before inserting it. The next value from the sequence can be called by the application and added into an insert statement.
Or the value for \texttt{id} is empty in the insert statement and the default value defined for the \texttt{id} attribute calls the sequence.

Besides the primary key column \texttt{id} there are columns \texttt{tt\_begin} and \texttt{modified\_by} in order to keep track of from whom and when a tuple was entered, edited or deleted. The modifier data is set by a trigger. Additionally the field \texttt{modifier\_comment} is there for entering any comments about the reason a tuple was updated or deleted.

### 5.3.2 Static relations

For entities with static attributes, a \textit{static relation} of the following pattern exists:

```sql
CREATE TABLE Entity_X_static(
  id bigint NOT NULL DEFAULT nextval('seq_Entity_X_static_id'),
  /* static attributes */
  ...
  tt\_begin timestamp NOT NULL DEFAULT now(),
  modified\_by text,
  modifier\_comment text,
  CONSTRAINT Entity_X_static_pkey PRIMARY KEY (id)
);
```

In order to track data manipulation on this table a trigger function is implemented. The following snippets in the \texttt{historize} trigger function are responsible to write to the history table: For the case of deletion:

```sql
... INSERT INTO Entity_X_static_hist(
  Entity_X_static_id,
  /* static attributes */
  ...
  modified\_by,
  modifier\_comment,
  tt\_begin,
  tt\_end,
  deleted)
VALUES(
  OLD.id,
  /* static attributes */
  OLD. ... OLD.modified\_by,
  OLD.modifier\_comment,
  OLD.tt\_begin,
  current\_timestamp,
```
'true');
RETURN OLD;
...

For the update case:

IF /* any static attribute has changed */ THEN
INSERT INTO Entity_X_static_hist(
Entity_X_static_id,
/* static attributes */
... modified_by,
modifier_comment,
tt_begin,
tt_end,
deleted)
VALUES(
OLD .id,
/* static attributes */
OLD. ... 
OLD.modified_by,
OLD.modifier_comment,
OLD.tt_begin,
current_timestamp,
'false');
NEW.tt_begin := current_timestamp;
END IF;
RETURN NEW;
...

The history table looks as follows. It contains the same attributes than its current equivalent, yet the additional attributes \texttt{tt\_end} and \texttt{deleted}.

\begin{verbatim}
CREATE TABLE Entity_X_static_hist(
   id bigint NOT NULL DEFAULT nextval('seq_Entity_X_static_hist_id'),
   Entity_X_static_id bigint NOT NULL,
   /* static attributes */
   ...
   tt_begin timestamp NOT NULL,
   tt_end timestamp NOT NULL DEFAULT now(),
   modified_by text,
   modifier_comment text,
   deleted boolean NOT NULL DEFAULT false,
   CONSTRAINT Entity_X_static_hist_pkey PRIMARY KEY (id)
)
\end{verbatim}

Notice there is no foreign key constraint from the history to the current table since tuples can be deleted on the current table, but their history remains in the database.
5.3.3 Time-varying relations

Attributes which change over time are stored in one or more time-varying relation of the following pattern:

```sql
CREATE TABLE Entity_X_timevar(
    id bigint NOT NULL DEFAULT nextval(’. seq_Entity_X_timevar_id’),
    Entity_X_static_id bigint NOT NULL,
    /* timevar attributes */
    ...,
    vt_begin timestamp NOT NULL DEFAULT ‘- infinity’::timestamp,
    vt_end timestamp NOT NULL DEFAULT ‘infinity’::timestamp,
    tt_begin timestamp DEFAULT now(),
    modified_by text,
    modifier_comment text,
    CONSTRAINT Entity_X_timevar_pkey PRIMARY KEY (id),
    CONSTRAINT Entity_X_timevar_Entity_X_static_id_fkey FOREIGN KEY (Entity_X_static_id)
        REFERENCES Entity_X_static (id) MATCH SIMPLE
        ON UPDATE RESTRICT ON DELETE CASCADE
)
```

A foreign key constraint from the time-varying relation to the static relation is appropriate because each time-varying tuple belongs to a static tuple. Furthermore for each static tuple exists at least one time-varying tuple. The later constraint is assured by the insert rule on the view V_Entity_X.

The option ON DELETE CASCADE assures that if the static tuple gets deleted, the time-varying are deleted as well.

The history table for the time-varying relation looks as follows:

```sql
CREATE TABLE Entity_X_timevar_hist(
    id bigint NOT NULL DEFAULT nextval(’seq_Entity_X_timevar_hist_id’),
    Entity_X_timevar_id bigint NOT NULL,
    Entity_X_static_id bigint NOT NULL,
    /* timevar attributes */
    ...,
    vt_begin timestamp NOT NULL,
    vt_end timestamp NOT NULL,
    tt_begin timestamp NOT NULL,
    tt_end timestamp NOT NULL DEFAULT now(),
    modified_by text,
    modifier_comment text,
    deleted boolean NOT NULL DEFAULT false,
    CONSTRAINT Entity_X_timevar_hist_pkey PRIMARY KEY (id)
)
```
The trigger function for historization of the time-varying relation is analogue to the one of the static relation shown above.

### 5.3.4 Views

#### 5.3.4.1 The “reintegration” view

Every relation split into static and time-varying tables is joined together to a view, so that the application does not have to deal with multiple relations. The application performs INSERT, UPDATE and DELETE operation on this view as if it were a relation.

A view joining a static and a time varying relation of Entity_X together has the following structure:

```sql
CREATE OR REPLACE VIEW v_Entity_X AS
SELECT
    Entity_X_static.id AS Entity_X_static_id,
    Entity_X_timevar.id AS Entity_X_timevar_id,
    /* static attributes */
    ...
    /* timevar attributes */
    ...
    Entity_X_timevar.vt_begin,
    Entity_X_timevar.vt_end,
    CASE
        WHEN Entity_X_timevar.tt_begin > Entity_X_static.tt_begin
        THEN Entity_X_timevar.tt_begin
        ELSE Entity_X_static.tt_begin
    END AS tt_begin,
    CASE
        WHEN Entity_X_timevar.tt_begin > Entity_X_static.tt_begin
        THEN Entity_X_timevar.modified_by
        ELSE Entity_X_static.modified_by
    END AS modified_by,
    CASE
        WHEN Entity_X_timevar.tt_begin > Entity_X_static.tt_begin
        THEN Entity_X_timevar.modifier_comment
        ELSE Entity_X_static.modifier_comment
    END AS modifier_comment
FROM Entity_X_static
JOIN Entity_X_timevar ON Entity_X_static.id = Entity_X_timevar.
    Entity_X_static_id;
```

Entity_X_static_id and Entity_X_timevar_id form a primary key. Since
the administrative attribute \texttt{tt\_begin}, \texttt{modified\_by} and \texttt{modifier\_comment}
appear in both relations, the value of the tuple updated lastly is taken.

In case there are more than one time-varying relations, only tuples where
the \textit{valid time} overlap are shown. In the case of two time-varying relations 
\texttt{Entity\_X\_timevar} and \texttt{Entity\_X\_timevar2} the following additional condition
is part of the definition of the view \texttt{v\_Entity\_X}:

\begin{verbatim}
... WHERE "overlaps" (Entity\_X\_timevar\_vt\_begin, Entity\_X\_timevar\_vt\_end,
Entity\_X\_timevar2\_vt\_begin, Entity\_X\_timevar2\_vt\_end);
\end{verbatim}

In our approach, inserting a tuple into \texttt{V\_Entity\_X} can have two semantics.
In case \texttt{A}, a new object is inserted, which did not exist before in the database.
In the other case \texttt{B}, a new version of an already existing object is inserted.
In the first case \texttt{A}, a new tuple is inserted into the \textit{static} as well as into the
\textit{time-varying relations}. In the second case \texttt{B}, only a tuple is inserted into the
\textit{time-varying relations}.

Between the two different kinds of \texttt{INSERT} statements \texttt{A} or \texttt{B} can be dis-
tinguished via the \textit{primary key} of the \textit{static relation}. If a tuple is inserted with
a already existing \textit{static primary key}, it is a \texttt{B} and the \texttt{INSERT RULE} performs
an insert only into the \textit{time-varying relation}. The \textit{static relation} is updated.
If the \textit{primary key} in the \texttt{INSERT} is left out or it does not exist in the relation,
it is an \texttt{A} and tuples are inserted into the \textit{static} and \textit{time-varying relations}.

The rule performing this behaviour looks as follows:

\begin{verbatim}
CREATE OR REPLACE RULE "_INSERT" AS
ON INSERT TO v_Entity_X DO INSTEAD (INSERT INTO Entity_X_static (id, ..., modifier_comment)
SELECT NEW.Entity_X_static_id, ..., NEW.modifier_comment
WHERE NOT (EXISTS (SELECT *
FROM Entity_X_static
WHERE Entity_X_static.id = NEW.Entity_X_static_id));
UPDATE Entity_X_static SET
...=NEW. ..., 
modifier_comment = NEW.modifier_comment
WHERE Entity_X_static.id = NEW.Entity_X_static_id;
INSERT INTO Entity_X_timevar (id,
Entity_X_static_id, ...
,...

49
\end{verbatim}
5.3.4.2 History views

Current and historic data are kept in distinct tables. But since entities might split into several relations, a current tuple might be part of a historic version of an object. Therefore exists for every pair of current and history relations a view unifying the current and historic tuples:

```sql
CREATE OR REPLACE VIEW v_Entity_X_static_hist AS
SELECT Entity_X_static_hist.Entity_X_static_id,
       Entity_X_static_hist.tt_begin,
       Entity_X_static_hist.tt_end,
       Entity_X_static_hist.modified_by,
       Entity_X_static_hist.modifier_comment,
       Entity_X_static_hist.deleted
FROM Entity_X_static_hist
UNION ALL
SELECT Entity_X_static.id AS Entity_X_static_id,
       Entity_X_static.date_of_birth,
       Entity_X_static.tt_begin,
       Entity_X_static.modified_by,
       Entity_X_static.modifier_comment,
       'infinity'::timestamp AS tt_end,
       false AS deleted
FROM Entity_X_static;
```
CHAPTER 5. IMPLEMENTATION

For each object fragmented into several relations should be one history view providing the historic versions of the object. Similar to `v_Entity_X`, there is a `v_Entity_X_hist`.

```sql
CREATE OR REPLACE VIEW v_Entity_X_hist AS
SELECT
    v_Entity_X_timevar.Entity_X_static_id,
    v_Entity_X_timevar.Entity_X_timevar_id,
    /* static attributes */
    ...
    /* timevar attributes */
    ...
    v_Entity_X_timevar.vt_begin,
    v_Entity_X_timevar.vt_end,
    CASE
        WHEN v_Entity_X_timevar.tt_begin > v_Entity_X_static.tt_begin
        THEN v_Entity_X_timevar.tt_begin
        ELSE v_Entity_X_static.tt_begin
    END AS tt_begin,
    WHEN v_Entity_X_timevar.tt_end < v_Entity_X_static.tt_end
    THEN v_Entity_X_timevar.tt_end
    ELSE v_Entity_X_static.tt_end
    END AS tt_end,
    CASE
        WHEN v_Entity_X_timevar.tt_begin > v_Entity_X_static.tt_begin
        THEN v_Entity_X_timevar.modified_by
        ELSE v_Entity_X_static.modified_by
    END AS modified_by,
    CASE
        WHEN v_Entity_X_timevar.tt_begin > v_Entity_X_static.tt_begin
        THEN v_Entity_X_timevar.modifier_comment
        ELSE v_Entity_X_static.modifier_comment
    END AS modifier_comment,
    v_Entity_timevar_hist.deleted OR v_Entity_static_hist.deleted AS deleted
FROM v_Entity_X_static_hist
JOIN v_Entity_X_timevar_hist
    ON v_Entity_X_static_hist.Entity_X_static_id = v_Entity_X_timevar_hist.Entity_X_static_id
    AND v_Entity_static_hist.tt_end >= v_Entity_timevar_hist.tt_begin
    AND v_Entity_static_hist.tt_begin <= v_Entity_timevar_hist.tt_end;
```

5.3.5 Temporal Constraints

There are four constraints the time attributes must fullfill:

(1) $VT_{\text{begin}} < VT_{\text{end}}$

(2) $TT_{\text{begin}} < TT_{\text{end}}$
(3) **Valid time intervals** of the same fact must not overlap.

(4) **Transaction time intervals** of the same fact must not overlap.

Constraint (1) is assured by the following check constraint:

```sql
ALTER TABLE Entity\_X_timevar
ADD CONSTRAINT Entity\_X_timevar_check_begin_smaller_end
CHECK (vt\_begin < vt\_end);
```

Constraint (2) and (4) do not have to be checked explicitly because the transaction time values are always generated by the system. **tt\_begin** is never modified after a tuple is inserted. **tt\_end** is set after a tuple got overwritten or deleted. And this is of course after it was created. Overlapping of the transaction time intervals of a version of a can not occur because transaction time is consecutive.

Constraint (3) is more complex to redeem because the valid time values are entered by the users. Our approach for handling situations of overlappings intends to prioritize a new inserted tuple over the existing tuples. Therefore we have to make sure the valid time intervals of the existing tuples must not overlap with the new one. As introduced in section 3.2.4, there are 13 constellations two intervals can be arranged. In two cases, no action needs to be taken, because there is no overlapping. In all 11 other cases, the valid time interval of the existing tuples need to be shortened or the tuple needs to be replaced completely.

The numbering of overlapping refers to the illustration 4.4 on page 35. The function **time\_atom()** returns the *chronon* of the datatype **TIMESTAMP**, which is the shortest possible time interval.

```sql
CREATE FUNCTION check\_overlap\_Entity\_X\_timevar()
RETURNS "trigger" AS
DECLARE
pt\_record RECORD; /*a record for holding results in loop*/
a\_ts TIMESTAMP;
a\_id BIGINT;
BEGIN
/*Update all existing intervals that cause overlap conflict of types*/
-(10) y10 left-covered x
-(11) y11 right-overlaps x
-(12) y12 met x
*/
UPDATE Entity\_X\_timevar
```
CHAPTER 5. IMPLEMENTATION

```
SET vt_end=NEW.vt_begin-time_atom()
WHERE id<>NEW.id
  AND Entity_X_static_id=NEW.Entity_X_static_id
  AND NEW.vt_begin>vt_begin
  AND NEW.vt_begin<=vt_end
  AND NEW.vt_end>=vt_end;

/* Update all existing intervals that cause overlap conflict of type
    -(2) y2 meets x
    -(3) y3 left-overlaps x
    -(6) y6 right-covered x
*/
UPDATE Entity_X_timevar
SET vt_begin=NEW.vt_end+time_atom()
WHERE id<>NEW.id
  AND Entity_X_static_id=NEW.Entity_X_static_id
  AND NEW.vt_begin<=vt_begin
  AND NEW.vt_end>=vt_begin
  AND NEW.vt_end<vt_end;

/* Loop over all existing intervals that cause overlap conflict of type
    -(9) y9 covered x
*/
FOR pt_record IN
  SELECT * FROM Entity_X_timevar
  WHERE id<>NEW.id
  AND Entity_X_static_id=NEW.Entity_X_static_id
  AND NEW.vt_begin>vt_begin
  AND NEW.vt_end<vt_end
LOOP
  /* shorten the existing record */
  UPDATE Entity_X_timevar
  SET vt_end=NEW.vt_begin-time_atom()
  WHERE id=pt_record.id;

  /* copy the existing record */
  INSERT INTO Entity_X_timevar
  (Entity_X_static_id
    ...
    , vt_begin
    , vt_end
    , modified_by
    , modifier_comment)
  VALUES(pt_record.Entity_X_static_id
    , pt_record....
    , NEW.vt_end+time_atom()
    , pt_record.vt_end
    , pt_record.modified_by
    , pt_record.modifier_comment);
END LOOP;
```
/* DELETE all existing records which are completely overlapped by the new record (4, 5, 7, 8) */
DELETE FROM Entity_X_timevar
WHERE id <> NEW.id
AND Entity_X_static_id = NEW.Entity_X_static_id
AND NEW.vt_end >= vt_end
    AND NEW.vt_begin <= vt_begin;

RETURN NEW;
END;

5.4 Java Client

For administrating the study subjects of sesam a java client was realized. The capabilities of this application includes insertion, querying, viewing and modifying of study subjects, employees and other persons including their's addresses, phones and e-mail addresses.

5.4.1 Database interface

5.4.1.1 Hibernate and the persistency classes

Hibernate is a widespread relational persistency framework for Java and .NET applications [Hibernate, 2007]. The main benefit from using this framework is that the developer does not have to struggle with SQL statements, transactions and rollbacks. Once the data model is mapped with the framework, it is not required to write any SQL statements, but one can program (pure) Java to create, read and write data. Nevertheless Hibernate offers the possibility to write queries in SQL or in its own query language HQL.

In a Hibernate application, there is for each database table or view the application accesses, a corresponding Java class. Tables and views are mapped to Java classes using the Hibernate Persistency Framework. The mapping of the attributes, including the data type mapping from SQL to Java, is defined in Hibernate mapping files (hbm.xml). The classes are defined in XML and the accordant Java sources are generated by Hibernate Tools. These a generated classes, like e.g. Person features a private field for every attribute of the
corresponding database table or view, \texttt{V\_PERSON} in this case and for every field exists a public setter and getter method. Further settings like default values or necessary imports are also defined in the \textit{Hibernate mapping file}.

The \textit{Hibernate} persistency classes are packed in \texttt{org.sesam.subjectmgt.-bo}.

### 5.4.1.2 Relationships with Hibernate

\textit{Hibernate} is a powerful framework not just for loading and saving single tuples from and to the database. It also allows joining one-to-one and one-to-many relationship. For a one-to-many relation between person and address it is possible to define this relationship in XML and then access a person’s phone numbers easily by calling the \texttt{getPhones()} method of an instance of \texttt{Person} to receive a collection of \texttt{Phones}. Adding an \texttt{Phone} instance to an instance of a \texttt{Person} is done by just adding an instance to the collection and save it. \textit{Hibernate} fully takes care of setting the foreign key values in the affected objects and tuples. [Hibernate, 2007]

The speciality of our application is that for one person of the real-world exists not just only one instance in the database and therefore instances of the \texttt{Person} class, but there are several versions due to the temporal recording. There also exist several instances of the same phone because a phone as we define it in our application features time-varying attributes. Therefore the cardinality of the relationship between person and address is actually many-to-many. But the references between the relations do not point to a key but only a part of the key, typically the \texttt{static_id}.

Even though \textit{Hibernate} does support many-to-many relation ships, it requires the underlying data model properly maps foreign keys to primary keys. That means for a many-to-many relation ship between two tables a there would have to be a intermediate table. REF.

Figure 5.3 shows a part of our data model. Both the relation ship between \texttt{V\_person} and \texttt{V\_person\_phone} is a many-to-many relationship without a in- termediate relation, and thus the relation ship between \texttt{V\_person\_phone} and
Figure 5.3: Relationships among views

v_phone. The fact that these are views can be ignored at the moment since we made our views behave like tables.

Therefore we cannot use Hibernate for accessing dependent objects, but we have to load them separately when we need them.

Although the use of Hibernate is limited in our case, we use the capability of Hibernate for accessing depending collections only for getting the history objects of each current object. There we have a “conventional” one-to-many relationship. A person’s primary key is composed from a person_static_id and a person_timevar_id. The corresponding history object’s primary key consists of the same combination and tt_begin.

To conclude, we use Hibernate to load and save single objects and we make use of the Hibernate transaction management. Objects are save only through the static method DB.saveOrUpdate(...). We do not have to distinguish whether the tuple of a Java object already exists in the database or not. If we pass an object with an already existing primary key value, the corresponding tuple on the database is updated. If the object is new, Hibernate autonomous performs an insert statement on the database.

5.4.2 GUI classes

The application consists of one main window (class MainShell) and one person can be administrated at once (Figure 5.4). Depending on wheter the person is a study subject or an employee, a different set of tabs with detailed information is shown.
CHAPTER 5. IMPLEMENTATION

5.4.2.1 GUI composites

The different tabs and dialogs are composed of various composites, Java classes extending `org.eclipse.swt.widgets.Composite`. Each composite class consists of a constructor, with a `parent` and `style` parameter and optionally with a `mode` parameter in order to use the same class for different purposes. There is always a setter method, e.g. `setPerson(Person person)`, to pass over the object holding the data to be displayed. For composites having a form, there is a getter method, e.g. `Person getPerson()`, returning the modified object.

The GUI components of the main windows are part of package `org.-`
sesam.subjectmgt.gui. The dialogs and the composites used in dialogs are in package \texttt{org.sesam.subjectmgt.guidialogs}.

### 5.4.2.2 Edit dialogs

For editing data a there is an accordant button on each screen which pops up a dialog. Data can not be modified directly in the main window. there is a dialog for each modification. Dialogs are classes extending \texttt{org.eclipse.swt.widgets.Dialog}. They are opened by calling method \texttt{openNew} for entering a new object or \texttt{openEdit} for editing an already existing object. These methods return the object modified in the form, or \texttt{null}, if the editing was cancelled.

![Figure 5.5: Example of an edit dialog](image)

### 5.4.2.3 Search dialogs

Search dialogs are also classes extending \texttt{org.eclipse.swt.widgets.Dialog}. They are opened by calling method \texttt{open}() which returns either the object selected on the screen, or \texttt{null}, if the action was cancelled.
5.4.2.4 History dialogs

The pop-up windows displaying the history of each current object look all the in the same manner. Therefore the common display and behaviour is implemented in the abstract class BaseHistoryDialog. The history dialogs are in package org.sesam.subjectmgt.gui.dialogs.hist.

5.5 Tests

In order to test the correctness of the application behaviour, three kind of tests were implemented. JUnit tests were implemented to test certain database trigger functions, particularly the triggers assuring the valid time interval overlapping. The performance was tested by generating random data sets exceeding the amount of data the application is designed for by a factor 3. The general application behaviour and the correct displaying of the information on the graphical user interface was tested by users. Bugs encountered were entered into a project management tool, prioritized and fixed accordingly.
Figure 5.6: Example of an search dialog
Chapter 6

Summary and Conclusions

6.1 Summary

6.2 Conclusions

Designing and implementing a bitemporal on top of a conventional relational database One does not have to deal with n

For our application, the number of tuples is expected to be around 3000 per relation. Therefore we do not have to worry about performance issues. In applications with a larger amount of data, one would have to turn one’s attention to performance and index technologies.

6.3 Outlook

6.3.1 Use of supporting tools and frameworks

Defining relations and triggers involves a lot of manual work, which is a source of errors. The size of the present data model is just on the limit manual definition of views, rules and triggers is feasible. For larger project the use
of a framework supporting the automated generation of bitemporal relations, trigger functions, history view, etc. would be advisable.

There exist extensions supporting temporal data models on the database level. For PostgreSQL DBMS there is a project from the Ruby Community [Matsumoto, 2002]. It allows developers to define relations without time attributes. The framework takes care of the time attributes.

On the application level exist frameworks providing functionalities for time related applications. One such framework developed open-source is JTemporal. It features basic functionalities handling the most common temporal aspects [JTemporal, 2002].

### 6.3.2 Additional functionality

Further plausibility checks could be developed. Until now, only some basic checks of consistency are implemented, like $VT_{begin} < VT_{end}$ and the overlapping of valid time intervals introduced in section 5.3.5. Particularly plausibility checks for the inter subject relations and roles are advisable. Until now all possible and impossible relationship among study subjects can be entered into the application without warnings.
Appendix A

Database schema
Figure A.1: The persons schema
Figure A.2: The persons schema
Figure A.3: The persons schema
Appendix B

Java class hierarchy
B.1 Package Hierarchies

* org.sesam.subjectmgt * org.sesam.subjectmgt.bo * org.sesam.subjectmgt.bo.code
* org.sesam.subjectmgt.db * org.sesam.subjectmgt.gui * org.sesam.subjectmgt.gui.common
* org.sesam.subjectmgt.gui.dialogs * org.sesam.subjectmgt.gui.dialogs.hist

B.2 Class Hierarchy

APPENDIX B. JAVA CLASS HIERARCHY

(implies java.io.Serializable) o org.sesam.subjectmgt.bo.ContactConsent
(implies org.sesam.subjectmgt.bo.IBo, org.sesam.subjectmgt.bo.IContactConsent,
java.io.Serializable) o org.sesam.subjectmgt.bo.ContactConsentHist
(implies org.sesam.subjectmgt.bo.IBo, org.sesam.subjectmgt.bo.IContactConsentHist,
java.io.Serializable) o org.sesam.subjectmgt.bo.ContactingHistory
(implies org.sesam.subjectmgt.bo.IBo, org.sesam.subjectmgt.bo.IContactingHistory,
java.io.Serializable) o org.sesam.subjectmgt.bo.ContactingHistoryHist
(implies org.sesam.subjectmgt.bo.IBo, org.sesam.subjectmgt.bo.IContactingHistory,
java.io.Serializable) o org.sesam.subjectmgt.bo.ContactingHistoryHist
(implies org.sesam.subjectmgt.bo.code.ICodes) + org.sesam.subjectmgt.db.Codes
+ org.sesam.subjectmgt.db.DB + org.sesam.subjectmgt.db.DBCheckOverlaps
+ org.sesam.subjectmgt.db.DBCriteria
+ org.sesam.subjectmgt.db.DBCreateEntity + org.sesam.subjectmgt.db.DBCriteria
gui.dialogs.hist.AnnouncementHistoryDialogHist
+ org.sesam.subjectmgt.gui.dialogs.hist.InterSubjectRelationHistoryDialog + org.sesam.subjectmgt.
gui.dialogs.hist.InterSubjectRelationHistoryDialogHist
gui.dialogs.hist.PersonAddressHistoryDialogHist
+ org.sesam.subjectmgt.gui.dialogs.hist.PersonEmailAddressHistoryDialog + org.sesam.subjectmgt.
gui.dialogs.hist.PersonEmailAddressHistoryDialogHist
+ org.sesam.subjectmgt.gui.dialogs.hist.PersonPhoneHistoryDialog + org.sesam.subjectmgt.
gui.dialogs.hist.PersonPhoneHistoryDialogHist
+ org.sesam.subjectmgt.gui.dialogs.hist.StudySubjectHistoryDialog + org.sesam.subjectmgt.
gui.dialogs.hist.StudySubjectHistoryDialogHist
dialogs.search.SearchAddressDialog + org.sesam.subjectmgt.gui.sign.SearchAddressDialogHist
gui.sign.PersonSearchDialogHist
gui.sign.StudySubjectSearchDialogHist
+ org.sesam.subjectmgt.gui.common.IConstants + org.sesam.subjectmgt.gui.
dialogs.hist.PersonEmailAddressHistoryDialogHist
+ org.sesam.subjectmgt.gui.common.IConstants + org.sesam.subjectmgt.
gui.dialogs.hist.InterSubjectRelationHistoryDialogHist
+ org.sesam.subjectmgt.gui.common.IConstants + org.sesam.subjectmgt.
gui.dialogs.hist.PersonAddressHistoryDialogHist
+ org.sesam.subjectmgt.gui.common.IConstants + org.sesam.subjectmgt.
gui.dialogs.hist.PersonPhoneHistoryDialogHist
+ org.sesam.subjectmgt.gui.common.IConstants + org.sesam.subjectmgt.
gui.dialogs.hist.StudySubjectHistoryDialogHist
+ org.sesam.subjectmgt.gui.common.IConstants + org.sesam.subjectmgt.
gui.sign.SearchAddressDialogHist
+ org.sesam.subjectmgt.gui.common.IConstants + org.sesam.subjectmgt.
gui.sign.PersonSearchDialogHist
+ org.sesam.subjectmgt.gui.common.IConstants + org.sesam.subjectmgt.
gui.sign.StudySubjectSearchDialogHist
(implies java.io.Serializable) o org.sesam.subjectmgt.bo.Education
(implies org.sesam.subjectmgt.bo.IEducationHist) + org.sesam.subjectmgt.
bo.EducationHist
+ org.sesam.subjectmgt.bo.IEducation + org.sesam.subjectmgt.
bo.IEducationHist
+ org.sesam.subjectmgt.bo.EducationHist
APPENDIX B. JAVA CLASS HIERARCHY

APPENDIX B. JAVA CLASS HIERARCHY

APPENDIX B. JAVA CLASS HIERARCHY

o org.sesam.subjectmgt.gui.EmailComposite o org.sesam.subjectmgt.gui.EmployeeComposite
o org.sesam.subjectmgt.gui.common.HistoryButtonComposite o org.sesam.subjectmgt.gui.InterSubjectRelationComposite
o org.sesam.subjectmgt.gui.PersonalDataComposite o org.sesam.subjectmgt.gui.PhoneComposite
o org.sesam.subjectmgt.gui.PhoneTabComposite o org.sesam.subjectmgt.gui.RelationComposite
o org.sesam.subjectmgt.gui.RoleTableComposite o org.sesam.subjectmgt.gui.dialogs.SearchPersonComposite
o org.sesam.subjectmgt.gui.StatusBarComposite o org.sesam.subjectmgt.gui.StudyComposite
o org.sesam.subjectmgt.gui.SubStudyComposite o org.sesam.subjectmgt.gui.SummaryComposite
o org.sesam.subjectmgt.gui.dialogs.UnbornComposite

B.3 Interface Hierarchy

* org.sesam.subjectmgt.bo.IAddress * org.sesam.subjectmgt.bo.IAnnouncement
* org.sesam.subjectmgt.bo.IBo o org.sesam.subjectmgt.bo.IInterSubjectRelation
o org.sesam.subjectmgt.bo.IInterSubjectRelationHist + org.sesam.subjectmgt.bo.IInterSubjectRelation
* org.sesam.subjectmgt.bo.code.ICodes * org.sesam.subjectmgt.gui.common.IConstants
* org.sesam.subjectmgt.bo.IContactConsent * org.sesam.subjectmgt.bo.IContactingHistory
* org.sesam.subjectmgt.bo.IEducation * org.sesam.subjectmgt.bo.IEmailAddress
* org.sesam.subjectmgt.bo.IEmployee * org.sesam.subjectmgt.bo.IEmployeeWorking
* org.sesam.subjectmgt.bo.ILanguage * org.sesam.subjectmgt.bo.ILifeDate *
org.sesam.subjectmgt.bo.IParticipationStatus * org.sesam.subjectmgt.bo.IPerson
* org.sesam.subjectmgt.bo.IPersonAddress * org.sesam.subjectmgt.bo.IPersonAssociation
* org.sesam.subjectmgt.bo.IPersonEmailAddress * org.sesam.subjectmgt.bo.IPersonPhone
* org.sesam.subjectmgt.bo.IPhone * org.sesam.subjectmgt.bo.IPregnancy * org.sesam.subjectmgt.bo...
List of Figures

1.1 Spatial illustration of bitemporal data (following [Schmidt, 2005]) 5

2.1 The sesam logo ............................ 8

4.1 Split into static and time-varying relations ............... 32

4.2 A database consisting of a current relation (black) and a history relation (grey). ......................... 33

4.3 Our approach of a bitemporal database .................... 34

4.4 Interval overlaps [Tansel et al., 1993](p 71) ............... 35

4.5 The overlap issue with main addresses .................... 38

5.1 The architecture ................................ 41

5.2 Overview of database model ........................... 43

5.3 Relationships among views ............................ 56

5.4 The main window of the sesam Subject Management application displaying a study subject ......................... 57

5.5 Example of an edit dialog ................................ 58

5.6 Example of an search dialog ............................. 60

A.1 The persons schema .................................. 64

A.2 The persons schema .................................. 65
LIST OF FIGURES

A.3 The persons schema ........................................... 66
## List of Tables

1.1 Bitemporal database example (following [Wikipedia, 2005]) . . . 5

3.1 Synchronous time-varying attribute . . . . . . . . . . . . . . . . . . 20
3.2 Time normalization (following [Tansel et al., 1993](p 97)) . . . . 21
3.3 Temporal join example following [Gao et al., 2003](p 97)) . . . . 24
3.4 Result of temporal Cartesian product . . . . . . . . . . . . . . . . . . 25
3.5 Example: The snapshot equivalent . . . . . . . . . . . . . . . . . . . . 27
3.6 Example: Relation not in TNF . . . . . . . . . . . . . . . . . . . . . . . 27
3.7 Example: Time normalized . . . . . . . . . . . . . . . . . . . . . . . . . 28
Bibliography


