TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Informaatikainstituut

Infosüsteemide õppetool

Ajaandmete haldamise üks võimalik käsitlus PostgreSQL andmebaasisüsteemi näitel

Magistritöö

Üliõpilane: Sander Laasik

Üliõpilaskood: 132460IAPM

Juhendaja: dotsent Erki Eessaar

Tallinn
2015

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

(*kuupäev*) (*allkiri*)

Annotatsioon

Ajaandmete haldamise üks võimalik käsitlus PostgreSQL andmebaasisüsteemi näitel.

Temporaalsed on sellised andmed, millele on lisatud ajaline mõõde – see tähendab, et iga faktiga käib alati kaasas ka ajavahemik, mille jooksul antud fakt kehtib. Andmetele ajalise mõõtme andmine suurendab olulisel määral andmete analüüsimise võimalusi, lihtsustades seeläbi äriliste otsuste tegemisi. Samas lisab see arendajatele ka palju probleeme, sest erinevate vahemike võrdlemine, tabelipõhiste kitsenduste loomine ja tabelitevahelise andmekvaliteedi kontrollimine võib kujuneda märksa keerulisemaks kui esmapilgul tundub.

C.J. Date, Hugh Darwen ja Nikos A. Lorenzos kirjeldavad oma raamatus *„Temporal Data and The Relational Model: A Detailed Investigation into the Application of Interval and Relation Theory to the Problem of Temporal Database Management“* (Date et al 2002) (edaspidi viidatud kui **raamat**) üht võimalikku aja- ehk temporaalandmete käsitlust relatsioonilistes andmebaasides. **Raamatus** pakutakse välja rida põhimõtteid, mida järgides peaks temporaalandmete hoidmine, lisamine ning nende kasutamine muutuma selgemaks ning lihtsamaks.

Kuigi SQL on loodud relatsioonilisele andmemudelile põhinedes, on relatsioonilise andmemudeli ja SQL-i aluseks oleva andmemudeli vahel erinevusi. Samuti on praegustel populaarsetel SQL-andmebaasisüsteemidel (DBMS-id) relatsioonilise andmudeliga võrreldes üsna palju lisakitsendusi ning puudusi. Seega pole kindel, kas **raamatus** toodud käsitluse kasutamine SQL-andmebaasides on üldse võimalik.

Käesoleva töö eesmärk on PostgreSQL 9.3 andmebaasisüsteemi näitel proovida realiseerida selle käsitluse kasutuselevõtmiseks vajalik funktsionaalsus. Selle eesmärgi saavutamiseks on kõigepealt vajalik selle käsitlusega tutvuda ja seda ka magistritöö lugejatele tutvustada. Käsitluse realiseerimine ühes SQL-andmebaasisüsteemis annab kindlust (kuid muidugi mitte garantii), et seda on põhimõtteliselt võimalik realiseerida ka teistes SQL andmebaasisüsteemides.

Käesoleva töö tulemusena on täidetud järgnevad ülesanded:

1. Kirjeldada **raamatus** pakutud lahendust. See on refereering, püüdmaks võimalikult hästi

edasi anda autorite ideid ning põhimõtteid.

2. Kasutades PostgreSQL 9.3 andmebaasisüsteemi, realiseerida võimalikult suur hulk **raamatus** väljapakutud operaatoreid ja disainipõhimõtteid.
3. Kasutades loodud funktsionaalsust, disainida ning realiseerida näidisandmebaas, mis peaks neid põhimõtteid võimalikult lihtsasti kirjeldama.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 88 leheküljel, 3 peatükki, 2 joonist, 52 tabelit.

Abstract

One Possible Approach of Temporal Data Management Using PostgreSQL Database Management System as an Example

Data is called temporal when it has the dimension of time – every fact has a range of time indicating the period when this fact was considered to be true. Making data temporal drastically increases the possibilities how this data can be analysed. It can make business decisions much easier. On the other hand, many problems arise for developers because comparing different data ranges, creating constraints on tables, and checking data quality between tables becomes much more complex than it might seem at first.

C.J. Date, Hugh Darwen, and Nikos A. Lorenzos describe in their book *„Temporal Data and The Relational Model: A Detailed Investigation into the Application of Interval and Relation Theory to the Problem of Temporal Database Management“* (Date et al 2002) (Referenced as **the book** from this point forward) one possible approach for the management of temporal data in relational databases. **The book** provides many principles that should make storing, adding, and using the temporal data easier and more transparent.

Although SQL has been created based on the relational data model there are differences in the relational data model and the underlying data model of SQL. In addition, current popular SQL database management systems (DBMSs) have quite a lot of additional restrictions and deficiencies compared to the relational data model. Thus, it is not clear as to whether one can use the approach from the book in SQL databases.

The goal of the thesis is to implement in the example of PostgreSQL 9.3 DBMS the functionality that is needed to support the approach offered in the book. To achieve the goal, one firstly has to understand the approach and introduce it to the readers of the thesis. Implementation of the approach in case of one SQL DBMS gives confidence (but of course not a guarantee) that one could implement the approach in other SQL DBMSs.

The following main tasks are accomplished as a result of this thesis.

1. We describe the solution provided in **the book**. This is a summary of the ideas and

principles of the authors.

2. We use PostgreSQL 9.3 DBMS to implement as much different operators and design principles from **the book** as possible.

3. We use the new functionality to create an example database that should explain the solution as well as possible.

The thesis is in English and contains 88 pages of text, 3 chapters, 2 figures and 52 tables.

Definitions and abbreviations

DBMS	**Database Management System – "**a set of programs that enables users to store, modify and extract data from a database" (Webopedia, 2015). It is also the gatekeeper of the database meaning that all the usage of the management of the structure, behaviour, and data content of the database goes through it.
SQL	**Structured Query Language –** SQL is a standardized database language for requesting information from a database, assigning new values to the database, transaction and privilege control as well as managing data types, data structures, constraints, operators, and their surrounding ecosystem of other types of database objects (Webopedia, 2015). It is a domain-specific computer language in the domain of databases. SQL has been created based on (is an implementation of) the relational data model but does not follow it completely.
DML	**Data Manipulation Language –** a subset of a database language meant for inserting, retrieving, and changing data in the database. Common statement types in SQL for doing it are INSERT, UPDATE and DELETE
DDL	**Data Definition Language –** a subset of a database language meant for managing the structure and behaviour of data types, data structures, constraints, operators, and their surrounding ecosystem of other types of database objects. Common statement types in SQL for changing the structure of the database are CREATE, ALTER and DROP.
SQL DBMS	A database management system where one can use SQL for the data management and that uses the underlying data model of SQL to organize data in the database

List of figures

List of tables

Table of Contents

1. Introduction

1.1 The Background and the Problem

Temporal data has the time dimension meaning that every fact has a time period when it was considered to be true. This means that we can store historicized data in the database and do various analyses on it that can lead to making better business decisions, cutting costs, and earning higher profit for the company. For many decades, storing history data was not very easy but as the processors have become faster and disk storage has become cheaper, more and more companies invest into building their data warehouses. A data warehouse provides the company a single point of truth – data from different source (mainly operational) databases is loaded and transformed into the data warehouse structures, various metrics are calculated, and reports are put together. All of this has a business value only in case the data is trustworthy and reliable meaning that there are no history overlaps, gaps, or redundancy in the data. Data quality checks, constraints, and transparent development process contribute towards achieving this.

The literature recognizes two separate dimensions of time:

1. *Valid time* – the period of time when some fact was considered to be true. Thus, valid time reflects the *real world* understanding.
2. *Transaction time* – the period of time when the database showed that some fact was considered to be true. Transaction time can be thought of as a timestamp that is used for logging the states of the database information over time.

A table that contains information about both of these dimensions is called a *bi-temporal* table.

In **the book** (Date *et* al, 2002), an approach of handling temporal data is described. One major difference compared to other approaches (though not all) is that it uses a special datatype – interval – to store the periods. The interval value consists of values of the same point type and has start and end points defined. For example, a date interval could consist of days (represented by date values) between June 1, 2015 and June 5, 2015. Additionally, **the book** provides a list of operators, design principles, and system level actions that should make

common temporal data related actions much easier, clearer, and quite fool proof. The ideas are described to be used in a fully relational database with operators and examples written in the database language called Tutorial D. SQL is a database language (a domain-specific programming language) that is based on the relational model but does not follow it completely. Thus, the main goal of this thesis is to try to implement this approach in an SQL DBMS (more specifically PostgreSQL 9.3) as much as possible, and find out the obstacles and difficulties that might come across when doing it.

1.2 The Tasks of this Thesis

The following tasks are accomplished as the result of this thesis:

1. We explain the solution provided in **the book**. Though the ideas in the book are meant for using in a relational database, we explain them right away by using the concepts of SQL.

2. We use PostgreSQL 9.3 DBMS, to implement as much different operators and system level actions from **the book** as possible. Only support for data type *DATE* will be implemented as it is sufficient for creating an example of a fully temporal database. Adding support for other data types is seen as one of the possibilities to continue the work of this thesis.

3. We use the new functionality, to create an example database that should illustrate the solution as well as possible.

1.3 Methodology

The methodology we are using is *design science research* (Hevner et al, 2004). In the design-science paradigm, knowledge and understanding of a problem domain and its solution are achieved in the building and application of the designed artefact.

1. We shortly point to the other approaches of managing temporal data in relational/SQL databases. Comparing these with the ideas and principles of **the book** is a possible subject of another thesis. However, continuing work in this field should show that it is an important and emerging field of study.

2. We summarize the approach that is described in **the book**. The approach requires various operators that are needed for working with temporal data, the design pattern that is based on the sixth normal form, and system level actions that are triggered when DML or DDL statements are executed. We use examples to describe various operators.

3. We have selected PostgreSQL as the DBMS where to implement the approach provided in **the book** because PostgreSQL is popular, accessible to many users (because it is free), and has also good built-in extensibility mechanism. In addition, the advanced features of PostgreSQL make it suitable for data warehousing that could benefit from the approach offered in the book. In May 2015 PostgreSQL is the fifth most popular DBMS according the DB-Engines Ranking and fourth most popular SQL DBMS (DB-Engines Ranking, 2015). Implementation of the solution does not mean modification of the source code (although the source of PostgreSQL is public) but the use of the extensibility mechanism of the DBMS to create new triggers, functions, and operators. Implementation of the approach in case of one SQL DBMS gives confidence (but of course not a guarantee) that one could implement the approach in other SQL DBMSs.

4. To implement the functions, operators, and triggers, we firstly analyse the support for temporal data in PostgreSQL 9.3 (PostgreSQL, 2015). Based on that, additional PL/pgSQL functions and triggers are created to implement all the ideas described in the first task. Although PostgreSQL supports many procedural languages and allows developers to define new procedural languages, we use PL/pgSQL to implement the functions. We do it because it is the best known of the PostgreSQL procedural languages, it is installed to each database by default, and there are the most supporting materials for this language.

5. We will create a simple database by using PostgreSQL 9.3 to better describe the principles of the approach and conduct initial performance measurements. In addition, we provide example statements for demonstrating that everything is working as expected.

1.4 Overview of the Work

In the first chapter it is explained how a fully temporal relational database should look like design wise, which data types and operators could be used, and how DML and DDL statements should work there.

In the second chapter an overview of already existing temporal support in PostgreSQL 9.3 is given. The list of operators from **the book** is provided and it is compared to the PostgreSQL built-in functions and operators. Based on that, a list of user defined PL/pgSQL functions is given that are implemented in the context of this thesis. We also describe different types of base table (table) designs that facilitate temporal support. Finally, an example database is created to better describe the functionality of trigger procedures and functions needed for the temporal support. In this database, the needed temporal constraints are in place and all the facts are stored in the tables that are in sixth normal form. We will use the example temporal database to measure the performance of database queries and data manipulation statements.

1.5 Some Other Approaches of Maintaining Temporal Data in SQL Databases

1.5.1 SQL:2011 Standard

The SQL:2011 standard introduced a set of temporal features (Kulkarni et al, 2012). The *transaction time* dimension is called *SYSTEM_TIME* in SQL:2011 and the *valid time* is referred as *application-time period* that can have any name the user specifies.

SQL:2011 does not introduce a new data type for storing periods as some of the DBMS vendors do. Instead, it introduces *period definitions* as metadata to tables. It is a named table component that identifies a pair of columns that act as start and end points of the period. The period uses the closed-open approach, meaning that the start point of the period is inclusive and the end point is exclusive. The period member columns must be of data type *DATE* or *TIMESTAMP* and they both must be of the same data type.

One problem with this approach is that the *valid time* period is not a column but a metadata of the table. It cannot be used in the SELECT clause, thus the period cannot be used outside of a subquery that references the history table (Darwen, 2013).

SQL:2011 also specifies a PORTION syntax that can be used for specifying the period when some changes should be applied on. This can be used only in UPDATE and DELETE statements. As a result, the UPDATE and DELETE statements whose search condition finds one or more rows result with insertion of new rows to the same table. Each row in the table represents a proposition that was true during the given application-time period.

The WITHOUT OVERLAPS syntax can be used when creating primary key constraints that involve application-time periods. This means that the table cannot contain two or more primary key values where application-time period overlaps. Furthermore, application-period can be used for creating FOREIGN KEY constraints to make sure that each foreign key value that is active during some period has the corresponding value active in the referenced table. For example, there can be no contracts for customer 1 on day *D1* if the customer 1 is only active since the day *D4*.

The SQL:2011 standard also introduces a set of operators that provide a shorthand for better using the period. This includes the operators CONTAINS, OVERLAPS, EQUALS, PRECEDES, SUCCEEDS, IMMEDIATELY PRECEDES, and IMMEDIATELY SUCCEEDS.

When the WITH SYSTEM VERSIONING syntax is used while creating a table, the transaction time dimension will be handled automatically by the system. This means that the user cannot change the values of the member columns of SYSTEM_TIME and each time an INSERT, UPDATE or DELETE statement is executed on a system versioned table, the system will automatically make the needed changes in the database. As a result, UPDATE statements whose search condition finds one or more rows result with insertion of new rows to the same table to record historic row versions. DELETE statements whose search condition finds one or more rows result with updating instead of deleting the rows to indicate the end time of the existence of these rows as current rows. The table contains both historic and current row versions but users can only update or delete current row versions.

1.5.2 Anchor Modelling

Anchor Modelling is an Open Source database modelling technique that supports handling temporal data in an SQL/relational database in a way that a large change outside the model will result as a small change within the model (Anchor Modelling, 2015). It is based on the sixth normal form meaning that each fact could have a time dimension added to it, making it independent from other facts. For example, we can track the history of changes in customer's height and weight over time when we have these values stored in tables that are in the sixth normal form. Tables in the sixth normal form cannot be decomposed in a nonloss manner to tables that have fewer columns than the original (Date, 2006).

Anchor Modelling has four basic concepts that are used for modelling (Rönnbäck et al, 2010):

- *Anchor* that is used to identify entities with the same type (for example Customer).
- *Attribute* that corresponds to a named property of sets of entities with the same type (for example, First Name of customers).
- *Tie* that is used for modelling types of relationships between Anchors (for example a relationship type between contract and its owner).
- *Knot* that is used for modelling classifiers (for example contract status types).

One can historicize attributes and ties to facilitate storing of their corresponding data changes over time. Anchor Modelling offers both web-based modelling software for creating anchor models as well as generators for generating corresponding base tables, views, and functions. Views and functions offer more "traditional" denormalized view to the data. There are already generators for MS SQL Server, Oracle, and PostgreSQL (Saal, 2015).

Anchor Modelling will result in many tables as opposed to fewer tables with many columns. The benefit of this approach is that the system is handling different changes better as every change is an independent extension that keeps the current applications unaffected. The biggest threat is that joining all of these separate tables can result in bad performance, though it is said that modern DBMSs can cope with it using table elimination optimization while executing the queries.

Unfortunately, we could not use Anchor Modelling as a basis of our implementation of the ideas from **the book** because it uses only the start point for keeping the history of a fact. On the other hand, the cornerstone of the ideas in the book is to have interval data types in place to store the validity period of a fact as a single value. Analysing the possibilities of making Anchor Modelling compatible with interval data types would be an interesting subject to some other thesis, though.

1.5.3 Oracle Workspace Manager

Oracle Workspace Manager is a feature of Oracle Database DBMS that enables application developers and DBAs to manage current, proposed and historical versions of data in the same database. It provides workspaces as a virtual environment to isolate a collection of changes to production data, keep a history of changes to data and create multiple data scenarios for "what if" analysis (Oracle, 2015).

If versioning is turned on for a table, then Workspace manager drops the base table and creates a set of tables, views and triggers that all contribute to keeping the versioning up and running. This involves a table with suffix *_AUX* for handling workspace conflicts and table with suffix *_LT* for storing the last available version. Additionally, a set of system views (with a read-only access for the users) are created for handling metadata about tables, workspaces, saving points, locks, users, privileges and conflicts. Moreover, there are views with the following suffixes created – for every versioned table:

- *_CONF* – view for the conflicts,
- *_DIFF* – view for the version differences,
- *_HIST* – history view (if storing of history is enabled),
- *_LOCK* – view for the workspace locks,
- *_MW* – multi-workspace view.

A set of *INSTEAD OF* triggers are also created for deleting, inserting and updating data in the non-temporal tables. This is done because the end user still sees the versioned table with the same structure as it was created (Potter, 2013).

An important difference from the approaches that use tables in the sixth normal form is that if a row is updated (values in some fields are replaced with some new values), then the old version of the entire row (including fields that were not modified) is kept by the system. If one uses tables in the sixth normal form, and modifies some attribute value, then the system does not have to duplicate unchanged data to keep the old version. Similarity with the Anchor Modelling approach is that internal "machinations" of the system to keep old versions of facts are hidden behind virtual database layer that is an interface consisting of functions and views. Database user can/should access the elements of this layer.

One can think about the entire versioning approach of the Workspace Manager as a possible implementation of system versioning specified in the SQL:2011 standard.

1.5.4 Teradata Temporal Table Support

Teradata is a DBMS that is mainly developed for data warehouses. Thus, Teradata 13.10 has a vast support for temporal concepts (Teradata, 2010). Teradata 13.10 supports both *valid time* and *transaction time* concepts, where the valid time shows us the period when some fact was considered to be true and the transaction time shows the time when some row was physically marked as active in the database. Teradata provides a special data type *PERIOD* for storing the time periods that can consist of values of data types *DATE*, *TIME* and *TIMESTAMP* for valid time columns and *TIMESTAMP* for transaction time columns. The begin point of a period is always considered inclusive and the end point is considered exclusive. Furthermore, there are special variables to mark the end point of periods that are still active – *UNTIL_CLOSED* is used for transaction time and *UNTIL_CHANGED* is used for valid time columns. The data type of *UNTIL_CLOSED* is *TIMESTAMP(6) WITH TIME ZONE* and its value is equal to *TIMESTAMP '9999-12-31 23:59:59.999999+00:00'*. *UNTIL_CHANGED* can be used only for periods that consist of values of data types *DATE* and *TIMESTAMP* and has the value equal to *DATE '9999-12-31'* or *TIMESTAMP '9999-12-31 23:59:59.999999+00:00'* correspondingly.

Teradata 13.10 provides a set of shorthand commands that can be used for retrieving currently active information, information that was active between a specified period, or information that has been active as of some point in the history. Furthermore, when a column is specified as a *valid time* column and the user executes a statement that changes the data, then automatic statements are executed additionally that are needed for maintaining the history correctly.

It is difficult to cover all the features that Teradata provides for temporal support, thus we can conclude this section by stating that it is one of the best DMBS currently available for maintaining a truly temporal database. Unfortunately, all this comes with a price tag since Teradata is not an open source system nor it is free. They have stated, though, that their products are highly price-competitive compared to their biggest competitors (Teradata Magazine, 2011)

One problem we have with Teradata is that it uses the design of *historical tables only* (to be discussed later) for temporal tables. It brings along the problem of using a variable (*UNTIL_CHANGED*) to store the end point of an active fact – this, unfortunately, as we will discuss later, is not a wise approach according to the authors of **the book**.

2. Principles from the Book *"Temporal Data and the Relational Model"*

2.1 Introduction

In this section, a summary of the ideas and principles from **the book** are provided. The order of introducing the topics mostly remains the same as in **the book**. First, a small overview of the temporal approach is provided. Next, the concepts *point* and *interval types*, which play an important role in this approach, are introduced. After that, all the needed operators on these types are listed and finally the database design and needed system level actions are described.

Note that the authors of **the book** have meant the ideas to be used in a database that fully supports the relational model. As the aim of our work is to implement them in an SQL DBMS (that does not completely support the relational model), we will explain the ideas using SQL counterparts for the terms instead. For example, instead of terms *relation variable* or *relvar, virtual relvar* and *attribute* we use *table, view* and *column*. Please note that this is not only the question of terminology but these terms represent different concepts that have considerable differences in details. We will point to the differences where it is needed.

One can very loosely divide databases into the following groups based on the pattern of their usage:

1. **Operational (non-temporal) databases** that store only the current data, meaning that if something is deleted or updated in the database, then no history is kept.
2. **Temporal databases** that store the historical data instead of or in addition to the current data. For example, data warehouses, which have become widespread in the recent decades.
3. **Hybrid** of the previous two where in case of some types of facts the history is kept and in case of some, it is not.

If data in general can be regarded as encoded representation of propositions, then temporal data can be regarded as propositions with timestamps. For example, an operational database

can answer a question *like "How big is the current debt of customer J. Smith?"*, then temporal database can answer questions like *"What date did customer J. Smith's first overdue start on?"* or *"How big was customer J. Smith's debt on 24th April 2013?"*. These were all examples of the *valid time* that shows when some fact in real world was considered to be true as opposed to the *transaction time* that shows when the database showed that some fact was considered to be true. Though the authors of the book also describe the *transaction time* concept quite thoroughly, we will leave it out of the scope of this thesis because we are more interested in the ideas about storing the *valid time* information.

Temporal data is usually represented by using *Start* and *End* date columns for showing the duration of the period when some fact was considered to be true. Such approach makes it very difficult to define different data quality and integrity constraints, and write complex database queries. Many examples are given in the book to illustrate the problem. As an example of using *Start* and *End* dates, Table 1 shows the table *CUSTOMER_STATUS_HIST* that stores status history for a customer with *Customer_Id* 1 (*Status_Start_Date* and *Status_End_Date* are both inclusive)

Table 1

CUSTOMER_STATUS_HIST			
Customer_Id	Status_Type_Code	Status_Start_Date	Status_End_Date
1	1	2014-06-28	2014-07-01
1	2	2014-07-02	NULL

These records show that this customer had status 1 from 2014-06-28 to 2014-07-01. Starting from 2014-07-02 the status has been 2. Usually a *NULL* (often incorrectly called "NULL value") is used to represent the current state of the fact but this is not how it should be done according the authors of **the book**. The relational model does not support NULLs (it is a difference of the relational model and the underlying model of SQL) and requires that in each field of each row there must be exactly one value that belongs to the type of the corresponding column. The proposed approach is to have two different tables for storing the temporal data in a fully temporal way.

The first table is for storing fully historical data with *Start* and *End* dates both earlier than *today*. Instead of using *Start* and *End* dates, an *interval* data type is introduced to store the whole period in one field. The historical period from our example (the first row in *Table 1*) can be represented in four ways if using an *interval* type:

- [2014-06-28:2014-07-01] – Period start and end dates are both inclusive
- [2014-06-28:2014-07-02) – Period start date is inclusive and end date is exclusive
- (2014-06-27:2014-07-01] – Period start date is exclusive and end date is inclusive
- (2014-06-27:2014-07-02) – Period start and end dates are both exclusive

Thus, the fully historical table *CUSTOMER_STATUS_HIST* could look like shown in *Table 2*

Table 2

CUSTOMER_STATUS_HIST		
Customer_Id	Status_Type_Code	Status_During
1	1	[2014-06-28:2014-07-01]

The second table is for storing only the current status of a customer with the *Since* column to show the date when current status was assigned to the customer. Thus, the second table *CUSTOMER_STATUS_CURRENT* could look like shown in *Table 3*.

Table 3

CUSTOMER_STATUS_CURRENT		
Customer_Id	Status_Type_Code	Status_Since
1	2	2014-07-02

We will come back to this approach in a more detailed manner, after we have explained the concept of the *interval* type, the *point* type, and the operations used on them.

2.2 Point Type, Interval Type, and Operations on Them

2.2.1 Point Type

Each data type is a named set of values. Point type is the data type of the values that the interval consists of. In the introduction of this chapter, the column *Status_During* contained interval values consisting of date values. Thus, the *point type* corresponding to *Status_During* was *DATE.* Since our aim is to provide a better way to store temporal data, then temporal data types are in the most interest for us. Intervals can consist of values with other kind of point types as well, though. For example, these could be integers or decimals.

In order to use a type *T* as a point type, it must fulfil all of the following requirements.

- A total ordering, according to which the operator ">" (greater than) is defined for every pair of values *v1* and *v2* of type *T*; if *v1* and *v2* are distinct, exactly one of the expressions "*v1 > v2*" and "*v2 > v1*" returns true and the other returns false.
- Nullary (with no arguments) "first" and "last" operators that return the smallest and the largest value of *T*, respectively, according to the aforementioned ordering.
- Monadic (with one argument) "next" and "prior" operators that return the successor and the predecessor, respectively, of any given value of type *T*, according to the aforementioned ordering. The "next" and "prior" operators are undefined if the given value of type *T* is in fact the "last" or "first" value, respectively, of that type.

Explicit operators *FIRST_T(), LAST_T(), NEXT_T(t T),* and *PRIOR_T(t T)* should be defined for each point type where *T* is the corresponding type name and *t* a value of type *T*. For example, in an Oracle database, FIRST_INTEGER() should return -2147483648 and NEXT_DATE(TO_DATE('2014-12-01', 'YYYY-MM-DD')) should return the date '2014-12-02'.

2.2.2 Interval Type

An interval value consists of the values of a corresponding point type. For each point type there should be a separate interval invocation selector *INTERVAL_T()* defined, where *T* again represents the type name of the point type. Thus, in our example, the definition of an interval

would be *INTERVAL_DATE([2014-06-28:2014-07-01])*. Selector is "an operator for selecting, or specifying, an arbitrary value of a given type" (Date, 2006).

Interval type has several very useful operators that make the creation of temporal data related constraints and complex queries much easier. We will now list all of them, using *i* to represent an interval value (consisting of points of type *T*) and *p* to represent a value of type *T*.

2.2.2.1 Single Interval Operators

The following operators are meant to provide shorthand for quickly getting the most needed point values of an interval and to check as to whether a point is contained in an interval.

2.2.2.1.1 BEGIN

BEGIN(i) returns the begin point of the interval. For example, *BEGIN ([2014-06-28:2014-07-01])* would return the date *2014-06-28*

2.2.2.1.2 END

END(i) returns the end point of the interval. For example, *END ([2014-06-28:2014-07-01])* would return the date *2014-07-01*

2.2.2.1.3 $p \in i$

$p \in i$ returns *TRUE* if and only if $p \geq BEGIN(i)\ AND\ p \leq END(i)$. For example, 2014-06-30 ∈ *[2014-06-28:2014-07-01]* would return *TRUE*, while *2014-07-03* ∈ *[2014-06-28:2014-07-01]* would return *FALSE*

2.2.2.1.4 PRE

PRE (i) returns *BEGIN (i) – 1* (what the "-1" means depends on the semantics of the point type; in this case it means "-1 day"). For example, *PRE ([2014-06-28:2014-07-01])* would return the date *2014-06-27*

2.2.2.1.5 POST

POST (i) returns *END (i) + 1* (what the "+1" means depends on the semantics of the point type; in this case it means "+1 day"). For example, POST ([2014-06-28:2014-07-01]) would return the date 2014-07-02

2.2.2.1.6 $i \ni p$ or CONTAINS(i, p)

$i \ni p$ returns *TRUE* if and only if $p \in i$ returns *TRUE*

2.2.2.2 Comparison operators

The following operators, called ***Allen's operators*** are meant for comparing two intervals that must be of the same interval type *INTERVAL_T*. For simplicity, *b1* and *e1* will represent begin and end values of interval value *i1* correspondingly. *b2* and *e2* will mean the same for interval value *i2*.

2.2.2.2.1 EQUALS

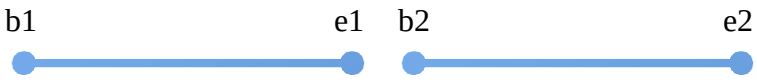
i1 EQUALS i2 or *i1=i2* returns *TRUE* if and only if *b1=b2* and *e1=e2* both return *TRUE*.

2.2.2.2.2 INCLUDES (⊇) and INCLUDED_IN (⊆)

$i1 \supseteq i2$ is *TRUE* if and only if $b1 \leq b2$ and $e1 \geq b2$ are both *TRUE*. $i2 \subseteq i1$ is *TRUE* if $i1 \supseteq i2$ is *TRUE*.

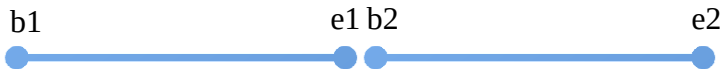
2.2.2.2.3 BEFORE and AFTER

i1 BEFORE i2 is *TRUE* if and only if *e1<b2* is *TRUE*. *i2 AFTER i1* is true if *i1 BEFORE i2* is *TRUE*



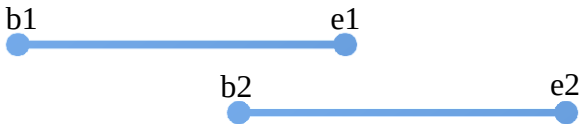
2.2.2.2.4 MEETS

i1MEETS i2 is *TRUE* if and only if *b1=e1+1* is *TRUE* or *b1=e2+1* is *TRUE* (it follows that *i2MEETS i1* is *TRUE* if *i1 MEETS i2* is *TRUE*)



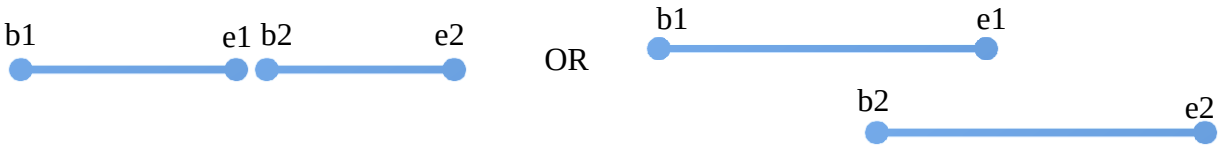
2.2.2.2.5 OVERLAPS

i1 OVERLAPS i2 is *TRUE* if and only if $b1 \leq e2$ and $b2 \leq e1$ are both *TRUE* (it follows that *i2 OVERLAPS i1* is *TRUE* if *i1 OVERLAPS i2* is *TRUE*).



2.2.2.2.6 MERGES

i1 MERGES i2 is *TRUE* if and only if *i1 OVERLAPS i2* is *TRUE* or *i1MEETS i2* is *TRUE* (it follows that *i2 MERGES i1* is *TRUE* if and only if *i1 MERGES i2* is *TRUE*).



2.2.2.2.7 BEGINS

i1 BEGINS i2 is *TRUE* if and only if $b1 = b2$ is *TRUE* and $e1 \leq e2$ is *TRUE*

2.2.2.2.8 ENDS

i1 ENDS i2 is *TRUE* if and only if $e1 = e2$ is *TRUE* and $b1 \geq b2$ is *TRUE*

2.2.2.3 Other operators

2.2.2.3.1 COUNT

COUNT(i) returns a count of the number of points in interval *i* (also called the *cardinality* or the *length* of the interval). For example, *COUNT ([2014-06-28:2014-07-01])* would return 4.

2.2.2.3.2 MAX

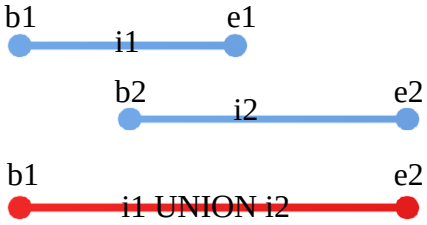
If *p1* and *p2* are values of point type *T*, then *MAX* (*p1*, *p2*) returns *p1*, if *p1≥p2* and *p2*, if *p2>p1*

2.2.2.3.3 MIN

If *p1* and *p2* are values of point type *T*, then *MIN* (*p1*, *p2*) returns *p1*, if *p1≤p2* and *p2*, if *p2<p1*

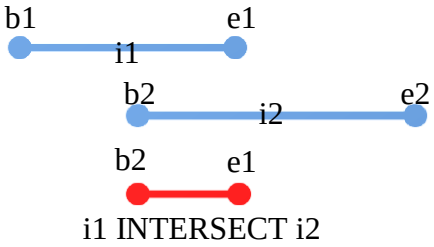
2.2.2.3.4 UNION

i1 UNION i2 returns an interval *[MIN(b1, b2):MAX(e1, e2)]* if *i1 MERGES i2* is *TRUE* and is otherwise undefined



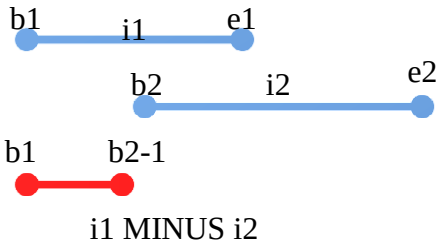
2.2.2.3.5 INTERSECT

i1 INTERSECT i2 returns an interval *[MAX(b1, b2):MIN(e1, e2)]* if *i1 OVERLAPS i2* is *TRUE* and is otherwise undefined



2.2.2.3.6 MINUS

i1 MINUS i2 returns an interval *[b1:MIN(b2-1,e1)]* if *b1<b2* and *e1≤e2* are both *TRUE*, *[MAX(e2+1, b1):e1]* if *b1≥b2* and *e1>e2* are both *TRUE*, and us otherwise undefined.



2.2.3 Summary of Point and Interval Types

In *Figure 1*, a class diagram for visualizing the point type and interval type is provided. Here, the PointType class is abstract, with the required four operators that must be provided. As said earlier, each concrete data type acting as a point type must have its own operators defined.

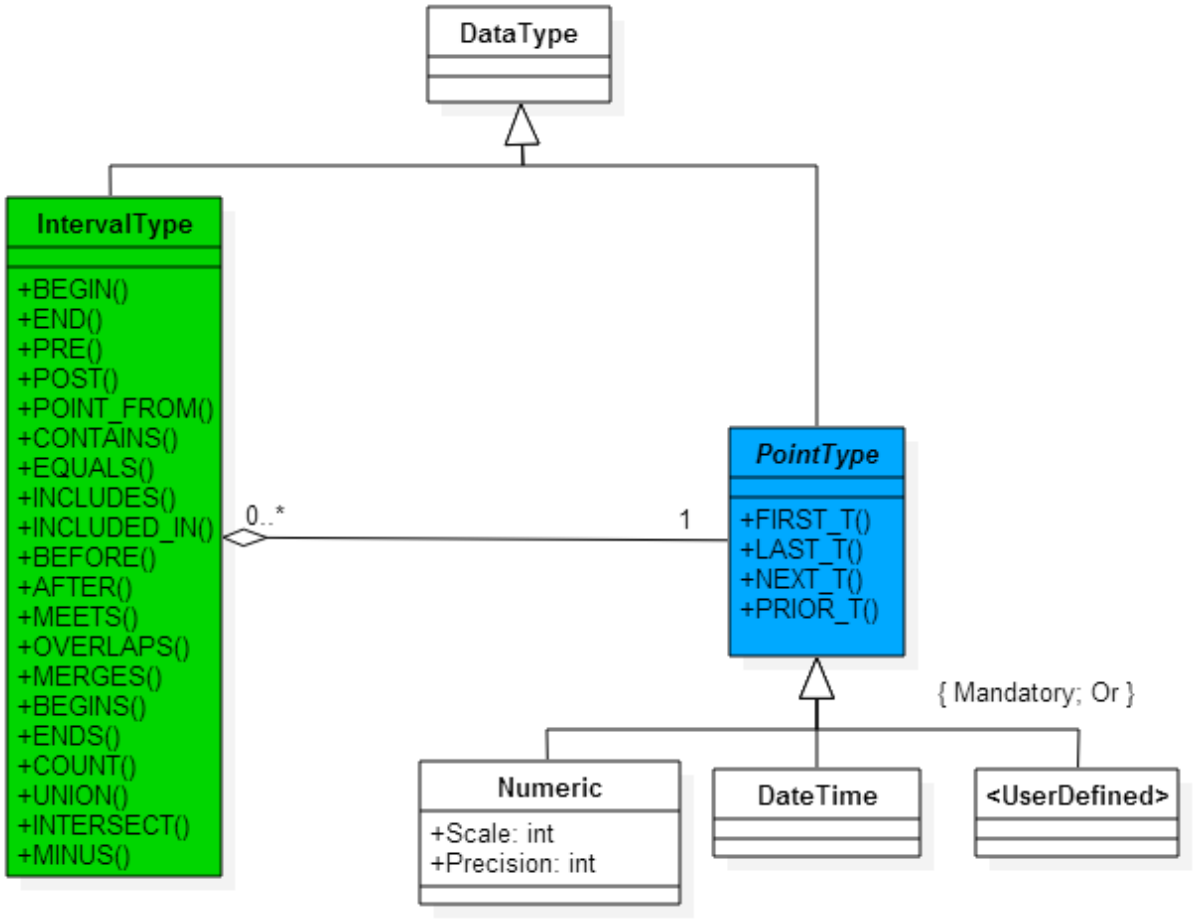


Figure 1 Conceptual view on point and interval types

2.3 The EXPAND and COLLAPSE Operators

In this section operators EXPAND and COLLAPSE are introduced that apply on sets of interval values (intervals), unlike the operators described previously, which applied on a single interval or pairs of intervals. They take a set of intervals (all of the same interval type) as input and they produce another set of intervals as their result. The result can be regarded as a particular canonical form of the input set. In **the book**, operators EXPAND and COLLAPSE were first described as applying to regular sets of intervals. We describe them straight away as applying on unary tables (one column) of such intervals.

2.3.1 The EXPAND Operator

Let X1 and X2 be two unary tables consisting of values of the same interval type. We can say that these two tables are equivalent if the set of all points contained in X1 intervals is equal to the set of all points contained in X2 intervals. This by the way assumes that the table has a key (in this case (X1)) that prevents duplicate rows in the table. This is something that the relational model requires (each base table must have a key) but SQL does not. Moreover, the relational model requires that all the duplicate rows must be automatically eliminated from the query result but SQL does not.

For example, tables X1 and X2 are shown in *Table 4* and *Table 5*. They both contain date intervals.

Table 4

X1
[2014-06-28:2014-06-30]
[2014-06-28:2014-06-29]
[2014-07-04:2014-07-04]
[2014-07-03:2014-07-05]

Table 5

X2
[2014-06-28:2014-06-28]
[2014-06-29:2014-06-30]
[2014-07-03:2014-07-04]
[2014-07-05:2014-07-05]

These tables are clearly not equal but they are equivalent according to our definition, since they both contain only the dates 2014-06-28, 2014-06-29, 2014-06-30, 2014-07-03, 2014-07-04 and 2014-07-05.

For reasons that will come clear soon, we are actually interested not so much in the points, but the *unit intervals* instead. An interval is a unit interval in case its begin and end points are equal. Thus, unit intervals in case of these tables are [2014-06-28:2014-06-28], [2014-06-29:2014-06-29], [2014-06-30:2014-06-30], [2014-07-03:2014-07-03], [2014-07-04:2014-07-04] and [2014-07-05:2014-07-05].

EXPAND operator does exactly that – transforms the input unary table to its expanded form. In our case, either of the tables X1 or X2, when EXPAND operator is applied on them, will result in the expanded form X3 shown in *Table 6*.

Table 6

X3
[2014-06-28:2014-06-28]
[2014-06-29:2014-06-29]
[2014-06-30:2014-06-30]
[2014-07-03:2014-07-03]
[2014-07-04:2014-07-04]

X3
[2014-07-05:2014-07-05]

Thus, we can say that two unary tables containing intervals are equivalent if they have the same expanded form.

2.3.2 The COLLAPSE Operator

Tables X1, X2 and X3 in the previous sub-section had different *cardinality* (count of rows). For X1 it was four, for X2 it was four, and for X3 it was six. The result of the COLLAPSE operator must produce a table that has the same expanded form as the input table but with *minimum cardinality*. This is called the *collapsed form*. In our example, the collapsed form of tables X1 and X2 is the table X4 shown in *Table 7:*

Table 7

X4
[2014-06-28:2014-06-30]
[2014-07-03:2014-07-05]

If X is a table that contains values of the same interval type, then a table Y is the collapsed form of X, when:

- X and Y have the same expanded form
- No two distinct intervals i1 and i2 in Y are such that i1 MERGES i2 is defined.
- No two distinct intervals i1 and i2 in Y are such that i1 INTERSECT i2 is defined.
- No two distinct intervals i1 and i2 in Y are such that i1 UNION i2 is defined.

It follows from this last point that Y can be computed from X by successively replacing pairs of intervals in X by their union until no further such replacements are possible.

2.4 The PACK and UNPACK Operator

The PACK and UNPACK operators are certain relational operators that build on operators COLLAPSE and EXPAND introduced in the previous chapter. They both provide shorthand to use the COLLAPSE and EXPAND operators on multiple attributes with one command and provide some additional benefit as well.

2.4.1 The UNPACK Operator

We start with the UNPACK operator. To better explain it, we use the example from *Table 8*:

Table 8

CUSTOMER_STATUS_HIST		
Customer_Id	Status_Type_Code	Status_During
1	1	[2014-06-28:2014-07-01]
1	2	[2014-07-02:2014-07-03]

The expression „*UNPACK CUSTOMER_STATUS_HIST ON Status_During*“ would then result as shown in *Table 9*. Note that „*ON Status_During*“ means that grouping is done based on all other columns apart *Status_During*.

Table 9

CUSTOMER_STATUS_HIST		
Customer_Id	Status_Type_Code	Status_During
1	1	[2014-06-28:2014-06-28]
1	1	[2014-06-29:2014-06-29]
1	1	[2014-06-30:2014-06-30]
1	1	[2014-07-01:2014-07-01]
1	2	[2014-07-02:2014-07-02]

CUSTOMER_STATUS_HIST		
Customer_Id	Status_Type_Code	Status_During
1	2	[2014-07-03:2014-07-03]

The result is called the unpacked form of the initial table and could be used to get the information in an atomic level.

2.4.2 The PACK Operator

The PACK operator provides us the *packed form* of the initial table and could be used to get the information in *clumps*. For example, we have, for some reason, such overlapping in *Table 10*:

Table 10

CUSTOMER_STATUS_HIST		
Customer_Id	Status_Type_Code	Status_During
1	1	[2014-06-28:2014-06-30]
1	1	[2014-06-29:2014-07-01]
1	2	[2014-07-02:2014-07-02]
1	2	[2014-07-02:2014-07-03]

The expression „*PACK CUSTOMER_STATUS_HIST ON Status_During*“ would then result as shown in *Table 11*.

Table 11

CUSTOMER_STATUS_HIST		
Customer_Id	Status_Type_Code	Status_During

CUSTOMER_STATUS_HIST		
Customer_Id	Status_Type_Code	Status_During
1	1	[2014-06-28:2014-07-01]
1	2	[2014-07-02:2014-07-03]

Note that „*ON Status_During*“ means that grouping is done based on all other columns apart *Status_During*, so in this case based on columns *Customer_Id* and *Status_Type_Code*.

2.4.3 Packing and Unpacking on no Columns and on Several Columns

Up to now, we have used packing and unpacking only on a single column. However, it is possible to use them on any set of columns as long as all of the columns are of interval type. And since empty set is a subset of any set, packing and unpacking can be used on no columns as well. The result of packing or unpacking a table *t* on no columns would return the same table.

For example, if *CUSTOMER_STATUS_HIST* in *Table 11* would be packed on no columns, then the same *CUSTOMER_STATUS_HIST* would be the result.

The following two conclusions can be made when packing and unpacking is used on no columns:

1. Packing a table *t* on no columns and then unpacking the result, also on no columns, returns *t.*
2. Unpacking a table *t* on no columns and then packing the result, again on no columns, returns *t*.

Unpacking on several columns:

UNPACK r ON (AI, A2, ..., An) = UNPACK (... (UNPACK (UNPACK r ON B1) ON B2) ...) ON Bn, where the sequence of column names *B1, B2 .... , Bn* is some arbitrary permutation of the specified sequence of column names *AI, A2, ..., An*.

So UNPACK on several columns just calls UNPACK on all the specified columns in any order. And *UNPACK (A1, A2) = UNPACK (A2, A1)*

In *Table 12* there is a table *CUSTOMER_PRODUCT_HIST* with 3 columns: *Customer_Id* – identifier of the customer, *Product_Id_Interval* – the range of product id values that the customer is consuming, *Date_Interval* – the range of dates during the customer was consuming the list of products. For example, if we look at the first row, then it states that customer with *Customer_Id* 1 was consuming products with *Product_Id* values 3, 4, 5 during days 2014-06-28, 2014-06-29 and 2014-06-30.

Table 12

CUSTOMER_PRODUCT_HIST		
Customer_Id	Product_Id_Interval	Date_Interval
1	[3:5]	[2014-06-28:2014-06-30]
1	[6:7]	[2014-06-29:2014-07-01]
1	[8:8]	[2014-07-01:2014-07-01]
1	[9:9]	[2014-07-01:2014-07-01]

If we would now unpack this table on columns *Product_Id_Interval* and *Date_Interval*, then by definition we would have two consecutive calls of the *UNPACK* operator:

```
UNPACK (
        UNPACK
                CUSTOMER_PRODUCT_HIST
        ON Product_Id_Interval
) ON Date_Interval
```

After the first unpacking (done on column *Product_Id_Interval*) the intermediate result would look like as shown in *Table 13.*

Table 13

CUSTOMER_PRODUCT_HIST		
Customer_Id	Product_Id_Interval	Date_Interval
1	[3:3]	[2014-06-28:2014-06-30]
1	[4:4]	[2014-06-28:2014-06-30]
1	[5:5]	[2014-06-28:2014-06-30]
1	[6:6]	[2014-06-29:2014-07-01]
1	[7:7]	[2014-06-29:2014-07-01]
1	[8.8]	[2014-07-01:2014-07-01]
1	[9:9]	[2014-07-01:2014-07-01]

Next we will unpack this intermediate result on column *Date_Interval* and we will get the final result in *Table 14*

Table 14

CUSTOMER_PRODUCT_HIST		
Customer_Id	Product_Id_Interval	Date_Interval
1	[3:3]	[2014-06-28:2014-06-28]
1	[3:3]	[2014-06-29:2014-06-29]
1	[3:3]	[2014-06-30:2014-06-30]
1	[4:4]	[2014-06-28:2014-06-28]
1	[4:4]	[2014-06-29:2014-06-29]
1	[4:4]	[2014-06-30:2014-06-30]

CUSTOMER_PRODUCT_HIST		
Customer_Id	Product_Id_Interval	Date_Interval
1	[5:5]	[2014-06-28:2014-06-28]
1	[5:5]	[2014-06-29:2014-06-29]
1	[5:5]	[2014-06-30:2014-06-30]
1	[6:6]	[2014-06-29:2014-06-29]
1	[6:6]	[2014-06-30:2014-06-30]
1	[6:6]	[2014-07-01:2014-07-01]
1	[7:7]	[2014-06-29:2014-06-29]
1	[7:7]	[2014-06-30:2014-06-30]
1	[7:7]	[2014-07-01:2014-07-01]
1	[8:8]	[2014-07-01:2014-07-01]
1	[9:9]	[2014-07-01:2014-07-01]

Packing on several columns:

PACK r ON (At, A2, ... An) = PACK (... (PACK (PACK r' ON AI) ON A2) ...) ON An, where r' is *UNPACK r ON (At, A2, . . . , An).*

Thus, order of PACK operations is important, and UNPACK (A1, A2) ≠ UNPACK (A2, A1)

Also, a preliminary unpacking on all the columns is needed to make sure that there would not be any redundancy in the result.

So, if we would pack the table *CUST_PRODUCT_HIST* from *Table 12* on columns *Product_Id_Interval* and *Date_Interval*, we would make such steps:

PACK (

PACK (

UNPACK CUSTOMER_PRODUCT_HIST ON (Product_Id_Interval, Date_Interval)

) ON Product_Id_Interval

) ON Date_Interval

As we have the result of the preliminary unpacking already in *Table 14*, we can right away carry on with the packing on column *Product_Id_Interval*. The result of this first packing is seen in *Table 15*.

Table 15

CUSTOMER_PRODUCT_HIST		
Customer_Id	Product_Id_Interval	Date_Interval
1	[3:5]	[2014-06-28:2014-06-28]
1	[3:7]	[2014-06-29:2014-06-29]
1	[3:7]	[2014-06-30:2014-06-30]
1	[6:9]	[2014-07-01:2014-07-01]

Now, to get the final result, we will do the second packing – this time on column *Date_Interval*. The final result is seen in *Table 16*. Table *CUSTOMER_PRODUCT_HIST* is in its packed form - with minimum cardinality and without any redundancy. By "without any redundancy" is meant that there are no two rows that are stating the same fact twice. For example, it is only said in the first row that customer with *Customer_Id* 1 was consuming product with *Product_Id* 3 on date 2014-06-28.

Table 16

CUSTOMER_PRODUCT_HIST		
Customer_Id	Product_Id_Interval	Date_Interval
1	[3:5]	[2014-06-28:2014-06-28]

CUSTOMER_PRODUCT_HIST		
Customer_Id	Product_Id_Interval	Date_Interval
1	[3:7]	[2014-06-29:2014-06-30]
1	[6:9]	[2014-07-01:2014-07-01]

For showing that the order of the columns the table is packed on is important, we provide a step by step example of *PACK CUSTOMER_PRODUCT_HIST(Date_Interval, Product_Id_Interval)* as well.

So, if we would pack the table *CUST_PRODUCT_HIST* from *Table 12* on columns *Date_Interval* and *Product_Id_Interval*, we would make such steps:

```
PACK (
    PACK (
        UNPACK CUSTOMER_PRODUCT_HIST ON (Date_Interval, Product_Id_Interval)
    ) ON Date_Interval
) ON Product_Id_Interval
```

As we have the result of the preliminary unpacking already in *Table 14* (since the order of the parameters is not important in an *UNPACK* operation), we can right away carry on with the packing on column *Date_Interval*. The result of this first packing is shown in *Table 17*.

Table 17

CUSTOMER_PRODUCT_HIST		
Customer_Id	Product_Id_Interval	Date_Interval
1	[3:3]	[2014-06-28:2014-06-30]
1	[4:4]	[2014-06-28:2014-06-30]
1	[5:5]	[2014-06-28:2014-06-30]

CUSTOMER_PRODUCT_HIST		
Customer_Id	Product_Id_Interval	Date_Interval
1	[6:6]	[2014-06-29:2014-07-01]
1	[7:7]	[2014-06-29:2014-07-01]
1	[8:8]	[2014-07-01:2014-07-01]
1	[9:9]	[2014-07-01:2014-07-01]

Now, to get the final result, we will do the second packing – this time on column *Product_Id_Interval*. The final result is seen in *Table 16*. As it is clearly seen, results in *Table 16* and *Table 18* are different but they are equivalent. The result is still the packed form of *CUSTOMER_STATUS_HIST* and it is only said in the first row that customer with *Customer_Id* 1 was consuming product with *Product_Id* 3 on date 2014-06-28.

Table 18

CUSTOMER_PRODUCT_HIST		
Customer_Id	Product_Id_Interval	Date_Interval
1	[3:5]	[2014-06-28:2014-06-30]
1	[6:7]	[2014-06-29:2014-07-01]
1	[8:9]	[2014-07-01:2014-07-01]

2.4.4 Relational Operators

Relational operators, such as UNION, MINUS and INTERSECT are used for combining multiple tables (constructed by using a query) together. In **the book**, a set of new operators are introduced, with a slightly improved functionality for providing better support for queries

that involve columns with an interval data type. The new operators are named U_UNION, U_MINUS, U_INTERSECT, etc. and their logic is similar:

1. UNPACK both of the queries using some interval column. Note that the prefix *"U_"* is added because of the keyword USING that specifies that grouping is be done based on all the columns except the one stated after the USING keyword. For example, if the component queries return columns *Customer_Id, Country_Code* and *During*, and the epression is *USING During SELECT Query_1 U_MINUS Query_2*, then results of both of the component queries will first be unpacked with GROUP BY *Customer_Id, Country_Code.*

2. Run the corresponding operator on the unpacked query results

3. PACK the result of the operator on the same interval column

This provides us the functionality to look at the interval columns as they really are – a set of point type values as opposed to the way the regular relational operators see them. For example, consider the following two queries with their results shown in Table 19.

Query 1: `SELECT Customer_Status_During AS During FROM CUSTOMER_STATUS_HIST WHERE Customer_Id=1 AND Customer_Status_Code=1`

Query 2: `SELECT Customer_Status_During AS During FROM CUSTOMER_STATUS_HIST WHERE Customer_Id=2 AND Customer_Status_Code=1`

Table 19

Query1 *During*	Query2 *During*
[2014-06-27:2014-07-01]	[2014-06-29:2014-07-02]

If the regular MINUS is used then *SELECT Query1 MINUS Query2* gives us the result shown in Table 20 – the result is equal to the result of *Query1*.

Table 20

Query1 MINUS Query2 *During*
[2014-06-27:2014-07-01]

Now, if we use the operator U_MINUS, then first the results of both of the queries are unpacked as seen in Table 21.

Table 21

Query1_Unapacked *During*	Query2_Unapacked *During*
[2014-06-27:2014-06-27]	[2014-06-29:2014-06-29]
[2014-06-28:2014-06-28]	[2014-06-30:2014-06-30]
[2014-06-29:2014-06-29]	[2014-07-01:2014-07-01]
[2014-06-30:2014-06-30]	[2014-07-02:2014-07-02]
[2014-07-01:2014-07-01]	

Then the regular MINUS operator is run on the unpacked queries and the result is shown in Table 22.

Table 22

Query1_Unapacked MINUS Query2_Unapacked *During*
[2014-06-27:2014-06-27]
[2014-06-28:2014-06-28]

Finally, the result of the MINUS operator is packed and it gives us the following interval as a result: [2014-06-27:2014-06-28].

In addition to these three operators, there were more listed and modified version was created – U_JOIN and U_= among others. We will not give an example of each of those operators but

will wrap up this section by stating that they all provide convenient shorthand for using the interval data types as any other data type.

2.4.5 Possible Database Designs to Represent Temporal Data

2.4.5.1 Current Tables Only

Most of the operational databases store the information in a semi-temporal manner. It means that there is usually a *Since* column that shows since when the current information shown in the row has been considered to be true. There is a big problem with this approach, though, because right after a value in some of the column changes, we will lose the information about the past. For example, as shown in Table 23, the fact that customer 1 has had the name Mari Tamm and status 1 since 2015-01-01 is shown by the column *Since.*

Table 23

CUSTOMER			
Customer_Id	Since	Customer_Name	Customer_Status
1	2015-01-01	Mari Tamm	1

Now if she would get married on 2015-04-01, and the name would change to Mari Kask, then the column *Since* would get a new value 2015-04-01 and we would lose the information that she has had the status 1 since 2015-01-01. However, this problem is easy to fix: we will replace the *Since* column by three such columns, one for each of the other ("nonsince") columns. After this *horizontal decomposition*, the table would look like as seen in Table 24.

Table 24

CUSTOMER					
Customer_Id	Customer_Since	Customer_Name	Customer_Name_Since	Customer_Status	Customer_Status_Since
1	2015-01-01	Mari Kask	2015-04-01	1	2015-01-01

The problem still remains that it can show the current information only and we have lost the information that customer 1 had last name Tamm from 2015-01-01 to 2015-04-01.

We add a constraint right away that all the values of the *Since* columns that do not correspond to the primary key must be greater or equal to the *Since* column that goes along the primary key. In the table shown in Table 24, columns *Customer_Name_Since* and *Customer_Status_Since* must always have their value less than or equal to the value of column *Customer_Since*.

Note: Yes, this table can show some historical information – namely the period that is between the Since column and today. We also note that adding a Since column to each "nonsince" column is of course not needed if we are not interested in the history of this column. For instance, the value of column Customer_Name hardly changes for a customer and it might not be needed to add the Customer_Name_Since column. But this is a decision to be made when designing a concrete database.

2.4.5.2 Historical Tables Only

We now turn to the design that includes historical information only, loosely, since such tables can also contain information about the future. An example of such table is shown in Table 25.

Table 25

CUSTOMER_HIST			
Customer_Id	During	Customer_Name	Customer_Status
1	[2015-01-01:"the last day")	Mari Tamm	1

Note: The marker "the last day" is used in this example for showing that the end point of the During value is not known currently so the fact is currently considered to be true. We will have more to say in the end of this section.

Suppose now we find out that the customer got married on 2015-04-01 and her last name changed to Kask. In order to make the needed changes in the database, we need to make the following changes:

1. Update the current row and set the end point of the *During* column value to 2015-03-31.

2. Insert a new row to showing that customer 1 has last name Kask from 2015-04-01 to "the last day").

This example shows that such design is not very good because it timestamps a combination of all of the columns. We would rather do a *vertical decomposition* and create a separate history table for each of those columns. Thus, in the example case, we would have three separate tables for customer, customer name, and customer status as shown in Table 26, Table 27, and Table 28 respectively.

Table 26

CUSTOMER_HIST	
Customer_Id	During
1	[2015-01-01:"the last day")

Table 27

CUSTOMER_NAME_HIST		
Customer_Id	Customer_Name	During
1	Mari Tamm	[2015-01-01:2015-04-01)
1	Mari Kask	[2015-04-01:"the last day")

Table 28

CUSTOMER_STATUS_HIST		
Customer_Id	Customer_Status	During
1	1	[2015-01-01: "the last day")

Vertical decomposition transformed the information to be stored in the tables in the sixth normal form. The tables have primary key (*Customer_Id*, *Country_Code* and *During*) and at most one additional column. Values of the columns in a traditional historical table can change with a different rate (for instance, customer's status can change much more often than his/her customer's name) and it is not wise to "drag" the values of other columns along each time. One of the benefits of the vertical decomposition is that we can update the values of each attribute independently from others (*Customer_Status* and *Customer_Name* are attributes of entity type *Customer*).

Please note that the support to application-time and system-time period by SQL:2011 means that current and historic data is together in the same table. Creating these tables in sixth normal form is a possible design approach of these tables.

However, we still cannot be one hundred per cent happy with the approach of using historical tables only. The reason is that, in case of a currently effective fact – current customer name, for example – we need to put something to mark the end point of the interval. What we need is something saying that this fact is currently true until further notice. This is the problem of "the moving point *now*". One possibility that has been under consideration is a *NOW* variable indicating the current time. An example of such interval value is [2015-01-01:*NOW*] that denotes the interval between January 1, 2015 and current date (end points included). However, this brings a lot of problems especially when the granularity of the point the interval consists of is very small – for instance with the precision of a millisecond. Moreover, the authors of **the book** believe that introducing such a variable would be a mistake – as it was a big logical mistake to implement *NULL* in SQL (Date, 2006). Actually many databases have *NULL* indicating the end point of some period but as said, this is not how it should be. Another option would be to use a variable referring to "the last day" but this would mean that

a. we would store a lie in the database since the end point of such fact clearly cannot be "the last day"

b. users should interpret "the last day" to mean "until further notice" in most of the cases

Thus, we need an approach that does not tell. The best way not to tell a lie is not to say it at all – if we currently do not know what the end point of some period is, then we simply do not show it. This brings us to the proposed design that we will discuss in the next section.

2.4.5.3 Both Current and Historical Tables

As described in the previous two sections, the first approach of only current tables is great for storing current information and the second approach of having only historical tables is great for storing historical information. The proposed solution in **the book** is a combination of both of these approaches:

- We will store the current information in the current tables.
- We will store the historical information in the historical tables. NB! Only the information with validity in the past will be stored in them.

With this approach, we can correctly store the historical information while avoiding the problem of "the moving point *now*". This approach is actually a good example of the "separation of concerns" design principle (Separation of concerns). There are separate tables for current data (one concern) and separate tables for historic data (another concern). Concern is "A canonical solution abstraction that is relevant for a given problem" (Separation of concerns). One problem (that needs a solution) is how to represent current data and another how to represent historic data.

Let's go through an example of how we get the proposed design from a regular historical table shown in Table 29.

Table 29

CUSTOMER_HIST			
Customer_Id	During	Customer_Name	Customer_Status
1	[2015-01-01:2015-04-01)	Mari Tamm	1
1	[2015-04-01:2015-05-01)	Mari Kask	1
1	[2015-05-01:"Last day")	Mari Kask	2

First, we create a current table CUSTOMER with all the columns except *During* and we add a corresponding *Since* column for each of these columns. We include only the current information about each attribute into this table. The result is shown in Table 30.

Table 30

CUSTOMER					
Customer_Id	Customer_Since	Customer_Name	Customer_Name_Since	Customer_Status	Customer_Status_Since
1	2015-01-01	Mari Kask	2015-04-01	2	2015-05-01

Next, a historical table is created for each of these columns and we will move the remaining information there from Table 29. The result is shown in Table 31, Table 32, and Table 33. Note that *CUSTOMER_HIST* does not have any rows since there is only current information about it.

Table 31

CUSTOMER_HIST	
Customer_Id	During

Table 32

CUSTOMER_NAME_HIST		
Customer_Id	Customer_Name	During
1	Mari Tamm	[2015-01-01:2015-04-01)

Table 33

CUSTOMER_STATUS_HIST		
Customer_Id	Customer_Status	During
1	1	[2015-01-01: 2015-05-01)

In **the book** a set of integrity constraints we described that should be created on these tables for protecting the data from various temporal data quality issues. Additionally, it was proposed to create a view for each of the temporalized attributes that would combine the current and historical information of each of these attributes (undoing the horizontal

decomposition). Furthermore, there should be a view created that would join all of these attribute views together for providing users a comprehensive picture about the history of the entity and its attributes. Finally, there should be a mechanism in place on top of these views that should make data changes easier for the user by generating and executing DML statements automatically on the background. All of these actions should be done automatically by the system, after the user has specified that an attribute needs to be temporalized. We will cover them in more detail in the next chapter where the implementation of the ideas in PostgreSQL is introduced.

3. Implementing the Ideas into a PostgreSQL Database

3.1 Introduction

In this chapter we will describe the development tasks needed for implementing the ideas proposed in **the book** using PostgreSQL 9.3 DBMS. All the database objects that were developed are uploaded to a GitHub repository that is accessible with the following link: https://github.com/SanderLaasik/temporal. The code is open source with MIT license, meaning that everyone can copy, modify, and publish it. There are two ways for adding the database objects from this repository:

a) Execute all the statements from the file *temporal--x.y* (where *x.y* is the version number of the file)

b) Put the files *temporal--x.y* and *temporal.control* into the PostgreSQL installation directory `SHAREDIR/extension` and then use the CREATE EXTENSION syntax for creating all the database objects from the file *temporal--x.y*.

As a result of executing the statements from the file *temporal--x.y*, the schema named *TEMPORAL* is created. All the functions are created into this schema and therefore the schema name should be specified when using any of the functions. Furthermore, the table named *TEMPORAL_METADATA* is created that will store the metadata about the tables and attributes that the temporal support is enabled on. This table is used by some functions and trigger procedures for lookup of the temporal metadata. We will provide the list of its columns and their descriptions later on.

For representing and storing the intervals, PostgreSQL has a set of *range* data types. The boundary values can be set exactly how we need it and how it was described in **the book** – for inclusive bounds brackets "[]" and for explicit bounds parenthesis "()" should be used. In addition, there is a special range point variable *infinity* introduced for showing that the range's

upper bound is infinity. Currently there are six system-defined (built-in) range types implemented:

- int4range – Range of *integer*
- int8range – Range of *bigint*
- numrange – Range of *numeric*
- tsrange – Range of *timestamp without time zone*
- tstzrange – Range of *timestamp with time zone*
- daterange – Range of *date*

As the aim of this thesis is to add the temporal support to intervals that are consisting of date values, we can use *daterange* for it.

PostgreSQL also allows user-defined range types to be created using the CREATE TYPE syntax.

3.2 Point Type and Interval Type Operators

PostgreSQL has a set of system-defined operators for dealing with intervals. We will now provide a comparison of the interval operators described in **the book** and the operators that are available in PostgreSQL. The comparison is shown in Table 34 and the description of each of these operators can be found in Sections 2.2.1 and 2.2.2. As seen from Table 34, most of the needed operators are already available in PostgreSQL but we have also created corresponding PL/pgSQL functions based on the logic provided in **the book** to give a more understandable name for them. For the operators that are missing in PostgreSQL, we have created a PL/pgSQL function based on the logic provided in **the book**.

Table 34

Point type operators for data type *date*			
Operator from the book	**PostgreSQL equivalent**	**Comment**	**My implementation for data type *date***
NEXT_T	*NA*		NEXT_DATE(*Date d*)
PRIOR_T	*NA*		PRIOR_DATE(*Date d*)

Point type operators for data type *date*			
Operator from the book	**PostgreSQL equivalent**	**Comment**	**My implementation for data type *date***
FIRST_T	*NA*		FIRST_DATE()
LAST_T	*NA*		LAST_DATE()
Singe interval operators			
Operator from the book	**PostgreSQL equivalent**	**Comment**	**My implementation for data type *date***
INTERVAL_T	Regular data type specification		Used PostgreSQL system defined type DATERANGE
BEGIN	LOWER()		BEGIN(*Daterange i)*
END	UPPER()-1*	* - for some reason, UPPER returns “The last point + 1"	END(*Daterange i)*
p ∈ i	<@	element is contained by range	CONTAINED_IN(*Date d*, *Daterange i)*
I ∋ p	@>	contains element	CONTAINS(*Daterange i, Date d)*
PRE	*NA*		PRE(*Daterange i)*
POST	UPPER()*	* - for some reason, UPPER returns “The last point + 1"	POST(*Daterange i)*
COMPARISON OPERATORS			
Operator from the book	**PostgreSQL equivalent**	**Comment**	**My implementation for data type *date***
EQUALS (=)	=	equals	EQUALS(*Daterange i, Daterange i)*
INCLUDES (i1 ⊇ i2)	@>	contains range	INCLUDES(*Daterange i, Daterange i)*
INCLUDED_IN (i1 ⊆ i2)	<@	range is contained by	INCLUDED_IN(*Daterange i, Daterange i)*
BEFORE	<<	strictly left of	BEFORE(*Daterange i, Daterange i)*
AFTER	>>	strictly right of	AFTER(*Daterange i, Daterange i)*
MEETS	-|-	is adjacent to	MEETS(*Daterange i, Daterange i)*
OVERLAPS	&&	overlap (have points in common)	OVERLAPS(*Daterange i, Daterange i)*
MERGES	*NA*		MERGES(*Daterange i, Daterange i)*
BEGINS	*NA*		BEGINS(*Daterange i, Daterange i)*
ENDS	*NA*		ENDS(*Daterange i, Daterange i)*
OTHER OPERATORS			
Operator from the book	**PostgreSQL equivalent**	**Comment**	**Our implementation for data type *date***
COUNT	*NA*		COUNT(*Daterange i)*
MAX	*NA*		MAX(*Daterange i)*
MIN	*NA*		MIN(*Daterange i)*
UNION	+	union	UNION(*Daterange i, Daterange i)*
INTERSECT	*	intersection	INTERSECT(*Daterange i, Daterange i)*
MINUS	-|-	difference	MINUS(*Daterange i, Daterange i)*
EXPAND	NA		EXPAND (*Daterange[] i*) Note: First use ARRAY_AGG() on the input parameter to transform rows into an array of *dateranges*
COLLAPSE	NA		COLLAPSE(*Daterange[] i*) Note: First use ARRAY_AGG() on the input parameter to transform rows into an array of *dateranges*
PACK	NA		EXPAND (using GROUP BY)
UNPACK	NA		COLLAPSE (using GROUP BY)

We have not implemented the PACK and UNPACK operators because the result of their logic (used on one column) can be accomplished using the EXPAND and COLLAPSE operators, and the GROUP BY syntax. Support of using the PACK and UNPACK operators on multiple attributes is not supported because based on our knowledge and understanding it would have required modifications in the source code of PostgreSQL by adding a USING keyword with the logic described in **the book**. The main reason is that the current grouping functionality in SQL – GROUP BY – expects the user to provide the list of columns that the dataset should be grouped by. The "USING <*column name of interval data type*>" syntax, as described in **the book,** will do the grouping automatically based on all the other columns except the <*column name of interval data type*>. Most probably it is possible to add this support using PL/pgSQL also but we decided not to spend more time on it. Our decision is supported by the fact that the proposed database design can survive without having it – none of the tables contains more than one column with an interval data type. But we agree that support to such functionality should be added with some future developments.

NB! There is a limitation in the function EXPAND that if the upper bound of an interval is *'infinity'*, then the unit intervals are only created up to *current_date*. Without this limitation, the execution would run forever since it is defined as "*later than all other time stamps*".

3.3 Relational Operators

As described in section 2.4.4 the regular relational operators such as MINUS, INTERSECT and JOIN were used to create modified versions of them, named U_MINUS, U_INTERSECT, and U_JOIN respectively. We have implemented five of them as PL/pgSQL functions named *TEMPORAL_EQUALS*, *TEMPORAL_INTERSECT*, *TEMPORAL_MINUS*, *TEMPORAL_UNION*, and *TEMPORAL_JOIN*. Again, they were created to support only the data type *daterange*. They all work as expected but they look very clumsy and are not very easy to use. Mostly because of the same problem as stated in the previous section – SQL does not have the USING keyword in place. As said earlier, all of them use the *PACK* and *UNPACK* operators that are aggregate in their logic and this brings us back to the need for specifying the list of columns to be used for grouping the rows. Nevertheless, we have used some of them (or at least the logic of them) when creating temporal constraints that will be discussed in the next section.

3.4 Example Database with Temporal Support

3.4.1 Introduction

In this section, the creation of a fully temporal database is described. We will build it up step by step, starting from defining the initial semi-temporal database and then adding temporal support to its tables and their columns.

We will use the term *entity* for referring to an object of a fact – for example a customer – and the term *attribute* for referring to the *named properties* of these *entities* – for example the name of the customer. *Entity type* is a set of entities with common attributes.

As said earlier, this design keeps the current data of all attributes in a current table (with a corresponding *Since* column for each attribute, and no *Since* column can have value later than *today*) and the historical data of each attribute is stored in a separate history table. We will call the historical table the *DURING* table and the current table the *SINCE* table from this point forward.

After a user enables temporal support on any of the columns, all the needed activities are done automatically by the system. This includes creating a history table for the attribute in question, creating a set of PL/pgSQL trigger procedures for keeping the data satisfying a set of temporal requirements, creating a view for this attribute that combines the historical and current data together and also creating (or replacing) a summary view that combines together data from all of these views.

3.4.2 Design Model of Initial Database

We now introduce design model of the initial database. Conceptually the database stores information about customers and their contracts. Figure 2 shows the database design model in detail.

Customer data is stored in a table named *CUSTOMER*. It has a primary key consisting of columns *Customer_Id* and *Country_Code*. In addition it has columns *Customer_Name* and *Customer_Segment_Code*, and columns *Customer_Name_Since* and *Customer_Segment_Since* that shows the validity start date of these attributes correspondingly.

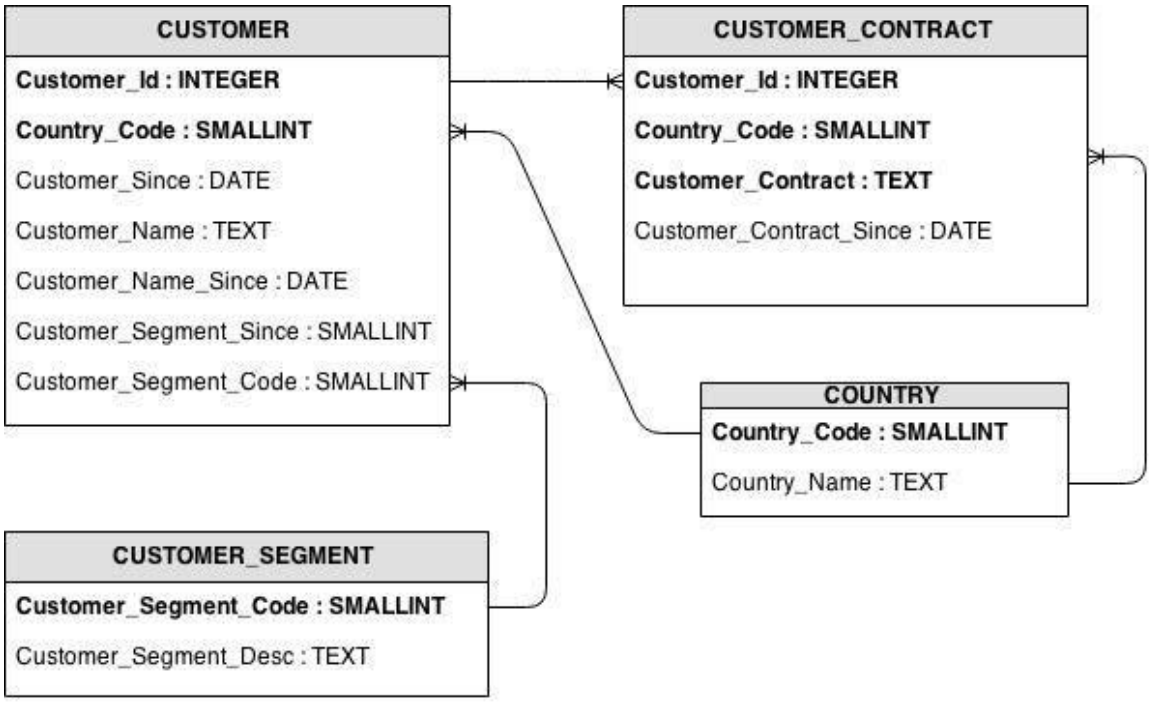


Figure 2 Model of the initial database

Customer's relations to its contracts are stored in table *CUSTOMER_CONTRACT* that has the primary key consisting of columns *Customer_Id*, *Country_Code* and *Contract_Nbr*. Furthermore, columns *Customer_Id* and *Country_Code* are foreign key members referencing the primary key of table *CUSTOMER*.

In addition, there are two classifier tables *COUNTRY* and *CUSTOMER_SEGMENT_TYPE* that store for each classifier value a code and a description.

3.4.3 Enabling Temporal Support on an Attribute

In order to enable temporal support for some entity, the *SINCE* table needs to be in place and preferably populated with the current data about this entity. This *SINCE* table needs to have a corresponding *Since* column for each attribute that needs to be temporalized, and no *Since* column can have value later than *today*.

We have created a PL/pgSQL function called *TEMPORALIZE* to enable temporal support on some attribute. The function uses dynamic SQL for reusing the same code for all the attributes that one may want to temporalize. It has six input parameters that require corresponding argument when one invokes the function. For each parameter, we next provide its name, type, and a short description.

1. *In_Schema_Name* TEXT – specifies the name of the schema where the *SINCE* table is located. For example *FTE*

2. *In_Table_Name* TEXT – specifies the name of the *SINCE* table of the entity in question. For example *CUSTOMER*

3. *In_Table_Columns_Names* TEXT – specifies the list of column names that the attribute consists of. For normal cases there is only one column (for example *Customer_Name*) but in some cases there can be multiple columns, especially when enabling temporal support on the primary key (for example, if the table *CUSTOMER* has primary key consisting of columns *Customer_Id* and *Country_Code*)

4. *In_Since_Name* – specifies the name of the *Since* column in the *SINCE* table. For example *Customer_Name_Since*

5. *In_Hist_Table_Name* – specifies the name of the "to be" *DURING* table name. For example *CUSTOMER_NAME_HIST*

6. *In_Combining_View_Name* – specifies the name of the "to be" view name that will combine the historical and current information about this attribute

If a table *T* contains data corresponding to N attributes and one wants to temporalize (in terms of making possible preserving historic attribute values) all these attributes, then:

- one has to add the *since* column for each attribute,

- one has to invoke the TEMPORALIZE function N times – once for each attribute.

The *TEMPORALIZE* function and the trigger procedures it references use the table *TEMPORAL_METADATA* for storing and using metadata about the temporalization of some entity and its attributes. The table consists of the following columns.

1. *Schema_Name* VARCHAR(63) – value of input parameter *In_Schema_Name* will be stored here

2. *Table_Name* VARCHAR(63) – value of input parameter *In_Table_Name* will be stored here

3. *Property_Column_List* TEXT – value of input parameter *In_Table_Columns_Names* will be stored here

4. *Is_Property_PK* BOOLEAN – if the columns listed in *In_Table_Columns_Names* match the primary key members of the table, then the value will be *TRUE* and *FALSE otherwise*

5. *Since_Column_Name* VARCHAR(63) – value of input parameter *In_Since_Name* will be stored here

6. *Hist_Table_Name* VARCHAR(63) – value of input parameter *In_Hist_Table_Name* will be stored here

7. *Combining_View_Name* VARCHAR(63) – value of input parameter *In_Combining_View_Name* will be stored here

The type is VARCHAR(63) because by default in PostgreSQL the maximum identifier length is 63.

The primary key of the table *TEMPORAL_METADATA* consists of columns *Schema_Name, Table_Name* and *Property_Column_List*.

After calling the function *TEMPORALIZE* for some attribute, the following actions will be executed.

1. If this is not a primary key attribute, then a check is made – based on the information in *TEMPORAL_METADATA* – if the table in question has temporal support enabled for primary key. If, then the temporalization fails because it is mandatory to have temporal support enabled on the primary key first.

2. *DURING* table for this attribute is created in the specified schema with the specified name. It has the following columns: *SINCE* table's primary key members + the columns listed in the input parameter *In_Columns_Names* + column *During*. Its primary key will consist of the *SINCE* table's primary key members + *During*. For example, if the attribute *Customer_Name* is temporalized (with *In_Hist_Table_*Name value *CUSTOMER_NAME_HIST*) then the corresponding *DURING* table with the name *CUSTOMER_NAME_HIST* is created that will have columns *Customer_Id*,

Country_Code, *Customer_Na*me and *During*. Furthermore, the primary key will consist of columns *Customer_Id*, *Country_Code* and *During*.

3. The fact that this attribute is temporalized is inserted into a metadata table *TEMPORAL.TEMPORAL_METADATA*.

4. A set of constraint triggers are created both on the *SINCE* and *DURING* tables to make sure that the information stored in the database is in accordance with the temporal requirements. We will discuss these requirements and the trigger procedures the triggers invoke in detail in the next section.

5. A view is created that will combine the historical and current information of this attribute together. It takes all the rows from the *DURING* table of this attribute and the needed columns from the *SINCE* table and – using the UNION ALL operator – combines them together. The header of the view is equal to the header of the *DURING* table. Furthermore, the *Since* value of this attribute is casted as an interval, with begin point equal to the value of *Since* and end point is set to *'infinity'*. For example, the view for attribute *Customer_Name* would have columns *Customer_Id, Country_Code*, *Customer_Name* and *During*. This view returns all the rows from *CUSTOMER_NAME_HIST*. In addition, it returns all the rows from *CUSTOMER* with the needed columns and *During* values with begin point equal to *Customer_Name_Since* and end point set to *'infinity'*.

6. Finally, a view with the name *<In_Table_Name>_FULL_VW* is created that will join all the combining views of attributes together. Note that the logic of operator *U_JOIN* is used for doing it. First, all the attribute views are unpacked to contain only unit intervals in their *During* columns. Then they are joined using the columns of the primary key view and the expanded *During* value. Finally, the resulting rows are packed using the *During* column of the primary key view. For example, in our example the combining views *CUSTOMER_VW*, *CUSTOMER_NAME_VW* and *CUSTOMER_SEGMENT_VW* are joined together and the view named *CUSTOMER_FULL_VW* will be created.

3.4.4 Trigger Procedures for Satisfying the Temporal Requirements

We will now describe the temporal requirements described in **the book** and create a set of PL/pgSQL trigger procedures to satisfy them. The aim of these constraints is to make sure that all the data contained in this database is both logically and design wise correct. This involves avoiding redundancy, circumlocution and contradiction in the data.

We will now explain the constraints one by one and describe the *trigger procedures* and *constraint triggers* that are created to satisfy them. We use the example database to illustrate the constraint triggers and the trigger procedures they invoke.

The logic of implementing these constraints is as follows.

1. The file *temporal—x.y* contains a set of PL/pgSQL trigger procedures with dynamic SQL that make the needed checks on the specified tables and their columns in a declarative manner

2. The function *TEMPORALIZE* creates constraint triggers that invoke these trigger procedures on the specified tables and columns. All the constraint triggers are specified to execute for each row modification and after all the statements in the transaction have finished – this is done by using the DEFERRABLE INITIALLY DEFERRED syntax:

```
CREATE CONSTRAINT TRIGGER <Trigger name>
AFTER INSERT OR UPDATE OR DELETE
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW EXECUTE PROCEDURE <Trigger procedure name
with parameters>
```

Possibility of creating constraint triggers, which execution can be deferred to the end of a transaction, is specific to PostgreSQL. It is not specified in the SQL standard (the last version at the time of writing the thesis is SQL:2011). Thus, there is no reason to expect such feature in other SQL DBMSs. We would have better used assertion objects (Gulutzan and Pelzer, 1999) that are general declarative constraints created as separate schema objects. Unfortunately, it was not supported in PostgreSQL at the time of writing the thesis. The author is currently aware of only one DBMS - TimeDB - that implements SQL assertion

object (TimeDB). In addition, PostgreSQL currently (spring 2015) does not permit subqueries in table CHECK constraints that could also be used for this purpose.

We created and ran a set of tests on the constraints to make sure that everything is working as expected. The tests can be seen in the file *constraint_tests.sql*, located in the GitHub repository accessible with the following link: https://github.com/SanderLaasik/temporal. We will specify each test when we describe the corresponding constraint.

3.4.4.1 Constraints on the *DURING* table

First, there are two integrity constraints that should be added to all *DURING* tables (in **the book**, there were also exceptions listed but we do not discuss them in scope of this thesis). **The book** describes both of them as lines of text and we have done our best to implement them using PL/pgSQL.

The first of these constraints prevents two integrity problems – redundancy and circumlocution – from occurring in a table.

Redundancy means that there are multiple rows saying the same thing. For example, the situation shown in Table 35 is a redundancy problem.

Note: we remind that using brackets means that the begin/end point of an interval is inclusive and using parenthesis means that it is exclusive.

Table 35

CUSTOMER _NAME_HIST			
Customer_Id	Country_Code	Customer_Name	DURING
1	1	Mari Tamm	[2015-01-01:2015-02-01)
1	1	Mari Tamm	[2015-01-30:2015-07-01)

Table *CUSTOMER_NAME_SINCE* clearly says twice that customer 1 had the name "Mari Tamm" on days 2015-01-30 and 2015-01-31. We would better like to see this information stored in a single row with a *During* value [2015-01-01:2015-07-01).

Circumlocution means that there are multiple rows saying something that could better be said with a single row. For example, the situation shown in Table 36 is a circumlocution problem.

Table 36

CUSTOMER _NAME_HIST			
Customer_Id	Country_Code	Customer_Name	DURING
1	1	Mari Tamm	[2015-01-01:2015-02-01)
1	1	Mari Tamm	[2015-02-01:2015-07-01)

It uses two separate rows for saying that Customer 1 had the name "Mari Tamm" during period [2015-01-01:2015-07-01).

Thus, both of these examples should actually be stored as shown in Table 37 where the table *CUSTOMER_NAME_HIST* is in its packed form. Therefore, **the book** proposes to introduce a table level PACKED ON constraint for making this check.

Table 37

CUSTOMER_NAME_HIST			
Customer_Id	Country_Code	Customer_Name	DURING
1	1	Mari Tamm	[2015-01-01:2015-07-01)

We have implemented it as a PL/pgSQL trigger procedure called *CHK_PACKED_ON*. As the name suggests, it checks if the *DURING* table is still in its packed form. It is invoked by a constraint trigger for each row after INSERT, UPDATE or DELETE statement is executed on the *DURING* table. For example, a part of check made on table *CUSTOMER_NAME_HIST* would look as follows:

```
SELECT COUNT(*) AS Error_Cnt
FROM (
        SELECT Customer_Id, Country_Code, Customer_Name, DURING
        FROM FTE.CUSTOMER_NAME_HIST AS T1
        EXCEPT
        SELECT Customer_Id, Country_Code, Customer_Name, TEMPORAL.COLLAPSE(ARRAY_AGG(DURING))
        FROM FTE.CUSTOMER_NAME_HIST AS T2
        GROUP BY Customer_Id, Country_Code
) SUB;
```

The *Error_Cnt* must be equal to zero, meaning that the current dataset of the table is equal to the collapsed form of the dataset, otherwise the check will fail.

An example of such situation is provided in the test named *Test_CHK_PACKED_ON*.

The second constraint is meant for avoiding the *contradiction* problem. What it means is that a table contains records with contradicting information. For example, the rows shown in Table 38 clearly say that Customer 1 had two names – Mari Tamm and Mari Kask – on days 2015-01-30 and 2015-01-31.

Table 38

CUSTOMER_NAME_HIST			
Customer_Id	Country_Code	Customer_Name	DURING
1	1	Mari Tamm	[2015-01-01:2015-02-01)
1	1	Mari Kask	[2015-01-30:2015-07-01)

To fix the contradiction problem, **the book** proposes to introduce the WHEN/THEN constraint that checks that if a table would be UNPACKED (each row's *During* value is a unit interval), then the primary key constraint should still be satisfied.

We have implemented it as a PL/pgSQL trigger procedure called *CHK_WHEN_UNPACKED_THEN_KEY*. As the name suggests, it checks that if the *DURING* table would be in its unpacked form then the primary key constraint would still be satisfied. It is invoked by a constraint trigger for each row after INSERT, UPDATE or DELETE statement is executed on the *DURING* table. For example, a part of check made on table *CUSTOMER_NAME_HIST* would look as follows:

```
SELECT COUNT(*) AS Error_Cnt
FROM (
        SELECT Customer_Id, Country_Code, DURING, COUNT(*) C
        FROM (
                SELECT          Customer_Id,          Country_Code,          Customer_Name,
TEMPORAL.EXPAND(ARRAY_AGG(DURING)) AS DURING
                FROM FTE.CUSTOMER_NAME_HIST
                GROUP BY Customer_Id, Country_Code, Customer_Name) SUB
        GROUP BY Customer_Id, Country_Code, DURING
        HAVING COUNT(*)>1
```

```
) SUB1;
```

The *Error_Cnt* must be equal to zero, meaning that if the current dataset of the table is unpacked using *During*, then there are no two names for a customer active on the same day. Otherwise the check will fail.

An example of such situation is provided in the file *constraint_tests.sql*, test named *Test_CHK_WHEN_UNPACKED_THEN_KEY*.

3.4.4.2 Data Integrity Requirements Across the Tables

Next, we turn to temporal requirements that are needed to check the data integrity across the tables. There were nine of them in total presented in **the book** but as some of them were logically the same (just enabled for different tables and some were combined together) we can narrow the count to four. **The book** described the constraints implementing these requirements using the database language Tutorial D. We tried to use the same logic of these Tutorial D statements to write the constraints using PL/pgSQL. Next, we will discuss all of them and provide the trigger procedures that implement the constraints.

- **REQUIREMENT R1 – If the database shows some fact to be true on day *D1*, then it must contain only one row that shows the fact.**

This constraint is meant for avoiding redundancy across the *DURING* and *SINCE* tables as we have already made sure that the *DURING* nor the *SINCE* table themselves cannot contain such information (PACKED ON and WHEN/THEN constraints in the *DURING* table and primary key constraint in the *SINCE* table). So we need to make sure that if there is a record in *SINCE* table showing some fact to be true on day *D1* then the *DURING* table must not contain a record that says the same thing and vice versa. For example, without this constraint, the situation shown in tables Table 39 and Table 40 could occur.

Table 39

CUSTOMER				
Customer_Id	Country_Code	…	Customer_Name	Customer_Name_Since
1	1	…	Mari Tamm	2015-01-31

Table 40

CUSTOMER_NAME_HIST			
Customer_Id	Country_Code	Customer_Name	DURING
1	1	Mari Tamm	[2015-01-01:2015-02-01)

These tables both say that customer 1 had the name Mari Tamm on 2015-01-31. For fixing this problem, we need to check if all the values of column *During* in the *DURING* table are less than the corresponding *Since* value in the *SINCE* table.

We have implemented it as a PL/pgSQL trigger procedure called *CHK_NO_REDUNDANCY_ACROSS_SINCE_AND_DURING*. It checks if the *Since* value of an attribute in *SINCE* table is greater than the *During* value in the *DURING* table. It is invoked by constraint triggers for each row after an INSERT, UPDATE or DELETE statement is executed on *SINCE* or *DURING* table. For example, a part of check made on tables *CUSTOMER_NAME_HIST* and *CUSTOMER* would look as follows:

```
SELECT COUNT(*) AS Error_Cnt
FROM FTE.CUSTOMER AS T_SINCE
JOIN FTE.CUSTOMER_NAME_HIST AS T_DURING
        ON T_SINCE.Customer_Id=T_DURING.Customer_Id
        AND T_SINCE.Country_Code=T_DURING.Country_Code
        AND T_SINCE.Customer_Name_Since<=TEMPORAL.END(T_DURING.DURING);
```

The *Error_Cnt* must be equal to zero, meaning that none of the values of column *Customer_Since* is less or equal to the end point of the corresponding *During* value. Otherwise the check will fail.

Examples of such situation are provided in the file *constraint_tests.sql*. The test named *Test_CHK_NO_REDUNDANCY_ACROSS_SINCE_AND_DURING* shows that the constraint works on the *SINCE* table and the test named *Test_CHK_NO_REDUNDANCY_ACROSS_DURING_AND_SINCE* does the same on the *DURING* table.

Note: this constraint also works in case the attribute's values in the SINCE and DURING tables are indeed different but the value of column Since is less or equal to the end point of the During value.

- **REQUIREMENT R2 – If the database shows some fact to be true on day *D1* and *D1+1*, then it must contain only one row that shows the fact.**

This constraint is meant for avoiding circumlocution problem across the *DURING* and *SINCE* tables as we have already made sure that the *DURING* nor the *SINCE* table themselves cannot contain such information (PACKED ON constraint in the *DURING* table and primary key constraint in the *SINCE* table). Thus, we need to make sure that if there is a record in *SINCE* table showing some fact to be true on day *D1* then the *DURING* table must not contain a record that shows the same fact being true on day *D1*+1 and vice versa. For example, without this constraint, the situation shown in Table 41 and Table 42 could occur.

Table 41

CUSTOMER				
Customer_Id	Country_Code	…	Customer_Name	Customer_Name_Since
1	1	…	Mari Tamm	2015-01-31

Table 42

CUSTOMER_NAME_HIST			
Customer_Id	Country_Code	Customer_Name	DURING
1	1	Mari Tamm	[2015-01-01:2015-01-31)

As said earlier, in case of a circumlocution problem, something is said with multiple rows when a single row would suffice. This is the case with these rows in tables *CUSTOMER* and *CUSTOMER_NAME_HIST* as well. Instead, we would better like to store this information as shown in Table 43.

Table 43

CUSTOMER				
Customer_Id	Country_Code	…	Customer_Name	Customer_Name_Since
1	1	…	Mari Tamm	2015-01-01

To avoid the circumlocution problem from occurring across the *DURING* and *SINCE* tables, we need to introduce a constraint that in case the attribute values are equal in both of the tables, the value of column *Since* cannot be equal to the day immediately after the end point of the *During* column value.

We have implemented it as a PL/pgSQL trigger procedure called *CHK_NO_CIRCUMLOCUTION_ACROSS_SINCE_AND_DURING.* It runs the abovementioned check on *SINCE* and *DURING* tables in case the values of the attribute are equal. It is invoked by constraint triggers after INSERT, UPDATE or DELETE statement is executed on *SINCE* or *DURING* table. For example, a part of check made on tables *CUSTOMER_NAME_HIST* and *CUSTOMER* would look as follows:

```
SELECT COUNT(*) AS Error_Cnt
FROM FTE.CUSTOMER AS T_SINCE
JOIN FTE.CUSTOMER_NAME_HIST AS T_DURING
        ON T_SINCE.Customer_Id=T_DURING.Customer_Id
        AND T_SINCE.Country_Code=T_DURING.Country_Code
        AND T_SINCE.Customer_Name=T_DURING.Customer_Name
        AND T_SINCE.Customer_Name_Since=TEMPORAL.NEXT_DATE(TEMPORAL.END(DURING));
```

The *Error_Cnt* must be equal to zero, meaning that if a customer has same values of column *Customer_Name* in tables *CUSTOMER* and *CUSTOMER_NAME_HIST*, then the value of column *Customer_Name_Since* is not equal to the next day after the end point of the *During* value. Otherwise the check will fail.

Examples of such situation are provided in the file *constraint_tests.sql*. The test named *Test_CHK_NO_CIRCUMLOCUTION_ACROSS_SINCE_AND_DURING* shows that the constraint works on the *SINCE* table and the test named *Test_CHK_NO_ CIRCUMLOCUTION_ACROSS_DURING_AND_SINCE* does the same on the *DURING* table.

- **REQUIREMENT R3 – If the database shows some entity to be present on day *D1*, then all of its attributes must have some value on that day as well (and vice versa).**

This constraint is meant for making sure that all the entity attributes have a value assigned throughout the time it exists in the database. For example, in our example database, the customer must have a name assigned on every day the customer is linked to the company. Therefore, if a customer is shown as linked to the company by column *During* in table *CUSTOMER_HIST* and/or column *Customer_Since* in table *CUSTOMER*, then corresponding values must be present in the column *During* of table *CUSTOMER_NAME_HIST* and/or in the column *Customer_Name_Since* of table *CUSTOMER*. Furthermore, this must be true the other way around as well – for each day when a customer name is shown to be true in a database, the customer must be linked to the company.

First, we need to make sure that the value of the attribute's *SINCE* column is greater or equal than the *SINCE* column value of the entity. We have created a CHECK constraint on the *SINCE* table that will take care of this.

Secondly, in the *SINCE* table, we only have to check the cases where the attribute's *Since* column value is greater than the entity's *Since* value – in our example it means that we make this check only if *Customer_Name_Since* value is greater than the value of *Customer_Since*. If this is the case, then the table *CUSTOMER_NAME_HIST* must contain row(s) that show customer's name values for all the days between the values of *Customer_Since* and *Customer_Name_Since*-1.

An example situation where this requirement is not met for table *CUSTOMER* is shown in Table 44 in case the table *CUSTOMER_NAME_HIST* does not contain the row shown in Table 45.

Table 44

CUSTOMER				
Customer_Id	Country_Code	Customer_Since	Customer_Name	Customer_Name_Since
1	1	2015-01-01	Mari Tamm	2015-01-03

Table 45

CUSTOMER_NAME_HIST			
Customer_Id	Country_Code	Customer_Name	DURING
1	1	Mari Mets	[2015-01-01:2015-01-03)

We have implemented it as a PL/pgSQL trigger procedure called *CHK_ATTRIBUTE_TEMPORAL_INTEGRITY*. It runs the abovementioned check on the *SINCE* table, on the attribute's *DURING* table, and on the *DURING* table of the entity. It is invoked by constraint triggers after INSERT, UPDATE, or DELETE statement is executed on the *SINCE* table, on the attribute's *DURING* table or on the *DURING* table of the entity. For example, a part of check made on tables *CUSTOMER_NAME_HIST* and *CUSTOMER* would look as follows:

```
SELECT TEMPORAL.TEMPORAL_EQUALS(
  'SELECT Customer_Id, Country_Code, DURING
  FROM FTE.CUSTOMER_HIST AS T1
  UNION ALL
  SELECT Customer_Id, Country_Code, CAST(''['''||Customer_Since ||''','' ||
TEMPORAL.PRIOR_DATE(Customer_Name_Since)||''']'' AS DATERANGE) AS DURING
  FROM FTE.CUSTOMER
  WHERE Customer_Since<Customer_Name_Since',
  'SELECT Customer_Id, Country_Code, DURING
  FROM FTE.CUSTOMER_NAME_HIST AS T2',
  NULL,
  'DURING'
) AS Res;
```

Note that this constraint uses the function TEMPORAL_EQUALS, that implements the logic of operator U_= that was described in ***the book*** *and in sections 2.4.4 and 3.3.*

The *Res* must be *TRUE*, meaning that there is a corresponding record about the customer available in the union of tables *CUSTOMER* and *CUSTOMER_HIST* for each of the records in the table *CUSTOMER_NAME_HIST*. Otherwise the check will fail.

Examples of such situation are provided in the file *constraint_tests.sql*. The test named *Test_CHK_ATTRIBUTE_TEMPORAL_INTEGRITY1* shows that the constraint works on the *SINCE* table. The test named *Test_CHK_ATTRIBUTE_TEMPORAL_INTEGRITY2* does the same on the *DURING* table of the attribute. The test named *Test_CHK_ATTRIBUTE_TEMPORAL_INTEGRITY3* does the same on the *DURING* table of the entity.

After this constraint is created on the tables, then all the data manipulation statements that are needed for each other must be executed in the same transaction – *BEGIN* and *COMMIT* commands must be used. For example, if the name of the customer changes then the following statements are needed to be executed (we assume here that the all the foreign key constraints in other tables that reference the table *CUSTOMER* are defined as *DEFERRABLE*).

1. UPDATE row in table *CUSTOMER* that shows the new name of the customer and its starting point.

2. INSERT a new row into *CUSTOMER_NAME_HIST* that shows the period the customer had his/her previous name

3. COMMIT transaction that unleashes all the deferred constraint triggers.

All of them should be executed in the same transaction. Otherwise they will fail since the trigger procedure *CHK_ATTRIBUTE_TEMPORAL_INTEGRITY* reports an error about temporal integrity.

Note: This constraint brings up (at least) two problems when used in an SQL database.

1. When a new attribute, say phone number of a customer, is added to an entity and we only have the current information (with *Since* column value equal to *today*) about this attribute for all entities, then this constraint causes the insert into table *SINCE* to fail because there is no history to put into the corresponding *DURING* table. A workaround for it would be to set the new attribute's *Since* column value equal to the *Since* value of the primary key, though this is basically telling a lie again. It means that the constraint starts to hamper evolution of database schema and one must decide as to whether to lie to have constraints in place or not to enforce this constraint in the database.

2. Another problem is that when we do not know what the phone number of some customer is, or the customer does not have a phone (nor a phone number) at all. Most SQL databases would have a *NULL* marker showing it but as said earlier, many authors consider it a logical mistake in SQL. One of the possibilities of avoiding *NULLs* is to design the database using further horizontal decomposition by creating tables in the sixth normal form (Darwen, 2003). However, it still would not help us

with our problem because this approach would just create us more tables where this constraint should added to. Thus, as a workaround, the constraint could be modified to check only the rows not containing *NULL* markers for the attributes.

- **REQUIREMENT R4 – If the database shows some fact to be true on day *D1*, and the table that stores this fact has a foreign key constraint, then there must be a row in the referenced table that shows a corresponding fact to be true on that day**

This requirement is needed for avoiding contradiction across the fact table and the referenced table. For example, if a row in *CUSTOMER_CONTRACT* or *CUSTOMER_CONTRACT_HIST* shows that customer 1 has a contract "A1" on day *D1* then there must be a record in *CUSTOMER* or *CUSTOMER_HIST* that shows that customer 1 was linked to the company on *D1*.

An example situation where this requirement is not met for table *CUSTOMER_CONTRACT* is shown in Table 46 in case the row shown in Table 47 is present in table *CUSTOMER*.

Table 46

CUSTOMER_CONTRACT			
Customer_Id	Country_Code	Contract_Nbr	Customer_Contract_Since
1	1	A1	2014-12-01

Table 47

CUSTOMER		
Customer_Id	Country_Code	Customer_Since
1	1	2015-01-01

These rows say that customer 1 had contract "A1" since 2014-12-01 while the customer is linked to the company not earlier than 2015-01-01.

The first constraint that we need is to check if the value of *Customer_Since* in table *CUSTOMER* is less or equal to the value of *Customer_Contract_Since* in table *CUSTOMER_CONTRACT*.

We have implemented it as a PL/pgSQL trigger procedure named *CHK_SINCE_IN_FK_SINCE*. It runs the abovementioned check on the *SINCE* table. It is invoked by a constraint trigger after INSERT, UPDATE, or DELETE statement is executed on the *SINCE* table or the referenced table. For example, a part of check made on tables *CUSTOMER_CONTRACT* and *CUSTOMER* would look as follows:

```
SELECT COUNT(*) AS Error_Cnt
FROM FTE.CUSTOMER AS T1
JOIN FTE.CUSTOMER_CONTRACT AS T2
   ON T1.Customer_Id=T2.Customer_Id
   AND T1.Country_Code=T2.Country_Code
   AND T1.Customer_Since>T2.Customer_Contract_Since;
```

The *Error_Cnt* must be equal to zero, meaning that there is no record in the table *CUSTOMER_CONTRACT* with the value of column *Customer_Contract_Since* less than the value of column *Customer_Since* in the table *CUSTOMER*. Otherwise the check will fail.

NB! There is a limitation that the foreign key member columns must have the same names in both of the tables. Otherwise this constraint is not created automatically and must be created manually by the user. The reason is that we could not find – based on the metadata available in PG_CATALOG and INFORMATION_SCHEMA - a solution how to determine which columns should be matched in the JOIN condition when making this check. So this is done currently based on the equal column names.

An example of such situation is provided in the file *constraint_tests.sql*. The test named *Test_CHK_SINCE_IN_FK_SINCE* shows that the constraint works on the *SINCE* table that has a foreign key referencing a table with enabled temporal support.

The second constraint is for checking the temporal integrity across the fact and its foreign key reference. For example, if the table *CUSTOMER_CONTRACT_HIST* contains a row as shown in Table 48, then there must be a row in table *CUSTOMER* or *CUSTOMER_HIST* showing that customer 1 was linked to the company during this period.

Table 48

CUSTOMER_CONTRACT_HIST			
Customer_Id	Country_Code	Contract_Nbr	DURING
1	1	A1	[2014-12-01, 2015-04-01)

We have implemented it as a PL/pgSQL trigger procedure named *CHK_FK_TEMPORAL_INTEGRITY*. It runs the abovementioned check on the *DURING* table and on the referenced *SINCE* table and its *DURING* table. It is invoked by constraint triggers after INSERT, UPDATE or DELETE statement is executed on the *DURING* table or on the referenced *SINCE* table and its *DURING* table. For example, a part of check made on tables *CUSTOMER_CONTRACT* and *CUSTOMER* would look as follows:

```
SELECT COUNT(*) AS Error_Cnt
FROM ((
   SELECT SUB1.*
   FROM (
      SELECT Customer_Id, Country_Code, TEMPORAL.EXPAND(ARRAY_AGG(DURING)) AS DURING
      FROM (
         SELECT Customer_Id, Country_Code, DURING
         FROM FTE.CUSTOMER_CONTRACT_HIST AS T1
      ) AS QUERY1
      GROUP BY Customer_Id, Country_Code
   ) SUB1
) EXCEPT (
   SELECT SUB2.*
   FROM (
      SELECT Customer_Id, Country_Code, TEMPORAL.EXPAND(ARRAY_AGG(DURING)) AS DURING
      FROM (
         SELECT Customer_Id, Country_Code, DURING
         FROM FTE.CUSTOMER_HIST AS T2
         UNION ALL
         SELECT  Customer_Id,  Country_Code,  CAST('['||Customer_Since  ||',  INFINITY]'  AS
DATERANGE) AS DURING
         FROM FTE.CUSTOMER
   ) AS QUERY2
   GROUP BY Customer_Id, Country_Code
   ) SUB2
)
) RES;
```

Note that this constraint uses the logic of function TEMPORAL_MINUS, that implements the logic of operator U_MINUS that was described in ***the book*** *and in sections 2.4.4 and 3.3. We could not use the function TEMPORAL_MINUS itself due to some technical difficulties.*

The *Error_Cnt* must be equal to zero, meaning that there is no record in the table *CUSTOMER_CONTRACT_HIST* that does not have a corresponding record in the union of tables *CUSTOMER and CUSTOMER_HIST*. Otherwise the check will fail.

An example of such situation is provided in the file *constraint_tests.sql*. The test named Test_CHK_INTEGRITY_IN_FK_TABLES shows that the constraint works on table *CUSTOMER_CONTRACT_HIST*.

3.4.5 Views for Providing Shorthand for the Users

After the constraints are in place, a couple of views are created. They provide the user a level of abstraction for accessing the information in a more convenient way.

First, the original horizontal decomposition will be undone. What we mean by the original horizontal decomposition, is that the historical information from the initial semi-temporal *SINCE* table was split into a set of *DURING* tables, each for every attribute. The *TEMPORALIZE* function will create a view for each attribute that will combine the attribute's historical and current information. This is done with the help of the UNION ALL operator. We take all the rows from the attribute's *DURING* table, then all the information about this attribute from the *SINCE* table, and then the two datasets are combined together. Furthermore, the value of the attribute's *Since* column is transformed into an interval with begin point equal to the *Since* column value and the end point is set to *'infinity'*. For example, the view that is created for accessing the full information of customer names will combine all the rows from *CUSTOMER_NAME_HIST* table plus current customer name values from the *CUSTOMER* table. To better illustrate this example, the result of view *CUSTOMER_NAME_VW* is shown in Table 49 in case the tables *CUSTOMER* and *CUSTOMER_NAME_HIST* contain the rows as shown in Table 44 and Table 45 correspondingly.

Table 49

CUSTOMER_NAME_VW			
Customer_Id	Country_Code	Customer_Name	DURING
1	1	Mari Tamm	[2015-01-03:INFINITY)
1	1	Mari Mets	[2015-01-01:2015-01-03)

We also provide an example definition of the view *CUSTOMER_NAME_VW*:

```
CREATE VIEW FTE.CUSTOMER_NAME_VW AS (
SELECT customer_id, country_code, customer_name,
           (('['::text || customer_name_since) || ',INFINITY)'::text)::daterange AS during
```

```
FROM customer
UNION ALL
SELECT customer_id, country_code, customer_name, during
FROM customer_name_hist
);
```

Finally, *TEMPORALIZE* will create one more view that is replaced every time temporal support is enabled on some attribute. This view will undo the original vertical decomposition, meaning that all the views of different attributes are joined together, providing the user a possibility to see all the information about all the attributes in one place. Furthermore, first all the views are unpacked and then joined together using the unit interval values. After the joining is done, only one *During* column is included into the final column list and the data is packed on this column. This will give an overview of the history, when some combination of attribute values was valid for each entity. For example, the view *CUSTOMER_FULL_VW* will contain all the information from each of the attribute views. To better illustrate this example, the result of view *CUSTOMER_FULL_VW* is shown in Table 52 in case the views *CUSTOMER_VW*, *CUSTOMER_NAME_VW*, and *CUSTOMER_SEGMENT_VW* contain the rows as shown in Table 50, Table 49, and Table 51 correspondingly.

Table 50

CUSTOMER_VW		
Customer_Id	Country_Code	DURING
1	1	[2015-01-01:INFINITY)

Table 51

CUSTOMER_SEGMENT_VW			
Customer_Id	Country_Code	Customer_Segment_Code	DURING
1	1	2	[2015-01-04:INFINITY)
1	1	1	[2015-01-01:2015-01-04)

Table 52

CUSTOMER_FULL_VW				
Customer_Id	Country_Code	Customer_Name	Customer_Segment_Code	DURING

CUSTOMER_FULL_VW				
Customer_Id	Country_Code	Customer_Name	Customer_Segment_Code	DURING
1	1	Mari Tamm	2	[2015-01-04:INFINITY)
1	1	Mari Tamm	1	[2015-03-01:2015-01-04)
1	1	Mari Mets	1	[2015-01-01:2015-01-03)

We also provide an example definition of the view *CUSTOMER_FULL_VW*:

```
CREATE VIEW FTE.CUSTOMER_FULL_VW AS (
SELECT t1.customer_id,t1.country_code,t1.customer_name,t2.customer_segment_code,
    temporal.collapse(array_agg(CASE WHEN temporal."end"(t1.during) = 'now'::text::date THEN
(('['::text || temporal.begin(t1.during)) || ',INFINITY)'::text)::daterange ELSE t1.during
END)) AS during
FROM ( SELECT customer_name_vw.customer_id,
            customer_name_vw.country_code,
            customer_name_vw.customer_name,
            temporal.expand(array_agg(customer_name_vw.during)) AS during
           FROM customer_name_vw
          GROUP BY customer_name_vw.customer_id, customer_name_vw.country_code,
customer_name_vw.customer_name
) t1
JOIN ( SELECT customer_segment_vw.customer_id,
            customer_segment_vw.country_code,
            customer_segment_vw.customer_segment_code,
            temporal.expand(array_agg(customer_segment_vw.during)) AS during
           FROM customer_segment_vw
          GROUP BY customer_segment_vw.customer_id, customer_segment_vw.country_code,
customer_segment_vw.customer_segment_code
) t2
         ON t1.customer_id = t2.customer_id
         AND t1.country_code = t2.country_code
         AND t1.during = t2.during
JOIN ( SELECT customer_vw.customer_id,
       customer_vw.country_code,
       temporal.expand(array_agg(customer_vw.during)) AS during
      FROM customer_vw
     GROUP BY customer_vw.customer_id, customer_vw.country_code
) t3
         ON t2.customer_id = t3.customer_id
         AND t2.country_code = t3.country_code
         AND t2.during = t3.during
  GROUP BY t1.customer_id, t1.country_code, t1.customer_name, t2.customer_segment_code;);
```

3.4.6 Changing the Data in the Database

Writing queries to maintain the data in the temporal database can quickly become a complex task. There are a lot of things one needs to keep in mind when making changes in the data and the constraints described earlier help us to guard against some of the problems that could occur without them. Changing the data should not be thought of as inserting, updating, and deleting rows in the database. Rather it should be thought of as adding, modifying, and removing propositions. In the database with the proposed design, changing one proposition

very often results in changing multiple rows in the database. For example, if the customer name changes on day *D1* then there must be the following statements executed:

1. UPDATE row in table *CUSTOMER* that shows the new name of the customer and its starting point.

2. INSERT a new row into *CUSTOMER_NAME_HIST* that shows the period the customer had his/her previous name

3. COMMIT transaction that unleashes all the deferred constraint triggers.

The views that were created should provide a possibility to make the data changes a bit easier for users. PostgreSQL database supports the creation of INSTEAD OF triggers on views that could execute the needed statements on the background. For example, for the abovementioned change of customer name, the user should only execute an UPDATE statement on the view *CUSTOMER_NAME_VW* and then the data manipulation statements that are needed for persisting this change would be executed automatically. The automatic creation of such triggers is outside the scope of this thesis because it is clearly a complex assignment with a lot of things to analyse and should not be done in a rush. However, it would be an excellent subject for another thesis to provide a comprehensive set of such INSTEAD OF triggers.

3.4.7 The Performance

We have run some performance tests on our example database to understand a bit how the proposed approach is performing. The approach requires many row-level constraint triggers on the tables in case of our current implementation. Row-level means that trigger procedures are executed for each inserted/updated/deleted row. It is not a surprise that the performance suffers because of that. Furthermore, the bigger the volume of the data in the database is, the slower the data manipulation statements are. If we look at the table *CUSTOMER*, then we can see that there are *nine* constraint triggers in total that are executed when a row gets inserted, updated, or deleted in it.

We have created a simple java program that can be used for generating insert statements for testing the performance. It is named *Data generator.jar* and it is located in https://github.com/SanderLaasik/temporal. The user needs to fill in the output filename and

the count of rows to be inserted into each of the tables. Currently it creates statements for tables *CUSTOMER*, *CUSTOMER_HIST*, and *CUSTOMER_CONTRACT*.

The server we used for making these performance tests had the following technical characteristics: Virtual machine QEMU Virtual CPU version, 811 GB HDD, 40 GB RAM, 15 virtual CPUs, CentOS 6.4.

Our first test is to insert one row into each of the tables *CUSTOMER*, *CUSTOMER_HIST*, and *CUSTOMER_CONTRACT* when they are empty. These three inserts took 91 milliseconds in total, thus inserting into an empty table is with quite a good performance time wise.

Our second test is to measure the performance with inserting initial 500 rows into tables *CUSTOMER*, *CUSTOMER_HIST*, and *CUSTOMER_CONTRACT* – 1500 rows in total. Execution of these insert statements took us 200 milliseconds or 3.3 minutes to execute so the performance is considerably bad. Most probably there are many possibilities to improve this by, for example, creating indexes on the tables. The biggest win would come from improving the constraint triggers and the functions (especially the EXPAND and COLLAPSE functions) they use. After the 1500 rows had been added into the tables, the insertion of one more row into each table took 1452 milliseconds. Thus the execution time had increased almost 16 times. Finally we deleted all the rows form these three tables and it took us 13531 milliseconds.

Next, we performed the insertion of 10000 rows into each of the tables. Insertion of these rows took us over 18 hours. Thus, the performance has gone considerably worse. Inserting one additional row into each of the table1es containing 10000 rows took 15 seconds.

Data warehouses feature tables with millions of rows and loading data to the tables is one of the main type of operations in them. Thus, the current implementation is not suitable for actual data warehouse environment.

Using the combining views can be another bottleneck performance wise. The performance of both the attribute's view and the full view of the entity must be further analysed.

The attribute's view uses the UNION ALL operator that has better performance than the regular UNION operator does since it does not remove duplicates from the result. As we already know, the *DURING* table and the *SINCE* table cannot contain redundant information.

Therefore, there is no need to use the UNION operator. With big data volumes, though, the attribute's view can still become slow.

The entity's full view joins all the attribute views together so there should be the needed indexes in place on foreign keys that make the JOIN operator to perform faster. In case of our approach the history tables have primary keys that overlap with foreign keys that reference to the *since* table. Because PostgreSQL creates automatically indexes for the primary keys there are already indexes for the foreign keys as well. Furthermore, PostgreSQL has a feature called *join removal* that can help us to skip some joins when any of the attributes are not mentioned in the SELECT clause and the overall content would not change because of that (Postgres Wiki, 2014).

4. Summary

Maintaining the temporal data has long been with low priority for the SQL standard. Thus, the users of SQL databases have been struggling with designing and developing databases that are meant for storing temporal data. The aim of this thesis was to implement as much as possible the ideas provided in the book "*Temporal Data and the Relational Model*" by C.J. Date, H. Darwen and N.A. Lorenzos, using PostgreSQL 9.3 DBMS. A successful implementation in PostgreSQL would give us confidence, but not a guarantee of course, that the approach described in this book can also be implemented in other SQL database management systems.

All the developed source code is made accessible to all counterparties who have interest in this work via shared GitHub repository located at https://github.com/SanderLaasik/temporal. The most important results are that most of the operators described in the book were implemented successfully for the data type *date*, using PL/pgSQL functions. Additionally, function is provided that automatically creates:

- a history table and a combining view for the attribute that is wished to be temporalized;
- a set of constraint triggers that implement the constraints protect the data from several temporal integrity issues;
- aview that combines all the attributes of an entity together, providing a comprehensive picture about the entity and its attributes over time;

We can confidently say that it is possible to successfully implement the approach provided in the book in PostgreSQL. However, because the implementation relies extensively on deferred constraint triggers that are specific to PostgreSQL, we cannot claim that it is universally possible in case of any SQL DBMS. Furthermore, the created triggers did not have good performance characteristics. Thus, they can be considered as a "proof of concept" and further research is needed about improving the performance.

The implemented approach avoids the use of a *NOW* marker when storing the data, thus avoiding "*the problem of the moving point now*". Furthermore, the data is logically

decomposed both horizontally and vertically, meaning that the historical and current data are stored in different tables, and there is a separate history table for each of the attributes of an entity. The performance of the DML statements is quite bad thanks to the complex constraint triggers that are executed at the end of transactions for each modified row. Thus, the possibilities on how to make the constraints to perform better must be thoroughly analysed. In addition, the usage of the combining views can become a bottleneck soon if the creation of additional indexes is not added to the functionality and database statistics is not regularly refreshed.

Future work should also include the following.

1. Test the triggers in a situation when there are lot of parallel data modifications in the database.

2. To provide a support for other data types, for example *timestamp* with different precisions.

3. To provide a set of *INSTEAD OF* triggers on top of the combining views that make updating of data through the generated views possible. These triggers should be automatically generated. In the current implementation, data modifications in a *SINCE* table do not cause automatic addition of new rows to the history tables.

4. Implement the *USING* syntax that would make creation of many of the operators a lot easier and clearer, should be added.

5. Investigate the use of the approach in case of schema evolution where one wants to add columns to tables or remove columns from tables.

6. Investigate how to deal with missing information in case of this approach.

Kokkuvõte

Ajaandmete haldamine on pikka aega olnud SQL standardi jaoks madala prioriteediga, mistõttu on SQL-andmebaaside kasutajad pidanud ajaandmete hoidmise jaoks mõeldud andmebaaside disainimise ja loomisega palju vaeva nägema. Käesoleva töö eesmärk oli proovida võimalikult palju realiseerida ideid, mis pakuti välja C.J Date'i, H.Darweni ja N.A. Lorezose kirjutatud raamatus „*Temporal Data and the Relational Model*", kasutades PostgreSQL 9.3 andmebaasisüsteemi. Õnnestunud realisatsioon PostgreSQL andmebaasisüsteemis annaks meile kindluse, kuid mitte garantii, et antud põhimõtteid on võimalik realiseerida ka teistes PostgreSQL andmebaasisüsteemides.

Kogu arendatud lähtekood on kõigile huvilistele kättesaadav GitHub'i koodihoidla kaudu, millele saab ligi järgnevat linki kasutades: https://github.com/SanderLaasik/temporal. Käesoleva töö põhitulemuseks on, et enamik raamatus kirjeldatud operaatorid said PL/pgSQL funktsioone kasutades *date* andmetüübi jaoks realiseeritud. Lisaks loodi abifunktsioon, mis loob:

- ajalise toe vajadusega atribuudi jaoks ajalootabeli ja vaate, mis ühendab selle atribuudi ajaloolised ja praegused andmed
- hulk kitsendustrigereid, mille realiseeritavad kitsendused kaitsevad andmeid erinevate ajalise iseloomuga kvaliteediprobleemide eest
- vaate, mis ühendab antud olemi kõik atribuudid kokku, pakkudes ülevaatliku pildi selle olemi ja tema atribuutide ajaloo kohta.

Me võime kindlalt väita, et raamatus pakutud põhimõtteid on võimalik PostgreSQL andmebaasisüsteemis realiseerida. Samas, kuna antud realisatsioon sõltub tugevasti kitsendustrigeritest, mis on PostgreSQL-i spetsiifilised, ei saa me väita, et see on universaalselt võimalik kõigi SQL-andmebaasisüsteemide puhul. Lisaks on loodud trigeritel üsna halb jõudlus, mistõttu neid tuleks võtta kui antud lähenemise võimalikkuse tõestamist ning jõudluse parendamiseks on vajalik uurimistööga jätkata.

Antud lähenemine väldib andmete hoidmisel *NOW* markeri kasutamist, vältides seeläbi ka probleeme, mis sellega tavaliselt kaasnevad. Andmed on nii horisontaalselt kui ka vertikaalselt tükeldatud, mis tähendab, et ajaloolisi ja praegusi andmeid hoitakse eraldi tabelites, ning iga atribuudi ajalooandmete hoidmiseks on eraldi tabel. Andmete muutmise lausete jõudlus on üsna halb, kuna keerukaid kitsendustrigereid käivitatakse iga sisestatud, muudetud või kustutatud rea kohta. Seega peaks põhjalikult analüüsima erinevaid võimalusi, kuidas nimetatud kitsendusi parema jõudlusega tööle saada. Lisaks võib üheks pudelikaelaks osutuda loodud vaadete kasutamine, kui funktsionaalsusele ei lisata täiendavate indeksite loomist ning ei toimu regulaarset statistika kogumist.

Tulevaste arenduste hulka peaks samuti kuuluma järgnev.

1. Testida loodud trigereid olukorras kus andmebaasis on palju samaaegseid andmemuudatusi.

2. Luua tugi vähemalt andmetüübile *timestamp* ning selle erinevatele täpsusastmetele

3. Pakkuda loodud vaadetele hulk *INSTEAD OF* trigereid, mis võimaldaksid vaadete kaudu andmebaasis andmemuudatusi teha. Sellised trigerid tuleks automaatselt genereerida. Praeguses realisatsioonis ei põhjusta andmete muutmine hetkeversiooni sisaldavas tabelis uute ridade automaatset lisamist ajalooliste andmete tabelitesse.

4. Realiseerida *USING* süntaks, mis muudaks paljude operaatorite loomise märksa selgemaks ning lihtsamaks.

5. Uurida, kuidas selle lähenemise kontekstis peaks toimuma skeemi evolutsioon kui tabelisse lisatakse uusi veerge või tabelist eemaldatakse veerge.

6. Uurida, kuidas tulla selle lähenemise kontekstis toime puuduvate andmetega.

References

1. Anchor Modelling [WWW] http://www.anchormodeling.com (05.04.2015)

2. Darwen H., 2003 *How To Handle Missing Information Without Using NULL*.

3. Date C.J., Darwen H., Lorenzos N.A., 2002. *Temporal Data and the Relational Model: a detailed investigation into the application of interval and relation theory to the problem of temporal database management.* Morgan Kaufmann.

4. Darwen H., 2013. *Temporal Data and the Relational Model*. Warwick University.

5. Date, C.J., 2006. *The relational database dictionary. A comprehensive glossary of relational terms and concepts, with illustrative examples*. O'Reilly.

6. DB-Engines Ranking, 2015. May, 2015. [WWW] http://db-engines.com/en/ranking (09.05.2015)

7. Gulutzan, P., Pelzer, T., 1999. *SQL-99 Complete, Really*. Miller Freeman.

8. Hevner A.R, Salvatore T.M, Jinsoo P, Sudha R, 2004. *Design Science in Information Systems Research*. MIS Quarterly Vol. 28 No. 1, pp. 75-105/March 2004

9. Kulkarni K., Michels J.-E., 2012. *Temporal Features in SQL:2011*. SIGMOD Record, September 2012 (Vol. 41, No. 3)

10. Oracle Database documentation [WWW] http://www.oracle.com/technetwork/documentation/index-087067.html (05.04.2015)

11. PostgreSQL 9.3.1 Documentation, 2015. [WWW] http://www.postgresql.org/docs/9.3/interactive/index.html (05.09.2015)

12. PostgreSQL Wiki, 2014. [WWW] https://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.0#Join_Removal (12.05.2015)

13. Potter, E., 2013. A Comparison Between Anchor Modeling and Oracle Workspace Manager in Managing Temporal Data in SQL Databases. Bachelor thesis. TUT Institute of Informatics. (in Estonian)

14. Rönnbäck, L., Regardt, O., Bergholtz, M., Johannesson, P., Wohed, P., 2010. Anchor modeling —agile information modeling in evolving data environments. Data & Knowl. Eng. 69, 1229–1253.

15. Saal, E., 2015. A Generator for Generating Implementation of Anchor Modelling Models in PostgreSQL. Master thesis. TUT Institute of Informatics. (in Estonian)

16. Separation of Concerns [WWW] http://trese.cs.utwente.nl/taosad/separation_of_concerns.htm (13.05.2015)

17. Teradata Magazine – Busting the Pricing Myth [WWW], 2011 http://www.teradatamagazine.com/v11n01/Viewpoints/Busting-the-Pricing-Myth/ (12.05.2015)

18. Teradata Temporal Table Support [WWW] http://www.info.teradata.com/HTMLPubs/DB_TTU_13_10/index.html#page/SQL_Reference/B035_1182_109A/title.01.2.html (11.05.2015)

19. TimeDB - A Bitemporal Relational DBMS [WWW] http://www.timeconsult.com/Software/AboutTimeDB1.0.html (14.05.2015)

20. Webopedia [WWW] http://www.webopedia.com/ (22.03.2015)