MASTER THESIS

im Rahmen des Universitätslehrganges “Geographical Information Science & Systems” (UNIGIS MSc) am Zentrum für GeoInformatik (Z_GIS) der Paris Lodron-Universität Salzburg

zum Thema

“Implementation of a History Extension to the Open Source Database Management System PostgreSQL/PostGIS”

vorgelegt von

Dipl.-Ing.(FH) Michael Wagner
U1343, UNIGIS MSc Jahrgang 2007

Zur Erlangung des Grades “Master of Science (Geographical Information Science & Systems) MSc(GIS)”

Gutachter:
Ao. Univ. Prof. Dr. Josef Strobl

Leipzig – October 5, 2009
Acknowledgements

I would like to thank the following persons who provided me with valuable support during my master studies and the process of preparing my thesis:

- My employer Dr. Gernod Schindler for the financial support and encouraging me in doing these studies.

- My colleague Volkmar for the constructive discussions, good ideas and for proof-reading this document.

- My partner Adina for cheering me up and encouraging me in times of little motivation, as well as for proof-reading this document.

- My family for their continuous encouraging support.

Last but not least I would like to thank very much the UNIGIS Salzburg Team for the excellent support and assistance during the entire time of the studies.
Declaration

Ich versichere, diese Master Thesis ohne fremde Hilfe und ohne Verwendung anderer als der angeführten Quellen angefertigt zu haben, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat. Alle Ausführungen der Arbeit die wörtlich oder sinngemäß übernommen wurden, sind entsprechend gekennzeichnet.

Leipzig – October 5, 2009
Abstract

In many fields of application it is crucial to keep track of the changes occurring to objects in the real world in the particular domain. The division of a cadastral parcel in the sector of land administration is an example of such change. Even after the division, resulting in two or more successors, it is important to know how the original parcel looked like and when it existed. If such “historical” information is to be retained in a database, particular concepts have to be applied to allow for this.

Within the scope of this thesis a prototype of a history extension to the Open Source database management system PostgreSQL / PostGIS was implemented. The aim was to provide support for an automated maintenance of valid time and the related history, with valid time being the time when a fact happens or becomes true in the real world (Jensen et al., 1994). It was assumed that in many domains valid time is of greater interest than transaction time, i.e. the time when an information becomes known to the system (Jensen et al., 1994). The challenge was to implement the functionality required for maintenance of a history entirely on the database server. This way no modification of the client application would be necessary. Furthermore the valid-time period of a feature was to be stored as line, taking into consideration that time can be represented as geometry (Künzel, 2008; ISO, 2002b). Hence, the spatial functions of PostGIS could be used to perform temporal analysis.

With regard to time-oriented statements in databases three types can be differentiated: current (now), sequenced (at each instant of time) and non-sequenced ones (ignoring time) (Snodgrass, 2000). Sequenced statements are the most useful ones but also most complex to express in SQL. A sequenced update actually requires five SQL statements while a sequenced deletion requires four. Within the extension support for sequenced statements was implemented. Sequenced statements cover current statements implicitly. Given a temporal table the standard SQL PRIMARY KEY construct cannot be used any longer. A sequenced primary
key is required, one that applies at each instant of time. Such key must be ensured and implemented by a sequenced constraint. The same concept applies to foreign keys and referential integrity. Thus, a sequenced foreign key is necessary, again ensured by a constraint.

Converting an existing user table without temporal support to a temporal table required some crucial modifications of that table. First any existing primary or foreign key constraint was dropped. Next a column of type geometry was added to the table to record the valid-time period of the rows. After that the table was renamed and a view created on the table which was given the original name of the table. The view was made “updatable” by creating an INSERT, UPDATE and DELETE rule on it. Within each rule the actual SQL statements required for a sequenced modification (five for an update, four for a deletion and one for an insertion) are executed on the underlying table. Furthermore, triggers were created on the table that ensure the sequenced primary and foreign key(s). Besides that, a number of supportive database procedures were implemented that are required e.g. to convert time to geometry and vice versa. The entire changes necessary to make a table temporal, are basically hidden from the client. The client still finds the user table under the same name (although its actually a view), and also original primary and foreign key(s) are retained (implemented as sequenced keys). The only obvious change for the client application is the need to set the period of applicability (by calling a single database procedure) before performing a database operation. Once set, this period serves as a filter for all subsequent queries and modifications on the user table. The client application continues working with the database the way it did as without temporal support but behind the “scenes” a full history of the data is maintained.
Kurzfassung


In Bezug auf temporale Anweisungen in Datenbanken kann zwischen drei Arten unterschieden werden: aktuelle (jetzt), sequentielle (zu jedem Zeitpunkt) und nicht-sequentielle (Zeit wird ignoriert). Sequentielle Anweisungen sind am zweckdienlichsten aber auch am schwierigsten in SQL auszudrücken. Eine sequentielle Aktualisierung erfordert effektiv fünf SQL Anweisungen während eine sequen-
tielle Lösung vier Anweisungen erfordert. Das Historie-Modul wurde für die Unterstützung sequentieller Anweisungen entwickelt. Sequentielle Anweisungen beinhalten aktuelle Anweisungen implizit. In einer Tabelle mit Unterstützung für Gültigkeitszeit kann das SQL PRIMARY KEY Konstrukt nicht länger verwendet werden. Ein sequentieller Primärschlüssel ist erforderlich, d.h. ein Schlüssel, der zu jedem Zeitpunkt ein Schlüssel ist. Solch ein Schlüssel muss durch eine sequentielle Integritätsbedingung gewährleistet werden. Das gleiche Konzept gilt für Fremd-
schlüssel und referentielle Integrität. Ein sequentieller Fremd schlüssel ist erforder-
lich und wird wieder durch eine sequentielle Integritätsbedingung gewährleistet.

Die Konvertierung einer bestehenden Anwendertabelle ohne Unterstützung für temporale Datenhaltung in eine Tabelle mit entsprechender Unterstützung er-
forderte einige tiefgreifende Änderungen an der Tabelle. Zuerst mussten sämtliche
bestehenden Integritätsbedingungen für Primär- und Fremd schlüssel entfernt wer-
den. Danach wurde eine neue Spalte vom Datentyp “geometry” hinzugefügt, die zur Erfassung der Gültigkeitszeitspanne von Objekten benötigt wird. An-
schliessend wurde die Tabelle umbenannt und eine Ansicht (View) auf die Tabelle
erstellt, die den ursprünglichen Namen der Tabelle erhielt. Diese Ansicht wurde
aktualisierbar gemacht, indem entsprechende INSERT-, UPDATE- und DELETE-
Regeln (Rules) für die Ansicht erzeugt wurden. In jeder Regel werden die eigentlichen
SQL Anweisungen, die für eine sequentielle Modifikation notwendig sind, auf der
der Ansicht zugrunde liegenden Tabelle ausgeführt (fünf für eine Aktualisierung,
vier für eine Löschung und eine für eine Einfüge-Operation). Weiterhin wurden für
die Tabelle Trigger erstellt, um die sequentiellen Primär- und Fremd schlüssel zu
gewährleisten. Zusätzlich sind eine Reihe von Hilfsprozeduren für die Datenbank
entwickelt worden, z.B. um Zeit in Geometrie zu konvertieren und umgekehrt.
Die Änderungen, die an einer Tabelle notwendig sind um temporale Datenhal-
tung zu ermöglichen, sind für die Client-Anwendung weitgehend unsichtbar. Die
Client-Anwendung kann die Anwendertabelle weiterhin unter dem gleichen Na-
men ansprechen (auch wenn eigentlich die Ansicht angesprochen wird) und auch
die ursprünglichen Primär- und Fremd schlüssel bleiben erhalten (sind aber als
sequentielle Schlüssel implementiert). Die einzig offensichtliche Änderung für die
Client-Anwendung ist die Notwendigkeit, den Anwendungszeitraum (Period of Ap-
plicability) für temporale Anweisungen zu setzen, bevor eine Datenbank-Operation
ausgeführt wird. Dies geschieht durch den Aufruf einer einzigen Datenbankproze-
dur. Einmal gesetzt, ist der Anwendungszeitraum ein Filter für alle nachfolgen-
den Abfragen und Änderungen von Daten in der Anwendertabelle. Die Client-
Anwendung nutzt die Datenbank in der gleichen Art und Weise wie zuvor (d.h. zum Zeitpunkt ohne Unterstützung für temporale Datenhaltung), im Hintergrund wird aber eine komplette Historie geführt und gewährleistet.
Contents

List of Figures xii
List of Tables xiii
List of Listings xv
Acronyms xvi

1 Introduction 1
  1.1 Motivation ................................................. 1
  1.2 Hypothesis ............................................... 2
  1.3 Historical Versioning ................................. 2
    1.3.1 Basics ............................................. 2
    1.3.2 Implementation .................................. 3
  1.4 Expected Results .............................. 5
  1.5 Issues Not Covered .................................. 5
  1.6 Audience ................................................. 6
  1.7 Structure of Thesis ................................ 6

I Background Research 7

2 Fundamentals of Time 8
  2.1 Definitions ............................................... 8
  2.2 Time as Dimension .................................. 9
  2.3 Temporal Data Types ................................ 10
    2.3.1 Instants .......................................... 10
    2.3.2 Intervals .......................................... 10
    2.3.3 Periods .......................................... 10
  2.4 Relative Position .................................. 11
## CONTENTS

### 2.4 Temporal Relationships
- 2.4.1 Temporal Relationships .......................... 12
- 2.4.2 Spatial Relationships .............................. 14
- 2.4.3 Analogy of Temporal and Spatial Relationships .. 16

### 2.5 Temporal Reference Systems
- 2.5.1 Calendars and Clocks ............................ 18
- 2.5.2 Ordinal Reference System .......................... 19
- 2.5.3 Temporal Coordinate System ........................ 19

### 2.6 Other Aspects
- 2.6.1 Branching of Time .................................. 20
- 2.6.2 Feature Succession .................................. 21

### 3 Time in Databases
- 3.1 Support for Temporal Data Types ....................... 22
  - 3.1.1 Instants ........................................... 22
  - 3.1.2 Intervals .......................................... 23
  - 3.1.3 Periods ........................................... 23
- 3.2 Kinds of Time .......................................... 23
  - 3.2.1 Transaction Time .................................. 23
  - 3.2.2 Valid Time ........................................ 24
  - 3.2.3 User-Defined Time ................................ 24
  - 3.2.4 Classification of Temporal Databases ............... 25
- 3.3 Version Management .................................... 25
- 3.4 Temporal Keys ......................................... 27
  - 3.4.1 Candidate Keys and Functional Dependency .......... 27
  - 3.4.2 Sequenced Primary Key ............................ 28
  - 3.4.3 Duplicates ........................................ 29
  - 3.4.4 Referential Integrity ................................ 30
- 3.5 Time-Oriented Statements ............................. 33
  - 3.5.1 SELECT ........................................... 33
  - 3.5.2 INSERT ........................................... 34
  - 3.5.3 UPDATE ........................................... 34
  - 3.5.4 DELETE ........................................... 36

### 4 Implementation Samples
- 4.1 Oracle Workspace Manager 11g ........................ 38
- 4.2 PostgreSQL Time Travel ................................ 40
CONTENTS

7.5.1 Example 1 .............................................. 73
7.5.2 Example 2 .............................................. 76

8 Conclusion .............................................. 81
  8.1 Findings ............................................... 81
  8.2 Open Issues ......................................... 82
  8.3 Outlook ............................................... 84

A Database Procedures .................................. 85

Bibliography .............................................. 98
List of Figures

1.1 Modified database schema ................................................. 4
2.1 Temporal geometric primitives ........................................ 12
2.2 Temporal relationships between two periods ....................... 13
2.3 Period operations .......................................................... 14
2.4 The DE-9IM ................................................................. 15
2.5 Cell values for the DE-9IM ............................................... 15
2.6 Temporal reference systems .............................................. 18
2.7 Temporal coordinate system ............................................. 20
2.8 Data type $TM\_Coordinate$ .............................................. 20
2.9 Bidirectional branching of time ....................................... 21
3.1 Valid and transaction time ............................................... 24
3.2 Steps of a sequenced update ......................................... 35
3.3 Steps of a sequenced deletion ....................................... 37
6.1 Temporal Coordinates .................................................... 51
6.2 Target rows for sequenced update .................................. 52
6.3 Target rows for sequenced deletion ................................ 54
6.4 Foreign key violations ................................................... 56
6.5 Conceptual Data Model .................................................. 57
7.1 Test Scenario – Example 1 .............................................. 73
7.2 Example 1 – Result ....................................................... 76
7.3 Lineage of parcel '31' ..................................................... 77
7.4 Test Scenario – Example 2 .............................................. 78
7.5 Example 2 – Failed, and successful insertion attempt .......... 79
7.6 Example 2 – Result ....................................................... 79
7.7 Deletion attempt on parcel '31' ....................................... 80
List of Tables

3.1 Classification of temporal databases .................................. 25
3.2 personNr as candidate key ............................................. 27
3.3 Extract of table landuse .................................................. 28
3.4 Extract of table parcel .................................................... 31

7.1 tm_coordinatesystem ..................................................... 59
7.2 tm_config ................................................................. 60
7.3 tm_validtime ............................................................. 60

8.1 A case for coalescing ..................................................... 83
8.2 Result of coalescing ....................................................... 84
## List of Listings

3.1 Creating a *Period* data type ........................................... 23
3.2 Attempting to create a temporal key ................................. 28
3.3 Expressing a sequenced primary key .................................. 28
3.4 Preventing value equivalence ......................................... 29
3.5 Creating a foreign key .................................................. 30
3.6 Creating a sequenced foreign key .................................... 31
3.7 Creating a current foreign key ....................................... 32
3.8 Extracting the current state .......................................... 33
3.9 Extracting the state at any time ..................................... 33
3.10 Non-sequenced query .................................................. 34
3.11 Sequenced insertion .................................................... 34
4.1 Version-enabling a table .............................................. 39
4.2 Setting the period of applicability .................................. 39
5.1 Creating a view ......................................................... 43
5.2 Creating a trigger ....................................................... 43
5.3 Creating a rule .......................................................... 44
5.4 Creating a line from WKT .............................................. 45
5.5 Using the “&&” operator .................................................. 46
6.1 Finding the target rows of 1\textsuperscript{st} INSERT ............. 52
6.2 Finding the target rows of 2\textsuperscript{nd} INSERT ................. 53
6.3 Finding the target rows of 1\textsuperscript{st} UPDATE ................. 53
6.4 Finding the target rows of 2\textsuperscript{nd} UPDATE ................. 53
6.5 Finding the target rows of 3\textsuperscript{rd} UPDATE ................. 53
6.6 Finding the target rows of INSERT ................................... 54
6.7 Finding the target rows of 1\textsuperscript{st} UPDATE ................. 54
6.8 Finding the target rows of 2\textsuperscript{nd} UPDATE ................. 55
6.9 Finding the target rows of DELETE ................................... 55
6.10 Testing for primary key violations ................................... 55
6.11 Testing for foreign key violations ...................... 57
7.1 Creating the parcel table ................................ 60
7.2 Creating the landuse table ............................... 61
7.3 Creating a data type tm_coordinate ...................... 61
7.4 Initial table modification ................................ 63
7.5 Creating a view ............................................ 64
7.6 Creating an INSERT rule .................................. 65
7.7 Creating an UPDATE rule ................................. 65
7.8 Creating a DELETE rule ................................... 67
7.9 Trigger procedure ensuring the sequenced primary key ..... 68
7.10 Trigger procedure ensuring the sequenced foreign key (a) .... 69
7.11 Trigger procedure ensuring the sequenced foreign key (b) .... 71
7.12 Setting the temporal coordinate system ............... 72
7.13 Example 1 – Case A .................................... 74
7.14 Example 1 – Case B .................................... 74
7.15 Example 1 – Case C .................................... 74
7.16 Example 1 – Case D .................................... 75
7.17 Example 1 – Lineage .................................... 75
7.18 Example 2 – Case A .................................... 76
7.19 Example 2 – Case B .................................... 77
7.20 Example 2 – Case C .................................... 77
7.21 Example 2 – Deletion attempt ......................... 78
A.1 transformDateTime(timestamp with time zone):tm_coordinate ... 85
A.2 transformCoord(tm_coordinate):timestamp with time zone .... 87
A.3 lineFromTimeHelper(timestamp with tz, timestamp with tz) ... 90
A.4 setPA(timestamp with tz, timestamp with tz) ................ 91
A.5 paFrom():timestamp with time zone .................... 92
A.6 paTill():timestamp with time zone ..................... 93
A.7 lineFromPA():geometry .................................. 93
A.8 lineFromTime(timestamp with tz, timestamp with tz):geometry ... 93
A.9 validFrom(geometry):timestamp with time zone .......... 95
A.10 validTill(geometry):timestamp with time zone .......... 95
A.11 getValidTimestamp(int4):timestamp with time zone ...... 96
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>DDL</td>
<td>Data Definition Language</td>
</tr>
<tr>
<td>DML</td>
<td>Data Manipulation Language</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standardization Organization</td>
</tr>
<tr>
<td>OGC</td>
<td>Open Geospatial Consortium</td>
</tr>
<tr>
<td>PL</td>
<td>Procedural Language</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>TZ</td>
<td>Time Zone</td>
</tr>
<tr>
<td>UTC</td>
<td>Coordinated Universal Time</td>
</tr>
<tr>
<td>WKB</td>
<td>Well Known Binary</td>
</tr>
<tr>
<td>WKT</td>
<td>Well Known Text</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

In many fields of application it is crucial to keep track (maintain a *history*) of the changes occurring to objects in the real world. The division of a cadastral parcel in the domain of land administration is an example of such change. Even after the division, resulting in two or more successors, it is important to know how the original parcel looked like and when it was existing. If such “historical” information is to be retained in a database, particular concepts have to be applied to allow for this.

In databases where history of information is not considered explicitly, it is usually assumed that the stored data are the currently valid ones. A change occurring on an object in the real world would lead to an update of the related information in the database and simply replace the information previously available on that object. Thus, history is nonexistent.

The aim of this thesis is to develop an extension to the Open Source database management system (DBMS) PostgreSQL/PostGIS that enables users to maintain a history of the data (and thus, the modelled reality). The idea and challenge is to implement the functionality for maintaining history entirely on the database server. Hence, little or no modification of the client application is required in regard to the maintenance of history. The extension will be implemented using database functionality as triggers, stored procedures, etc.

The topic of the thesis is highly related to the requirements from practice. Its focus is rather on implementation of a history extension than on a pure study and analysis of the subject *history*. Geometry is closely related to time and history.
in the context of this thesis. The reasons for this are twofold. Firstly, the thesis focuses on maintenance of history in the domain of land administration. This inherently involves spatial information, i.e. information that usually has a geometry component. Secondly, time will be stored as geometry in the database since it can be represented as such. This concept is explained in detail in the work of Künzel (2008) and in the ISO standard 19108 (ISO, 2002b). By representing time as geometry, spatial functions and operators can be applied to time. These functions and operators are already available with the PostGIS extension to PostgreSQL. The assumption is that advanced analysis of time, which will be required for the implementation of the history extension can be facilitated by using the PostGIS capabilities.

1.2 Hypothesis

In many cases a customized client application is required for an automated maintenance of valid time and data history. The database is often used as a simple data store with its full capabilities not exploited.

Within this thesis it is expected that support for an automated maintenance of valid time and history can be implemented entirely on the database server. No modification or customization of the client software is required. Valid time is stored as geometry and only spatial operators and functions are required to test for temporal relationships.

1.3 Historical Versioning

This section briefly describes the concepts which are applied to implement the history extension. These concepts are explained in detail in the subsequent chapters of this document.

1.3.1 Basics

The history that can be maintained with the extension will consider valid time only, i.e. the time when a fact became true in the modelled reality (Jensen et al., 1994). According to the classification of Worboys (1993) such database would be a historic database. The extension will also allow storing information whose valid
time starts in the future although it will not support branching of time, e.g. to simulate various what-if scenarios.

Objects stored in the database are called features henceforth. Features are an abstraction of real world phenomena (e.g. of a parcel or building). The valid-time span of a feature is defined by two events, begin and end. The events occur because of a change happening to the feature. In regard to the temporal data type the time span is a period while the events are of type instant. An interval scale will be used to measure time as it can be used to describe both, ordering of values and distances between values. The distance depends on the granularity which is used to measure time and which is defined by the chronon, i.e. the shortest interval used as measure for a period and for the definition of scale intervals (Ott and Swiaczny, 2001).

Time and space are similar in a way that both have geometry and topology. A point in time occupies a position that can be identified in relation to a temporal reference system. Distance can be measured. Unlike space, however, time has a single dimension (ISO, 2002b). Within this thesis instants in time will be represented as point geometry while periods will be represented as lines.

Between features a temporal association can exist called feature succession. Feature succession is the replacement of one set of feature instances by another set of feature instances. The replacement implies that the life spans (valid time) of the feature instances in the first set come to an end at the instant when the life spans of feature instances in the second set begin (ISO, 2002b). The division of a cadastral parcel resulting in two new parcels is an example of feature succession. Feature succession shall be supported by the history extension.

1.3.2 Implementation

With the extension to the database, historical information will be retained by applying tuple-level versioning. Thus, historical information will be kept as additional tuples (rows) in the according table(s). Usually the period of validity for a row would be stored by appending two timestamp columns to each table for which history shall be maintained. One column would specify when a row became valid and one would specify when the row stopped being valid. Considering the similarity of space and time, the period of validity will be represented and stored as geometry, in particular as a line, with the history extension. Testing the relationship between two periods of validity can be done using the spatial relationship
functions as overlaps, within, etc. which are provided with the PostGIS extension to PostgreSQL.

When working with temporal databases three basic kinds of time-oriented statements exist; current (now), sequenced (at each instant of time) and non-sequenced (ignoring time) (Snodgrass, 2000). Out of the three, sequenced statements are the most natural and useful ones but also most difficult to express in SQL. With sequenced statements a logical deletion would actually result in an INSERT, two UPDATE and a DELETE statement while a logical update would result in two INSERT and three UPDATE statements. When adding time support to a table the original primary key must be changed, a temporal primary key is required. This inherently affects also referential integrity if a table is referenced by or referencing another table. To express temporal keys and ensure referential integrity, a number of constraints have to be applied to the involved tables.

The history extension will implement support for sequenced statements. All this shall be hidden from any client application that would just send a SELECT, INSERT, UPDATE or DELETE command as with a non-temporal database. The extension will make extensive use of triggers, views and rules as the main functionalities to implement support for valid time in the database. The table for which support for valid time is to be added will be renamed and a view created with the original name of the table (Figure 1.1).

![Figure 1.1: Modified database schema](image)

Rules will be generated to make the view “updatable”, allowing for insertions,
updates and deletions. Within the rules the actual statements that are required for a sequenced modification would be executed (four for a deletion, five for an update). Temporal keys and referential integrity will be ensured via triggers applied to the table. The main steps to add (automated) valid time support to a table are these:

1. Add a column of type geometry to record the valid time period of features.
2. Drop the original primary key constraint and, if applicable, foreign key constraint.
3. Rename the table.
4. Create a view on the table, given the original name of the table.
5. Create rules on the view for INSERT, UPDATE and DELETE.
6. Create triggers on the table to ensure the temporal primary key and, if applicable, referential integrity.

These steps are required for this particular implementation of a history extension to PostgreSQL. Different steps might be necessary with other database management systems and implementations for valid-time support.

### 1.4 Expected Results

The result of the work carried out in the context of this thesis shall be a prototype for a history extension to PostgreSQL/PostGIS DBMS. This extension will support sequenced modifications (covering insertions, updates and deletions), taking into account also referential integrity between temporal tables. Feature succession as e.g. the division of a cadastral parcel, will be supported. The extension will store the valid-time period of features as geometry and will make use of spatial functions and operators for the entire required temporal analysis.

### 1.5 Issues Not Covered

Since time is stored as geometry within the history extension, performance issues might come up when carrying out extensive temporal analysis via spatial functions and operators on a vast amount of data. On the other hand, a valid-time period is
stored as a simple line consisting of two vertices only, and indices are created for time as geometry to increase the speed of operations. Hence, performance issues might be negligible. Nevertheless, no particular performance test will be carried out in the context of this thesis.

This thesis is about historical versioning, not to be confused with just *versioning* which is a term commonly used for *concurrent versioning*. Concurrent versioning is a concept allowing multiple users to work on the same data simultaneously by creating a version of the data for each user. These versions can be merged later on again taking into account certain strategies.

### 1.6 Audience

This thesis addresses anybody who would like or is required to implement or maintain a historic database. It could be of interest for those who have no possibility to modify/customize their client application to add valid-time support accordingly. The thesis provides a prototype implementation which could serve as an idea or base for the development of a production system. As valid time is stored as geometry it is assumed that someone who would like to implement a similar extension, uses a DBMS with a spatial extension.

### 1.7 Structure of Thesis

Following this chapter, fundamentals of time are discussed. The similarity of space and time is considered and the representation of time as geometry is inherently analysed.

Chapter 3 discusses time in databases covering data types for temporal support, time-oriented statements, temporal keys and referential integrity as well as classifications of temporal databases. Two sample applications for temporal support in databases are looked into in Chapter 4.

In Chapter 5 particular functionalities of the PostgreSQL DBMS and its spatial extension PostGIS are examined, which are applied to implement the history extension. Chapters 6 and 7 cover concept and implementation of the actual prototype of the history extension.

The findings of the thesis and an outlook are discussed in Chapter 8.
Part I

Background Research
Chapter 2

Fundamentals of Time

This chapter discusses some of the fundamentals of time that are required for a full understanding of the subsequent sections of this document.

While there are numerous religious and philosophical points of view in regard to what is time, there are two main perspectives of time in physics. From the Newtonian perspective, time is considered as an independent dimension (i.e. independent from space). Einstein introduced the relativistic perspective in which space and time are perceived as interdependent and inseparable.

In the context of this thesis, time is looked at from the Newtonian perspective and considered as an independent dimension. Nevertheless, there is a lot of similarity between space and time, e.g. both, space and time are continuous and infinite dimensions. To store time as discrete and finite data in a database its complexity must be reduced. Some concepts are provided within this chapter.

2.1 Definitions

Within this thesis the following definitions apply.

**chronon** the smallest interval used as a measure for a period and for the definition of the scale intervals used to define the time of an event on the time axis (Ott and Swiacean, 2001)

**event** action which occurs at an instant (ISO, 2002b)

**feature** abstraction of real world phenomena (ISO, 2002a)
CHAPTER 2. FUNDAMENTALS OF TIME

granularity of time the level of precision and uncertainty with which time can be depicted; depends on the smallest chronon / granule applied (Ott and Swiaczny, 2001)

granule the same as a chronon

interval scale scale with an arbitrary origin which can be used to describe both ordering of values and distances between values (ISO, 2002b)

life span period during which something exists (ISO, 2002b)

ordinal scale scale which provides a basis for measuring only the relative position of an object (ISO, 2002b)

2.2 Time as Dimension

In ISO (2002b) it is stated that “Time is a dimension analogous to any of the spatial dimensions. Like space, time has geometry and topology. A point in time occupies a position that can be identified in relation to a temporal reference system. Distance can be measured. Unlike space, however, time has a single dimension — temporal reference systems are analogous to the linear referencing systems that are used to describe spatial position for some kinds of applications. Although time has an absolute directionality — movement in time is always forward — time can be measured in two directions.”

This concept of time as a linear dimension is also supported by the taxonomic model of Frank (1998). If time is treated as a linear dimension, a consecutive development on the time axis is implied with no point in time occurring more than once. Furthermore, Frank describes that time can be treated as cyclic dimension and hence, has reoccurring character (“every week”, “every Monday”, etc.).

Time can be measured on ordinal and interval scales. An ordinal scale provides information only about relative position in time, which might be sufficient for certain fields of application as geology or archaeology. An interval scale supports both, the ordering of values and the measurement of distances between them.
2.3 Temporal Data Types

2.3.1 Instants

Considering time as a linear dimension, an instant is an anchored location on the time line. An instant occurs once, and then is forever in the past (Snodgrass, 2000). June 11, 2009 at 2:00 a.m. would be an example for an instant.

Geometry of an instant  According to ISO (2002b) an instant is a temporal object in general and a 0-dimensional geometric primitive that represents position in time, in particular. In the latter an instant is equivalent to a point in space. The number of dimensions is based on the Simple Feature Access Specification of the Open Geospatial Consortium (OGC) in which a point is considered as 0-dimensional, a line as 1-dimensional and a polygon as 2-dimensional (OGC, 2006a). The instant as a temporal object has one attribute position which defines the position of the instant in relation to a temporal reference system.

2.3.2 Intervals

An interval is an unanchored contiguous portion of the time line and in that is a duration. An interval is relative in contrast to an instant, being absolute. Adding an interval to an instant will result in another instant. Inherently, the distance between two instants is an interval. Intervals have a direction. An interval can be positive or negative, indicating a shift to the future or to the past (Snodgrass, 2000). “Two months” is an example for an interval.

In the context of ISO (2002b) an interval is a data type (“TM.Duration”) used for describing length or distance in the temporal dimension.

2.3.3 Periods

A period is an anchored duration of the time line. To express that a fact in the database is true over a duration of time, a period is associated with that fact. A period is commonly represented by a pair of instants, usually of identical granularity. Even with a pair of instants various representations of a period are possible (Snodgrass, 2000). As an example the period from February 1, 2000 until June 30, 2000 is shown (both dates shall be included in the period):
Closed-closed period: The delimiting instants are in the period. Denoted with square brackets e.g. [DATE '2000-02-01', DATE '2000-06-30']

Closed-open period: The second instant represents the granule immediately following the last granule of the period. Denoted with a square and a round bracket e.g. [DATE '2000-02-01', DATE '2000-07-01')

Open-open period: Both delimiting instants are excluded from the period. Denoted with round brackets e.g. [DATE '2000-01-31', DATE '2000-07-01']

Open-closed period: The first instant represents the granule immediately preceding to the first granule of the period. Denoted with a round and a square bracket e.g. (DATE '2000-01-31', DATE '2000-06-30']

Periods could as well be represented by providing a starting date and an interval e.g. (DATE '2000-02-01', INTERVAL '25' DAY). Combinations are possible again. The preferred representation of a period is the closed-open one. It avoids the need for the “<=" operator when doing comparison and the need to add one granule ("+ 1") whenever a period is constructed from a date and an interval.

Geometry of a period  According to the definition in ISO (2002b) a period is a temporal object in general and a 1-dimensional geometric primitive that represents extent in time, in particular. In the latter a period is equivalent to a curve in space. Like a curve, it is an open interval bounded by beginning and end points (instants), and has length (duration). Its location in time is described by the temporal positions of the instants at which it begins and ends (Figure 2.1); its duration equals the temporal distance between those two temporal positions.

As a constraint the temporal position of the beginning of a period must be less/earlier than the temporal position of the end of the period.

2.4 Relative Position

In certain situations it is more relevant determining the relative position of two temporal objects (instants or periods) to each other than knowing their absolute position on the time line. Thirteen relative temporal relationships have been identified by Allen (1983) and are described in the next section.

Within OGC (2006a) topological spatial relationships are defined that can occur between two geometric objects. Since spatial and temporal objects are very
similar in that they have both, geometry and topology, it can be assumed that there is some analogy also among the temporal and spatial relationships. The identified analogies are described in Section 2.4.3.

2.4.1 Temporal Relationships

Between one instant (a) and another one (b) only three relationships can occur. The description of the relationships is based on the definition of the temporal objects \( TM_{\text{Instant}} \) and \( TM_{\text{Period}} \) as shown in Figure 2.1.

- a Before b \( \Leftrightarrow \) a \( Before^{-1} \) b \( \Leftrightarrow \) b After a if \( a.position < b.position \)
- a Equals b \( \Leftrightarrow \) b Equals a if \( a.position = b.position \)
- a After b \( \Leftrightarrow \) a \( After^{-1} \) b \( \Leftrightarrow \) b Before a if \( a.position > b.position \)

Between one period a and another period b thirteen relationships can occur as described here and shown in Figure 2.2 (period a in red color, period b in blue color).

- a Before b \( \Leftrightarrow \) a \( Before^{-1} \) b \( \Leftrightarrow \) b After a if \( a.end.position < b.begin.position \)
- a Meets b \( \Leftrightarrow \) a \( Meets^{-1} \) b \( \Leftrightarrow \) b MetBy a if \( a.end.position = b.begin.position \)
Figure 2.2: Temporal relationships between two periods

- a Overlaps b ⇔ a Overlaps⁻¹ b ⇔ b OverlappedBy a if \( a.begin.position < b.begin.position \) AND \( a.end.position > b.begin.position \) AND \( a.end.position < b.end.position \)

- a Begins b ⇔ a Begins⁻¹ b ⇔ b BegunBy a if \( a.begin.position = b.begin.position \) AND \( a.end.position < b.end.position \)

- a BegunBy b ⇔ a BegunBy⁻¹ b ⇔ b Begins a if \( a.begin.position = b.begin.position \) AND \( a.end.position > b.end.position \)

- a During b ⇔ a During⁻¹ b ⇔ b Contains a if \( a.begin.position > b.begin.position \) AND \( a.end.position < b.end.position \)

- a Contains b ⇔ a Contains⁻¹ b ⇔ b During a if \( a.begin.position < b.begin.position \) AND \( a.end.position > b.end.position \)

- a Equals b ⇔ b Equals a if \( a.begin.position = \text{other}.begin.position \) AND \( a.end.position = b.end.position \)

- a OverlappedBy b ⇔ a OverlappedBy⁻¹ b ⇔ b Overlaps a if \( a.begin.position > b.begin.position \) AND \( a.begin.position < b.end.position \) AND \( a.end.position > b.end.position \)

- a Ends b ⇔ a Ends⁻¹ b ⇔ b EndedBy a if \( a.begin.position > b.begin.position \) AND \( a.end.position = b.end.position \)
CHAPTER 2. FUNDAMENTALS OF TIME

- $a$ EndedBy $b$ ⇔ $a.\text{EndedBy}^{-1} \leftrightarrow b$ Ends $a$ if $a.\text{begin.position} < b.\text{begin.position}$ AND $a.\text{end.position} = b.\text{end.position}$

- $a$ MetBy $b$ ⇔ $a.\text{MetBy}^{-1} \leftrightarrow b$ Meets $a$ if $a.\text{begin.position} = b.\text{end.position}$

- $a$ After $b$ ⇔ $a.\text{After}^{-1} \leftrightarrow b$ Before $a$ if $a.\text{begin.position} > b.\text{end.position}$

Five relationships are defined between an instant and a period but are not explicitly described here. In the context of this thesis the relationships between two periods are considered as most relevant since the valid time or life span of features in the database will be described by a period.

There is a number of operations that can be applied to two periods, each operation resulting in one new period. These operations are $\text{Extend}$, $\text{Difference}$, $\text{Intersection}$ and $\text{Union}$. Figure 2.3 depicts the four operations and their results. The length of a period can be calculated by subtracting the first delimiting instant from the second one. This would result in an interval.

![Figure 2.3: Period operations (Snodgrass, 2000, p.96)](image)

2.4.2 Spatial Relationships

Spatial relationships between two geometric objects can be described and tested with the $\text{Dimensionally Extended Nine-Intersection Model}$ (DE-9IM). Within this model each geometric object $a$ is defined by its interior $I(a)$, boundary $B(a)$ and
exterior E(a). A function \( \text{dim}(x) \) is assumed that returns the maximum dimension (-1, 0, 1, or 2) of the geometric objects in \( x \), with a numeric value of -1 corresponding to \( \text{dim}(\emptyset) \). The intersection of any two of I(a), B(a) and E(a) can result in a set of geometric objects, \( x \), of mixed dimension. The results can be shown in a matrix (Figure 2.4), the *dimensionally extended nine-intersection matrix* (DE-9IM).

<table>
<thead>
<tr>
<th></th>
<th>Interior</th>
<th>Boundary</th>
<th>Exterior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interior</td>
<td>( \text{dim}(I(a)\cap I(b)) )</td>
<td>( \text{dim}(I(a)\cap B(b)) )</td>
<td>( \text{dim}(I(a)\cap E(b)) )</td>
</tr>
<tr>
<td>Boundary</td>
<td>( \text{dim}(B(a)\cap B(b)) )</td>
<td>( \text{dim}(B(a)\cap B(b)) )</td>
<td>( \text{dim}(B(a)\cap E(b)) )</td>
</tr>
<tr>
<td>Exterior</td>
<td>( \text{dim}(E(a)\cap B(b)) )</td>
<td>( \text{dim}(E(a)\cap B(b)) )</td>
<td>( \text{dim}(E(a)\cap E(b)) )</td>
</tr>
</tbody>
</table>

Figure 2.4: The DE-9IM (OGC, 2006a, p.35)

On two geometric objects, a spatial relationship predicate can be expressed as a formula that takes as input a pattern matrix representing the set of acceptable values for the DE-9IM for the two geometric objects. If the spatial relationship between the two geometric objects corresponds to one of the acceptable values as represented by the pattern matrix, then the predicate returns TRUE. For each cell of the matrix there are six possible values. These values and their meanings are as shown in Figure 2.5.

\[
\begin{align*}
p = T & \Rightarrow \text{dim}(x) \in \{0, 1, 2\}, \text{ i.e. } x \neq \emptyset \\
p = F & \Rightarrow \text{dim}(x) = -1, \text{ i.e. } x = \emptyset \\
p = * & \Rightarrow \text{dim}(x) \in \{-1, 0, 1, 2\}, \text{ i.e. Don't Care} \\
p = 0 & \Rightarrow \text{dim}(x) = 0 \\
p = 1 & \Rightarrow \text{dim}(x) = 1 \\
p = 2 & \Rightarrow \text{dim}(x) = 2
\end{align*}
\]

Figure 2.5: Cell values for the DE-9IM (OGC, 2006a, p.36)

The matrix can be represented as a list of nine characters in row major order. If \( a \) and \( b \) were two line geometries, a matrix of ‘1*******’ could be used to test if the interior of the two lines intersect; a *Relate* (\( b, \text{’1*******’} \)) shall return TRUE in this case.

**Named spatial relationships based on the DE-9IM** Using the matrix and the *Relate* predicate allows for a very fine tuned testing of spatial relationships. Nevertheless, the matrix is not the most natural way to express a spatial relationship between two geometric objects. Because of that, the most common rela-
tionships to be tested for have been named. The following list shows the named relationships and the related intersection matrix. With the aim to test the relationship between two periods (representable as lines) in particular, the matrix is provided for the relationships between two lines:

- a Overlaps b ⇔ a Relate (b, '1*T***T***)
- a Within b ⇔ a Relate (b, 'T*F**F***')
- a Contains b ⇔ b Within a
- a ContainsProperly b ⇔ a Relate (b, 'TTT******')
- a Disjoint b ⇔ a Relate (b, 'FF*FF****')
- a Intersects b ⇔ ! a Disjoint b
- a Equals b ⇔ a Relate (b, 'TFFFTFFFT')
- a CoveredBy b ⇔ a Relate (b, 'T*F**F***')
- a Covers b ⇔ b CoveredBy a
- a Crosses b ⇔ a Relate (b, '0********')
- a Touches b ⇔ [ a Relate (b, 'FT*******') \ a Relate (b, 'F**T*****') \ a Relate (b, 'F***T****') ]

The matrix is the same for the Within and CoveredBy relationships when testing two lines. However, it will be different when testing these relationships between e.g. a line and a polygon.

Similar to the constructive operations (Difference, Intersection and Union) for temporal objects, methods are defined for spatial objects. Applied to two lines, they will lead to the same result as shown in Figure 2.3. A method Length is defined for geometries of type curve (with a line just being a special type of curve).

### 2.4.3 Analogy of Temporal and Spatial Relationships

Because space and time have geometry and topology, it is assumed that there is some analogy also between the spatial and temporal relationships. However, there is one crucial difference in that the spatial relationships do not consider any kind of order or direction of the compared objects in space. Hence, when testing for
temporal relationships with spatial methods, some additional condition will be required. An attempt has been made to express the thirteen possible temporal relationships between two periods a and b by appropriate spatial methods. This implies that a period is represented as line geometry with its X minima defined as \( X_{\text{min}} \) and its X maxima defined as \( X_{\text{max}} \). The results are as follows:

- \( a \) Before \( b \) \iff \( X_{\text{max}}(a) < X_{\text{min}}(b) \)
- \( a \) Meets \( b \) \iff \( X_{\text{max}}(a) = X_{\text{min}}(b) \)
- \( a \) Overlaps \( b \) \iff \( a \) Overlaps \( b \) AND \( X_{\text{min}}(a) < X_{\text{min}}(b) \)
- \( a \) Begins \( b \) \iff \( X_{\text{min}}(a) = X_{\text{min}}(b) \) AND \( X_{\text{max}}(a) < X_{\text{max}}(b) \)
- \( a \) BegunBy \( b \) \iff \( X_{\text{min}}(a) = X_{\text{min}}(b) \) AND \( X_{\text{max}}(a) > X_{\text{max}}(b) \)
- \( a \) During \( b \) \iff \( b \) ContainsProperly \( a \)
- \( a \) Contains \( b \) \iff \( a \) ContainsProperly \( b \)
- \( a \) Equals \( b \) \iff \( a \) Equals \( b \)
- \( a \) OverlappedBy \( b \) \iff \( a \) Overlaps \( b \) AND \( X_{\text{min}}(a) > X_{\text{min}}(b) \)
- \( a \) Ends \( b \) \iff \( X_{\text{min}}(a) > X_{\text{min}}(b) \) AND \( X_{\text{max}}(a) = X_{\text{max}}(b) \)
- \( a \) EndedBy \( b \) \iff \( X_{\text{min}}(a) < X_{\text{min}}(b) \) AND \( X_{\text{max}}(a) = X_{\text{max}}(b) \)
- \( a \) MetBy \( b \) \iff \( X_{\text{min}}(a) = X_{\text{max}}(b) \)
- \( a \) After \( b \) \iff \( X_{\text{min}}(a) > X_{\text{max}}(b) \)

Only the temporal relationships \textit{During}, \textit{Contains} and \textit{Equals} can be expressed by a named spatial relationship alone. This is because the order of the two periods on the time axis is not relevant in these relationships. In all other relationships \( X_{\text{min}} \) and \( X_{\text{max}} \) are required to determine the order of two periods on the time axis.

Künzel (2008) suggests in his work to compare the distance of the two periods to the origin of the relating reference system, using the spatial method \textit{Distance}. The period with the smaller distance to the origin would occur \textit{before} the other one on the time axis. This approach implies that no period can occur before the origin of the reference system. Otherwise the approach will fail as the distance is calculated as absolute value.
If the order of the temporal objects on the time axis is relevant, the methods for testing spatial relationships might not be most appropriate as shown before. However, they might be used well in other cases. In Figure 2.2 the possible temporal relationships between two periods are shown. To identify all blue periods that share at least one granule with the red period (e.g. to find all features affected by a certain time span), nine temporal relationships have to be tested for (c)–(k)). Using the dimensionally extended nine-intersection matrix in this case would simplify the test. A matrix of '1********' would cover all nine cases.

### 2.5 Temporal Reference Systems

A temporal position is measured relative to a temporal reference system. Three types of temporal reference systems are defined in ISO (2002b), calendar and clocks, ordinal reference system and temporal coordinate system. The conceptual model for temporal reference systems is depicted in Figure 2.6.

![Temporal Reference Systems Diagram](image)

**Figure 2.6: Temporal reference systems (ISO, 2002b, p.17)**

#### 2.5.1 Calendars and Clocks

Both, calendars and clocks are based on an interval scale. A Calendar is a discrete reference system providing a granularity of one day. By including clocks a temporal resolution greater than a day can be achieved. A clock must be used together with a calendar to provide a complete description of a temporal position within a
particular day. Calendars are related to one or more calendar eras with each era defined by a reference date and event. Calendar years are numbered relative to this reference date.

2.5.2 Ordinal Reference System

An ordinal reference system is based on an ordinal scale. Only the relative position of temporal objects can be measured on such scale. Because of that an ordinal reference system is used in domains where relative position in time is known more precisely than duration e.g. in geology or archaeology. The order of events in time can be determined but the exact duration between two events cannot. An ordinal reference system consists of one or more ordinal eras; each defined by a name and, if known, by a begin date and end date.

2.5.3 Temporal Coordinate System

A temporal coordinate system is based on an interval scale and thus allows describing the order of temporal objects, as well as measuring duration (distance) between them. A temporal coordinate system is defined by date and time of the scale’s origin and an interval, which is the smallest unit of measure in this coordinate system. The interval determines the temporal granularity, and hence, the precision with which time can be measured. The origin shall be specified in the Gregorian calendar with time of day in Coordinated Universal Time (UTC). The temporal coordinate system is the most relevant of the three types of temporal reference systems in the context of this thesis since it is the key to representing time as geometry.

By expressing the distance between a specific instant in time and the origin of the coordinate system as a multiple of the defined interval for that system, a temporal coordinate is created. In regard to this concept ISO (2002b) defines two supportive methods that are shown in the conceptual model for the temporal coordinate system (Figure 2.7) and are specified as follows:

- \texttt{transformDateTime(dateTime: DateTime): TM\_Coordinate} accepts a DateTime value and transforms it into a temporal coordinate.

- \texttt{transformCoord(c\_value: TM\_Coordinate): DateTime} accepts a temporal coordinate and transforms it into a DateTime value.
CHAPTER 2. FUNDAMENTALS OF TIME

Figure 2.7: Temporal coordinate system (ISO, 2002b, p.19)

TM.Coordinate is a data type with two attributes, indeterminatePosition and coordinateValue (Figure 2.8). The latter attribute is of type Number and holds the actual distance from the scale origin. Given a scale origin of date '2000-01-01' and an interval of 'day', the coordinateValue of an instant with date '2000-01-08' would be 7.

Figure 2.8: Data type TM.Coordinate (ISO, 2002b, p.22)

2.6 Other Aspects

2.6.1 Branching of Time

There is the concept of branching time, according to which more than one time line can lead from present to past or future (Figure 2.9).

This concept would allow to represent multiple realities over time and to create various what-if scenarios. Support for branching of time in a database is rather difficult to implement since having multiple versions of one feature at the same time would violate basic concepts of keys and uniqueness.
2.6.2 Feature Succession

Feature succession is a type of temporal feature association, describing the fact where one set of features is replaced by another set of features. The life span of features in the first set ends at the instant when the life span of features in the second set begins. There is a temporal and spatial aspect in feature succession as the involved features occupy the same location at different times and in a particular order (ISO, 2002b). There are three kinds of feature associations, division, substitution and fusion. Splitting a cadastral parcel is an example of feature division, merging two parcels into one is a fusion.

Succession associations among features can be derived using their temporal and spatial relationships.
Chapter 3

Time in Databases

3.1 Support for Temporal Data Types

In Section 2.3 three kinds of temporal data types were described, Instant, Interval and Period. This section briefly examines their implementation in a database management system.

3.1.1 Instants

According to the SQL-92 standard, these data types should be implemented in a DBMS to represent an instant:

Date to store the year, month and day of an instant. The granule is a day.

Time to store the time of the day as hours, minutes and seconds. Most DBMS also support fractions of a second. The Time type can be with or without time zone.

TimeStamp to represent year, month and day together with the time of the day as hours, minutes and seconds. As with the Time type, optional support for the fractions of a second is possible. The TimeStamp can be with or without time zone.

Most existing database management systems support all of these types, at least if they claim to conform with the SQL-92 standard.
3.1.2 Intervals

The \texttt{INTERVAL} type is provided to support the handling of intervals. It can hold and be constructed from years, months, days, hours, minutes and seconds, and, optionally, fractions of a second. To restrict the set of fields used by the DBMS internally to store an interval, options as \texttt{YEAR TO MONTH}, \texttt{DAY TO HOUR} can be provided by the user when an interval is constructed.

3.1.3 Periods

Because periods are not provided in the SQL-92 standard, a particular data type is not implemented in most DBMS. However, a custom type to hold a period could be easily created, using the temporal data types supported by the DBMS. Listing 3.1 shows one way to create a \textit{Period} type.

\begin{verbatim}
CREATE TYPE period AS ( 
    validFrom timestamp with time zone, 
    validTill timestamp with time zone 
) ;
\end{verbatim}

Listing 3.1: Creating a \textit{Period} data type

3.2 Kinds of Time

When maintaining temporal information in a database, three types of time can be differentiated, valid time, transaction time and user defined time. Only support for valid time and/or transaction time makes a database \textit{temporal}.

3.2.1 Transaction Time

Transaction time is the time when a fact becomes current in the database and can be retrieved (Jensen et al., 1994). If a table has transaction time support, it captures the history of its changing state and hence can be reconstructed as of a previous date. That also implies that from such \textit{transaction-time state table} rows can never be deleted physically, only logically (Snodgrass, 2000).
3.2.2 Valid Time

Valid time is the time when a fact became true in reality (Jensen et al., 1994). Because of that it is often called event time or world time, too. Supporting valid time in the database allows recording the history of the modelled reality. A table with valid-time support records states, that is facts, that are true over a period of time (Snodgrass, 2000).

Valid time can never be later than transaction time if only events are considered that really happened. The situation is different when future events are stored as well, e.g. to simulate a what-if scenario. In a real time system valid time and transaction time would be identical. However, in practice events are usually stored some time after they occurred. The interrelation between valid time and transaction time is shown in Figure 3.1.

![Interrelation between valid time and transaction time](image)

Figure 3.1: Interrelation between valid time and transaction time (Ott and Świaczny, 2001, p.70)

3.2.3 User-Defined Time

If a table has a column of a temporal data type and this column does not indicate when other columns were valid or when a fact was stored or modified in the
database, this column is a user-defined time column. It is a column as any other column, but happens to be of a temporal data type. A column recording the birthday of a person would be an example for user-defined time.

### 3.2.4 Classification of Temporal Databases

According to the support available for valid time and/or transaction time, a database can be classified as shown in Table 3.1.

<table>
<thead>
<tr>
<th>Support for Valid Time</th>
<th>Support for Transaction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>No support</td>
<td>No support</td>
</tr>
<tr>
<td>Support</td>
<td>Support</td>
</tr>
<tr>
<td>Static</td>
<td>Rollback</td>
</tr>
<tr>
<td>Historic</td>
<td>Bi-temporal</td>
</tr>
</tbody>
</table>

Table 3.1: Classification of temporal databases (Worboys, 1994, p.28)

A static database has no temporal support at all. It stores only the information currently valid. Old data will simply be overwritten by new data. A bi-temporal database, storing valid time and transaction time, is most powerful. It allows to present the modelled reality for any given point in time as well as to find out when a fact became known to the database. This is very useful in the cases where facts are not recorded in the order in which they occur in reality.

### 3.3 Version Management

Version management in the context of this work describes how to keep a record of the various versions (historical, current and future ones) that might exist for a feature. A version is caused by an event happening to a real world phenomenon, represented by a feature in the database. In most cases it is historical versioning that is applied, meaning that versions are kept from the past up to now. Depending on the particular implementation, it might also be permitted to record versions with a valid time beginning in the future, e.g. to simulate a what-if scenario or to store a fact ahead of time that is known to happen.

Versioning discussed in the context of this work shall not be confused with concurrent versioning. Concurrent versioning is a technique applied to permit multiple users/clients to work with the same data at the same time and over a
longer period of time without conflicting with each other. Without concurrent versioning a dataset would have to be locked explicitly, permitting exactly one client to apply changes, with read access only for any other client. Concurrent versioning is basically implemented by creating a separate copy (a “version”) of the original data for each client. Once all changes are applied to a version, the version will be merged with the original data, taking into account certain rules to solve conflicts that might occur in that moment.

Version management as looked at in this document can occur on four levels: database, relation, tuple and attribute. With each level, data redundancy decreases.

Database level  This is probably the most archaic type of versioning. A snapshot of the entire database is created each time a change occurs. The data redundancy is inherently enormous. This kind of versioning might only be reasonable in a field of application where changes rarely occur (e.g. in geology) and if they occur, preferably multiple ones at the same time. Database-level versioning applied to a domain with frequent changes might become very inconvenient since creating a snapshot of the entire database will take time and affect the performance of the database if it is running during that time. The situation becomes even worse if the database needs to be shut down every time a snapshot is created.

Relation level  With this type of versioning a snapshot is created of the tables only in which a change occurred. Although data redundancy would be less than with database-level versioning, it is still considerable. Again, relation-level versioning might be suitable if many changes occur to the same table at the same time.

Tuple level  Only a snapshot of the row that experienced a change is created. To record the valid time, two timestamp columns for begin and end of the period are added to the table. Tuple-level versioning is probably the most common type implemented in systems, because it represents a good compromise between applicability and data redundancy. The topics discussed in the subsequent sections of this chapter (temporal keys and time-oriented statements) all relate to tuple-level versioning.
Attribute level Only the attributes that changed are included in the snapshot. Although this level of versioning is perfect in regard to data redundancy (there is no redundancy), it requires a very particular database schema. Because of that, attribute-level versioning is not found very often in existing systems.

3.4 Temporal Keys

3.4.1 Candidate Keys and Functional Dependency

Most of the existing database management systems are based on the Relational Model formulated and proposed by E. F. Codd in 1969. Given a number of value domains $D_1, D_2, \ldots, D_n$ that must contain atomic values only, a relation $R$ is, according to this model, defined as a subset of the Cartesian product of the $n$ domains (Kemper and Eickler, 2001):

$$R \subseteq D_1 \times \cdots \times D_n$$

A relation is implemented as a table with the table columns/attributes representing the domains. Between the sets of attributes in a relation a functional dependency can exist. With $\alpha$ representing one set of attributes and $\beta$ representing another set of attributes in $R$, $\alpha$ functionally determines $\beta$ ($\alpha \rightarrow \beta$) if, and only if one value of $\beta$ is associated with precisely one value of $\alpha$. Thus, $\beta$ is functionally dependent on $\alpha$. A candidate key is a minimal set of attributes that functionally determine all of the attributes in a relation. An example is given in Table 3.2. From all the attributes only $personNr$ would be a candidate key, since it is the only attribute that functionally determines all other attributes, \( \{personNr\} \rightarrow \{surname, name, birthday\} \).

<table>
<thead>
<tr>
<th>personNr</th>
<th>surname</th>
<th>name</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Wagner</td>
<td>Michael</td>
<td>1974-11-21</td>
</tr>
<tr>
<td>78</td>
<td>Wagner</td>
<td>Alexander</td>
<td>1979-01-04</td>
</tr>
<tr>
<td>212</td>
<td>Lottes</td>
<td>Horst</td>
<td>1931-02-13</td>
</tr>
<tr>
<td>439</td>
<td>Schmidt</td>
<td>Michael</td>
<td>1931-02-13</td>
</tr>
</tbody>
</table>

Table 3.2: personNr as candidate key

To ensure that a candidate key is not violated by a table entry, the SQL PRIMARY KEY construct can be used. It checks that the entries in one or more columns forming the candidate key are unique and not empty (“null”).
CHAPTER 3. TIME IN DATABASES

3.4.2 Sequenced Primary Key

Unfortunately the PRIMARY KEY construct is not adequate for a temporal database, in particular one with valid-time support. Table 3.3 shows a valid-time state table containing information about land use plots. Besides the id and type of land use, the table holds the id of the parcel this plot is related to, and the plot’s valid time (validFrom, validTill). The valid time is represented by a closed-open period.

<table>
<thead>
<tr>
<th>landuseId</th>
<th>landuseType</th>
<th>parcel</th>
<th>validFrom</th>
<th>validTill</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>Forest</td>
<td>200/2</td>
<td>1995-01-01</td>
<td>2002-08-01</td>
</tr>
<tr>
<td>112</td>
<td>Agriculture</td>
<td>200/2</td>
<td>2002-04-01</td>
<td>2004-02-01</td>
</tr>
</tbody>
</table>

Table 3.3: Extract of table landuse

This table basically states that land use plot 112 exists twice at the same time (for four months) which in reality cannot occur. Although the valid time must be somehow taken into account in the table’s primary key, none of the following primary key constraints would prevent the shown deficiency:

```
ALTER TABLE landuse ADD PRIMARY KEY (landuseId, validFrom);
ALTER TABLE landuse ADD PRIMARY KEY (landuseId, validTill);
ALTER TABLE landuse ADD PRIMARY KEY (landuseId, validFrom, validTill);
```

Listing 3.2: Attempting to create a temporal key

What is required is a sequenced primary key. Such key is implemented through a sequenced constraint. A sequenced constraint is one that is applied independently at each point in time (Snodgrass, 2000). A sequenced constraint for table landuse could be specified in SQL as follows:

```
CREATE ASSERTION seq_primary_key
CHECK (NOT EXISTS ( SELECT * FROM landuse AS lu1 WHERE 1 < (SELECT COUNT(landuseId) FROM landuse AS lu2 WHERE lu1.landuseId = lu2.landuseId AND lu1.validFrom < lu2.validTill AND lu2.validFrom < lu1.validTill))
AND NOT EXISTS ( SELECT * FROM landuse AS lu WHERE lu.landuseId IS NULL));
```
Listing 3.3: Expressing a sequenced primary key (adapted from Snodgrass (2000, p.118))

This constraint actually checks if the valid time periods of any land use plots with the same landuseId overlap.

### 3.4.3 Duplicates

Snodgrass (2000) identified four kinds of duplicates that can occur in a table with valid time support:

**Value equivalence** The first kind is value equivalence which occurs if the non-timestamp columns of two or more rows are identical. Value equivalence can be prevented by using the SQL UNIQUE constraint on the non-timestamp columns. For the table `landuse` the constraint looks like this:

```sql
CREATE TABLE landuse (  
  ...  
  UNIQUE(landuseId, landuseType, parcel)  
) ;
```

Listing 3.4: Preventing value equivalence

There are also cases that might require to allow for value equivalence. E.g. the same land use plot could be of type Forest for several years, later on of type Agriculture and at some point again of type Forest.

**Sequenced duplicates** Two rows are sequenced duplicates if they are duplicates at some instant. Sequenced duplicates are prevented with a sequenced constraint. For the table `landuse` the constraint in Listing 3.3 can be applied without any modification.

**Current duplicates** Two rows are current duplicates if they are sequenced duplicates at the current instant. Current duplicates are prevented implicitly by preventing sequenced duplicates.
**Non-sequenced duplicates** Two rows are non-sequenced duplicates if the values of all columns are identical. Non-sequenced duplicates are prevented implicitly by preventing value equivalence and/or sequenced duplicates.

### 3.4.4 Referential Integrity

If the candidate key of one relation is used as an attribute in another relation, it is called a *Foreign Key*. Let $R$ and $S$ be two relations and $\kappa$ the primary key of $R$. In this case is $\alpha \subseteq S$ a foreign key if for all tuples $s \in S$ applies (Kemper and Eickler, 2001):

- $s.\alpha$ contains either no null values or only values different from null.
- If $s.\alpha$ contains values different from null, a tuple $r \in R$ must exist with $s.\alpha = r.\kappa$.

Referential integrity is established if these conditions are met. How referential integrity is expressed and ensured depends on whether the referencing and referenced tables are temporal tables. Four cases were identified by Snodgrass (2000):

**Neither table is temporal** In such case standard SQL constructs are adequate. If the *landuse* table had no valid time support and shall reference a table *parcel* via an attribute named *parcel*, an SQL statement as follows could be used to declare this attribute as foreign key:

```sql
CREATE TABLE landuse (  
  ...  
  parcel varchar(10) REFERENCES parcel ,  
  ...  
);  
```

*Listing 3.5: Creating a foreign key*

**Only the referencing table is temporal** If only the table *landuse* were temporal, the same SQL statement as above applies.
Both tables are temporal. A sequenced foreign key is required in the case that both tables are temporal, and is expressed as an assertion or SQL constraint. A sequenced constraint always covers a current constraint implicitly (as a current constraint is just a special case of a sequenced constraint). The key is a sequenced foreign key if, for all rows $r$ in the referencing table,

- there is a row with that key value valid in the referenced table when $r$ started,
- there is a row with that key value valid in the referenced table when $r$ stopped,
- and there are no gaps when there are no rows in the referenced table, during $r$’s period of validity, that have that key value (Snodgrass, 2000).

Assuming a relation `parcel` with a schema as shown in Table 3.4, a sequenced foreign key for table `landuse` could be expressed with the SQL in Listing 3.6.

<table>
<thead>
<tr>
<th>parcelId</th>
<th>legalSize</th>
<th>description</th>
<th>validFrom</th>
<th>validTill</th>
</tr>
</thead>
<tbody>
<tr>
<td>200/2</td>
<td>560.40</td>
<td>XXX</td>
<td>1994-01-01</td>
<td>2002-08-01</td>
</tr>
<tr>
<td>312</td>
<td>1735.80</td>
<td>YYY</td>
<td>2006-04-01</td>
<td>2009-02-01</td>
</tr>
</tbody>
</table>

Table 3.4: Extract of table `parcel`

```
CREATE ASSERTION seq_foreign_key
CHECK ( NOT EXISTS ( 
  SELECT * FROM landuse AS Iu
  WHERE NOT EXISTS ( 
    SELECT * FROM parcel AS p
    WHERE Iu.parcel = p.parcelId
    AND p.validFrom <= Iu.validFrom
    AND Iu.validFrom < p.validTill
  ) OR NOT EXISTS ( 
    SELECT * FROM parcel AS p
    WHERE Iu.parcel = p.parcelId
    AND p.validFrom < Iu.validTill
    AND Iu.validTill <= p.validTill
  ) OR EXISTS ( 
    SELECT * FROM parcel AS p
    WHERE Iu.parcel = p.parcelId
  )
)
```
\textbf{CHAPTER 3. TIME IN DATABASES}

```
AND lu.validFrom < p.validTill
AND p.validTill < lu.validTill
AND NOT EXISTS (SELECT * FROM parcel AS p2
WHERE p2.parcelId = p.parcelId
AND p2.validFrom <= p.validTill
AND p.validTill < p2.validTill)));
```

Listing 3.6: Creating a sequenced foreign key (adapted from Snodgrass (2000, p.128))

In Chapter 6 it will be shown how this rather big statement can be simplified by representing the valid-time period of parcels and land use plots as geometry and using spatial functions to express the sequenced foreign key.

**Only the referenced table is temporal** If \textit{parcel} were a temporal table and \textit{landuse} were not, the foreign key could be expressed as follows:

```
CREATE ASSERTION current_foreign_key
CHECK (NOT EXISTS (SELECT *
FROM landuse AS lu
WHERE NOT EXISTS (SELECT *
FROM parcel AS p
WHERE lu.parcel = p.parcelId
AND p.validTill = DATE '9999-12-31')));
```

Listing 3.7: Creating a current foreign key (adapted from Snodgrass (2000, p.130))

This construct implies three facts:

1. Table \textit{parcel} contains no rows that start in the future.

2. A value of “9999-12-31” in \textit{validTill} of table \textit{parcel} indicates the rows that are currently valid.

3. The non-temporal table \textit{landuse} records current data only.
CHAPTER 3. TIME IN DATABASES

3.5 Time-Oriented Statements

The statements discussed in this section include query statements (SELECT) as well as statements to modify data (INSERT, UPDATE, DELETE). SELECT, INSERT, UPDATE and DELETE are the most relevant statements of the Data Manipulation Language (DML). In relation to valid-time state tables these statements can be of kind current (now), sequenced (at each instant of time) or non-sequenced (ignoring time) (Snodgrass, 2000). The SELECT statement will be looked at very briefly only since it is not as relevant in the context of this thesis and the history extension as the statements to modify data. In regard to the other statements their sequenced version will be examined in particular. The sequenced version is the most useful of the time-oriented statements, the most complex one to express, and covers the current version of the statements implicitly.

With the temporal statements described in this section, it is assumed that all information (historical, current and possible future one) is kept in one table. There is also a concept called temporal partitioning according to which the current information is stored in one table and historical one in another table. Current queries and insertions would be quite simple but future data cannot be stored applying this concept. Temporal partitioning is not further discussed here.

3.5.1 SELECT

To find out the land use plots that are true now, a query as follows could be used:

```sql
SELECT landuseId, landuseType, parcel
FROM landuse WHERE validTill = DATE '9999-12-31';
```

Listing 3.8: Extracting the current state

Again, this query implies that no future data is stored in landuse and that the date “9999-12-31” in validTill indicates rows currently valid. To find out the land use plots at any point in time, the query in Listing 3.9 can be used. Here data valid on March 1, 2009 shall be retrieved.

```sql
SELECT landuseId, landuseType, parcel
FROM landuse WHERE validFrom <= DATE '2009-03-01' AND DATE '2009-03-01' < validTill;
```

Listing 3.9: Extracting the state at any time
A similar construct could be used to retrieve the rows valid during a certain period of time.

A non-sequenced query would simply ignore the temporal nature of a table, e.g.:

```
SELECT * FROM landuse;
```

Listing 3.10: Non-sequenced query

### 3.5.2 INSERT

In a sequenced insertion the period of applicability is provided by the client, e.g.:

```
INSERT INTO landuse VALUES ('112', 'Forest', '200/2', '2006-01-03', '2008-05-12');
```

Listing 3.11: Sequenced insertion

Care must be taken to ensure the temporal key and referential integrity during a sequenced insertion. Thus, as described by Snodgrass (2000), a row is only inserted:

- if no duplicate exists during the period of applicability,
- and if there is a row in the referenced table at the start of the period of applicability,
- and if there is a row at the end of the period of applicability,
- and if there are no gaps during the period of applicability.

The first issue is covered by a sequenced primary key as shown in Listing 3.3. The last three points (referential integrity) are covered by a sequenced foreign key as in Listing 3.6.

### 3.5.3 UPDATE

As defined by Snodgrass (2000), “A sequenced update is the temporal analog of a nontemporal update, with a specified period of applicability...”. Implementing a sequenced update requires five statements, two INSERTs and three UPDATEs. These statements reflect the four ways the valid-time period of a row can intersect with the period of applicability. The meaning of the term intersect in this context is that the two periods share at least one granule of time.
This is what each of the five statements does (Snodgrass, 2000):

- Insert the old values from the start date to the beginning of the period of applicability.
- Insert the old values from the end of the period of applicability to the end date.
- Update the explicit columns of rows that overlap the period of applicability.
- Update the start date to begin at the beginning of the period of applicability of rows that overlap the period of applicability.
- Update the end date to end at the end of the period of applicability of rows that overlap the period of applicability.

The statements must be executed in exactly the order as listed. Figure 3.2 shows the result of each single statement. The period of applicability is shown in red color. The four cases of how the valid-time period initially intersects with the
period of applicability are shown in blue, green, yellow and grey color. The result of the first UPDATE statement remains the same in regard to the valid time since this UPDATE modifies only the non-timestamp columns of the table. Because of that the result of the first UPDATE is not explicitly shown, in fact it is still the result of the second INSERT.

Care must be taken if the updated table references another table. In this case the foreign key constraint in Listing 3.6 applies. An update of the referenced table does not affect the referencing table if its primary key is not updatable. Otherwise Listing 3.6 applies again.

A non-sequenced update treats the valid time columns as any other column and can modify them in an arbitrary way. Because of that non-sequenced updates (as well as deletions) should be avoided if sequenced constraints are present.

### 3.5.4 DELETE

A sequenced deletion is implemented by four statements, an INSERT, two UPDATES and a DELETE (Snodgrass, 2000):

- Insert the old values from the end of the period of applicability to the end of the period of validity of the original row.
- Update the end date to end at the beginning of the period of applicability.
- Update the start date to begin at the end of the period of applicability.
- Delete entirely rows that are covered by the period of applicability.

As with a sequenced update, the statements are required to take into account the four cases of how a row’s valid-time period might intersect the period of applicability. The order of the statements matters. Figure 3.3 depicts the result of each of the four statements.

A sequenced deletion on a referenced table could cause a gap, violating a foreign key in the referencing table. To prevent this, a sequenced foreign key constraint (Listing 3.6) must be applied. Both, sequenced updates and sequenced deletions might temporary violate the sequenced primary key because rows will overlap in their valid time during the execution of the five and four statements, respectively. This is caused by the INSERT statements (required for the “blue” rows) and can be seen in Figures 3.2 and 3.3. To prevent the update or deletion from being cancelled,
the primary key constraint must be declared as “DEFERRABLE INITIALLY DEFERRED”. In this case the constraint will not be applied after each statement, but only at the end of the transaction within which the statements are executed.
Chapter 4

Implementation Samples

In this chapter two examples for the implementation of temporal support in databases are discussed briefly.

4.1 Oracle Workspace Manager 11g

The Oracle Workspace Manager is a feature of the Oracle database management system. It is a PL / SQL (PL: Procedural Language) package that version-enables user tables (Oracle, 2007). The package consists of a number of database procedures, data types and operators that support not only versioning but also valid time and transaction time. In that sense an Oracle database used with the Workspace Manager is a real bi-temporal database. Versioning in the context of Workspace Manager is the capability to maintain multiple versions of the same data simultaneously by isolating them in so called workspaces. Workspaces are virtual only, the data itself still is kept in the version-enabled table. One of the main benefits of Workspace Manager is that users can work independently in their workspaces, without conflicting or affecting the production version of the data or data in other workspaces. Besides that, multiple what-if scenarios could be created, each scenario in its own workspace. Hence, branching of time is possible since various workspaces can hold scenarios that cover the same period of time.

When a table is version-enabled, the table is renamed and a view created on that table, given the original table name. INSTEAD OF triggers are created on the view that ensure that all INSERTs, UPDATEs and DELETEs executed on the view are applied to the underlying table. Metadata columns are added to the table to allow multiple version of a row with the same user-defined primary key to
exist in the database. Existing referential integrity is ensured. Thus, to a client application the database schema still looks the same, no changes are required at the client’s side. Besides the view just described, a number of supportive tables and views are created by Workspace Manager. This number varies, depending on whether support for valid time and/or transaction time is requested when a table is version-enabled. The unit of versioning is a row/tuple. This relates in a way to Section 3.3 where tuple-level versioning was described as a concept that is well applicable while data redundancy is negligible.

Listing 4.1 shows how a table is version-enabled in Oracle Workspace Manager (Oracle, 2008).

```sql
EXECUTE DBMS_WM.ENABLEVERSIONING ('landuse', 'VIEW_WO_OVERWRITE', FALSE, TRUE);
```

Listing 4.1: Version-enabling a table

The first argument in the `EnableVersioning(...)` procedure is the name of the table to be version-enabled. The second one enables support for transaction time in this particular sample, while the last argument enables support for valid time. If valid time support is enabled for a table, the user sets the valid time in his session context before executing any query or modification statement. To do this, Workspace Manager provides a procedure shown in Listing 4.2. The valid time (in fact the period of applicability) set this way, acts as filter on all subsequent queries or modifications. In regard to valid time, workspace manager supports sequenced queries and modifications, i.e. at each instant in time covering the past, now and the future.

```sql
EXECUTE DBMS_WM.SETVALIDTIME(TO_DATE( '01-01-1900' , 'MM-DD-YYYY') , TO_DATE( '01-01-9999' , 'MM-DD-YYYY')) ;
```

Listing 4.2: Setting the period of applicability

For the actual workspace management numerous procedures are available to create, goto, remove, merge, compress or rollback workspaces. Workspace management is beyond the scope of this thesis and not further discussed. Overall, Oracle Workspace Manager is with its support for concurrent versioning, transaction time and valid time, one of the most sophisticated implementations of temporal support in databases.
4.2 PostgreSQL Time Travel

*Time Travel* is a small extension to PostgreSQL enabling support for transaction time in user tables. To enable transaction time support, two columns of type *abstime* (timestamp-like) must be added to the user table to hold the *create* and *retire* time of the rows. Furthermore a trigger must be created that fires before every INSERT, UPDATE and DELETE of a row from this table. This trigger will execute a function timetravel() that is provided with the extension and that will take care of a behaviour as follows:

- If a row is inserted, its *create* timestamp is set to the current instant, the *retire* timestamp is set to 'infinity'.

- If a row is updated, the *retire* timestamp of the row containing the old data is set to the current instant and a new row is inserted, holding the new data. This row’s *create* timestamp is set to the current instant and its *retire* timestamp set to 'infinity'.

- Deleting a row will only cause the *retire* timestamp of the row set to the current instant. The row will not be deleted from the table physically.

- Once the *retire* timestamp of a row has been set to a date and time different from 'infinity', the row cannot be modified anymore.

Looking at this concept it is obvious that *tuple-level* versioning is applied again.
Chapter 5

PostgreSQL and PostGIS

This chapter provides a brief introduction to the Open Source database management system PostgreSQL and its spatial extension PostGIS. The focus is on those capabilities and functionality that will be used to implement the history extension.

5.1 PostgreSQL 8.4.1

PostgreSQL can, without any doubt, be described as the most advanced Open Source DBMS currently available. PostgreSQL is a object-relational DBMS and supports crucial database features, among them:

- Complex queries
- Referential integrity
- Views
- Triggers
- Transactions
- Multi-version concurrency control
- Cursors
- Sequences
- Various authentication schemas

Furthermore it can be extended by a user in many ways, e.g. by adding new:
CHAPTER 5. POSTGRESQL AND POSTGIS

• Data types
• Functions
• Operators
• Index methods
• Procedural languages

The current official manual contains 2119 pages (The PostgreSQL Global Development Group, 2009), the number giving a hint of the comprehensiveness of PostgreSQL.

5.1.1 Support for Temporal Data Types

PostgreSQL supports instants with three data types:

• timestamp with time zone (and without)
• time with time zone (and without)
• date

To represent intervals a data type interval is provided. No special support is available for periods. However, a custom data type to hold a period can easily be created as shown in Listing 3.1. A number of constructors, functions and operators are available for each of the data types. In PostgreSQL a special value infinity is defined, representing a timestamp later than all other timestamps. This value is very useful and can be applied to declare a row as currently valid by setting its valid-time end to infinity. Hence, a special date as ’9999-12-31’ is not required anymore. PostgreSQL uses the Julian calendar for all date/time calculations but with the assumption that a calendar year has a length of 365.2425 days, as in the Gregorian calendar. Thus, the precise Gregorian year (as very close to the solar calendar) is used together with the large range of the Julian calendar, reaching from 4713 BC to far in the future.
5.1.2 Views

PostgreSQL supports views as part of its Data Definition Language (DDL). A view is a kind of virtual table allowing different “views” on data from one or more physical tables. Besides viewing data from the physical tables, data could even be modified if views are used in combination with rules. The SQL statement in Listing 5.1 will create a view that will show only current land use plots. It is assumed again that date '9999-12-31' indicates the rows currently valid.

```
CREATE VIEW current_landuse_plots AS
SELECT landuseId, landuseType, parcel FROM landuse
WHERE validTill = DATE '9999-12-31';
```

Listing 5.1: Creating a view

5.1.3 Triggers

A trigger is a specification that will cause a particular function to be executed if a certain operation is performed on a table. Triggers are supported in PostgreSQL and can be defined to execute before or after any INSERT, UPDATE or DELETE operation. Whether triggers are fired before or after an operation, affects the visibility of the changes caused by the operation. Triggers can either fire once per each modified row or once per SQL statement.

Besides “before” and “after” as the moment when a trigger fires, the SQL standard defines also INSTEAD OF triggers. Such triggers are used to perform an operation different from the original operation (“instead of”) that caused the trigger to fire. INSTEAD OF triggers are not (yet) supported in PostgreSQL but rules can be used to simulate this functionality. In PostgreSQL, triggers have to be used instead of assertions and complex table constraints since those SQL constructs are not (yet) supported.

Listing 5.2 creates a trigger that is fired after each deletion of a row from table landuse. When fired the trigger will execute a function named xxx().

```
CREATE TRIGGER test_trigger
AFTER DELETE ON landuse
FOR EACH ROW EXECUTE PROCEDURE xxx();
```

Listing 5.2: Creating a trigger
5.1.4 Rules

Rules permit defining an additional or alternative action to be performed when a SELECT, INSERT, UPDATE or DELETE operation is applied to a table or view. This way a rule could be used to simulate an INSTEAD OF trigger. It can also be used to create an updatable view by replacing the INSERT, UPDATE or DELETE operation applied to the view with an appropriate operation on the relating table.

The rule in Listing 5.3 will prevent rows from being deleted from table landuse and show a message instead.

```
CREATE RULE prevent_deletion AS 
    ON DELETE TO landuse DO INSTEAD 
    ( 
        SELECT showMessage( 'Deletion not permitted!' ) 
    );
```

Listing 5.3: Creating a rule

5.1.5 PL/pgSQL

PL/pgSQL is a loadable procedural language for PostgreSQL ("loadable" into the database server). It can be used to:

1. create functions and trigger procedures,
2. add control structures to the SQL language,
3. and perform complex computations.

The functions and procedures written in PL/pgSQL are stored inside the database server ("stored procedures"). Once stored, they can be used as any built-in function. In the rule in Listing 5.3 it is shown how such function (showMessage(text)) is used.

5.2 PostGIS 1.4.0

PostGIS is the spatial extension to PostgreSQL, providing support for the storage of geometry and advanced GIS analysis in the database (Refractions Research, 2009). It complies with the Simple Feature Access Specification (SQL option) of
the OGC (OGC, 2006b). PostGIS provides a special data type *geometry* and a number of functions and operators for:

- creating geometry,
- accessing geometry,
- editing geometry,
- testing spatial relationships,
- processing geometry
- and linear referencing.

### 5.2.1 Creating Geometry

Geometry can be created from its representation as Well Know Text (WKT) which is a human readable format to describe geometry. Furthermore geometry can be constructed from the Well Known Binary (WKB) format. The statement in Listing 5.4 creates a line geometry from WKT, consisting of two vertexes.

```sql
SELECT ST_GeomFromText( 'LINESTRING(5,10 ,15,20) ' , -1) ;
```

Listing 5.4: Creating a line from WKT

### 5.2.2 Accessing Geometry

Various functions are available to access the components of a geometry, among them:

- **ST_StartPoint(geometry):Point** that returns the first point of a LINESTRING geometry as POINT geometry
- **ST_EndPoint(geometry):Point** that returns the last point of a LINESTRING geometry as POINT geometry
- **ST_XMin(geometry):Double** that returns the X minima of a bounding box or a geometry
- **ST_XMax(geometry):Double** that returns the X maxima of a bounding box or a geometry

The two latter functions are PostGIS specific and not defined in the Simple Feature Access Specification of the OGC.
5.2.3 Testing Spatial Relationships

Spatial operators The spatial operators in PostGIS test the spatial relationship of the bounding boxes of two geometries. All bounding box operators will make use of a spatial index if available for the geometries. Using the bounding box operator together with the actual spatial relationship function (e.g. `ST_Relate(...))` can speed up the query. In the example in Listing 5.5 the “&&” operator is used that will return ‘TRUE’ if the bounding boxes of the two lines intersect.

```sql
SELECT ST_GeomFromText( 'LINESTRING(5,10,15,20)' , -1) && ST_GeomFromText( 'LINESTRING(25,3,11,7)' , -1);
```

Listing 5.5: Using the “&&” operator

Spatial relationship functions PostGIS supports all the functions to test named spatial relationships as defined in OGC (2006a). These are:

- `ST_Contains(geometry a, geometry b):boolean`
- `ST_Within(geometry a, geometry b):boolean`
- `ST_Intersects(geometry a, geometry b):boolean`
- `ST_Equals(geometry a, geometry b):boolean`
- `ST_Disjoint(geometry a, geometry b):boolean`
- `ST_Crosses(geometry a, geometry b):boolean`
- `ST_Overlaps(geometry a, geometry b):boolean`
- `ST_Touches(geometry a, geometry b):boolean`
- `ST_Relate(geometry a, geometry b, text intersectionPatternMatrix):boolean`

The functions return `TRUE` if the tested relationship exists, otherwise they return `FALSE`.

Additionally PostGIS provides the following functions:

- `ST_Covers(geometry a, geometry b):boolean`
- `ST_CoveredBy(geometry a, geometry b):boolean`
• ST_ContainsProperly(\text{geometry a, geometry b}): boolean

• ST_Relate(\text{geometry a, geometry b}): text

The last function will return the intersection pattern matrix that describes the spatial relationship between the two provided geometries. The relating intersection matrices for all named spatial relationships can be found in Section 2.4.2.

5.2.4 Geometry Processing

In regard to geometry processing, PostGIS supports all functions as defined in OGC (2006a), among them:

• ST_Difference(\text{geometry a, geometry b}): geometry

• ST_Union(\text{geometry a, geometry b}): geometry

• ST_Intersection(\text{geometry a, geometry b}): geometry
Part II

Implementation of a Prototype
Chapter 6

Concept

Temporal support in databases is required in many fields of application. In most cases it is support for valid time that is needed since the moment when a change occurred in reality is more relevant than the moment when an information was entered in the system. However, there are certainly situations when transaction-time support is as crucial as support for valid time, e.g. if the instant when an information was entered or modified in the system has legal implications. The implementation of a prototypical history extension in the context of this thesis will provide support for valid time only. Although named history extension it will also allow for recording of information whose valid time begins in the future. In this regard, sequenced statements (i.e. at each instant of time) will be fully supported.

The term history extension is related to the classification of Worboys (1994) who categorized databases with only valid-time support as historic databases. In fact, the capability to keep track of the history of data is in most cases much more relevant than recording future data.

Concerning the type of version management in the extension, tuple-level versioning will be applied. The valid-time period of a feature will be stored as a line in a column of data type geometry. Instead of the line two timestamp columns could have been used as well to record begin and end of the valid-time period. However, part of the concept is to make use of the spatial analysis capabilities of PostGIS when working with time, assuming that certain kinds of temporal analysis are facilitated this way.
6.1 Scope of Work

The history extension will be implemented by making use of database functionalities as procedures, rules and triggers. The procedures will cover the conversion between time and geometry as well as setting and retrieving the period of applicability for queries and modifications. The rules are required to execute the actual sequenced insertions, updates and deletions while the triggers will ensure sequenced primary keys and referential integrity.

Two existing sample tables parcel and landuse will be converted to add support for valid time. The required conversion steps for a table are as follows:

1. Add a column of type geometry (line in particular) to record the valid-time period of features.
2. Drop the original primary key constraint and, if applicable, foreign key constraint.
3. Rename the table.
4. Create a view on the table, given the original name of the table.
5. Create rules on the view that implement a sequenced INSERT, UPDATE and DELETE.
6. Create triggers on the table to ensure sequenced primary key and, if applicable, referential integrity.

Finally the extension will be tested with the two sample tables by applying various modifications (insertions, updates and deletions) covering past, current and future ones. An example of feature succession will be shown by dividing a parcel. Using the spatial and temporal relationships among parcels the lineage of a particular parcel will be determined.

6.2 Conversion between Time and Geometry

The definition of a temporal coordinate system is provided in ISO (2002b). A discussion and interpretation of this concept can be found in Section 2.5.3 of this thesis. The principle of how to convert time to geometry is explained there, too. According to this principle, the temporal coordinate (C) for an instant of time is
created by expressing the distance between this instant (I) and the origin (O) of the coordinate system as a multiple of the defined scale interval (i) for that system: \( C = (I - O)[i] \). The scale interval can be 'second', 'minute', 'hour', 'day', etc. Figure 6.1 depicts this principle. Given a period defined by two instants of date '2005-01-08' and '2005-01-13', an origin of the temporal coordinate system defined by date '2005-01-01' and a scale interval of 'day', the temporal coordinates of the two instants are 7 and 12. With a scale interval of 'hour', the temporal coordinates were 168 and 288.

![Temporal Coordinates](image)

Figure 6.1: Temporal Coordinates

Vice versa, knowing the origin of the temporal coordinate system, its scale interval and a temporal coordinate in this system, the instant for that temporal coordinate can be calculated as \( I = O + C[i] \).

### 6.3 Sequenced Modifications

As described in the Sections 3.5.3 and 3.5.4, a sequenced update requires five SQL statements while a sequenced deletion requires four SQL statements. The number of statements is necessary to cover the four ways of how a feature’s valid-time period can intersect with the period of applicability. This section goes through each of the five and four statements, respectively and describes how the spatial relationship functions will be used to detect the rows that are targeted by each statement. It is assumed that the period of applicability (PA) as well as the period of validity (PV) of each row are represented by line geometries.

#### 6.3.1 Update

In Figure 6.2 the rows affected by each of the five statements are marked in blue color.
1st INSERT  Insert the old values from the start date to the beginning of the period of applicability. To detect these rows a test is done as follows:

\[
\text{WHERE} \ldots \text{PV} \&\& \text{PA AND ST.Relate(PV, PA, '1********')} \\
\text{AND ST.XMin(PV) < ST.XMin(PA)} ;
\]

Listing 6.1: Finding the target rows of 1st INSERT

The operator “&&” is used to verify if the bounding boxes of the two geometries intersect. Only then, the next condition (ST.Relate(...)) will be tested. The intersection pattern matrix '1********' will test if the interior of PV and the interior of PA intersect, i.e. the periods share at least one granule of time. To eliminate the rows overlapping the end of the PA an additional test is required that compares the X minima of both periods.
2\textsuperscript{nd} INSERT  Insert the old values from the end of the period of applicability to the end date. The target rows of this statement are detected as follows:

\begin{verbatim}
WHERE ...PV && PA AND ST_Relate(PV, PA, '1********')
AND ST_XMax(PV) > ST_XMax(PA);
\end{verbatim}

Listing 6.2: Finding the target rows of 2\textsuperscript{nd} INSERT

1\textsuperscript{st} UPDATE  Update the explicit columns of rows that overlap the period of applicability. The target rows of this statement are detected as follows:

\begin{verbatim}
WHERE ...PV && PA AND ST_Relate(PV, PA, '1********');
\end{verbatim}

Listing 6.3: Finding the target rows of 1\textsuperscript{st} UPDATE

This test will select all rows whose period of validity shares at least one granule of time with the period of applicability.

2\textsuperscript{nd} UPDATE  Update the start date to begin at the beginning of the period of applicability of rows that overlap the period of applicability. The target rows of this statement are detected as follows:

\begin{verbatim}
WHERE ...PV && PA AND ST_Relate(PV, PA, '1********')
AND ST_XMin(PV) < ST_XMin(PA);
\end{verbatim}

Listing 6.4: Finding the target rows of 2\textsuperscript{nd} UPDATE

3\textsuperscript{rd} UPDATE  Update the end date to end at the end of the period of applicability of rows that overlap the period of applicability. The target rows of this statement are detected as follows:

\begin{verbatim}
WHERE ...PV && PA AND ST_Relate(PV, PA, '1********')
AND ST_XMax(PV) > ST_XMax(PA);
\end{verbatim}

Listing 6.5: Finding the target rows of 3\textsuperscript{rd} UPDATE

6.3.2 Deletion

Again, the rows affected by each of the four statements required for a sequenced deletion are marked in blue color.
**CHAPTER 6. CONCEPT**

**Figure 6.3: Target rows for sequenced deletion (in blue)**

**INSERT**  
Insert the old values from the end of the period of applicability to the end of the period of validity of the original row. The target rows of this statement are detected as follows:

```
...WHERE...ST.ContainsProperly(PV, PA);
```

Listing 6.6: Finding the target rows of INSERT

The function ST.ContainsProperly(...) tests if interior and boundary of the line representing the period of applicability are within the interior of the line representing the row’s period of validity.

**1st UPDATE**  
Update the end date to end at the beginning of the period of applicability. The target rows of this statement are detected as follows:

```
...WHERE...PV && PA AND ST.Relate(PV, PA, '1********')
AND ST.XMin(PV) < ST.XMin(PA);
```

Listing 6.7: Finding the target rows of 1st UPDATE

**2nd UPDATE**  
Update the start date to begin at the end of the period of applicability. The target rows of this statement are detected as follows:
DELETE Delete entirely rows that are covered by the period of applicability. The target rows of this statement are detected as follows:

Listing 6.9: Finding the target rows of DELETE

6.4 Temporal Keys and Referential Integrity

Primary Key By definition a sequenced primary key is one that applies at each point in time. Hence, no sequenced duplicates (Section 3.4.3) are allowed. To verify this condition using spatial relationship functions, a statement as in Listing 6.10 can be used.

Listing 6.10: Testing for primary key violations

This statement counts the number of rows that represent the same feature but overlap each other. A number greater than 0 indicates that such rows exist and the sequenced primary key is violated. This sample is to show the principle. In practice this check has to be done before a new row is actually inserted in the table.

Referential Integrity If two temporal tables are involved in a relationship and thus one of the two tables references the other one, a sequenced foreign key must be applied to establish referential integrity. To recall the definition of Snodgrass (2000), a key is a sequenced foreign key if, for all rows $r$ in the referencing table,

- there is a row with that key value valid in the referenced table when $r$ started,
• there is a row with that key value valid in the referenced table when r stopped,

• and there are no gaps when there are no rows in the referenced table, during r’s period of validity, that have that key value.

Figure 6.4 shows in red color the period of validity of a land use plot to be inserted in the database. This land use plot shall reference an existing parcel that has a period of validity shown in blue color. Of the six cases presented, only e) and f) fulfil the definition of the sequenced foreign key as given above. Cases a)–d) violate the sequenced foreign key by either missing rows at the beginning / end or by gaps. From Figure 6.4 also the principle to verify the sequenced foreign key conditions by spatial analysis functions can be derived: Only if the spatial Difference between the period of validity of the land use plot and the valid-time period of the parcel is an empty geometry the sequenced foreign key is valid. The Difference will be an empty geometry only if the valid-time period of the parcel fully covers the valid-time period of the land use plot. Whenever the Difference is a non-empty geometry (shown as grey arrows in the figure), the sequenced foreign key is violated.

An SQL statement as follows can be used to test the foreign key conditions. Compared to the sequenced foreign key constraint provided in Listing 3.6, this one is simple but sufficient. If the statement returns a number greater than 0, the sequenced foreign key is violated. Again, in practice this test should be run before the new row is actually inserted. The $ST_{\text{Union}}(...)$ function is used to join the valid-time periods of the parcel as it would be required for case f) in Figure 6.4.
This must be done before the spatial difference between the valid-time period of the land use plot and the parcel is determined.

```
SELECT COUNT(*) FROM landuse lu WHERE lu.parcel = '18/2'
AND NOT ST_IsEmpty(ST_Difference(lu.validtime,
(SELECT ST_Union(p.validtime)
FROM parcel p WHERE p.parcelId = lu.parcel)));
```

Listing 6.11: Testing for foreign key violations

### 6.5 Period of Applicability

Within the history extension the period of applicability (the “global” valid time) must be set before any query or modification statement is executed. A conformable procedure will be provided. Once the period of applicability is set, it serves as a filter for all subsequent queries and modification statements. The client application never needs to provide the valid-time period in the SQL statements explicitly. In fact, even if the application did so, the time would simply be ignored and the period of applicability used instead.

### 6.6 Data Model

A conceptual data model was created in UML (Unified Modelling Language) and depicts the two feature classes that will be implemented as tables and be used to test the history extension (Figure 6.5). The classes as shown in the model have no temporal support yet. Both classes represent spatial features indicated by the geometry property. The relationship modelled between the two feature classes will be implemented by a foreign key constraint on table `landuse`.

![Figure 6.5: Conceptual Data Model](image-url)
Chapter 7

Implementation

This chapter goes through the steps required for the implementation of the history extension. Besides the triggers, rules and views, a number of database procedures had to be written, but only the most important ones among them will be discussed here. Many of the procedures are of supportive character only and are used from within other procedures.

7.1 Assumptions

A number of assumptions were required for the implementation of the history extension:

- The valid-time period of features is stored as line geometry.

- If the valid-time period of a feature is provided by two timestamps, the closed-open representation of the period shall be used.

- ‘infinity’ is the latest possible instant but will be represented as date '9999-12-31' when time is converted to geometry. If instants later than '9999-12-31' are provided, they will be set to '9999-12-31' implicitly.

- Branching of time is not supported.

- The primary key of a feature cannot be modified. This implies that new features (meaning features with a new ID) can only be created by an INSERT statement.

- The existing user tables have no temporal support initially.
• PostgreSQL 8.4.1 and PostGIS 1.4.0 are being used.

7.2 Tables

Tables in the context of the history extension are categorized in metadata tables and user tables. Metadata tables hold information explicitly required by the extension, while user tables record the information of a certain domain or field of application.

7.2.1 Metadata Tables

Three metadata tables are required by the extension: \texttt{tm\_coordinatesystem}, \texttt{tm\_config} and \texttt{tm\_validtime}.

\texttt{tm\_coordinatesystem}  This table records temporal coordinate systems, each defined by an ID, the origin of the system provided as an instant in the Gregorian calendar and UTC, and an interval which can currently be of type 'second', 'minute', 'hour' or 'day' (Table 7.1). This table should be granted write access only for administrators. If the origin or interval of a temporal coordinate system is changed while there are user tables that already hold valid time, unexpected results will occur.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\texttt{tcsid} & \texttt{origin} & \texttt{interval} \\
\hline
1 & 1800-01-01 00:00:00+00 & second \\
2 & 2000-05-01 00:00:00+00 & day \\
3 & 1910-10-23 00:00:00+00 & hour \\
\hline
\end{tabular}
\caption{tm\_coordinatesystem}
\end{table}

\texttt{tm\_config}  This table (Table 7.2) can hold various configuration parameters of the extension, but only one parameter is currently defined: \texttt{tcsid}, which is the ID of the temporal coordinate system to be used and refers to \texttt{tcsid} as defined in table \texttt{tm\_coordinatesystem}.

\texttt{tm\_validtime}  This is the table to record the period of applicability (or “global” valid time) that is set by each database user. The period of applicability is defined
Table 7.2: tm_config

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcsid</td>
<td>2</td>
</tr>
</tbody>
</table>

per user since two users might require a different period of applicability when querying or modifying data.

Table 7.3: tm_validtime

<table>
<thead>
<tr>
<th>userName</th>
<th>validFrom</th>
<th>validTill</th>
</tr>
</thead>
<tbody>
<tr>
<td>wagner</td>
<td>1995-01-01 00:00:00+00</td>
<td>infinity</td>
</tr>
<tr>
<td>schmieder</td>
<td>2000-01-01 00:00:00+00</td>
<td>2007-01-01 00:00:00+00</td>
</tr>
</tbody>
</table>

7.2.2 User Tables

Based on the conceptual data model defined in Section 6.6, two user tables parcel and landuse were created to record the application data. The required SQL statements are shown in the Listings 7.1 and 7.2.

Listing 7.1: Creating the parcel table

```sql
CREATE TABLE parcel (  
    fid serial unique not null, -- only required for the GIS  
    parcelId varchar(10) primary key,  
    legalSize decimal,  
    description text  
) ;
SELECT AddGeometryColumn ('', 'parcel', 'geometry',  
    32634, 'POLYGON', 2) ;
CREATE INDEX spatial_idx_parcel_geometry ON  
    parcel USING GIST (geometry) ;
```

Both tables are not yet temporal tables, support for valid time will be added in a later step. It can be observed that the column parcel is a foreign key in table landuse. This must be taken into account by a sequenced foreign key constraint when adding temporal support to the tables later on.

Either table has a column of type geometry since parcels and land use plots are considered as spatial objects. Both tables have also a column fid with a UNIQUE
CHAPTER 7. IMPLEMENTATION

constraint. This column was added only to satisfy the GIS client application. Many GIS clients expect a unique attribute of type Integer to be used as primary key internally. Serial is an auto-incrementing integer type available in PostgreSQL.

```
CREATE TABLE landuse (  
    fid serial unique not null, --only required for the GIS  
    landuseId varchar(10) primary key,  
    landuseType varchar(20),  
    description text,  
    parcel varchar(10) references parcel(parcelId)  
);  
SELECT AddGeometryColumn(' ', 'landuse', 'geometry',  
    32634, 'POLYGON', 2);  
CREATE INDEX spatial_idx_landuse.geometry ON  
    landuse USING GIST (geometry);  
```

Listing 7.2: Creating the landuse table

7.3 Temporal Procedures

In this section the database procedures that were written for temporal support are described. The first two procedures `transformDateTime(...)` and `transformCoord(...)` are based on the definition of a temporal coordinate system and its methods as defined in ISO (2002b). The listings for all procedures can be found in Appendix A to this thesis.

`transformDateTime(timestamp with time zone):tm_coordinate` The procedure converts an instant of time to a temporal coordinate based on the principle described in Section 6.2 of this thesis. `TM.Coordinate` is a data type defined in ISO (2002b) that consists of two attributes `indeterminatePosition` and `coordinateValue`. Providing a value for the first attribute is optional, while it is mandatory for the second one. `TM.Coordinate` was created as a custom data type in PostgreSQL with the statements as shown in Listing 7.3.

```
CREATE TYPE tm_indeterminateValue  
    AS ENUM ('unknown', 'now', 'before', 'after');  
CREATE TYPE tm_coordinate AS (  
```
indeterminatePosition tm_indeterminateValue, coordinateValue int8);

Listing 7.3: Creating a data type tm_coordinate

The procedure itself is shown in Listing A.1 in the appendix to this document. The procedure queries the table tm_config to find the temporal coordinate system currently set. It then queries table tm_coordinatesystem to retrieve origin and interval defined for that system. Given those two parameters the temporal coordinate tm_coordinate is calculated for the provided instant of time and returned by the procedure.

transformCoord(tm_coordinate):timestamp with time zone Given a temporal coordinate the procedure will convert it to an instant of time. The procedure retrieves the parameters required for the conversion, origin and interval of the temporal coordinate system from the related tables tm_config and tm_coordinatesystem. The procedure can be found in Listing A.2 in the appendix to this document.

lineFromTimeHelper(timestamp with time zone, timestamp with time zone):geometry Provided with two instants of time, the procedure will create a line geometry, representing a period of time. To construct the actual line, the Well Known Text format (WKT) is used. The procedure makes use of procedure transformDateTime(...) internally. It can be found in Listing A.3.

Further procedures Most of the following procedures are in some way based on the procedures previously described:

setPA(timestamp with time zone, timestamp with time zone) sets the period of applicability per database user. It will be stored in the metadata table tm_validtime together with the name of the database user. This is the only procedure to be used by a client application directly (Listing A.4).

paFrom():timestamp with time zone returns the instant of the begin of the period of applicability (Listing A.5).

paTill():timestamp with time zone returns the instant of the end of the period of applicability (Listing A.6).
lineFromPA(geometry) returns a line representing the period of applicability (Listing A.7).

lineFromTime(timestamp with tz, timestamp with tz):geometry creates a line from two instants of time. Should be used instead of lineFromTime-Helper(...) but uses that procedure internally (Listing A.8).

validFrom(geometry):timestamp with time zone provided with a line geometry representing a period of time, the procedure will return the instant of the begin of that period (Listing A.9).

validTill(geometry):timestamp with time zone provided with a line geometry representing a period of time, the procedure will return the instant of the end of that period (Listing A.10).

## 7.4 Conversion of User Tables

Since the assumption was made that existing user tables have no temporal support initially, valid-time support has to be added at some point later. Given the table landuse, Listing 7.4 shows the initial steps to convert it to a table with temporal support. All subsequent steps are explained for table landuse but can be applied in the same way to table parcel. If there is any difference, it will be mentioned explicitly.

```sql
SELECT AddGeometryColumn(' ', 'landuse', 'validtime', -1, 'LINESTRING', 2);
CREATE INDEX spatial_idx_landuse_validtime ON landuse USING GIST (validtime);
ALTER TABLE landuse DROP CONSTRAINT landuse_pkey;
ALTER TABLE landuse DROP CONSTRAINT landuse_parcel_fkey;
ALTER TABLE landuse RENAME to landuse_vt;
UPDATE landuse_vt SET validtime = lineFromTime(CURRENT_TIMESTAMP, 'infinity');
ALTER TABLE landuse_vt ALTER COLUMN landuseId SET NOT NULL;
ALTER TABLE landuse_vt ALTER COLUMN validtime SET NOT NULL;
```

Listing 7.4: Initial table modification
In line 1 a new attribute \textit{validtime} of type geometry (\textit{line}) is added to the table to record the valid-time period of land use plots. In line 2 a spatial index is created for the new attribute. In lines 3 and 4, the original primary and foreign key constraints are dropped. In line 5 the table is renamed (given the suffix \textit{vt} for valid time) and in line 6 the valid time of all rows is set to last from the current instant to infinity. As a result all rows are current. In lines 7 and 8 a NOT NULL constraint is set for the attributes \textit{landuseId} and \textit{validtime}. Excluding the statement in line 4, the same steps can be applied to table \textit{parcel}. Table \textit{parcel} has no foreign key and, thus, no foreign key constraint must be dropped.

7.4.1 Creating a View

As next step a view is created on the table and it is given the original name of the table. The SQL statement looks as follows:

\begin{verbatim}
CREATE OR REPLACE VIEW landuse AS
    SELECT fid, landuseId, landuseType, description, parcel,
        geometry, validFrom(validtime) AS validFrom,
        validTill(validtime) AS validTill FROM landuse_vt WHERE
        validtime && lineFromPA() AND
        ST_Relate(validtime, lineFromPA(), '1********');
\end{verbatim}

Listing 7.5: Creating a view

There are several things to be observed in this view. Begin and end of the valid-time period are provided as timestamps in the view. That makes sense since a client application cannot make use of the valid-time period if provided as line geometry. The procedures \textit{validFrom(...)} and \textit{validTill(...)} are used to retrieve the timestamps from the geometry. The view is created using a WHERE clause. Because of this clause, the view will select only those rows that overlap with the period of applicability. Within the limits of the period of applicability a client can execute further time-based queries using the timestamp attributes \textit{validFrom} and \textit{validTill} available with the view.

7.4.2 Rules and Sequenced Modifications

Given the \textit{landuse} view just created, the client cannot yet make any modifications to the data. \textit{Rules} are required to create the illusion of an updatable view. Within
these rules, the SQL statements required for a sequenced insertion, update or deletion will be executed on the underlying table `landuse_vt`.

**Insertion**  Listing 7.6 shows how the INSERT rule is created on the `landuse` view. This rule defines that if an INSERT command is applied to the `landuse` view, a sequenced insertion will be executed on table `landuse_vt` instead. The valid-time period for the new row is the period of applicability which is returned by the procedure `lineFromPA()`. **NEW** is a special variable available within an INSERT rule that references the new row to be inserted.

```sql
CREATE OR REPLACE RULE insert_landuse_rule AS
  ON INSERT TO landuse DO INSTEAD
  (  
    INSERT INTO landuse_vt (landuseId, landuseType, description, parcel, geometry, validtime) VALUES
      (NEW.landuseId, NEW.landuseType, NEW.description, NEW.parcel, NEW.geometry, lineFromPA());
  ) ;
```

Listing 7.6: Creating an INSERT rule

**Update**  The UPDATE rule for the `landuse` view is created using the statements in Listing 7.7. Within the rule five statements are executed on the underlying table. These are the five statements (two INSERTs and three UPDATEs) required for a sequenced update as described in Sections 3.5.3 and 6.3.1. Discussed in the mentioned sections is what each statement does in particular, as well as how the target rows of the statement are detected using spatial functions. Within the rule **NEW** is a variable referencing the new row being updated, while **OLD** refers to the existing row.

```sql
CREATE OR REPLACE RULE update_landuse_rule AS
  ON UPDATE TO landuse DO INSTEAD
  (  
    INSERT INTO landuse_vt (landuseId, landuseType, description, parcel, geometry, validTime) SELECT landuseId, landuseType, description, parcel, geometry, lineFromTime(validFrom(validTime),
```
paFrom() FROM landuse_vt WHERE landuseId = OLD.landuseId AND validTime && lineFromPA() AND ST_Relate(validTime, lineFromPA(), '1********') AND ST_XMin(validTime) < ST_XMin(lineFromPA()) AND landuse_vt.validtime = linefromtime(OLD.validfrom, OLD.validtill);

INSERT INTO landuse_vt (landuseId, landuseType, description, parcel, geometry, validTime) SELECT landuseId, landuseType, description, parcel, geometry, lineFromTime(paTill(), validTill(validTime)) FROM landuse_vt WHERE landuseId = OLD.landuseId AND validTime && lineFromPA() AND ST_Relate(validTime, lineFromPA(), '1********') AND ST_XMax(validTime) > ST_XMax(lineFromPA()) AND landuse_vt.validtime = linefromtime(OLD.validfrom, OLD.validtill);

UPDATE landuse_vt SET landuseId = NEW.landuseId, landuseType = NEW.landuseType, description = NEW.description, parcel = NEW.parcel, geometry = NEW.geometry WHERE landuseId = OLD.landuseId AND validTime && lineFromPA() AND ST_Relate(validTime, lineFromPA(), '1********');

UPDATE landuse_vt SET validTime = lineFromTime(paFrom(), validTill(validTime)) WHERE landuseId = NEW.landuseId AND validTime && lineFromPA() AND ST_Relate(validTime, lineFromPA(), '1********') AND ST_XMin(validTime) < ST_XMin(lineFromPA());

UPDATE landuse_vt SET validTime = lineFromTime(validFrom(validTime), paTill()) WHERE landuseId = NEW.landuseId AND validTime && lineFromPA() AND ST_Relate(validTime, lineFromPA(), '1********');
Deletion  A sequenced deletion requires four statements, an INSERT, two UPDATES and a DELETE. These statements are executed within a DELETE rule that is created on the \textit{landuse} view (Listing 7.8). Discussed in Sections 3.5.4 and 6.3.2 is the meaning of each statement, as well as how the target rows of the statement are detected. Within the rule \textit{OLD} references the row to be deleted.

```
CREATE OR REPLACE RULE delete_landuse_rule AS
  ON DELETE TO landuse DO INSTEAD
  (  
    INSERT INTO landuse_vt (landuseId, landuseType, description, parcel, geometry, validTime) SELECT 
      landuseId, landuseType, description, parcel, geometry, lineFromTime(patill()), validTill(validTime) FROM landuse_vt WHERE landuseId = OLD.landuseId AND ST_ContainsProperly(validTime, lineFromPA());

    UPDATE landuse_vt SET validTime = lineFromTime(patill(), validTill(validTime)) WHERE landuseId = OLD.landuseId AND validTime && lineFromPA() AND ST_XMin(validTime) < ST_XMin(lineFromPA());

    UPDATE landuse_vt SET validTime = lineFromTime(patill(), validTill(validTime)) WHERE landuseId = OLD.landuseId AND ST_Overlaps(validTime, lineFromPA());

    DELETE FROM landuse_vt WHERE landuseId = OLD.landuseId AND ST_Within(validTime, lineFromPA());  
  );
```

Listing 7.7: Creating an UPDATE rule
7.4.3 Triggers, Keys and Referential Integrity

Ensuring the sequenced primary key SQL Assertions and complex table constraints are not supported by PostgreSQL. Hence, a trigger must be used to ensure the sequenced primary key for table landuse_vt. This trigger must be fired before a row is actually inserted or updated in landuse_vt. When the trigger fires, it executes the trigger procedure that is shown in Listing 7.9.

```
CREATE OR REPLACE FUNCTION seq_pkey_landuse() RETURNS trigger AS
$BODY$
DECLARE
  v_count int4;
BEGIN

  IF (TG_OP = 'UPDATE') THEN
    IF (NEW.landuseId != OLD.landuseId) THEN
      RAISE EXCEPTION 'Cannot modify primary key values for tables with valid-time support!';
    ELSE
      RETURN NEW;
    END IF;
  END IF;

  IF (NOT ST_Within(NEW.validTime, lineFromPA())) THEN
    RETURN NEW;
  END IF;

  SELECT count(*) FROM landuse_vt p WHERE p.landuseId = NEW.landuseId AND p.validTime && NEW.validTime AND ST_Relate(p.validtime, NEW.validtime, '1********') INTO v_count;

$BODY$
```

Listing 7.8: Creating a DELETE rule
IF (v_count > 0) THEN
   RAISE EXCEPTION 'Sequenced primary key violated!';
END IF;

RETURN NEW;

Listing 7.9: Trigger procedure ensuring the sequenced primary key

As with the rules, within a trigger procedure the existing row to be updated or deleted is referred to by the OLD variable, while the new row to be updated or inserted is referenced by NEW. In lines 8–14 it is checked that a row update does not change the primary key since this is currently not supported by the extension. Lines 16–18 ensure that a row insertion caused by a sequenced update or deletion does not undergo the test for overlapping since it would (temporary) fail the test and the operation would be cancelled. However, a row insertion caused by an INSERT command of the client is tested in lines 20–24. Should this row overlap any existing row with the same landuseId, a warning will be shown and the insertion will be cancelled. The principle of this (spatial) test is discussed in Section 6.4.

Ensuring the sequenced foreign key A second trigger is fired before an insertion or update on table landuse_vt, ensuring the sequenced foreign key. The related trigger procedure is shown in Listing 7.10.

CREATE OR REPLACE FUNCTION seq_fkey_landuse ()
   RETURNS trigger AS
$BODY$
DECLARE
   v_count int4;
   v_gap geometry;
BEGIN


IF (NEW.parcel IS NULL) THEN
  RAISE EXCEPTION '<parcel> cannot be null!';
END IF;

IF (NOT ST_Within(NEW.validTime, lineFromPA())) THEN
  RETURN NEW;
END IF;

SELECT count(*) FROM parcel_vt INTO v_count WHERE parcelId = NEW.parcel;

IF (v_count = 0) THEN
  RAISE EXCEPTION 'No parcel with ID <parcel> existing!', NEW.parcel;
END IF;

SELECT ST_Difference(NEW.validtime, (SELECT st_union(validtime) FROM parcel_vt WHERE parcelId = NEW.parcel)) INTO v_gap;

IF (NOT ST_IsEmpty(v_gap)) THEN
  RAISE EXCEPTION 'Sequenced foreign key violated! Parcel <parcel> to be referenced by land use plot <plot> must at least be valid during the same period!', NEW.parcel, NEW.landuseId;
END IF;

RETURN NEW;

END;

$BODY$
  LANGUAGE 'plpgsql' VOLATILE
  COST 100;
Listing 7.10: Trigger procedure ensuring the sequenced foreign key (a)

This trigger checks various conditions. Lines 9–11 verify that the foreign key in table `landuse_vt` actually contains a value. Each land use plot must reference exactly one parcel. In case a foreign key value is provided, lines 17–21 take care that a parcel with that key exists. Lines 23–27 test that the referenced parcel exists for the whole life time (valid time) of the inserted or updated land use plot. Again, the testing principle is explained in Section 6.4.

Unfortunately this trigger is not sufficient to ensure the sequenced foreign key in table `landuse_vt`. The key can still be violated by a sequenced deletion on the referenced table (`parcel_vt`). It could even be violated by a sequenced update on that table if primary keys were allowed to change. Hence, another trigger is required, but on table `parcel_vt` this time. This trigger fires after an update or deletion on `parcel_vt` and executes the procedure in Listing 7.11. It is crucial that this trigger fires AFTER the update or deletion. Otherwise the changes to be verified will not be visible to the test statement yet. The test that is done within this trigger is similar to the one in the previous trigger.

```sql
CREATE OR REPLACE FUNCTION seq_fkey_landuse_rev()
RETURNS trigger AS
$BODY$
DECLARE
    v_landuse RECORD;
BEGIN

    SELECT * FROM landuse_vt l INTO v_landuse WHERE l.parcel = OLD.parcelId AND NOT ST.IsEmpty(ST.Difference(l.validtime, (SELECT st_union(p.validtime) FROM parcel_vt p WHERE p.parcelId = OLD.parcelId)))

    IF FOUND THEN
        RAISE EXCEPTION 'Sequenced foreign key violated! Parcel is referenced by land use plot and must at least be'
    END IF

$BODY$
END;
```

valid during the same period!
parcellId, v_landuse.landuseId;

END IF;

RETURN NULL;
END;

$BODY$
LANGUAGE 'plpgsql' VOLATILE
COST 100;

Listing 7.11: Trigger procedure ensuring the sequenced foreign key (b)

To summarize what has been discussed so far; on either table a trigger is required that ensures the primary key of that table. Furthermore a second trigger is required, again on both tables, ensuring only the sequenced foreign key in table landuse_vt. With the triggers created, the conversion of the tables is complete—the tables are temporal tables with support for an automated maintenance of history.

7.5 Testing the Extension

Two fictive scenarios from the domain of land administration are used to test the extension. The results are shown in psql, a command-line client for PostgreSQL. To demonstrate the extension’s support for sequenced modifications (at each instant of time), the tests are done using future data as well. Before creating any data, the temporal coordinate system must be set properly. For the test a system with an origin of date '1900-01-01 00:00:00+00' and an interval of 'second' will be used. The system is created and set as follows:

| INSERT INTO tm_coordinateSystem VALUES (5, '1900-01-01-00:00:00+00', 'second'); |
| UPDATE tm_config SET value = '5' WHERE parameter = 'tcsid'; |

Listing 7.12: Setting the temporal coordinate system
7.5.1 Example 1

A scenario as depicted in Figure 7.1 is used. The scenario shows some cadastral parcels that will experience various mutations over time. These mutations include feature succession (in particular two divisions and one fusion), i.e. one set of parcels will be replaced by another set of parcels as occurring in cases b)–d). Only sequenced insertions and deletions are required to implement the example. Sequenced updates and referential integrity are covered by Example 2.

Case a) Either parcel starts becoming valid on May 12, 2002. The period of applicability must be set accordingly and two insertions executed. The required statements are shown in Listing 7.13. If only one instant of time is provided to the setPA(...) procedure, it is assumed as the begin of the period of applicability. The end of the period will be set to 'infinity' in this case.
CHAPTER 7. IMPLEMENTATION

74

select setPA('2002-05-12');
insert into parcel (parcelId, description, geometry) values
('10/2', 'Case a', ST_GeomFromText('POLYGON((100,200,100,400,300,400,300,300,100,200))', 32634));
insert into parcel (parcelId, description, geometry) values
('27', 'Case a', ST_GeomFromText('POLYGON((300,200,300,400,500,400,500,500,200,300,200))', 32634));

Listing 7.13: Example 1 – Case A

Case b) On October 25, 2004 parcels '10/3' and '10/4' become “alive” as a result of the division of parcel '10/2'. Thus, a feature succession occurs. Parcel '10/2' will be deleted and two new parcels created instead. The related statements can be found in Listing 7.14. Parcel '10/2' will not be deleted physically from the database but its valid-time end will be set accordingly. All results will be shown shortly.

select setPA('2004-10-25');
delete from parcel where parcelId = '10/2';
insert into parcel (parcelId, description, geometry) values
('10/3', 'Case b', ST_GeomFromText('POLYGON((100,200,100,400,200,400,200,200,100,200))', 32634));
insert into parcel (parcelId, description, geometry) values
('10/4', 'Case b', ST_GeomFromText('POLYGON((200,200,200,400,300,400,300,300,200,200,300,200))', 32634));

Listing 7.14: Example 1 – Case B

Case c) Parcels '27/1' and '27/2' become alive on July 11, 2007 as a result of the division of parcel '27'. Again, a sequenced deletion and two insertions are required (Listing 7.15).

select setPA('2007-07-11');
delete from parcel where parcelId = '27';
insert into parcel (parcelId, description, geometry) values
('27/1', 'Case c', ST_GeomFromText('POLYGON((300,200,300,400,400,400,400,200,300,200))', 32634));
**CHAPTER 7. IMPLEMENTATION**

```sql
insert into parcel (parcelId, description, geometry) values
( '27/2', 'Case c', ST_GeomFromText('POLYGON((400,200,400,400,500,400,400,500,200,200,200,200))', 32634));
```

Listing 7.15: Example 1 – Case C

**Case d)** Finally, parcel '31' is created on December 24, 2012 as a result of the fusion of parcels '10/4' and '27/1'. This requires two deletions and one insertion (Listing 7.16).

```sql
select setPA('2012-12-24');
delete from parcel where parcelId = '10/4';
delete from parcel where parcelId = '27/1';
insert into parcel (parcelId, description, geometry) values
( '31', 'Case d', ST_GeomFromText('POLYGON((200,200,200,400,400,400,200,200,200))', 32634));
```

Listing 7.16: Example 1 – Case D

The resulting `parcel` table is shown in Figure 7.2. The parcels are ordered by begin of their valid time. To be able to list all parcels from the example, the period of applicability has to be set to an instant equal to or before the first mutation. For a better readability the `geometry` column of the parcels is not shown.

Temporal and spatial relationships among features can now be used to determine their lineage. To find all predecessors of parcel '31' after November 1, 2004 the statements as follows can be applied:

```sql
select setPA('2004-11-01');
select b.parcelId, b.description, b.validFrom, b.validTill
from parcel a, parcel b
where a.parcelId = '31' and st_relate(a.geometry, b.geometry, '2*********') and a != b
order by validFrom;
```

Listing 7.17: Example 1 – Lineage

The result is shown in Figure 7.3. The additional condition `a != b` is required to exclude parcel '31' itself from the result list.
7.5.2 Example 2

Figure 7.4 shows the scenario for Example 2. Information about a land use plot (ID '13') became known. The plot is located on parcel '31' and of type 'Forest'. At some point later the forest is cut and the plot used for 'Agriculture'. Again later, the land of the plot is required for a north-south road. This example covers sequenced insertions, updates and referential integrity.

Case a  To create the plot a sequenced insertion is required. The plot is known to become valid on November 20, 2012. The plot has a foreign key which is the ID of parcel '31'. An initial attempt to insert the plot will fail as parcel '31' does not exist yet at that time and the foreign key would be violated. Thus, the plot’s valid-time begin will be set to December 24 only (the day the parcel becomes valid). The required statements are as follows, while the results are shown in Figure 7.5.

```
select setPA('2012-11-20');
-- next statement will fail
insert into landuse (landuseId, landuseType, parcel, geometry) values ('13', 'Forest', '31', ST_GeomFromText('POLYGON((330 200, 330 400, 400 400, 400 200, 330 200))', 32634));
```
CHAPTER 7. IMPLEMENTATION

Figure 7.3: Lineage of parcel ’31’

```
select setPA('2012-12-24');

-- next statement will work
insert into landuse (landuseId, landuseType, parcel, geometry)
values('13', 'Forest', '31', ST_GeomFromText('POLYGON((330 200, 330 400, 400 400, 400 200, 330 200))', 32634));
```

Listing 7.18: Example 2 – Case A

**Case b**  Beginning on May 2, 2017 the plot will be used for agricultural purposes. This requires a sequenced update as follows:

```
select setPA('2015-05-02');
update landuse set landuseType = 'Agriculture' where landuseId = '13';
```

Listing 7.19: Example 2 – Case B

**Case c**  Another change occurs on June 6, 2020 when the plot starts being used for a road. Again, a sequenced update is required:

```
select setPA('2020-06-06');
```
update landuse set landuseType = 'Road' where landuseId = '13';

Listing 7.20: Example 2 – Case C

The resulting landuse table with the three cases implemented is shown in Figure 7.6. The geometry column was excluded again for the purpose of a better readability.

Finally the effectiveness of the sequenced foreign key constraint is tested once more. Assuming that the valid time of parcel ’31’ ends on February 1, 2022, an attempt is made to execute a sequenced deletion accordingly:

select setPA ('2022-02-01');
delete from parcel where parcelId = '31';

Listing 7.21: Example 2 – Deletion attempt

This attempt will fail as the parcel is still referenced by land use plot ’13’ (Figure 7.7).
 CHAPTER 7. IMPLEMENTATION

Figure 7.5: Example 2 – Failed, and successful insertion attempt

Figure 7.6: Example 2 – Result
CHAPTER 7. IMPLEMENTATION

Figure 7.7: Deletion attempt on parcel '31'
Chapter 8

Conclusion

8.1 Findings

The aim of this thesis was to implement a prototype of an extension to PostgreSQL/PostGIS allowing for an automated maintenance of valid time and the related history. The implementation was to be performed entirely on the database server, making the modification of any client application unnecessary. Under such implementation, the valid time of features was to be stored as geometry.

Observing the results achieved, they come close to the original goal and prove that the required functionality can be provided solely by the database server. Thus, a client application can express the changes to the data in its domain in a “natural” way without the need of complex time-oriented SQL statements. A single procedure (setPA(’2006-03-01’, ’infinity’)) is required to be called by the client to set the period of applicability for all subsequent queries and modifications. Everything else is handled by the database server. To implement the required functionality, some crucial modifications had to be made to the existing database schema. They included the creation of database procedures and a number of triggers, rules and views on the existing user tables. If referential integrity was established in the original schema, it had to be ensured also in the schema with temporal support. For the prototype the user tables were manually converted to tables with temporal support. That was feasible since only two tables were used. However, in case of a vast database schema with many user tables, procedure which performs the necessary conversion in an automated way would be required. This procedure would be provided only with the name of the user table to be converted e.g. addValidTimeSupport(’parcel’).
CHAPTER 8. CONCLUSION

Given that the valid-time period of features is stored as geometry, the entire temporal analysis required for the implementation of the extension is performed using solely spatial functions, which works without any restrictions. The functions include relationship functions as well as creative operations as difference, union, etc. However, there is no obvious benefit from treating time as geometry, at least not in the context of this thesis. There are situations in which a temporal condition can be tested easier by using a spatial function, while there are others where a native temporal function or operator might be more appropriate.

Problem which occurs while testing the extension is the fact that most (GIS) client applications require either a PRIMARY KEY or a UNIQUE constraint on some column of the user table. If such constraint cannot be found, the applications refuse to work with the table. A temporal primary key is ensured via a sequenced constraint—thus, a PRIMARY KEY constraint as such does not exist anymore. A combined UNIQUE constraint could be created on the original key column and the valid-time column. It would satisfy the client application in some cases, but would not be sufficient to ensure the temporal primary key (the sequenced constraint is still required).

Another obstacle is the fact that in most GIS applications the splitting of a polygon (as required for a parcel division) will result in an UPDATE and an INSERT command. Since the original polygon is replaced by two (or more) new polygons, it would only be natural that the original polygon is removed by a DELETE command and not simply be updated to become one of the new polygons. The history extension expects a deletion and two insertions for the described case. It will not function correctly otherwise. This is related to the current restriction that temporal primary keys cannot be updated. It is on the other hand arguable whether a key should be updatable considering that it is the unique identifier of a feature (for a lifetime?).

8.2 Open Issues

The history extension is a prototype only. In order for it to be used in a production system it will have to be extended and tested thoroughly to determine if all temporal situations which might occur in a domain or field of application are covered. It is usually the special cases that present obstacles. As mentioned in the previous section, a supportive procedure (e.g. addValidTimeSupport(tableName varchar(30))) would be very helpful if a vast number of user tables need to be
converted to become temporal. Behind the “scenes” this procedure has to re-
trieve information from a number of PostgreSQL’s metadata views and tables to
determine the schema of the particular user table (i.e. what columns, constraints,
primary and foreign keys, etc.). It can then create the views, triggers and rules on
the user table that are required for temporal support and that were only manually
added in the prototype.

So far, the extension was tested using only a small amount of data. It might
be worthwhile to analyse how the extension performs (particularly the spatial
functions) with hundreds of thousands of rows in a table. On the other hand, the
line geometry representing the valid-time period is very simple and a spatial index
has been created on that column. Thus, performance might not really be an issue.

As described in Section 7.4.2, the five and four statements, respectively that
are required for a sequenced update and deletion are executed within a rule. If
a rule is applied, information about the number of affected rows will stem only
from the last statement that is executed within that rule. This information is not
very useful for the client. It would be much better if a summary of the number
of rows affected by all statements is returned to the client—which would require
a customized procedure.

In certain situations a sequenced insertion or update might result in a table as
follows:

<table>
<thead>
<tr>
<th>landuseId</th>
<th>landuseType</th>
<th>parcel</th>
<th>validFrom</th>
<th>validTill</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>Forest</td>
<td>200/2</td>
<td>1995-01-01</td>
<td>2002-08-01</td>
</tr>
<tr>
<td>112</td>
<td>Agriculture</td>
<td>200/2</td>
<td>2002-08-01</td>
<td>2004-02-01</td>
</tr>
<tr>
<td>112</td>
<td>Agriculture</td>
<td>200/2</td>
<td>2004-02-01</td>
<td>2007-05-11</td>
</tr>
<tr>
<td>112</td>
<td>Agriculture</td>
<td>200/2</td>
<td>2007-05-11</td>
<td>2013-02-01</td>
</tr>
<tr>
<td>112</td>
<td>Road</td>
<td>200/2</td>
<td>2013-02-01</td>
<td>infinity</td>
</tr>
</tbody>
</table>

Table 8.1: A case for coalescing

Rows 2–4 are value-equivalent as they have identical values in their non-timestamp
columns. A coalescing algorithm could be applied to reduce the number of rows
by merging the periods of validity of these value-equivalent rows. Coalescing is
currently not implemented in the prototype and would require a special database
procedure. Coalescing applied to the table above would result in Table 8.2.
### 8.3 Outlook

Considering the issues discussed in the previous section, extending and improving the prototype in order to make it more useful and applicable is reasonable. The issue that had not been mentioned so far is support for transaction time. If this could be added to the existing implementation at some point, the result would be a real bi-temporal database.
Appendix A

Database Procedures

All procedures that were written in the context of the history extension can be found in this chapter.

CREATE OR REPLACE FUNCTION transformdatetime(timestamp with time zone)
RETURNS tm_coordinate AS
$BODY$
DECLARE
  v_datetime_in alias for $1;
  v_datetime timestamp with time zone;
  v_origin timestamp with time zone;
  v_interval tm_interval;
  v_days int8;
  v_hours int8;
  v_min int8;
  v_sec int8;
  v_coordinate tm_coordinate;
BEGIN
  IF (v_datetime_in IS NULL) THEN
    RAISE EXCEPTION 'The provided timestamp must not be null!';
  END IF;

  SELECT origin FROM tm_coordinateSystem INTO

END;
$BODY$
END ;
v._origin WHERE tcsid = (SELECT value::int4 FROM tm_config WHERE parameter = 'tcsid');

IF NOT FOUND THEN
    RAISE EXCEPTION 'No time coordinate system provided; check tables <tm_coordinatesystem> and <tm_config>!';
END IF;

SELECT interval FROM tm_coordinateSystem INTO v.interval WHERE tcsid = (SELECT value::int4 FROM tm_config WHERE parameter = 'tcsid');

IF NOT FOUND THEN
    RAISE EXCEPTION 'No time coordinate system provided; check tables <tm_coordinatesystem> and <tm_config>!';
END IF;

/* remove milliseconds since 'second' is the smallest interval / chronon / granule */
v.datetime := date_trunc('second', v.datetime_in);

v.days := extract(day from (v.datetime - v.origin));

IF (v.interval = 'day') THEN
    v.coordinate := ROW(null, v.days);
    return v.coordinate;
END IF;

v.hours := extract(hour from (v.datetime - v.origin));

IF (v.interval = 'hour') THEN
v_coordinate := ROW(null, v_days*24 +
  v_hours);
return v_coordinate;
END IF;

v_min := extract (minute from (v_datetime -
  v_origin));

IF (v_interval = 'minute') THEN
  v_coordinate := ROW(null, (v_days*24*60) +
    (v_hours*60) + v_min);
  return v_coordinate;
END IF;

v_sec := extract (second from (v_datetime -
  v_origin));

v_coordinate := ROW(null, (v_days*24*60*60) +
  (v_hours*60*60) + (v_min*60) + v_sec);
return v_coordinate;
END; $BODY$

Listing A.1: transformDateTime(timestamp with time zone):tm_coordinate

CREATE OR REPLACE FUNCTION transformCoord(tm_coordinate)
  RETURNS timestamp with time zone AS
$BODY$
DECLARE
v_coordinate alias for $1;
v_origin timestamp with time zone;
v_interval tm_interval;
v_datetime timestamp with time zone;
v_hours int8;
v_minutes int8;
v_seconds int8;
BEGIN
  IF (v_coordinate is null) THEN
    RAISE EXCEPTION 'Provide a valid time coordinate!';
  END IF;

  SELECT origin FROM tm_coordinateSystem INTO v_origin
  WHERE tcsid = (SELECT value::int4 FROM tm_config
  WHERE parameter = 'tcsid');

  IF NOT FOUND THEN
    RAISE EXCEPTION 'No time coordinate system provided; check tables <
    tm_coordinatesystem> and <tm_config>!';
  END IF;

  SELECT interval FROM tm_coordinateSystem INTO v_interval
  WHERE tcsid = (SELECT value::int4 FROM tm_config
  WHERE parameter = 'tcsid');

  IF NOT FOUND THEN
    RAISE EXCEPTION 'No time coordinate system provided; check tables <
    tm_coordinatesystem> and <tm_config>!';
  END IF;

  -- Intervals are constructed from hours, minutes and
  -- seconds only. When using 'days' a wrong result
  -- might occur because
  -- across daylight saving time changes, an interval
  -- 1 day does not necessarily equal an
  -- interval 24 hours.

  CASE v_interval
    WHEN 'day' THEN

v_hours := v_coordinate.
coordinateValue * 24;
EXECUTE 'SELECT || quote_literal(v_origin) || '::timestamp with time_zone + INTERVAL ' ||
quote_literal('PT' || v_hours::text || 'H') INTO v_datetime;

WHEN 'hour' THEN
v_hours := v_coordinate.
coordinateValue;
EXECUTE 'SELECT || quote_literal(v_origin) || '::timestamp with time_zone + INTERVAL ' ||
quote_literal('PT' || v_hours::text || 'H') INTO v_datetime;

WHEN 'minute' THEN
v_hours := v_coordinate.
coordinateValue / 60;
v_minutes := v_coordinate.
coordinateValue % 60;
EXECUTE 'SELECT || quote_literal(v_origin) || '::timestamp with time_zone + INTERVAL ' ||
quote_literal('PT' || v_hours::text || 'H' || v_minutes::text || 'M') INTO v_datetime;

ELSE — second
v_hours := v_coordinate.
coordinateValue / (60*60);
v_minutes := v_coordinate.
coordinateValue % (60*60) / 60;
v_seconds := v_coordinate.
coordinateValue % 60;
EXECUTE 'SELECT || quote_literal(v_origin) || '::timestamp with time_zone + INTERVAL ' ||
quote_literal('PT' || v_hours::text || 'H' || v_minutes::text || 'M' || v_seconds::text || 'S') INTO v_datetime;
quote_literal('PT'||v_hours::text||'H'||v_minutes::text||'M'||v_seconds::text||'S') INTO v_datetime;

END CASE;

--RAISE NOTICE 'Result: %', quote_literal('PT'||v_hours::text||'H'||v_minutes::text||'M'||v_seconds::text||'S');

IF (v_datetime >= '9999-12-31 00:00:00+00') THEN
   v_datetime := 'infinity';
END IF;

return v_datetime;

END; $BODY$
LANGUAGE 'plpgsql' VOLATILE
COST 100;

Listing A.2: transformCoord(tm_coordinate):timestamp with time zone

CREATE OR REPLACE FUNCTION lineFromTimeHelper (timestamp with time zone, timestamp with time zone) RETURNS geometry AS $BODY$
DECLARE
   v_from alias for $1;
   v_till alias for $2;
   v_geom_text text;
   v_coordinate tm_coordinate;
BEGIN

   v_coordinate := transformDateTime(v_from);
   v_geom_text := 'LINESTRING('||v_coordinate.
      coordinateValue::text||'0,'||
   v_coordinate := transformDateTime(v_till);

END; $BODY$
v_geom_text := v_geom_text || v_coordinate.
    coordinateValue::text || '0');

return GeomFromText(v_geom_text, -1);

END; $BODY$
LANGUAGE 'plpgsql' VOLATILE
COST 100;

Listing A.3: lineFromTimeHelper(timestamp with tz, timestamp with tz)

CREATE OR REPLACE FUNCTION setPA (timestamp with time zone default null, timestamp with time zone default null)
RETURNS void AS
$BODY$
DECLARE
    v_from_in alias for $1;
    v_till_in alias for $2;
    v_from timestamp with time zone;
    v_till timestamp with time zone;
    v_count int4;
BEGIN

    v_from := v_from_in;
    v_till := v_till_in;

    IF (v_from >= '9999-12-31 00:00:00+00') THEN
        RAISE EXCEPTION 'validFrom has to be earlier than <9999-12-31 00:00:00+00>!';
    END IF;

    IF (v_from >= v_till) THEN
        RAISE EXCEPTION 'validTill has to be greater than validFrom!';
    END IF;

$BODY$
IF (v_till >= '9999-12-31 00:00:00+00' AND v_till <> 'infinity') THEN
    v_till := 'infinity';
    RAISE NOTICE 'validTill has been set to < infinity >!';
END IF;

SELECT COUNT(*) FROM tm_validtime INTO v_count
    WHERE userName = user;

IF (v_count = 0) THEN
    INSERT INTO tm_validtime VALUES(user, v_from, v_till);
ELSE
    UPDATE tm_validtime SET validFrom = v_from, validTill = v_till WHERE userName = user;
END IF;

END; $BODY$
LANGUAGE 'plpgsql' VOLATILE
COST 100;

Listing A.4: setPA(timestamp with tz, timestamp with tz)

CREATE OR REPLACE FUNCTION paFrom()
    RETURNS timestamp with time zone AS
$BODY$
BEGIN

    return getValidTimestamp(1);

END; $BODY$
LANGUAGE 'plpgsql' VOLATILE
COST 100;

Listing A.5: paFrom():timestamp with time zone
CREATE OR REPLACE FUNCTION paTill()
    RETURNS timestamp with time zone AS
$BODY$
BEGIN

    return getValidTimestamp(2);

END; $BODY$

LANGUAGE 'plpgsql' VOLATILE
COST 100;

Listing A.6: paTill():timestamp with time zone

CREATE OR REPLACE FUNCTION lineFromPA()
    RETURNS geometry AS
$BODY$
DECLARE

    v_from timestamp with time zone;
    v_till timestamp with time zone;

BEGIN

    v_from := paFrom();
    v_till := paTill();

    IF (v_till = 'infinity') THEN
        v_till := '9999-12-31 00:00:00+00';
    END IF;

    return lineFromTimeHelper(v_from, v_till);

END; $BODY$

LANGUAGE 'plpgsql' VOLATILE
COST 100;

Listing A.7: lineFromPA():geometry
CREATE OR REPLACE FUNCTION lineFromTime(timestamp with time zone, timestamp with time zone)
RETURNS geometry AS $BODY$
DECLARE
  v_from_in alias for $1;
  v_till_in alias for $2;
  v_from timestamp with time zone;
  v_till timestamp with time zone;
BEGIN

  v_from := v_from_in;
  v_till := v_till_in;

  IF (v_from IS NULL AND v_till IS NOT NULL) THEN
    RAISE EXCEPTION 'validFrom must not be null !';
  END IF;

  IF (v_from >= '9999-12-31 00:00:00+00') THEN
    RAISE EXCEPTION 'validFrom has to be smaller than <9999-12-31 00:00:00+00> !';
  END IF;

  IF (v_from >= v_till) THEN
    RAISE EXCEPTION 'validTill has to be greater than validFrom !';
  END IF;

  IF (v_from IS NULL) THEN
    v_from := paFrom();
  END IF;

  IF (v_till IS NULL) THEN
    v_till = paTill();
    IF (v_till = 'infinity') THEN
      END IF;

  END IF;

$BODY$
v_till := '9999-12-31 00:00:00+00';
END IF;
ELSE
IF (v_till = 'infinity' OR v_till >= '9999-12-31 00:00:00+00') THEN
v_till := '9999-12-31 00:00:00+00';
END IF;
END IF;

return lineFromTimeHelper(v_from, v_till);

Listing A.8: lineFromTime(timestamp with tz, timestamp with tz):geometry

CREATE OR REPLACE FUNCTION validFrom(geometry)
RETURNS timestamp with time zone AS
$BODY$
DECLARE
v_validtime alias for $1;

v_coordinate tm_coordinate;
BEGIN

v_coordinate := ROW(null, st_x(st_startpoint(

v_validtime))::bigint);
return transformcoord(v_coordinate);

END; $BODY$

LANGUAGE 'plpgsql' VOLATILE
COST 100;

Listing A.9: validFrom(geometry):timestamp with time zone

CREATE OR REPLACE FUNCTION validTill(geometry)
RETURNS timestamp with time zone AS
$BODY$
DECLARE
  v_validtime alias for $1;
  v_coordinate tm_coordinate;
BEGIN
  v_coordinate := ROW(null, st_x(st_endpoint(
    v_validtime))::bigint);
  return transformcoord(v_coordinate);
END;$BODY$
LANGUAGE 'plpgsql' VOLATILE
COST 100;

Listing A.10: validTill(geometry):timestamp with time zone

CREATE OR REPLACE FUNCTION getValidTimestamp(int4)
RETURNS timestamp with time zone AS
$BODY$
DECLARE
  v_type alias for $1;
  v_timestamp timestamp with time zone := null;
  v_interval tm_interval;
BEGIN

  SELECT interval FROM tm_coordinateSystem INTO v_interval
    WHERE tcsid = (SELECT value::int4 FROM tm_config WHERE parameter = 'tcsid');

  IF NOT FOUND THEN
    RAISE EXCEPTION 'No time coordinate system provided; check tables <tm_coordindatesystem> and <tm_config>!';
  END IF;

  IF (v_type = 1) THEN — validFrom
```sql
SELECT validFrom FROM tm_validtime INTO v_timestamp WHERE userName = user;
IF (v_timestamp IS NULL) THEN
    v_timestamp := CURRENT_TIMESTAMP;
END IF;
ELSE —— validTill

SELECT validTill FROM tm_validtime INTO v_timestamp WHERE userName = user;
IF (v_timestamp IS NULL) THEN
    v_timestamp := 'infinity';
END IF;
END IF;

return v_timestamp;

END; $BODY$

LANGUAGE 'plpgsql' VOLATILE
COST 100;
```

Listing A.11: getValidTimestamp(int4):timestamp with time zone
Bibliography


URL: http://infolab.usc.edu/csci599/Fall2001/paper/glossary.pdf


URL: http://www.opengeospatial.org/standards/sfa


URL: http://download.oracle.com/docs/cd/E11882_01/appdev.112/e11826.pdf


URL: http://postgis.refractions.net/download/postgis-1.4.0.pdf


URL: http://www.spatial.maine.edu/~worboys/mywebpapers/comp%20journal%201994.pdf