TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Koit Saarevet 204716IVGM

METHODS OF ACCESS TO SERIES OF ARCHIVED DATABASE SNAPSHOTS

Master's Thesis

Supervisor: Innar Liiv
PhD

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Koit Saarevet 204716IVGM

MEETODID JUURDEPÄÄSUKS ARHIVEERITUD ANDMEBAASITÕMMISTE SEERIATELE

Magistritöö

Juhendaja: Innar Liiv
PhD

Tallinn 2024

Author’s Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Koit Saarevet

18.05.2024

Abstract

Archives around the world do a lot of database preservation, often in the form of regular snapshots. Some have done it for half a century. But nobody in the archival community has a tool for accessing data across multiple snapshots simultaneously, neither is there awareness of such a tool existing at all.

This thesis delivers that tool.

The research started from examining the state of the art in relevant fields to find either complete solutions (assuming they exist, just the community is unaware) or building blocks to create the solutions. The search did not detect any complete solutions, but identified temporal databases as a technology with the greatest potential. The work then went on to analyse an existing abstract algorithm for merging database snapshots, and to enhance it to suit the goals of the thesis. This was followed by the development of a prototype that can merge several snapshots of a database into one temporal database in MariaDB Relational Database Management System (RDBMS). Finally, the listing of the basic types of temporal queries was compiled, together with examples on their use.

Keywords: database preservation, database snapshots, archival access, temporal databases, MariaDB, DBPTK, SIARD, long-term digital preservation, digital archives, relational databases.

The thesis is written in English and is 90 pages long (of which 64 pages are the main document), including 6 chapters, 3 figures, 18 tables and 3 algorithms.

Annotatsioon

Meetodid juurdepääsuks arhiveeritud andmebaasitõmmiste seeriatele

Arhiivid üle maailma tegelevad palju andmebaaside säilitamisega, sageli regulaarsete tõmmiste kujul. Mõned on seda teinud juba pool sajandit. Kuid kellelgi arhiivikogukonnas pole tööriista mitme tõmmise andmete samaaegseks kasutamiseks, ja pole teada, et taoline tööriist üldse eksisteeriks.

Antud magistritöö raames loodi see tööriist.

Uurimine algas asjakohaste valdkondade taseme (state of the art) uurimisest, et leida kas terviklikud lahendused (juhul, kui need on olemas, aga lihtsalt kogukond pole teadlik) või siis ehituskivid uute lahenduste loomiseks. Otsinguga terviklikke lahendusi ei leitud, kuid suurima potentsiaaliga tehnoloogiana tuvastati ajalised andmebaasid (temporal databases). Seejärel analüüsiti olemasolevat abstraktset algoritmi andmebaasi hetktõmmiste ühendamiseks ajalisse andmebaasi ja täiustati seda, et see vastaks magistritöö eesmärkidele. Sellele järgnevalt loodi prototüüp, mis suudab liita sama andmebaasi mitu hetketõmmist üheks ajaliseks andmebaasiks MariaDB andmebaasimootoril. Lõpuks koostati loetelu tüüpilistest ajamõõtmega päringutest koos näidetega nende kasutamise kohta.

Märksõnad: andmebaasi säilitamine, andmebaasi hetktõmmised, juurdepääs arhiivile, ajalised andmebaasid, MariaDB, DBPTK, SIARD, pikaajaline digitaalne säilitamine, digitaalarhiivid, relatsioonilised andmebaasid.

Lõputöö on kirjutatud inglise keeles ja on 90 lehekülge pikk (millest 64 lehekülge on põhidokument), sealhulgas 6 peatükki, 3 joonist, 18 tabelit ja 3 algoritmi.

Acknowledgment

Thank you to Ingrid Pappel for luring me into the eGov program. Thank you to Aivi Remmelg and Anastasiia Dudko for taking care of me so that I survived the consequences.

Thank you to Kuldar Aas for suggesting the topic.

Thank you to Innar Liiv for patience and creativity in producing arguments why I should continue with the thesis work, and for providing precision-targeted hints to get me unstuck whenever it was necessary.

Thank you to Priit Raspel for teaching me the foundations of relational databases in an enlightening, entertaining and engaging way in 2004, and for also sharing his wisdom during the research for this thesis.

Thank you to Kristjan Kolde for recommending the Kimball's book and illuminating me the field of data warehouses.

Thank you to Antoon Bronselaer for publishing the article on temporal merges and thereby saving me from reinventing the wheel.

Thank you to Jaak Tepandi for sharing the detailed and practical guidelines for graduating students.

Thank you to my colleagues at the National Archives for tolerating my months-long aloofness in all topics but the thesis. Especially big thank you to Lauri Leht, who, I am sure, had to fend off many a complaint in this regard.

List of Abbreviations and Terms

API	Application Programming Interface
DBPTK	Database Preservation Toolkit
DSR	Design Science Research
DW	Data Warehouse
ERD	Entity Relationship Diagram
FK	Foreign key
JSON	JavaScript Object Notation, a plain text based, human- and machine-readable data presentation format
NAE	The National Archives of Estonia
PK	Primary key
RDBMS	Relational Database Management System
Relation	“Relation” is the abstraction of “table” in a database. It is a collection of tuples (rows), that consist of an ordered list of elements or domains (fields or columns), which have the same semantics for all tuples (rows). The schema of a relation defines the semantics for each positional element of the tuples (rows).
Relationship	A reference between tables, i.e., a foregin key field in a table referencing the primary key field of another table
SQL	Structured Query Language
SQL:2011	ISO/IEC 9075-2, the official standard for SQL, revision from 2011
XML	Extensible Markup Language

Table of Contents

List of Algorithms

List of Figures

List of Tables

1. Introduction

This chapter begins with setting the background: what is snapshot-based database preservation and how it is done. It then highlights a problem with the current state of affairs and formulates a research objective to tackle the problem. Further sections define specific research questions and some guiding principles to help stay on the right path. The chapter then explains the significance of the problem and of the planned solution. The final section describes the document structure.

1.1 Background

A large proportion electronic information systems use relational databases for storing their data. Some of this data needs to be preserved for the long term and the dominant method for doing that is database snapshots[1]. Historically, the combination of CSV files (for the data) and SQL files (for the structure) was the go-to choice of the archival format. The new millennium brought two XML-based formats, the first of which, DBML (Data Base Markup Language), developed by a group of Portuguese researchers, was announced in 2002 [1]. The other, SIARD (Software Independent Archiving of Relational Databases) was initiated at the Swiss Federal Archives in 2000 and got published in 2004 [2]. SIARD started to gain ground, becoming the official format for archiving relational databases in the Open PLANETS EU cooperation project in 2008 [3] and in the E-ARK project in 2014. SIARD has become the de facto standard format for database preservation in many countries (in addition to being a de jure e-Government standard in Switzerland [4]).

A strength of the XML-based formats is their independence of the RDBMS[2]. For example, SIARD relies on standards like SQL:2008, Unicode, XML, URI (RFC 3986) and ZIP, making it much more sustainable and portable than the native "dump" formats of the RDBMS [5, p. 1]. Another strength is self-containedness: both DBML and SIARD store the database structure definitions (originally formed using `CREATE TABLE` statements) with the necessary detail to rebuild the database from scratch (see [1] and [5, pp. 22-37]). On the foundation of these two characteristics, an interesting feature emerges: the possibility to use the archival formats as a medium for migrating data between different

[1]A perfectly valid question at this point is why just the database, not the whole information system, e.g., as an image of a virtual machine. The reason is: modern information systems consist of so many interconnected components that keeping the whole apparatus functioning reliably and securely over the long term is prohibitively costly.

[2]Relational Database Management System

RDBMS. Figure 1 illustrates some of the input and output options (called modules) in DBPTK (Database Preservation Toolkit). The source of the image is unknown, but a very similar one attributed to KEEP SOLUTIONS (the developer of DBPTK) is in [6].

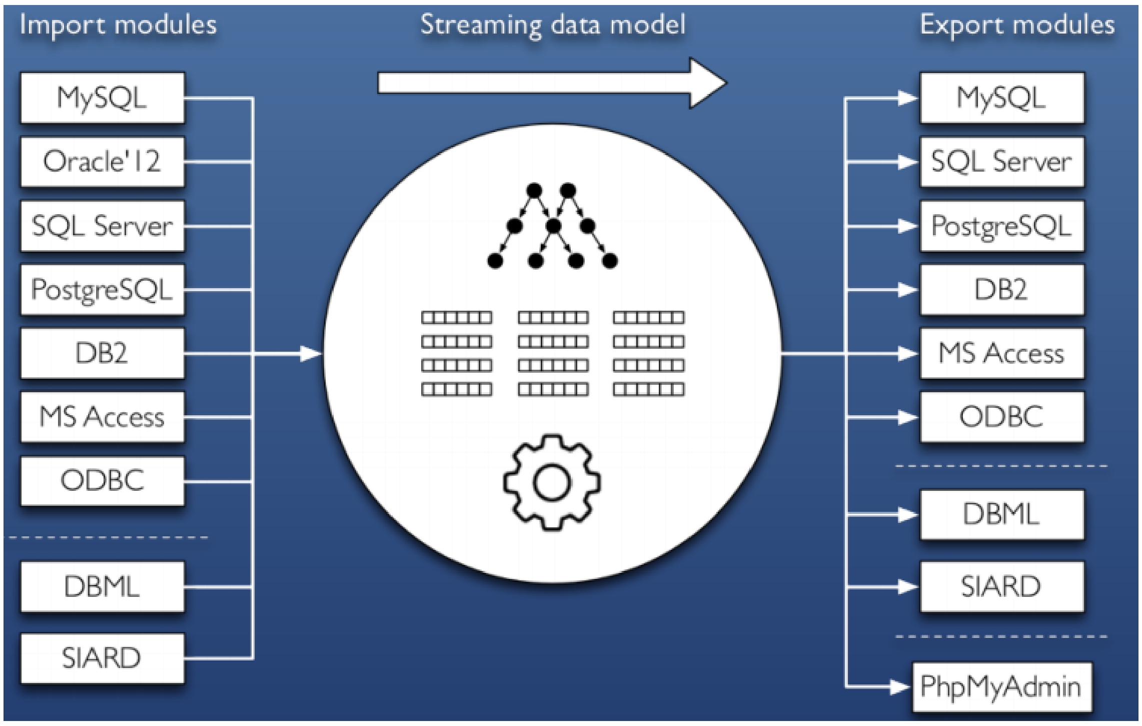

Figure 1. *RDBMS-SIARD-RDBMS transformations in DBPTK.*

Well-established tools exist for the creation of the SIARD snapshots. For example, The Swiss Federal Archives, when they originally created the SIARD format, also created a software package for creating and viewing the SIARD files. The program was initially called SIARD, but to reduce the confusion a bit, was then renamed to SIARD Suite. There are also commercial tools, like CHRONOS[3]. The SIARD creators produce one logical[4] SIARD file per snapshot. Assuming a one-year archiving interval[5], a series of snapshots will be produced as illustrated in Figure 2.

How are the snapshots accessed? One way is to use the migration feature from Figure 1 and load the data from the SIARD file into a live database engine, then use SQL for queries. The other, more common way is to use dedicated viewer tools, such as the SIARD Suite[6], DBPTK[7], dbDIPview[8], SIARDexcerpt[9] or others. These provide various options to examine the data without uploading the full snapshot into an RDBMS first.

[3]https://www.csp-sw.com/quality-management-software-solutions/data-archiving-with-chronos/
[4]In practice, external files from the original system and larger objects from the tables (so-called LOBs) are often stored outside the SIARD file.
[5]As of spring 2024, the target intervals for the archival creators in Estonia are set at 2-5 years.
[6]https://www.bar.admin.ch/bar/en/home/archiving/tools/siard-suite.html
[7]https://database-preservation.com
[8]https://github.com/dbdipview/dbdipview/
[9]https://github.com/KOST-CECO/SIARDexcerpt

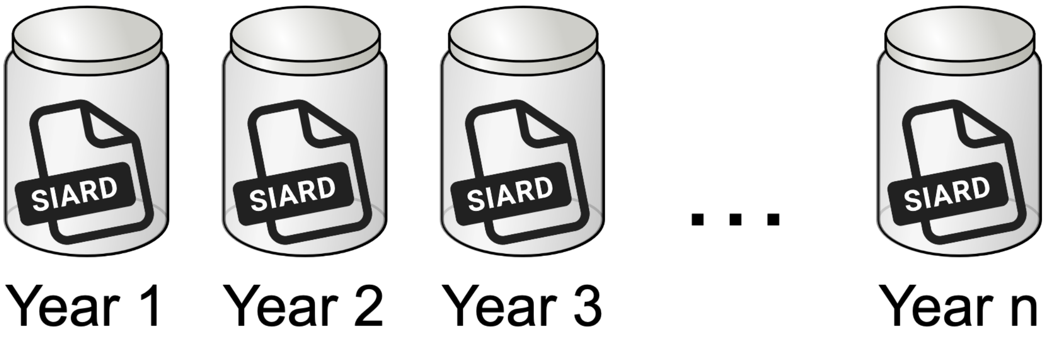

Figure 2. *A series of preserved SIARD snapshots.*

However, as already stated in the abstract, all these tools share a weakness: they handle one snapshot at a time. Even tools like the DBPTK (whose single instance can ingest and make available many snapshots) keep each snapshot separate with no cross-snapshot queries.

1.2 Problem statement

Imagine a public vehicle registry that stores data about vehicles, their owners and the insurance policies. Let's say it is an old-fashioned database where modifications to the data overwrite the previous values, which means that it only includes current data, no history. It is a somewhat artificial assumption for the vehicle ownership data, as modern vehicle registries probably keep some length of history, but the experience of the National Archives of Estonia (NAE) shows that there are still plenty of databases in production use where overwriting systematically occurs for some data fields. Now let us assume the database gets archived every five years in the form of a full snapshot to SIARD.

Business value and archival value can diverge. For instance, the transportation administration that operates the registry is tasked with maintaining traffic safety and lawfulness – every car must have a mandatory insurance policy, owner's data must be up to date etc. None of the agency's tasks require ownership data from five years ago, or the data about cars that have been scrapped. On the other hand, these exact data might be very attractive for a historian who wants to analyse the trends of cars per capita or the age of the fleet across decades.

Business value dictates the content of the database. Due to circumstantial overwriting of individual data elements and due to systematic deletion of out-of-date records, what is present in snapshot one is not fully there in snapshot five. Thus, no snapshot is complete, not even the most recent one. An illustration of this is presented in Table 1, which is a listing from the hypothetical vehicle registry. It is from the table `usage_log` that stores the count of users who interacted with the system that day. The data points are from a 16-year period, but due to the policy of regularly deleting the records that are older than ten years, the full list is not present in any one snapshot. `row_start` and `row_end` indicate the range of snapshots that contain this row of data. Note that the end date is exclusive, meaning that from this date, the row is not available any more. With archiving happening on January 1 of 2000, 2005, 2010, 2015 and 2020, the user would need to access the four snapshots from 2005-2020. 2005 cannot be left out, because 2010 already does not include the lines from the 2000 snapshot, and so on.

Table 1. *Table* `usage_log`*, full history*

date	**users**	**row_start**	**row_end**
1999-12-30	70	2000-01-01	2010-01-01
1999-12-31	50	2000-01-01	2010-01-01
2003-03-03	90	2005-01-01	2015-01-01
2004-04-04	75	2005-01-01	2015-01-01
2005-05-05	100	2010-01-01	2020-01-01
2007-07-07	105	2010-01-01	2020-01-01
2013-03-13	120	2015-01-01	2038-01-19
2014-04-14	130	2015-01-01	2038-01-19
2015-05-15	140	2020-01-01	2038-01-19
2017-07-17	160	2020-01-01	2038-01-19

This digging from four different snapshots is now all manual, no automation exists. The user is forced to load all the relevant SIARD files one after the other, work with each of them separately and then aggregate the results manually. That cumbersome process is further exacerbated if the data model has changed (i.e., the snapshots have different structures).

1.3 Objectives

The vision was of a bright future, where there is an application for quickly loading multiple snapshots, querying them with intuitive tools and visualising the results so they become instantly comprehensible. Plus, this all would tolerate schema evolutions, i.e., the structure of all snapshots not being identical.

The research objective was to materialise as much of the vision as possible, with a strong emphasis on usefulness in real life. Despite having worked for the National Archives for the whole 3rd millennium, this specific niche of digital preservation had stayed completely unexplored for the author. Consequently, there could be no detailed outline of the expected result. The most concise wording possible:

The objective of the research is to develop a method for accessing a series of archived database snapshots.

The success criteria:

1. It is possible to prepare several snapshots for simultaneous access.
2. It is possible to make cross-snapshot temporal queries.
3. The solution works with real SIARD files (i.e., not just a theoretical model).

1.4 Research questions

The main research question (MRQ) and two supportive research questions (SRQ) were worded.

MRQ: Is it possible to access a series of archived database snapshots simultaneously?

SRQ1: Are there existing tools for multi-snapshot access?

SRQ2: Are there theoretical methods for multi-snapshot access?

As a note added during final editing: the somewhat counter intuitive ordering of SRQ1 and SRQ2 stemmed from the vagueness of the objective, while there being an acknowledged emphasis on practice. Had the examination of the state of the art revealed a functioning software package, the focus of the research would have stayed there and the effort would have gone into determining its fit for purpose, developing an implementation plan etc. Only after failing to find an existing tool was the attention to be directed to theory.

1.5 Guiding principles

In the course of the research there were many situations that required small decisions between equally good alternatives. Pondering over these led to the distillation of the three principles that greatly sped up the choices at crossroads.

1. **Do not reinvent the wheel** – established tools, technologies, methods are preferred.
2. **Prefer open source** – the archival community loves it and has learned to collectively maintain the best tools.
3. **Avoid hacks** – non-standard use of tools may provide attractive shortcuts, but it introduces the risk of things breaking down.

As with all rules, they were occasionally broken, but only upon an informed decision.

1.6 Significance of the study

Initial confidence in the significance came from the fact that the thesis topic was proposed by Kuldar Aas, a colleague at The National Archives of Estonia 2002-2022 and since then the Data Governance Program Manager at the Ministry of Economic Affairs and Communications. Kuldar is (without exaggeration) one of the top experts in database preservation globally. He started working on database preservation in 2002, defended his master's thesis on it in 2004 [7], and led the efforts in the area at NAE until 2022. He was a founding father of the E-ARK project, which since 2014 has been developing the SIARD format and supporting the DBPTK software. Kuldar's assessment was, that at the time, there was no solution to the multi-snapshot access problem. This impression was confirmed by the author's own communications with the major actors in the international database archiving community, thus it would have required an extreme silo effect for the whole community to be clueless of a suitable solution existing, just in another field.

The problem is relevant to all organisations that archive databases: national archives, libraries, university archives, large corporations etc. The institutions in the regular communications circle of NAE include the National Archives of Denmark, Finland, Slovenia and the USA, The Swiss Federal Archives, The Hungarian Historical Archives of the State Security Services, State Archives of Schleswig-Holstein, Norwegian Digital Resource Center for Municipal Archives, and several others.

Some of them are at an early stage, others are experienced. For instance, the Schleswig-Holstein archives are conducting their first pilot project in archiving a database into the SIARD format. On the other hand, the Danish National Archives has over 40 years of experience, has preserved more than 4400 databases and ingests over 200 new ones per year [8].

The database preservation specialists of these agencies were asked during this research if they already have a solution to the multi-snapshot access problem and if they would be interested in getting one. All of them confirmed their being unaware of any solution and also expressed their interest in having one.

In sum, there is a significant likelihood of some people saying thank you for the results of this research.

1.7 Document structure

The remaining parts of the document are structured as follows.

Chapter 2 deals with the methodology. First, it introduces the Design Science Research framework, followed by the more specific work plan. Then it discusses the use of data. The final section is a summary on the use of generative AI.

Chapter 3 is literature review. It is divided into four thematic areas: Data warehouse, XML, Semantic web and Temporal database, each containing a selection of the most relevant or otherwise remarkable documental findings.

Chapter 4 is about the main results, of which there are three: 4.1 the merging algorithm in the abstract form, 4.2 the merging algorithm implemented and 4.3 the methods for querying. However, the third is realised using the standard features of MariaDB, so by the sheer amount of work that went into each, the first two are in a totally different league. The chapter is concluded by 4.4 (the tools and technologies used) and 4.5 (instructions on downloading the supplemental files).

Chapter 5 discusses the results. First, the evaluation: was the objective achieved? Second, contributions: which outcomes are useful and for whom. Third, future work: eight areas that are worth exploring further.

Chapter 6 summarises the thesis. It starts from recalling the objective and acknowledging its achievement. Then it provides a concise description of the multi-snapshot access method. The chapter wraps up by going over the three research questions and checking their answers.

The core part of the document is followed by the reference list and the publication license.

Appendix 2 contains the source code listings for all the original components of the prototype: a Bash script, three SQL stored procedures and one file with SQL `CALL` statements. There are actually two more code files that were left out for brevity. One is an alternative version of Main_workflow.sh that works with source data in the form of SQL `INSERT` statements (instead of the normal SIARD). The other is siard_create.sh, which extracts data from a live RDBMS and stores it as SIARD files. Both code files are included in the File supplement (see section 4.5).

Appendix 3 has brief installation instructions for the third party software (MariaDB, DBPTK, DBeaver).

Appendix 4 lists some deliberate formatting decisions. Each of them constituted a choice between two or more options, and time was spent on studying reputable sources to make an informed decision. With that much effort already invested, it seemed reasonable to document the decisions.

2. Method

This chapter outlines the foundational research method and then a more concrete plan. There is also a chapter on data, i.e., the databases that were used in the development of the prototype. The chapter ends with ruminations on the usefulness of Large Language Models.

The word "method" is overloaded in this document: it means both the method of access to database snapshots and the method of developing it. The current chapter is about the latter, i.e., the research method for developing the snapshot-access method.

This thesis is submitted for the candidacy for the degree of Master of Science in Engineering, therefore an engineering problem was chosen. The degree is pursued in the curriculum "E-Governance Technologies and Services," which is an amalgamation of technology and management subjects – this calls for the research method to also have a good balance.

2.1 Design Science Research

A great one with these characteristics is the Design Science Research (DSR) framework popularised by professors Hevner, March, Park and Ram in 2004 [9]. In another article by Hevner and March, they describe it as follows: "Design science seeks to create innovations, or artifacts, that embody the ideas, practices, technical capabilities, and products required to efficiently accomplish the analysis, design, implementation, and use of information systems" [10, p. 111].

They identify four types of artifacts: constructs, models, methods, and instantiations. Of these, methods and instantiations are relevant to the current thesis, quoted from [10, p. 111]:

- Methods define solution processes. They can range from formal, mathematical algorithms that explicitly define the search process to informal, textual descriptions of "best practice" approaches.
- Instantiations show how to implement constructs, models, or methods in a working system. They demonstrate feasibility, enabling concrete assessment of an artifact's suitability to its intended purpose. Researchers can use instantiations to learn about the real world, how the artifact affects

it, and how users appropriate it.

These are precisely the two artifacts the thesis tries to produce. Further, what fully stole the heart of this author was a glance at the guidelines [9, p. 83], reprinted here in Table 2 – these are delightfully reasonable and down-to-earth.

Table 2. *Design-science research guidelines*

Guideline	Description
1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Each of the guidelines is elaborated later in the article [9], some in great detail (e.g., 6: Evaluation is supplemented with a whole toolbox of evaluation methods). The guidelines are intentionally universal and therefore not applicable to a single case in their entirety. A selection is needed and Table 3 depicts the thesis author's plan for implementing the DSR guidelines.

Table 3. *DSR guidelines implementation plan*

Guideline	**Implementation plan**
P1: Design as an Artifact	This research aims to produce a viable method of accessing database snapshots and an instantiation of it that can work with real archival records.
P2: Problem Relevance	The objective is to build a solution to the snapshot access problem and to the best knowledge available, it is important and relevant to every archives around the world that collects regular snapshots of databases.
P3: Design Evaluation	The utility, quality, and efficacy will be rigorously demonstrated. Of the evaluation methods in Table 2 of [9, p. 86], experimental (mainly in the form of simulations), and testing (both black and white box) are planned. For the abstract algorithm, some static analysis is likely necessary.
P4: Research Contributions	The purpose of the research is to provide real contributions, the outcomes are described in section 5.2.
P5: Research Rigor	This research tries to strike a reasonable balance by first expressing the algorithm using mathematical formalism, then describe the implementation of it in more natural language. Such approach aligns well with a section of the guideline description: "Design-science research often relies on mathematical formalism to describe the specified and constructed artifact. /.../ Again, an overemphasis on rigor can lessen relevance" [9, p. 88].
P6: Design as a Search Process	The approach for this thesis is to be flexible and iterative: look for tools and methods in literature, when something is found then try to use it, test it, if it does not work, go to a new round.

Continues...

Table 3 – *Continues…*

Guideline	Implementation plan
P7: Communication of Research	The thesis document will be structured and worded clearly to maximise understandability for both tech and managerial audiences. Dissemination of the results will start right after the defence. An international database preservation interest group is one venue. Attempts will be made to publish the results in a journal and at conferences.

All of the above was in the plans already before discovering DSR, but it was reassuring to see that several smart people had given it a thought and considered it a reasonable way to solve problems. By the way, a lot of people seem to be of the same mind. While compiling the reference list, the author of this thesis accidentally noticed the changed citation count for [9] between the BibTeX file saved previously, and the Scopus website. After that, the figures were occasionally checked a few more times, and an interesting trend emerged:

Table 4. *DSR paper's citation count*

Date	Citations	Change
2024-04-18	9483	
2024-04-30	9495	+12
2024-05-14	9529	+34
2024-05-16	9595	+66

2.2 Work plan

Supported by the DSR framework, the following approach to the task was laid out:

- Search the web and academic literature databases to uncover an existing complete solution.
- Should that fail, review scientific literature to identify methods and tools that can be used as the basis for a solution.
- Analyse the candidates.
- Pick the most promising one.
- Try to develop a solution based on it, incl. both the abstract algorithm and a working prototype.

2.3 Data

The core data in this research are the database snapshots that were used to build and test the prototype. The snapshots fall into three categories:

- Custom-made for this thesis (Vehicle registry).
- Public sample databases (World [11], Sakila [12]).
- Items from NAE collections.

The reason for using artificial datasets is efficiency: it allows for testing of exactly the features that are needed in the current phase of development, without any unnecessary ballast. This is especially important in the early stages, were only very limited functionality is implemented and overly complicated data would simply crash the code. Therefore the majority of development was done on the vehicle registry. As the Entity-Relationship Diagram on Figure 3 demonstrates, `vehreg` is simple, yet contains realistic data structures, plus all the necessary variations of primary key that are needed for testing the merge algorithm.

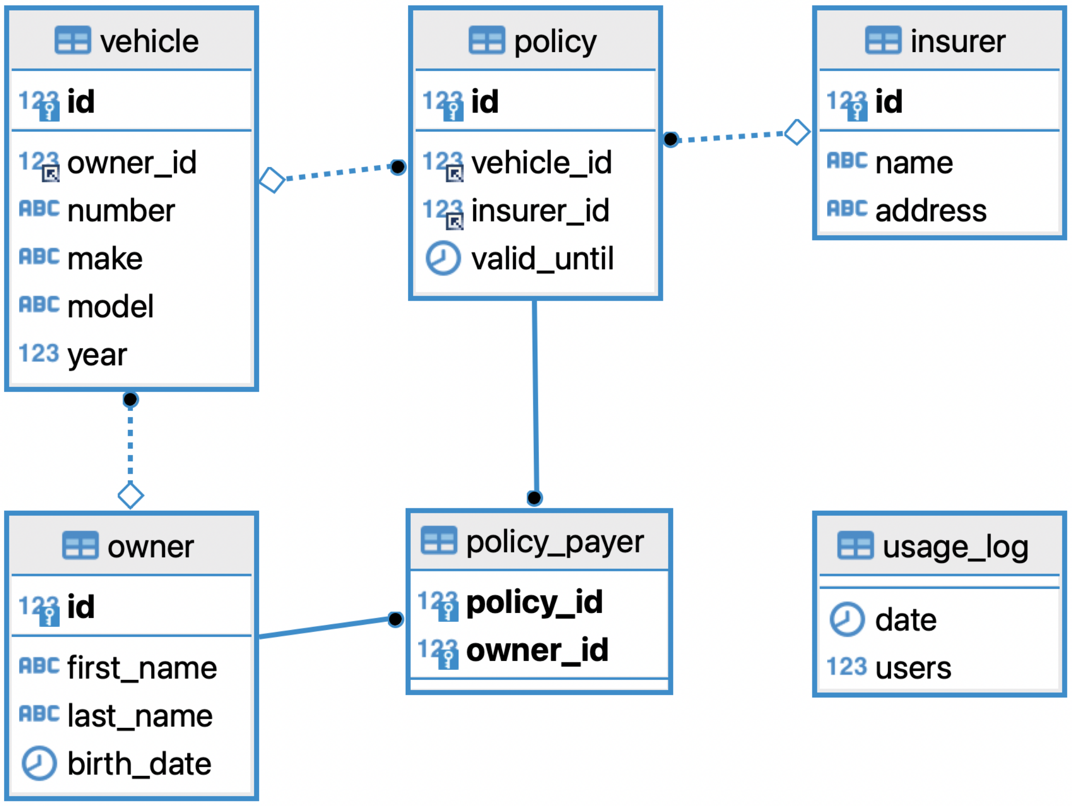

Figure 3. *Vehicle registry ERD.*

Later rounds of testing were done with World (slightly more data) and Sakila (more realistic structural complexity and even more data). The vehicle registry dataset is provided as part of the file supplement to the thesis (see section 4.5).

Another issue with the existing sample databases (such as World, Sakila and Employees) is that they are current databases, meaning that they contain only one set of values, while this research requires multiple sets of values from different points in time. A workaround from this is to create multiple copies of the database and alter them in a suitable manner, such as in the example in Listing 1. Here we have created three identical snapshots of the World sample database, named `world_s1`, `world_s2` and `world_s3`, intended to represent the state of affairs at the end of the year of 2000, 2005 and 2010. The data in World is from late 2000, so `world_s1` was already correct, the other two were updated for their years.

```sql
-- Estonian presidents' time in office:
-- Arnold Rüütel  08.10.2001 - 09.10.2006
-- Toomas Hendrik Ilves 09.10.2006 - 10.10.2016
-- Snapshot 2 valid at 31.12.2005
UPDATE world_s2.country SET HeadOfState = 'Arnold Rüütel'
WHERE Code = 'EST';
-- Snapshot 3 valid at 31.12.2010
UPDATE world_s3.country SET HeadOfState = 'Toomas Hendrik Ilves'
WHERE Code = 'EST';
```

Listing 1. Differentiating snapshots

Note that the updates were made to only a select few values, enough to have the differences to test the merging algorithm. Attempts were made to find newer vintages of World data, but none were found in accessible sources. There were many interesting time series datasets[1] that could have been transformed into yearly snapshots, but they were all single-relation, i.e., year-value pairs or sometimes year followed by several values, but no one-to-many relationships that are needed to simulate a real relational database.

The collection of archived database snapshots at the National Archives of Estonia was also examined for test data. Unfortunately, these real life databases tend to be large, often several terabytes, and therefore difficult to test on an ordinary laptop computer. Running multiple snapshots of these would have required the use of external storage and possibly would have still been too slow due to insufficient operating memory.

One target was too tempting to resist – the Land Mass Registry, of which there are four snapshots (the highest count in NAE collections), from the years 2019, 2020, 2021 and 2023. Unfortunately, they are archived not as SIARD files, but as CSV and this greatly

[1]For instance, https://ourworldindata.org/ has country data on fertility rate, obesity, diet compositions and type of government, and https://data.worldbank.org/ has per capita electric power consumption, all of which include annual (or almost annual) values for several decades.

complicates the task. For one, CSV does not come with proper structure definitions (compared to SIARD, which includes all the core aspects from the `CREATE TABLE` statements), which is especially troublesome when there are schema changes across snapshots. For the other obstacle, the `mariadb-import` utility is rather capricious to the idiosyncrasies of CSV files, e.g., the character encoding or the enclosing or not enclosing of values in quotes. These issues made the attempt take more time than was feasible for this thesis. Another try will be given after the thesis is submitted.

2.4 Use of generative AI

Attempts at using artificial intelligence (AI) in the form of Large Language Models (LLM) in this research were made, with moderate success. The earliest were the requests to ChatGPT 4 to produce code for merging the snapshots. The first result was awe-inspiring: a complete program with inline comments and detailed explanations on how it works. The attempt to execute it failed brutally, as did many subsequent ones to debug it using both artificial and biological intelligence. Several days of relentless tinkering with this hallucination were nevertheless useful, as they led to a broad vision on the architecture of the solution. Footprints of ChatGPT are still visible in the syntax of the SQL cursor that iterates over all tables in a database using the data in `INFORMATION_SCHEMA.TABLES`, and in some of the `JOIN` statements used in table-by-table merging. Splitting of the code between Bash scripts and SQL procedures was also inspired by ChatGPT's habit of proposing code snippets intermittently in these languages.

Another try was to get suggestions for the structure of the document from ChatGPT. The result was reasonable, but somewhat repetitive and incoherent. During writing, this draft structure was changed significantly and recurrently, and in the end looks much more like the one recommended by Prof. Jaak Tepandi in his guidelines for graduating students [13, Sec. 9.1] than the original output of the LLM. This seriously questions the value of LLMs supposedly tailor-made responses – yes, they are generated to the specific details in the prompt, but they lack the wisdom of a seasoned professor, who can outline a structure that is well suited for a whole category of thesis topics.

ChatGPT was also consulted while formulating Algorithm 3 (e.g., for the "update" statement in the comments to step 9). After a lengthy back-and-forth, the basic building blocks were in the LLM's output, but they had to be reshaped and reassembled to get the needed result.

Where ChatGPT really shone was concrete small tasks like rewriting name lists from “Firstname1 Lastname1, Firstname2 Lastname2” to the BibTeX format “Lastname1, Firstname1 and Lastname2, Firstname2" or creating the ISO 4 abbreviations of journal names, or "I have the CREATE TABLE statement and some tab-separated data. Please convert the data into INSERT statements."

In sum, the role of ChatGPT in this thesis can be described as that of a moderately knowledgable friend, who was always there to listen. The discussions helped clarify thoughts and get unstuck on some occasions, but did not provide complete solutions or brilliant insights.

3. Literature Review

This chapter summarises the search for tools, methods and algorithms that would answer the research questions SRQ1 and SRQ2 (see section 1.4). The areas covered were data warehouses, XML, semantic web and temporal databases.

The starting point was: any technology from any field is OK, so search was conducted using a variety of terms. Looking at it now, during the final editing, everything seems obvious: temporal databases were the correct path all along. This of course is hindsight bias at its purest. At the outset of the research, the author had not even heard the term temporal database, let alone comprehended its meaning.

The search effort started from terms describing the problem or the expected solution: access to archived databases, archived snapshot access, etc. Very little useful was found, thus the search was broadened, while keeping the term “archival” (or “preservation”): relational database preservation, database preservation, digital preservation database, digital archiving database. This produced a lot of interesting reading on our field in general, but still did not yield much for the concrete task at hand. Hints at archiving had to disappear completely as there are next to no publications of such solutions for archiving purposes. Searches for schema versioning, database snapshot, data versioning, etc. were when things got better, and discovering the word “temporal” was the final turning point.

Among the first terms to provide interesting results was data versioning, on which there is a long history of research. Data versioning denotes the situation where a data entity has recorded values reflecting different points in time, or alternatives that are valid in parallel, depending on the value of some property. In simpler cases, these needs can be solved by supplementing the table with a version column that holds either a timestamp or an identifier. More complex cases need more complex solutions, e.g., temporal databases.

Theoretical discussions were well under way in the early 1980s, e.g., in 1983, James F. Allen presented his interval algebra [14]. More concrete work followed, e.g., in a paper from 1991, Edward Sciore discusses the use of annotations to support versioning in object-oriented databases [15]. In 1994, the TSQL2 Language Design Committee led by Richard Snodgrass published the TSQL2 Language Specification [16]. After years of struggle, the ideas from TSQL2 finally made it to the SQL standard in 2011 [17].

Data versioning mostly looks at versions of data inside a single database instance, not at versions of the same data in different database snapshots. However, some methods and tools have potential use for snapshots, too. For instance, OrpheusDB is a system "to 'bolt-on' versioning capabilities to a traditional relational database system that is unaware of the existence of versions," [18], so there might be a way to attach it to databases prior to archiving, to enable versioning, and at a later date aggregate the versioning-enabled snapshots into a single instance for easy access.

A similar approach is TSAPI (Temporal SQL API): a method for adding system-versioning functionalities to existing relational databases without changing the existing tables [19]. The result is achieved by creating special history tables and a set of triggers[1] to maintain them. The researchers have created a prototype implementation on MySQL, together with web-based user interfaces, while the "API" in "TSAPI" signals the possibility of accessing the functionalities via web services [19].

The last interesting paper from the early phase of literature review deals with the efficient creation of "temporal merges," a term defined as "a joint representation of a set of snapshots of the same database at different points in time," [20, p. 473]. At first, it did not properly register. It took an odyssey through data warehouses, XML and semantic web to realise the value of temporal relational databases and the brilliance of this specific article. For it is this very paper by Antoon Bronselaer and colleagues that contains Algorithm 1 for direct temporal merges – the key to solving our research problem.

Such wandering and stumbling advancement is well in line with the description of DSR guideline 6: Design as a Search Process, as explained in [9, pp. 88-90] – it is iterative and heuristic.

3.1 Data warehouse

At the first glance, data warehouse (DW) seems a perfect tool for bringing together multiple snapshots of archived data for a comfortable analysis. After all, data warehouse is defined as "[A] collection of integrated, subject-oriented databases designed to support the DSS function, where each unit of data is relevant to some moment in time" [21, p. 389]. DSS stands for Decision Support System, hinting at the business ancestry of DW. In the definition, the words "collection of databases" and "each unit of data is relevant to some

[1]A trigger is a stored procedure that executes automatically upon a certain event, e.g., a BEFORE INSERT trigger runs after the user calls an INSERT statement, but before the RDBMS writes the values into the table, so it can be used to copy the existing values into a history table before overwriting them by the new values provided by the user.

moment in time" are tempting like siren song.

The author of the thesis was lucky to have a chance to consult with an expert at an early stage in the research. The expert – Kristjan Kolde, the Data Warehouse Manager at Health and Welfare Information Systems Centre (TEHIK) – recommended the canonical textbook "The Data Warehouse Toolkit" by Ralph Kimball and Margy Ross [22]. This engaging text provided not only a thorough understanding on how data warehouses are designed and operated, but also the realisation, that they will not provide the two-click solution that they appear to.

The key is how the data gets from the original form in the databases into the final form that is usable in the data warehouse. This is brilliantly explained in the other canonical textbook "Building the Data Warehouse," written by William H. Inmon, the other half of the Inmon-Kimball holy binity of DW foundational theory. Page 19 of [21] reads:

> "In every environment the unintegrated operational data is complex and difficult to deal with. This is simply a fact of life. And the task of getting one's hands dirty with the process of integration is never pleasant. In order to achieve the real benefits of a data warehouse, though, it is necessary to undergo this painful, complex, and time-consuming exercise."

The powerful query and analysis capabilities of data warehouses are realised through intelligent integration and remodelling of the source data. The two books are obviously ancient, but the expert confirmed that what they encapsulate is still the state of the art. In sum, data warehousing has great potential for certain kinds of use cases of archived snapshots, but it is definitely not the low-hanging fruit that has to be picked first.

3.2 XML

SIARD, the widely used format for the preservation of relational databases is based on XML. There is another similar preservation format called DBML (Data Base Markup Language), introduced in the paper "Bidirectional Conversion between XML Documents and Relational Data Bases" [1]. That bidirectionality of the conversion RDBMS $\leftrightarrow$ XML is shared by both formats, implying that XML is a viable direction to search for the means of multi-snapshot access to relational data.

Several methods of handling temporal data in XML have been proposed.

In 2004, F. Currim, S. Currim, Dyreson and Snodgrass (the latter two very well-known experts in the field of temporal databases) presented a comprehensive solution for creating a temporal schema from a non-temporal schema, called τXSchema. They handle temporality through annotations, which "… specify which portion(s) of an XML document can vary over time, how the document can change, and where timestamps should be placed" [23]. The authors then further elaborated the method on the example of biological data from the National Center for Biotechnology Information (NCBI) [24]. A concise explanation is provided in [25]:

> "τXSchema has a three-level architecture for specifying a schema for time-varying data. The first level is for the *conventional schema* which is a standard XML Schema document that describes the structure of a standard XML document, without any temporal aspect. The second level is for the *logical annotations* of the conventional schema, which identify which elements can vary over time. The third level is for the *physical annotations* of the conventional schema, which describe where timestamps should be placed and how the time-varying aspects should be represented."

The original team developed it further over the years, e.g., at a conference in 2006 [26] (and then in 2008 twice as thoroughly in a journal [27]) they presented schema versioning for τXSchema. This is for the situation where not only the data values change but also their structure. Then the team of Brahmia, Bouaziz, Grandi and Oliboni took over and further developed schema versioning, defining "… a sound and complete set of schema change primitives …" that are needed "for the management of schema versioning in the τXSchema framework" [25]. As this work focused on changes in the levels two (logical annotations) and three (physical annotations), the logical next step was level one – the conventional schema, which they tackled in the 2012 paper [28].

The authors continued by proposing a language for updating the temporal XML data in the τXSchema framework. This was done through an extension to the W3C XQuery Update Facility Language (XUF), called τXUF [29]. In parallel, they ported τXSchema to the world of JSON, with what they called τJSchema – Temporal JSON Schema, a framework for managing temporality in JSON documents [30]. From there, they proceeded to invent τJUpdate – a language for modifying temporal JSON data [31].

The above 20-year τ-journey illustrates the breadth, depth and consistency of research around temporal XML. But it is by no means the only body of research. Just one example: in 2008, Rizzolo and Vaisman [32] introduced one that allows storing several timestamped versions of data in an XML tree and provided algorithms for validating the document against temporal constraints. For querying they created the TXPath language, an extension of XPath 2.0 [32].

By now it should be clear that temporal data management in XML is mature, with long-pedigree methods competing with the newcomers and no lack of choice for the user. However, there is a problem, quoting [28]:

> "Whereas schema versioning is required by several applications using multi-temporal XML repositories, both commercial XML tools (like Stylus Studio or XML Spy) and commercial DBMSs (like Oracle 11g, Tamino, or DB2 v.9) have no support for that feature until now, as surveyed in [6]. Therefore, XML Schema designers and developers use ad hoc methods to manage schema versioning."

The quote is from 2012. In 2024, googling for the temporal XML query languages TTXPath, τXQuery and TXPath, and τXSchema, still almost exclusively return links to scientific journals and university web sites. Support by software has not caught up, which greatly reduces the hope of building a temporal access solution on XML.

3.3 Semantic web

There is a lot of interesting research done around the semantic web technologies. At its core is RDF (Resource Description Framework) for representing data as triples, on top of it RDFS (RDF Schema) and OWL (Web Ontology Language) for including ontology elements in RDF. Databases of RDF data are called triplestores and are queried using the SPARQL language.

These technologies are often used for publishing open data, which in turn is usually just data derived from relational databases. Another use is semantic interoperability – making sure the meaning of data entities is defined unambiguously and presented in a machine-readable form. There are many tools[2] for converting data from relational databases to RDF, incl. some that derive the schema of the conversion automatically from the database structural definitions.

Management of change over time in the semantic web is an active research area, e.g., see

[2]See https://www.w3.org/wiki/ConverterToRdf and https://www.w3.org/wiki/RdfAndSql for links to over ten of them.

"RDF for temporal data management – a survey" from 2021 [33]. Many temporal query languages have been developed – T-SPARQL, stSPARQL, SPARQL-ST, SPARQ-LTL and AnQL are mentioned in [34]. The latter paper is also interesting for its use of the term "RDF archives." It is not concisely defined in the paper, so it largely looks synonymous to "RDF database" or "triplestore," but the repeated mentioning of archive.org implies that the authors probably have a longer term preservation in mind.

The authors of [35] propose using RDF as a format for long-term preservation of data stored originally in relational databases. They formulate the use case: "The proposed approach is suitable for archiving scientific data used in scientific publications where it is desirable to preserve only parts of an RDB" [35, p. 1].

All of the above is intellectually stimulating, but the author of the thesis cannot get over the feeling that relational databases and semantic web are parallel universes with completely different physics. In principle, the same things can exist, but to understand the way they are constructed and presented, one must acquire a large body of new knowledge. It is therefore an unlikely candidate for a quick and easy method for accessing database snapshots.

3.4 Temporal database

The need for recording time varying information in databases has been recognised at least since the 1970s [36, Sec. 1].

Temporal databases are a category of relational databases that include special time dimensions. They are of interest for this research because there is a solid body of theory on handling these time dimensions and there are some RDBMS that have built-in functionalities for time manipulations.

"The consensus glossary of temporal database concepts – February 1998 version" [37] defines the term *temporal database* through the inclusion of time: "A temporal database is a database that supports some aspect of time, not counting user-defined time." The glossary goes on to define three flavours of time:

- *Valid time* – "The valid time of a fact is the time when the fact is true in the modeled reality. /. . . / Valid times are usually supplied by the user."
- *Transaction time* – "A database fact is stored in a database at some point in time, and after it is stored, it is current until logically deleted. The transaction time of a database fact is the time when the fact is current in the database and may be retrieved. /. . . / Transaction times may be implemented using transaction commit times, and

are system-generated and -supplied.”

- *User-defined time* – “… is parallel to domains such as “money” and integer– unlike transaction time and valid time, it has no special query language support. It may be used for attributes such as ‘birth day’ and ‘hiring date.’” [37]

From the above, user-defined time is the clearest – it is any user-defined time attribute that is not about the validity of the row in a database. The other two warrant some more exploration.

Tom Johnston does a great job at explaining the whole paradigm of temporal databases in his book “Bitemporal Data: Theory and Practice” [38]. In there, he introduces his own naming for the two times implied in “bi”: “‘State time’ is my term for what computer scientists, vendors, and the SQL standards – i.e. pretty much everybody else – calls ‘valid time’. ‘Assertion time’, excluding its extension into future time, is my term for what nearly everybody else calls ‘transaction time.’” [38, Preface, p. xxix]. His rationale is that in tables that have the *valid time* attributes, each row refers to a different state of the data, or put another way: each row represents a timeslice from the life history of the data, thus *state time* [38, p. 8]. And the *transaction time* periods mark the time during which a row of data is asserted to make a true statement, thus *assertion time* [38, p. 9].

This discussion cannot be concluded without further muddying the waters, as the SQL standard (section “Introduction to periods,” [17, Sec. 4.14.1] for SQL:2011 and [39, Sec. 4.16.1] for SQL:2023) and some RDBMS vendors, like IBM [40, p. 205] and MariaDB [41], have their own terms, all summed up in the following table:

Table 5. *Variations in naming of the time periods*

Consensus glossary	**Tom Johnston**	**SQL, IBM, MariaDB**	**Definition**
Valid time	State time	Application time[3]	Time when the fact is true in the modeled reality
Transaction time	Assertion time	System time	Time when the fact is current in the database

System time and *system versioning* (i.e., the process of managing system time) are the preferred terms in this document because they are used by MariaDB and thus appear in the SQL code listings.

[3]In its SQL dialect, IBM uses `BUSINESS_TIME` for application time.

Finally, the temporal capabilities of RDBMS were explored.

Commercial RDBMS have dealt with time for a long time. For instance, Oracle has had its flavour of system versioning, called Flashback, since at least Oracle9i, released in 2002 [42]. Teradata implemented TSQL2, a predecessor to SQL:2011 temporal standard, in 2010 [38, p. xv]. IBM implemented SQL:2011 in DB2 already before its release, based on drafts circulating in the standardisation work groups, although it might have been easier for them than it seems, as they claim to have been the source of the approach that got adopted: "... the ANSI and ISO SQL committees have accepted an early IBM proposal for tables with system time periods" [43].

And we should not forget that several of the key researchers have worked for database vendors (e.g., Edgar F. Codd at IBM in 1970 [44]) or have been funded by them (e.g., Richard T. Snodgrass by IBM in 1995 for writing his book "The TSQL2 Temporal Query Language" [45, p. xxiii]). Currently, temporal features are known to exist in at least IBM DB2, Oracle DBMS, Teradata, and MS SQL Server commercial products [46].

In the open source camp, the choice is narrower. PostgreSQL has provided temporal functionalities via external modules [47, p. 22], some of which got later included in the core, most notably range data types (introduced in 2012 [48]). However, PostgreSQL does not have built-in support for SQL:2011, specifically for the features *T180 System-versioned tables* and *T181 Application-time period tables* [49]. MySQL is not known to have built-in temporal support either, but its sister MariaDB has a largely SQL:2011 compliant set of bitemporal features [41].

4. Results

This chapter presents the core output of the research: the abstract algorithm and its concrete implementation. It starts from section 4.1 that lays out the necessary terminology and then deep-dives into relational algebra and the SQL standard, explaining the rationale for the algorithm, and the steps of its functioning. The theory is followed by practice in section 4.2, where the prototype implementation is introduced. The third major part is 4.3, which explains why this whole endeavour makes sense, by showing how easy it is to query the data across many snapshots. The chapter is concluded by a brief overview of the tools and technologies involved in the prototype, and a note on where to find the remaining artefacts of this research (i.e., outside the thesis document itself).

The solution to the research problem (i.e., the method of multi-snapshot access) consists of two parts:

1. Preparation – merging multiple snapshots into one temporal database.
2. Access – performing queries on the temporal database.

4.1 Solution part 1: Merge snapshots – Theory

This section describes part one, and more narrowly, its theoretical side. At its core it is a variation of what Bronselaer et al. call a "direct merge," which itself is a kind of "temporal merge" [20, p. 477].

4.1.1 Terminology

Before diving into the details, it is necessary to establish the symbols that will be used and this is done in Table 6, which is a copy of Table 1 in [20], minus items that are not relevant to this thesis, plus some new ones that are.

Table 6. *Overview of symbols and key concepts*

Symbol	**Meaning**
R	Relation
$\mathcal{R}$	Schema of R
R^*	Temporal relation

Continues…

Table 6 – *Continues...*

Symbol	**Meaning**
D	Database
$\mathcal{D}$	Schema of D
D^*	Temporal database
$D_{(i)}$	Snapshot i of database D
$R_{(i,j)}$	Relation j of snapshot i of database D
$K_{\mathcal{R}}$	Primary key[1] of relation schema $\mathcal{R}$
$R[K]$	Projection over K, i.e., the primary key attribute(s) of relation R
$R[\mathcal{R}]$	Projection over $\mathcal{R}$, i.e., all attributes of relation R
S	Start attribute of validity interval
E	End attribute of validity interval
t_i	Timestamp at the moment of creation of shapshot i
t_{max}	Maximum value allowed by the data type of timestamps
r_i	Tuple r as of shapshot i

"Relation" is the abstraction of "table" in a database. It is a collection of tuples (rows), that consist of an ordered list of elements/attributes/domains (fields/columns), which have the same semantics for all tuples (rows). The schema of a relation defines the semantics for each positional element of the tuples (rows). Relations have their roots in mathematics, but their use in databases was proposed by Edgar F. Codd in 1970 [44] (see especially Section 1.3).

Unfortunately, the word "relation" has become to be used mostly in the sense of references between tables (i.e., a foreign key field in a table referencing a primary key field of another table), which in some contexts causes confusion. Thus, in this document, "relation" is always in the sense used in relational algebra (i.e., basically a synonym to "table") and the references between tables are denoted with the word "relationship." The term is used similarly by the long-time researcher of temporal databases Abdullah Uz Tansel, e.g., in [50, Sec. 4.3].[2]

"Key" and some of its flavours need to be defined. "In the relational model, a key for a relational schema is a set of attributes whose value(s) uniquely identify a tuple in a valid instance of the relation" [51, p. 1587]. It can be one attribute in the relation (i.e., one

[1]Bronselaer et al. use K for natural key

[2]To add to the confusion, Codd introduced another meaning to "relationship": content-identical domain-unordered relations [44, Sec. 1.3]. E.g., `Employee(id, first_name, last_name)` and `Employee(id, last_name, first_name)` are different relations, but one relationship. However, this definition seems to have completely fallen out of use.

column in the table) or the combination of several attributes, important is that the set of values that form the key never repeat.

In the relational model, duplicate tuples (i.e., where all attribute values are shared between tuples) are not allowed, thus by definition, there always exists at least one key. In practice, RDBMS[3] allow duplicates and consequently tables without a key.

"Candidate key." There can be several attributes or combinations of them that have the unique identification property – these alternatives are sometimes referred to as candidate keys. [51, p. 1587]

"Natural key." "Natural keys are meaningful values that identify records, such as social security numbers, that identify specific customers, calendar dates in a time dimension, or SKU numbers in a product dimension. A natural key is a column that has a logical relationship to other pieces of data within the record." [4] [52]

"Surrogate key" aka "synthetic key." "… surrogate keys are meaningless generated values that uniquely identify the rows in a table." Their merit is in durability (they do not have to change to reflect changes in the real world) and compactness (they are usually just one numeric attribute). [53].

"Primary key" is the key (out of all candidates) that is chosen to serve as the unique identifier [51, p. 1587]. In the context of this thesis, the formal declaration of the primary key in the definition of the relation schema (i.e., in practice, in the `CREATE TABLE` statement) plays an important role.

"Foreign key" is an attribute (or a set of them) that refer to an attribute (or a set of attributes) in another relation. More formally, according to the SQL standard, FK is a constraint that restricts the values of the referring attributes to the values present in the referred attributes [17, Sec. 11.8]. In practice, FK usually refers to a PK.

"Projection," in simple terms, is a vertical slicing of a tuple (row) or a whole relation (table). E.g., if we have a tuple t with attributes $\{a_1, a_2, a_3\}$, then the "projection of t over a_2," expressed as $t[a_2]$, is what is left of the original t when a_1 and a_3 are removed.

[3]At least MariaDB is known to allow.

[4]This definition is from the domain of data warehousing, thus the use of the word "dimension" that has a special meaning there.

4.1.2 Foundations

Bronselaer et al. define *temporal merge* as follows [20, Definition 2]:

“**Definition 2** (*Temporal merge*) Let D be a relational database with schema $\mathcal{D}$. A temporal database D^* is a temporal merge of the snapshots $D_{(1)}, \dots, D_{(m)}$, denoted by $D^* = D_{(1)} \oplus \dots \oplus D_{(m)}$, if we have for any $i \in \{1, \dots, m\}$ that we can reconstruct $D_{(i)}$ from D^*, denoted by $D^* \vdash D_{(i)}$.”

Based on that, they define *direct merge* as follows [20, Definition 3]:

“**Definition 3** (*Direct merge*) A temporal database D^* with schema $\mathcal{D}$ is called a direct merge if, for any i, we have that:

$\forall R_{(i)} \in D_{(i)} : R_{(i)} = R^*_{S \leq i \wedge i \leq E}[\mathcal{R}_{(i)}]$.”

Definition 3 has a problem: index i is overloaded. This might be acceptable in abstract formulations, but makes comprehension more difficult. It places $R_{(i)}$ and $D_{(i)}$ in different ontological categories: $D_{(i)}$ is a single item, the snapshot i in a set of snapshots of database D, while $R_{(i)}$ is a collection, the set of all relations in the database snapshot $D_{(i)}$. Having $D_{(i)}$ and $R_{(j)}$ would fix it:

$\forall R_{(j)} \in D_{(i)} : R_{(j)} = R^*_{(j)_{S \leq i \wedge i \leq E}}[\mathcal{R}_{(j)}]$.

An even clearer notation would be one with dual indices $R_{(i,j)}$, conveying the meaning “relation j of snapshot i of database D”:

$\forall R_{(i,j)} \in D_{(i)} : R_{(i,j)} = R^*_{(S \leq i \wedge i \leq E, j)}[\mathcal{R}_{(j)}]$.

Bronselaer et al. go on to formulate an algorithm for direct merges[5], i.e., the addition of snapshots into a temporal database that already contains at least one snapshot [20, Algorithm 1], printed here verbatim as Algorithm 1.

There are three weaknesses in Algorithm 1.

Weakness 1: it shares the overloaded index i issue with Definition 3. In addition to complicating abstract discussions, it can cause errors in implementations if insufficient

[5]According to the authors, the algorithm is actually for *minimal direct merge* as it is “minimal in the number of tuples they require to model all snapshots” [20, p. 480].

Algorithm 1 Addition of $D_{(i)}$ to $D_{(1)} \oplus \cdots \oplus D_{(i-1)}$

```
Require: Snapshot $D_{(i)}$
Require: $D^* = D_{(1)} \oplus \cdots \oplus D_{(i-1)}$
Ensure: $D^* = D_{(1)} \oplus \cdots \oplus D_{(i)}$
 1: Get $D_{(i-1)}$ from $D^*$
 2: Sort relations in $D_{(i)}$ s.t. $R_1 \prec R_2$ if $R_2 \rhd R_1$
 3: for all $R_{(i)} \in D_{(i)}$ do
 4:     for all $r \in R_{(i)}$ do
 5:         if $r \notin R_{(i-1)}$ then
 6:             insert $r$ in $R^*$ time $[i, i]$
 7:         else
 8:             update $r$ in $R^*$ set $E = i$
 9:         end if
10:     end for
11: end for
```

care is taken in translating the pseudocode to an actual programming language. Namely, if the traditional for-loop is used and i is adopted as the control variable.

Weakness 2: step 6 "**insert** r **in** R^* **time** $[i, i]$" cannot be executed for system-versioned tables[6] in an SQL:2011 compliant RDBMS, as the standard requires that at `INSERT`, the RDBMS sets `row_start = timestamp` and `row_end = max_timestamp`. **time** $[i, i]$ can be achieved if the RDBMS has relevant non-standard extensions.

In the case of MariaDB, there are two and one of them provides a solution. System variable `@@timestamp` allows overriding the normal behaviour where the `row_start` and `row_end` are filled with the current time (or `max_timestamp` for `row_end` in the case of current records). When `@@timestamp` is set to a specific value, this value is used in all cases that would normally get the current timestamp. This directly solves `row_start` for the newly inserted rows. To fix `row_end` as well, every `INSERT` statement must be followed by a `DELETE`. The process is illustrated by the following example.

```sql
SET @@timestamp = UNIX_TIMESTAMP('2022-02-22 22:22:22');
INSERT INTO insurer (id, name, address)
  VALUES (3, 'Ergo', 'Main Str. 1');
DELETE FROM insurer WHERE id = 3;
SELECT *, row_start, row_end
  FROM insurer FOR SYSTEM_TIME ALL
  WHERE id = 3;
```

Listing 2. Insert row with `row_start = row_end`

[6]Similarly to this thesis, the article by Bronselaer et al. explicitly deals with system time aka transaction time: "… in what follows, we focus on the transaction time of databases" [20, Sec. 4.1]

The `INSERT` statement will produce the row with the desired `row_start` value:

id	name	address	row_start	row_end
3	Ergo	Main Str. 1	2022-02-22 22:22:22.000	2038-01-19 05:14:07.999

Then the `DELETE` statement will set the `row_end` value:

id	name	address	row_start	row_end
3	Ergo	Main Str. 1	2022-02-22 22:22:22.000	2022-02-22 22:22:22.000

The other extension in MariaDB that gives hope for **time** $[i, i]$ periods is the system variable value `SET @@system_versioning_insert_history = 1`, which switches off the timestamp management altogether and allows for `INSERT` statements with explicit values for `row_start` and `row_end`. However, it turns out this mode enforces `row_start < row_end` and is thus not usable for the current goal.

Weakness 3: similarly to step 6, step 8 "**update** r **in** R^* **set** $E = i$" cannot be executed in an SQL:2011 compliant RDBMS, as it automatically sets `row_end = max_timestamp`. The same workaround as with step 6 could be applied here.

The following Algorithm 2 is a version of Algorithm 1 by Bronselaer et al., where the above-mentioned weakness 1 has been addressed.

Algorithm 2 Addition of $D_{(i)}$ to $D_{(1)} \oplus \cdots \oplus D_{(i-1)}$ (improved)

```
Require: Snapshot $D_{(i)}$
Require: $D^* = D_{(1)} \oplus \cdots \oplus D_{(i-1)}$
Ensure: $D^* = D_{(1)} \oplus \cdots \oplus D_{(i)}$
 1: Get $D_{(i-1)}$ from $D^*$
 2: Sort relations in $D_{(i)}$ s.t. $R_1 \prec R_2$ if $R_2 \triangleright R_1$
 3: for all $R_{(i,j)} \in D_{(i)}$ do
 4:     for all $r \in R_{(i,j)}$ do
 5:         if $r \notin R_{(i-1,j)}$ then
 6:             insert $r$ in $R^*$ time $[i, i]$
 7:         else
 8:             update $r$ in $R^*$ set $E = i$
 9:         end if
10:     end for
11: end for
```

Step 3 is to be read: "for all relations j in snapshot i do." Step 4 is to be read: "for all tuples in relation j of snapshot i do."

4.1.3 Algorithm

The core product of this thesis is the algorithm for merging the archived database snapshots into a temporal database. The algorithm has two manifestations: the abstract one, expressed in the terminology of relational algebra, and the concrete one, expressed as Bash scripts and SQL procedures. This section presents the abstract manifestation, labelled Algorithm 3. ChatGPT v.3.5 was used in drafting some of the formulas [54].

Although still abstract, it includes elements that make practical implementation on a temporal RDBMS easier than that of Algorithm 1. The most apparent of these, almost doubling the number of steps, is the special treatment for relations that have a formally defined primary key (see the explanation of Step 4 below the algorithm listing). Another is the introduction of timestamps in addition to snapshot indices. In most contexts, indices are a more efficient tool for addressing the snapshots, but for the values of `row_start` and `row_end`, the SQL standard and the RDBMS examined for this thesis[7], mandate timestamps. Thus, it makes sense to also use timestamps in the abstract algorithm.

Step 1 is important for the abstract algorithm, while in practical implementations, it is performed implicitly whenever calls to $D_{(i-1)}$ or its components are made.

Step 2 sorting of the relations is retained for universality, while in practical implementations it can usually be bypassed. The SQL:2011 standard[8] includes the optional feature F492, "Optional table constraint enforcement" that allows the constraints to be disabled by supplementing their definition with the keyword `NOT ENFORCED` [17, Sec. 10.8]. When applied to all foreign key constraints, tables can be processed in any order, with no regard to their functional dependencies. Obviously, this can break referential integrity if the data or the merging algorithm are faulty. In the current case, data can be assumed to be of perfect referential integrity as the snapshots are created from real, functioning databases and are technically validated after creation.

RDBMS support of the feature F492 is not uniform. IBM DB2 is claimed to be perfectly compliant [55]. Microsoft SQL Server provides a close-to-the-standard keyword that can be invoked using `ALTER TABLE emp NOCHECK CONSTRAINT FK_emp_dept_dept_id;` [56]. MariaDB lacks the surgical precision of F492 but has a system variable for the sweeping `SET FOREIGN_KEY_CHECKS = 0;` [57], which in the prototype implementation in this

[7]The most attention was put on MariaDB, but IBM DB2 and MS SQL Server were looked at, too.

[8]In this thesis, the references to the temporal features of the standard are labeled SQL:2011. In this regard, a late draft of SQL:2011 was used. At a further point in time, the official text of SQL:2023 was obtained and cross-checked with the SQL:2011 draft. When the two concur, only SQL:2011 is cited, otherwise both are, with differences highlighted.

Algorithm 3 Addition of $D_{(i)}$ to $D_{(1)} \oplus \cdots \oplus D_{(i-1)}$ (enhanced)

```
Require: Snapshot $D_{(i)}$
Require: $D^* = D_{(1)} \oplus \cdots \oplus D_{(i-1)}$
Ensure: $D^* = D_{(1)} \oplus \cdots \oplus D_{(i)}$
 1: Get $D_{(i-1)}$ from $D^*$
 2: Sort relations in $D_{(i)}$ s.t. $R_1 \prec R_2$ if $R_2 \triangleright R_1$
 3: for all $R_{(i,j)} \in D_{(i)}$ do
 4:     if $K_j$ is defined for schema $\mathcal{R}_j$ then
 5:         for all $r \in R_{(i,j)}$ do
 6:             if $r[K_j] \notin R_{(i-1,j)}[K_j]$ then
 7:                 insert $r$ in $R_j^*$ time $[t_i, t_{max}]$
 8:             else if $r[K_j] \in R_{(i-1,j)}[K_j] \wedge r[\mathcal{R}_j] \notin R_{(i-1,j)}[\mathcal{R}_j]$ then
 9:                 update $r_{(i-1)}$ in $R_j^*$ set $E = t_i$
10:                 insert $r$ in $R_j^*$ time $[t_i, t_{max}]$
11:             end if
12:         end for
13:         for all $r \in R_{(i-1,j)}$ do
14:             if $r[K_j] \notin R_{(i,j)}[K_j]$ then
15:                 update $r_{(i-1)}$ in $R_j^*$ set $E = t_i$
16:             end if
17:         end for
18:     else
19:         for all $r \in R_{(i,j)}$ do
20:             if $r \notin R_{(i-1,j)}$ then
21:                 insert $r$ in $R_j^*$ time $[t_i, t_{max}]$
22:             end if
23:         end for
24:         for all $r \in R_{(i-1,j)}$ do
25:             if $r \notin R_{(i,j)}$ then
26:                 update $r_{(i-1)}$ in $R_j^*$ set $E = t_i$
27:             end if
28:         end for
29:     end if
30: end for
```

thesis is called in the beginning of the main procedure `sync_databases()`. Regardless of the details, there always seems to be a way to get around the sorting step.

Step 3 iterates over all relations in snapshot i, using j as index for the relation to be processed.

Step 4 splits the algorithm into two major parts, to provide a standards-compliant treatment of tables that have an *explicitly defined* primary key (which in the experience of NAE is most tables in most databases). The emphasis is on *explicitly defined*, as opposed to any candidate key that is not formally defined in the relation schema. Primary key is denoted here by K with the index j (reminding that the key is specific to the relation). Similarly, the j in $\mathcal{R}_j$ emphasises the specificity of the schema to the relation that is being processed in the current run of the for-loop.

Respecting formal primary keys allows for uniquely identifying a row in both snapshots and then comparing the values of the current with the previous, and if change is detected, updating the row. Ignoring the primary keys and instead comparing all columns is non-standard and risks breaking referential integrity. This is against the guiding principle nr 3: Avoid hacks (see section 1.5).

Step 5 iterates over all tuples in relation j from snapshot i.

Step 6 tests if the tuple was present in the previous snapshot. The test relies on primary keys, i.e., only the values of the attributes that form the primary key K_j are compared.

Step 7 inserts the new tuple into the temporal relation, setting system versioning period's $S = t_i$ (the time of creation of snapshot i) and $E = t_{max}$ (to indicate the tuple is the most current one available). The result is that the temporal relation R^*_j contains exactly one tuple with this specific primary key $r[K_j]$ and its system versioning period is $[t_i, t_{max}]$.

Step 8 tests for tuples that have changed, i.e., a tuple with the same primary key existed in the previous snapshot, but the tuples are not identical in all attributes (across the full schema $\mathcal{R}_j$).

Step 9 marks the tuple that already exists in the temporal relation as historical by setting its period end timestamp E to the timestamp of the current snapshot i. E.g., assuming the tuple was first inserted for snapshot $i - 1$, its time period is now $[t_{(i-1)}, t_i]$, which in the open-closed[9] period semantics means the tuple became the official recorded state of the data at $t_{(i-1)}$ and ceased being official at t_i (i.e., was official up until right before t_i).

There is a semantic finesse: the r in "$r_{(i-1)}$ **in** R^*_j" is not structurally identical to the loop variable r in "$r \in R_{(i,j)}$" (step 5) because the temporal relation R^*_j contains all the attributes of the non-temporal R_j, plus the system versioning period's start (S) and end (E) timestamps. Therefore, the fully accurate formulation would be this SQL-like statement:

update R^*_j **set** $E = t_i$ **where** $r[K_j] = r_{(i-1)}[K_j]$.

However, this statement is confusing, because it is structurally identical to the SQL

[9]For some explanations, check out Table 10 and the paragraph below it.

statement that applies to the whole table, not just to the one row being processed here in the for-loop. Consequently, the simpler wording for step 9 is left in place and it is meant to be read: "in R_j^*, update the tuple that was current in snapshot $i-1$ and set its $E = t_i$."

Step 10 inserts the tuple from snapshot i into the temporal relation and sets its system time period to $[t_i, t_{max}]$ (indicating that the tuple was recorded at t_i and is still current).

After steps 9 and 10, there are two versions of tuple r in R_j^* (K denotes primary key, v_c and w_c are values of attribute c):

$(K, v_1, v_2, \ldots, v_n, t_{(i-1)}, t_i)$
$(K, w_1, w_2, \ldots, w_n, t_i, t_{max})$

In an implementation on a temporal RDBMS, steps 9 and 10 would be called by one statement, expressed here in abstract pseudocode:

update r **in** R_j^* **set** $A_c = r_i[A_c]$ **where** $r_{i-1}[A_c] \neq r_i[A_c]$ **time** $[t_i, t_{max}]$.

In effect, the RDBMS would be told to just update all the attributes that have changed, and it would then break the statement down to its constituents depending on the temporal storage setup. In the case of all records being in the same table (as is the default in MariaDB, see section "Storing the History Separately" in [41]), the statements in step 9 and 10 are executed. In the case of historical records residing in a separate table (as is the case with Microsoft SQL Server [58]), the steps are: 1) insert the current row into the history table, only changing the period end to $E = i$, 2) update the current row to match the attribute values in snapshot i, plus set $S = i$ and $E = t_{max}$. This process is concisely explained in a Microsoft Learn article: "When data is updated, it is versioned, with the previous version of each updated row is inserted into the history table. When data is deleted, the delete is logical, with the row moved into the history table from the current table – it is not permanently deleted" [59].

Step 13 is the mirror image of step 5: it iterates over all tuples in relation j from snapshot $i-1$ (vs. i in step 5). The goal is to detect tuples that were present in snapshot $i-1$ but are not in snapshot i, which effectively means they have been deleted after snapshot $i-1$ was created.

Step 14 tests if the tuple from the previous snapshot has been deleted.

Step 15 is identical to step 9, but there it was part of a higher level update, while here it performs what on a temporal RDBMS would be "**delete** r **from** R_j^*".

Step 18 begins the treatment of tuples that have no formally declared primary key.

Step 19 iterates over all tuples of the relation in the current snapshot.

Step 20 tests if the tuple was present in the previous snapshot. The comparison of the tuples is done on all attributes (except for the timestamps S and E), i.e., a match means $r_{(i,j)}[\mathcal{R}_j] = r_{(i-1,j)}[\mathcal{R}_j]$.

Step 21 (identically to step 7) inserts the new tuple into the temporal relation.

Step 24 (identically to step 13) starts the loop to search for tuples that have been deleted after snapshot $i - 1$ was created.

Step 25 (similarly to step 14) tests if the tuple from the previous snapshot has been deleted, but dissimilarly to 14, does so based on all attributes.

Step 26 (identically to step 15) marks the tuple that was current as of t_{i-1} to have ceased to be the current state of r at t_i.

Missing step? The steps above lack one to handle the situation where $r[\mathcal{R}_j] = r_{(i-1)}[\mathcal{R}_j]$, i.e., the tuple has not changed. That is because in the SQL:2011-style timestamping, current tuples are marked $E = t_{max}$ and therefore stay current until changed.

4.2 Solution part 1: Merge snapshots – Implementation

The prototype[10] implementation of the abstract algorithm is based on MariaDB, which was chosen for its best match to the guiding principles (see section 1.5). Namely, principle 1 "Do not reinvent the wheel" by having a largely SQL:2011 compliant set of temporal functionalities, and principle 2 "Prefer open source" by being a proper open source project.

MariaDB handles both the system time (aka transaction time) and application time dimensions (see section 3.4). The choice between these two fell on system time for two reasons. First, it is supported by better functionality for the current task. For example, "UPDATE

[10]The terms "Prototype" and "Proof of Concept" are somewhat vague and overlapping, e.g., see SoftKraft, "Prototype vs Proof of Concept — 6 Key Differences to Know," Jan 17, 2023, https://medium.com/@softkraft/prototype-vs-proof-of-concept-6-key-differences-to-know-e47bf32670b1 and N. Ferdous, "Proof of Concept Vs. Prototype: How Do They Differ," Sep 12, 2022, https://differencecamp.com/proof-of-concept-vs-prototype/. For the product of this research, the term "prototype" seems more accurate.

and DELETE on system-versioned tables result in the automatic insertion of a historical system row for every current system row that is updated or deleted" [60] – a behaviour directly exploited by the prototype implementation. Second, it is more semantically accurate, because it is concerned with the time when the data was recorded in the database, and that is precisely what the snapshot creation time communicates. Application time is about when the stored facts were true in the real world, but we have no information about that.

Using the prototype requires the following steps:

1. Find a computer with a Unix/Linux operating system, e.g., a Mac[11].
2. Obtain administrator level access to a MariaDB server.
3. Install DBPTK Developer.
4. Collect the SIARD files into a folder, order chronologically and name systematically, starting from the oldest, using the naming pattern `dbname_s[i].siard`, where `[i]` is an integer counter that starts from 1.
5. Configure the connection parameters, folder for SIARD files and the naming prefix in the beginning of `Main_workflow.sh`.
6. Configure the snapshot timestamps in `Import_snapshots_into_temporal_db.sql`.
7. Run `Main_workflow.sh`.
8. Connect to the MariaDB server with your favourite database management tool (e.g., DBeaver) and start making queries.

The following is a brief description of the major workflow steps, further comments can be found inside the code presented in Appendix 2. ChatGPT v.4 was used in drafting the code [54] (for details see section 2.4).

4.2.1 Create blank databases

A for-loop in the main workflow script creates blank databases into the MariaDB engine. The key is using a fixed naming pattern for both the SIARD files and database names.

4.2.2 Upload SIARD snapshots

Process the SIARD snapshots chronologically, starting from the oldest. For each SIARD snapshot, upload it into the blank database created in the previous step. Technically it is

[11]Windows is also usable but more complicated. The problem part is the main workflow Bash script – Windows cannot run it natively. There are several solutions, the easiest of which seems to be the Windows Subsystem for Linux [61].

two substeps: first, create the structure defined in the SIARD file, and second, fill the tables with the data. The upload is performed by DBPTK command line utility (called DBPTK Developer).

This is all for the SIARD files, further on, all activities take place in the RDBMS.

4.2.3 Clone aggregate database from snapshot 1

Use the live database instance of the oldest snapshot to create a new database with the same structure (but without data) – this will become the aggregate temporal database.

4.2.4 Add system versioning to aggregate database

Enable the temporal features for all tables in the newly created aggregate database. The result is the creation of the columns `row_start` and `row_end` that store the system time period, and the activation of the automated time-stamping of rows. Therefore, this step must happen before merging the data.

4.2.5 Sync databases

This step is a concrete realisation of the abstract algorithm 3. For each snapshot database, starting from the oldest (i.e., the one named `dbname_s1`), merge its data into the aggregate temporal database (named named `dbname_s0`). Each row in the temporal database will get a “system time” timestamp into its system-versioning period start field – use the snapshot creation time to fill that field (i.e., all rows from all tables in the snapshot will get an identical period designation).

Merging is done one table at a time, iterating over all the tables in the database. For each row in a table, there are three possible cases for its presence in the current snapshot as compared to the previous one:

- **Case 1: INSERT**. The row didn’t exist in the last snapshot, i.e., it has been created after the last snapshot was made.
- **Case 2: UPDATE**. The row already existed in the last snapshot, but at least one column differs.
- **Case 3: DELETE**. The row existed in the last snapshot, but not in the current one, i.e., it was deleted after the last snapshot was made.

An example of these scenarios can be seen in Table 9. The data is from the table `Product` that has three fields: `ID` – integer that serves as a surrogate[12] primary key, `Item` – product name, and `Quantity` – quantity of the product in stock. For instance, the first row in S1 `(1, Apple, 100)` means that there were 100 apples in stock at the time. Snapshots were taken after the end of the business day on May 1 and May 2.

Table 9. *Two snapshots of table Product(ID, Item, Quantity)*

S1 – May 1	S2 – May 2	Case
(1, Apple, 100)	(1, Apple, 40)	UPDATE
	(5, Banana, 10)	INSERT
(2, Carrot, 33)		DELETE

The merging of these rows from the two snapshots into a temporal database is processed with SQL `UPDATE`, `INSERT` and `DELETE` statements correspondingly. By nature of SQL, these statements can be crafted to apply for the whole table, without the need to manually code row-by-row iteration.

The outcome of the operations above is a temporal database that contains all the data from the archived snapshots, stored efficiently, without duplicates. Every data row (or more specifically, every version of every row) is timestamped, so that the full history of data evolution can be reconstructed.

The result of the aggregation of the sample data can be seen in Table 10. The `Product` table in the aggregate temporal database has two extra fields: `row_start` and `row_end`, which indicate the time when the data in the row started to be current state and the time when it ceased to. Note that the end time is exclusive, meaning that the data was valid up until that time, but stopped being valid at exactly that time. `row_end = 9999` stands for currently official data, i.e., the end of it being official has not been recorded yet.

Table 10. *Two snapshots of table Product(ID, Item, Quantity) merged*

S1 – May 1	S2 – May 2	Aggregate	Case
(1, Apple, 100)	(1, Apple, 40)	(1, Apple, 100, May 1, May 2) (1, Apple, 40, May 2, 9999)[13]	UPDATE

Continues...

[12]Surrogate as opposed to natural, in the sense that a natural key consists of data that have a meaning in real life, whereas a surrogate key is generated by the system with the sole purpose of identifying a record.

[13]"9999" denotes the largest possible time value the RDBMS can handle (notation borrowed from [38]) and conveys the message that the data in this row are considered currently valid. Relational algebra would call for NULL, the main definition of which is "value at present unknown" [62, p. 403], but to make the queries

Table 10 – *Continues...*

S1 – May 1	S2 – May 2	Aggregate	Case
	(5, Banana, 10)	(5, Banana, 10, May 2, 9999)	INSERT
(2, Carrot, 33)		(2, Carrot, 33, May 1, May 2)	DELETE

The default interpretation of timestamps in temporal databases assumes a continuous timeline, i.e., the data are valid for the whole interval `[row_start, row_end)` (square bracket denoting that start is inclusive, round bracket that end is exclusive). In the case of infrequent snapshotting we can be almost certain that this is not a correct documentation of the state of affairs in the real life, because the data could have gone through multiple changes and we have only captured their state at a few moments. For instance, the data tells us that there were 100 apples in stock in the evening of May 1 and 40 in the evening of May 2. We do not know the level of stock at 10 AM or at noon of May 2. We also do not know if the end state of 40 is a result of 60 apples being sold in one transaction at noon of May 2, or was there a 10 AM incoming batch of 100 apples from the wholesaler and then multiple sales totalling to 160 apples over the day.

Therefore the correct interpretation in the case of aggregated snapshots is that the data are *known* to be valid *at* the snapshot times. For all other times, the values indicate the *best archived state* of the data. We know them to be in a generally plausible range (after all, at least at some point in time the data had been in this state), and in many cases they end up being exactly correct, but we should always acknowledge the inherent indeterminacy.

4.3 Solution part 2: Query data

Querying is done using the standard SQL:2011 or later temporal query constructs. The contribution of the author of this thesis is limited to gathering the examples and tweaking the syntax in minor ways.

The options for querying are discussed on the example of `vehreg` – the database of a hypothetical state vehicle registry. For a deeper understanding of the example queries, the reader is encouraged to inspect the commented raw data files. The link to these files is provided in section 4.5 and the rationale for using artificial data in the first place in section 2.3. ERD is also there as Figure 3. The examples use unqualified table names, i.e., the name of the database is omitted. For this to work, the aggregate temporal database must be set as the default database for the session. The current default can be displayed by:

on open-ended time periods work identically to closed time periods, the RDBMS use the largest possible value instead. The SQL standard also follows this approach [17, Sec. 4.15.2.2], [39, Sec. 4.17.2.3].

```
SELECT DATABASE();
```

A new default can be set by (note that in the prototype, the aggregate temporal database always has "s0" at the end of its name):

```
USE vehreg_s0;
```

The basic syntax of temporal queries on a single table is described in the MariaDB knowledge base article “System-Versioned Tables” [41]. The following are the key items from the section “Querying Historical Data” of that article, with examples replaced by the author of the thesis.

4.3.1 Single table, at a moment

“To query the historical data one uses the clause `FOR SYSTEM_TIME` directly after the table name (before the table alias, if any). /. . . / `AS OF` is used to see the table as it was at a specific point in time in the past” [41].

```
SELECT *, row_start, row_end FROM vehicle
FOR SYSTEM_TIME AS OF '2016-10-09';
```

Table 11. *Table `vehicle` at a moment*

id	owner_id	number	make	model	year	row_start	row_end
2	2	222BBB	Zil	130	1970	2000-01-01	2038-01-19
3	3	333CCC	MB	E220	2002	2005-01-01	2038-01-19
4	4	444DDD	Audi	A4	2007	2010-01-01	2020-01-01
5	5	555EEE	Opel	Astra	2014	2015-01-01	2038-01-19

To conserve space and the reader’s attention, time is left out of the `TIMESTAMP` values in the examples. In reality, MariaDB times are at microsecond precision, thus queries like this are also possible:

```
SELECT *, row_start, row_end FROM vehicle
FOR SYSTEM_TIME AS OF '2016-10-09 08:07:06.000001';
```

If the user interface of the database management software hides some of the precision from the query results (e.g., DBeaver defaults to milliseconds), then one way to see the full values is to use the `DATE_FORMAT()` function:

```sql
SELECT DATE_FORMAT(row_end,'%Y-%m-%d %H:%i:%s.%f') FROM vehicle
FOR SYSTEM_TIME AS OF '2016-10-09';
```

4.3.2 Single table, over a period

"`BETWEEN start AND end` will show all rows that were visible at any point between two specified points in time. It works inclusively, a row visible exactly at start or exactly at end will be shown too" [41].

```sql
SELECT *, row_start, row_end FROM vehicle
FOR SYSTEM_TIME BETWEEN '2017-05-05' AND '2024-05-05';
```

Table 12. *Table `vehicle` over a period*

id	owner_id	number	make	model	year	row_start	row_end
2	2	222BBB	Zil	130	1970	2000-01-01	2038-01-19
3	3	333CCC	MB	E220	2002	2005-01-01	2038-01-19
4	4	444DDD	Audi	A4	2007	2010-01-01	2020-01-01
4	6	444DDD	Audi	A4	2007	2020-01-01	2038-01-19
5	5	555EEE	Opel	Astra	2014	2015-01-01	2038-01-19
6	4	666FFF	Nissan	Leaf	2019	2020-01-01	2038-01-19

Note that the period queries allow some temporal calculations, e.g., on 5 May 2024, the exact[14] same result could have been gotten using the "give me the last 7 years" syntax:

```sql
SELECT *, row_start, row_end FROM vehicle
FOR SYSTEM_TIME BETWEEN (NOW() - INTERVAL 7 YEAR) AND NOW();
```

There is an alternative syntax to achieve the same result: "`FROM start TO end` will also show all rows that were visible at any point between two specified points in time, including start, but excluding end" [41].

[14]As stated in the previous section, we are ignoring the time component of the `TIMESTAMP` fields, otherwise the results could differ, as `NOW()` gives a timestamp with one second precision.

```
SELECT *, row_start, row_end FROM vehicle
FOR SYSTEM_TIME FROM '2017-05-05' TO '2024-05-06';
```

The `TO ...` value of `FROM ... TO ...` is not included in the period, so to get identical results to `BETWEEN ... AND ...` we have to increase it by one unit of time. In these examples we operate with days, thus +1 day. In reality, the `TIMESTAMP` type in MariaDB has microsecond accuracy, so we would need to pick a moment 1 microsecond later, e.g., `'2024-05-05 00:00:00.000001'`. This of course is artificial – the alternative clauses `BETWEEN ... AND ...` and `FROM ... TO ...` are provided precisely for the purpose of avoiding such play with microseconds.

The results above might seem counter intuitive because they contain rows that had `row_start` before the start time specified in the query and `row_end` later than the specified end time. This is because both the `BETWEEN ... AND ...` and `FROM ... TO ...` syntaxes match rows using a concrete definition of *overlap*, as already quoted above from the MariaDB manuals: "rows that were visible at any point between two specified points in time" [41]. Safe navigation of temporal queries requires the comprehension of period relationships, famously systematised by James F. Allen in 1983 [14] and their implementation in modern RDBMS. A concise listing of the SQL standard's interpretation can be found in its section "Operations involving periods," which is numbered 4.14.2 in SQL:2011 [17] and 4.16.2 in SQL:2023 [39].

If deviations from the standard *overlap* are desired, additional criteria to `SELECT` can be used. For instance, to get only the rows that became current during the period defined in the query, the following can be used (adopted from [63]):

```
SELECT *, row_start, row_end FROM vehicle
FOR SYSTEM_TIME FROM '2017-05-05' TO '2024-05-06'
WHERE row_start >= '2017-05-05';
```

Table 13. *Table `vehicle` over a period, filtered by `row_start`*

id	owner_id	number	make	model	year	row_start	row_end
4	6	444DDD	Audi	A4	2007	2020-01-01	2038-01-19
6	4	666FFF	Nissan	Leaf	2019	2020-01-01	2038-01-19

4.3.3 Single table, whole history

The above query constructs were from SQL:2011. MariaDB complements them with the non-standard extension ALL that will show all rows, historical and current [41].

```
SELECT *, row_start, row_end FROM vehicle
FOR SYSTEM_TIME ALL;
```

This is identical to an SQL:2011-compliant explicit period query that uses the min and max values of the TIMESTAMP data type:

```
SELECT *, row_start, row_end FROM vehicle
FOR SYSTEM_TIME
BETWEEN '1970-01-01 00:00:00' AND '2038-01-19 03:14:07';
```

Table 14. *Table vehicle whole history*

id	owner_id	number	make	model	year	row_start	row_end
1	1	111AAA	Ford	Ka	1999	2000-01-01	2005-01-01
1	2	111AAA	Ford	Ka	1999	2005-01-01	2010-01-01
1	3	111AAA	Ford	Ka	1999	2010-01-01	2015-01-01
2	2	222BBB	Zil	130	1970	2000-01-01	2038-01-19
3	3	333CCC	MB	E220	2002	2005-01-01	2038-01-19
4	4	444DDD	Audi	A4	2007	2010-01-01	2020-01-01
4	6	444DDD	Audi	A4	2007	2020-01-01	2038-01-19
5	5	555EEE	Opel	Astra	2014	2015-01-01	2038-01-19
6	4	666FFF	Nissan	Leaf	2019	2020-01-01	2038-01-19

This looks similar to the result set in Table 12, but the difference is the group of three rows with id = 1 that were left out previously due to period start having been set at '2017-05-05'.

4.3.4 Single table, current data

"If the FOR SYSTEM_TIME clause is not used, the table will show the current data. This is usually the same as if one had specified FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP, unless one has adjusted the row_start value" [41].

```
SELECT *, row_start, row_end FROM vehicle;
```

Table 15. *Table* `vehicle` *current data*

id	owner_id	number	make	model	year	row_start	row_end
2	2	222BBB	Zil	130	1970	2000-01-01	2038-01-19
3	3	333CCC	MB	E220	2002	2005-01-01	2038-01-19
4	6	444DDD	Audi	A4	2007	2020-01-01	2038-01-19
5	5	555EEE	Opel	Astra	2014	2015-01-01	2038-01-19
6	4	666FFF	Nissan	Leaf	2019	2020-01-01	2038-01-19

4.3.5 Two tables, temporally joined

All the system versioning period specifications from `SELECT` can also be used as part of `JOIN` clauses [64]:

```
FOR SYSTEM_TIME AS OF point_in_time
FOR SYSTEM_TIME BETWEEN point_in_time AND point_in_time
FOR SYSTEM_TIME FROM point_in_time TO point_in_time
FOR SYSTEM_TIME ALL
```

The following statement reports the owner of the vehicle at a moment in time. It requires a `JOIN`, because the data are stored in two tables: vehicle's details in `vehicle` and the owner's names in `owner`.

```
SELECT v.id, v.number, v.make, v.model, v.row_start, v.row_end,
       o.first_name, o.last_name
FROM vehicle FOR SYSTEM_TIME AS OF '2007-01-01' AS v
LEFT JOIN owner FOR SYSTEM_TIME AS OF '2007-01-01' AS o
ON v.owner_id = o.id
WHERE v.number = '111AAA';
```

Table 16. *Tables* `vehicle` *and* `owner` *joined over a moment*

id	number	make	model	row_start	row_end	first_name	last_name
1	111AAA	Ford	Ka	2005-01-01	2010-01-01	Ben	Bennett

Similarly to single table queries, joined tables can also be queried over a period. Here is an example for getting the ownership history over a ten year period.

```
SELECT v.id, v.number, v.make, v.model, v.row_start, v.row_end,
       o.first_name, o.last_name
FROM vehicle FOR SYSTEM_TIME FROM '2002-02-02' TO '2012-02-02' AS v
LEFT JOIN owner FOR SYSTEM_TIME FROM '2002-02-02' TO '2012-02-02' AS o
ON v.owner_id = o.id
WHERE v.number = '111AAA';
```

Table 17. *Tables* `vehicle` *and* `owner` *joined over a period*

id	number	make	model	row_start	row_end	first_name	last_name
1	111AAA	Ford	Ka	2000-01-01	2005-01-01	Alice	Adams
1	111AAA	Ford	Ka	2005-01-01	2010-01-01	Ben	Bennett
1	111AAA	Ford	Ka	2010-01-01	2015-01-01	Charlie	Collins

4.3.6 System variable "system_versioning_asof"

There is a useful system variable for studying a specific moment in history (i.e., for making many different queries for that time). "If set to a specific timestamp value, an implicit `FOR SYSTEM_TIME AS OF` clause will be applied to all queries" [41].

```
SET @@system_versioning_asof = '2016-10-09';
```

After executing that statement, the no timestamp (i.e., current time) query:

```
SELECT *, row_start, row_end FROM vehicle;
```

and the explicit timestamp query:

```
SELECT *, row_start, row_end FROM vehicle
FOR SYSTEM_TIME AS OF '2016-10-09';
```

will produce the same result (for the exact output see Table 11). The modified default can still be overriden by supplementing the query with a different timestamp (for the exact output of the following see Table 15):

```
SELECT *, row_start, row_end FROM vehicle
FOR SYSTEM_TIME AS OF '2024-01-01';
```

or a period (for the exact output of the following see Table 12):

```
SELECT *, row_start, row_end FROM vehicle
FOR SYSTEM_TIME BETWEEN '2017-05-05' AND '2024-05-05';
```

Normal behaviour can be restored using:

```
SET @@system_versioning_asof = DEFAULT;
```

4.4 Tools and technologies

The following tools and technologies are involved in the functioning of the prototype.

RDBMS is the server software for operating relational databases. Often used interchangeably with the term database engine, although technically the engine is only a component of the RDBMS. The word is an initialism for Relational Data Base Management System.

https://www.techtarget.com/searchdatamanagement/definition/RDBMS-relational-database-management-system

MariaDB Server (or simply MariaDB) is an open source RDBMS that was originally cloned from the MySQL code base (and is still claimed to be functionally compatible), but has over the years received unique features, too. Most notably, MariaDB has support for SQL:2011 temporal features, which MySQL lacks.

https://mariadb.com/kb/en/documentation/

SIARD is an XML-based file format for archiving relational databases. The word is an acronym for Software Independent Archiving of Relational Databases.

https://siard.dilcis.eu

DBPTK is an open source program for creating and viewing SIARD files. The word is an initialism for Database Preservation Toolkit.

https://database-preservation.com

Bash is an open source Unix command line interpreter and programming language. The word is an acronym for Bourne-Again SHell, hinting at its predecessor, the Bourne shell.

https://www.gnu.org/software/bash/

SQL is a language for interacting with RDBMS. The word is an initialism for Structured Query Language. It includes many statements, out of which three groups are the most relevant to the current thesis:

- Statements for defining the database structure, e.g., `CREATE` and `ALTER`, also called Data Definition Language (DDL).
- Statements for reading and writing the data, e.g., `SELECT`, `INSERT`, `UPDATE` and `DELETE`, also called Data Manipulation Language (DML).
- Programming statements that allow for the automation of DDL and DML operations, including in the form of stored procedures that can be called upon need.

SQL is standardised as ISO/IEC 9075, the latest version being from 2023. For a brief history and an overview of the constituent parts of the standard see [65].

https://www.w3schools.com/sql/

DBeaver is an open source program for administering databases, able to work with tens of RDBMS.

https://dbeaver.io

4.5 File supplement

The program code and vehicle registry data are made available at:

URL:	http://mantagir.ee/thesis/
File:	Koit_Saarevet_MSc_File_Supplement.zip
Size:	59830
SHA256:	9438092ff55c2774524a83f30fef18630c923a49755374b584e748d07ddc3add

The file supplement contains two sets of files: the ones in the folder INSERT and the others in SIARD. INSERT has the data files in the form of SQL `INSERT` statements, while SIARD has data as SIARD files. Each folder contains program code for merging the respective kind of source data (run `Main_workflow.sh`). Additionally, the INSERT folder contains the program `siard_create.sh` to transform snapshots from the live database into SIARD files.

Table 18. *Contents of the file supplement*

File	**Description**
INSERT/Create_procedure_sync_databases.sql	SQL procedure used for merging
INSERT/Create_procedure_sync_single_table_no_pk.sql	SQL procedure used for merging
INSERT/Create_procedure_sync_single_table.sql	SQL procedure used for merging
INSERT/Create_tables_into_snapshot_db.sql	SQL procedure used for merging
INSERT/Import_snapshots_into_temporal_db.sql	Configuration of snapshot timestamps
INSERT/Main_workflow.sh	Main program (bash script)
INSERT/siard_create.sh	Create SIARD files from the snapshots in RDBMS
INSERT/Datafiles/vehreg_s1.sql	Snapshot data
INSERT/Datafiles/vehreg_s2.sql	Snapshot data
INSERT/Datafiles/vehreg_s3.sql	Snapshot data
INSERT/Datafiles/vehreg_s4.sql	Snapshot data
INSERT/Datafiles/vehreg_s5.sql	Snapshot data
SIARD/Create_procedure_sync_databases.sql	SQL procedure used for merging
SIARD/Create_procedure_sync_single_table_no_pk.sql	SQL procedure used for merging
SIARD/Create_procedure_sync_single_table.sql	SQL procedure used for merging
SIARD/Import_snapshots_into_temporal_db.sql	Configuration of snapshot timestamps
SIARD/Main_workflow.sh	Main program (bash script)
SIARD/Datafiles/vehreg_s1.siard	Snapshot data
SIARD/Datafiles/vehreg_s2.siard	Snapshot data
SIARD/Datafiles/vehreg_s3.siard	Snapshot data

Continues…

Table 18 – *Continues…*

File	**Description**
SIARD/Datafiles/vehreg_s4.siard	Snapshot data
SIARD/Datafiles/vehreg_s5.siard	Snapshot data
Readme.txt	Comments

5. Discussion

This chapter is divided into three, starting from the evaluation of the work done. The middle part, titled Contributions, highlights several ways of this thesis making the world a better place. The third third starts from going into more detail on the highest-potential impact, proceeding then to seven other ideas on how the journey might continue beyond the commencement ceremony.

5.1 Evaluation

The objective of the research was to develop a method for accessing a series of archived database snapshots. The success criteria were:

1. **It is possible to prepare several snapshots for simultaneous access** – this was achieved, both theoretically in the form of algorithm 3 and in practice (see section 4.2).
2. **It is possible to make cross-snapshot temporal queries** – demonstrated in section 4.3.
3. **The solution works with real SIARD files (i.e., not just a theoretical model)** – demonstrated through the sample queries in section 4.3, also the reader can use the files and data in File supplement to repeat the experiments.

In conclusion, the research objective was achieved.

Next, referring to the plural "Methods" in the title: the research identified three methods of access:

1. **Manual aggregation** – load one snapshot at a time into a viewer to perform the queries, export results into an analytic tool, aggregate the results there.
2. **Bronselaer algorithm 1** – merge the snapshots into an aggregate database that does not have system versioning enabled, thus construct temporal queries in a bespoke manner.
3. **The method from this research (algorithm 3)** – merge the snapshots into an aggregate *temporal* database and query using the temporal extensions in SQL:2011.

A comparative analysis of these was not in the scope of this research. Intuitively, the manual method is clearly inferior to the others due to being more time consuming and error prone. The affairs of methods 2 and 3 are more murky and worth exploring (see subsection 5.3.8).

5.2 Contributions

First, the prototype produced is ready to use for series of identically structured SIARD snapshots (or with some tweaking, CSV snapshots). It can also be developed to handle series of snapshots that have slightly differing structure, but this requires ad hoc decision-making on how to handle the structural differences and then manual work in coding the decisions in the Bash and SQL scripts.

This provides immediate value to the archival community, which was partly described in section 1.6, but is actually wider than that – any medium or large sized private organisation has likely the need, too. For example, in Denmark, there is a whole industry of private companies that provide the service of preparing SIARD snapshots for transfer to the National Archives. The official list contains 39 names of "suppliers who have prepared and had one or more archiving versions of IT systems approved" [66]. It is highly likely that they attempt to sell the same services to the business sector, too.

Second, multi-snapshot access is not only about very long time periods with a large number of snapshots (as might seem at the first glance). The solution developed through this research is already valuable for just two subsequent snapshots. The reason: most archiving today is done in all-rows snapshots, i.e., there is no filtering out the rows that were already archived in the previous snapshot. The archives would very much like to filter, but there is simply no means of detecting such rows and even if there was, it would be too risky to leave some rows out because any mistake can result in broken referential integrity.

Such incremental archiving would be easy to do from a system-versioned database, or from a system where the selective export functionality is built in. Neither is the case in practice: as of the completion of this thesis, NAE has not ingested (or even seen) any system-versioned databases, nor has it seen an information system with a proper selective export. Consequently, subsequent snapshots contain a lot of duplicate data, while the duplicates are hard to detect. Typically, data are changed frequently in the early stages of the life cycle and become rather static after that. Nevertheless, old data still get updated occasionally. The solution produced in this thesis automatically resolves the problem of duplicates, allowing the user to perform queries on clean data.

Third, there is a hypothesis that the solution from this research can (with moderate effort) be developed further into a solution for incremental archiving. Not in the sense of the previous paragraph, where the value is in the simplified access to ordinary snapshots, but for producing archival snapshots that are already incremental (i.e., without duplications of records that have been present in previous snapshots), see subsection 5.3.1 for details. If this turns out to be feasible, the impact of the thesis will grow by an order of magnitude, from providing comfortable access in multi-snapshot scenarios (a niche business) to saving major storage and time in almost every database ingest (a mass market service).

Finally, in addition to the significant value delivered to the archival community, this thesis also made a minor contribution to the database science by developing a temporal merge algorithm that is executable on a temporal RDBMS.

Paraphrasing Neil Armstrong:

That's one small step for database science, one giant leap for the archival community.

5.3 Future work

This section enumerates several directions for future research and development.

5.3.1 Incremental archiving

As started in the previous section, the current vague idea is to include the solution from this thesis into the archiving process, e.g., first create a full snapshot, then merge it into the temporal database and have that temporal database be the preservation copy. Or add one more step and extract a delta snapshot from the temporal database, and archive only that as the snapshot from the current year. Both approaches result in a no-duplicates archival copy.

This is a big deal for the general principle of avoiding duplicates, but also for the storage costs. Modern databases tend to be large, usually over a terabyte and some significantly larger (the largest now at NAE is the Buildings Registry, which stands at 20 TB). Exacerbating the problem: often times duplicate data are archived on purpose, e.g., the full snapshot with all the data and another one where personal data has been censored. Then the whole set is kept in redundant storage, usually 3-4 copies, so the net 20 TB can end up being gross 120-160 TB.

Another side avenue is the handling of external files – in most cases the files comprise the bulk of the byte volume (in the Buildings Registry, 12 out of 20 TB are the files). Most of these files stay in the live system for a long time and thus end up in multiple snapshots, ballooning the storage needs. It seems relatively easy to adapt the solution from this thesis to prevent duplicate files – a hash-based mechanism for detecting content changes plus the algorithm to manage it at the row level.

A quick Scopus search `Article title, Abstract, Keywords = incremental AND archiving` gave only 70 results, of which none were directly relevant and the first one potentially indirectly useful was from 2011. Thus, there seems to be a research gap.

5.3.2 Schema versioning

The current solution does not handle evolving schemas. Unlike incremental archiving, where 70 results from 40 years contained effectively nothing, database schema versioning has been systematically explored since the 1990s, as exemplified by the paper from De Castro, Grandi and Scalas, where they discuss schema versioning for temporal databases in 1997 [67]. A recent study of interest is from another Italian team who studied ways to keep queries and views functional throughout the evolution of the underlying schema [68].

A related area is XML schema versioning – technically different, but conceptually the same, so even if the research on relational database schema versioning proves sufficient to lay the ground, it is still worth to review the XML field to avoid suffering from the silo effect. For instance, a 2019 paper by Brahmia and colleagues that explores multi-temporal and multi-schema-version XML databases [69]. Snodgrass and colleagues have also addressed schema versioning time-varying XML documents in [27] as has Brahmia's team, who also provided a list of references to other relevant works in their "Related Work" section [25, Sec. V].

In sum: there is plenty of material to explore about schema versioning.

5.3.3 Temporal queries

The product would be a library/taxonomy of temporal queries – the SQL syntax and really the whole logic of temporal queries is unfamiliar for most users. Not all RDBMS have temporal features (e.g., PostgreSQL has no native support) and the implementations vary. Additionally, proper bitemporal thinking, or even the full implementation of one time dimension is rare among database designers. Thus, it would be highly valuable to provide the user with a toolkit of frequently needed queries. The review paper by Böhlen et al. might be a good place to start [47, Sec. 6.4].

5.3.4 System-versioned snapshots

How to handle snapshots with tables that already had system versioning? One option is to switch off system versioning (`ALTER TABLE t DROP SYSTEM VERSIONING`) and treat the `row_start` and `row_end` columns as ordinary timestamp columns. Another option is to build a complete history based on the system-versioning periods in different snapshots.

5.3.5 More universal algorithm

A weak point in the abstract algorithm (see Algorithm 3) is the insertion directly into the system-versioning columns. SQL:2011 does not define a method for it, so the implementation has to rely on the idiosyncrasies of RDBMS. In the case of MariaDB, there was an easy one at reach: a system variable to fix the current-time timestamp to a bespoke value, but other RDBMS do not have that feature. Microsoft SQL Server seems to have solution through the non-standard feature `ALTER TABLE t SET SYSTEM_VERSIONING = OFF;`[1].

A potential standard-based solution is to first create the temporal table with the `row_start` and `row_end` columns, but not add system versioning, then load all the data with the required timestamps, then run `ALTER TABLE t ADD SYSTEM VERSIONING;` [17, Sec. 11.29]. A few hours of tinkering with MariaDB showed that this path can be traversed, but it does not lead to the expected destination: upon adding system versioning, all the values in `row_start` and `row_end` get overwritten, nullifying all gains. Additionally, some clauses of the standard might prevent adding system versioning on existing columns [17, Sec. 11.27], in which case application time might be the way to go, as it is more flexible, although semantically less accurate (see section 4.2) for timing the snapshots. In any case, further research is needed.

5.3.6 Prototype improvements

The prototype has many weaknesses that need to be remedied before use in production environments, e.g.:

[1]Aspects of this approach are discussed in these articles: https://sqlspreads.com/blog/temporal-tables-in-sql-server/ and https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-table-system-consistency-checks?view=sql-server-ver16, both accessed on May 1, 2024.

- Add proper error handling.
- Optimise for performance.
- Dynamic SQL is a potential SQL smell – no thought has been given on security, incl. protection against SQL injection.
- The current Dynamic SQL is hard to read: sometimes the strings are compiled using `SELECT`, other times with string `CONCAT`.
- Analyse the potential weakness: `INSERT` into PK fields might be non-universal, i.e., not possible for some RDBMS and some data types, like for autoincrement.
- Most variables are passed between procedures as user-defined variables, not parameters.

5.3.7 Visualisation

DBPTK has visualisation capabilities for the structure of single snapshots. They are useful, but adhering to the 80:20 mindset[2], they do not cater to the more sophisticated use scenarios. Nevertheless, they are way better than the visualisations for the time dimension, which simply do not exist at all, in any SIARD access tool, for the simple reason that none of the tools provide multi-snapshot access. Fortunately, there are many great examples on how to present time in a search engine, e.g., the Estonian movie portal Arkaader[3], which itself was inspired by other user interfaces, and there is also a significant body of scientific research on the topic.

5.3.8 Comparison of methods

The overall, all-things-considered goodness of methods 2 (Bronselaer) and 3 (this thesis) is not straight forward to assess and is worth exploring systematically. For the end user, the queries in 3 are simpler, but after considering other vital aspects, the final conclusion might differ. Bronselaer's algorithm is only one third the length, so for large datasets there might be a significant performance benefit. The added end user complexity might be alleviated by a special front end application, or a set of stored procedures that hide some of the complexity.

[2]https://www.investopedia.com/terms/1/80-20-rule.asp
[3]arkaader.ee > ENG > Timeline, https://arkaader.ee/landing/br/rHczO7kKnl/0KAFk1XQlz.

6. Summary

The objective of the research was to develop a method for accessing a series of archived database snapshots. The objective was achieved in developing a method and its prototype implementation.

According to the method, a number of identically structured snapshots in the XML-based SIARD format are ordered chronologically and loaded into a temporal database starting from the oldest. The system versioning period start timestamp for every row in the snapshot is set to the creation time of the snapshot. This will result in an evolutionary timeline for the data, e.g., if a row was present in the oldest snapshot, and one of its columns got a new value in each subsequent snapshot, the temporal table will have the whole history with row start and row end timestamps. The history can be queried using the temporal extensions present in the SQL:2011 standard and in MariaDB, the RDBMS chosen for the prototype. Thanks to the temporal features of the RDBMS, temporal referential integrity is preserved automatically, i.e. foreign key values point to the version of the record that was valid at the time.

Three research questions had been formulated, one main and two supportive.

MRQ: Is it possible to access a series of archived database snapshots simultaneously?

SRQ1: Are there existing tools for multi-snapshot access?

SRQ2: Are there theoretical methods for multi-snapshot access?

Literature review and personal communications with the experts confirmed the answer to SRQ1 being negative – no complete tools exist. However, a number of tools were identified that can serve as building blocks for creating a solution (MariaDB, DBPTK, etc., see section 4.4).

The answer to SRQ2 is a qualified yes. Qualified, because the algorithm proposed by Bronselaer et al. (see algorithm 1) is not usable on a temporal database engine, therefore, the temporal query extensions of SQL cannot be used, too. Consequently, the user must construct query statements that are more complicated than would be the ones on a temporal engine. To address this shortcoming, the algorithm was developed further to be executable on a temporal RDBMS, and its correctness was proven via a prototype implementation on MariaDB.

In sum, the answer to the main question is yes, it is possible to access a series of archived database snapshots simultaneously.

References

[1] M. H. Jacinto, G. R. Librelotto, J. C. Ramalho, and P. R. Henriques, “Bidirectional conversion between XML documents and relational data bases,” in *Proc. Int. Conf. on Computer Supported Cooperative Work in Design*, vol. 7, 2002, pp. 437–443.

[2] S. Heuscher, S. Jaermann, P. Keller-Marxer, and F. Moehle, *Providing authentic long-term archival access to complex relational data*, 2004. arXiv: cs/0408054.

[3] DILCIS Board, “Siard,” *DILCIS Board*, Accessed: May 8, 2024. [Online]. Available: https://siard.dilcis.eu.

[4] DILCIS Board, “eCH-0165 SIARD-formatspezifikation,” *eCH E-Government Standards*, Accessed: May 8, 2024. [Online]. Available: https://www.ech.ch/de/ech/ech-0165/1.0.

[5] *SIARD Format Specification*, SIARD-2.2. 2021.

[6] Rigsarkivet, “Extraction tool – freely available,” Copenhagen, Denmark, Government record, 2022, Accessed: May 7, 2024. [Online]. Available: https://en.rigsarkivet.dk/wp-content/uploads/2022/09/DBPTK_introduction.pdf.

[7] K. Aas, “Digitaalsete andmebaaside ettevalmistamine pikaajaliseks säilitamiseks,” M.S. thesis, Univ. Tartu, Tartu, Estonia, 2004.

[8] P. Tømmerholt, “Our story: Danish National Archive,” *E-ARK Project*, Accessed: May 7, 2024. [Online]. Available: https://www.eark-project.com/stories/28-user-stories/96-dna-story.html.

[9] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in information systems research,” *MIS Q.*, vol. 28, no. 1, pp. 75–105, 2004. DOI: 10.2307/25148625.

[10] A. R. Hevner and S. T. March, “The information systems research cycle,” *Comp.*, vol. 36, no. 11, pp. 111–113, 2003. DOI: 10.1109/MC.2003.1244541.

[11] Oracle Corporation, “World sample database,” 2024. [Online]. Available: https://dev.mysql.com/doc/world-setup/en/.

[12] Oracle Corporation, “Sakila sample database,” 2024. [Online]. Available: https://dev.mysql.com/doc/sakila/en/.

[13] J. Tepandi, *Kuidas kirjutada ja kaitsta edukalt lõputööd*, Accessed: Apr. 17, 2024, Mar. 27, 2019. [Online]. Available: https://tepandi.ee/juhendatavatele.pdf.

[14] J. F. Allen, “Maintaining knowledge about temporal intervals,” *Commun. ACM"*, vol. 26, no. 11, pp. 832–843, 1983. DOI: 10.1145/182.358434.

[15] E. Sciore, "Using annotations to support multiple kinds of versioning in an object-oriented database system," *ACM Trans. Database Syst. (TODS)*, vol. 16, no. 3, pp. 417–438, 1991. DOI: 10.1145/111197.111205.

[16] R. T. Snodgrass, I. Ahn, G. Ariav, *et al.*, "TSQL2 language specification," *ACM SIGMOD Rec.*, vol. 23, no. 1, pp. 65–86, 1994. DOI: 10.1145/181550.181562.

[17] *Information technology – Database languages SQL – Part 2: Foundation (SQL/-Foundation)*, ISO/IEC 9075-2. 2011.

[18] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. Parameswaran, "ORPHEUS DB: bolt-on versioning for relational databases (extended version)," *VLDB J.*, vol. 29, no. 1, pp. 509–538, 2020. DOI: 10.1007/s00778-019-00594-5.

[19] C. Ramanathan and A. Kulkarni, "TSAPI: Enhancing existing OLTP databases with temporal query capabilities for analytics," in *ACM Int. Conf. Proc. Ser.*, 2023, pp. 268–271. DOI: 10.1145/3570991.3571069.

[20] A. Bronselaer, C. Billiet, R. De Mol, J. Nielandt, and G. De Tré, "Compact representations of temporal databases," *VLDB J.*, vol. 28, no. 4, pp. 473–496, 2018. DOI: 10.1007/s00778-018-0535-4.

[21] W. H. Inmon, *Building the Data Warehouse, 3rd Ed.* New York, NY, USA: Wiley, 2002.

[22] R. Kimball and M. Ross, *The Data Warehouse Toolkit, 3rd Ed.* Indianapolis, IN, USA: Wiley, 2013.

[23] F. Currim, S. Currim, C. Dyreson, and R. T. Snodgrass, "A tale of two schemas: Creating a temporal XML schema from a snapshot schema with τXSchema," *Lect. Notes Comput. Sci.*, vol. 2992, pp. 348–365, 2004. DOI: 10.1007/978-3-540-24741-8_21.

[24] C. Dyreson, R. T. Snodgrass, F. Currim, and S. Currim, "Schema-mediated exchange of temporal XML data," *Lect. Notes Comput. Sci.*, vol. 4215 LNCS, pp. 212–227, 2006. DOI: 10.1007/11901181_17.

[25] Z. Brahmia, R. Bouaziz, F. Grandi, and B. Oliboni, "Schema versioning in τxSchema-based multitemporal XML repositories," in *Proc. Int. Conf. Res. Challenges Inf. Sci.*, 2011. DOI: 10.1109/RCIS.2011.6006845.

[26] C. Dyreson, R. T. Snodgrass, F. Currim, S. Currim, and S. Joshi, "Validating quicksand: Schema versioning in τXSchema," in *ICDEW 2006 - Proc. 22nd Int. Conf. Data Eng. Workshops*, 2006. DOI: 10.1109/ICDEW.2006.161.

[27] R. T. Snodgrass, C. Dyreson, F. Currim, S. Currim, and S. Joshi, "Validating quicksand: Temporal schema versioning in τXSchema," *Data Knowl. Eng.*, vol. 65, no. 2, pp. 223–242, 2008. DOI: 10.1016/j.datak.2007.09.003.

[28] Z. Brahmia, F. Grandi, B. Oliboni, and R. Bouaziz, "Versioning of conventional schema in the τxSchema framework," 2012, pp. 510–518. DOI: 10.1109/SITIS.2012.153.

[29] Z. Brahmia, F. Grandi, and R. Bouaziz, "TauXUF: A temporal extension of the XQuery Update Facility language for the tauXSchema framework," in *Proc. Int. Workshop Temp. Represent. Reason.*, Cited by: 7, vol. 2016-December, 2016, pp. 140–148. DOI: 10.1109/TIME.2016.22.

[30] S. Brahmia, Z. Brahmia, F. Grandi, and R. Bouaziz, "τJSchema: A framework for managing temporal JSON-based NoSQL databases," *Lect. Notes Comput. Sci.*, vol. 9828 LNCS, pp. 167–181, 2016. DOI: 10.1007/978-3-319-44406-2_13.

[31] Z. Brahmia, F. Grandi, S. Brahmia, and R. Bouaziz, "τJUpdate: An update language for time-varying JSON data," *J. Comp. Lang.*, vol. 79, 2024, Cited by: 0. DOI: 10.1016/j.cola.2024.101258.

[32] F. Rizzolo and A. A. Vaisman, "Temporal XML: Modeling, indexing, and query processing," *VLDB J.*, vol. 17, no. 5, pp. 1179–1212, 2008. DOI: 10.1007/s00778-007-0058-x.

[33] F. Zhang, Z. Li, D. Peng, and J. Cheng, "RDF for temporal data management – a survey," *Earth Science Informatics*, vol. 14, no. 2, pp. 563–599, 2021. DOI: 10.1007/s12145-021-00574-w.

[34] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "Evaluating query and storage strategies for RDF archives," *Semantic Web*, vol. 10, no. 2, pp. 247–291, 2019. DOI: 10.3233/SW-180309.

[35] S. Stefanova and T. Risch, "Scalable long-term preservation of relational data through SPARQL queries," *Semantic Web*, vol. 7, no. 2, pp. 117–137, 2016, Cited by: 1. DOI: 10.3233/SW-150173.

[36] R. Snodgrass and I. Ahn, "A taxonomy of time in databases," *ACM SIGMOD Rec.*, vol. 14, no. 4, pp. 236–246, 1985. DOI: 10.1145/971699.318921.

[37] C. S. Jensen, C. E. Dyreson, M. Bohlen, *et al.*, "The consensus glossary of temporal database concepts – February 1998 version," *Lect. Notes Comput. Sci.*, vol. 1399, pp. 367–405, 1998. DOI: 10.1007/bfb0053710.

[38] T. Johnston, *Bitemporal Data: Theory and Practice*. Waltham, MA, USA: Morgan Kaufmann, 2014. DOI: 10.1016/C2012-0-06609-4.

[39] *Information technology – Database languages SQL – Part 2: Foundation (SQL/-Foundation)*, ISO/IEC 9075-2. 2023.

[40] P. Bruni, R. Garcia, S. Kaschta, *et al.*, *DB2 10 for z/OS Technical Overview*. Armonk, NY, USA: IBM, 2010, Accessed: Apr. 17, 2024. [Online]. Available: https://www.redbooks.ibm.com/redbooks/pdfs/sg247892.pdf.

[41] MariaDB, "System-versioned tables," *MariaDB KnowledgeBase*, Mar. 12, 2024, Accessed: Apr. 17, 2024. [Online]. Available: https://mariadb.com/kb/en/system-versioned-tables/.

[42] T. Burroughs and S. Cheevers, "SELECT," *Oracle9i SQL Reference Release 2 (9.2)*, Mar. 2002, Accessed: May 6, 2024. [Online]. Available: https://docs.oracle.com/cd/A97630_01/server.920/a96531/title.htm.

[43] C. M. Saracco, M. Nicola, and L. Gandhi, "A matter of time: Temporal data management in DB2 for z/OS," IBM, Armonk, NY, USA, White paper, 2010, Accessed: Apr. 17, 2024. [Online]. Available: https://public.dhe.ibm.com/software/data/sw-library/db2/papers/A_Matter_of_Time_-_DB2_zOS_Temporal_Tables_-_White_Paper_v1.4.1.pdf.

[44] E. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970. DOI: 10.1145/362384.362685.

[45] R. Snodgrass, *The TSQL2 Temporal Query Language*. New York, NY, USA: Springer, 1995.

[46] J. Gamper, M. Ceccarello, and A. Dignös, "What's new in temporal databases?" *Lect. Notes Comput. Sci.*, vol. 13389 LNCS, pp. 45–58, 2022. DOI: 10.1007/978-3-031-15740-0_5.

[47] M. H. Böhlen, A. Dignös, J. Gamper, and C. S. Jensen, "Temporal data management – an overview," *Lect. Notes Bus. Inf. Process.*, vol. 324, pp. 51–83, 2018, Accessed: May 7, 2024. DOI: 10.1007/978-3-319-96655-7_3.

[48] PostgreSQL GDG, "Release notes," *Documentation - PostgreSQL 9.2*, Sep. 10, 2012, Accessed: May 7, 2024. [Online]. Available: https://www.postgresql.org/docs/release/9.2.0/.

[49] PostgreSQL GDG, "D.2. unsupported features," *Documentation - PostgreSQL 16*, Accessed: May 4, 2024. [Online]. Available: https://www.postgresql.org/docs/16/unsupported-features-sql-standard.html.

[50] A. U. Tansel, "Temporal data modelling and integrity constraints in relational databases*," *Int. J. Comput. Math. Comput. Syst. Theory"*, vol. 9, no. 1, pp. 1–20, 2024. DOI: 10.1080/23799927.2023.2300083.

[51] L. Liu and M. T. Özsu, *Encyclopedia of Database Systems*. New York, NY, USA: Springer, 2009. DOI: 10.1007/978-0-387-39940-9.

[52] IBM, “Natural key analysis,” *IBM InfoSphere Information Server 11.7 Documentation*, Feb. 26, 2021, Accessed: May 2, 2024. [Online]. Available: https://www.ibm.com/docs/en/iis/11.7?topic=relationships-natural-key-analysis.

[53] IBM, “Surrogate keys,” *IBM Db2 10.5 Documentation*, Mar. 1, 2021, Accessed: May 2, 2024. [Online]. Available: https://www.ibm.com/docs/en/db2/10.5?topic=operators-surrogate-keys.

[54] OpenAI, *ChatGPT large language model*, 2024. [Online]. Available: https://chatgpt.com/.

[55] IBM, “CREATE TABLE statement,” *IBM Db2 11.5 Documentation*, Jan. 25, 2024, Accessed: May 1, 2024. [Online]. Available: https://www.ibm.com/docs/en/db2/11.5?topic=statements-create-table#sdx-synid_enforced.

[56] WilliamDAssafMSFT, rwestMSFT, rothja, *et al.*, “Disable foreign key constraints with INSERT and UPDATE statements,” *Microsoft Learn SQL*, Mar. 3, 2023, Accessed: May 1, 2024. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/tables/disable-foreign-key-constraints-with-insert-and-update-statements?view=sql-server-ver16.

[57] MariaDB, “Foreign_key_checks,” *MariaDB Documentation*, Accessed: May 1, 2024. [Online]. Available: https://mariadb.com/docs/server/ref/mdb/system-variables/foreign_key_checks/.

[58] rwestMSFT, rothja, pritamso, *et al.*, “Temporal tables,” *Microsoft Learn SQL*, Oct. 16, 2023, Accessed: May 5, 2024. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables?view=sql-server-ver16.

[59] rwestMSFT, rothja, pcpronk, *et al.*, “Modifying data in a system-versioned temporal table,” *Microsoft Learn SQL*, Mar. 1, 2023, Accessed: May 3, 2024. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/tables/modifying-data-in-a-system-versioned-temporal-table?view=sql-server-ver16.

[60] K. Kulkarni and J.-E. Michels, “Temporal features in SQL:2011,” *SIGMOD Rec.*, vol. 41, no. 3, pp. 34–43, 2012, Cited by: 138. DOI: 10.1145/2380776.2380786.

[61] craigloewen-msft, mattwojo, suelsp, *et al.*, “How to install Linux on Windows with WSL,” *Microsoft Learn Windows*, Aug. 28, 2023, Accessed: May 13, 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/install.

[62] E. Codd, “Extending the database relational model to capture more meaning,” *ACM Trans. Database Syst. (TODS)*, vol. 4, no. 4, pp. 397–434, 1979. DOI: 10.1145/320107.320109.

[63] ysth and Tim, “Selecting from SYSTEM VERSIONING with BETWEEN not working?” *stack overflow*, Apr. 4, 2024, Accessed: Apr. 21, 2024. [Online]. Available: https://stackoverflow.com/questions/78276246/selecting-from-system-versioning-with-between-not-working.

[64] MariaDB, “JOIN syntax,” *MariaDB KnowledgeBase*, Jan. 18, 2018, Accessed: May 11, 2024. [Online]. Available: https://mariadb.com/kb/en/join-syntax/.

[65] B. Kelechava, “The SQL Standard – ISO/IEC 9075:2023 (ANSI X3.135),” *ANSI Blog*, 2023, Accessed: Apr. 18, 2024. [Online]. Available: https://blog.ansi.org/sql-standard-iso-iec-9075-2023-ansi-x3-135/.

[66] Rigsarkivet, “Leverandøroversigt,” Copenhagen, Denmark, Government record, 2022, Accessed: May 8, 2024. [Online]. Available: https://en.rigsarkivet.dk/wp-content/uploads/2022/08/Leverandoroversigt-februar-2022.pdf.

[67] C. De Castro, F. Grandi, and M. R. Scalas, “Schema versioning for multitemporal relational databases,” *Inf. Syst.*, vol. 22, no. 5, pp. 249–290, 1997, Cited by: 62. DOI: 10.1016/S0306-4379(97)00017-3.

[68] L. Caruccio, G. Polese, and G. Tortora, “Synchronization of queries and views upon schema evolutions: A survey,” *ACM Trans. Database Syst.*, vol. 41, no. 2, 2016, Cited by: 28; All Open Access, Bronze Open Access. DOI: 10.1145/2903726.

[69] Z. Brahmia, H. Hamrouni, and R. Bouaziz, “TempoX: A disciplined approach for data management in multi-temporal and multi-schema-version XML databases,” *J. King Saud Univ. Comput. Inf. Sci*, vol. 34, no. 1, pp. 1472–1488, 2022, Cited by: 7; All Open Access, Gold Open Access. DOI: 10.1016/j.jksuci.2019.08.009.

[70] TalTech, *Author guidelines and formatting requirements for thesis preparation*, Accessed: Apr. 17, 2024, Apr. 27, 2021. [Online]. Available: https://haldus.taltech.ee/sites/default/files/2021-04/FIT_Author_Guidelines_ENG.pdf.

[71] T. Nugteren, J. Buijs, F. Korving, and K. Janson, “Tallinn University of Technology - bachelor, master thesis template,” *Overleaf*, Accessed: Apr. 3, 2024. [Online]. Available: https://www.overleaf.com/latex/templates/tallinn-university-of-technology-bachelor-master-thesis-template/ptxvgdhnvmhc.

[72] IEEE, *Reference guide*, Accessed: Apr. 17, 2024, Nov. 29, 2023. [Online]. Available: http://journals.ieeeauthorcenter.ieee.org/wp-content/uploads/sites/7/IEEE_Reference_Guide.pdf.

[73] IEEE, *IEEE editorial style manual for authors*, Accessed: Apr. 17, 2024, Feb. 29, 2024. [Online]. Available: http://journals.ieeeauthorcenter.ieee.org/wp-content/uploads/sites/7/IEEE-Editorial-Style-Manual-for-Authors.pdf.

[74] E. L. Ayubi, C. L. Bromstad Lee, H. S. Kamin, T. L. McAdoo, A. T. Woodworth, and A. A. Adams, *Publication Manual of the American Psychological Association, Seventh Edition (2020)*. Washington, DC, USA: APA, 2020.

[75] S. Vinz, "Capitalization in titles and headings," *Scribbr*, Jul. 23, 2023, Accessed: Apr. 17, 2024. [Online]. Available: https://www.scribbr.com/academic-writing/capitalization-titles-headings/.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Koit Saarevet

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Methods of Access to Series of Archived Database Snapshots", supervised by Innar Liiv
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

18.05.2024

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Source Code Listing

The three "Create procedure . . . " scripts are kept separate for clearer presentation, otherwise they could have been placed into one .sql file.

A2.1 Main_workflow.sh

This is the main program. Implemented as a Bash script because the Unix shell is familiar to a wider audience than the MariaDB/MySQL management tools. Its job is to create the blank databases, enable the temporal features, create the SQL stored procedures and then run the merging procedure.

Having a password inside a source file, in plain text, is a painful sight for any security-conscious person, but it is used here as the least effort solution. This MariaDB server was a local instance without network access.

```
#!/bin/bash

# Set MariaDB credentials and db name
user="dba"
password="pwd123"
db="vehreg_" # root name for databases
snapshots_count=5 # number of snapshots
host_name="localhost"
port_number=3306
siard_folder="Datafiles" # Use "." for current folder

# This workflow assumes:
# - MariaDB server and an administrator account for accessing it
# - $snapshots_count identically structured SIARD files
# - SIARDs are ordered chronologically and named ${db}s${i}.siard,
↪  where:
#    - ${db} is the root of the name and
#    - ${i} is a continuously incrementing integer {1, 2, ...,
↪  $snapshots_count}

# Step 1: Create blank databases
# s0 - aggregate temporal db, named 0 to simplify for-loops
# s[1..n] - source snapshots
for ((i = 0; i <= snapshots_count; i++)); do
```

```
  mariadb -u $user -p"$password" <<-EOB

    CREATE DATABASE ${db}s${i}
    DEFAULT CHARACTER SET utf8mb3
    DEFAULT COLLATE utf8mb3_general_ci;

  EOB
done
echo "Done: Step 1: Create blank databases"

# Step 2: Load snapshots (structure and data) from SIARD
for ((i = 1; i <= snapshots_count; i++)); do
  echo "Start processing of ${siard_folder}/${db}s${i}.siard" >>
   ↪  "dbptk_${db}_siard_upload_log.txt"
  date "+%H:%M:%S" >> "dbptk_${db}_siard_upload_log.txt"
  java -jar "-Dfile.encoding=UTF-8" /Applications/dbptk-app-2.11.0.jar
   ↪  migrate \
  --import=siard-2 \
  --import-file="${siard_folder}/${db}s${i}.siard" \
  --export=mysql \
  --export-hostname="$host_name" \
  --export-database="${db}s${i}" \
  --export-username="$user" \
  --export-password="$password" \
  --export-port-number="$port_number" \
  1>>"dbptk_${db}_siard_upload_log.txt"
  echo >> "dbptk_${db}_siard_upload_log.txt"
done
echo "Done: Step 2: Load snapshots (structure and data) from SIARD"

# Step 3: Clone snapshot 1 to aggregate temporal db (structure only)
mysqldump $db"s1" --no-data -u $user -p"$password" | mysql $db"s0" -u
 ↪  $user -p"$password"
echo "Done: Step 3: Clone snapshot 1 to aggregate temporal db
 ↪  (structure only)"

# Step 4: Add system versioning to aggregate temporal db
# Get all table names in the database
tables=$(mariadb -u $user -p"$password" -D $db"s0" -Bse "SHOW TABLES")
# Enable system versioning for each table
for table in $tables; do
  mariadb -u $user -p"$password" -D $db"s0" -e "ALTER TABLE $table ADD
   ↪  SYSTEM VERSIONING;"
done
echo "Done: Step 4: Add system versioning to aggregate temporal db"
```

```
# Step 5: Create procedure: sync_single_table
mariadb $db"s0" -u $user -p"$password" <
 ↪  "Create_procedure_sync_single_table.sql"
echo "Done: Step 5: Create procedure: sync_single_table"


# Step 6: Create procedure: sync_single_table_no_pk
mariadb $db"s0" -u $user -p"$password" <
 ↪  "Create_procedure_sync_single_table_no_pk.sql"
echo "Done: Step 6: Create procedure: sync_single_table_no_pk"


# Step 7: Create procedure: sync_databases (main procedure)
mariadb $db"s0" -u $user -p"$password" <
 ↪  "Create_procedure_sync_databases.sql"
echo "Done: Step 7: Create procedure: sync_databases (main procedure)"


# Step 8: Import snapshots into temporal db
mariadb $db"s0" -u $user -p"$password" <
 ↪  "Import_snapshots_into_temporal_db.sql"
echo "Done: Step 8: Import snapshots into temporal db"
```

A2.2 Create_procedure_sync_single_table.sql

`sync_single_table()` merges the data of one table from the source database to the target temporal database. This procedure relies on primary keys and thus can distinguish between new rows (`INSERT`), changed rows (`UPDATE`) and removed rows (`DELETE`).

```
DELIMITER //
CREATE PROCEDURE sync_single_table(
  IN source_table_name VARCHAR(128), -- fully qualified, i.e.
   ↪  database_name.table_name
  IN target_table_name VARCHAR(128) -- fully qualified, i.e.
   ↪  database_name.table_name
)

BEGIN

  -- Case 1: INSERT
  -- The row didn't exist in the last snapshot, i.e., it has been
   ↪  created
  -- after the last snapshot was made.
```

```
SET @insert_query = CONCAT(
  'INSERT INTO ', target_table_name, ' (', @all_col_names_list, ')',
  ' SELECT ', @all_col_names_s_list,
  ' FROM ', source_table_name, ' s',
  ' LEFT JOIN ', target_table_name, ' t ON ', @pk_col_join_on_clause,
  ' WHERE t.', @col1_name, ' IS NULL;'
);
-- @all_col_names_list =  "a, b, c"
-- @all_col_names_s_list = "s.a, s.b, s.c"
-- @pk_col_join_on_clause = "s.a = t.a AND s.b = t.b" for PK(a, b)

PREPARE stmt_insert FROM @insert_query;
EXECUTE stmt_insert;


-- Case 2: UPDATE
-- The row already existed in the last snapshot, but at least one
↪  column differs.

SET @update_query = CONCAT(
  'UPDATE ', target_table_name, ' t',
  ' JOIN ', source_table_name, ' s ON ', @pk_col_join_on_clause,
  ' SET ', @set_clause,
  ' WHERE ', @where_clause, ';'
);
-- @set_clause = "t.b = s.b, t.c = s.c" for a table (a, b, c) with
↪  PK(a)
-- @where_clause = "t.b != s.b OR t.c != s.c" for a table (a, b, c)
↪  with PK(a)
PREPARE stmt_update FROM @update_query;
EXECUTE stmt_update;


-- Case 3: DELETE
-- The row existed in the last snapshot, but not in the current one,
-- i.e., it was deleted after the last snapshot was made.

SET @delete_query = CONCAT(
  'DELETE t',
  ' FROM ', target_table_name, ' t',
  ' LEFT JOIN ', source_table_name, ' s ON ', @pk_col_join_on_clause,
  ' WHERE s.', @col1_name, ' IS NULL;'
);

PREPARE stmt_delete FROM @delete_query;
EXECUTE stmt_delete;
```

```
  DEALLOCATE PREPARE stmt_insert;
  DEALLOCATE PREPARE stmt_update;
  DEALLOCATE PREPARE stmt_delete;
END;
//
DELIMITER ;
```

A2.3 Create_procedure_sync_single_table_no_pk.sql

`sync_single_table_no_pk()` is similar to `sync_single_table()` in that it merges one table, but it is used for tables that have no formally defined primary key. Therefore it only handles two scenarios: new rows (`INSERT`) and removed rows (`DELETE`). Changed rows are indistinguishable from the combination of `DELETE` and `INSERT`, so there is no separate handling for them.

```
DELIMITER //
CREATE PROCEDURE sync_single_table_no_pk(
  IN source_table_name VARCHAR(128), -- fully qualified, i.e.
   ↪  database_name.table_name
  IN target_table_name VARCHAR(128) -- fully qualified, i.e.
   ↪  database_name.table_name
)

BEGIN
  -- Case 1: INSERT
  -- No row with this exact combination of column values existed in the
   ↪  last snapshot,
  -- i.e., it has been created after the last snapshot was made. This
   ↪  is logically
  -- equivalent to an existing row having been updated, as rows without
   ↪  keys have no
  -- identity (RDBMS may implement autogenerated hidden primary keys
   ↪  but these are not
  -- visible to the user).

  SET @insert_query = CONCAT(
    'INSERT INTO ', target_table_name, ' (', @all_col_names_list, ')',
    ' SELECT ', @all_col_names_s_list,
    ' FROM ', source_table_name, ' s',
    ' LEFT JOIN ', target_table_name, ' t ON ',
     ↪  @all_col_join_on_clause,
    ' WHERE t.', @col1_name, ' IS NULL;'
  );
  -- @all_col_names_list "a, b, c"
  -- @all_col_names_s_list "s.a, s.b, s.c"
  -- @all_col_join_on_clause "s.a = t.a AND s.b = t.b AND s.c = t.c"


  PREPARE stmt_insert FROM @insert_query;
  EXECUTE stmt_insert;

  -- Case 2: DELETE
  -- The row existed in the last snapshot, but not in the current one,
  -- i.e., it was deleted after the last snapshot was made. This is
   ↪  logically
  -- equivalent to an update, see the note above for INSERT.

  SET @delete_query = CONCAT(
    'DELETE t',
    ' FROM ', target_table_name, ' t',
    ' LEFT JOIN ', source_table_name, ' s ON ',
     ↪  @all_col_join_on_clause,
    ' WHERE s.', @col1_name, ' IS NULL;'
  );

  PREPARE stmt_delete FROM @delete_query;
  EXECUTE stmt_delete;

  DEALLOCATE PREPARE stmt_insert;
  DEALLOCATE PREPARE stmt_delete;

END;
//
DELIMITER ;
```

A2.4 Create_procedure_sync_databases.sql

`sync_databases()` is the main procedure for managing the merging inside the RDBMS. Its core function is to iterate over all tables in the database and, depending on the existence of a primary key, call the appropriate procedure to merge that table. Much of the code are the creators of the strings that are needed to build the dynamic SQL statements that do the actual merging.

```
DELIMITER //
CREATE PROCEDURE sync_databases(IN source_db VARCHAR(64), IN target_db
 ↪  VARCHAR(64), IN snapshot_time TIMESTAMP(6))
BEGIN
```

```
DECLARE done INT DEFAULT 0; -- cursor control variable
DECLARE tbl_name VARCHAR(64);
DECLARE cur CURSOR FOR
  SELECT TABLE_NAME
  FROM INFORMATION_SCHEMA.TABLES
  WHERE TABLE_SCHEMA = source_db;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

SET FOREIGN_KEY_CHECKS = 0;

-- MariaDB uses @@timestamp to fill the row_start and row_end
 ↪  columns.
-- Change it to use snapshot_time instead of the current clock time.
SET @@timestamp = UNIX_TIMESTAMP(snapshot_time);

OPEN cur;
-- Loop through all tables in source_db
read_loop: LOOP
  FETCH cur INTO tbl_name;

  IF done THEN
    LEAVE read_loop;
  END IF;

  -- Generate the dynamic SQL parts for the sync_single_table()
   ↪  procedure

  -- @pk_col_names_list
  -- List of PK column names
  -- Provided as "a, b" for PK(a, b)
  SET @pk_col_names_list = (
    SELECT GROUP_CONCAT(COLUMN_NAME)
    FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
    WHERE CONSTRAINT_SCHEMA = source_db AND CONSTRAINT_NAME =
     ↪  'PRIMARY' AND TABLE_NAME = tbl_name
  );
  SET @pk_col_names_list = IFNULL(@pk_col_names_list, '');

  -- @pk_col_names_quot_list
  -- List of PK column names in quotes
  -- Provided as "'a', 'b'" for PK(a, b)
  SET @pk_col_names_quot_list = (
    SELECT GROUP_CONCAT(CONCAT("'", COLUMN_NAME, "'"))
    FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
    WHERE CONSTRAINT_SCHEMA = source_db AND CONSTRAINT_NAME =
     ↪  'PRIMARY' AND TABLE_NAME = tbl_name
  );
```

```
-- @col1_name
-- Name of column 1
SET @col1_name = (
  SELECT COLUMN_NAME
  FROM INFORMATION_SCHEMA.COLUMNS
  WHERE TABLE_SCHEMA = source_db
  AND TABLE_NAME = tbl_name AND ORDINAL_POSITION = 1
);

-- @all_col_names_list
-- List of all columns
-- Provided as "a, b, c" for a table with columns (a, b, c)
SET @all_col_names_list = (
  SELECT GROUP_CONCAT(COLUMN_NAME ORDER BY ORDINAL_POSITION)
  FROM INFORMATION_SCHEMA.COLUMNS
  WHERE TABLE_SCHEMA = source_db
  AND TABLE_NAME = tbl_name
);

-- @all_col_names_s_list
-- List of all columns prefixed "s."
-- Provided as "s.a, s.b, s.c" for a table with columns (a, b, c)
SET @all_col_names_s_list = (
  SELECT GROUP_CONCAT(CONCAT("s.", COLUMN_NAME) ORDER BY
   ↪  ORDINAL_POSITION)
  FROM INFORMATION_SCHEMA.COLUMNS
  WHERE TABLE_SCHEMA = source_db
  AND TABLE_NAME = tbl_name
);

IF @pk_col_names_list != '' THEN -- PK exists, consists of 1 or
 ↪  more columns

  -- @pk_col_join_on_clause
  -- List of PK columns for JOIN ON clause
  -- Provided as "s.a = t.a AND s.b = t.b" for PK(a, b)
  SET @pk_col_join_on_clause = (
    SELECT GROUP_CONCAT(CONCAT("s.", COLUMN_NAME, " = t.",
     ↪  COLUMN_NAME) ORDER BY ORDINAL_POSITION SEPARATOR " AND ")
    FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
    WHERE CONSTRAINT_SCHEMA = source_db AND CONSTRAINT_NAME =
     ↪  'PRIMARY' AND TABLE_NAME = tbl_name
  );

  -- @pk_col_names_compare_clause
  -- List of PK columns for "WHERE COLUMN_NAME !=" clause
-- Provided as "AND COLUMN_NAME != pk_col1_name AND COLUMN_NAME
↪  != pk_col2_name"
-- for PK(pk_col1_name, pk_col2_name)
SET @pk_col_names_compare_clause = (
  SELECT CONCAT(" AND ", GROUP_CONCAT(CONCAT("COLUMN_NAME != '",
  ↪  COLUMN_NAME, "'") ORDER BY ORDINAL_POSITION SEPARATOR " AND
  ↪  "))
  FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
  WHERE CONSTRAINT_SCHEMA = source_db AND CONSTRAINT_NAME =
  ↪  'PRIMARY' AND TABLE_NAME = tbl_name
);
SET @pk_col_names_compare_clause =
↪  IFNULL(@pk_col_names_compare_clause, '');

-- @data_col_names_list
-- List of data columns, i.e. all columns except PK
-- Provided as "c, d, e" for a table (a, b, c, d, e) with PK(a,
↪  b)
SET @data_col_names_dyn_query = CONCAT(
  "SET @data_col_names_list = (",
    "SELECT GROUP_CONCAT(COLUMN_NAME)",
    " FROM INFORMATION_SCHEMA.COLUMNS",
    " WHERE TABLE_SCHEMA = '", source_db, "'",
    " AND TABLE_NAME = '", tbl_name, "'",
    @pk_col_names_compare_clause,
  ");"
);

PREPARE stmt_data_col_names_composer FROM
↪  @data_col_names_dyn_query;
EXECUTE stmt_data_col_names_composer;
DEALLOCATE PREPARE stmt_data_col_names_composer;
SET @data_col_names_list = IFNULL(@data_col_names_list, '');

-- @set_clause
-- List of data column pairs for SET clause
-- Provided as "t.b = s.b, t.c = s.c" for a table (a, b, c) with
↪  PK(a)
SET @set_clause = (
  SELECT GROUP_CONCAT(CONCAT('t.', COLUMN_NAME, ' = s.',
  ↪  COLUMN_NAME))
  FROM INFORMATION_SCHEMA.COLUMNS
  WHERE TABLE_SCHEMA = source_db
  AND TABLE_NAME = tbl_name
  AND COLUMN_NAME NOT IN (@pk_col_names_quot_list)
);
```

```
      -- @where_clause
      -- List of data column pairs for WHERE clause
      -- Provided as "t.b != s.b OR t.c != s.c" for a table (a, b, c)
      ↪  with PK(a)
      SET @where_clause = (
        SELECT GROUP_CONCAT(CONCAT('t.', COLUMN_NAME, ' != s.',
        ↪  COLUMN_NAME) SEPARATOR ' OR ')
        FROM INFORMATION_SCHEMA.COLUMNS
        WHERE TABLE_SCHEMA = source_db
        AND TABLE_NAME = tbl_name
        AND COLUMN_NAME NOT IN (@pk_col_names_quot_list)
      );

      CALL sync_single_table(CONCAT(source_db, '.', tbl_name),
      ↪  CONCAT(target_db, '.', tbl_name));

    ELSE -- no PK is defined for the table

      -- @all_col_join_on_clause
      -- List of all columns for JOIN ON clause
      -- Provided as "s.a = t.a AND s.b = t.b AND s.c = t.c" for a
      ↪  table (a, b, c)
      SET @all_col_join_on_clause = (
        SELECT GROUP_CONCAT(CONCAT("s.", COLUMN_NAME, " = t.",
        ↪  COLUMN_NAME) ORDER BY ORDINAL_POSITION SEPARATOR " AND ")
        FROM INFORMATION_SCHEMA.COLUMNS
        WHERE TABLE_SCHEMA = source_db AND TABLE_NAME = tbl_name
      );

      CALL sync_single_table_no_pk(CONCAT(source_db, '.', tbl_name),
      ↪  CONCAT(target_db, '.', tbl_name));
    END IF;
  END LOOP;

  CLOSE cur;
  SET @@timestamp = default;
  SET FOREIGN_KEY_CHECKS = 1;
END;
//
DELIMITER ;
```

A2.5 Import_snapshots_into_temporal_db.sql

`Import_snapshots_into_temporal_db.sql` is effectively a configuration file that sets the snapshot creation time for each snapshot. It could have been designed as a traditional config file with key-value pairs, e.g., {snapshot_id, timestamp} and the `CALL` statements executed from a loop in `Main_workflow.sh`, but this optimisation idea, like many others, did not get implemented due to time restrictions.

```sql
-- The timestamps here are used as snapshot timestamps in the aggregate
↪  temporal db
CALL sync_databases('vehreg_s1', 'vehreg_s0', '2000-01-01
↪  00:00:00.000000');
CALL sync_databases('vehreg_s2', 'vehreg_s0', '2005-01-01
↪  00:00:00.000000');
CALL sync_databases('vehreg_s3', 'vehreg_s0', '2010-01-01
↪  00:00:00.000000');
CALL sync_databases('vehreg_s4', 'vehreg_s0', '2015-01-01
↪  00:00:00.000000');
CALL sync_databases('vehreg_s5', 'vehreg_s0', '2020-01-01
↪  00:00:00.000000');
```

Appendix 3 – Installation Instructions

The following instructions apply for macOS and assume the package manager Macports is installed (see https://www.macports.org/install.php).

A3.1 MariaDB

```
# Determine the newest available version and the exact name of the port
$ port search mariadb

# Install the port
$ sudo port install mariadb-10.11

# Run MariaDB installer
$ sudo -u _mysql /opt/local/lib/mariadb-10.11/bin/mysql_install_db

# Set the PATH variable
$ echo "/opt/local/lib/mariadb-10.11/bin" | sudo tee
↪  /etc/paths.d/mariadb

# Run the MariaDB server
$ cd '/opt/local' ; sudo -u _mysql
↪  /opt/local/lib/mariadb-10.11/bin/mysqld_safe
↪  --datadir='/opt/local/var/db/mariadb-10.11'

# Create user dba
# From an account that has sudo rights, sudo to root privileges and
↪  connect to mariadb.
# It uses the unix_socket authentication, i.e., no password needed
$ sudo -u _mysql mariadb

CREATE USER 'dba'@'localhost' IDENTIFIED BY 'pwd123';
GRANT ALL PRIVILEGES ON *.* TO 'dba'@'localhost' WITH GRANT OPTION;
FLUSH PRIVILEGES;
EXIT;

# To shut down the server:
$ sudo -u _mysql /opt/local/lib/mariadb-10.11/bin/mariadb-admin
↪  SHUTDOWN
```

A3.2 DBPTK

Download the dbptk-developer from:

https://github.com/keeps/dbptk-developer/releases

A3.3 DBeaver

DBeaver is not strictly mecessary, but has a highly functional, time-saving GUI.

```
$ sudo port install dbeaver-community
```

Appendix 4 – Formatting Conventions

This appendix lists some of the deliberate formatting decisions applied to this document.

The foundation was formed by the School of IT's "Author guidelines and formatting requirements for thesis preparation" [70] and the LaTeX template [71], both linked from https://taltech.ee/en/thesis-and-graduation-it.

The next layer of conventions was taken from IEEE "Reference Guide" [72], i.e., the list of references is formatted as closely as possible to its requirements. Unfortunately it is not fully achieved due to the limitations of the default styles in BibLaTeX, the technology used for reference management in LaTeX. Achieving perfection here would have required unreasonable effort and caused the content of the thesis to suffer. However, consistency is maintained, in that each kind of imperfection appears in all instances of that type of source.

URLs are provided as clickable links for the benefit of the reader, in conflict with "URLs are not hyperlinked in the proof" [72, p. 22]. The perfectly reasonable IEEE guidelines for URL breaking (e.g., "Break 'before' the hyphen that is part of an address, but do not break after") [72, p. 22] are not fully met due to limitations of the default macros in LaTeX.

"IEEE Editorial Style Manual for Authors" [73] was used for guidance in more general stylistic issues, such as the spelling and singular form of the chapter title "Acknowledgment."

The use of verb tense was guided by APA Publication Manual, [74, Sec. 4.12].

Capitalisation of titles and headings was inspired by the recommendations of Scribbr, a language services company [75], and the appearance of TalTech's guidelines [70], then finalised by the author of the thesis:

- Thesis title is in title case (e.g., "This is Title Case").
- Chapter titles are in title case.
- Section titles and below are in sentence case (e.g., "This is sentence case").
- Captions for figures, tables and code listings are in sentence case.

Latin words, phrases, and abbreviations that appear in English dictionaries are not italicised

[74, Sec. 6.22].

The rule against single-paragraph, or worse, single-sentence sections is knowingly violated in sections 4.2.1 – 4.2.4 for the purpose of listing these steps in the Table of Contents.

The uneven line spacing in the List of Figures and the List of Tables is not a formatting convention, it is a LaTeX bug. Apologies for the aesthetic atrocity.