FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Native Temporal Relation Database for Haskell

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Lukas Maczejka
Matrikelnummer 0426085

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: O.Univ.Prof. Dipl.-Ing. Dr.techn. A. U. Frank

Wien, __,__,____ _____ _____

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

A Native Temporal Relation Database for Haskell

Lukas Maczejka

2009-2010

Matr.# 0426085
Correspondence ID 4658
lukas.maczejka@compucare.at

December 6, 2010

Abstract

Representation of time-specific data is a widely discussed issue in modern database management systems. After a brief introduction to database basics, this work explains the currently accepted concepts related to time-specific terminology, differences in representation of time, and the storage of temporal data in detail. Furthermore, the basics behind functional programming in general and Haskell in particular are briefly discussed. Using both technologies mentioned above, a native temporal database management system for Haskell is introduced. The prototype developed from these specifications is described both from the developer's and the user's point of view. Finally, a critical evaluation of the proposed system and the prototype is given, and possible further work is discussed.

Contents

1 Introduction

This section describes the goals this work is trying to achieve, as well as the motivation to do so. Existing restrictions are discussed and a short overview of relevant fields of study is given. Finally, this section will provide a brief description of the structure of this work.

1.1 Goals

Existing software modules, developed in Haskell, represent and store data as binary relations. The current method of data storage used by these modules is inadequate considering efficiency and the scope of data. This work aims to propose a new, better method of storing relations in Haskell. The goal is not only to increase ease of access and efficiency, but to further expand the scope of such a system by the time dimension, allowing the state of data to be stored and retrieved for any given point in the past or future as well as to restore the state of the entire database to any point in history. A prototype is developed to show the application of the proposed methodology and algorithms and to evaluate them considering the task at hand. This prototype is meant for real-life application as a method of data management for existing software modules.

1.2 Previous Work

This work covers large portions of various fields of study, aiming to apply the most appropriate methods to a very specific problem. The first part will concentrate on highlighting the basic concepts behind the terminology used in this paper in accordance to its usage in previous works. An overview of historic and current database design and development paradigms will be given, followed by a more specific description of the concepts behind temporal databases in general and time-specific data in particular. Since the prototype is to be developed in Haskell, this work will also cover the basic concepts behind functional programming in general and Haskell in particular.

1.3 Development

Since a large part of this work consists of the development of a working prototype, the methods used to develop this application will be discussed in detail. Parts of the text will be concerned not with the particular results of research, but the problems and difficulties in achieving these results.

1.4 Requirements

The proposed system must, at least, provide the following functionalities.

- **Haskell.** The system is to be written in Haskell for seamless integration into existing software. Further restrictions concerning system environment and third-party software might also apply and will be discussed in detail at the time of their occurrence.
- **Relations.** The database must be able to store various types of data within a set of defined relations, forming a database.
- **Database Access.** Haskell functions for access and manipulation of the database must be provided. This includes basic data manipulation as well as more complex database queries. To provide a maximum ease of usage, access functionality should resemble well known database query languages (SQL).
- **Temporalisation.** The system must support the time dimension and be able to retrieve its contents for any given point in time, if data exists in the database which is valid for that particular point in time. Additionally, the database must be able to restore itself to any state it had at any point in the past since its initialisation.
- **Extensibility.** The goal is to develop the database system as flexible as possible, to make future enhancements as simple as possible.

1.5 Restrictions

Different restrictions apply concerning the desired results, specific technologies to be used, and the scope of this work.

- **Specialisation.** The proposed system must fulfill all the requirements, but must do so in an efficient manner. Existing database technologies are to be considered and, if applicable, integrated into the design. However, functionality beyond the scope of this work is not to be implemented, even though the system might provide possibilities for further enhancement.
- **Binary relations.** A database system with the ability to manage binary relations is not necessarily a relational database management system. The distinction between relation database and relational database is important in this context, though binary relations might not suffice for describing all the necessary data structures, as will be evident in later sections.
- **Query language.** Access to the database management system's functionality is provided with specifically designed Haskell functions. No established or newly developed query language is to be implemented. However, to make access to these functions as simple as possible, statements can be expressed closely resembling SQL-statements.
- **Performance.** Though important in any database system, performance will initially only be a secondary goal. Performance issues will be discussed during the remainder of this work, however, optimising the system's performance will be left for future development.

1.6 Structure

Section 2 will cover the state of the art in database development, relevant terminology, and current database design paradigms. Section 3 is concerned with data persistence and temporal data as well as the terminology and classification of time-specific database

systems. Section 4 will give an overview of functional programming in general and Haskell in particular, as well as currently existing technologies for database access in Haskell.

After evaluating the available technologies, the most appropriate are chosen to specify a database system meeting all the requirements for this work. This will be covered in section 5. The system is then developed in Haskell, detailing the specifics of the implementation. Finally, in section 7, the resulting prototype is evaluated against the initial requirements and possible further development is discussed.

2 Database Design

This section will give an overview of accepted database models, design, and application. The applicability of existing database systems to the problem at hand will be evaluated.

2.1 Database Systems

A database system is nothing more than a computerised record-keeping system (Date 2004). The goal is to store and retrieve different types of information as efficiently as possible. The main benefits of storing information in a database system can be summarised as follows (Date 2004):

- Sharing of data
- Reduction of redundancy
- Avoiding of Inconsistencies
- Transaction Support
- Maintaining Integrity
- Security
- Balancing Requirements
- Enforcing of Standards
- Data Independence

To explain the benefits of database systems in more detail would go beyond the scope of this work. However, there is ample literature available for the interested reader.

The most widely used database systems today are relational database management systems (Hellerstein, Stonebraker & Hamilton 2007). Data is stored in tables of records of various formats, representing relations between such records.

In almost all commonly used database systems two types of information are stored separately. On one hand there is the data itself, containing raw, coherent information. On the other hand relations are defined as links between parts of these coherent data sets. Thus, related data can be stored efficiently.

2.2 Database Management Systems

A database management system is the layer of software between the physical database (the data physically stored) and the user (Date 2004). All data access and manipulation are handled by the Database Management System (DBMS). The DBMS acts as an abstraction layer between actual data and processed data, shielding the user from hardware-level details.

What is generally referred to when talking about databases is the database management system, not the database itself. Thus, the goal of this work is not only to create a relation database for Haskell (the model), but also a DBMS to manage such a database in Haskell.

2.3 Database Models

The vast majority of research into database systems in the past 30 years has been based on the **relational model**. As a result, relational database systems (and SQL Systems in general) have come to dominate the market (Date 2004).

Other approaches include **inverted lists**, **hierarchic**, and **network** systems, as well as **object**, **object-relational**, **multi-dimensional**, **logic-based** and **semistructured** approaches.

C.F. Codd, the inventor of the relational model (Date 2004), defines a data model as the combination of three components (Codd 1980):

- A collection of data structure types
- A collection of operators or inferencing rules

- A collection of general integrity rules

The following sections will briefly describe the most common database models.

2.3.1 The Relational Model

Relational Database Management Systems (RDBMS) are based on the relational model of data first put forward by C.F. Codd (Codd 1970), which can be described by the following three aspects (Date 2004):

Structural aspect. The data is perceived by the user as nothing but tables.

Integrity aspect. Those tables satisfy certain integrity constraints.

Manipulative aspect. Operators for manipulating tables derive tables from tables.

Tables, however, are only the logical structure of the database. The actual physical storage can be managed by the RDBMS in any way, provided it can map it to the logical table structure. All Information is stored in column positions in rows in tables (as a collection of relational variables, or *relvars*), on the logical level there are no pointers or any other additional information.

In the context of relational databases, a *relation* is nothing more than the mathematical term for a table of specific kind (Date 2004).

In 1970, Codd suggested the internal organisation of large databases must be transparent to its users. Tree-structured files or network data models are inadequate (Codd 1970). Nowadays, the overwhelming majority of database systems are based on the model put forward by Codd, reinforcing his views on the subject.

There is no way around the relational model in almost all modern databases, partly due to superior performance and the technology's unrivaled market position. It also needs to be considered as a solution for the problem at hand, be it at a slightly modified point of view. However, in order to choose the ideal technology, a closer look at other models is required.

2.3.2 Object Databases

According to Atkinson et. al. an object oriented database system (OODBS) must possess all of the following thirteen features to be classified as both a database management system and as an object-oriented system (Atkinson, Bancilhon, DeWitt, Dittrich, Maier & Zdonik 1989):

Complex Objects. The system must support complex objects by applying constructors such as tuples, sets or lists to simple objects such as integers or strings.

Object Identity. An object has an existence which is independent of its value. Thus, two objects can be considered as equal having the same value, or identical being the same object. Objects can be shared, and any changes to one object are seen by any other related object.

Encapsulation. An object consists of an interface and an implementation part. The interface part is the set of operations that can be performed by the object and therefore the only visible part of the object. The implementation part consists of the data store and the procedure part. The object encapsulates both program and data.

Types and Classes. Types are abstract data types, summarising the common features of a set of objects, while classes also contain an object warehouse providing the option of manipulating all objects of one class. Same as in the world of object oriented programming, an OODBS can support either approach.

Class or Type Hierarchies. The keyword here is inheritance. Types should be able to inherit from other types. Which style of inheritance is supported and to which degree it is supported is not relevant to the definition.

Overriding, overloading and late binding. Objects of different types should be able to share methods of the same name (overriding). This results in different programs with a single name (overloading). To provide this functionality, operation names cannot be mapped to programs at compile time, but must be resolved at runtime (late binding).

Computational Completeness. The Database system must be able to express any

computable function.

Extensibility. There has to be a way to define new types, and the handling of these types must be no different than that of predefined types.

Persistence. Data must implicitly survive the execution of a process in order to eventually use it in another process.

Secondary storage management. This includes performance features such as index management, data clustering or query optimisation.

Concurrency. Multiple users must be able to concurrently interact with the system.

Recovery. The system should be able to recover from software or hardware failures, being able to return to some coherent state of data.

Ad Hoc Query Facility. The system must provide the user with an easy way to ask simple queries. This does not necessarily mean the system needs to provide a fully functional query language.

In conclusion, an OODBS must contain the five features attributed to database systems and the eight features attributed to object oriented languages mentioned above. For further detail and a list of optional features see (Atkinson et al. 1989).

C.J. Date (Date 2004) argues that

> The sole good idea of object systems in general is **proper data type support**; everything else - including in particular the notion of user-defined operators - follows from that one idea. But that idea is hardly new!

This again highlights the dominance of the relational model, partly due to the strong opinions within the database community. While possibly a fitting solution for specific database tasks, the characteristics of object databases do not overlap with the requirements of the prototype to be developed.

2.3.3 Object/Relational Databases - The Third Generation

Relational systems should evolve to incorporate the (good) features of objects (Date 2004).

As a counterproposal to the *Object-Oriented Database System Manifesto* (Atkinson et al. 1989) (see section 2.3.2), the Committee for Advanced DBMS Function published the *Third-Generation Database Management System Manifesto* (Stonebraker, Rowe, Lindsay, Gray, Carey, Brodie, Bernstein & Beech 1990), describing an Object/Relational Database Model.

According to Stonebraker et. al. the three basic principles defining a third generation DBMS are (Stonebraker et al. 1990):

- Support for richer object structures and rules
- Sub-summation of second generation DBMSs (Defined as the current collection of relational systems)
- Third generation DBMSs must be open to other subsystems

What this means in detail is explained by another set of thirteen propositions:

Rich type system. A third generation DBMS must support various type constructors, an abstract data type system, functions as a type, and a recursive composition of type constructors.

Inheritance. The system must allow types to be organised in a multiple inheritance hierarchy.

Functions and encapsulation. Encapsulated functions provide performance and structuring benefits. Functions should be written in a higher level language.

Optional system-assigned record IDs. If no user-defined primary key is available, the system should provide the record with a generated unique identifier.

Rules, not tied to specific objects. Rules should not be implemented in functions, but be enforced by the DBMS itself.

Navigation only as a last resort. The expressive power of a query language must be present in every programmatic interface and it should be used for essentially all access to data.

Automatically and user-maintained collections. There should be two ways to specify collections. The first being *extensionally*, through a collection of pointers, the second being *intensionally*, or automatic, through expressions.

Updateable views. Support for update-able virtual collections (views) is required.

Hidden performance modules. Data clustering, indices and other performance-related indicators have little to do with data models and must be hidden from the user.

Multiple Language Support. A third generation DBMS must be accessible from multiple higher level languages.

Persistence regardless of type system. Compiler extensions and a complex run time system will ensure programming language and DBMS independence.

SQL support. SQL is the universal way of expressing queries, and therefore the system must support SQL or an extended version of SQL.

Queries and results as the lowest level of communication. Queries, even if expressed in a function, must be the lowest level of communication. Remote procedure calls and SQL queries provide an appropriate interface.

In conclusion, Object/Relational databases are nothing more than relational systems that allow users to define their own types (Date 2004).

2.3.4 The Logic-Based Approach

Another completely different approach are database systems based on logic. New facts can be derived from facts already explicitly introduced to the database (Gallaire, Minker & Nicolas 1984). The distinction between traditional database systems and the alternative, logic-based systems can be described as the difference between **model-theoretic** and **proof-theoretic** systems (Reiter 1984). The difference can be explained as follows (Date 2004):

Model-theoretic database systems. The database is a set of explicit relations, each containing a set of explicit tuples. Executing a query means evaluating an expression of those relations and tuples.

Proof-theoretic database systems. The database is a set of axioms. Executing a query means proving a formula as the logical consequence of the database, proving it as a theorem.

There is a number of features inherent to deductive database management systems (DBMSs implementing the proof-theoretic approach) that provide certain advantages (Date 2004):

Representational uniformity. Everything in a database language, from values and relations to queries and integrity constraints can be represented in the same uniform way.

Operational uniformity. Query optimisation, constraint enforcement, database design, and other seemingly independent problems can be tackled on the same basis.

Semantic modeling. The basic model can be extended by a variety of semantic features.

Extended application. Certain issues, such as disjunctive information, that model-theoretic systems have trouble coping with can be dealt with by proof-theoretic systems.

Logic-based database systems have the possibility of bridging the gap between databases and general-purpose programming languages (Date 2004). Contrary to the "sub-language" approach taken by SQL systems, deductive databases could be seamlessly integrated into

programming languages, no more distinction made between shared data and data local to the program.

2.3.5 Semistructured Data and XML

With the advent of the world wide web and hypertext came the need for semantically structured documents. The most widespread approach to this problem currently is XML (Extensible Markup Language), with a variety of added features from definition to transformation available. This work will not delve deeper into the features of XML, however, even if such approaches to database systems are viewed as an exercise in futility by some (Date 2004), its use as a database system will be briefly discussed in this section.

There are different methods of representing data structured in an XML document in database terms, from *shredding* and storing the data in a conventional database to storing the entire document as part of an attribute-value tuple (Date 2004). For this work, native XML databases are of interest.

Though it is possible to use an XML document by itself as a database (in the strictest sense of the word) in a low-requirement environment it will certainly fail in a real world production environment (Bourret 2005). A native XML database system can be defined as a DBMS that represents data using the XML model. This, as with relational database systems, does not mean that the physical data storage must be XML.

Native XML databases can be classified as one of the following (Bourret 2005):

Text-based native XML databases. XML is stored as text, be it in a file, in a field in a relational database, or in any other text format. The XML document is indexed for ease of access to any point in the document. This is a far more efficient method than restoring separately stored XML fragments in a classic relational database or in some model-based XMLDBMS. Thus, text-based XML databases bear a striking resemblance to hierarchical databases.

Model-based native XML databases. An internal object model is created from the

XML document, and this model is stored in the database. The method of storage is not of importance. Model-based systems are usually built on other database models.

There is a number of working examples of XML databases, a large portion of which are developed by the open source community. While there is potential for this technology to find a market, it is not suitable for achieving the goal of this work.

2.4 A Fitting Approach?

All of the models mentioned above cover a broad spectrum of functionality. Support for the time dimension is also needed, and most of the models discussed can theoretically provide this support. Some of the points of argument, such as the need for a complex type system, can be easily solved within a Haskell environment. Other described features, though highly relevant in generalised DBMS, add levels of complexity not needed for this database management system.

Due to the specifics and restrictions of the problem at hand, a specialised new system must be developed, resembling in functionality some of the approaches described above. The following sections will discuss the addition of the temporal aspect to database models and give an overview of functional programming. Finally, the new system will be proposed.

3 Persistence and Temporal Data

This section will briefly explain the concept of data persistence, temporal databases, the concepts behind data actuality and various approaches to solve these problems. Furthermore data structures for storing temporal data will be discussed.

3.1 Persistence

C.J. Date describes persistence in databases as follows (Date 2004):

> It is customary to refer to the data in a database as "persistent" (though it might not actually persist for very long!). ... once it has been accepted by the DBMS for entry into the database in the first place, it can subsequently be removed from the database only by some explicit request to the DBMS, not as a mere side effect ...

Thus, any data stored in a persistent database is stored until it is removed by the DBMS. No additional explicit command is necessary to store the data, it is available for the duration of the DBMS' existence, though this does not necessarily mean only for or beyond the runtime of a program using the database.

3.2 Persistent Data Structures

In contrast to ephemeral data structures, where every change results in the destruction of the old structure in favour of the new changed structure, persistent data structures allow access to different versions (Driscoll, Sarnak, Sleator & Tarjan 1989). Further subcategorising, one speaks of partially persistent data structures if all versions can be accessed but modification is only possible on the current structure, and fully persistent data structures if all manifestations can both be accessed and modified.

Common to both definitions is the requirement of retaining any and all previous versions of the data structure within itself.

The method of data storage proposed in this paper (see section 5.3) fits this requirement, and is therefore at least a partially persistent data structure. Additionally manipulation is also possible after accessing a previous version of the database, however this does not affect the current (newest) version. By loading a previous version in the proposed system, all newer modifications are discarded and the previous version becomes, for the purpose of this instance, the current (newest) version. This data structure can then be modified, without affecting the previously discarded modifications.

3.3 Time

Temporal database systems include special support for the time dimension (Date, Darwen & Lorentzos 2003). They must provide special facilities for storing and querying historical and future data. Almost all conventional Database Management Systems are not temporal in this sense. However, for a number of reasons, from cheap disk storage to incorporation of temporal features into widespread standards, this is likely to change soon.

There are a number of possible ways to incorporate the time dimension into a database, as well as a number of different classes of resulting database systems, according to the type of time implemented in these systems. The most basic distinction is to be made between **transaction time**, **valid time** and **user defined time** (Snodgrass & Ahn 1985). Transaction time denotes the time at which data was stored in the database (Snodgrass & Ahn 1986). Every database transaction is marked at the time of its occurrence. It is thus possible to retrieve the state of the entire database at any point in (real) time. Valid time applies not to the transaction of data, but to data itself. A historical state is stored for each relation, representing the time at which the relationship was valid in the modeled world (Snodgrass & Ahn 1986). Finally, the term user-defined time is used for additional time-specific information not covered by transaction time or valid time. This information is application-specific (Snodgrass & Ahn 1985) and thus not interpreted by the database

management system. Support for user-defined time can be provided by simply adding an internal representation of time (Snodgrass & Ahn 1985).

3.4 Classification

According to the level of support of the time dimension as discussed earlier, database systems can be classified as follows (Snodgrass & Ahn 1985):

Snapshot databases, or static databases, represent a model at its current state. No time-specific information is stored, it is therefore impossible to directly access the time dimension. If the application requires this, time-specific support must be separately implemented (Snodgrass & Ahn 1986).

Rollback databases include the concept of transaction time, and adds a third dimension to the stored relations. It is possible to derive a snapshot database by selecting a specific historical state and retrieving the according data. This process is called rollback (Snodgrass & Ahn 1986).

Historical databases do not support transaction time, but valid time. Contrary to rollback databases, it is impossible to view the database as it was in the past (Snodgrass & Ahn 1986), as changes are not recorded. However, historical information is stored for each relation and each relation can assume different states over time. Historical databases closely model reality in this respect (Snodgrass & Ahn 1986).

Temporal databases denote a combination of both rollback and historical paradigms. Both valid time and transaction time are supported, adding a fourth dimension and full support for all temporal information (Snodgrass & Ahn 1986), as well as an increasing level of complexity.

3.4.1 Data structures

This section will describe some data structures capable of storing and organising time-specific data. Special attention is given to binary search trees as the method of choice

for this work.

Relational databases. The problem can be, in its simplest form, easily solved in a relational environment by simply adding **timestamp attributes** (*from date* and *to date* or *intervals* to represent valid time, or transaction timestamps to represent transaction time) to the data model (Date 2004). After applying the appropriate constraints, special query facilities can retrieve the relevant data.

Binary search trees. One of the most self-evident methods of temporalising data. A binary search tree, built with (*timestamp, value*) tuples for each piece of information in the database can represent both valid and transaction time. Access to data is hindered by another dimension. For each data retrieval operation the tree has to be searched, and for each update operation the tree has to be modified.

Red-Black trees. Due to performance issues with simple binary search trees, a more efficient approach is needed. Approximately balancing the tree can greatly improve both search and manipulation performance on ordered data. Red-black trees are among the most popular balanced binary search trees and work very well on ordered data (Okasaki 2008), which clearly applies here. Red-Black trees are used to represent data in the system proposed in this work and will be described in more detail in the following sections.

4 Functional Programming and Haskell

This section will give a short introduction into the world of functional programming with emphasis on the Haskell programming language and closely related topics and theories, and will elaborate on currently used database technologies available for Haskell.

4.1 History

The starting point for the evolution of functional programming was Alonzo Church's development of a method for describing arbitrary functions, the Lambda Calculus. Any computable function can be expressed and evaluated using the λ calculus. It is computationally universal, and thus equivalent to Turing machines. (Rojas 1998).

Generally regarded as the first functional programming language, John McCarthy developed LISP (List Processor) in 1958. Though having some influences from the λ-calculus, variable assignments where still at the core of the language (Hutton 2007).

ISWIM, the first pure functional programming language, was developed by Peter Landin in 1966.

Various programming languages are built on the principles of the λ calculus, from ML to Lisp, Erlang, and especially important for this work, Haskell.

4.2 Definition

Functional programming can be viewed as a style of programming in which the basic method of computation is the application of functions to arguments (Hutton 2007). Functional programming reflects a mathematical way of thinking, rather than reflecting the underlying machine (Goldberg 1994).

The key to functional programming's power is greatly improved modularisation as compared to common structured programming languages. Modularity leads to successful

programming (Hughes 1984), and thus functional programs have a great advantage over their structured counterparts.

4.3 Functional Programming from an Imperative Programmer's Point of View

Before delving deeper into the mathematical background of functional programming and the programming language this work is concerned with, this section will try to make the concept of functional programs more accessible to the average developer used to imperative languages. Over the past years, the author himself was, with a few exceptions, almost exclusively engaged in systems developed in widespread imperative programming languages. Since almost everyone concerned with functional programming is coming from that school of thought, it seems important to accentuate the transition involved in developing a functional program.

The most basic distinction might be that while an imperative program describes *how* something is computed, a functional program merely describes *what* is to be computed. This obviously leads to a higher level of abstraction.

Imperative programs rely heavily on order of execution. Modules are executed in a strict, predefined order. Moreover, the execution of these modules can (and often will) have an influence on the execution of other pieces of code. Memory is manipulated, values are assigned, manipulated, and reassigned.

Functional programming works in a very different manner. Variables, once given a value, never change, since functional programs, contrary to imperative programs, contain no assignment statements at all. In fact, functional programs contain no side-effects at all. By calling a function nothing else but that functions result is computed, with no influence on the rest of the program whatsoever (Hughes 1984).

Thus the order of execution becomes irrelevant, and programs become mathematically more traceable (Hughes 1984).

The crucial difference in thought is, as mentioned earlier, not trying to write a program that comes to a solution in a certain way, but trying to define the solution itself in a declarative manner.

4.4 Haskell

In the late 1980 dozens of lazy functional languages were being worked on. In 1990 a committee, formed to design a common language, published the Haskell 1.0 specifications, naming the language after Haskell Curry, an influential logician (O'Sullivan, Stewart & Goerzen 2008).

This section will give an overview of Haskell's functions and facilities and is not meant to cover the language as a whole. Basics as well as concepts which seem the most important to the author, considering the task at hand, are briefly described, while other seemingly important features might be omitted.

4.5 Database access in Haskell

This section will describe and evaluate existing approaches to connect Haskell to common databases.

4.5.1 HaskellDB

Originally proposed by Daan Leijen and Erik Meijer (Leijen & Meijer 1999), HaskellDB is a library for expressing queries and operations on relational databases in a type safe and declarative way (Bringert & Höckersten 2004). It is a domain-specific library for programming against relational data (Meijer 2007). The original HaskellDB concept was developed to work with several requirements, creating a number of version conflicts. It was redesigned and improved in 2004 (Bringert & Höckersten 2004) to make it a more practical and usable database library.

Though a number of SQL-based database interfaces to Haskell exists (see the following sections for a short overview), the method put forward with HaskellDB has several advantages (Bringert & Höckersten 2004):

- Syntactically or semantically invalid queries are detected before execution.
- No string manipulation avoids security issues such as SQL injections.
- No language disparity. The programmer does not have to work in two separate languages.

Even though the goal of this work is not to create a database interface but a native solution, being the most widespread solution to database interfacing with Haskell, it is worth to take a closer look at HaskellDB's features and functionality.

Bringert and HÃ¶ckersten define the advantages of HaskellDB as follows (Bringert & Höckersten 2004):

Query correctness. All database operations are checked by the Haskell compiler at compile time rather than by the database system at runtime. SQL injections are prevented by automatic quoting of supplied constants.

Ease of programming. Query errors are caught at compile time rather than at runtime, greatly improving the speed of the program/debug cycle. Knowledge of SQL is not required.

Expressive power. Abstraction of common patterns can be achieved by utilising all of Haskell's features since queries are written in Haskell, not in SQL.

Platform independence. By changing the connection function, HaskellDB can be used in any environment.

HaskellDB is an interface between Haskell and the relational model. The interesting aspect is that, from the programmer's point of view, it can be handled solely in Haskell. No knowledge of the actual DBMS behind HaskellDB or the used query language (SQL)

is needed. It is therefore a suitable solution to the problem, though probably far to powerful for the task at hand.

5 A Temporal Relation Database in Haskell

This section discusses the theoretical basics behind the proposed database system, the data model, modes of storage and data representations and any design decisions. A more detailed insight into the specific implementation and usage will be given in section 6.

5.1 Binary Relations

Initial requirements proposed a system managing binary relations, as the system is meant to be applied in an environment where all the relevant data can be expressed by binary relations. A relation is, non-mathematically, defined as a list of interrelated values. Binary relations form the special case of always interrelating two values, as compared to the more general definition of n-ary relations. Restricting the number of relvars in a relation simplifies operations on the relation, but also limits its functionality.

A Composition creates a new binary relation by combining two previous binary relations over a common property. This property, however, is lost by the composition and not present in the newly created binary relation. Join operations known from relational database systems are a generalised form of composition applying to n-ary relations.

The proposed system supports n-ary relations. This includes the possibility of managing binary relations. The difference in operation can primarily be found by examining the difference between a binary composition and an n-ary join. The following tables define two example relations R and S:

First Name	**Last Name**
Haskell	Curry
Mary	Wheatley

Table 1: Relation R

First Name	**Middle Name**
Haskell	Brooks
Mary	Virginia

Table 2: Relation S

Forming a binary composition over the $FirstName$ value creates a new binary relation relating $MiddleName$ to $LastName$. The information contained in $FirstName$ is no longer present in the newly created relation.

Middle Name	Last Name
Brooks	Curry
Virginia	Wheatley

Table 3: $R \circ S$

A natural join over $FirstName$ would create the relation seen in the following table. This is no longer a binary relation.

First Name	Middle Name	Last Name
Haskell	Brooks	Curry
Mary	Virginia	Wheatley

Table 4: $R \bowtie S$

5.2 Data Model

This section will propose a data model for a native temporal relation database for Haskell. Data is to be stored in relations with support for Haskell's powerful type system. Relations must be defined as flexible as possible to meet any application-specific requirements. Since a temporal database, by definition, supports both transaction time and valid time, facilities for both must be available within the data model. The main memory database structure may differ from the physical structure and both will be discussed in the following sections in addition to the representation of basic database concepts.

5.2.1 Values

Values within relations are represented by the Haskell type *RDBo* (for Relation Database Object) and can contain any Haskell type implementing the necessary functionality. Thus, the database is able to store almost any data which can be defined as a Haskell type, as well as support mixed-type values within one relation. For physical representation and manipulation of values within the database management system's index structure some restrictions apply, and the stored Haskell types must support ordering and equality tests as well as implement *read* and *show*.

5.2.2 Relations

Relations hold a fixed number of values for each data set and are identified by an integer within the main memory database structure.

$$(Identifier, Index) \tag{1}$$

Within the relation index, value sets (rows) are identified by a textual representation of the row's content. Considering a relation for the storage of employee personal data, a relation R might contain indices for *first name* and *last name*. Internally, these indices are built within a tree structure, as will be discussed in detail in section 6.1.2. A main memory database consists of any number of relations.

Mathematically, a relation R on a set is defined as a collection of related tuples of that set. Thus, if the set $(x, y)\epsilon R$, x is *R-related* to y.

5.2.3 Temporalising the Database

Both transaction time and valid time must be existent in the database system. However, due to the difference between the physical and main memory database the two temporal aspects are stored in different manners.

Timespans. Time can be stored as the number of seconds since January 1, 1970 (commonly known as Unix time). This allows for storage within an Integer, as well as for simple manipulation and computation of time values. The database system handles timespans as tuples of two Unix time integers, the first denoting the begin of the timespan, while the second marks the end. Additionally, the system provides the special timespan *ALWAYS* as a shortcut to the user, which is still represented as a tuple of Unix timestamps within the physical and main memory databases.

Transaction time. Every data manipulation is timestamped, and thus any operation can be traced to the time it occurred. Transaction time is, in this sense, not user-specific

but automatic, and there is no need for user transparency or even representation in the main memory database. As will be discussed in section 5.3.1, the physical database consists of a list of transactions. Each of these is marked with a timestamp, and since no information can be removed from the physical database, it can be restored to its state at any point in time since its creation. Since a transaction happens at a specific point in time, a single Unix timestamp is used instead of a timespan.

Valid time. Any data set is labelled with a timespan marking its valid period. Query facilities allowing for definition and selection of time periods are provided to the user. The indexing subsystem allows for multiple entries identified by the same value but differing in valid time, representing user-specific data change over time.

5.2.4 Transactions

As the method of physical storage, any manipulation of the database is defined by a transaction. The system supports the following three major types of operations:

- Defining relations
- Inserting data sets
- Deleting data sets

With these three transaction types, the prototype supports almost all possible data manipulations.

5.3 Data Storage and Manipulation

This work proposes a twofold method of data storage. The system distinguishes between data stored on disk (the database file, in following called the *physical database*) and the data preprocessed for access and manipulation stored in main memory (the *main memory database*).

In order to represent any historical state of the database, every transaction with an effect on actual data (namely insertions and deletions) is stored within the physical database. The physical database therefore is represented as a file consisting of a list of transactions. Only changes to the database are stored, and the actual database can be built from this list of changes.

Once the physical database is loaded into memory it ceases to contain all historical data and furthermore only represent the current state of the database. Additionally the data is moved into more efficient data structures to allow for better access. After transferring the physical database to these data structures in main memory it is referred to as the main memory database.

Further transactions are also kept within the main memory. By storing the current main memory database to disk, this list of transactions appended to the initially loaded physical database depicts the new physical database.

Additional facilities are provided for direct manipulation of the physical database. Function from within this automated persistence module automatically apply changes to both manifestations of the database, and manual storage to disk is no longer necessary.

5.3.1 The Physical Database

Due to the restrictions of this work, enough memory for the representation of the entire database can be assumed to exist. Therefore, loading the entire database into memory at the beginning of the session and storing the entire database to disk at the end of the session is a sufficient implementation. The physical database is only accessed by direct request, and no database operation has a side effect manipulating the physical database unless that manipulation is specifically executed, or unless the manipulation is executed using the automated storage module.

The DBMS' historical aspect requires the system to be able to revert to any state it had at any point in time since it's existence. Since the entire database is loaded into memory, the main memory database can be rebuilt to any historic state during that time. By

managing the physical database as a list of data manipulating transactions, that is any transaction that has an effect on the state of the main memory database, and executing these transactions in order of occurrence, the main memory database can be built from the physical database.

Additionally, tagging each transaction with a Unix timestamp enables the rollback feature of the DBMS. During the process of rebuilding the main memory database transactions tagged with a timestamp after a specified time, newer transactions can be omitted from the rebuild process and the database will resemble its state at the specified time.

Thus, the physical database is stored in a file as a list of timestamped transactions.

5.3.2 The Main Memory Database

Contrary to the physical database, the main memory database must aim to provide efficient access to the stored data sets in order to enable data extraction and further computation. Additionally, to allow for the representation of valid time and the full functionality of a temporal database, each data set is tagged with a timespan denoting its validity.

To enable efficient access, indices are built for each relational variable within each relation. These indices are stored as red-black binary search trees, where each tree node represents one data set. Tree ordering is based primarily on each particular relational variable, the data set's corresponding valid time, and additionally on a database-internal unique identifier to allow for duplicate entries. Each relation contains a number of indices equal to its arity, one for each relational variable.

Accordingly, the main memory database consists of a list of relations, each of which consists of a list of index trees.

5.3.3 Database Aspects and Temporal Aspects

The separation of a physical and a main memory database also allows for a clean separation of the two time aspects.

Transaction time is only relevant when the database is to be restored to a previous state in time. Thus, by linking transaction time directly to the physical database, the full functionality of a rollback database can be achieved. A timestamp is stored with each transaction in the physical database. By transforming only the transactions prior to a certain time into the main memory database, a historical database state can be rebuilt.

Valid time is represented as an aspect of a data set and is interweaved into the main memory database index trees to allow for temporal queries and joins. The two different time aspects are therefore handled by different aspects of the database.

5.4 The Database Model

Combining both the physical and main memory database models, the DBMS manages the data contained within a database as a tuple of these two data structures. Operations are applied immediately to the main memory database and the triggering transaction is added to the physical database for later or immedeate storage to disk.

5.5 Data Structures

Apart from the basic data management facilities of Haskell, wherein value and relation data will be stored in tuples, lists, or lists of tuples, the internal representation of relation indices, and therefore of the entire database, will me managed with binary search trees.

5.5.1 Balanced Binary Search Trees

Within an index, relation data can be uniquely identified with the addition of an internal identification tag. Instead of simply storing a list of data sets, a much faster approach was chosen. The object and relation indices are represented as balanced binary search trees, limiting the operation time significantly.

A certain kind of balanced binary search tree will be used to implement most data structures for the proposed system: **red-black trees**. The tree's balance can be ensured by applying specific rules to the structure of the search tree. In addition to any application-specific data every node contains colouring information, marking the node either as red or as black. If the following rules are considered in every tree operation, it remains balanced.

- No red node has a red child
- Every path from the root to an empty node has the same number of black nodes

Thus, the longest possible path is no more than twice as long as the shortest possible path, and no individual operation takes more than $O(logn)$ time (Okasaki 2008). It therefore provides a significant increase in performance on ordered data as compared to regular binary search trees (Okasaki 2008).

Implementation of efficient binary search trees in functional programming languages is comparatively simple. Thus, the relation indices can be elegantly and effectively represented using red-black trees. Even considering the expected amount of data to be relatively small, a fast implementation of such a data structure still seems the right choice for the problem at hand.

5.6 Database Operations

Specified as Haskell types resembling common SQL statements, the following functions for data manipulation and retrieval are offered to the user:

Database creation. A blank database is returned, presented as a tuple of an empty list of relations and an empty list of transactions.

Relation definition. Adds a new relation with a specified relation identifier and a corresponding list of relational variable identifiers to the database. All further operations are based on relations, which are therefore the main target of computation within the database management system.

Insertion of data sets. Adds a data set consisting of a list of $(relvar, value)$ tuples to the specified relation. All index trees are updated accordingly.

Deletion of data sets. Creates a list of delete transactions for each data set meeting the provided condition. Each of these transactions is then applied to the main memory database and added to the physical database.

Updating data sets. No facilities for changing data sets directly are provided. SQL-like update operations can be achieved by sequential deletion of the deprecated data set and insertion of the new data set.

Data retrieval. Contrary to the operations mentioned above, retrieval operations have no effect on the database. Data sets from multiple relations are selected according to specified value conditions, join operations and temporal conditions. The resulting data sets are combined within a newly, non-persistent relation which is returned for further computation. The following section will describe the retrieval capabilities in detail.

5.7 Data Access and Relational Algebra

The database management system allows for several restrictions for the specification of data sets to be retrieved. This section describes these constraints in detail.

Relational variables. A list of relvar identifiers is provided by the user to specify the requested data. All information not stored within these variables is omitted in the result.

Relations. The list of relations which the retrieval constraints are applied to.

Value conditions. On each data set a specified function of a relvar and a comparison value is applied. If the result is positive the data set is included, otherwise it is omitted.

Temporal conditions. Similar to value conditions, a function of a valid time timestamp and a comparison timestamp is applied to each data set to determine its relevance to the query.

Temporal Joins. Two relations are reduced to one by applying to every dataset a specified function of two relvars, one from each relation. If the result is positive, the two datasets are joined over the two relvars into a new dataset within the result relation. This has to be done considering any temporal constraints, which might lead to a number of different result data sets for different periods of valid time caused by overlapping or distinct valid time of the source data sets.

5.8 Data Manipulation on Disk

The prototype described in the following sections seperates data manipulation and storage clearly. Even though the system distinguishes the physical database as the format of disk storage from the main memory database as the data structure for queries, any changes made to the database during runtime are lost if the physical database is not explicitly stored to a file. However, an additional persistence module is implemented which executes automatic updates of the physical database upon each manipulating transaction.

To achieve persistence in the common sense (as mentioned by C.J. Date, see section 3.1), the database management system must implicitly store all transactions in the database file. As of now the prototype supports a usage mode that does not achieve this as all transactions are processed in main memory, as well as a mode that automatically stores all changes to both database manifestations. Additionaly, functions are provided that allow for easy storage of the physical database. There are several advantages and disadvantages to this method, and this section will highlight them as well as provide reasons for the chosen model.

Direct data manipulation on disk introduces an additional layer of complexity. Data

has to be stored efficiently and in a structured manner on disk to avoid unnecessary performance losses during access and manipulation operations. Additional concerns such as concurrency and data integrity also apply. For non-large databases representation of the entire database in main memory can increase performance as compared to continuing disk operations. By excluding issues such as concurrency and adding an automatic update of the physical database upon each manipulation a solution was chosen that can achieve the requirement of automatic persistence.

Resource conservation while handling large databases is the main advantage of data manipulation on disk. Only the parts of the database currently needed have to reside in main memory, providing a scalable database system. Mapping the entire database in main memory restricts its extent to the available resources.

The prototype developed for this work manages the entire database both in main memory and on disk. This method was chosen to provide the necessary functionality while at the same time avoiding the distraction of the additional complexities introduced by direct manipulation on disk.

5.8.1 Future Enhancements

With additional indexing and structuring of the physical database, the system might be modified to load only the relations required for a specific operation into memory. Considering these factors, the prototype can be refined into a database management system manipulating databases directly on disk.

6 Implementation

This section will describe the process of implementing the database prototype in detail, the techniques used and highlight functionality and access in order to integrate the prototype into other Haskell programs.

6.1 Basic Types and Data Structures

The following sections will elaborate on the selected types and structures for the representation of data within the system.

6.1.1 Values

The value of a relational variable can be any type of the form

```
data RDBo =   RInt Integer
            | RFloat Float
            | RChar Char
            | RString String
            | RList [RDBo]
            | RNull
            | RForeign RDBo
           deriving (Show,Read,Ord,Eq)
```

Figure 1: Basic data types

By amending the type *RDBo* with additional arbitrary Haskell types, the database system is capable of storing any data implementing the required instances. The value *RNull* is reserved for requests specifically not returning any results for a given data set. However, due to the design of the main memory database and the omission of missing values in non-affected indices, the value is not used in the current system. *RForeign* is used to avoid conflicts during the combination of two relations and possible duplicate values.

6.1.2 Database Structures

The internal representation of any database consists of the tuple

$$(LogicalDB, PhysicalDB) \tag{2}$$

where $PhysicalDB$ consists of a list of timestamped transactions and $LogicalDB$ of a list of $(Identifier, Relation)$ tuples, and $LogicalDB$ denotes the main memory database. The relations are further subdivided into

$$Relation = [([String], IndexTree)] \tag{3}$$

which is a list of $(RelvarIdentifier, Index)$ tuples. All actual data sets within the main memory database are stored within these indices.

6.1.3 Index Trees

Index trees within relations are defined as trees of the type

```
type RDIndexTree = IVTree (RDBo, Timespan, Integer) [([String],RDBo)]
```

Figure 2: Index trees

which defines red-black id-value ($IVTree$) trees where each tree node is identified by the triple $(Indexrelvar, validtime, identifier)$ and holds a list of relvars not indexed in this tree. Since every tree node must be uniquely identified to guarantee correct operation, the additional internal identifier is used to avoid conflicts with duplicate data sets which might arise from temporal joins or other complex operations on the relation.

6.1.4 Database manipulation and retrieval

A database statement describing any of the three major operations is defined in pseudo-SQL as follows:

```
data RDStatement = CREATE_TABLE Integer [String]
  | INSERT Integer [([String],RDBo)] Timespan
  | DEL Integer [String] (RDBo, Timespan)
  deriving (Show,Read)
```

Figure 3: Statements

The $CREATE\_TABLE$ statement is used to define a new relation with the specified identifier and variable list, $INSERT$ adds a new data set with a specified valid time to a relation, and DEL removes a dataset from a relation.

For storage within the physical database a $(Integer, RDStatement)$ tuple is used, amending the statement with a timestamp representing transaction time.

In addition to these statements, the type *RDSelect* is used to define operations which have no direct influence on the database:

```
data RDSelect = SELECT [[String]] [Integer] [Condition] [Join] [
    TempCondition]
  | DELETE Integer Condition
```

Figure 4: Database operations

$DELETE$ statements are not executed immediately, but rather subdivided into DEL statements which can be applied to both the main memory and physical database.

The more complex $SELECT$ statement stores a number of arguments required for the retrieval of data sets. Its functionality will be described in detail in section 6.2.4.

6.1.5 Conditions and Joins

The system manages three types of conditions, each applying to a different aspect of a relation and the data sets within.

Value conditions. Being of the type $([String], (RDBo- > RDBo- > Bool), RDBo)$, value conditions apply a specified function to a relvar and a static value. If the function returns $True$, the condition is met and the operation is allowed to continue. For example, the condition

$$([\text{"}LastName\text{"}], (==), (RString\text{"}Curry\text{"})) \tag{4}$$

would apply to all data sets where the relvar $LastName$ is equal to "$Curry$".

Temporal Conditions of the type $(String, (Integer- > Timespan- > Bool), Integer)$ apply to the valid time of each data set within a relation. A relvar identifier is provided to select the index tree upon which the condition will be applied. A function of a timespan and a timestamp on the corresponding data set is provided to specify the temporal condition. The temporal condition

$$(\text{"}Student\text{"}, before, 638928000) \tag{5}$$

would apply to all *Student* data sets with a valid time before April 1990.

Joins are used to combine two relations according to a specified condition. A join defined as $((Integer, String), (RDBo- > RDBo- > Bool), (Integer, String))$ will combine the two relations specified by their according identifier on each data set where the specified relvars meet the condition. For example, the join condition

$$((0, \text{"}LastName\text{"}), (==), (1, \text{"}MaidenName\text{"})) \tag{6}$$

would create a new relation combining all data sets where $LastName$ and $MaidenName$ are equal. By allowing arbitrary join conditions, the system is able to express a variety of different join operations.

6.2 Database Access

The system provides various functions for database input/output as well as for applying the statements mentioned above to a database. This section will describe these functions in detail.

6.2.1 Database I/O

In addition to the basic function *createDatabase*, which returns an empty tuple, the system provides a function for storing the database to disk and two functions for restoring the main memory database from the physical database.

Unlike the functions *storeToDisk* and $loadFromDisk$, which read or write the entire physical database to disk, the function $loadFromDiskHist$ also takes a timestamp as a parameter to allow for the implementation of rollback database features. The list of statements composing the physical database are only read and executed up to the point in time given by the timestamp.

6.2.2 Basic Statements, delete and select

In order to apply the available statements to the database, three functions are provided by the system.

query applies the statements described in section 6.1.4 to a database. Since the DEL statement is usually generated and not expressed by the user, *query* will be mainly used for $CREATE\_TABLE$ and $INSERT$ statements.

delete is a helper function for $DELETE$ statements, executing the deletion condition

and separating the original statement into a list of *DEL* transactions capable of being handled by *query*.

select applies a *SELECT* statement to a database. However, contrary to *query* and *delete* this function does not return the changed database structure, since it has no manipulating effect on the database whatsoever. Instead, the resulting relation is returned for further computation or data extraction.

6.2.3 Persistence module

In addition to the functions mentioned above, the persisence module provides the following functions:

- *persistentQuery*
- *persistentDelete*

These functions have the same purpose as their common counterparts, with the difference of automatic storage to disk. For this purpose the structure of a *RDQuery* is modified to a *RDPersistentQuery*:

```
type RDPersistentQuery = (RDatabase, String, RDStatement)
```

Figure 5: RDPersistentQuery type

The *String* denotes the filename of the physical database to which the transactions are to be stored.

6.2.4 Data Retrieval with SELECT

Revisiting section 6.1.4, the *SELECT* statement is of the form

containing five parameter lists for the *select* function. The first list defines the relvars to be included in the result relation, the second list contains the source relations for the

```
SELECT [[String]] [Integer] [Condition] [Join] [TempCondition]
```

Figure 6: SELECT type

select statement. The final three parameters describe the value and temporal conditions to be applied, as well as any join operations on the source relations.

6.2.5 Data Extraction and Output

The system provides basic functionality for database and relation output on screen as can be seen in the *Print* module.

6.3 Time and Temporal Joins

Using Haskell's *System.Time* module the database system is able to create current Unix timestamps on the executing machine. These timestamps, stored either as single integers or as tuples of the *Timespan* format form the basis for all temporal calculations within the system. Timespans also provide the shortcut $ALWAYS$, which is internally stored as a timespan from the minimum to the maximum 32-bit integer. This can be easily adapted to avoid future time overflow problems.

6.3.1 Time-specific Functionality

Apart from the retrieval of the current Unix timestamp and basic time types, the system also provides functionality for more complex timespan arithmetic. The following comparison functions are implemented to assist in specifying temporal conditions. These are always functions of an integer and a timespan to a boolean value.

- **contains** returns *True* if the provided timestamp is within the boundary of the timespan.
- **before** computes if a timestamp denotes a point in time before the start of the

timespan.

- **after** computes if a timestamp denotes a point in time after the end point of the timespan.
- **from** and **to** are used to describe a second timespan, and return $True$ if the two provided timespans overlap.

6.3.2 Temporal Joins

The additional dimension added to relations by valid time is also relevant to join arithmetic. Considering the following example relations

ID	Last Name
1	Curry
2	Wheatley

Table 5: Relation A

ID	First Name	Middle Name
1	Haskell	Brooks
2	Mary	Virginia

Table 6: Relation B

a natural join over the corresponding identifiers would yield the following relation:

ID	First Name	Middle Name	Last Name
1	Haskell	Brooks	Curry
2	Mary	Virginia	Wheatley

Table 7: $A \bowtie_{ID_A=ID_B} B$

By adding overlapping valid time to the two original relations the resulting joined relation becomes more complex. Considering the following nonsensical example for the demonstration of temporal joins:

ID	Last Name	Valid
1	Curry	1900-1982

Table 8: Relation A_1

ID	First Name	Middle Name	Valid
1	Haskell	Brooks	1910-1990

Table 9: Relation B_1

Joining these relations considering valid time will not result in one data set in the joined relation, but three.

ID	First Name	Middle Name	Last Name	Valid
1	Haskell	Brooks	Curry	1910-1982
1	-	-	Curry	1900-1910
1	Haskell	Brooks	-	1982-1990

Table 10: $A_1 \bowtie_{ID_{A_1}=ID_{B_1}} B_1$

This does not only allow for temporally restricted validity of data sets, but for different combinations of data for different time periods. Considering the following example source relations and the corresponding joined relation, the application of temporal joins becomes evident.

ID	Last Name	Valid
2	Wheatley	1903-1928
2	Curry	1928-

Table 11: Relation A_2

ID	First Name	Middle Name	Valid
2	Mary	Virginia	1903-

Table 12: Relation B_2

ID	First Name	Middle Name	Last Name	Valid
2	Mary	Virginia	Wheatley	1903-1928
2	Mary	Virginia	Curry	1928-

Table 13: $A_2 \bowtie_{ID_{A_2}=ID_{B_2}} B_2$

By adding a temporal condition to the query, the database system would provide different results for different periods of valid time.

6.4 Temporal Aspects

As described in section 5.3.3, support transaction time and valid time are implemented separately within the physical and main memory database manifestations. This section will describe in detail the methods of implementation.

6.4.1 Transaction Time

As mentioned earlier, the physical database consists of a list of timestamped statements, each of which manipulating the database.

```
[(1253960971,CREATE_TABLE 0 ["ID","First␣Name","Middle␣Name"]),
(1257071212,INSERT 0 [(["ID"],RInt 0),(["First␣Name"],RString "Haskell"),(["
    "Middle␣Name"],RString "Brooks")] (VALID (-2147483648,2147483647))) ]
```

Figure 7: Physical DB excerpt

Figure 7 shows a small excerpt from the physical database, containing two statements. Each of these statements contains a timestamp denoting the point in time at which the statement was processed by the system. The first transaction creates a new relation identified by 0 on September 26th, 2009, while the second transaction adds a data set to that relation on November 1st, 2009. Since all modifications of the database are expressed by appending additional transaction to the physical database transaction list, the list is automatically ordered by transaction time. A previous state can be computed by sequentially executing all transactions up to a specified timestamp.

6.4.2 Valid Time

Valid time is represented both within the physical database and the main memory database, but functionality utilising valid time applies only to the main memory database. Considering the second transaction in figure 7, we see that the newly created data set contains a valid timespan. That timespan defines the valid time of the data set, in this example starting from December 13th, 1901 up to January 19th, 2038. Each data set contains such a timespan, expressing valid time within the entire database.

The index tree structures that make up the main memory database contain this valid time timespan at each node (each node representing a data set). These trees are secondarily indexed by valid time (and primarily index by the corresponding relvar). Data sets that contain different manifestations over time are therefore ordered together within the search tree. For a detailed description of the implementation of the main memory database see

section 6.7.2.

6.5 Database Manipulation

As mentioned above, the system allows for three kinds of data manipulation within a database and provides a function for initialising a blank database.

6.5.1 Database Creation

Calling the function

```
let db <- createDatabase
```

Figure 8: Database creation

stores an empty $(LogicalDB, PhysicalDB)$ tuple within *db*. All further operations will be computed on this database object.

6.5.2 Relation Definition

The statement in figure 9 adds a new relation with the identifier 0 and three specified relational variables to the database *db* and replaces the old database object.

```
db <- query (db, (CREATE_TABLE 0 ["ID","First Name","Middle Name"]))
```

Figure 9: Relation definition

Further on accessed by its identifier 0, this relation can now be used for data insertion and extraction.

6.5.3 Data Insertion

Data Insertion is achieved by executing an $INSERT$ statement. The following example would add two data sets consisting of one integer and two strings to the corresponding relvars within relation 0. No specific valid time is given for this data set, instead $ALWAYS$ is used to define unrestricted validity.

```
db <- query (db, (INSERT 0 [(["ID"], RInt 1),(["First Name" ], RString "
    Haskell"),(["Middle Name"] , RString "Brooks")] ALWAYS ))
db <- query (db, (INSERT 0 [(["ID"], RInt 2),(["First Name" ], RString "
    Mary"),(["Middle Name"] , RString "Virginia")] ALWAYS ))
```

Figure 10: Relation definition

Examining the examples above, the relation 0 would hold the following data:

ID	**First Name**	**Middle Name**	**Valid**
RInt 1	RString "Haskell"	RString "Brooks"	ALWAYS
RInt 2	RString "Mary"	RString "Virginia"	ALWAYS

Table 14: Example Relation 0

6.5.4 Deletion

Due to its conditional nature, the $DELETE$ statement is stored differently within the physical database than it is available to the user. The $DELETE$ statement

```
db <- delete db (DELETE 0 (["First Name"],(like),(RString "kell")))
```

Figure 11: Conditional Deletion

contains a condition requiring all data sets where $FirstName$ contains the string "$kell$" to be deleted from relation 0. Since this might affect more than one data set, the statement must first be translated into a list of DEL operations which can be applied to the

main memory database and added to the physical database. Thus, the following operation is created by applying the original $DELETE$ statement to the database:

```
DEL 0 ["First Name"] (RString "Haskell", ALWAYS)
```

Figure 12: Deletion

After executing this operation, the example relation will hold the following data:

ID	**First Name**	**Middle Name**	**Valid**
RInt 2	RString "Mary"	RString "Virgina"	ALWAYS

Table 15: Example Relation 0 after deletion

6.6 Data Retrieval

Revisiting the $SELECT$ type described in section 6.2.4, this section will describe the functionality of data retrieval in detail. The following $SELECT$ statements incorporate most of the functionality available for data retrieval:

```
let result1 = select db (SELECT [["ID"], ["Last Name"]] [0] [(["Last Name"
    ],(==),(RString "Curry"))] [] [])

let result2 = select db (SELECT [["ID","ID"], ["First Name"], ["Last Name"
    ]] [0,1] [] [(0,"ID"),(==),(1,"ID"))] [("ID",from,1940)])
```

Figure 13: Data Retrieval

The first statement selects a data set from a single relation, restricted by a value condition ruling out all data sets where $LastName$ does not equal "$Curry$". The second statement forces a temporal join on two relations over the relvar ID and adds a temporal restriction ruling out all data sets not valid after 1940.

Considering the example relations A_2 and B_2 as described in tables 7 and 8, the two results computed by the specified $SELECT$ statements would appear as follows:

ID	Last Name	Valid
RInt 2	RString "Curry"	1928-

Table 16: result1

ID,ID	First Name	Last Name	Valid
RInt 2	RString "Mary"	RString "Curry"	1928-

Table 17: result2

The first statement simply returns all data sets that meet the specified requirements. Since there is only one data set with the last name valued "Curry", only one row is returned. The join in the second statement produces a temporary relation containing two result rows with different values and different valid time. However, after applying the temporal condition, only one of these data sets remains.

6.7 Data representation

This section will describe in detail the methods of representation for both the physical and the main memory aspect of the database.

6.7.1 The physical database

The physical database is defined as seen in figure 14.

```
type PhysicalDB = [RDTransaction]
type RDTransaction = (Integer, RDStatement)

data RDStatement = CREATE_TABLE Integer [String]
  | INSERT Integer [([String],RDBo)] Timespan
  | DEL Integer [String] (RDBo, Timespan)
  deriving (Show,Read)
```

Figure 14: Physical database

A transaction is therefore defined as a data manipulating statement (namely relation definition, insertion and deletion) amended with a timestamp. The physical database is

a list of such timestamped statements. The database is manipulated by adding additional transactions to this list.

6.7.2 The main memory database

The main memory database is defined as:

```
type LogicalDB = [(Integer, RDRelation)]
type RDRelation = [RDIndex]
type RDIndex = ([String], RDIndexTree)
type RDIndexTree = IVTree (RDBo, Timespan, Integer) RDRelationType
type RDRelationType = [([String],RDBo)]
```

Figure 15: Main memory database

In order to explain the tree structures behind the main memory database properly each of these definitions will be discussed.

RDIndexTree describes the type of nodes within a red black id-value tree that are to be used to represent the data. The identifier consists of a triple of the form (*RDBo*, *Timespan*, *Integer*). Each node is therefore identified by the indexed column object (*RDBo*), its corresponding valid time (the *Timespan*) as well as an internal identifier used to distinguish the data set for join operations. The value of each tree node consists of a **RDRelationType**, which is a list of tuples containing value objects and their corresponding column identifiers. For each column within a relation such an index tree is built and stored within the **RDIndex** type in a tuple with its corresponding column identifier. The sum of all index trees within a relation defines the relation itself, and a **RDRelation** is therefore a list of *RDIndex* types. The main memory database type (**LogicalDB**) amends these relation index trees with a relation identifier and stores them as a list of tuples.

Thus there is an index for each relvar of each relation. The correct indices are chosen for data retrieval and join operations to allow for efficient access.

```
type RDatabase = (LogicalDB, PhysicalDB)
```

Figure 16: Combining main memory and physical DB

6.7.3 Conversion

While active, both the main memory and the physical database are managed by the database system. Whenever a data manipulating transaction occurs, the transaction list within the physical database is amended and the index trees of the main memory database are updated accordingly. It is not necessary to recreate the physical database from main memory as it is also stored there. However, after execution of the system only the physical database remains, if stored on disk. The main memory database can then be restored from the database file by applying the list of transactions to an initially empty main memory database. By executing each transaction stored in the physical database the index trees are rebuilt to their former state.

7 Conclusion

This section will reflect on the work done on the proposed system and the prototype. The functionality of the prototype will be evaluated against the goals set at the beginning of the work, difficulties during research and development will be highlighted, and possibilities for improvement and further development will be discussed. Finally, the author will give a critical evaluation of the work as a whole from his point of view.

7.1 Achieved Goals

This work's goal was not to provide an exhaustive description of all involved technologies, but to give an overview of the basics behind database design, temporal data, and functional programming. The literature review presented in sections 2 through 4 not only provided that overview, but presented an extensive list of possible solutions to the problem at hand. Merits and demerits where discussed in order to establish the basis for an informed decision concerning the selection of methods for the proposal of the system and the implementation of the prototype.

Section 5 describes the proposed system in detail without concerning itself with specific issues of implementation. A system was proposed that meets all the requirements within the scope of this work.

Both the prototype itself and its detailed description in section 6 form the main part of this thesis. Specific methods of implementation as well as usage within other systems are discussed in detail.

7.2 Difficulties

Apart from the author's habitual difficulties with non-imperative programming languages discussed in section 4.3 some difficulties arose during the creation of this work and especially during the development of the prototype.

Initial requirements stated the need for a binary relation database system, however, during the development of the system it became apparent that such a system would not be powerful enough to handle the complex join operations. The system had to be redesigned to support n-ary relations in order to allow for these computations. Its proposed field of use is still concerned with binary relations, however, reducing the result relations to such data structures can be achieved by the applications utilising the prototype.

The system utilises a minimal number of Haskell libraries without, to the best knowledge of the author, implementing features that are already available. This leads to a large number of required basic functionality, for which optimal implementation is not guaranteed.

Finally, since the prototype constitutes the main part of this work, choosing the correct amount of literature to be documented and the detail thereof proved difficult to the author.

7.3 Ties to Literature

The structure of this work is clearly separated between the literature review in sections 2 through 4 and the author's proposal and documentation in sections 5 and 6. The literature review lays the groundwork for any further design decisions and ties this thesis strongly to previous works in the field of database management systems. The later sections contain few references to literature since, to the best knowledge of the author, few of the implemented features can be tied to a specific previous work.

The developed prototype is no reimplementation of any previous work, and any decision basis and terminology has been tied to its original source within the literature review.

7.4 Further Development

Even though the thesis covers all of the requirements, there are many starting points for further development. Optimisation of the prototype is an issue, especially concerning

performance-related optimisation. One example would be the optimisation of the tree-deletion algorithm, which is not optimal.

Additionally, interfaces for ease of access to the functionality might be added to allow a broader field of possible users to apply this database management system to Haskell software.

7.5 Evaluation

Expected results of this work, apart from the main parts concerning system definition and prototype implementation, included a detailed description of the problem at hand and an overview of possible solutions present in current literature. Both sections 2 and 4 are concerned with possible solutions to achieving the task of creating a database management system. Due to the specific requirements, mainly the implementation in a functional programming language and the inclusion of temporal aspects, no approaches found in the reviewed literature could be applied directly to the problem at hand. The specific design choices and corresponding reasoning as well as a detailed analysis of the problem and its possible solution can be found in the later sections describing the system solution and implementation.

Two design choices specifically vary as compared to the initially proposed design, and are worth a more detailed look.

Binary relations. As described in detail in section 5.1, binary relations suffice for representation of the target data sets. Binary relations and the corresponding simple compositions of these relations were replaced by a system managing more powerful yet far more complex n-ary relations and corresponding complex joins. This was done primarily to allow for more complete join operations, especially temporal joins.

Data manipulation on disk. Though not mentioned explicitly in the initial proposal, the capability of handling large databases was discussed. The developed prototype has a different data structure for storage on disk (the physical database) than for data access and manipulation (the main memory manifestation of the database).

Two operation modes are available in the implemented system, one managing the database only in memory. The *persistent* operation mode is based on the same operations, but stores changes to the database not only in memory but also automatically within a specified database file on disk. Any state of the database, current or historical, can be extracted from this file.

7.6 Personal Evaluation

During the first months of working on this thesis it became apparent that it would be far more complex that initially assumed. Due to my lack of experience with both Haskell and database theory and the resulting misunderstandings of requirements, the entire scope of this work only became clear after some time. However, concerning myself with these topics and gaining the required knowledge to start design and implementation, I was able to create a work that meets or exceeds the initial requirements, includes multiple features, and was able to rouse further interest in the concerned topics.

7.7 Acknowledgements

First, I would like to thank my parents, without whom I would have never been able to achieve my academic goals. I would also like to thank Dr. A. Frank for introducing me to the subject and for his supervision of this work, and Andreas Bolka, Viktor Pavlu and Christian Seidl, without whom I would not have attempted a masters degree. Special thanks go to Nora Rosmann for her continuous support during the writing of this work.

References

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. & Zdonik, S. (1989), 'The object-oriented database system manifesto'.

Bourret, R. (2005), 'Xml and databases', *http://www.rpbourret.com/xml/XMLAndDatabases.htm* .

Bringert, B. & Höckersten, A. (2004), 'Student paper: Haskelldb improved'.

Codd, E. F. (1970), 'A relational model of data for large shared data banks', *Commun. ACM* **13**(6), 377–387.

Codd, E. F. (1980), Data models in database management, *in* 'Proceedings of the 1980 workshop on Data abstraction, databases and conceptual modeling', ACM, New York, NY, USA, pp. 112–114.

Date, C. (2004), *An Introduction to Database Systems*, 8th edn, Addison Wesley.

Date, C., Darwen, H. & Lorentzos, N. A. (2003), *Temporal Data and the Relational Model*, Morgan Kaufmann.

Driscoll, J. R., Sarnak, N., Sleator, D. D. & Tarjan, R. E. (1989), 'Making data structures persistent', *Journal of Computer and System Sciences* **38**.

Evans, D. (2008), *Computational Thinking*, `http://www.cs.virginia.edu/~evans/ctbook/`.

Gallaire, H., Minker, J. & Nicolas, J.-M. (1984), 'Logic and databases: A deductive approach', *Computing Surveys* **16**(2).

Goldberg, B. (1994), 'Functional programming languages'.

Hellerstein, J. M., Stonebraker, M. & Hamilton, J. (2007), 'Architecture of a database system'.

Hudak, P., Hughes, J., Jones, S. P. & Wadler, P. (2007), 'A history of haskell: Being lazy with class'.

Hughes, J. (1984), 'Why functional programming matters'.

Hutton, G. (2007), *Programming in Haskell*, Cambridge University Press.

Jensen, C., Clifford, J., Dyreson, C., Gadia, S., i, F., Jajodia, S., Kline, N., Montanari, A., Nonen, D., Peressi, E., Pernici, B., Roddick, J., Sarda, N., lal L, Scalas, M., Segev, A., Snodgrass, R., Soo, M., Tansel, A. & Tiberio, P. (1993*a*), Addendum to "proposed temporal database concepts - may 1993".

Jensen, C., Clifford, J., Dyreson, C., Gadia, S., i, F., Jajodia, S., Kline, N., Montanari, A., Nonen, D., Peressi, E., Pernici, B., Roddick, J., Sarda, N., lal L, Scalas, M., Segev, A., Snodgrass, R., Soo, M., Tansel, A. & Tiberio, P. (1993*b*), Proposed temporal database concepts - may 1993.

Jones, S. P. (2003), *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press.

Jones, S. P. (2009), 'Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell'.

Jung, A. (2004), 'A short introduction to the lambda calculus'.

Klinger, S. (2005), 'The haskell programmer's gudie to the io monad'.

Leijen, D. & Meijer, E. (1999), Domain specific embedded compilers, *in* 'Proceedings of DSL'99: The 2nd Conference on Domain-Specific Languages'.

Meijer, E. (2007), Confessions of a used programming language salesman, *in* 'OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications', ACM, New York, NY, USA, pp. 677–694.

Okasaki, C. (2008), *Purely Functional Data Structures*, Cambridge University Press.

O'Sullivan, B., Stewart, D. & Goerzen, J. (2008), *Real World Haskell*, O'Reilly Media, Inc.

Ozsoyoglu, G. & Snodgrass, R. T. (1995), 'Temporal and real-time databases: A survey', *IEEE Transactions on Knowledge and Data Engineering* **7**(4), 513–532.

Reiter, R. (1984), 'Towards a logical reconstruction of relational database theory'.

Rojas, R. (1998), 'A tutorial introduction to the lambda calculus'.

Snodgrass, R. (1990), 'Temporal databases status and research directions', *SIGMOD Rec.* **19**(4), 83–89.

Snodgrass, R. T. & Ahn, I. (1985), A taxonomy of time in databases, *in* S. B. Navathe, ed., 'Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985', ACM Press, pp. 236–246.

Snodgrass, R. T. & Ahn, I. (1986), 'Temporal databases', *IEEE Computer 19(9)* pp. 35–42.

Stonebraker, M., Rowe, L. A., Lindsay, B., Gray, J., Carey, M., Brodie, M., Bernstein, P. & Beech, D. (1990), 'Third-generation database system manifesto'.

Sun (2008), 'The java tutorials', `http://java.sun.com/docs/books/tutorial/java/concepts/object.html`.

List of Figures

List of Tables

A Haskell Source

A.1 BasicTypes

```
module TempDB.BasicTypes where

{-

basic data types

prototype developed as a part of
"A Native Temporal Relation Database for Haskell"

Lukas Maczejka, 2009-2010

-}



-- values, can be any type meeting the requirements
data RDBo =    RInt Integer
  | RFloat Float
  | RChar Char
  | RString String
  | RList [RDBo]
  | RNull
  | RForeign RDBo
  deriving (Show,Read,Ord,Eq)
```

A.2 Data

```
module TempDB.Data where

{-

types

prototype developed as a part of
"A Native Temporal Relation Database for Haskell"

Lukas Maczejka, 2009-2010


-}




import TempDB.Tree
import TempDB.BasicTypes
import TempDB.Time




-- query engine

-- select (column names) (from relations) (where) (joins) (temporal
    conditions)
data RDSelect = SELECT [[String]] [Integer] [Condition] [Join] [
    TempCondition]
  | DELETE Integer Condition

-- column name, funcion, value
type Condition = ([String],(RDBo -> RDBo -> Bool),RDBo)

-- column name, funcion, column name
type Join = ((Integer,String),(RDBo -> RDBo -> Bool),(Integer,String))

-- function, value
type TempCondition = (String,(Integer -> Timespan -> Bool),Integer)


-- database manipulation
```

```
data RDStatement = CREATE_TABLE Integer [String]                    --
    relation id
                 | INSERT Integer [([String],RDBo)] Timespan        --
                     relation id, object1, object 2, valid time
                 | DEL Integer [String] (RDBo, Timespan)            --
                     relation id, col name, object id to delete
                 deriving (Show,Read)

-- a query is a statement on a database
type RDQuery = (RDatabase, RDStatement)

-- a query that stores any changes immedeately to the db file
type RDPersistentQuery = (RDatabase, String, RDStatement)

-- transactions are timestamped statements (no db info), transaction time
    level
type RDTransaction = (Integer, RDStatement)



--  database

-- database is logical db and physical db
type RDatabase = (LogicalDB,PhysicalDB)

-- relation types
type RDRelationType = [([String],RDBo)]

-- index tree
type RDIndexTree = IVTree (RDBo, Timespan, Integer) RDRelationType

-- index
type RDIndex = ([String],RDIndexTree)

-- relations are represented by a list of index trees
type RDRelation = [RDIndex]

-- logical db is list of relations with relation ids
type LogicalDB = [(Integer,RDRelation)]

-- database is stored as a list of transactions
type PhysicalDB = [RDTransaction]
```

A.3 Indices

```
module TempDB.Indices where


{-

common index manipulation (relations, transactions)

prototype developed as a part of
"A Native Temporal Relation Database for Haskell"

Lukas Maczejka, 2009-2010

-}


import TempDB.Data
import TempDB.BasicTypes
import TempDB.Time
import TempDB.Tree



-- insert value/timespan into all indices
insertIndices :: [RDIndex] -> [([String],RDBo)] -> Timespan -> [RDIndex]
insertIndices [] _ _ = []
insertIndices (h:t) obs ts = [insertIndex h obs obs ts] ++ insertIndices t
    obs ts


-- insert value/timespan into index
insertIndex :: RDIndex -> [([String],RDBo)] -> [([String],RDBo)] ->
    Timespan -> RDIndex
insertIndex idx [] _ _ = idx
insertIndex idx@(iname,tree) ((oname,o):t) oos ts | oname == iname = (iname
    ,ivtInsert tree (o,ts,(ivtNextId tree)) (excludeOs oos oname) )
  | otherwise = insertIndex idx t oos ts

-- get object list without specified id
excludeOs :: [([String],RDBo)] -> [String] -> [([String],RDBo)]
excludeOs [] _ = []
excludeOs ((nm,o):t) ex | nm == ex  = excludeOs t ex
  | otherwise = [(nm,o)] ++ excludeOs t ex
```

A.4 IO

```
module TempDB.IO where

{-

disk i/o

prototype developed as a part of
"A Native Temporal Relation Database for Haskell"

Lukas Maczejka, 2009-2010

-}

import TempDB.Data
import TempDB.Transactions
import System.IO (openFile, hClose, hPrint, hGetContents, IOMode(ReadMode,
    WriteMode)) -- file operations
import Control.Parallel.Strategies (rnf) -- required for lazy hGetContents


-- file output

storeToFile :: RDatabase -> String -> IO ()
storeToFile (_,pdb) filename = do
  fh <- openFile filename WriteMode
  hPrint fh pdb
  hClose fh


-- file input

loadFromFile :: String -> IO RDatabase
loadFromFile filename = do
  fh <- openFile filename ReadMode
  cont <- hGetContents fh
  rnf cont `seq` hClose fh
  let pdb = (read cont)::PhysicalDB
  ldb <- executeTransactions pdb ([],[])
  return (ldb,pdb)


-- load rollback
```

```
loadFromFileHist :: String -> Integer -> IO RDatabase
loadFromFileHist filename rollback = do
  fh <- openFile filename ReadMode
  cont <- hGetContents fh
  rnf cont `seq` hClose fh
  let pdb = truncHist ((read cont)::PhysicalDB) rollback
  ldb <- executeTransactions pdb ([],[])
  return (ldb,pdb)


truncHist :: PhysicalDB -> Integer -> PhysicalDB
truncHist [] _ = []
truncHist ((ts,state):t) rollback | ts < rollback = [(ts,state)] ++
    truncHist t rollback
                                  | otherwise = truncHist t rollback

-- execute transaction list

executeTransactions :: PhysicalDB -> RDatabase -> IO LogicalDB
executeTransactions [] (ldb,_) = return ldb
executeTransactions ((ts,statement):tl) db = do
  db <- (query (db, statement))
  executeTransactions tl db
```

A.5 Print

```
module TempDB.Print where

{-

screen output, printing, debug

prototype developed as a part of
"A Native Temporal Relation Database for Haskell"

Lukas Maczejka, 2009-2010


-}


import TempDB.Data
import TempDB.BasicTypes
import TempDB.Tree
import TempDB.Relations
import TempDB.Time



{-

-}


-- print list of relations
printRelations [] = putStr ""
printRelations ((i,h):t) = do
  putStr (show i)
  putStr "\n"
  printRelation h
  putStr "\n"
  printRelations t


-- print single relation
printRelation [] = do
  putStr "";

printRelation r@(rel:t) = do
```

```
  printRelHeader r
  let (nm,idx) = rel
  putStr ("index for " ++ (show nm) ++ "\n\n")
  printRows idx (getRelColTitles idx [])
  putStr "\n\n"
  printRelation t


-- print relation header
printRelHeader rel = do
  putStr "\n"
  printRelCols (getRelCols (head rel))
  putStr "VALID"
  putStr "\n\n"

-- print relation columns (header ids)
printRelCols :: [[String]] -> IO ()
printRelCols [] = do
  putStr ""
printRelCols (h:t) = do
  printRelCol h
  putStr ","
  printRelCols t

-- print relation column id
printRelCol :: [String] -> IO ()
printRelCol l = putStr (show (concat l))



-- print relation values (rows)
printRows :: RDIndexTree -> [[String]] -> IO ()
printRows IVEmpty _ = putStr ""
printRows (IVNode _ l (i,ts,iid) cont r) lst = do
  printO i
  printOrderedElements cont lst
  putStr "\t"
  printTimespan ts
  putStr "\t("
  putStr (show iid)
  putStr ")\n"
  printRows l lst
  printRows r lst
```

```
-- print values in correct order
printOrderedElements :: [([String],RDBo)] -> [[String]] -> IO ()
printOrderedElements cont [] = putStr ""
printOrderedElements cont (h:t) = do
  printElement h cont
  printOrderedElements cont t

-- print single value
printElement :: [String] -> [([String],RDBo)] -> IO ()
printElement l [] = putStr ",-"
printElement l ((ht,el):t) | l == ht = do
                             putStr ", "
                             printO el
                           | otherwise = printElement l t

-- print timespan
printTimespan :: Timespan -> IO ()
printTimespan ALWAYS = putStr "ALWAYS"
printTimespan (VALID (f,t)) = putStr ((show f) ++ " to " ++ (show t))



-- print RDBo type
printO :: RDBo -> IO ()
printO (RInt i) = putStr (show i)
printO (RFloat f) = putStr (show f)
printO (RChar c) = putStr ("'" ++ [c] ++ "'")
printO (RString s) = putStr s
printO (RList (h:t)) = do
  printO h
  putStr ","
  printO (RList t)
printO (RList []) = putStr ""
printO _ = putStr "?"  -- unknown



-- print db to screen
printDB :: RDatabase -> IO ()
printDB (ldb,pdb) = do
                    printList ldb
                    printList pdb
```

```
printList :: (Show a) => [a] -> IO ()
printList (h:t) = do
                  print h
                  printList t
printList [] = putStrLn ""
```

A.6 Relations

```
module TempDB.Relations where

{-

relations

prototype developed as a part of
"A Native Temporal Relation Database for Haskell"

Lukas Maczejka, 2009-2010

-}

import TempDB.Data
import TempDB.BasicTypes
import TempDB.Tree
import TempDB.Tools
import TempDB.Time
import TempDB.Indices
import Data.List (isInfixOf) -- string comparions condition: 'like'




-- get column names of index
getRelCols :: RDIndex -> [[String]]
getRelCols (s,idx) = xAdd (getRelColTitles idx []) [s]

-- build list of column names for index tree
getRelColTitles :: RDIndexTree -> [[String]] -> [[String]]
getRelColTitles IVEmpty lst = lst
getRelColTitles (IVNode _ l (i,_,_) cont r) lst =  getRelColTitles r (
    getRelColTitles l (xAdd lst (relIndexColFlatten cont)))

-- build list of ids from tree node content
relIndexColFlatten :: [([String],RDBo)] -> [[String]]
relIndexColFlatten [] = []
relIndexColFlatten ((s,_):t) = [s] ++ relIndexColFlatten t


-- combine multiple relations to one
flattenRels :: [([Integer],RDRelation)] -> RDRelation
flattenRels [] = [([""],IVEmpty)]
```

```
flattenRels rel@((i,h):t) | (length rel) == 1 = h
  | (length rel) == 2 = fCombine h (flattenRelC (t !! 0))
  | otherwise         = fCombine h (flattenRels t)

-- combine first index of two relations, representative since all indices
    contain all data
fCombine :: RDRelation -> RDRelation -> RDRelation
fCombine rel1 rel2 = [fComb (rel1 !! 1) (rel2 !! 1)]

-- combine two indices
fComb :: RDIndex -> RDIndex -> RDIndex
fComb (s1,i1) (s2,i2) = (s2, idxCombine i1 i2 s1)

-- combine index trees, add values of second index tree as RForeign to
    avoid conflicts
idxCombine ::  RDIndexTree -> RDIndexTree -> [String] -> RDIndexTree
idxCombine IVEmpty tree tix = tree
idxCombine (IVNode _ l (h,ts,iid) t r) tree tix = ivtInsert (idxCombine l (
    idxCombine r tree tix) tix) (RForeign h,ts,iid) (t ++ [(tix,h)])

-- remove identifier from relation
flattenRelC :: ([Integer],RDRelation) -> RDRelation
flattenRelC (_,r) = r



-- find index

getIndexTree :: RDRelation -> [String] -> IVTree (RDBo,Timespan,Integer)
    RDRelationType
getIndexTree ((iid,idx):t) [""] = idx
getIndexTree [] _ = IVEmpty
getIndexTree ((iid,idx):t) sid | sid == iid = idx
                               | otherwise  = getIndexTree t sid


getIndexTree3 :: RDRelation -> String -> IVTree (RDBo,Timespan,Integer)
    RDRelationType
getIndexTree3 ((iid,idx):t) "" = idx
getIndexTree3 [] _ = IVEmpty
getIndexTree3 ((iid,idx):t) sid | sid `elem` iid = idx
                               | otherwise  = getIndexTree3 t sid

-- remove index
```

```
cutIndexTree :: RDRelation -> [String] -> RDRelation
cutIndexTree [] _ = []
cutIndexTree (i@(iid,idx):t) sid | sid == iid = cutIndexTree t sid
                                 | otherwise  = [i] ++ cutIndexTree t sid

-- find index with err
getIndexTree2 :: RDRelation -> [String] -> Maybe (IVTree (RDBo, Timespan,
    Integer) RDRelationType)
getIndexTree2 [] _ = Nothing
getIndexTree2 ((iid,idx):t) sid | sid == iid = Just idx
                                | otherwise  = getIndexTree2 t sid


-- get relation by id
getRelation :: RDatabase -> Integer -> RDRelation
getRelation (ldb,_) x = getRel ldb x

getRel :: LogicalDB -> Integer -> RDRelation
getRel [] _ = []
getRel ((i,rel):t) x | i == x = rel
  | otherwise = getRel t x

getRelLst :: [([Integer],RDRelation)] -> Integer -> RDRelation
getRelLst [] _ = []
getRelLst ((i,rel):t) x | x `elem` i = rel
  | otherwise = getRelLst t x


-- relation to list
relToList :: RDRelation -> [(Timespan,[RDBo])]
relToList ((_,idx):_) = relIndexToList idx

relIndexToList :: IVTree (RDBo, Timespan, Integer) RDRelationType -> [(
    Timespan,[RDBo])]
relIndexToList IVEmpty = []
relIndexToList (IVNode _ l (o1,t,_) o2 r) = [(t, [o1] ++
    relIndexContentFlatten o2)] ++ relIndexToList l ++ relIndexToList r

relIndexContentFlatten :: [([String],RDBo)] -> [RDBo]
relIndexContentFlatten [] = []
relIndexContentFlatten ((_,o):t) = [o] ++ relIndexContentFlatten t
```

```
-- additional comparison functions/conditions
like :: RDBo -> RDBo -> Bool
like (RString x) (RString y) = isInfixOf y x
like x y = x == y



-- apply condidion
applyCondition :: RDRelation -> ([String],(RDBo -> RDBo -> Bool),RDBo) ->
    RDRelation
applyCondition rel (sidx,func,comp) | (getIndexTree rel sidx) == IVEmpty =
    rel
  | otherwise = indexGuard (joinBuildIndex [(sidx, (applyConditionToIndex (
      getIndexTree rel sidx) (func,comp) IVEmpty) )] [])


-- make sure an index is returned, even if empty
indexGuard :: RDRelation -> RDRelation
indexGuard [] = [([""],IVEmpty)]
indexGuard r = r


-- build new index tree, only with values meeting the condition
applyConditionToIndex :: RDIndexTree -> ((RDBo -> RDBo -> Bool),RDBo) ->
    RDIndexTree ->  RDIndexTree
applyConditionToIndex IVEmpty  _ ctree = ctree
-- optimisations
--applyConditionToIndex tree f@((==),comp) ctree = applyConditionToIndexOp
    tree f ctree
--applyConditionToIndex tree f@((>=),comp) ctree = applyConditionToIndexOp
    tree f ctree
--applyConditionToIndex tree f@((<=),comp) ctree = applyConditionToIndexOp
    tree f ctree
--applyConditionToIndex tree f@((>),comp) ctree = applyConditionToIndexOp
    tree f ctree
--applyConditionToIndex tree f@((<),comp) ctree = applyConditionToIndexOp
    tree f ctree
-- general conditions
```

```
applyConditionToIndex (IVNode _ l idx@(i,ts,_) cont r) (func,comp) ctree |
    func i comp = ivtInsert (applyConditionToIndex l (func,comp) (
    applyConditionToIndex r (func,comp) ctree)) idx cont
  | otherwise = applyConditionToIndex l (func,comp) (applyConditionToIndex
      r (func,comp) ctree)



--optimised tree search
applyConditionToIndexOp :: RDIndexTree -> ((RDBo -> RDBo -> Bool),RDBo) ->
     RDIndexTree -> RDIndexTree
applyConditionToIndexOp (IVNode _ l idx@(i,ts,_) cont r) (func,comp) ctree
    | func i comp = ivtInsert (applyConditionToIndexOp l (func,comp) (
    applyConditionToIndexOp r (func,comp) ctree)) idx cont --all subtrees,
    due to multiple timestamped ids
  | i < comp    = applyConditionToIndexOp l (func,comp) ctree -- only left
      subtree
  | i > comp    = applyConditionToIndexOp r (func,comp) ctree -- only right
       subtree



-- temporal conditions
applyTCondition :: RDRelation -> (String,(Integer -> Timespan -> Bool),
    Integer) -> RDRelation
applyTCondition rel (sidx,func,comp) | (getIndexTree3 rel sidx) == IVEmpty
    = rel
  | otherwise = indexGuard (joinBuildIndex [([sidx], (
      applyTConditionToIndex (getIndexTree3 rel sidx) (func,comp) IVEmpty) )
      ] [])


applyTConditionToIndex :: RDIndexTree -> ((Integer -> Timespan -> Bool),
    Integer) -> RDIndexTree -> RDIndexTree
applyTConditionToIndex IVEmpty _ ctree = ctree
applyTConditionToIndex (IVNode _ l idx@(i,ts,_) cont r) (func,comp) ctree |
     func comp ts = ivtInsert (applyTConditionToIndex l (func,comp) (
    applyTConditionToIndex r (func,comp) ctree)) idx cont
  | otherwise = applyTConditionToIndex l (func,comp) (
      applyTConditionToIndex r (func,comp) ctree)
```

```
-- helper functions to represent temporal conditions

-- point contained in timespan
contains :: Integer -> Timespan -> Bool
contains _ ALWAYS = True
contains i (VALID (f,t)) = f <= i && i <= t

-- point after timespan
after :: Integer -> Timespan -> Bool
after _ ALWAYS = False
after i (VALID (f,t)) = i > t

-- point before timespan
before :: Integer -> Timespan -> Bool
before _ ALWAYS = False
before i (VALID (f,t)) = i < f

-- from/to conditions: all timespans overlapping the from/to points
from :: Integer -> Timespan -> Bool
from _ ALWAYS = False
from i (VALID (f,t)) = f >= i || t >= i

to :: Integer -> Timespan -> Bool
to _ ALWAYS = False
to i (VALID (f,t)) = f <= i || t <= i



-- temporal conditional join, with respect to timespanned duplicate keys (
    valid time)
tjoin :: RDRelation -> RDRelation -> (String, String) -> (RDBo -> RDBo ->
    Bool) -> RDRelation
tjoin r1 r2 (id1,id2) cond = indexGuard ( joinBuildIndex [([id1,id2], (
    tjoinIndex (getIndexTree r1 [id1]) (getIndexTree r2 [id2]) IVEmpty cond)
    )] [])


tjoinIndex :: RDIndexTree -> RDIndexTree -> RDIndexTree -> (RDBo -> RDBo ->
     Bool) -> RDIndexTree
tjoinIndex (IVNode _ l (i,ts,_) cont r) other jtree cond =  tjoinCombInsert
     i (tjoinGetComb (i,ts) other cont [] cond) (tjoinIndex l other (
    tjoinIndex r other jtree cond) cond)
tjoinIndex IVEmpty _ jtree cond = jtree
```

```
tjoinGetComb :: (RDBo, Timespan) -> RDIndexTree -> [([String],RDBo)] -> [(
    Timespan,[([String],RDBo)])] -> (RDBo -> RDBo -> Bool) -> [(Timespan,[([
    String],RDBo)])]    -- [([String]),RDBo)
tjoinGetComb (_,ts) IVEmpty mcont clist cond = clist
tjoinGetComb (i,ts) (IVNode _ l (oi,ots,_) cont r) mcont clist cond | cond
    oi i = tjoinGetComb (i,ts) r mcont (tjoinGetComb (i,ts) l mcont (tsocadd
     (tsoCombine (ts,mcont) (ots,cont)) clist) cond) cond
  | otherwise = tjoinGetComb (i,ts) r mcont (tjoinGetComb (i,ts) l mcont
      clist cond) cond

tjoinCombInsert :: RDBo -> [(Timespan,[([String],RDBo)])] -> RDIndexTree ->
     RDIndexTree
tjoinCombInsert _ [] jtree = jtree
tjoinCombInsert i ((ts,cont):t) jtree = tjoinCombInsert i t (ivtInsert
    jtree (i,ts,(ivtNextId jtree)) cont)

joinBuildIndex :: RDRelation -> RDRelation -> RDRelation
joinBuildIndex ((nm, (IVNode _ l (nmo,ts,_) lst r)):_) rel = joinBuildIndex
     [(nm,l)] (joinBuildIndex [(nm,r)] (joinBuildIndex2 rel ([(nm,nmo)] ++
    lst) ([(nm,nmo)] ++ lst) ts))
joinBuildIndex (i@(_,IVEmpty):_) rel = rel


joinBuildIndex2 :: RDRelation -> [([String],RDBo)] -> [([String],RDBo)] ->
    Timespan -> RDRelation
joinBuildIndex2 rel [] oos _ = rel
joinBuildIndex2 rel ((i,o):t) oos ts = case getIndexTree2 rel i of
  Just idx -> joinBuildIndex2 ((cutIndexTree rel i) ++ [(insertIndex (i,idx
      ) oos oos ts)]) t oos ts
  Nothing -> joinBuildIndex2 (rel ++ [(insertIndex (i,IVEmpty) oos oos ts)
      ]) t oos ts
```

A.7 Time

```
module TempDB.Time where

{-

time related functions

prototype developed as a part of
"A Native Temporal Relation Database for Haskell"

Lukas Maczejka, 2009-2010

-}

import TempDB.BasicTypes
import System.Time (getClockTime, ClockTime(TOD)) -- unix timestamp
    creation


-- timespan, valid time level
data Timespan = VALID (Integer, Integer)  -- valid from/until timestamp
              | ALWAYS
              deriving (Show,Read,Ord,Eq)


-- get current timestamp
timestamp :: IO Integer
timestamp = getClockTime >>= (\(TOD sec _) -> return sec)




-- timespans

inTimespan :: Integer -> Timespan -> Bool
inTimespan _ ALWAYS = True
inTimespan i (VALID (f,t)) = (i >= f) && (i <= t)



-- timespans with object lists
```

```
tsoAdd :: [([String],RDBo)] -> [([String],RDBo)] -> [([String],RDBo)]
tsoAdd l [] = l
tsoAdd l (h:t) | h `elem` l = tsoAdd l t
               | otherwise = [h] ++ tsoAdd l t


-- do two timespans overlap? if not, they are seperates
tsoSeperate :: (Timespan,[([String],RDBo)]) -> (Timespan,[([String],RDBo)])
    -> Bool
tsoSeperate ((VALID (f1,t1)),_) ((VALID (f2,t2)),_) | f1 > t2  || f2 > t1 =
    True
                                                    | otherwise = False
tsoeperate _ _ = False  -- ALWAYS always overlaps


tsoOverlap :: (Timespan,[([String],RDBo)]) -> (Timespan,[([String],RDBo)])
   -> Bool
tsoOverlap ts1 ts2 = not (tsoSeperate ts1 ts2)



-- build timespan/object list from two timespan/object lists with respect
    to overlapping validity
tsoCombine :: (Timespan,[([String],RDBo)]) -> (Timespan,[([String],RDBo)])
   -> [(Timespan,[([String],RDBo)])]
tsoCombine tso1@(ts1@(VALID (f1,t1)),o1) tso2@(ts2@(VALID (f2,t2)),o2) |
    tsoSeperate tso1 tso2 = [tso1,tso2]
  | ts1 == ts2 = [(ts1,tsoAdd o1 o2)]
  | f1 < f2 && t2 > t1   = [ ((VALID (f1,f2-1)),o1), ((VALID (f2,t1)),
      tsoAdd o1 o2), ((VALID (t1+1,t2)), o2)]
  | f1 < f2 && t2 < t1   = [ ((VALID (f1,f2-1)),o1), ((VALID (f2,t2)),
      tsoAdd o1 o2), ((VALID (t2+1,t1)), o1)]
  | f1 < f2 && t2 == t1  = [ ((VALID (f1,f2-1)),o1), ((VALID (f2,t1)),
      tsoAdd o1 o2)]
  | f1 > f2 && t2 < t1   = [ ((VALID (f2,f1-1)),o2), ((VALID (f1,t2)),
      tsoAdd o1 o2), ((VALID (t2+1,t1)), o1)]
  | f1 > f2 && t2 > t1   = [ ((VALID (f2,f1-1)),o2), ((VALID (f1,t1)),
      tsoAdd o1 o2), ((VALID (t1+1,t2)), o2)]
  | f1 > f2 && t2 == t1  = [ ((VALID (f2,f1-1)),o2), ((VALID (f1,t2)),
      tsoAdd o1 o2)]
  | f1 == f2 && t1 < t2  = [ ((VALID (f1,t1)),tsoAdd o1 o2), ((VALID (t1+1,
      t2)), o2)]
  | f1 == f2 && t1 > t2  = [ ((VALID (f1,t2)),tsoAdd o1 o2), ((VALID (t2+1,
      t1)), o1)]
```

```
  | otherwise = []

-- any overlap over list?
tsoOverlapsList :: (Timespan,[([String],RDBo)]) -> [(Timespan,[([String],
    RDBo)])] -> Bool
tsoOverlapsList ts [] = False
tsoOverlapsList ts (h:t) = tsoOverlap ts h || tsoOverlapsList ts t

tsoOverlapList :: (Timespan,[([String],RDBo)]) -> [(Timespan,[([String],
    RDBo)])] -> [(Timespan,[([String],RDBo)])]
tsoOverlapList ts [] = []
tsoOverlapList ts (h:t) | tsoOverlap ts h = [h] ++ tsoOverlapList ts t
  | otherwise = tsoOverlapList ts t


tsoSeperateList :: (Timespan,[([String],RDBo)]) -> [(Timespan,[([String],
    RDBo)])] -> [(Timespan,[([String],RDBo)])]
tsoSeperateList ts [] = []
tsoSeperateList ts (h:t) | tsoSeperate ts h = [h] ++ tsoSeperateList ts t
  | otherwise = tsoSeperateList ts t


tsoCom :: [(Timespan,[([String],RDBo)])] -> [(Timespan,[([String],RDBo)])]
tsoCom t = tsoc t []

tsoc :: [(Timespan,[([String],RDBo)])] -> [(Timespan,[([String],RDBo)])] ->
     [(Timespan,[([String],RDBo)])]
tsoc [] l = l
tsoc (h:t) l = tsoc t (tsocc h t l)

tsocc :: (Timespan,[([String],RDBo)]) -> [(Timespan,[([String],RDBo)])] ->
    [(Timespan,[([String],RDBo)])] -> [(Timespan,[([String],RDBo)])]
tsocc ts [] l = l
tsocc ts (h:t) l = tsocc ts t (tsocadd (tsoCombine ts h) l)

tsocadd :: [(Timespan,[([String],RDBo)])] -> [(Timespan,[([String],RDBo)])]
     -> [(Timespan,[([String],RDBo)])]
tsocadd [] l = l
tsocadd (h:t) l | tsoTSElement h l = tsocadd t (tsoaddTSElement h l)
  | tsoOverlapsList h l = tsocadd t (tsoSeperateList h l ++ tsoCom ([h] ++
      tsoOverlapList h l))
  | otherwise = [h] ++ tsocadd t l
```

```
tsoTSElement :: (Timespan,[([String],RDBo)]) -> [(Timespan,[([String],RDBo)
    ])] -> Bool
tsoTSElement ts [] = False
tsoTSElement a@(ts,_) ((h,_):t) | ts == h = True
  | otherwise = tsoTSElement a t

tsoaddTSElement :: (Timespan,[([String],RDBo)]) -> [(Timespan,[([String],
    RDBo)])] -> [(Timespan,[([String],RDBo)])]
tsoaddTSElement ts [] = []
tsoaddTSElement a@(ts,o1) (b@(h,o2):t) | ts == h = [(ts,tsoAdd o1 o2)] ++
    tsoaddTSElement a t
  | otherwise = [b] ++ tsoaddTSElement a t
```

A.8 Tools

```
module TempDB.Tools where

{-

misc. tools

prototype developed as a part of
"A Native Temporal Relation Database for Haskell"

Lukas Maczejka, 2009-2010

-}




-- exclusive add
-- concat list elements of list 2 to list 1 only if not already part of
    list 1
xAdd :: Eq a => [a] -> [a] -> [a]
xAdd l [] = l
xAdd l (h:t) | h `elem` l = xAdd l t
  | otherwise = [h] ++ xAdd l t



-- maximum signed 32bit integer
maxInt :: Integer
maxInt = 2147483647


-- minimum signed 32bit integer
minInt :: Integer
minInt = -2147483648
```

A.9 Transactions

```
module TempDB.Transactions where


{-

db transactions

prototype developed as a part of
"A Native Temporal Relation Database for Haskell"

Lukas Maczejka, 2009-2010

-}


import TempDB.Data
import TempDB.BasicTypes
import TempDB.Tools
import TempDB.Time
import TempDB.Tree
import TempDB.Relations
import TempDB.Indices


-- apply query to database
-- convert to transaction, call respective functions
--
-- create table
-- insert
-- delete

query :: RDQuery -> IO RDatabase
query ((ldb,pdb), s@(CREATE_TABLE i nms)) = do
  nt <- toTransaction s
  return (ldb ++ [createTable i nms], pdb ++[nt])

query (db, (INSERT i obs ALWAYS)) = query (db, INSERT i obs (VALID (minInt,
    maxInt)))
query ((ldb,pdb), s@(INSERT i obs tm@(VALID (from,to)))) = do
  nt <- toTransaction s
  return (insert ldb i obs tm, pdb ++[nt])

query (db, del@(DEL _ _ _)) = applyDelete db [del]
```

```
-- add timestamp to statement
toTransaction ::  RDStatement -> IO RDTransaction
toTransaction s = timestamp >>= \ts -> return (ts, s)



-- create new empty database
createDatabase :: IO RDatabase
createDatabase = return ([],[])



-- create new relation
createTable :: Integer -> [String] -> (Integer,RDRelation)
createTable i rows = (i,createIndices rows)

-- build empty indices with column names
createIndices :: [String] -> [RDIndex]
createIndices [] = []
createIndices (h:t) = [([h],IVEmpty)] ++ createIndices t




-- insert into logical db
insert :: LogicalDB -> Integer -> [([String],RDBo)] -> Timespan ->
    LogicalDB
insert [] _ _ _ = []
insert (h:t) x obs tm = [insertRelation h x obs tm] ++ insert t x obs tm

-- insert into relation
insertRelation :: (Integer,RDRelation) -> Integer -> [([String],RDBo)] ->
    Timespan -> (Integer,RDRelation)
insertRelation (n,rel) x obs ts | n == x    = (n, (insertIndices rel obs ts
    ))
  | otherwise = (n,rel)
```

```
-- delete statement (RDSelect because there is no DELETE in the physical DB
    - it is resolved into a list of DEL statements)
delete :: RDatabase -> RDSelect -> IO RDatabase
delete db (DELETE i cond@(sidx,_,_)) = (applyDelete db (deleteCond i (
   getIndexTree (getRelation db i) sidx) cond))
delete db _ = return db

-- create transaction from DEL, apply to database
applyDelete :: RDatabase -> [RDStatement] -> IO RDatabase
applyDelete db [] = return db
applyDelete (ldb,pdb) (h:t) = do
  nt <- toTransaction h
  applyDelete (deleteFromLdb ldb h, pdb ++[nt]) t

-- delete from logical db
deleteFromLdb :: LogicalDB -> RDStatement -> LogicalDB
deleteFromLdb ldb (DEL i sidx ob) = [(i, indexGuard (joinBuildIndex [(sidx,
   deleteFromLdb2 (getIndexTree (getRel ldb i) sidx) ob IVEmpty)] [])),] ++
   excludeIdx ldb i
deleteFromLdb ldb _ = ldb

-- get relation list without specified relation id
excludeIdx :: LogicalDB -> Integer -> LogicalDB
excludeIdx [] i = []
excludeIdx (h@(ii,_):t) i | ii == i = excludeIdx t i
                          | otherwise = excludeIdx t i ++ [h]

-- delete from index tree
deleteFromLdb2 :: RDIndexTree -> (RDBo,Timespan) -> RDIndexTree ->
   RDIndexTree
deleteFromLdb2 IVEmpty _ dtree = dtree
deleteFromLdb2 (IVNode _ l ob@(oo,ot,oi) cont r) o dtree | (oo,ot) == o =
   deleteFromLdb2 l o (deleteFromLdb2 r o dtree)
  | otherwise = ivtInsert (deleteFromLdb2 l o (deleteFromLdb2 r o dtree))
      ob cont
```

```
-- resolve delete condition, create DEL statements
deleteCond :: Integer -> RDIndexTree -> Condition -> [RDStatement]
deleteCond _ IVEmpty _ = []
deleteCond i (IVNode _ l (oi,ots,iid) cont r) cond@(sidx,func,comp) | func
    oi comp = [(DEL i sidx (oi,ots))] ++ deleteCond i l cond ++ deleteCond i
     r cond
  | otherwise = deleteCond i l cond ++ deleteCond i r cond




-- build new relation meeting select conditions
select :: RDatabase -> RDSelect -> RDRelation
select db (SELECT cols rels cond join tcond) = selectCols (flattenRels (
    selectTCond (selectJn (selectCond db rels cond) join) tcond)) cols

-- remove all but specified columns
selectCols :: RDRelation -> [[String]] -> RDRelation
selectCols rel [] = rel
selectCols rel lst = selectStripContent (selectGetCols rel lst) lst

-- remove column from relation
selectStripContent :: RDRelation -> [[String]] -> RDRelation
selectStripContent [] _ = []
selectStripContent ((s,i):t) lst = [(s,selectStripIndex i lst)] ++
    selectStripContent t lst

-- remove column from index tree
selectStripIndex :: RDIndexTree -> [[String]] -> RDIndexTree
selectStripIndex IVEmpty _ = IVEmpty
selectStripIndex (IVNode c l ob cont r) lst = (IVNode c (selectStripIndex l
     lst) ob (selectStripIndexContent cont lst) (selectStripIndex r lst))

-- remove column from value list
selectStripIndexContent :: [([String],RDBo)] -> [[String]] -> [([String],
    RDBo)]
selectStripIndexContent [] _ = []
selectStripIndexContent (h@(s,_):t) lst | s `elem` lst = [h] ++
    selectStripIndexContent t lst
  | otherwise = selectStripIndexContent t lst

-- remove index trees of not selected columns
```

```
selectGetCols :: RDRelation -> [[String]] -> RDRelation
selectGetCols [] _ = []
selectGetCols (rel@(s,_):t) lst | s `elem` lst = [rel] ++ selectGetCols t
    lst
  | otherwise = selectGetCols t lst




-- apply joins from select condition
selectJn :: [([Integer],RDRelation)] -> [Join] -> [([Integer],RDRelation)]
selectJn rels [] = rels
selectJn rels (h:t) = selectJn (selectJnApply h rels) t

-- apply single join to relation list
selectJnApply :: Join -> [([Integer],RDRelation)] -> [([Integer],RDRelation
    )]
selectJnApply _ [] = []
selectJnApply jn@( (id1,_), _, (id2,_) ) rels = selectJnStripJrel jn rels
   ++ [(selectJnApplyRel jn (getRelLst rels id1) (getRelLst rels id2))]

-- apply single temporal join to relation (see tempdb.relations)
selectJnApplyRel :: Join -> RDRelation -> RDRelation -> ([Integer],
    RDRelation)
selectJnApplyRel ( (id1,cols1), func, (id2,cols2) ) r1 r2 = ([id1] ++ [id2
    ], tjoin r1 r2 (cols1,cols2) func)


-- remove relations not needed for this join (for later rebuild with new
    joined relation)
selectJnStripJrel :: Join -> [([Integer],RDRelation)] -> [([Integer],
    RDRelation)]
selectJnStripJrel _ [] = []
selectJnStripJrel jn@( (id1,cols1), func, (id2,cols2) ) rel@((i,_):t) | id1
     `elem` i || id2 `elem` i = selectJnStripJrel jn t
  | otherwise = rel ++ selectJnStripJrel jn t
```

```
-- apply conditions from select
selectCond :: RDatabase -> [Integer] -> [Condition] -> [([Integer],
    RDRelation)]
selectCond _ [] _ = []
selectCond db (h:t) cond = [([h],(selectCondApply (getRelation db h) cond))
    ] ++ selectCond db t cond

-- apply condition list to relation
selectCondApply :: RDRelation -> [Condition] -> RDRelation
selectCondApply rel [] = rel
selectCondApply rel (h:t) = applyCondition (selectCondApply rel t) h




-- apply temporal conditions from select
selectTCond :: [([Integer],RDRelation)] -> [TempCondition] -> [([Integer],
    RDRelation)]
selectTCond [] _ = []
selectTCond (h:t) cond = [selectTCondApply h cond] ++ selectTCond t cond


-- apply list of temporal conditions to single relation
selectTCondApply :: ([Integer],RDRelation) -> [TempCondition] -> ([Integer
    ],RDRelation)
selectTCondApply rel [] = rel
selectTCondApply rel@(i,r) (h:t) = (i,applyTCondition (flattenRelC (
    selectTCondApply rel t)) h)
```

A.10 Tree

```
module TempDB.Tree where

{-

red/black balanced binary search tree
based on (okasaki, 2008)

prototype developed as a part of
"A Native Temporal Relation Database for Haskell"

Lukas Maczejka, 2009-2010


-}


import TempDB.BasicTypes
import TempDB.Time

------------ red black id/value tree
-----------------------------------


data Color = R | B deriving (Show,Eq)
data IVTree a b = IVEmpty | IVNode Color (IVTree a b) a b (IVTree a b)
    deriving (Show,Eq)


-- print tree to string
ivtShow :: (Show a, Show b) => IVTree a b -> String
ivtShow (IVEmpty) = "-"
ivtShow (IVNode c l x v r) = show x ++ show c ++ "=" ++ show v ++ "(" ++
    ivtShow l ++ "," ++ ivtShow r ++ ")"


-- member function
ivtMember :: Ord a => IVTree a b -> a -> Bool
ivtMember (IVEmpty) x = False
ivtMember (IVNode _ l y _ r) x | x < y = ivtMember l x
  | x > y = ivtMember r x
  | otherwise = True
```

```
-- insert new node into tree
ivtInsert :: Ord a => IVTree a b -> a -> b -> IVTree a b
ivtInsert s x v = IVNode B l z w r
  where
  IVNode _ l z w r = ivtIns s
  ivtIns IVEmpty = IVNode R IVEmpty x v IVEmpty
  ivtIns (IVNode B a y q b) | x < y = ivtBalance (ivtIns a) y q b
    | x > y = ivtBalance a y q (ivtIns b)
    | otherwise = IVNode B a y q b
  ivtIns (IVNode R a y q b) | x < y = IVNode R (ivtIns a) y q b
    | x > y = IVNode R a y q (ivtIns b)
    | otherwise = IVNode R a y q b

-- balance tree / insertion
ivtBalance :: IVTree a b -> a -> b -> IVTree a b -> IVTree a b

ivtBalance (IVNode R a x xv b) y yv (IVNode R c z zv d) = IVNode R (IVNode
   B a x xv b) y yv (IVNode B c z zv d)
ivtBalance (IVNode R (IVNode R a x xv b) y yv c) z zv d = IVNode R (IVNode
   B a x xv b) y yv (IVNode B c z zv d)
ivtBalance (IVNode R a x xv (IVNode R b y yv c)) z zv d = IVNode R (IVNode
   B a x xv b) y yv (IVNode B c z zv d)
ivtBalance a x xv (IVNode R b y yv (IVNode R c z zv d)) = IVNode R (IVNode
   B a x xv b) y yv (IVNode B c z zv d)
ivtBalance a x xv (IVNode R (IVNode R b y yv c) z zv d) = IVNode R (IVNode
   B a x xv b) y yv (IVNode B c z zv d)

ivtBalance a x xv b = IVNode B a x xv b



-- delete node from tree
ivtDelete :: Ord a => IVTree a b -> a -> IVTree a b
ivtDelete t i = ivtSimpleDelete t i


-- simple delete function: build new tree without element to delete
ivtSimpleDelete :: Ord a => IVTree a b -> a -> IVTree a b
ivtSimpleDelete IVEmpty x = IVEmpty
ivtSimpleDelete t x = ivtSDel t x IVEmpty
  where
  ivtSDel (IVNode c IVEmpty y yv IVEmpty) x nt | x == y = nt
    | otherwise = ivtInsert nt y yv
  ivtSDel (IVNode _ IVEmpty y yv r) x nt | x == y = ivtSDel r x nt
```

```
    | otherwise = ivtInsert (ivtSDel r x nt) y yv
  ivtSDel (IVNode _ l y yv IVEmpty) x nt | x == y = ivtSDel l x nt
    | otherwise = ivtInsert (ivtSDel l x nt) y yv
  ivtSDel (IVNode _ l y yv r) x nt | x == y = (ivtSDel l x (ivtSDel r x nt)
    )
    | otherwise = ivtInsert (ivtSDel l x (ivtSDel r x nt)) y yv



-- get highest id in index tree
ivtMaxId :: IVTree (RDBo, Timespan, Integer) [([String],RDBo)] -> Integer
ivtMaxId IVEmpty = 0
ivtMaxId (IVNode _ _ (_,_,x) _ IVEmpty) = x
ivtMaxId (IVNode _ _ _ _ r) = ivtMaxId r

-- get next id in index tree (ivtMaxId + 1)
ivtNextId :: IVTree (RDBo, Timespan, Integer) [([String],RDBo)] -> Integer
ivtNextId t = succ (ivtMaxId t)
```

A.11 Persistent Transactions

```
module TempDB.PersistentTransactions where


{-

persistent db transactions
after each db manipulation the indices are updated on disk

prototype developed as a part of
"A Native Temporal Relation Database for Haskell"

Lukas Maczejka, 2009-2010

-}

import TempDB.Data
import TempDB.BasicTypes
import TempDB.Tools
import TempDB.Time
import TempDB.Tree
import TempDB.Relations
import TempDB.Indices
import TempDB.Transactions
import TempDB.IO


-- execute a query and immedeately store it to disk
persistentQuery :: RDPersistentQuery -> IO RDatabase
persistentQuery (db, dbname, s) = do
  ndb <- query (db, s)
  storeToFile ndb dbname
  return ndb

-- resolve a delete and immedeately store it to disk
persistentDelete :: (RDatabase,String) -> RDSelect -> IO RDatabase
persistentDelete (db,dbname) s = do
 ndb <- delete db s
 storeToFile ndb dbname
 return ndb
```