Informatics

Reproducible Query Processing in Relational Databases with Evolving Database Schemas

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

066 645 Data Science

eingereicht von

Moritz Staudinger, BSc
Matrikelnummer 11777768

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Mitwirkung: Dr.techn. Mag. Tomasz Miksa

Wien, 4. Mai 2023

Moritz Staudinger   Andreas Rauber

Informatics

Reproducible Query Processing in Relational Databases with Evolving Database Schemas

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

066 645 Data Science

by

Moritz Staudinger, BSc
Registration Number 11777768

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Assistance: Dr.techn. Mag. Tomasz Miksa

Vienna, 4th May, 2023

Moritz Staudinger

Andreas Rauber

Erklärung zur Verfassung der Arbeit

Moritz Staudinger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. Mai 2023

Moritz Staudinger

Danksagung

Wie bei meiner Bachelorarbeit, möchte ich mich bei den vielen helfenden Händen, die mir in so vielen verschiedenen Belangen geholfen haben, bedanken. Egal ob diese Hilfe von meinen Studien- oder Arbeitskollegen, meinen Betreuern, dem ISMN Team oder von Freunden oder Verwandten gekommen ist, sie war immer willkommen, egal in welcher Form.

Allen voran möchte ich mich bei meinen beiden Betreuern Dr. Andreas Rauber und Dr. Tomasz Miksa bedanken. Ihr wart immer für mich da, egal wann ich eure Hilfe gebraucht habe, und über welche Herausforderungen ich während meiner Arbeit gestolpert bin.

Desweiteren will ich mich bei dem Team des International Soil Moisture Networks der TU Wien bedanken, welche mir ihre Datenbank als Real-World Use Case und zum Evaluieren meiner Lösung zur Verfügung gestellt haben und mir auch bei Fragen immer hilfreich zur Seite gestanden sind.

Auch möchte ich mich bei meiner Freundin und meiner Familie bedanken, danke dass ihr mich immer wieder motiviert habt und auch wenn notwendig abgelenkt habt, damit ich weiter nach der besten Lösung für meine Probleme suche und nicht an der ganzen Arbeit verzweifle.

Acknowledgements

As with my bachelor's thesis, I would like to thank the many helping hands that have helped me in so many different ways. Whether this help has come from my study or work colleagues, my supervisors, the ISMN team, or from friends or relatives, it has always been appreciated, no matter which form it took.

First and foremost, I would like to thank my two supervisors, Dr. Andreas Rauber and Dr. Tomasz Miksa. You were always there for me no matter when I needed your help and what challenges I stumbled upon during my work.

Furthermore, I want to thank the team of the International Soil Moisture Network of the Vienna University of Technology, who provided me with their database as a real-world use case and to evaluate my solution on, and also always helped me with questions.

I would also like to thank my girlfriend and my family, thank you for always motivating me and distracting me when necessary, so that I continue to search for the best solution to my problems and do not despair of all the work.

Kurzfassung

Heutzutage ändern und entwickeln sich Daten ständig, egal ob es sich nun um Texte, Websites, Tweets oder Sensormesswerte handelt. All diese verschiedenen Arten von Daten müssen in irgendeiner Form gespeichert werden, sei es in einer dateibasierten Struktur, einer relationalen Datenbank, einer Graphdatenbank oder einer anderen Form.

Wenn Sie von Ihnen verwendeten Daten referenzieren oder sie sogar zitieren wollen, beginnen die Probleme. Die zugrundeliegenden Daten sind umgezogen, die Daten haben sich geändert, oder die Struktur der Daten hat sich geändert und ist nicht mehr verfügbar. In der Forschung helfen diese Daten, wissenschaftliche Entdeckungen zu beschleunigen und Ergebnisse zu verifizieren, wenn die Daten verfügbar sind. In den letzten Jahren hat es in diesem Bereich große Fortschritte gegeben, da die Zitierung von Daten immer wichtiger geworden ist und viele Konferenzen von den Autoren verlangen, dass sie ihre verwendeten und generierten Daten zur Verfügung stellen. Wenn die generierten Daten statisch sind, kann dies durch Hochladen des Datensatzes geschehen, aber für sich dynamisch entwickelnde Datensätze wäre dies ineffektiv. Daher ist es wichtig bestehende Ansätze für die dynamische Datenzitierbarkeit weiterzuentwickeln.

In dieser Masterarbeit präsentieren wir ein Framework für dynamische Datenzitierbarkeit in PostgreSQL. Unsere Arbeit besteht darin, drei verschiedene Ansätze zur Datenversionierung für PostgreSQL zu implementieren und zu untersuchen wie diese sich in einem realen Szenario verhalten. Wir haben die RDA Dynamic Data Citation Guidelines auf das International Soil Moisture Network angewendet und die Auswirkungen auf die Leistung gemessen. Außerdem haben wir eine Architektur zur Speicherung von Abfragen vorgestellt, die es ermöglicht, ein Set von Abfragen zu zitieren und gleichzeitig die Korrektheit der Reproduzierbarkeit jeder einzelnen Abfrage zu überprüfen. Da Schemaänderungen in Forschungsdatenbanken relevant sind, haben wir untersucht, wie Schemaänderungen automatisch implementiert werden können, um die Reproduzierbarkeit zuvor ausgeführter Abfragen zu gewährleisten.

Unsere Implementierung ist verfügbar unter MIT license online[1].

[1]`https://github.com/MoritzStaudinger/sql-data-versioner`

Abstract

In today's world, data is constantly changing and evolving, whether it is text, websites, tweets or sensor readings. All these different types of data need to be stored in some form, whether it is in a file-based structure, a relational database, a graph database or some other form. If you want to reference or even cite the data you are using, the problems start. The underlying data has moved, the data has changed, or the structure of the data has changed and is no longer available. In research, this data helps to speed up scientific discovery and to verify results when the data is available. There has been much progress in this area in recent years, as data citation has become increasingly important and many conferences now require authors to provide their used and generated data. If the generated data is static, this can be done by uploading the dataset, but for dynamically evolving datasets this would be ineffective. Therefore, it is necessary to extent existing solutions for dynamic data citation.

In this master thesis we proposed and evaluated a framework for dynamic data citation in PostgreSQL. Our work consists of implementing and evaluating three different data versioning approaches and then adapt our framework to fit the need of a real-world scenario and evaluate it on this scenario. We applied the RDA Dynamic Data Citation Guidelines to the International Soil Moisture Network and measured the impact on performance. We also presented a query storage architecture that allows sets of queries to be cited at once, while verifying the correctness of the reproducibility of each query. As schema changes are common in research databases, we also evaluated how schema changes can be automatically implemented to ensure the reproducibility of previously executed queries.

Our implementation is available under MIT license online[2].

[2]`https://github.com/MoritzStaudinger/sql-data-versioner`

Contents

CHAPTER 1

Introduction

1.1 Motivation

A desirable characteristic of research is to allow the reproduction of experiments. Starting with a hypothesis, researchers collect and analyze data, as a foundation of their scientific experiments. These experiments are essential for expanding the global knowledge and the state of scientific research. As research is increasingly shifting to data-intensive approaches, it becomes more and more important to cite datasets, which have been used, as well. By citing these datasets, researchers receive proper credit for their data contributions and data can be reused and reproduced by other researchers.

The FAIR principles[22] have been established by the scientific community to promote the sharing and reuse of digital resources, and lead to more effective and impactful scientific research. These principles aim to make research data findable, accessible, interoperable and reusable. This is done by promoting the use of standardized metadata and ontologies to describe digital resources and encourage open standards and formats for data and metadata.

A current solution for datasets, which are not subject to change and thus are static, is to provide them in a file-based data repository as InvenioRDM[1]. This is not suitable for all situations, as data is changing and data formats are changing. Data Providers of evolving datasets, such as databases, are now facing the issue of providing different versions of their dataset so that all researchers can work on the latest data, while also ensuring the scientific integrity of their dataset to allow verifying results of other researchers. Trivial solutions for such use cases, range from storing all extracted subsets to only providing weekly or monthly freezes of the database to researchers. All of these approaches would heavily impact the necessary storage load, as huge parts of the database would be saved redundantly. Other more advanced settings as tracking the difference between two

[1]`https://inveniosoftware.org/products/rdm/`

database versions via file-based solutions such as git is another valid approach, which leads to long computation times, to extract the correct version of the database. Up to now, the most promising approach for databases is tuple- or record-based versioning, which tracks changes in a database for each tuple, to allow the re-execution of already issued queries, by retrieving the tuples present in the database at the time of original execution.

To assist with the implementation of custom approaches for persistent identification of arbitrary subsets of evolving datasets, the Working Group on Data Citation (WGDC) of the Research Data Alliance published 14 recommendations for Data Citation in dynamic environments [16]. These recommendations discuss the necessary steps to allow the precise and persistent identification of subsets, by applying versioning to the dataset, and storing executed queries with an execution timestamp and other associated metadata.

Over the last few years, pilot adopters have implemented systems, which are following these guidelines, to show the feasibility and the benefits of such systems.

All of these systems see schema representations as rather static, and require manual adaptions and query rewriting in the case of schema modifications. Other systems, which are supporting schema evolutions, are not supporting the WGDC Guidelines and are therefore not able to reproduce the same results as in the original query. According to a study conducted in 2013 [5], schema evolutions are common in research databases, with over 51 different versions of the CERN-DQ2 database over 1.3 years.

As an increasing amount of conferences and journals[2,3] require researchers to provide their used datasets alongside their published articles, data hosting facilities need to accommodate this necessity. The International Soil Moisture Network wants to support the publication of research data from their platform, and implement these guidelines as the most commonly used in-situ soil moisture database worldwide.

1.2 Aim and Contributions

The aim of this master thesis is to build a novel framework for Dynamic Data Citation in PostgreSQL and adapt this framework to support the needs of the International Soil Moisture Network database [7], and to evaluate performance issues caused by data versioning and reproducible querying to enhance the FAIRness of the underlying data. Furthermore, this framework should include a novel schema versioning approach, which provides automatic schema adaptions and query rewriting for the most common DDL commands.

[2]https://www.springernature.com/gp/authors/research-data-policy/data-availability-statements

[3]https://journals.plos.org/plosone/s/data-availability

Our work makes the following scientific contributions:

1. A prototypical framework for Dynamic Data Citation for PostgreSQL, supporting different versioning approaches and schema evolution's
2. An adaption of the framework to support the specific use-case of the International Soil Moisture team
3. Automatic schema adaptions and query rewritings for the most common database DDL commands
4. Alternative solutions, to prevent race conditions without locking full tables in the database, by using timestamp information or a correction algorithm to obtain the correct results on re-execution

1.3 Thesis Structure

The remainder of this thesis is structured as follows. In Chapter 2 we give an overview of relevant literature to this thesis. We will discuss the FAIR principles [22] the Dynamic Data Citation Guidelines [16] with corresponding prototype implementations and how schema modifications are fitting into these guidelines. In Chapter 3 we describe the International Soil Moisture Network and the current workflows for data integration and retrieval. In Chapter 4 we discuss, which requirements our designed framework has to fulfill and which different design choices are available. In Chapter 5 we further discuss how the framework is implemented, to minimize the integration overhead into existing systems. In Chapter 6 we evaluate our framework qualitatively on the Dynamic Data Citation Guidelines and quantitatively on two different databases. Chapter 7 then summarizes the findings of this thesis and provides an outlook about future research in this area.

CHAPTER 2

Related Work

In this chapter, we will discuss first in Section 2.1 Data Citation and the importance of publishing data. Then we will discuss in Section 2.2 why the FAIRness of research data and code is vital, to speed up scientific discovery and how researchers can ensure that their artifacts are FAIR. We then discuss in Section 2.3 what the RDA Data Citation Guidelines are and how these guidelines should help to make dynamically evolving data sets FAIR, by allowing the reproduction and extraction of subsets of an evolving data set. Next, we will talk in Section 2.3.1 about available reference implementations, which are following the FAIR guidelines, and how they are implementing them in their respective fields. In Section 2.4, we will discuss how different schema changes are handled in databases.

2.1 Data Citation

Data Citation is the general idea of citing static and non-static datasets, by issuing a persistent identifier for them. The importance of data was discussed numerous times over the last decade, as it is an ongoing problem where different solutions have been created. In 2008, Heidorn [10] discussed the importance of data being made available for scientists, as otherwise, it will remain underutilized and eventually lost. Here we will provide a brief history of relevant events in data citation.

In 2009 Costello [4] highlighted the motivation behind publishing data online. He mentioned concerns of researchers, for sharing their data exist and that data citation mechanisms need to be created to allow researchers to officially publish their data. These publications would benefit researchers, by allowing them to reuse data, reanalyze it and verify reached conclusions.

Mons et al. [11] discussed the value of data in 2011 and how to connect datasets with traditional publications best. The authors further mentioned that without a scientific reward system for curating and providing datasets, it is not beneficial for researchers to publish their data. They present a new way of adding nanopublications of data to their respective articles by using the DOI system to verify the validity of claims.

In 2013 a study by Piwowar and Vision [12] found a significant increase in the likelihood of citations, if the published article includes data sources. They further stated that of 100 datasets used in PubMed publications and published alongside the articles, over the course of five years, 150 reuses have been documented.

In 2016, a year after Rauber et al. published the WGDC Dynamic Data Citation Guidelines [16], Buneman et al. [3] discussed the problems behind Data Citation in the area of databases and further extended these problems in 2020 [2]. The opinions and approaches are looking from a different perspectives on the topic of data citation. Buneman et al. see the problems in reproducing the same subset, as persistent identifiers do not ensure that the query recreates the same subset, as the data is subject to change, and fixity information is necessary to be saved accordingly to verify the correctness. Although this fixity information is saved, it is computationally expensive to provide the data for a query, as the citation would need to link to both the query and the database. Furthermore, they state that it is impracticable and unlikely that data providers can link a citation with the arbitrary complex query [3].

In 2020 [2], Buneman et al.'s work focused mostly on the automation of data citations, which we will not discuss here in detail, but more on their claim on why data citation is not working. Buneman et al. claim that data citation is not working, as many citation analyzers are not able to cite databases, as these are extracting information from textual documents rather than from databases, and that it is necessary to create database summaries that contain information about the database, which will serve as documents. These database summaries seem to follow the same purpose as the guideline R12 of the WGDC Dynamic Data Citation guidelines, by providing users with a machine actionable landing page, which can be mined by citation analyzers.

To solve these problems, there have been made several advancements in research, such as the FAIR principles, which are discussed in the next section, and the creation of the WGDC guidelines for dynamic data citation in Section 2.3.

2.2 FAIRness

The FAIR Principles by Wilkinson et al. [22] have been established to account for the need of reusing scholarly data and code. The principles should guide researchers to make their data findable, accessible, interoperable, and reusable. Therefore it is necessary to discuss the focus of each of these principles properly.

2.2.1 Findability

Findability targets that data is easily discoverable and identifiable, with descriptive and standardized metadata. Each dataset has to be clearly identifiable by a persistent identifier and described, so that it can be searched for by a retrieval system.

Therefore the data needs to be enriched with the following, according to the FAIR principles[22]:

F1 (meta)data are assigned a globally unique and persistent identifier

F2 data are described with rich metadata

F3 metadata clearly and explicitly include the identifier of the data it describes

F4 (meta)data are registered or indexed in a searchable resource

By applying a persistent identifier, as a DOI[1] or a Handle[2] to each dataset, it can be clearly identifiable globally. Data should be described using metadata that is rich enough to enable users to discover and evaluate its relevance and quality. The metadata should include a data description and relevant keywords. Furthermore, data and metadata should be indexed and registered in a searchable source, as a metadata catalog or a data repository using community-based standards.

2.2.2 Accessibility

Accessibility aims to precisely document how to access data and metadata, by providing a clearly structured protocol, with all relevant limitations. Therefore data needs to fulfill the following requirements, according to the FAIR principles[22]:

A1 (meta)data are retrievable by their identifier using a standardized communications protocol

A1.1 the protocol is open, free, and universally implementable

A1.2 the protocol allows for an authentication and authorization procedure, where necessary

A2 metadata are accessible, even when the data are no longer available

By allowing data and metadata to be retrievable by standardized open protocols, it is ensured that everyone can access the data. These protocols can restrict access by verifying the user identity, and allow users to only access limited subsets of the data or only the metadata. If data is no longer available, due to technology migration, expiration, copyright claims, or other reasons, it only keeps the metadata available. Nevertheless, data should be kept available as long as possible. When looking at the ISMN portal, requires user authentication, and the retrieval of data is currently only available after the authentication, while the associated metadata is available without. Furthermore, the platform restricts access to the precise coordinates of the measurement stations to not endanger to vandalism or theft.

[1]https://www.doi.org/
[2]https://www.handle.net/

2.2.3 Interoperability

Interoperability aims to define the proper use and re-use of data after it was retrieved. Retrieved data and metadata need to be in a broadly applicable and usable format and have a clear and unambiguous documentation and metadata.

Therefore, data needs to fulfill the following requirements, according to the FAIR principles[22]:

I1 (meta)data use a formal, accessible, shared, and broadly applicable language for knowledge representation

I2 (meta)data use vocabularies that follow FAIR principles

I3 (meta)data include qualified references to other (meta)data

Provided data needs to be made available in an appropriate format, with a specific documentation, that uses standardized vocabulary. These vocabularies should be part of ontologies to enable the exchange of meanings between different systems and domains. Overall we can say that interoperability enables the sharing and integration of data across different platforms and domains and promotes the advancement of scientific knowledge.

2.2.4 Reusability

Reusability combines and extends the previous three principles, to allow the data to be reused in the future. All the previous requirements are relevant, and additionally also the following ones, according to the FAIR principles[22]:

R1 meta(data) are richly described with a plurality of accurate and relevant attributes

  R1.1 (meta)data are released with a clear and accessible data usage license

  R1.2 (meta)data are associated with detailed provenance

  R1.3 (meta)data meet domain-relevant community standards

Reusability requires that digital resources as data are shared in a standardized, machine-actionable format with defined terms and conditions and an appropriate license.

With all of these guidelines and recommendations to enhance the FAIRness of data. These guidelines go hand in hand with the Dynamic Data Citation Recommendations, which are described in Section 2.3 and look into this problem from the perspective of constantly evolving datasets and how to track these changes best.

2.3 Precise identification of arbitrary subsets of dynamic data

Rauber et al.[16] published 14 guidelines for dynamic data citation, which aim to help researchers and data providers make their evolving datasets reproducible. They provide them with recommendations to allow the recreation of subsets of an evolving dataset, without creating a snapshot for each change in the dataset. These guidelines are presented in Table 2.1.

Category	Recommendation
Preparing the Data and Query Store	R1 - Data Versioning R2 - Timestamping R3 - Query Store Facilities
Persistently Identifying Specific Datasets	R4 - Query Uniqueness R5 - Stable Sorting R6 - Result Set Verification R7 - Query Timestamping R8 - Query PID R9 - Store Query R10 - Automated Citation Texts
Resolving PIDs and Retrieving Data	R11 - Landing Page R12 - Machine Actionability
Modifications to the Data Infrastructure	R13 - Technology Migration R14 - Migration Verification

Table 2.1: The RDA Dynamic Data Citation Guidelines

Each of these recommendations fulfills a different purpose, to allow Data Citation in the long run. In the following, we discuss all the guidelines in detail.

R1 - Data Versioning

"Apply versioning to ensure earlier states of data sets can be retrieved"

Data Versioning aims to version the underlying data, to allow retrieval of all different versions of a dataset. This can be done in databases by adding a validity period for each tuple, also called system-versioned tables [3,4], or by using git[13].

[3]https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables?view=sql-server-ver16

[4]https://mariadb.com/kb/en/system-versioned-tables/

R2 - Timestamping

"Ensure that operations on data are timestamped, i.e. any additions, deletions are marked with a timestamp"

This recommendation goes hand-in-hand with data versioning when using tuple-based versioning approaches. As all operations are timestamped on all affected tuples, to be able to retrieve the correct version. Different approaches for tuple-based versioning have been discussed by Pröll et al. [14].

R3 - Query Store Facilities

"Provide means for storing queries and the associated metadata in order to re-execute them in the future"

To be able to re-execute queries, these queries need to be saved with associated metadata, so it is possible to verify the correctness of the retrieved data, a description of the query and information, and when and by whom the query was executed.

R4 - Query Uniqueness

"Re-write the query to a normalised form so that identical queries can be detected. Compute a checksum of the normalized query to efficiently detect identical queries"

To determine multiple identical queries, these queries need to be transformed into a standardised format, to be able to compare them. This is important to ensure that queries that are semantically identical return the same subset and are also assigned the same PID.

R5 - Stable Sorting

"Ensure that the sorting of the records in the data set is unambiguous and reproducible"

If subsequent processing steps depend on the sequence that data is presented in, it is important to provide the same unambiguous ordering of records, as different orderings can influence the results of analysis and also affect R6 - Result Set Verification.

R6 - Result Set Verification

"Compute fixity information (checksum) of the query result set to enable verification of the correctness of a result upon reexecution"

By computing and storing a checksum of the obtained results, it is possible to verify the correctness of the results. This is most commonly done by using a hashing algorithm and storing this alongside the number of returned records in the query store.

R7 - Query Timestamping

"Assign a timestamp to the query based on the last update to the entire database (or the last update to the selection of data affected by the query or the query execution time). This allows retrieving the data as it existed at the time a user issued a query"

By adding a timestamp to each executed query it is possible to obtain the same subset as before. For tuple-based versioning/timeseries data, this can be done by filtering for active tuples at the original transaction time of the query [14].

R8 - Query PID

"Assign a new PID to the query if either the query is new or if the result set returned from an earlier identical query is different due to changes in the data. Otherwise, return the existing PID"

By assigning a PID, each query is persistently identifiable and can be referenced. In databases, a trivial persistent identifier can be the primary key of the query store, to which a global persistent identifier (e.g. DOI, handle) can be added upon publication.

R9 - Store Query

"Store query and metadata (e.g. PID, original and normalized query, query & result set checksum, timestamp, superset PID, data set description, and other) in the query store"

This recommendation is closely related to the recommendations R3 to R8, as it combines and stores the query execution information in the query store.

R10 - Automated Citation Texts

"Generate citation texts in the format prevalent in the designated community for lowering the barrier for citing the data. Include the PID into the citation text snippet"

To minimize the barrier of citing data, citation texts should be generated from all the associated metadata. This information can be then enriched with a global persistent identifier (e.g. DOI) to include a link to the data.

R11 - Landing Page

"Make the PIDs resolve to a human readable landing page that provides the data (via query re-execution) and metadata, including a link to the superset (PID of the data source) and citation text snippet"

To allow researchers to find and view the metadata, a landing page has to be created, with all associated metadata, with a link to the subset and the superset, as well as a citation text, so that it can be cited.

R12 - Machine Actionability

"Provide an API / machine actionable landing page to access metadata and data via query re-execution."

As more and more metadata and data are automatically indexed by information retrieval systems or stored in (meta)data repositories, it is important to provide a machine actionable interface, to allow the retrieval of metadata or data.

R13 - Technology Migration

"When data is migrated to a new representation (e.g. new database system, a new schema or a completely different technology), migrate also the queries and associated fixity information"

When migrating data to a new schema, a new database system, or a different architecture, it is necessary to also migrate all associated queries and metadata, to allow the reproducibility of all previously executed queries.

R14 - Migration Verification

"Verify successful data and query migration, ensuring that queries can be reexecuted correctly"

By using the same stored checksum as for R6, it is possible to verify the correctness of results, even when the underlying data is changed. The process of hashing is immutable for an already executed query, and it needs to be ensured that all data is migrated to the same format.

2.3.1 Reference Implementations and Frameworks

Over the last few years, the WGDC Dynamic Data Citations have been implemented, and there now exist various different proof-of-concepts solutions and reference implementations. We will give a short overview of some of the different systems available and on the development of the first proof-of-concepts.

In 2013 Pröll and Rauber presented a first analysis on how to approach dynamic data citation in databases [14] and provided a starting point for the Dynamic Data citation guidelines. In their work, they presented three different versioning strategies for timeseries data, called integrated, separated, and hybrid, which are described in detail in Section 4.1 and introduced the concept of a query store, which stores all executed queries. They then further tested their system on a hydroelectric power generation database system. In 2014 Pröll and Rauber applied their presented approach to CSV files for Music Classification [15] and published the recommendations discussed in the previous section, in 2015 [16]. We will evaluate all presented versioning strategies from [14] on the International Soil Moisture Network and pick the best strategy for the use case.

After the publication of these recommendations, several pilot adopters implemented them to their needs and documented their approaches.

The Virtual Atomic and Molecular Data Centre (VAMDC) [8][1] implemented [25][24] the RDA Guidelines for their 38 different database systems, that they are combining in their research portal from 2016 onwards. For versioning, they are using two different mechanisms, which first is a coarse versioning, which tracks which queries could lead to different results, and a fine-grained versioning with a versioning element in their VAMDC-XSAMS standard. The second versioning element indicates which information has changed between two different versions.

The Center for Biomedical Informatics (CBMI) at the Informatics Institute in St. Louis implemented the guidelines in their database, which hosts over 6 million patient records back in 2017 [9]. They used the temporal_tables extension of PostgreSQL 9.5 for their versioning (no longer actively developed and was not updated since the end of 2017), which is a row-based/tuple-based versioning approach, similar to timeseries versioning or temporal versioning. We used this as a starting point for possible implementations of the SQL:2011 standard[5] in PostgreSQL and found that a merge request enabling native system-versioned tables was rejected in 2021[6] with major revisions. To our knowledge, only one extension[7] is currently integrating system-versioned-tables in PostgreSQL to some extent on the latest PostgreSQL versions, but does not allow schema modifications without removing all timestamp information.

In 2021 these adopters and a few more, have been analyzed in a meta-analysis by Rauber et al. [17]. In their work, they describe the different available implementations for the RDA Dynamic Data Citation guidelines. They describe the different versioning approaches for databases, introduced by Pröll and Rauber [14] and file-based solutions. Then they focus on already available implementations, such as the implementations of the VAMDC and the CBMI. For each of the different implementations, they discussed the implemented guidelines and gave a brief overview of the system.

2.4 Schema Evolution in Databases

Schema evolution in databases is a necessity, as requirements, processes or scientific protocols are subject to change and therefore the structure of a database needs to be adapted to fit the new needs. The schema evolution process consists first of a schema modification operation(SMO) through a data definition language (DDL) [8] command. This SMO changes the structure of the database and then a query rewriting operation migrates all queries to the new schema. As manual adaptions of complex database structures are complex and human errors can lead to long migration processes, this process has tried to be automated numerous times. To our knowledge, there exists no

[5]https://www.iso.org/standard/53681.html
[6]https://commitfest.postgresql.org/35/2316/
[7]https://github.com/xocolatl/periods/
[8]https://www.postgresql.org/docs/current/ddl.html

framework for PostgreSQL that supports the automated migration of stored queries of a PostgreSQL database to allow reproducibility according to the WGDC Dynamic Data Citation Guidelines [16].

In 1995, Roddick [18] discussed the terms schema versioning and schema evolution and their effects on timeseries data and issues when applying it to a database. Schema evolution is the modification of a database schema without the loss of information, while schema versioning is when a database system allows the access of all data retrospectively and prospectively through interfaces. Therefore, the main difference between these two terms is, that schema versioning has the possibility for users to identify stable points and to query past schema definitions. In his work, he mentions that the combination of timeseries data and schema versioning led to three different timelines, with valid-time (real-world time), transaction-time (time data was introduced in the database), and schema-time (time the schema was introduced in the database). He further mentions that it is necessary to query data in all three timelines, to have the maximum expressiveness. This expressiveness is vital for reproducibility, as the results of re-executed queries need to return the exact same subset as originally obtained.

Curino et al. created the PRISM [6] framework, and extended it to the PRISM++ framework [5] to automate the migration of databases, by rewriting queries and updates. Their updated framework supports rewriting of structural schema changes and integrity constraint evolution. This is done by complementing each SMO with a set of integrity constraints, constraining the use of SMOs, and rewriting the affected queries. They further evaluated schema changes on different databases and wikis. The scientific CERN-DQ2 database had over the course of 1.3 years, 51 schema changes, while the Ensembl database had 412 changes over 9.8 years. We are picking up the idea of automatically rewriting queries to allow the adaption of the database schema, and combining this with the WGDC Dynamic Data Citation Guidelines [16].

In 2020 Schuler and Kesselman [19] presented their CHiSEL framework to simplify database evolutions. According to them, these are the most complicated part of database management. They created a Python library, which is closely following relational algebra methodics to minimize the necessary schema modification operations. Based on their evaluation of different genom databases, their framework uses a quarter of the necessary SQL operations, by minimizing complex schema beforehand with relational algebra.

In 2018 Säuerl [20] discussed how schema modifications can be implemented on databases that are following the WGDC guidelines on dynamic data citation, and discussed the importance of integration as database schemas are regularly evolving. We are building upon the discussed schema modification strategies, and are using them in PostgreSQL to allow schema modifications, while ensuring reproducibility.

In 2022 Xin et al. [23] presented a solution to track and query over timeseries data with schema evolutions in Apache IoTDB, with a more data-centric approach. Due to language barriers, it was impossible to analyze the paper in detail.

CHAPTER 3

International Soil Moisture Network

The International Soil Moisture Network (ISMN) [7], is a community based research portal, funded by the European Space Agency to provide researchers and data providers with a centralised data hosting facility. Its main focus is to gather heterogeneous soil moisture data from different networks, distributed all over the world, and harmonize them into a centralized database. A network refers to a data provider that operates stations with different in-situ soil moisture sensors.

They gather data, which runs through various processes, such as harmonizing measurement units and sampling rates, while also applying quality control flags to each tuple in the database. Over the years they have grown to the most commonly used in-situ soil moisture reference database worldwide, with over 1600 active users, over 1000 publications, and a daily growing and evolving database with over 1 billion tuples.

3.1 Database

The database is the heart of the ISMN portal, where all data and metadata are stored in a community agreed standardized format.

The ISMN tables are structured in different categories, based on the information they store:

- Network/Station related metadata
- Data related metadata
- Dynamic and static data

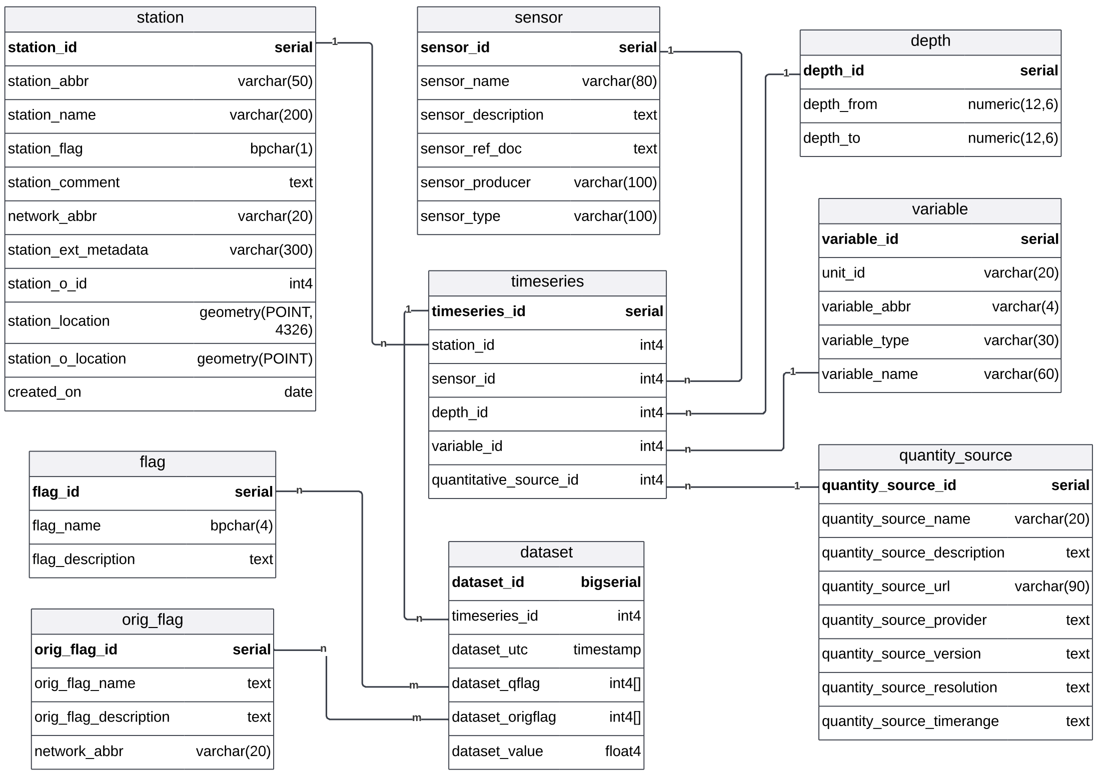


Figure 3.1: Subset of the ISMN database schema, which is most relevant for data exploration and the download of arbitrary subsets of the database

The network and station related metadata remains mainly static and only changes when new organisations, networks, or quality information are added or updated. This data consists of information about the different networks, as the country in which the sensor-network operates in, the encoding and language of the data, quality information that is sent with all measurements, and means of contact if problems arise.

The data related metadata stores information about stations and their locations, used sensor with their depths, measurement units, and quantitative source information. These data changes whenever a new sensor is added, a sensor is replaced with a different sensor or other information about the stations and sensors change.

The data is then stored in the dataset table, with two ancillary tables, which are keeping additional information. This data is updated daily by near-realtime networks (NRT), which are directly connected to the ISMN system via an API and manually when not connected networks provide datasets or quality information.

In Figure 3.1 we see the part of the ISMN schema, which is most important for data exploration and downloads. In the top left we see the *station* table, which stores information about all stations for each network. The two tables in the top right *sensor* and *depth* keep information about the used sensor and in which depths it is measuring.

The *variable* and *quantity_source* tables are storing information about the measurements which are conducted, as the units of measurement or the type of measurement which is conducted.

In the *timeseries* table all this information are intersecting and is then linked to measurements in the *dataset* table so that each measurement has all associated metadata information assigned to it. The *dataset* table also contains two arrays of quality flags, which can be resolved by joining them with the corresponding flag tables. The *flag* table has quality information, which are the same for all different stations and networks, as they are following the central ISMN standard, while the *orig_flag* table contains information for all the different flags assigned by data providers.

The table sizes of the data namespace are heavily skewed, with the displayed tables taking over the majority of storage needs, while other tables only take a fraction of the storage. Based on the analysis of a dump from March 2022, of the 28 tables only the dataset is the largest, with 123 GB in data storage and 65 GB in keys, and 8.5 GB in indices, totaling over 196.5 GB of total storage for this one table alone. The other 27 tables accumulate roughly 30 MB of storage, and PostgreSQL internal tables and different namespaces account for the remaining 70 GB of storage.

Historic storage consumptions are only sparsely available, as the historic storage consumption was measured after each project phase concluded, with the exact means of measurement unknown and also if the data was compressed before. At the end of 2012, the storage consumption was measured at 4 GB and increased to 51 GB in 2016 and via various cleanups to 318 GB in December 2022. These increases are heavily affected by the number and the size of the added networks, and were approximately 15 GB for the year 2022, delivered by 76 different networks[1].

3.1.1 Schema Adaptions

After the creation of the initial database schema between 2010 and 2012, the ISMN made several changes to their database schema over time and plans additional changes in the near future to adapt it to their new requirements after the portation of the platform to the BfG[2].

During the SMOS CCN3 project, which ran from June 2014 to March 2016 the International Soil Moisture network performed several schema modifications, to enrich the functionality and design of the ISMN portal.

The table *network_publications* was added, which keeps information about publications of the different data providers, to allow the acknowledgement and referencing of the data providers.

[1]https://ismn.earth/en/networks/

[2]https://www.bafg.de/DE/Home/homepage_node.html

There also have been three columns added to the *image* table to allow the storage of thumbnails with their associated metadata (height and width) for organisations and stations. The thumbnail itself is stored in the database as a base64 encoded byte array, while the other two columns are integers.

Additionally, two columns were dropped from the *variable* table, which modified the graphical dataviewer, by providing minimum and maximum values for each station.

Furthermore, as the ISMN moved to the Bundesamt für Gewässerkunde in January 2023, further changes were discussed to reduce maintenance overhead and querying times, while also adding new functionalities. These changes include the restructuring of several metadata tables, to reduce the necessary amount of joins to extract relevant metadata and the addition of new quality information for each timeseries.

3.2 Data Integration Workflow

As the data providers are using different measurement units and do not deliver the data over the same protocols or in the same format, it is necessary to preprocess and harmonize the data before storing them in the ISMN database.

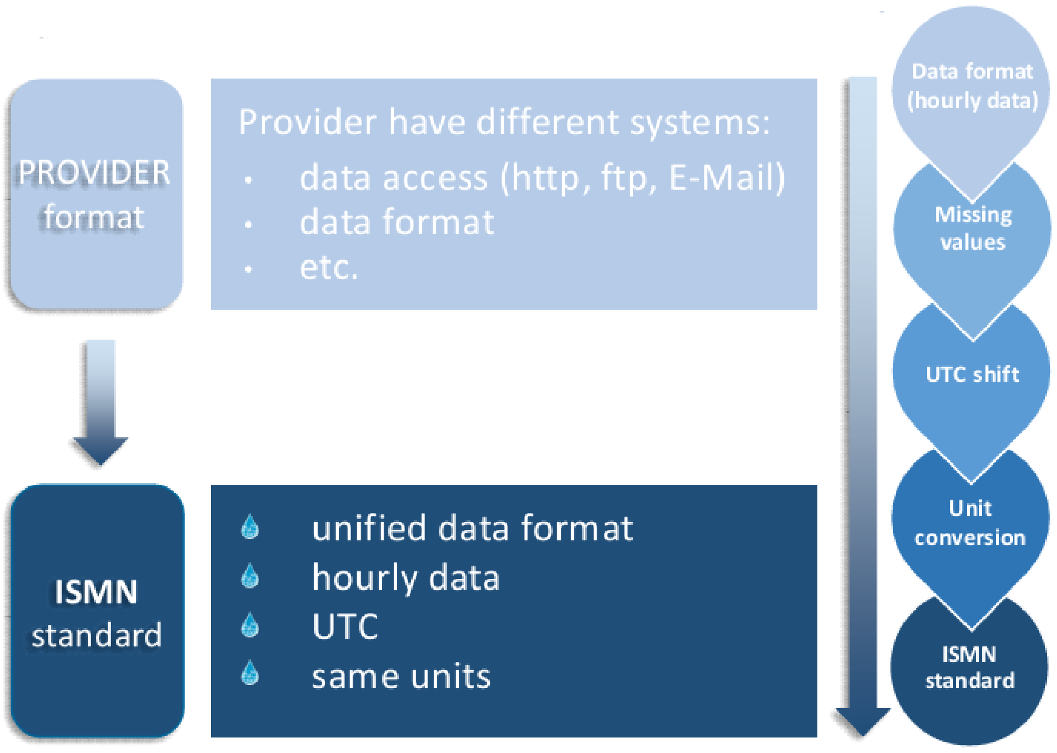


Figure 3.2: Harmonization process of the International Soil Moisture Network
Source: `https://ismn.earth/en/data/harmonization/`

In Figure 3.2 we can see the harmonization process to transform data from a provider format to the ISMN standard format. Therefore, the format must be transformed for each different provider to the ISMN format in hourly format. Afterwards, all missing values are handled and the timestamps are changed to UTC+0, so that only one timezone is used. Then the different soil moisture definitions (e.g. volumetric, water depth, mass, Soil Moisture Index) are transformed to fractional volumetric soil moisture, to have them comparable to each other. In the last step, this data is then inserted in the database[3].

After this process is concluded, quality control flags are updated twice, during separate update routines. The first process is detecting dubious soil moisture observations by geophysical dynamic range verification and marks the data with the corresponding quality flags[4]. In the second process, which runs when the associated Global Land Data Assimilation System data (GLDAS)[5] data is available, geophysical consistency methods and spectrum-based methods are used to detect conditions in the soil moisture observations.

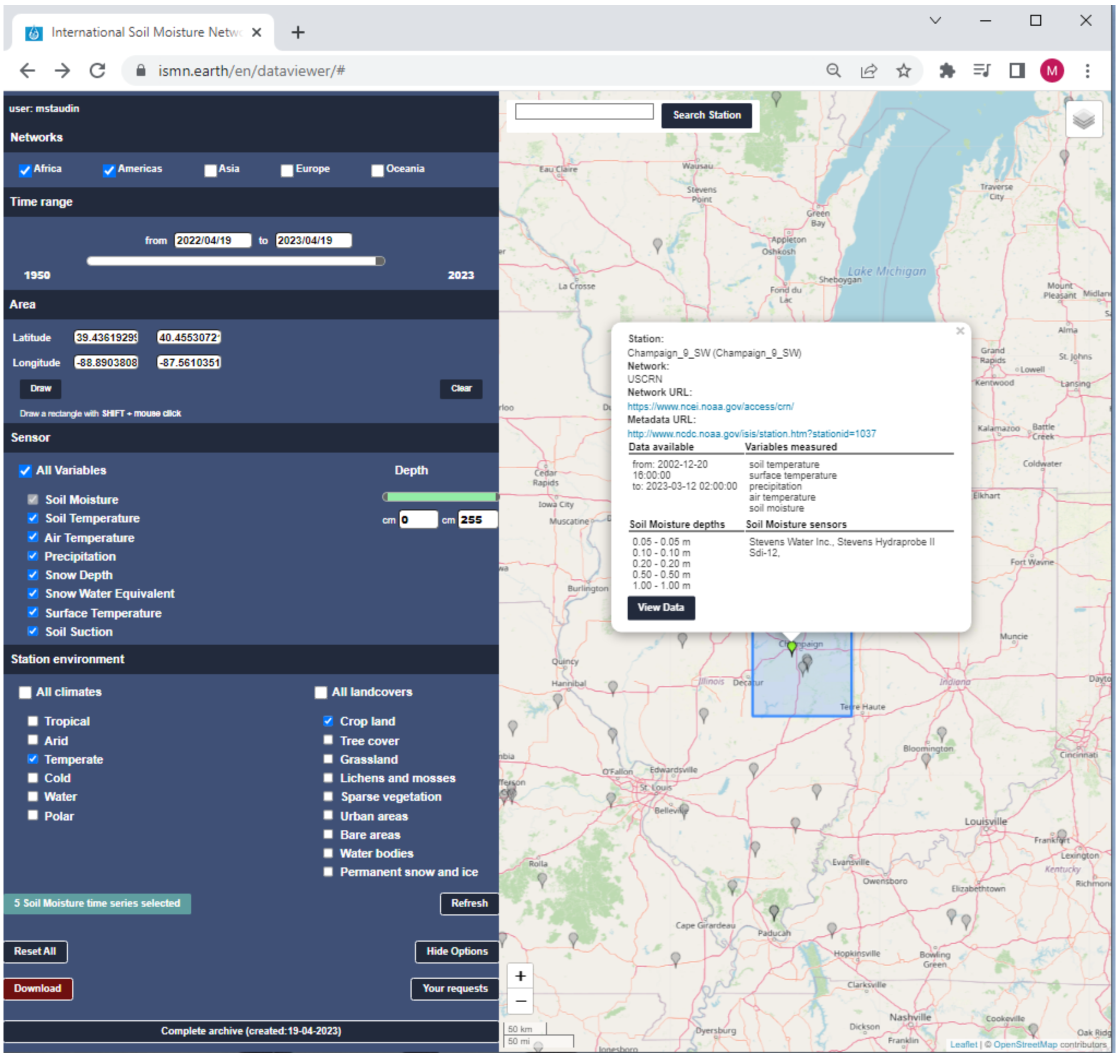
Therefore, each tuple in the *dataset* table is with the adapted quality information twice. One of the updates occurs right after the tuples have been inserted, and the second update can happen several months after the data was inserted.

3.3 Download Workflow

The data is then available for exploration in the dataviewer[6], where data and metadata can be displayed. After a successful login, each user can download data by specifying the wanted area/continent, type of landcover, climate zone, and the type of sensors with the wanted depth to be downloaded.

In Figure 3.3 we see the dataviewer of the International Soil Moisture Network. On the map on the right, we can see that we selected an area in central Illinois, which includes the measurement station *Champaign_9_SW*. For this station the metadata can be seen, including the network this sensor belongs to, data availability, and information about the available sensors and variables measured. On the left, we can see the filter parameters for the download request, which we customized to include all data available which was selected in temperate climate on crop land. When clicking the download button, a REST request is sent to the backend which includes all selected parameters.

[3]https://ismn.earth/en/data/harmonization/
[4]https://ismn.earth/en/data/flag-overview/
[5]https://ldas.gsfc.nasa.gov/gldas
[6]https://ismn.earth/en/dataviewer/


International Soil Moisture Netw
ismn.earth/en/dataviewer/#

user: mstaudin

Networks
✓ Africa ✓ Americas ☐ Asia ☐ Europe ☐ Oceania

Time range
from 2022/04/19 to 2023/04/19
1950 2023

Area
Latitude 39.43619299 40.4553072
Longitude -88.8903808 -87.5610351
Draw Clear
Draw a rectangle with SHIFT + mouse click

Sensor
✓ All Variables — Depth
✓ Soil Moisture — cm 0 cm 255
✓ Soil Temperature
✓ Air Temperature
✓ Precipitation
✓ Snow Depth
✓ Snow Water Equivalent
✓ Surface Temperature
✓ Soil Suction

Station environment
☐ All climates
☐ Tropical
☐ Arid
✓ Temperate
☐ Cold
☐ Water
☐ Polar

☐ All landcovers
✓ Crop land
☐ Tree cover
☐ Grassland
☐ Lichens and mosses
☐ Sparse vegetation
☐ Urban areas
☐ Bare areas
☐ Water bodies
☐ Permanent snow and ice

5 Soil Moisture time series selected
Refresh
Reset All
Hide Options
Download
Your requests
Complete archive (created:19-04-2023)

Search Station

Station:
Champaign_9_SW (Champaign_9_SW)
Network:
USCRN
Network URL:
https://www.ncei.noaa.gov/access/crn/
Metadata URL:
http://www.ncdc.noaa.gov/isis/station.htm?stationid=1037

Data available	Variables measured
from: 2002-12-20 16:00:00 to: 2023-03-12 02:00:00	soil temperature surface temperature precipitation air temperature soil moisture

Soil Moisture depths	Soil Moisture sensors
0.05 - 0.05 m 0.10 - 0.10 m 0.20 - 0.20 m 0.50 - 0.50 m 1.00 - 1.00 m	Stevens Water Inc., Stevens Hydraprobe II Sdi-12,

View Data

Figure 3.3: Webviewer of the International Soil Moisture Network
Source: `https://ismn.earth/en/dataviewer/`

```
https://ismn.earth/en/dataviewer/download_selection/
?start=2022/04/19&end=2023/04/19
&variables=Soil Moisture, Soil Temperature, Air Temperature,
 ↪  Precipitation, Snow Depth, Snow Water Equivalent,
 ↪  Surface Temperature, Soil Suction
&climates=Temperate
&landcovers=Crop land
&depth_from=0&depth_to=255
&fill_nan=0&g_flag_only=0
&swlat=39.436192999314095&swlng=-88.89038085937501
&nelat=40.455307212131494&nelng=-87.56103515625
```

Listing 1: REST download request, which is sent to the backend

In Listing 1 we can see the download request created by the screenshot seen in Figure 3.3. From line 2 to 7 we see the selected parameters for the download. In lines 8 and 9 we see the south-west and north-east coordinates of the drawn rectangle. In the body of the request there is further information, which includes all networks as filter criteria.

This request is then processed in the backend and transformed into SQL queries, in the following order:

1. Query stations per network
2. Query metadata per station
3. Query data per timeseries

For the first and third querytype, there exist two different variations, dependent if coordinates are used or data gaps should be filled by null values respectively.

```
1  SELECT station_id,
2         station_name,
3         network_abbr,
4         Round(Cast(St_x(station_location) AS NUMERIC),5),
5         Round(Cast(St_x(station_location) AS NUMERIC),5),
6         St_z(station_location)
7  FROM   station
8  WHERE  network_abbr IN {}
9  AND    st_y(station_location) >= {}
10 AND    st_y(station_location) <= {}
11 AND    st_x(station_location) >= {}
12 AND    st_x(station_location) <= {}
```

Listing 2: SELECT query to obtain all different stations in the boundaries of the drawn rectangular

In Listing 2 we can see the SQL query to obtain all different stations, which are in the defined boundary box. From line 1 to 6 the id, name, network and the location are selected, for all networks, which are passed as arguments in line 8 and the coordinates are handed over in the lines 9 to 12. This query is executed up to 76 times, as currently there exist 76 networks in the ISMN database.

```
1  SELECT timeseries_id,
2         variable.variable_abbr,
3         variable.variable_id,
4         depth.depth_id,
5         Round(depth.depth_from, 2),
6         Round(depth.depth_to,2),
```

```sql
7           sensor.sensor_id,
8           sensor.sensor_name
9  FROM        timeseries
10 INNER JOIN variable
11 ON          timeseries.variable_id = variable.variable_id
12 INNER JOIN depth
13 ON          timeseries.depth_id = depth.depth_id
14 INNER JOIN sensor
15 ON          timeseries.sensor_id = sensor.sensor_id
16 WHERE       station_id = {}
17 AND         variable.variable_type <> 'static'
18 AND         quantity_source_id = 0
```

Listing 3: SELECT query to obtain all metadata per station

Then for each station returned by the first query, the query displayed in Listing 3 is executed, which can lead up to 3000 executed queries. From line 1 to 8 the metadata for a specific station is selected, which is specified in line 16. Further filtering is performed in line 17 and 18.

```sql
SELECT to_char(dataset.dataset_utc, 'YYYY-MM-DD HH24:MI'),
       dataset.dataset_value,
       (SELECT COALESCE(String_agg(flag.flag_name, ','),
       ↪   'M') AS dataset_qflag
        FROM   Unnest(dataset_qflag) dqflag
               LEFT JOIN flag
                      ON flag.flag_id = dqflag),
       (SELECT COALESCE(String_agg(orig_flag.orig_flag_name,
       ↪   ','), 'M') AS
               dataset_origflag
        FROM   Unnest(dataset_origflag) dqflag
               LEFT JOIN orig_flag
                      ON orig_flag.orig_flag_id = dqflag),
       {}     AS depth_from,
       {}       AS depth_to,
       {} AS sensor_name
FROM   dataset
WHERE  timeseries_id = {}
       AND dataset_utc >= {}
       AND dataset_utc <= {}
ORDER  BY dataset.dataset_utc
```

Listing 4: SELECT query to obtain all data per timeseries

In Listing 4 we see the final data query, which is executed up to 30,000 times with each different timeseries_id. The ISMN quality flags are extracted from their array and joined with the *flag* table from line 3 to 6 and similar for the provider flags from line 7 to 11. The PostgreSQL unnest function expands an array to a set of rows. In line 12 to 14 parameters, obtained from the second query are passed as columns for this query and from line 16 to 18 the query is filtered based on the *timeseries_id* and the time of measurement *dataset_utc*.

The results of each of these queries is then written in one of two standardized data formats[7] and then provided to the user in a ZIP file with all other query results.

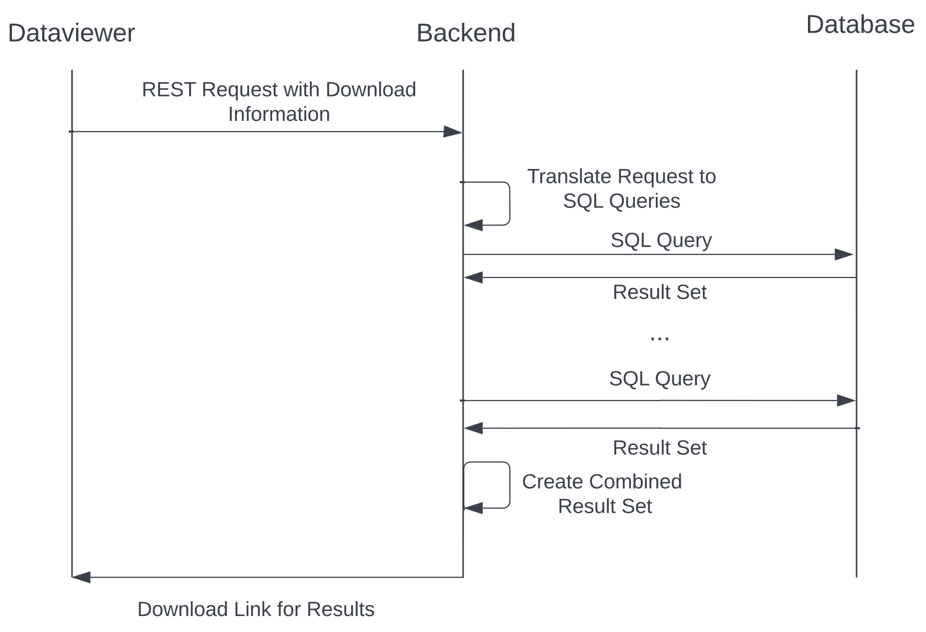

Dataviewer
Backend
Database
REST Request with Download Information
Translate Request to SQL Queries
SQL Query
Result Set
...
SQL Query
Result Set
Create Combined Result Set
Download Link for Results

Figure 3.4: The process of a user triggered download

In Figure 3.4 the whole download process is visualized for each component. The user starts via the online data viewer and sends a REST request to the backend, containing all download information. This request is then translated to SQL queries, which are then sent to the database and the result sets are gathered and combined to a compressed folder. This folder is then available to the user via a download link.

[7] `https://ismn.earth/en/data/formats/`

CHAPTER 4

Conceptual Design

To design a dynamic data citation framework for PostgreSQL, which is adapted to the use case of the International Soil Moisture Network [7], and also can be generalized, we first have to understand and analyze the needs of such a system. We used the Data Citation Guidelines [16] as a starting point and discussed all necessary features, such a system should have. In the following, we will present the Data Citation Guidelines and how they should be implemented in the system and how different approaches can have different impacts on them.

4.1 Data Versioning

Following the guidelines R1 - Data Versioning and R2 - Timestamping, it is mandatory to track all changes in a database to allow to re-create the data, which was present in the database, at the time a query was executed. For databases, the most common type of versioning is tuple-based versioning, which is preferable over change-logs or data freezes, as not the whole database needs to be rolled-back to a specific timestamp. This was introduced in the SQL standard SQL:2011[1] and has since been implemented in many different databases[2,3], but not yet in PostgreSQL. There are several extensions that implement parts of the SQL standard[4,5], but advancements to include temporal tables in PostgreSQL natively failed in 2021[6], as major revisions were necessary.

Tuple-based versioning adds a validity period, consisting of two timestamps to each tuple, to track in which time period a specific tuple was active in the database.

[1]https://www.iso.org/standard/53681.html

[2]https://mariadb.com/kb/en/system-versioned-tables/

[3]https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables

[4]https://github.com/xocolatl/periods

[5]https://github.com/arkhipov/temporal_tables

[6]https://commitfest.postgresql.org/35/2316/

Tuple-based versioning can be applied in three different ways:

- Integrated Versioning
- Separated Versioning
- Hybrid Versioning

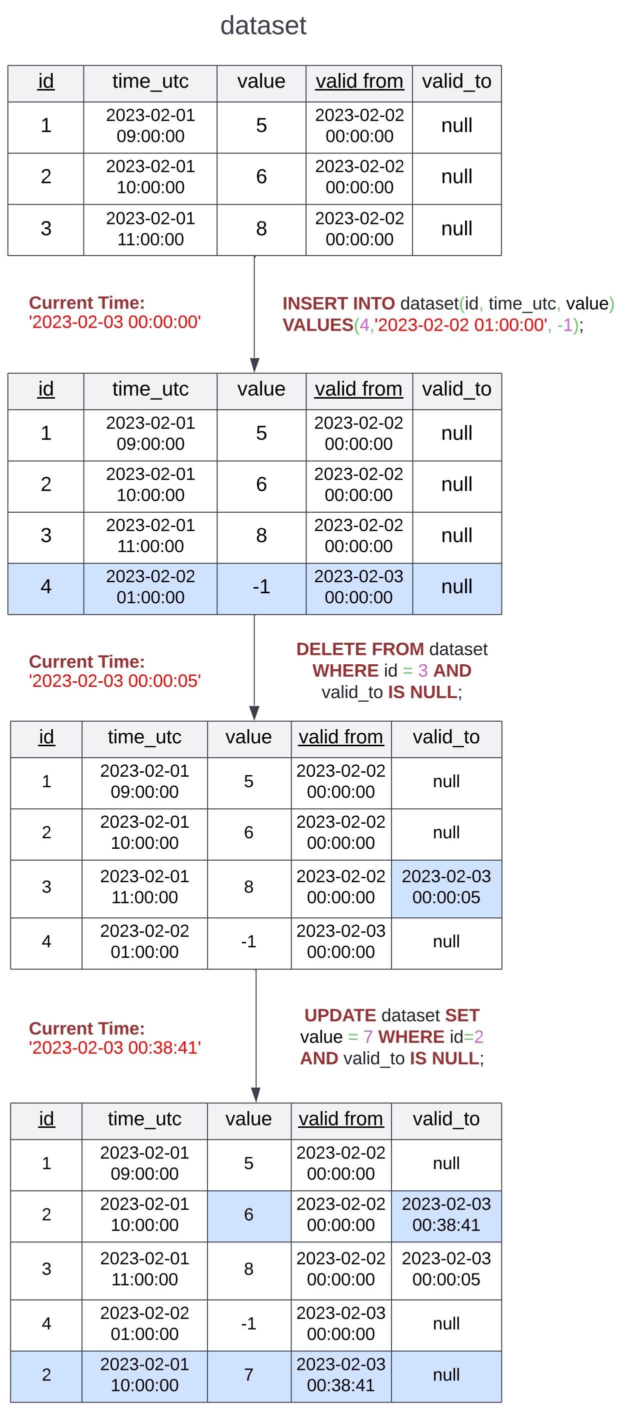
Each of these tuple-based versioning approaches, has its own advantages and disadvantages, which include different storage-overhead, different query-execution and re-execution times and also a different complexity for query rewriting. Versioning is a main concern for the International Soil Moisture Network, as versioning and timestamping all operations in a database creates a large storage- and performance-overhead, as their current setup is updating each datapoint twice. Based on the measurement on a synthetic database [20] the minimal overhead for all tuple-based versioning approaches is between 14.8% (Integrated/Hybrid) and 114.7% (Separated). As this certainly differs between different databases, we will re-evaluate the performance of database operations.

In the following, we will describe the different versioning approaches and compare them, and talk about methods to ensure concurrency in databases when using these versioning strategies.

4.1.1 Integrated Versioning

In the integrated versioning approach, two additional timestamp columns are added to the original table, one column that marks the time the tuple was first introduced (valid_from) into the table, and one column that marks, when the tuple was removed from the table(valid_to). When both timestamps are set, the tuple is no longer active, and we call it a historized tuple. All historized tuples are directly stored in the same table, which also holds the most recent version. The INSERT command does not need to be altered, but the UPDATE and DELETE commands need to behave differently. The UPDATE command needs to deactivate old tuples by setting the valid_to timestamp and inserting the updated tuples into the table. The DELETE command is not allowed to delete the tuple, but instead sets the valid_to timestamp and deactivates the tuple. It is also necessary to adapt the primary key of the original table to also include the valid_from column in the primary key, as the original primary keys are duplicates in the case of an update. For constraints, it is also necessary to evaluate, how different constraints affect the versioning, as uniqueness constraints need to be adapted to include the valid_from timestamp.

To retrieve any different version of this table, it is necessary that it is always filtered based on the valid_from and valid_to timestamps, as otherwise multiple versions will collide and the retrieved data is not actually representing the active data in the table. For the most recent version, it is sufficient to filter only based on the *valid_to* column.



id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	null

Current Time:
'2023-02-03 00:00:00'

```
INSERT INTO dataset(id, time_utc, value)
VALUES(4,'2023-02-02 01:00:00', -1);
```

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	null
4	2023-02-02 01:00:00	-1	2023-02-03 00:00:00	null

Current Time:
'2023-02-03 00:00:05'

```
DELETE FROM dataset
WHERE id = 3 AND
valid_to IS NULL;
```

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	2023-02-03 00:00:05
4	2023-02-02 01:00:00	-1	2023-02-03 00:00:00	null

Current Time:
'2023-02-03 00:38:41'

```
UPDATE dataset SET
value = 7 WHERE id=2
AND valid_to IS NULL;
```

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	2023-02-03 00:38:41
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	2023-02-03 00:00:05
4	2023-02-02 01:00:00	-1	2023-02-03 00:00:00	null
2	2023-02-01 10:00:00	7	2023-02-03 00:38:41	null

Figure 4.1: Example of Integrated Versioning on the dataset table with all different CRUD operations, with changes marked in blue

In Figure 4.1 we can see the dataset table of the ISMN network with integrated versioning applied, it consists of the columns id, *time_utc*, value *valid_from* and *valid_to*. We can see that in the first table, three tuples are active, as all of them have an unset *valid_to* timestamp. After an insert, the *valid_from* timestamp is set automatically, and the new table now has four active tuples. After a delete on $id = 3$, the *valid_to* value of the corresponding tuple is updated with the operation time and therefore the tuple is marked as deleted. After an update on $id = 2$, we can see that the original tuple is marked as deleted, as the *valid_to* timestamp is set and a new row, with the updated value, is inserted instead.

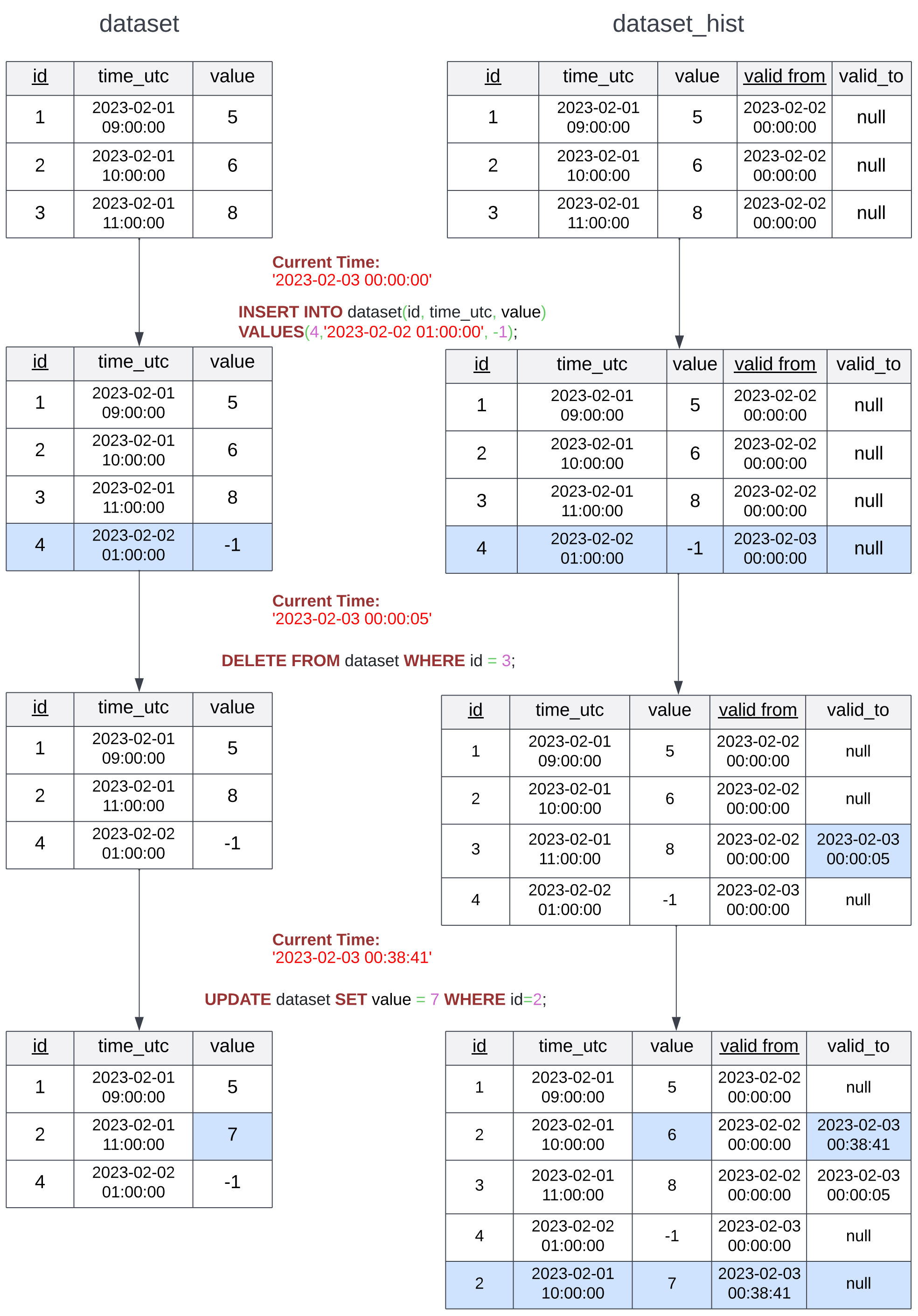
With this approach, the storage of the table will increase by 32 bytes (16 bytes per timestamp, according to PostgreSQL documentation [7]) per tuple, and the query execution and re-execution times will increase, as the data always needs to be filtered and all the data stays in the same table. As this approach keeps the historized versions of the table in the same table as the actual table, query rewriting is only necessary to enrich the initial query to include system versioning timestamps and all used columns, to not include the additionally added columns.

4.1.2 Separated Versioning

In the separated versioning approach, the original table is not altered, but a history table is created, which holds a copy of the data of the original table, with all historized data. Therefore the history table consists of all the original columns of the table and the two timestamp columns. In the case of CRUD operations on the original table, all operations are similarly performed on the history table, with the behaviour described in the integrated approach before. As the original table is not altered, constraints do not need to be adapted. Furthermore, as no data is inserted in the history table via direct CRUD operations, we also do not need to copy these constraints to the history table, as they will be enforced via the original table. In comparison to the integrated approach, the most recent version can be retrieved by simply querying the original table without any constraints, although for allowing reproducibility, the timestamp of execution needs to be saved.

In Figure 4.2 we can see the dataset table of the ISMN network with separated versioning applied, it consists of the original table on the left, and the history table on the right. We can see that in the first row, both tables contain the same data, but the history table has a set *valid_from* column with the timestamp, when the data was included in the database and a *valid_to* column, which is set, when the data is deleted. When performing an insert, as can be seen in the second row of tables, the data is automatically copied to the history table, and the validity period is started. On a delete operation on $id = 3$, the tuple is dropped from the original table, and the corresponding tuple in the history table is marked as deleted. On an update operation on $id = 2$, the value is normally updated

[7]https://www.postgresql.org/docs/current/datatype-datetime.html



dataset

id	time_utc	value
1	2023-02-01 09:00:00	5
2	2023-02-01 10:00:00	6
3	2023-02-01 11:00:00	8

dataset_hist

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	null

Current Time:
'2023-02-03 00:00:00'

```
INSERT INTO dataset(id, time_utc, value)
VALUES(4,'2023-02-02 01:00:00', -1);
```

id	time_utc	value
1	2023-02-01 09:00:00	5
2	2023-02-01 10:00:00	6
3	2023-02-01 11:00:00	8
4	2023-02-02 01:00:00	-1

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	null
4	2023-02-02 01:00:00	-1	2023-02-03 00:00:00	null

Current Time:
'2023-02-03 00:00:05'

```
DELETE FROM dataset WHERE id = 3;
```

id	time_utc	value
1	2023-02-01 09:00:00	5
2	2023-02-01 11:00:00	8
4	2023-02-02 01:00:00	-1

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	2023-02-03 00:00:05
4	2023-02-02 01:00:00	-1	2023-02-03 00:00:00	null

Current Time:
'2023-02-03 00:38:41'

```
UPDATE dataset SET value = 7 WHERE id=2;
```

id	time_utc	value
1	2023-02-01 09:00:00	5
2	2023-02-01 11:00:00	7
4	2023-02-02 01:00:00	-1

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	2023-02-03 00:38:41
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	2023-02-03 00:00:05
4	2023-02-02 01:00:00	-1	2023-02-03 00:00:00	null
2	2023-02-01 10:00:00	7	2023-02-03 00:38:41	null

Figure 4.2: Example of Separated Versioning on the dataset table with all different CRUD operations, with changes marked in blue

in the original table, as can be seen in the fourth row of tables, while it is marked as deleted, and inserted again in the history table.

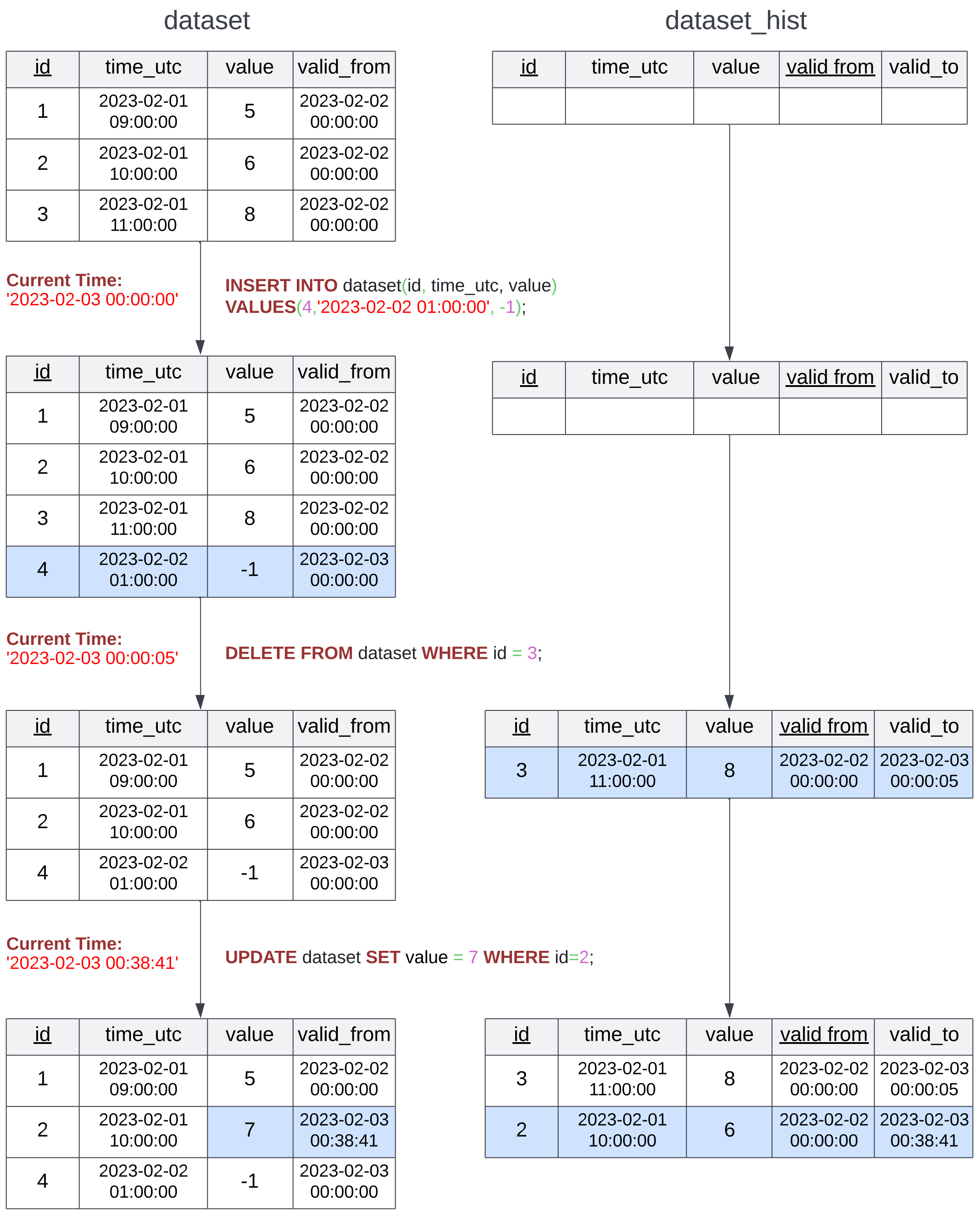
With this approach, the storage will increase the most, as the whole table needs to be saved with two additional timestamps (32 bytes per tuple), which leads to an increase of over 100%. The query execution time will stay the same as the original table remains unchanged, and the re-execution time will be similar to the integrated approach. Minor query rewriting is necessary, as all queries executed on the original table need to be rewritten to be re-executed on the history table.

4.1.3 Hybrid Versioning

In the hybrid versioning approach, only one timestamp is added to the original table, which traces when a tuple was added to this table and a history table with two timestamps is additionally created, but no data is copied into this table. Only if a tuple is deleted it is then copied to the history table, and the *valid_to* timestamp is set. Constraints on the original table need to be checked, but should be able to remain unchanged as no duplicate keys are allowed, as such tuples should be in the history table only.

In Figure 4.3 we can see the dataset table of the ISMN network with hybrid versioning applied, it consists of the original table on the left, and the history table on the right. We can see that the original table contains three active tuples at the start. After the INSERT of a fourth row, the original table contains now four rows, while the history table remains empty. After a delete on the tuple with $id = 3$ from the original table, the row is transferred to the history table and the *valid_to* timestamp is set. After an Update on the tuple with $id = 2$ the tuple in the original table is updated, while the old tuple is transferred to the history table, similar to a delete and insert.

In comparison to the integrated approach, the most recent version can be retrieved by simply querying the original table without any constraints. Also, the storage does not increase as much as in the separated approach, and as the most recent data can be retrieved, the query execution times should not increase. For saving and re-executing queries this approach is the one with the most overhead. As each re-executed query needs to be executed on both the history, and the original table and then needs to be combined, which results in an overhead when being compared to the other two approaches. Also does the query need to be rewritten so that it queries both tables and combines both result sets. Säuerl[20] analyzed all three versioning strategies, and concluded that the hybrid approach is not well fitted for real-world scenarios due to the slow re-execution performance. We will consider this approach, as a different schema, different join strategies and different user behaviours can change the viability of this strategy.



id	time_utc	value	valid_from
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00

id	time_utc	value	valid from	valid_to

Current Time:
'2023-02-03 00:00:00'

```
INSERT INTO dataset(id, time_utc, value)
VALUES(4,'2023-02-02 01:00:00', -1);
```

id	time_utc	value	valid_from
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00
4	2023-02-02 01:00:00	-1	2023-02-03 00:00:00

id	time_utc	value	valid from	valid_to

Current Time:
'2023-02-03 00:00:05'

```
DELETE FROM dataset WHERE id = 3;
```

id	time_utc	value	valid_from
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00
4	2023-02-02 01:00:00	-1	2023-02-03 00:00:00

id	time_utc	value	valid from	valid_to
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	2023-02-03 00:00:05

Current Time:
'2023-02-03 00:38:41'

```
UPDATE dataset SET value = 7 WHERE id=2;
```

id	time_utc	value	valid_from
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00
2	2023-02-01 10:00:00	7	2023-02-03 00:38:41
4	2023-02-02 01:00:00	-1	2023-02-03 00:00:00

id	time_utc	value	valid from	valid_to
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	2023-02-03 00:00:05
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	2023-02-03 00:38:41

Figure 4.3: Example of Hybrid Versioning on the dataset table with all different CRUD operations, with changes marked in blue

4.1.4 Comparison

All of the versioning approaches are different in their behaviour, therefore it is necessary to highlight the most important differences between the three versioning approaches on a theoretical level.
In Table 4.1 we can see the differences between our different approaches. In terms of versioning, do the integrated and the separated approach, each version one table, while the hybrid approach versions the original and the history table. As the separated and the hybrid approach always keep the most recent version in the original table, it is not necessary that in these two approaches, any filtering is necessary to obtain the most recent version. On the other hand in the integrated approach, it is necessary to always filter for the correct version, which needs to specify the correct *valid_from* and *valid_to* timestamp information. This clearly has a performance impact, as each query needs to filter, if a tuple is active at the current time, which means that the *valid_from* timestamp is prior to the query execution timestamp and the *valid_to* timestamp is null or posterior to the query execution timestamp.

From a storage point of view, we can see that the hybrid approach needs, in the best case (no updates or deletes), only one timestamp column (16 bytes), while the integrated approach always has two timestamp columns (2x 16 byte). As the separated approach copies all tuples into a separated history table and adds two timestamp columns to the history table, the overhead is the biggest. When looking at the re-execution of queries, the integrated approach allows to re-execute the original queries, without any rewriting, as all timestamp information is already present. For the separated and the hybrid approach, it is necessary to rewrite the query. For the separated approach, this is done in the same manner as for the integrated approach, by adding three additional where clauses per queried table. For the hybrid approach, we need to combine the results of the original table, with one added timestamp information and the results of the history table, as the information is divided between these two tables. The most important metric is the retrieval time of queries, which differs significantly between the different versioning techniques. The integrated versioning increases the query execution time, as all different versions of a tuple are obtained and need to be filtered for execution and re-execution. The separated and hybrid approaches have a similar execution time for the query-execution as an unversioned database, as they keep the most recent version in a dedicated table. The query re-execution in the separated versioning is similar to the query re-execution in the integrated versioning, as all different versions are kept in one table. For the hybrid versioning re-execution it heavily depends on the type of query, as the queries can be run parallelized, but for each join it is necessary to join with the original and the history table, which results in increased re-execution time.

So we can conclude, that the integrated approach is the easiest to implement, as we do not need additional tables, and the same query can be used for execution and re-execution. From a performance perspective, this approach is not as efficient in the original retrieval of data, as timestamp information need to be added to each query. From a storage and performance perspective, the hybrid approach has benefits, as it does not store every

	Integrated	**Separated**	**Hybrid**
Versioned Table	original table	history table	both tables
Performance Impact	3 extra constraints	Query remains unchanged	Query remains unchanged
Minimal Storage Increase	32 byte per tuple	32 byte + tuple per tuple	16 byte per tuple
Query Re-Execution	Re-Execution with same query	Re-Execution on history table with adapted query	Re-Execution on original and history table with adapted query

Table 4.1: Comparison of the different versioning approaches

tuple twice, but keeps the most recent version for each table. The query re-execution for this approach is the most complicated, as the results from querying the original and the history table need to be combined. The separated approach has the same performance as the hybrid table on standard retrieval, but can retrieve the results from a history table that keeps all tuples. This approach saves each tuple twice, but separates the execution and the re-execution tables.

4.1.5 Database Concurrency Control

Database concurrency control is an essential feature in database management systems, and facilitates the concurrent access of multiple users to the same data. For systems as the International Soil Moisture Network, which aim to allow the reproducibility of executed queries, data integrity and consistency are necessary requirements. Therefore, it is important to ensure that the underlying data is not evolving, while the data is queried, so that no concurrency defects occur. As the write operations in the ISMN database are centrally managed and are running sequentially, two concurrent write operations cannot occur and thus cannot cause issues in the reproducibility, but race conditions which would lead to inconsistent reads can occur.

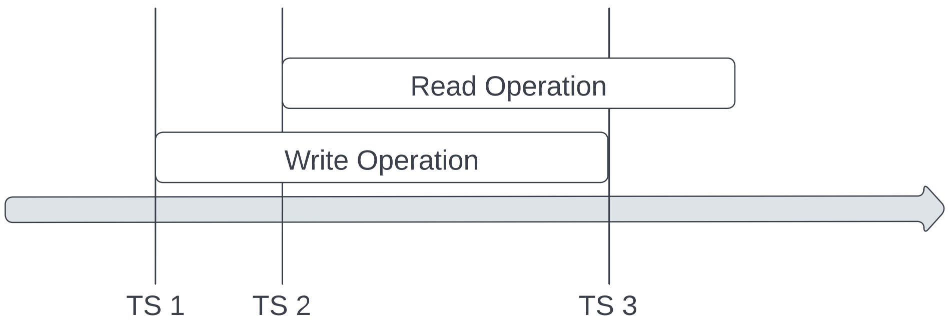


Figure 4.4: Illustration of two parallel operations, which can possibly cause race conditions

In Figure 4.4 we see an illustration of a race condition. The write operation starts at *TS 1*, but does not conclude before the read operation starts at *TS 2*, but finishes at *TS 3*. The read operation queries the database at *TS 2*, but as the write operation only commits data at *TS 3* the written data of this operation is not visible for the read operation. Upon re-execution these problems materialize, as new tuples are included in the results, which were not included in the original results, or tuples are excluded in the reproduced results, but not in the original results.

As the International Soil Moisture network has different integration frequencies, which reach from daily updated networks, over irregular updated networks to historical datasets, which are only inserted and not reprocessed later on. These updates vary in length, with most daily updates finished within ten minutes, while other datasets vary based on their sizes. Several times a year, several sensors and networks are reprocessed with corrected data or new quality flags. This process can take up to a few days to conclude. All of these updates only change the *dataset* table, and there are only further changes, if new sensors or networks are added.

To avoid race conditions, there are a few different solutions, which we will discuss below.

Solution: Locking

One valid approach to solve race conditions is to use locking. The main problem with an exclusive lock[8] on a database is, that no other operations are allowed in the meantime on a given table. Frequent updates and long running update queries, can limit the performance of the database extensively, as write operations are always running exclusively on a given table. Read operations can run in parallel, but each read needs to complete before a write operation can occur.

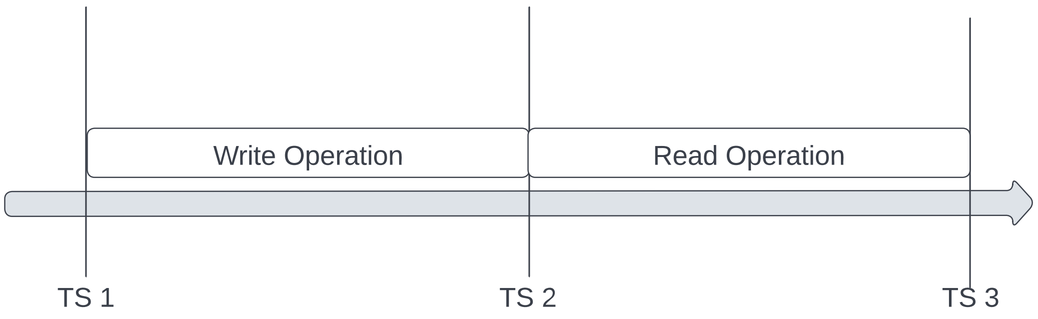


Figure 4.5: Solution to handle race conditions, by locking the database on write operations

In Figure 4.5 we see an illustration how exclusive locking in a database works. The write operation first needs to conclude at *TS 2* before the read operation is allowed to start. Read operations are allowed to run parallel to write operations, as long as the read operation was started first.

We can conclude, that locking the database would prevent race conditions from happening, and need to be used in update operations which involve more than one table. For the ISMN locking is a valid strategy, with a daily downtime of around ten minutes and possible longer downtimes for each other update.

[8]https://www.postgresql.org/docs/current/explicit-locking.html

Solution: Backdating Query Timestamp

An easy to implement approach to circumvene race conditions is to not query the most recent data, but use a negative offset, so that the execution timestamp of the SQL query is earlier than when the query actually was executed. Therefore, we need to issue the SQL query timestamp for a time that already passed, so that all write operations for this timestamp are already concluded.

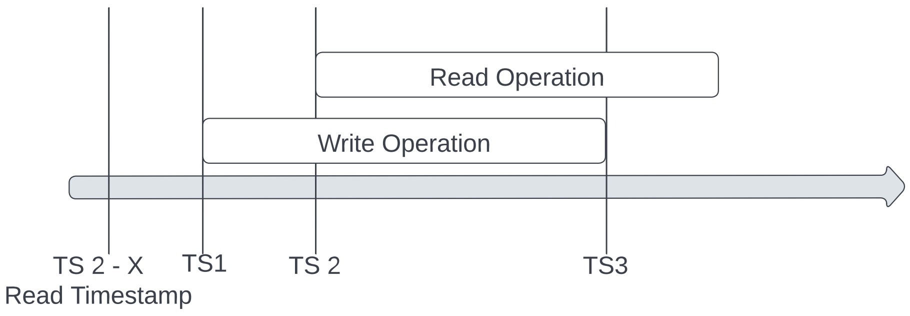


Figure 4.6: Solution to handle race conditions, by backdating the query execution timestamp

In Figure 4.6 we can see an illustration of this process. The write operation starts at *TS 1* and concludes at *TS 3*, while our read operation starts at *TS 2*. Instead of using timestamp *TS 2* the read operation takes a timestamp which has an offset of *X* from *TS 2* seen on the far left of the figure. With this offset, the query uses only data which has already been written to the database and circumvenes race conditions therefore.

For this strategy, only the integrated versioning approach can be used, as neither the hybrid nor the separated approach are using timestamps as conditions for filtering, when querying the most recent state of the database. It is not possible in the hybrid versioning, as we cannot determine if an update occurred between $TS2 - X$ and $TS1$ and if a tuple is no longer present in the original table. Therefore, we cannot use the *valid_from* column from the original table as filter criteria and would need to combine it to a re-executable SQL query using both the *valid_from* and *valid_to* timestamps from the history table as well. Although the separated approach can be theoretically used, as its history table has all timestamp information and tuples, similar to the original table from the integrated approach, this would be a workaround, which is not using the original table in any SQL query anymore.

Solution: Postdating Data Timestamp

Another way of changing the behaviour of timestamps, is instead of backdating SQL query timestamps to postdate timestamps of the underlying data. Therefore, we need to issue timestamps, when inserting or updating data so that they are in the future and only become active after a certain amount of time (e.g. 5 minutes). Therefore, race conditions will be circumvened as each write operation is only visible after a certain time.

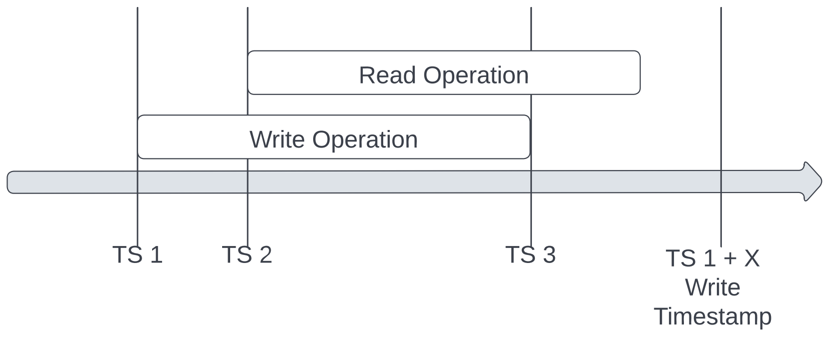


Figure 4.7: Solution to handle race conditions, by postdating the data versioning timestamps

In Figure 4.6 we can see an illustration of this process. The write operation starts at *TS 1* and concludes at *TS 3*, while our read operation starts at *TS 2*. Instead of using timestamp *TS 1* the write operation takes a timestamp which has an offset of *X* from *TS 1* seen on the right of the figure. With this offset, the inserted data is not active until all read operations have concluded and avoids race conditions.

Similar to the backdating of the SQL query timestamp, this approach only works with the integrated approach. For the hybrid approach, it does not work, because it cannot be ensured that no update or delete operations removed tuples from the original table, which are then not contained in the result set, but in the re-executed query. This would again require, that both the history table and the original table are both queried for the SQL query. For the separated approach this would work similar as before by using the history table only, and would no longer use the original table.

4.2 Persistent Identification of Subsets

The persistent and precise identification of subsets is vital to ensure that the data can be reproduced and changes in the retrieved data can be detected. This follows the guidelines R3 - Query Store Facilities, R5 - Stable Sorting, R6 - Result Set Verification, R7 - Query Timestamping, R8 - Query PID and R9 - Store Query to allow to re-execute already executed queries. The guideline R4 - Query Uniqueness is not necessary, as it can be guaranteed by design, as download requests consist of the same five queries, which cannot be adapted by the user. Guideline R10 - Automated Citation Texts is also not a part of this thesis, as these citations are generated in the frontend with retrieved data from the stored queries and are part of a complementary project.

For query re-execution, the system allows in the best case the re-execution of a given query (displayed in Listing 1) in the backend, with the state of the source-code and the state of the database at the time of execution. As the source is frequently changing, and further relies on many dependencies from other modules, this requires a rollback to a given git version, additionally to a versioned database. Due to the source code architecture, frequent source code adaptions and planned database adaptions, we will refrain from relying on source code versioning and rollbacks for the query correctness, and will store all created SQL queries (further called subqueries, to distinguish from the query which is sent to the backend) to allow the verification of each result set of a subquery.

Upon a user triggered download operation, the query needs to be stored with an execution timestamp and assigned to the users account, so that each user can only see their downloads. It is important that during the execution of a database the state of the database, on which the query is executed-on remains unchanged. If the state of the database changes, the re-execution may not produce the same results as the original execution. Possible database concurrency control strategies are discussed in Section 4.1.5. It is further necessary to specify all column-names for subqueries explicitly and do not use asterisks to obtain all columns of tables. If columns added or deleted this would then lead to problems with the number of columns not matching the expected number of columns, when processing the results in the backend, while also ensuring that the columns have an unambiguous ordering. Moreover, must each sub-query specify an explicit ordering of all tuples, so that the result set is always deterministic and can then be processed similarly by the backend. To verify the correctness of the re-execution of the query, as well as for each subquery, the hash of the result set is stored together with the associated query or subquery. The hash is calculated over the whole result set, including all columns and rows, so that the authenticitiy of the result set can be guaranteed.

From these requirements, we can derive a query store, which meets the needs of the ISMN database and supports their download process, by allowing the re-creation of the original result set, by combining all sub-query results. Therefore, the query store needs to store the original query, with a hash for verification and three different subqueries with up to 33,000 different parameter settings, with the corresponding result hashes, and result numbers. Therefore, the query store needs to consist of two tables, one holding all information regarding queries and the second one holding all information regarding subqueries. In the next two sections, we will discuss two different approaches, which can be used to store the queries efficiently in the database.

4.2.1 Basic Implementation

In the basic implementation, the query store consists of two tables, the *query* table, and the *subquery* table, as can be seen in Figure 4.8. On the left side, you can see the *query* table, which holds information about the user who started the download, the request, the timestamp when it was executed, and other metadata such as a digital object identifier(DOI)[9] and a result hash to verify the correctness of the query. In the *subquery*

[9]https://www.doi.org/

table, the original subquery is saved, together with a rewritten subquery, a result hash, and a number of retrieved results. The original query is the originally executed query on the system, while the re-executable query is the query that needs to be executed to obtain the same results again. Therefore, the re-executable query is a rewritten original query, enriched with timestamp information and using, if applicable, history tables. Both tables are interlinked by a foreign key, so that each query can be linked to a download and when a re-execution is called for a download, all associated queries can be executed.

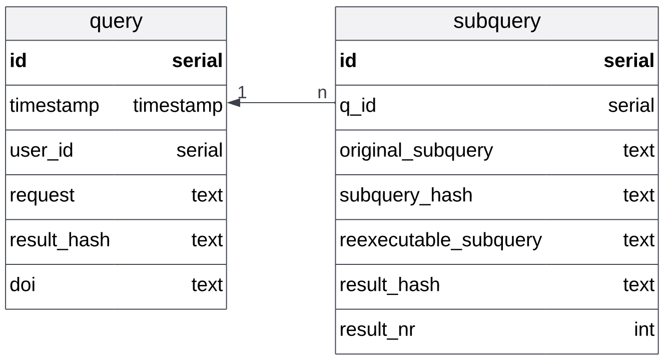


Figure 4.8: Schema of the query store, with an additional downloads table to track user downloads

The benefit of a simple approach as this is, as the subqueries are saved without any further compression to save storage space, all subqueries for a specific query can be obtained by querying the *subquery* table and filtering for a specific *q_id*. On the downside, we are saving the same 5 subqueries up to 33,000 times, which requires additional storage space. This impact could be decreased by the use of compression algorithms, as there are only five different queries, where each query has a different parameter setting. In such a setting, a compression algorithm could work well. An example of an already out-of-the-box working compression algorithm is TOAST[10]. It could be further decreased by using information from the *request* column from the *query* table, but as this information is processed in the backend and information is passed between queries, rollbacks to specific code versions are necessary.

4.2.2 Parameters Separated

A more complex implementation could consist of a three table setup, again with a download and query table, but all the parameters are saved in a different table, and each query is only saved once. The different parameters are then saved for each query in a different table, with the corresponding result hash. When we look at Figure 4.9, we can see that the parameters table contains three different arrays for the datatypes *int*, *text* and *timestamp*, so that all parameters can be saved accordingly and then used in the query

[10]https://www.postgresql.org/docs/current/storage-toast.html

correctly. The *timestamp_parameters* column here is used for the sensor measurement timestamps, which are seen in line 17 and 18 of Listing 4. In the *original_query* and the *re-executable_query* fields in the query table, these values would be needed to be set with wildcards, so they can be replaced with each different parameter setup. Furthermore, the *result_nr* and *result_hash* are in the parameters table, as we need to know these values per parameter setting.

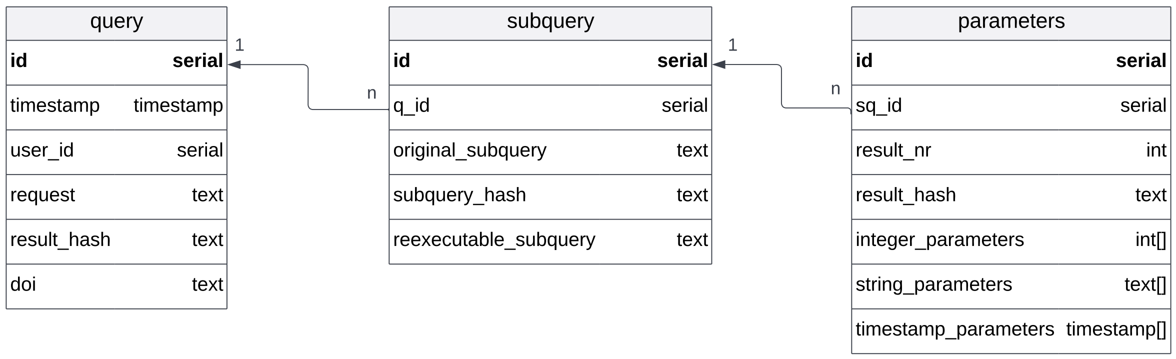


Figure 4.9: Schema of the query store, with an additional downloads table to track user downloads and the parameters separated in an additional table

```
SELECT * FROM data.dataset
WHERE id = #i1 AND value > #i2 AND
dataset_utc >= #t1 AND dataset_utc <= #t2
```

Listing 5: Possible Rewrite to save storage space for each query

This should, compared to the basic approach, lead to less storage space required, as the queries are only saved once, but we need to take into account that all queries would need to be re-written, as they are stored in an optimized form, which can be seen in Listing 5. Instead of saving the whole query string for each query, we are just saving a query with placeholders. For each different datatype a different placeholder exists, so the $\#i2$ placeholder represents the second element in the integer array, while $\#t1$ refers to the first timestamp in the timestamp array.

4.3 Resolving and Sharing Subsets

Resolving and sharing a subset via a persistent identifier is vital to speed up scientific discovery and allow the verification of research results. In the guidelines R11 - Landing Page and R12 - Machine Actionability are these two recommendations discussed. The ISMN webportal needs to be adapted to accommodate the needs of retrieving machine actionable data, have a landing page presenting all necessary metadata and a way to see and re-execute all previously made downloads. Furthermore, the Landing Page needs to allow to publish a DOI directly, so that a citation can be generated automatically based on all gathered data of the database and without onward distributing the data.

This task is separated from this thesis and completed in a complementary project, as it is out of the scope of this project. The landing page will list the executed downloads per user and the user can request a DOI for each download. Moreover, a Bibtex citation is automatically generated, so that the user can easily give credit to all networks, which have contributed to the downloaded data.

4.4 Modification of Infrastructure

As databases are constantly evolving, we also need to take into account how a change of the underlying data structure would affect the reproducibility of our system. Therefore it is necessary to plan for a potential technology migration and a verification of the correct migration, as discussed in R13 and R14. The verification can be achieved by reusing the same hashes, used for the verification of the re-execution of the queries, but for the different transformations a database can do, we need to look into some of the different possible Data Definition Language commands possible for such a transformation and what type of rewriting is necessary for these approaches.

We will limit the modification of the infrastructure to *CREATE*, *DROP*, and *RENAME* operations for tables and columns, as these are the most commonly used ones. Over the years there have been several schema changes within the ISMN database and further changes are planned (discussed in Section 3.1.1).

Below we will describe each operation in detail and how the schema needs to be adapted. These adaptions are similar to the schema adaptions presented by Säuerl[20], but with the goal of automatically processing them from within the PostgreSQL database. All the presented figures are shown for the separated approach, but work similarly for the integrated and the hybrid approach.

4.4.1 Create Table

One of the most common use cases in a database is to create a new table. Therefore, we need to apply versioning to added tables automatically to ensure the reproducibility of queries. It is necessary to exclude namespaces and specific naming conventions for tables, as otherwise recursive function calls can happen (e.g. for each history table, another history table would be created). In Figure 4.10, we can see on the creation of the table *dataset*, the table *dataset_hist* is automatically created as well, with versioning already applied to it. So that whenever a new tuple is introduced to the original table it is versioned automatically. In temporal tables this can be achieved by adding WITH SYSTEM VERSIONING to any CREATE TABLE statement, so that automatically system-versioning is applied to the table.

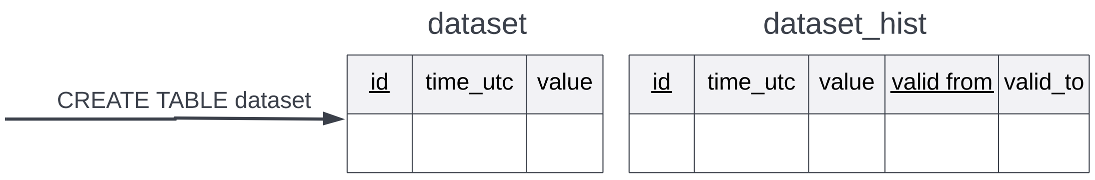
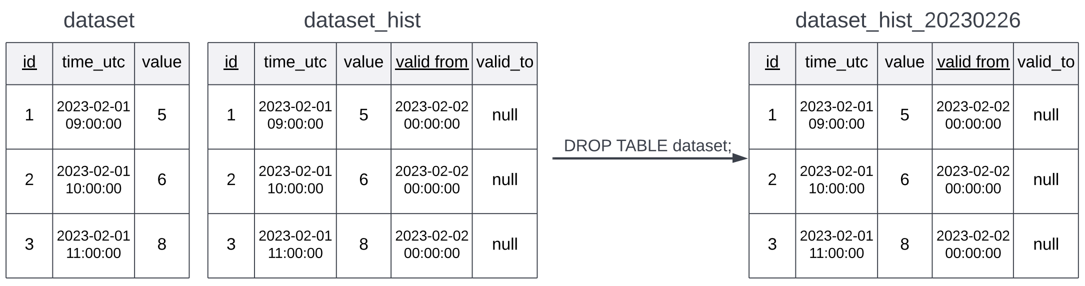


Figure 4.10: Creation of table dataset, and automatic creation of associated history table

4.4.2 Drop Table

If a drop table event is called, it is important, that no data is lost in the process, as queries using this table are not reproducible anymore. For the separated approach, this is not as important, as for the integrated or hybrid approach, as the data is already stored redundantly and will not be deleted. For the integrated and the hybrid approach, the data from the original table needs to be renamed or copied, to mark it as a deleted table. In Figure 4.11 we drop the table *dataset* and the history table *dataset_hist* is automatically archived, by renaming it and possibly moving it to a different namespace. With renaming the table, we allow to avoid problems in the future, if the same table is reused. Further, it is necessary to rewrite all queries in the query store to the new table name, and the table should be immutable, as the table is officially deleted and only used for re-executing queries.



dataset

id	time_utc	value
1	2023-02-01 09:00:00	5
2	2023-02-01 10:00:00	6
3	2023-02-01 11:00:00	8

dataset_hist

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	null

dataset_hist_20230226

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	null

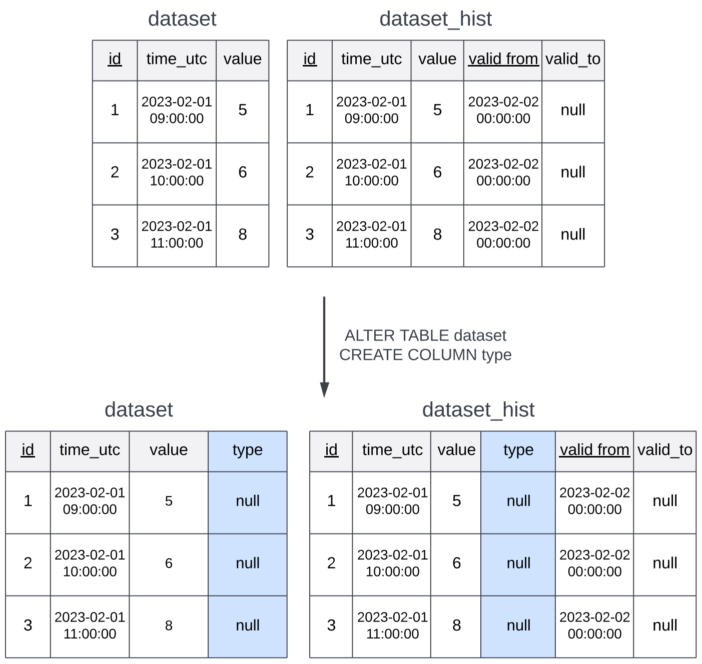
Figure 4.11: Deletion of table dataset, and automatic renaming and historization of history table

4.4.3 Rename Table

If a table is renamed, it is necessary to rewrite all queries in the query store and also rename the history table as well. This is due to the fact, that the history table can cause problems, if the table with the original name is newly created in the database, as the history table cannot be created as it already exists. The renaming process would work similarly for the integrated and hybrid approach, by only renaming the target table and rewriting all queries to the new table name. Also created database structures, as triggers and procedures, which are referencing the old table name need to be changed.

4.4.4 Add Column

When a column is added, the column also needs to be added to the history table. The values will by default be initialized as null and as all query columns need to be fully specified in any query, there is no need to rewrite any already executed query. This is because stored queries are not allowed to contain any asterisk, as then the number of columns and returned ordering of the columns is not specified.



dataset

id	time_utc	value
1	2023-02-01 09:00:00	5
2	2023-02-01 10:00:00	6
3	2023-02-01 11:00:00	8

dataset_hist

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	null

dataset

id	time_utc	value	type
1	2023-02-01 09:00:00	5	null
2	2023-02-01 10:00:00	6	null
3	2023-02-01 11:00:00	8	null

dataset_hist

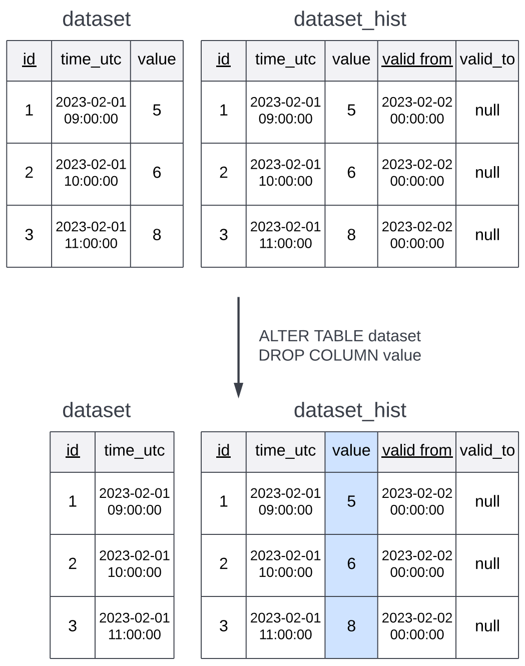
id	time_utc	value	type	valid from	valid_to
1	2023-02-01 09:00:00	5	null	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	null	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	null	2023-02-02 00:00:00	null

Figure 4.12: Addition of a column type to the dataset table, and automatic update of the history table

In Figure 4.12, we see that the column type is added to the dataset table. This column is upon insertion automatically transferred to the history table, and is initialized in both tables with the value *null*. All versioning on the dataset table, needs to be adapted, so that it also writes changes that are part of the newly created *type* column, as in the SQL procedures these columns are not specified to be included in versioning.

4.4.5 Drop Column

When dropping a column, we do not want to change the history table, as we might delete vital information, when reprocessing a query. Therefore, any dropped column will be kept in the history table.



dataset

id	time_utc	value
1	2023-02-01 09:00:00	5
2	2023-02-01 10:00:00	6
3	2023-02-01 11:00:00	8

dataset_hist

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	null

ALTER TABLE dataset
DROP COLUMN value

dataset

id	time_utc
1	2023-02-01 09:00:00
2	2023-02-01 10:00:00
3	2023-02-01 11:00:00

dataset_hist

id	time_utc	value	valid from	valid_to
1	2023-02-01 09:00:00	5	2023-02-02 00:00:00	null
2	2023-02-01 10:00:00	6	2023-02-02 00:00:00	null
3	2023-02-01 11:00:00	8	2023-02-02 00:00:00	null

Figure 4.13: Drop of a column of the dataset table, and automatic update of the history table

In Figure 4.13 this process is illustrated, on a drop column event, the *value* column is dropped from the *dataset* table, but is kept in the *dataset_hist* table. No rewriting of columns is necessary during this process, as the data for the re-execution stays the same, but used triggers and rules may need to be rewritten during this event.

4.4.6 Rename Column

When renaming a column, it is necessary to also rename the column in the history table, so that columns can easily be mapped between the original and the history table. Furthermore, it is necessary to rewrite any queries that are using the column, to ensure that the re-execution of these queries is working correctly. Also, all stored procedures and views that are referencing this column need to be rewritten, to work correctly, as they are not automatically updated. All column names are best written in an easily detectable format, so that all queries containing the column name can be identified without problems and when rewriting, only the renamed column is changed.

CHAPTER 5

Implementation

For the implementation, we used solely SQL and PL/SQL statements in a PostgreSQL database in version 14.6. This was important, as the workflow of the ISMN project is complex, and larger changes to the codebase could most likely not be integrated well into their system. Furthermore, their system is migrated from the Technical University of Vienna, to the BfG[1] in Germany, which is another reason, the integration of the versioning should be low effort. As it is not possible to control queries or lock tables from within a PostgreSQL database, these are adaptions, which need to be performed in the codebase and not in the database itself.

In Section 5.1 we will discuss how we implemented the three different versioning approaches, and how they needed to be adapted to fit the ISMN database schema and the corresponding processes. Then we will describe in Section 5.2 how the query store was implemented. In Section 5.3 we will then discuss how the process of the ISMN backend needs to be adapted to rewrite all queries to use the versioned database. In Section 5.4 we elaborate on the use of event triggers, to catch schema changes and how to adapt the schema in case of different changes. In Section 5.5 we will discuss strategies to prevent race conditions in our implementation, which must be implemented outside of the PostgreSQL database.

5.1 Versioning

For the versioning, we used a combination of Rules[2] and Triggers, which are customized for each different table. Rules in general can overwrite or extend commands in SQL. One major difference between Triggers and Rules is when they are triggered. On COPY INTO statements, in which a table is populated by a textfile, a rule is not activated, as no INSERT command is used, but a trigger is, as the table is changed.

[1]https://www.bafg.de
[2]https://www.postgresql.org/docs/15/rules-update.html

To add versioning to the database we are following the steps below:

1. Add Versioning columns to target table
2. Adapt the Primary Key
3. Set CRUD behaviour

In the following, we will discuss how we implemented the versioning for the three different approaches.

5.1.1 Integrated Versioning

As described in depth in Section 4.1, integrated versioning uses the same table for versioning which is also used for standard retrieval. Therefore, it is necessary to change the table before or after applying versioning to it, by adapting uniqueness constraints. This is relevant, as uniqueness constraints are not working unconditionally together with versioning approaches, as unique values are not necessarily unique after updates anymore.

```
1      EXECUTE format('ALTER TABLE %s ADD valid_from
        ↪  Timestamp;', tablename);
2      EXECUTE format('ALTER TABLE %s ALTER COLUMN valid_from
        ↪  SET default now();', tablename);
3      EXECUTE format('UPDATE %s SET valid_from = now() WHERE
        ↪  valid_from is NULL;', tablename);
4      EXECUTE format('ALTER TABLE %s ADD valid_to Timestamp;',
        ↪  tablename);
5      EXECUTE format('ALTER TABLE %s ADD CONSTRAINT %s_pkey
        ↪  primary key(%s,valid_from) ',
        ↪  tablename,tablename_wo_schema,primary_key);
```

Listing 6: Integrated: Altering Table to create *valid_from* and *valid_to* columns and set *valid_to* to now and adapt primary key

In Listing 6, we can see an excerpt of the general approach of adapting a table to support versioning. For readability, the functions are split into several parts and parameter retrievals and constraint adaptions are left out. They are implemented inside of EXECUTE FORMAT commands, so that the statements are correctly set for the corresponding table. Versioning is done by adding two additional columns, which are created in Line 1 and 4. In Line 2 a default value for the *valid_from* column is set, so that all future inserts are valid upon insertion time. In Line 4 all tuples, which are in the database at the time of versioning, are updated and the *valid_from* timestamp is set to the actual time. Then in Line 5, the primary key, which has been deleted before is added, with the original primary key columns, and additionally also the *valid_from* column.

```
1       -- Delete Rule
2       EXECUTE format('CREATE RULE delete_from_%3$s AS ON
         ↪  DELETE TO %1$s ' ||
3                      'DO INSTEAD (' ||
4                      '    UPDATE %1$s SET valid_to = now()
                        ↪  where %2$s;' ||
5                      ');', tablename,primary_key_conditions,
                        ↪  tablename_wo_schema);
6       -- Update Function
7       EXECUTE format('CREATE OR REPLACE FUNCTION
         ↪  update_trigger_function_%3$s() ' ||
8                      'RETURNS TRIGGER ' ||
9                      'LANGUAGE PLPGSQL ' ||
10                     'AS $update_trigger_function$ ' ||
11                     'BEGIN' ||
12                     '    IF new.valid_to IS NULL THEN ' ||
13                     '    INSERT INTO %5$s%3$s(%1$s)
                        ↪  VALUES(%2$s);' ||
14                     '    %4$s;' ||
15                     '    NEW.valid_to = now();' ||
16                     '    END IF;' ||
17                     '    RETURN NEW;' ||
18                     'END; $update_trigger_function$ ',
                        ↪  columnnames, new_columnnames,
                        ↪  tablename_wo_schema, column_reset,
                        ↪  schema_name);
19      -- Update Trigger
20      EXECUTE format('CREATE TRIGGER update_trigger_%2$s ' ||
21                     'BEFORE UPDATE ' ||
22                     'ON %1$s ' ||
23                     'FOR EACH ROW ' ||
24                     '    WHEN (pg_trigger_depth() < 1)' ||
25                     '        EXECUTE PROCEDURE
                        ↪  update_trigger_function_%2$s();',
                        ↪  tablename, tablename_wo_schema);
```

Listing 7: Integrated: Creation of Rules and Triggers to version all changes created by CRUD operations

In Listing 7, we can see the general approach for CRUD operations. As the versioning is handled directly in the original table, there is no need for an additional INSERT Rule or Trigger, as a default value for new inserts is set for each tuple. We can see the DELETE Rule from Line 1 to 5, where we are defining the behaviour, when a deletion from the table occurs. Instead of deleting the actual values, the process is changed and instead (See Line 3), an UPDATE is triggered in Line 4, which deactivates the tuple by setting the *valid_to* timestamp. The UPDATE function starts in Line 8 and goes until Line 19 and the Line 8 to 12 and 19 are the function syntax. In Line 13, we are testing if the new *valid_to* value is null, and terminate the function if it is not, to adapt for the triggering by the DELETE rule. In Line 13 we check if the *valid_to* value is not set. If it is not set, we insert the new tuple in the database in Line 14 and then reset all column values to the original ones in Line 15 and set the *valid_to* timestamp. In the trigger definition from Line 21 to 26, it is worth mentioning, that we circumvent recursive calls in Line 25, by specifying the trigger depth as 1, which prevents nested trigger calls.

5.1.2 Separated Versioning

As described in depth in Section 4.2, separated versioning uses a copy of the table with additional valid times for tuples, a so called history table, which is only used for query re-execution. All queries on the system can be run without a change, but need to be documented and can then be re-run on the history tables. As all original tables remain unchanged, we do not need to lift any constraints on the original tables.

```
1     EXECUTE format('CREATE TABLE hist.%s_hist(like %s);',
       ↪  tablename_wo_schema, tablename);
2     -- Add valid_from column
3     EXECUTE format('ALTER TABLE hist.%s_hist ADD valid_from
       ↪  Timestamp;', tablename_wo_schema);
4     EXECUTE format('ALTER TABLE hist.%s_hist ALTER COLUMN
       ↪  valid_from SET default now();',
       ↪  tablename_wo_schema);
5     EXECUTE format('ALTER TABLE hist.%s_hist ADD valid_to
       ↪  Timestamp;', tablename_wo_schema);
6     EXECUTE format('INSERT INTO hist.%s_hist SELECT * FROM
       ↪  %s;', tablename_wo_schema, tablename);
```

Listing 8: Separated: Altering Table to create history table with *valid_from* and *valid_to* columns and copy all data to history table

In Listing 8, we can see a generalized form for applying the initial separated versioning to a table. First, we create in Line 1 a history table with the same columns as in the original table and then create a *valid_from* column with default value in Line 3 and 4. In Line 5 we create the *valid_to* column and then copy all data from the original table to the history table.

```
    -- Delete Rule
    EXECUTE format('CREATE RULE delete_from_%3$s AS ON
     ↪  DELETE TO %1$s ' ||
                   'DO ALSO (' ||
                   '    UPDATE hist.%3$s_hist SET valid_to
                    ↪  = now() where %2$s;' ||
                   ');', tablename,primary_key_conditions,
                    ↪  tablename_wo_schema);
    -- Update Function
    EXECUTE format('CREATE OR REPLACE FUNCTION
     ↪  update_trigger_function_%3$s() ' ||
                   'RETURNS TRIGGER ' ||
                   'LANGUAGE PLPGSQL ' ||
                   'AS $update_trigger_function$ ' ||
                   'BEGIN' ||
                   '    UPDATE hist.%3$s_hist SET valid_to
                    ↪  = now() WHERE %6$s ;' ||
                   '    INSERT INTO hist.%3$s_hist(%1$s)
                    ↪  VALUES(%2$s);' ||
                   '    %4$s;' ||
                   '    RETURN NEW;' ||
                   'END; $update_trigger_function$ ',
                    ↪  columnnames, new_columnnames,
                    ↪  tablename_wo_schema, column_reset,
                    ↪  schema_name,
                    ↪  primary_key_conditions);
    -- Insert Function
    EXECUTE format('CREATE OR REPLACE FUNCTION
     ↪  insert_trigger_function_%3$s() ' ||
                   'RETURNS TRIGGER ' ||
                   'LANGUAGE PLPGSQL ' ||
                   'AS $insert_trigger_function$ ' ||
                   'BEGIN' ||
                   '    INSERT INTO hist.%3$s_hist(%1$s)
                    ↪  VALUES(%2$s);' ||
                   '    RETURN NEW;' ||
                   'END; $insert_trigger_function$ ',
                    ↪  columnnames, new_columnnames,
                    ↪  tablename_wo_schema);
```

Listing 9: Separated with Rules: Creation of Rules to version all changes created by CRUD operations

In Listing 9 we can see the defined rules and triggers for all CRUD operations. From

Line 2 to 5 is the rule for deleting entries, which additionally (Line 3) updates the history table and sets the *valid_to* timestamp to now() in Line 4 and 5. The trigger function for an update is in Line 7 to 16. In Line 12, the row, which is updated in the original table is marked as deleted in the history table. In Line 13 the updated tuple is then inserted in the history table. From Line 18 to 25 we see the insert function. On each insert to the original table, the result is copied to the history table, as seen in Line 23.

Separated Versioning with Rules

A second, easier readable version for the separated approach was created, which uses only rules for INSERT, UPDATE, and DELETE in comparison to the before described approach. This approach has the limitation, that INSERT and UPDATE commands are not applying versioning, when used in combination with the COPY FROM command[3].

```
1   EXECUTE format('CREATE RULE delete_from_%3$s AS ON DELETE
     ↪  TO %1$s ' ||
2                  'DO ALSO (' ||
3                  '    UPDATE hist.%3$s_hist SET valid_to =
                    ↪  now() where %2$s;' ||
4                  ');', tablename,primary_key_conditions,
                    ↪  tablename_wo_schema);
5   EXECUTE format('CREATE RULE update_from_%3$s AS ON UPDATE
     ↪  TO %1$s ' ||
6                  'DO ALSO (' ||
7                  '    UPDATE hist.%3$s_hist SET valid_to =
                    ↪  now() where %2$s;' ||
8                  '    INSERT INTO hist.%3$s_hist SELECT *
                    ↪  FROM %1$s WHERE %2$s;' ||
9                  ');', tablename,primary_key_conditions,
                    ↪  tablename_wo_schema);
10  EXECUTE format('CREATE RULE insert_from_%3$s AS ON INSERT
     ↪  TO %1$s ' ||
11                 'DO ALSO (' ||
12                 '    INSERT INTO hist.%3$s_hist SELECT *
                    ↪  FROM %1$s WHERE %2$s ' ||
13                 ');', tablename, primary_key_conditions_new,
                    ↪  tablename_wo_schema);
```

Listing 10: Separated with Rules: Creation of Rules to version all changes created by CRUD operations

In Listing 10 we see this approach. From Line 1 to 4, we see the rule for deletion, which is the same as for the approach with triggers, as deletions cannot be performed by COPY

[3]https://www.postgresql.org/docs/current/sql-copy.html

FROM. In Line 5 to 9 we see the rule for update, which additionally (Line 6) updates the history table and deactivates the tuple which is updated in Line 7 and inserts the new updated tuple in Line 8. From Line 10 to 13 we see the insert rule, which copies all inserted data to the history table in Line 12.

As this approach does not support the COPY FROM command, and corresponding data is not transferred to the history table, when using this command - this would lead to a diverging number of tuples and further lead to non-reproducible queries. As the COPY FROM operation only affects INSERT AND UPDATE operations, it is save to use rules for DELETE operations, as these cannot be triggered by a COPY FROM statement.

5.1.3 Hybrid Versioning

As described in depth in Section 4.3, hybrid versioning transfers all inactive tuples of the original table with a set *valid_to* time to a so called history table, which is only used for query re-execution. All queries on the system can be run without a change, but need to be documented and can then be re-run on the history tables. As the original tables remain unchanged, we do not need to add any additional constraints on the original query.

```
1 EXECUTE format('ALTER TABLE %s ADD valid_from Timestamp;',
  ↪  tablename);
2 EXECUTE format('ALTER TABLE %s ALTER COLUMN valid_from SET
  ↪  default now();', tablename);
3 EXECUTE format('UPDATE %s SET valid_from = now() WHERE
  ↪  valid_from is NULL;', tablename);
4 -- CREATE HYBRID
5 EXECUTE format('CREATE TABLE hist.%1$s_hist(like %2$s);',
  ↪  tablename_wo_schema, tablename);
6 EXECUTE format('ALTER TABLE hist.%s_hist ADD valid_to
  ↪  Timestamp;', tablename_wo_schema);
```

Listing 11: Hybrid: Altering Table to create history table with *valid_from* and *valid_to* columns and add *valid_from* column to the original table

In Listing 11, we can see a generalized form for applying the initial hybrid versioning to a table. First, we add the *valid_from* column in Line 1 to the original table, and set the current timestamp as default value in Line 2. In Line 3 we update all already present tuples to the actual time. In Line 4 we then create a history table with the same columns as in the original table, and add the *valid_to* column in Line 5.

```
 1  -- Delete Rule
 2  EXECUTE format('CREATE RULE delete_from_%3$s AS ON DELETE
     ↪  TO %1$s ' ||
 3              'DO ALSO (' ||
 4              '    INSERT INTO hist.%3$s_hist SELECT * FROM
                 ↪  %1$s WHERE %2$s;' ||
 5              '    UPDATE hist.%3$s_hist SET valid_to = now()
                 ↪  where %2$s;' ||
 6              ');', tablename,primary_key_conditions,
                 ↪  tablename_wo_schema);
 7  -- Update Function
 8  EXECUTE format('CREATE OR REPLACE FUNCTION
     ↪  update_trigger_function_%3$s() ' ||
 9              'RETURNS TRIGGER ' ||
10              'LANGUAGE PLPGSQL ' ||
11              'AS $update_trigger_function$ ' ||
12              'BEGIN'
13              '    INSERT INTO hist.%3$s_hist(%1$s, valid_to)
                 ↪  VALUES(%2$s, now());' ||
14              '    RETURN NEW;' ||
15              'END; $update_trigger_function$ ', columnnames,
                 ↪  old_columnnames, tablename_wo_schema);
```

Listing 12: Hybrid Versioning: Creation of Rule and Trigger to version all changes created by CRUD operations

In Listing 12 we can see the two necessary functions to apply versioning to the hybrid approach. From Line 2 to 6 we see the deletion rule, which copies the tuple in Line 4 to the history table, and sets the *valid_to* timestamp in Line 5. In Line 8 to 16 we see the update function, which transfers the updated tuple to the history table, by first inserting it into the history table in Line 13 and setting the *valid_to* timestamp to the current time. As triggers are executed during the same transaction as the update, and the PostgreSQL transaction system only creates one timestamp per transaction (at the start), there cannot exist gaps in the timeline where no version of the tuple was active.

5.2 Query Store

The implementation of the query store is straight forward and we follow our presented simple schema (see Figure 4.8) and add one function and two stored procedure to improve the insert process. As the tables are linked by a foreign key, the function *save_query* assigns a given query to a user and stores the associated metadata (user_id, download_request) in the query table and returns the pid and the timestamp, which can then be used to link all associated subqueries to the query and add the execution timestamp to each subquery.

For the subqueries a procedure exists that stores a given pair of SQL queries (once the original query, and a rewritten query for re-execution) with the hash, the number of tuples of the result set and the query id and automatically hashes the original query, with a SHA256 hash, upon insertion.

After all subqueries have concluded, the function *save_query_result* stores the hash for a given query, upon calculating the SHA256 hash of the final result set.

This structure has the benefit, that the query store table is not directly accessed by the backend, and the backend only connects to the query store by functions and procedure calls. Furthermore, the query store logic can be shifted to the database, and operations such as the query rewrite process can be performed by the database. This would help, as the backend only needs to call the *save_subquery* function with the original query and the given results and the database could perform the query rewriting.

5.3 Query Rewriting

To be able to re-execute the subqueries to return the same results as in the original table, it is necessary that the subqueries are rewritten before they are executed or re-executed, as subqueries without timestamp information are not necessarily using the correct state of the data. The rewriting is a process, which is necessary, as different SQL queries need to be executed for the original execution and the re-execution. While in the separated and the hybrid approach no adaption of the original subquery is necessary, the SQL query for re-execution needs to be changed so that it uses the corresponding history tables. For the integrated versioning strategy on the other hand, the original subquery does not necessarily need to differ, by including all timestamp information already. As the information is split differently in each versioning approach, the query rewriting differs for each approach.

5.3.1 Integrated Versioning

As the integrated approach does not use history tables, the rewriting process only needs to be done for the original query.

The original query needs to be enriched with timestamp information, so that the correct state of the data is returned, as each table contains multiple different states. As the subquery is already executed on a versioned table, with all the necessary timestamp information, it is not necessary to rewrite the subquery for re-execution but we can use the same query again. It would be further possible to use only the *valid_from* column in the database for the original subquery execution, and use both timestamp information(*valid_from*,*valid_to*) only upon re-execution. The query execution timestamp is stored together with the query, when the query was started and is then used for enriching subqueries with timestamp information.

```
1  SELECT s.station_id
2  FROM   data.station
3  WHERE  valid_from <= '2023-02-24 15:00:00'
4         AND ( valid_to IS NULL
5               OR valid_to > '2023-02-24 15:00:00' )
6         AND network_abbr = 'ARM';
```

Listing 13: Query Rewriting to obtain the correct data by adding a validity period when querying the data

In Listing 13 we see a SQL query issued on the *data.station* table in the integrated approach. The subquery is extended with versioning information in Line 3 to 5 we see that this versioning information is added. In Line 3 we filter tuples, which have been inserted before 15:00 on February 24th 2023. In Line 4 and 5 we then further filter, that the tuple needs to be active (not deleted) or that the deletion of the tuple happened later than the query execution.

5.3.2 Separated Versioning

For the separated versioning approach, we do not need to rewrite the original query, as we keep the most recent state of the data in this table. But for any re-execution of a query, it is necessary to rewrite it, so that it is executed on the history table with timestamp information.

```
1  SELECT s.station_id
2  FROM   hist.station_hist
3  WHERE  valid_from <= '2023-02-24 15:00:00'
4         AND ( valid_to IS NULL
5               OR valid_to > '2023-02-24 15:00:00' )
6         AND network_abbr = 'ARM';
```

Listing 14: Query Rewriting to obtain the correct data by using the history table and adding a validity period when re-execution a query

In Figure 14 we see a rewritten query, which can be used to retrieve results from the history table *hist.station_hist*. Similar to the integrated approach, we need to add the same timestamp conditions from Line 3 to 5. Additionally, we need to change the tablename to the corresponding history table, which happens in the highlighted Line 2.

5.3.3 Hybrid Versioning

For the re-execution in the hybrid approach, it is necessary to combine the result sets from the original table and the history table. The original query can be kept the same, as the original table keeps the most recent state of the data, while all inactive tuples are moved to the history table.

```
1  SELECT s.station_id
2  FROM   data.station
3  WHERE  valid_from < '2023-02-24 15:00:00'
4         AND network_abbr = 'ARM'
5  UNION
6  SELECT s.station_id
7  FROM   hist.station_hist
8  WHERE  valid_from <= '2023-02-24 15:00:00'
9         AND valid_to > '2023-02-24 15:00:00'
10        AND network_abbr = 'ARM';
```

Listing 15: Query Rewriting to obtain the correct data by using the history table and the original table and unioning the results

In Listing 15 we see a rewritten query for the hybrid approach. In this approach, it is necessary to combine the results from the original *data.station* table with the results from the history table *hist.station_hist* with a union (Line 5). In the original table, we need to filter on the *valid_from* column, as seen in Line 3, to ensure that we only use data for the re-execution which was present in the table on execution time. In Line 8 and 9, we see the restrictions for the *valid_from* and *valid_to* timestamps to obtain the correct versioning from the history table. It is worth mentioning here, that as records are only transferred to the history table upon deletion in the original table, we do not need a condition to check if *valid_to* is null.

As at no time, we know if a tuple is present in the original or in the history table it is always necessary to query both tables to find the correct result. Therefore, it is necessary to combine the results, for each usage of a history table. This behaviour can be abstracted by the usage of SQL views to automatically combine these two tables to a single result set, which is similar to the history table of the separated approach.

```
CREATE VIEW hist.station_combined
AS
  (SELECT s.station_id,
          s.valid_from,
          NULL
   FROM   data.station
   UNION
   SELECT s.station_id,
          s.valid_from,
          s.valid_to
   FROM   hist.station_hist);
```

Listing 16: Usage of a view, to combine the results from the history and the original table for the hybrid approach

In Listing 16, we are creating a view that combines the original and the history table. As the original table, does not contain a *valid_to* column, as can be seen in Line 5, it is set to null. The history table provides the *valid_to* column in Line 10 for all inactive tuples, so that we have all necessary timestamp information. Therefore, it is possible to use the same query syntax (with adaptions that the query needs to use the view name) as in the separated approach, with the downside of longer retrieving times, as two tables need to be merged and filtered. This overhead can be decreased by using materialized views, to store the combined results of the original and the history table in the database, but has several downsides as materialized views are not automatically refreshing and are materialized in the database for fast retrieval. This would significantly increase the storage consumption, as each tuple would be stored twice with the corresponding timestamp information.

5.4 Schema Changes

To implement schema changes, we use PostgreSQL event trigger[4] to catch any operation that changes our database schema, according to our defined change operations (see Section 2.4). These triggers are similar to normal triggers, but can be called before or after schema modifications. As the final modification is only known after the transaction has concluded, it is not possible to obtain information about the changes, from the PostgreSQL event trigger function *pg_event_trigger_ddl_commands()*, as this function is only available after a transaction has concluded and data may have already been altered or deleted. As this is for most operations too late, we used the *current_query()* function, which returns the issued command, which started the event. From this command on it is possible to obtain the tablename and the columnname.

All presented schema changes are shown for the separated approach, but work similarly for the integrated and hybrid approach, but with additional overhead for transferring tuples.

5.4.1 Create Table

When creating a table, we want to automatically version the table, by calling our *add_versioning* function, presented in Section 5.1. As this command is only introducing new information to the database and versioning can only be added after the table was created, this trigger must be called after the transaction was concluded.

[4]https://www.postgresql.org/docs/current/event-triggers.html

```
1  SELECT object_identity INTO name FROM
   ↪  pg_event_trigger_ddl_commands()
2  WHERE object_identity NOT LIKE '%_seq' AND
3  object_identity NOT LIKE '%_pkey' AND
4  object_identity NOT LIKE '%_hist';
5
6  IF name is null then
7     RETURN;
8  end if;
9
10 CALL add_versioning_separated(name);
```

Listing 17: Procedure to add versioning automatically upon CREATE TABLE event

In Listing 17 we can see the function, which automatically includes versioning. In Line 1 to 4 we obtain the created tablename from the *pg_event_trigger_ddl_commands()* function, and exclude tables that are automatically created, as sequences (Line 2), primary keys (Line 3) or history tables (Line 4).

5.4.2 Drop Table

When dropping a table, we want to archive the table, with all existing states. In the separated approach, it is sufficient to archive the history table. This is done, by renaming the table and rewriting all stored subqueries, so that they are executed on the new renamed table. For the integrated and the hybrid approach it is necessary, to prevent the table from being dropped, before all necessary information has been retrieved and stored. For the integrated approach, this can be done by moving/copying the table and for the hybrid approach we can do this by merging the original and the history table (e.g issuing a delete statement for every row in the original table and thus moving them to the history table) or by moving the original table. For all approaches, we need to rewrite the queries, so that they are using the correct tables, as the history table will be renamed during this process, to avoid problems if in the future a table with the same name is created.

```
1  SELECT current_query() into query;
2  SELECT TO_CHAR(now() :: DATE, 'yyyymmdd') into date;
3  IF LOWER(query) like '%drop table%' then
4          SELECT SPLIT_PART(SPLIT_PART(lower(query),
            ↪   lower('DROP TABLE '), 2),';',1) into name;
5          SELECT SPLIT_PART(name, '.', 2) INTO tablename;
6          IF (tablename not like '%_hist%') THEN
7              execute format('ALTER TABLE hist.%1$s_hist
                ↪   RENAME TO %1$s_hist_%2$s
                ↪   ;',tablename,date);
8              execute format('UPDATE query_simple SET
                ↪   re_execute_query = REPLACE(re_execute_query,
                ↪   ''hist.%1$s_hist'',
                ↪   ''hist.%1$s_hist_%2$s'') WHERE
                ↪   re_execute_query like ''%%hist.%1$s%%'' ',
                ↪   tablename,date);
9          end if;
10 end if;
```

Listing 18: Procedure to automatically archive a table, when it is dropped

In Listing 18 we see the drop table process illustrated. First we obtain the called query in Line 1 and the current date in Line 2. In Line 3 we are then validating if the obtained command is a drop table command, and then extract the tablename from the query in Line 4 and 5. In Line 7 we are then renaming the corresponding history table and adding the date so it has the format *<tablename>_hist_<yyyymmdd>*, so that the table is archived and not interfering with the versioning in the future. In Line 8 we are then rewriting all queries, which are containing the name of the history table, to the new name of the history table with a string replacement. For the separated approach this function can be called before or after the transaction is completed, but for the integrated and hybrid approach it is necessary to call the function before the transaction has started.

5.4.3 Rename Table

When renaming a table, we want to rename the corresponding history table as well, as otherwise we may run into versioning problems, if a table with the former name is created. It also helps with the readability as the original and the history table are always named the same. As no information can get lost in this event, and it is necessary that the renaming has already completed, the event trigger can only be fired after the transaction.

```
1  SELECT SPLIT_PART(SPLIT_PART(lower(query), lower('ALTER
   ↪  TABLE '), 2),lower(' RENAME TO'),1) into name;
2  SELECT SPLIT_PART(name, '.', 2) INTO orig_name;
3  SELECT SPLIT_PART(name, '.', 1) INTO schemaname;
4  SELECT SPLIT_PART(SPLIT_PART(lower(query), lower('RENAME TO
   ↪  '), 2),';',1) into new_name;
5  IF (orig_name not like '%_hist%') THEN
6      execute format('ALTER TABLE hist.%1$s_hist RENAME TO
       ↪  %2$s_hist', orig_name, new_name);
7      call adapt_triggers(orig_name, schemaname, new_name);
8
9      execute format('UPDATE query_simple SET
       ↪  re_execute_query = REPLACE(re_execute_query,
       ↪  ''hist.%1$s_hist'', ''hist.%2$s_hist'') WHERE
       ↪  re_execute_query like ''%%hist.%1$s%%'' ',
       ↪  orig_name,new_name);
10 end if;
```

Listing 19: Procedure to automatically rename the corresponding history table, if a table is renamed

In Listing 19 the renaming process can be seen. The table name, the schemaname and the new table name are extracted from the given query in Line 2, 3 and 4. These values are then used to apply the renaming to the history table as well in Line 6 and adapt the trigger in Line 7. In Line 9 all queries, which are using the history table, are rewritten, so that they use the renamed history table instead.

5.4.4 Add Column

When adding a column to a table, we want to add the column to the corresponding history table as well, so that future versions also include versions of this column. Additionally the versioning triggers need to be adapted, so that the values of the new column are copied to the history table as well. As information is added, it is necessary that the transaction has already completed, therefore the event trigger can only be fired after the transaction. To prevent problems with the re-execution, it is important to lock the table before the addition of a new column, as otherwise race conditions can occur, while the new column is added.

```
1 SELECT SPLIT_PART(SPLIT_PART(lower(query), lower('ALTER TABLE '), 2),lower(' ADD COLUMN '),1) into name;
2 SELECT SPLIT_PART(name, '.', 2) INTO orig_name;
3 SELECT SPLIT_PART(name, '.', 1) INTO schemaname;
4 SELECT SPLIT_PART(SPLIT_PART(lower(query), lower('ADD COLUMN '), 2),';',1) into columnname;
5
6 IF (orig_name not like '%_hist%') THEN
7     execute format('ALTER TABLE hist.%1$s_hist ADD COLUMN %2$s', orig_name, columnname);
8     call adapt_triggers(orig_name, schemaname,orig_name);
9 end if;
```

Listing 20: Procedure to automatically add a column to the corresponding history table and adapt the versioning triggers, if a column is added to the original table

In Listing 20 the add column process is illustrated. In Line 2 to 4 we are extracting the table name, schema name, and column name. In Line 8 we are then adding the additional column to the history table, and then in Line 9, we adapt the triggers so that they also include the new column.

5.4.5 Drop Column

When dropping a column, we want to keep the column in the corresponding history table, so that we can re-execute any query which uses the dropped column. Additionally, the versioning triggers need to be adapted, so that no errors occur because of a non-existing column.

```
1 SELECT SPLIT_PART(SPLIT_PART(lower(query), lower('ALTER TABLE '), 2),lower(' DROP COLUMN '),1) into name;
2 SELECT SPLIT_PART(name, '.', 2) INTO orig_name;
3 IF (orig_name not like '%_hist%') THEN
4     call adapt_triggers(orig_name, 'public',orig_name);
5     -- no rewriting necessary
6 end if;
7 end if;
```

Listing 21: Procedure to automatically adapt the versioning triggers, if a column is dropped

In Listing 21 we can see the drop column process illustrated. In Line 1 and 2 we extract the table name and adapt the triggers in Line 4. As the CRUD operations are performed by triggers, these triggers are updated to use the now active columns. This is done by reinitializing the functions displayed in Listing 9. We are not parsing the corresponding column, as the column is not necessary for the adaption of the history table.

5.4.6 Rename Column

When renaming a column, we want to rename the column in the corresponding history table as well, as otherwise, we may run into versioning problems, if a column with the former name is created. It also helps with the readability as the original and the history table have the same column names. As information is added, it is necessary that the transaction has already been completed, therefore the event trigger can only be fired after the transaction. When renaming columns, this can lead to problems in calculating the result hashes on re-execution, as the column-names differ. We therefore, did not use the column names for the hashing, but only all data points.

Another valid strategy is to assign an alias to all given columns of a SQL query and only replace the column name, and have the alias remain unchanged. This would then lead to the same hash, as the column names of the result set remain the same.

```
1  SELECT SPLIT_PART(SPLIT_PART(lower(query), lower('ALTER
     ↪  TABLE '), 2),lower(' RENAME '),1) into name;
2  SELECT SPLIT_PART(name, '.', 2) INTO orig_name;
3  SELECT SPLIT_PART(name, '.', 1) INTO schemaname;
4  SELECT SPLIT_PART(SPLIT_PART(lower(query), lower('RENAME '),
     ↪  2),';',1) into columnnames;
5  SELECT SPLIT_PART(lower(columnnames), lower(' TO'), 1) into
     ↪  orig_column;
6  SELECT SPLIT_PART(lower(columnnames), lower('TO '), 2) into
     ↪  new_column;
7  IF (orig_name not like '%_hist%') THEN
8      execute format('ALTER TABLE hist.%1$s_hist RENAME
         ↪  %2$s_hist', orig_name, columnnames);
9      call adapt_triggers(orig_name, schemaname,orig_name);
10     execute format('UPDATE query_simple SET
         ↪  re_execute_query = REPLACE(re_execute_query,
         ↪  ''%1$s'', ''%2$s'') WHERE re_execute_query like
         ↪  ''%%%1$s%%'' ', orig_column,new_column);
11 end if;
```

Listing 22: Procedure to automatically rename the column in the corresponding history table, if a table column is renamed

In Listing 22 we see the process of renaming a column illustrated. In Line 1-6 we are parsing table name, schema name, and the two column names. In Line 8 we are then altering the table and renaming the corresponding column in the history table, and are then adapting the triggers in Line 9 to include the renamed column.

In Line 10 we then rewrite the query in a limited way, by replacing the old column name with the new column name.

5.5 Adaptions to the Backend

As not the whole integration of our Dynamic Data Citation framework can run directly in PostgreSQL, it is necessary to make several adaptions to the backend to use our framework and combine the previous described parts. So we have to store queries and subqueries in the query store, enrich queries with timestamp information, as well as use strategies to prevent race conditions.

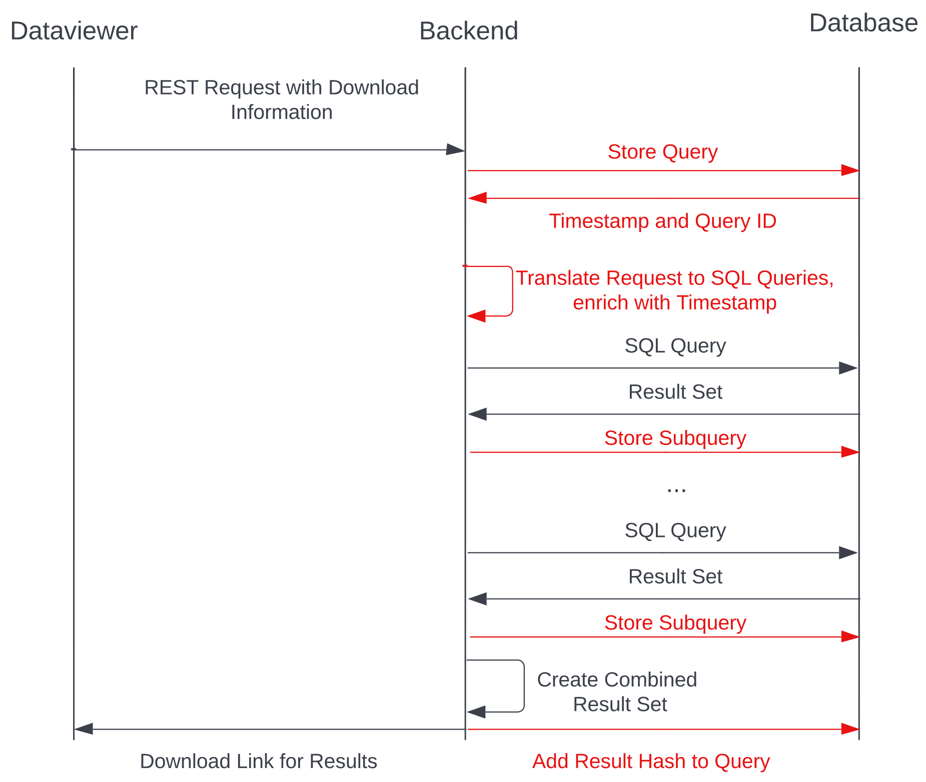


Figure 5.1: Illustration of the adapted download process for a user triggered download, with adaptions marked in red

We adapt the download process (seen in Figure 3.4) as illustrated in Figure 5.1. After the request is received in the backend, we are storing the request in the backend, and get timestamp information and a query id provided by the system. With this information we then rewrite our query to our subqueries and include timestamp information. Then all subqueries are executed and the result sets are returned. For each returned result set is then a SHA256 hash calculated for all datapoints, but without the columnnames. After all subqueries are executed, the combined result set is created and the user receives a download link for the results.

We will describe the adapted parts of the process below.

5.5.1 Storing Queries

For storing queries and subqueries we use the functions *save_query* and *save_subquery* in the database. At the start of the process the query is stored in the database, with the call displayed in Listing 23, and the execution timestamp and the id of the query is returned.

```
cur.execute(f"SELECT * FROM
↪   save_download_simple({user_id},'{request}')")
```

Listing 23: Storing a query in the database with a function

Then this information is used to rewrite the queries, by creating one re-executable subquery and one subquery to execute immediately. After the execution both queries are then stored in the query store by calling the procedure displayed in 24. The We before need to escape all single quotes, as SQL encoding is deleting unescaped quotes and calculate the hash of the result set.

```
cur.execute(f"CALL save_subquery('{original_query}'::text,
↪   '{reexecutable_query}'::text, {results_number},
↪   '{result_hash}'::text, {download_id})")
```

Listing 24: Storing a subquery in the database with a procedure

After all subqueries have concluded, and the final result set has been created, we calculate a hash for the result set and store it in the query store.

5.5.2 Hashing Result Sets

As mentioned before, we are using a SHA256 hash as our hashing function for our result sets as well as for the hashing of the subqueries. We implemented the hashing for the subqueries directly in SQL by using the built-in *SHA256()* function. For the result-set we hashed all returned tuples without headers, with the *hashlib.sha*256() function of python. In Listing 25 this process is displayed. We first rename all columns in our pandas dataframe with the rename function in line 1 to numbers, to prevent changes of hashes if columns are renamed. In Line 2 we then transform the dataset to a CSV and then hash the whole CSV file.

```
df.rename(columns={x:y for x,y in zip(df.columns
↪   ,range(0,len(df.columns)))})
hashlib.sha256(df.to_csv(sep=',').encode('utf-8'))
↪   .hexdigest()
```

Listing 25: Hashing the result set of a subquery

5.5.3 Concurrency control

As queries and updates occur simultaneously, it is important to use database concurrency control strategies, as described in Section 4.1.5, to handle them. We used primarily locking as strategy of choice, but also implemented the other two strategies as well.

```
LOCK TABLE dataset IN ACCESS EXCLUSIVE MODE;
```

Listing 26: Acquiring of an ACCESS EXCLUSIVE lock on the dataset table

We implemented locking on write operations with *ACCESS EXCLUSIVE* locks, as these operations can lead to inconsistent result sets, and use *ACCESS SHARE* locks for read operations in the database. The *ACCESS EXCLUSIVE* locks ensure that only the operation that holds this lock for a table, has the right to read or write a given table. In Listing 26, we can see how this lock can be obtained, after a transaction is started. The lock is held until the transaction concludes, which is for the ISMN update process after a batch of inserts is finished.

Regarding query backdating and data postdating, we implemented this behaviour with an offset of 15 minutes, as the daily updates normally conclude within ten minutes. In Listing 27 we see the data postdating, which adds 15 minutes to the *now()* function. This offset is also added to the *valid_to* column, which is set in update and delete triggers to avoid inconsistencies. For the subquery backdating, we adapted the *save_query* function, to return a timestamp that lies 15 minutes before the current time, which is then included in all re-executable, by using it for rewriting.

```
valid_from TIMESTAMP DEFAULT (now() + interval '15
↪   minutes')
```

Listing 27: Postdating the valid_from timestamp

5.6 Data Cite Standard

In a complementary project, the International Soil Moisture Network was extended to support the DataCite standard [5] and the minting of Digital Object Identifiers, while also integrating the following WGDC Guidelines[16]:

- R10 - Automated Citation Texts
- R11 - Landing Page
- R12 - Machine Actionability

The DataCite schema is a metadata schema that aims to make data citable, searchable, and accessible by enriching datasets with metadata. This is done by enriching each dataset, which the author wants to publish, with the associated metadata.

For the International Soil Moisture Network, this is done for each created subset of the database, which is created by querying and downloading the data. Each user triggered query is displayed on an overview page per user, with the time of generation and metadata about the requested data, as the time period which is downloaded. After the download is concluded, the user can view the auto-generated metadata on the landing page of a given query and add further metadata as a name identifier in the form of an ORCID[6] or an ISNI[7], the affiliation in form of a ROR[8] and a description of the downloaded dataset. The author can then request a DOI, by sending all metadata to the DOI-Service Austria[9], if the three before mentioned metadata fields are provided. Upon receiving the DOI, the DOI is stored with the corresponding query, and the Landing Page is made publicly available, to allow the view of the metadata and the re-creation of the original result set.

Without a user account, only the metadata can be viewed on the corresponding landing page or retrieved in JSON format, for machine actionability. The re-creation of the result set of a query on the other hand can only be requested if the user is logged in, to honor the agreements with the data providers to distribute the data only to registered users. The landing page further includes a generated citation text in APA format for the subset with the minted DOI.

[5] https://datacite.org/
[6] https://orcid.org/
[7] https://isni.org/
[8] https://ror.org/
[9] https://www.tuwien.at/bibliothek/doi-service-austria-orcid-austria

CHAPTER 6

Evaluation

For the evaluation, we are measuring our implementation in different ways. In Section 6.1 we are discussing to which extent our solution aligns with the RDA Recommendations for Data Citation. Then in Section 6.2 we will evaluate the storage increase trends on a small controllable database and will then go on and evaluate the storage increase, the execution time, and the query correctness on the International Soil Moisture Network in Section 6.3. We evaluate the storage consumption on two different databases, as optimizations performed by PostgreSQL are better controllable in a smaller database, and therefore we can show the trends of the storage increase better.

6.1 Accordance to RDA Dynamic Data Citation

To verify, that we have fulfilled all necessary recommendations for the International Soil Moisture Network, we will analyze if each of the recommendations was in scope, and how we have solved it. We will further provide functional test protocols.

6.1.1 R1 - Data Versioning

"Apply versioning to ensure earlier states of data sets can be retrieved"

This recommendation is fulfilled, as all data is versioned, as discussed in Section 5.1, by applying one of our versioning approaches (integrated, hybrid, separated) on the database. By filtering based on the two added timestamps *valid_from* and *valid_to* for each table, it is possible to retrieve earlier states of the database.

This is shown in the Test Cases displayed in Table 6.1, 6.2 and 6.3, with applying versioning to an initialized table, and verifying that the table is modified correctly and the corresponding history tables are generated.

Test name:	apply_versioning_integrated
Test case description:	Apply integrated versioning to the *dataset(id, time_utc,value)* table, which already contains 50,000 rows. Verify that the *dataset* table contains afterwards the *valid_from* and *valid_to* columns
Expected result:	*dataset* table contains 50,000 rows with 5 columns (including *valid_from* and *valid_to*) with all *valid_from* columns set to the current time and triggers created to handle update and delete operations
Actual result:	*dataset* table contains 50,000 rows with 5 columns (including *valid_from* and *valid_to*) with all *valid_from* columns set to the current time and triggers created to handle update and delete operations
Test passed:	true

Table 6.1: Test for applying integrated versioning to the dataset table

Test name:	apply_versioning_separated
Test case description:	Apply separated versioning to the *dataset(id, time_utc,value)* table, which already contains 50,000 rows. Verify that the *dataset_hist* table is created with 50,000 rows and contains the *valid_from* and *valid_to* columns
Expected result:	Table *dataset_hist* is created with 50,000 rows and 5 columns (including *valid_from* and *valid_to*) with all *valid_from* columns set to the current time and triggers to synchronize *dataset* and *dataset_hist* table
Actual result:	Table *dataset_hist* is created with 50,000 rows and 5 columns (including *valid_from* and *valid_to*) with all *valid_from* columns set to the current time and triggers to synchronize *dataset* and *dataset_hist* table
Test passed:	true

Table 6.2: Test for applying separated versioning to the dataset table

6.1.2 R2 - Timestamping

"Ensure that operations on data are timestamped, i.e. any additions, deletions are marked with a timestamp"

This recommendation is fulfilled, as all insert, update and delete operations are automatically timestamped. To each table in the database, a validity period is added, which consists of two timestamps. The *valid_from* timestamp is set on any inserts into the database, and the *valid_to* timestamp is only set, when a tuple is deleted. For any update, a delete and an insert are performed, as described in Section 5.1.

Test name:	apply_versioning_hybrid
Test case description:	Apply hybrid versioning to the *dataset(id, time_utc,value)* table, which already contains 50,000 rows. Verify that the dataset table contains afterwards the *valid_from* column and a *dataset_hist* table is created
Expected result:	dataset table contains 50,000 rows with 5 columns (including *valid_from*) with all *valid_from* columns set to the current time and an empty *dataset_hist* table with *valid_from* and *valid_to* columns and triggers to handle update and delete operations
Actual result:	dataset table contains 50,000 rows with 5 columns (including *valid_from*) with all *valid_from* columns set to the current time and an empty *dataset_hist* table with *valid_from* and *valid_to* columns and triggers to handle update and delete operations
Test passed:	true

Table 6.3: Test for applying hybrid versioning to the dataset table

This is shown in the Test Cases for the integrated versioning approach displayed in Table 6.4, 6.5 and 6.6. Each of these tests builds on the test before and starts with an empty *dataset* table, and then verifies the functionality of the versioning. First, a tuple is inserted into the empty *dataset* table with integrated versioning applied to, then the tuple is updated with a new value, and in the last test case, it is then deleted. This process is then repeated for the separated versioning approach (see Table 6.7, 6.8 and 6.9) and for the hybrid versioning approach (see Table 6.10, 6.11 and 6.12).

Test name:	add_tuple_integrated
Test case description:	Add a tuple to the empty *dataset* table, which has already integrated versioning applied to and then query the table.
Expected result:	id: 1, time_utc: '2023-04-29 14:00:00:00', value: 6, valid_from: '2023-04-29 13:53:59.493528', valid_to: null
Actual result:	id: 1, time_utc: '2023-04-29 14:00:00:00', value: 6, valid_from: '2023-04-29 13:53:59.493528', valid_to: null
Test passed:	true

Table 6.4: Test for adding a tuple to a table with already applied integrated versioning

6.1.3 R3 - Query Store Facilities

"Provide means for storing queries and the associated metadata in order to re-execute them in the future"

This recommendation is fulfilled, as we implemented a query store, which is described in

Test name:	update_tuple_integrated
Test case description:	Update the *value* column to 7 for the tuple with *id=1* and *valid_to=null* in the *dataset* table, then count all tuples with *id=1* and verify that the timestamps are set correctly
Expected result:	2
Actual result:	2
Test passed:	true

Table 6.5: Test for updating a tuple to a table with already applied integrated versioning

Test name:	delete_tuple_integrated
Test case description:	Delete the tuple with *id=1* and *valid_to=null* in the *dataset* table, then count all tuples with *id=1* and *valid_to!=null* and verify that the timestamps are set correctly
Expected result:	2
Actual result:	2
Test passed:	true

Table 6.6: Test for deleting a tuple with already applied integrated versioning

Test name:	add_tuple_separated
Test case description:	Add a tuple to the empty *dataset* table, which has already separated versioning applied to and then query the corresponding history table.
Expected result:	id: 1, time_utc: '2023-04-29 14:00:00:00', value: 6, valid_from: '2023-04-29 13:54:38.482957', valid_to: null
Actual result:	id: 1, time_utc: '2023-04-29 14:00:00:00', value: 6, valid_from: '2023-04-29 13:54:38.482957', valid_to: null
Test passed:	true

Table 6.7: Test for adding a tuple to a table with already applied separated versioning

Test name:	update_tuple_separated
Test case description:	Update the *value* column to 7 for the tuple with *id=1* in the *dataset* table, then count all tuples with *id=1* in the *dataset_hist* table and verify that the timestamps are set correctly
Expected result:	2
Actual result:	2
Test passed:	true

Table 6.8: Test for updating a tuple in a table with already applied separated versioning

Test name:	delete_tuple_separated
Test case description:	Delete the tuple with *id=1* in the *dataset* table, then count all tuples with *id=1* and *valid_to != null* in the *dataset_hist* table and verify that the timestamps are set correctly
Expected result:	2
Actual result:	2
Test passed:	true

Table 6.9: Test for deleting a tuple with already applied separated versioning

Test name:	add_tuple_hybrid
Test case description:	Add a tuple to the empty *dataset* table, which has already hybrid versioning applied to and then query the *dataset* table.
Expected result:	id: 1, time_utc: '2023-04-29 14:00:00:00', value: 6, valid_from: '2023-04-29 13:55:23.123554'
Actual result:	id: 1, time_utc: '2023-04-29 14:00:00:00', value: 6, valid_from: '2023-04-29 13:55:23.123554'
Test passed:	true

Table 6.10: Test for adding a tuple to a table with already applied hybrid versioning

Test name:	update_tuple_hybrid
Test case description:	Update the *value* column to 7 for the tuple with *id=1* in the *dataset* table, then count all tuples with *id=1* in the *dataset_hist* table and verify that the *valid_to* timestamp is set correctly
Expected result:	1
Actual result:	1
Test passed:	true

Table 6.11: Test for updating a tuple in a table with already applied hybrid versioning

Test name:	delete_tuple_hybrid
Test case description:	Delete the tuple with *id=1* in the *dataset* table, then count all tuples with *id=1* in the *dataset_hist* table and verify that the *valid_to* timestamp is set correctly
Expected result:	2
Actual result:	2
Test passed:	true

Table 6.12: Test for deleting a tuple with already applied hybrid versioning

detail in Section 5.2. The query store for our use case consists of two tables, one which holds information about user triggered queries, and one which holds all subqueries with information on how to re-execute these queries. A single execution timestamp at the request start is created for the query and all corresponding subqueries.

We verified this recommendation in our functional tests displayed in Table 6.13 for storing a query and receiving an id and a timestamp and in Table 6.14 for storing a subquery with the associated metadata. We shortened the queries and hashes for readability purposes.

Test name:	store_query
Test case description:	Store the query in the database by using the *store_query* function, and then check if the result is a valid query_id and the execution timestamp
Expected result:	id: 1, timestamp: '2023-04-30 07:12:58.459381'
Actual result:	id: 1, timestamp: '2023-04-30 07:12:58.459381'
Test passed:	true

Table 6.13: Test for storing the initial query in the querystore and obtain id and execution timestamp

Test name:	store_subquery
Test case description:	Store a subquery in the database by using the *store_subquery* function and verify that all fields are correctly set, by querying for the subquery
Expected result:	id: 1, original_subquery:'SELECT ...', reexecutable_subquery:'SELECT ...', subquery_hash:'3e73e60b...', result_hash:'7275de85...', result_nr:10000
Actual result:	id: 1, original_subquery:'SELECT ...', reexecutable_subquery:'SELECT ...', subquery_hash:'3e73e60b...', result_hash:'7275de85...', result_nr:10000
Test passed:	true

Table 6.14: Test for storing a subquery in the querystore

6.1.4 R4 - Query Uniqueness

"Re-write the query to a normalized form so that identical queries can be detected. Compute a checksum of the normalized query to efficiently detect identical queries"

This recommendation is not fulfilled, as it is not in the scope of this thesis. Queries by users of the International Soil Moisture Network are not normalized and compared. The uniqueness of subqueries on the other hand is guaranteed by design, as the user cannot alter the SQL queries, and the same queries, with different parameter settings are executed.

6.1.5 R5 - Stable Sorting

"Ensure that the sorting of the records in the data set is unambiguous and reproducible"

This recommendation is fulfilled, as the query is ordered in the backend based on all primary keys. Any new query introduced to the system needs to order by the primary key, so that an unambiguous ordering is guaranteed. Therefore each executed query, which is run during the download has a *ORDER BY* clause.

6.1.6 R6 - Result Set Verification

"Compute fixity information (checksum) of the query result set to enable verification of the correctness of a result upon re-execution"

The result verification is possible, by comparing the hash of the re-execution result set with the hash of the original result set. This hash is generated with a pandas dataframe, by renaming the column headers first and then transforming it to a CSV file and hashing the whole CSV file with the SHA256 library. Therefore, this recommendation is fulfilled.

We verified this recommendation by re-executing all subqueries for a download request and calculating the SHA256 hash and comparing it with the stored SHA256 hash. Our test is displayed in Table 6.15

Test name:	verify_correctness
Test case description:	Re-execute all stored subqueries for a download request, compare the hash of the result set with the stored hash, and print the query and the hash if the hash differs. The result should be the empty set.
Expected result:	{}
Actual result:	{}
Test passed:	true

Table 6.15: Test for re-executing all subqueries of a query and verifying all hashes

6.1.7 R7 - Query Timestamping

"Assign a timestamp to the query based on the last update to the entire database (or the last update to the selection of data affected by the query or the query execution time). This allows retrieving the data as it existed at the time a user issued a query"

Each query gets a timestamp assigned at the time of requesting data. This timestamp is the time of execution, and is used for query rewriting, so that each query can be re-executed with the same data, which was present at the time the user executed the query. Therefore, this recommendation is fulfilled.

The retrieval of a timestamp for a query is displayed in Table 6.13, which is then used to rewrite the queries, which is tested in Table 6.16. As this rewrite is done in the backend, all single quotes are doubled, as the PostgreSQL database escapes them.

Test name:	rewrite_query
Test case description:	Rewrite a subquery with the query-execution timestamp of the linked query and verify the syntactical correctness of the subquery
Expected result:	SELECT s.station_id, s.station_name, s.network_abbr, round(CAST(ST_X(station_location) AS NUMERIC),5), round(CAST(ST_Y(station_location) AS NUMERIC),5), ST_Z(station_location) FROM (SELECT s.station_id, s.station_name, s.network_abbr, station_location FROM hist.station_hist WHERE valid_from < "07:12:58.459381" AND (valid_to IS null OR valid_to > "07:12:58.459381")) as s WHERE s.network_abbr = "ARM" ORDER BY station_id;
Actual result:	SELECT s.station_id, s.station_name, s.network_abbr, round(CAST(ST_X(station_location) AS NUMERIC),5), round(CAST(ST_Y(station_location) AS NUMERIC),5), ST_Z(station_location) FROM (SELECT s.station_id, s.station_name, s.network_abbr, station_location FROM hist.station_hist WHERE valid_from < "07:12:58.459381" AND (valid_to IS null OR valid_to > "07:12:58.459381")) as s WHERE s.network_abbr = "ARM" ORDER BY station_id;
Test passed:	true

Table 6.16: Test for rewriting a subquery for re-execution

6.1.8 R8 - Assigning Query PID

"Assign a new PID to the query if either the query is new or if the result set returned from an earlier identical query is different due to changes in the data. Otherwise, return the existing PID"

This recommendation is not fulfilled as the query is private by design, and each user can request a PID for all started downloads and afterwards the subset is publicly available. In the future, queries with the same result set should get the same PID assigned.

6.1.9 R9 - Store the Query

"Store query and metadata (e.g. PID, original and normalized query, query & result set checksum, timestamp, superset PID, data set description, and other) in the query store"

The complementary project stores data about the involved networks and information according to DataCite standard[1] and we are storing all necessary data about the subset in the query store, and can link this information with the DataCite information.

6.1.10 R10 - Automated Citation Text Generation

"Generate citation texts in the format prevalent in the designated community for lowering the barrier for citing the data. Include the PID into the citation text snippet"

This recommendation is not in the scope of this thesis, but was implemented in a complementary project and automatically provides the user with a citation in text format, as this is the standard citation format for the community.

6.1.11 R11 - Landing Pages

"Make the PIDs resolve to a human readable landing page that provides the data (via query re-execution) and metadata, including a link to the superset (PID of the data source) and citation text snippet"

This recommendation is not in the scope of this thesis, but was implemented in a complementary project. All downloads are listed per user, and additionally for all published subsets. Upon clicking on a not yet published download, the user can add additional metadata and then mint a DOI for the subset. Upon publication, the landing page automatically links to all the corresponding metadata and allows registered users to download the published results.

6.1.12 R12 - Machine Actionability

"Provide an API / machine actionable landing page to access metadata and data via query re-execution."

This recommendation is not in the scope of this thesis, but was implemented in a complementary project. All the metadata for a download can be also retrieved in JSON format, so that it is machine actionable.

6.1.13 R13 - Technology Migration

"When data is migrated to a new representation (e.g. new database system, a new schema or a completely different technology), migrate also the queries and associated fixity information"

This recommendation is fulfilled for some common schema changes, but not for the migration to different database systems. These schema changes are automatically handled by event triggers and rewriting queries and triggers whenever necessary. Further information about schema changes can be found in Section 2.4 and Section 5.4.

[1]`https://datacite.org/`

In Table 6.17 and Table 6.18, we can see the test result that columns can be added and deleted and the versioning is not affected by it.

Test name:	add_column_separated
Test case description:	Add the column *new_value* to the dataset table, and then insert and update the *new_value* column of the tuple with *id=1* with the value 4. The history table should have 2 queries stored
Expected result:	2
Actual result:	2
Test passed:	true

Table 6.17: Test for adding a column to the dataset with separated versioning applied to

Test name:	drop_column_separated
Test case description:	Drop the column *new_value* from the dataset table, verify that the column still exists in the history table with two different values for *id=1*
Expected result:	id: 1, time_utc: '2023-04-29 14:00:00:00', value: 6, valid_from: '2023-05-01 04:53:18.285730', valid_to: '2023-05-01 04:54:24.491915' id: 1, time_utc: '2023-04-29 14:00:00:00', value: 4, valid_from: '2023-05-01 04:54:24.491915', valid_to: null
Actual result:	id: 1, time_utc: '2023-04-29 14:00:00:00', value: 6, valid_from: '2023-05-01 04:53:18.285730', valid_to: '2023-05-01 04:54:24.491915' id: 1, time_utc: '2023-04-29 14:00:00:00', value: 4, valid_from: '2023-05-01 04:54:24.491915', valid_to: null
Test passed:	true

Table 6.18: Test for dropping a column from the dataset table with separated versioning applied to

6.1.14 R14 - Migration Verification

"Verify successful data and query migration, ensuring that queries can be re-executed correctly"

The successful migration of the data to a new schema and the re-execution can be verified by rerunning all rewritten queries in the query store and verifying the obtained results. Be aware, that the results in the current implementation must be hashed without the headers, to allow the verification of the results, as otherwise different column or table names lead to a wrong hash. Otherwise, this recommendation is fulfilled.

	unversioned	integrated	hybrid	separated
Original size	58.824	66.672	66.672	58.824
history size	0	0	0	66.672
Total	58.824	66.672	66.672	125.495
Percentage	100	113.33	113.33	213.33

Table 6.19: Table sizes after 1,000,000 inserts in MB and relative to the unversioned approach

6.2 Trend Analysis

To analyze the best storage trends for each of the versioning approaches, we are using four similar tables, apply the three different versioning approaches to them and keep one unversioned table as a baseline. The table consists of the three columns *id*, *time_utc*, and *value*. In this setting, we will not evaluate additional indexes or complex primary key settings, as we will further evaluate the results on the ISMN database in the next section. In each of the three storage experiments, 10^6 tuples will be inserted, updated, or deleted and after each 50,000 written tuples (the daily update size of one large network), we will use PostgreSQLs database compression to free up unused storage space and measure the storage impact of each of the different versioning approaches. This compression is called by executing *VACUUM FULL;* and reclaims storage occupied by dead tuples[2].

We expect the storage usage to be according to the following formula:

$$storage(Separated) > storage(Integrated) \geq storage(Hybrid) > storage(Unversioned)$$

6.2.1 Insert 1,000,000 tuples

During the insert measurement, we inserted tuples in batches of 50,000 elements and measured the storage consumption after each batch.

In Table 6.19, we see the sizes after the last iteration. We can see that the hybrid and the integrated approach need the same amount of storage. This is most likely due to the optimizations of PostgreSQL, which cuts columns that only consist of Null values, as we are looking at the minimal storage consumption. Therefore the *valid_to* column of the integrated approach does not cause a storage overhead.

The separated approach on the other hand needs more than double the amount of the unversioned approach. This is because it stores an unversioned table and a table with integrated versioning. From a storage consumption in percent, this means, that both the Integrated and Hybrid approach have an overhead of at least 13.33% and the separated approach has an overhead of 213.33%.

[2]`https://www.postgresql.org/docs/current/sql-vacuum.html`

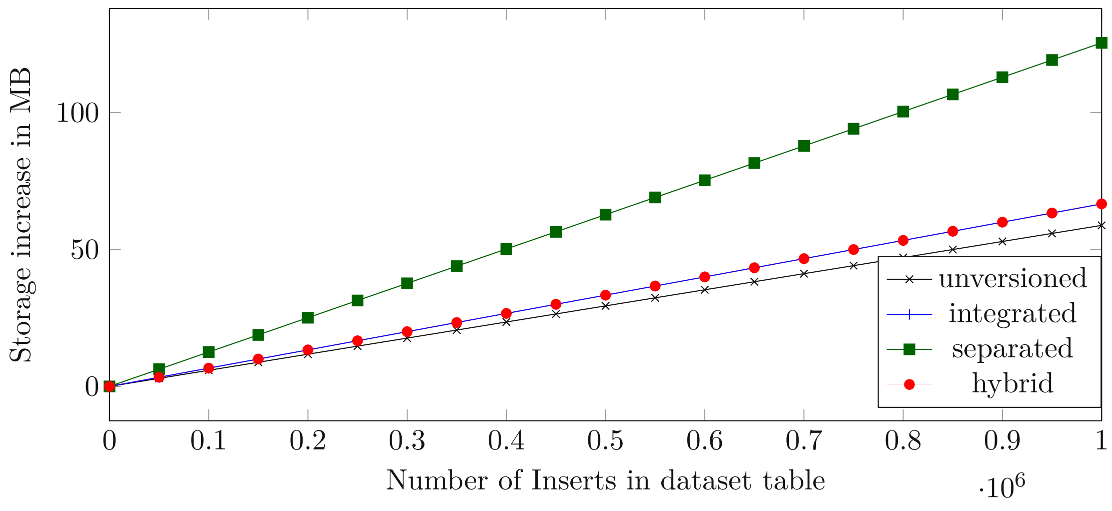


Figure 6.1: Storage increase in MB per variant for INSERT

	unversioned	**integrated**	**hybrid**	**separated**
Original size	58.824	141.6	66.672	58.824
history size	0	0	74.768	141.6
Total	58.824	141.6	141.44	200.424
Percentage	100	240.72	240.45	340.72

Table 6.20: Table sizes after 1,000,000 Updates in MB and relative to the unversioned approach

If we look at the storage increase over 20 iterations, displayed in Figure 6.1, we can see a linear trend for all different versioning approaches and the unversioned table. The results of the storage trends are similar to our assumption and are increasing constantly.

6.2.2 Update 1,000,000 tuples

During the update measurement, we updated tuples in batches of 50,000 elements, and measured the storage consumption after each batch. As PostgreSQL does not free storage occupied by dead tuples, we removed them with using the *VACUUM FULL;* command. The Update test took place after the insert test, and therefore there are 1 million tuples in all tables.

In Table 6.20, we see the sizes after the last update iteration. We can see that the hybrid and the integrated approach differ by about 160 KB. Based on our results for the inserts, these values are anticipated, as PostgreSQL is optimizing the null values and both tables store the same data.

The storage for the unversioned table stays the same overall updates, when comparing it to the last iteration of INSERTs, as no null values exist in the database. The integrated and hybrid approaches increase when all tuples have been updated once, for around 127%, while the separated approach increases to around 227%.

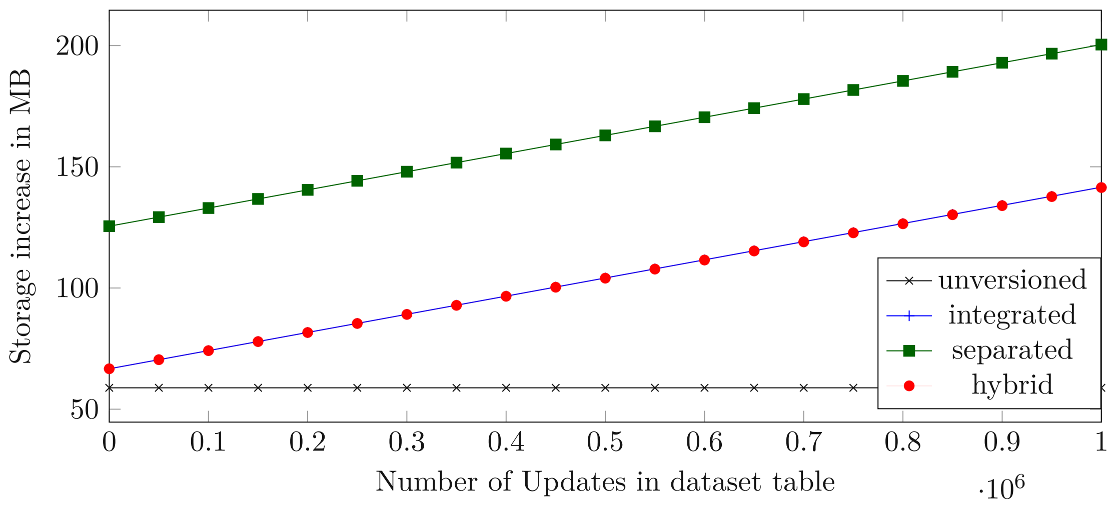


Figure 6.2: Storage increase in MB per variant for UPDATE

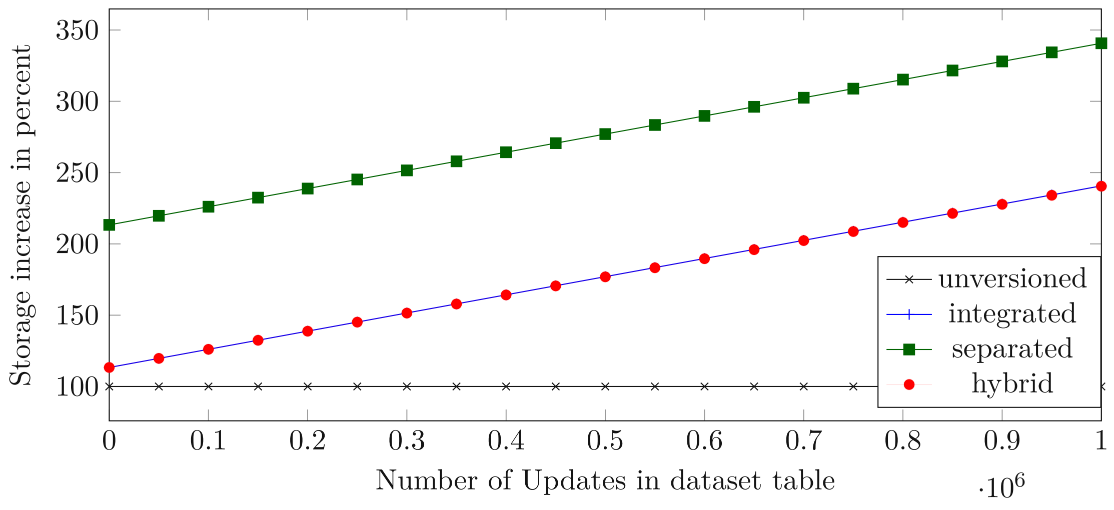


Figure 6.3: Storage increase in percent per variant for UPDATE

If we look at the storage increase over 20 iterations, displayed in Figure 6.2 in MB and in Figure 6.3 in percent of the unversioned table, we can see a linear trend for the three different versioning approaches, while the unversioned table remains the same. We can see that the storage increases roughly 20% for all 200,000 inserted rows for all three different versioning approaches. The results of the storage trends are similar to our assumption.

	unversioned	**integrated**	**hybrid**	**separated**
Original size	0.008	149.536	0.008	0.008
history size	0	0	149.536	149.536
Total	0.08	149.536	149.544	149.544
Percentage	0.01	254.21	254.22	254.22

Table 6.21: Table sizes after 1,000,000 Deletes in MB and relative to the unversioned approach before deletions

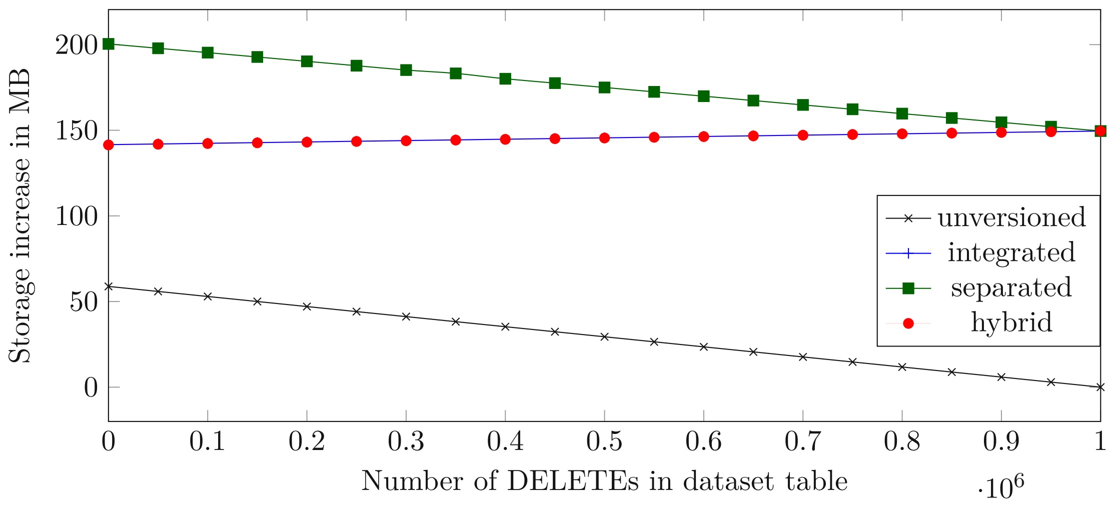


Figure 6.4: Storage increase in MB per variant for DELETE

6.2.3 Delete 1,000,000 tuples

During the delete measurements, we deleted tuples in batches of 50,000 elements, and measured the storage consumption after each batch. The Delete test took place after all the tuples had been inserted and updated once, therefore there are 2 million tuples in the versioned tables and 1 million tuples in the unversioned table.

In Table 6.21 we can see the final sizes after all tuples were deleted, and the relative size to the unversioned approach before deletions. We can see, that all three versioning approaches have a similar storage consumption, as all are only keeping the deleted tuples and no other tuples.

In Figure 6.4 we can see the storage trends of all different approaches. The unversioned table decreases steadily, to 0 MB, similar to the separated approach, which also deletes it unversioned table, but adds timestamp information to its history table. The Integrated and the hybrid approach are both increasing slightly, as timestamp information is added per iteration.

Nr.	Name	Inserted Files	Unique Rows
1	ARM	72.276	1.467.274
2	FMI	4850	442.888
3	KIHS_CMC	672	1.812.856
4	KIHS_SMC	531	1.268.592
5	RSMN	995	318.113
6	SCAN	279.091	6.798.302
7	SNOTEL	468.699	11.081.854
8	TAHMO	249	413.733
9	USCRN	195495	5.068.401
10	WEGENERNET	7972	208.367
11	FMI reprocess	132	172.908

Table 6.22: Different Networks, which were successively inserted into the database

6.3 International Soil Moisture Network

For the evaluation of the ISMN database, we used a server of the GEO department of TU Wien with 64 CPU cores, 252 GB of RAM, and PostgreSQL 14.6. All our tests are run against an International Soil Moisture Network dump, with over $1.4 \cdot 10^9$ tuples, and a size of over 250 GB (varies slightly as the database is not always freeing up unused storage space). The schema consists of 95 entities, with the largest one being the dataset table, which alone has $1.3 \cdot 10^9$ tuples and needs 123 GB of storage space, which holds most of the data of the database.

After restoring the database, we are either applying one of the different versioning approaches (see Section 4.1 and Section 5.1) or our baseline of no versioning and then updating the database with the normal inserting and updating procedures of the ISMN database. During this process, we will insert a total of 29,273,976 tuples and update 615,707 tuples. This data is not fully representative, as it does only include quality information for the FMI network, but reflects in general three to four months of new data.

These updates are different in their size and are shown together with their respective networks, in Table 6.22. The inserts are processed in alphabetical order, and then an update of the quality information of the FMI network is performed as the last iteration. The network updates vary between 172,908 processed rows (FMI reprocess), and 11,081,854 processed rows (SNOTEL) and are first preprocessed in the backend and then inserted in the database via batch files. The networks vary in size as the period of time that is updated in a single batch differs, and also the number of measured depth levels (the level at which a sensor is measuring soil moisture) differs. In Table 6.23 we can see for which time frame measurements were available, how many stations a network has, and how many different depth levels they are measuring. We can here see that, SCAN and SNOTEL measure 25 and 16 different depth levels respectively at 239 and 460

Nr.	Name	Startdate	Enddate	Stations	Depth Levels
1	ARM	2022-04-01	2022-05-01	35	14
2	FMI	2022-05-31	2022-11-01	27	7
3	KIHS_CMC	2022-08-30	2022-08-30	18	6
4	KIHS_SMC	2022-08-30	2022-08-31	19	6
5	RSMN	2022-05-05	2022-11-05	20	1
6	SCAN	2022-05-30	2022-11-20	239	25
7	SNOTEL	2022-06-01	2022-11-20	460	16
8	TAHMO	2022-07-12	2022-07-12	70	6
9	USCRN	2022-05-31	2022-11-19	115	5
10	WEGENERNET	2022-05-30	2022-11-24	12	2
11	FMI reprocess	2022-05-31	2022-11-01	27	7

Table 6.23: Different Networks, with the number of stations and the number of depths measurements per station

different locations, whereas WEGENERNET only measures 2 depth levels at 12 stations. More in detail description of the different networks used is available online[3].

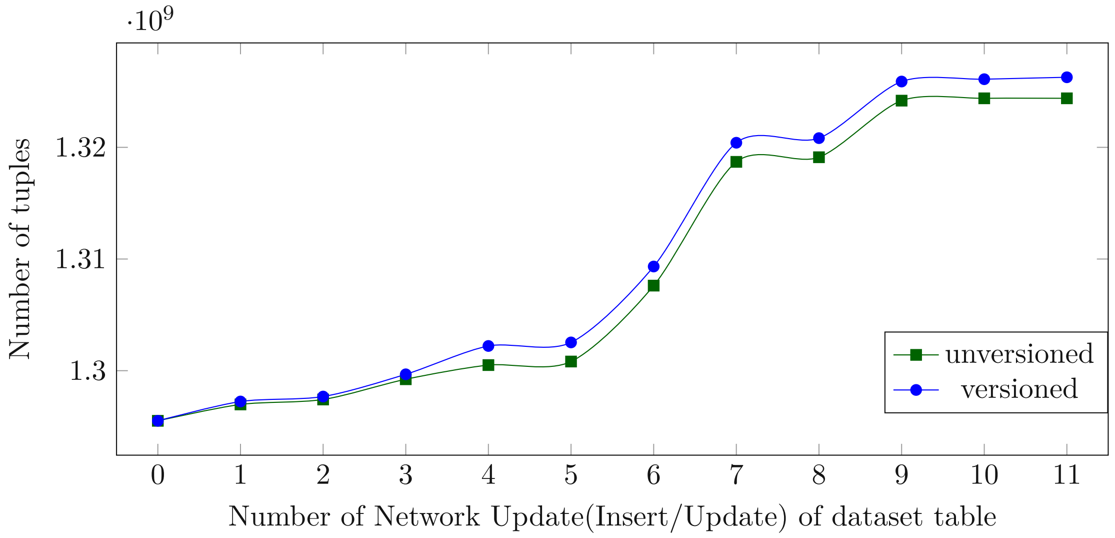


Figure 6.5: Number of tuples per network update of dataset table

In Figure 6.5 and Figure 6.6 we can see the number of active tuples in the dataset table, and the number of inserted and updated rows per network update. The number of rows increases from around $1.295 \cdot 10^9$ to around $1.325 \cdot 10^9$ tuples, while only 615,707 rows are updated. In Figure 6.6 we can see the difference between updated and inserted tuples,

[3]https://ismn.earth/en/networks/

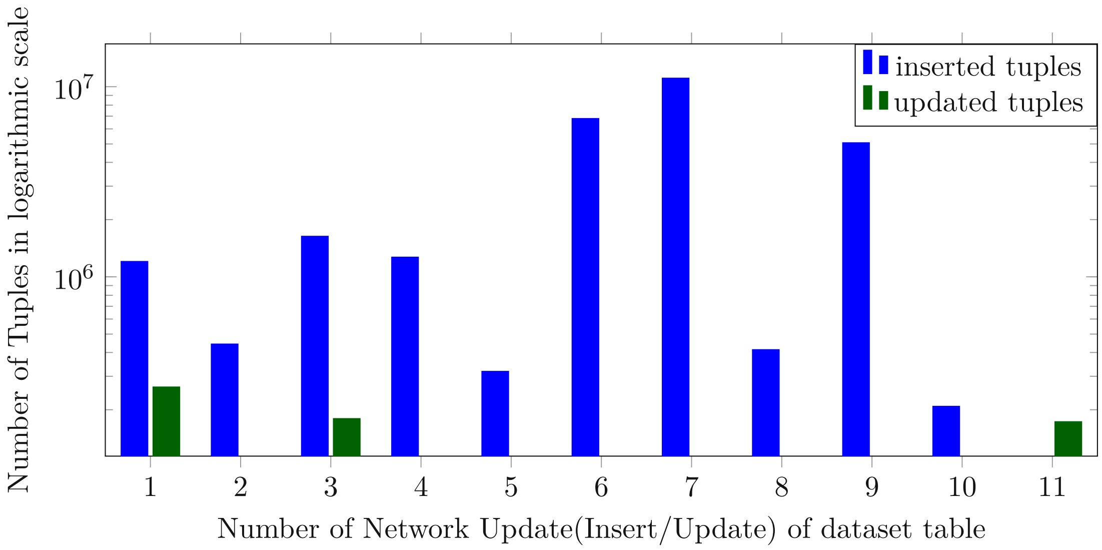


Figure 6.6: Inserted vs Updated Tuples per Network Update

as the networks 6 (SCAN) and 7 (SNOTEL) have updates of 6.7 million and 11 million tuples respectively, and the smallest networks only consist of roughly 200,000 rows.

For the evaluation of the different versioning approaches and the query store, we are following the process below:

1. Restore the database
2. Apply versioning
3. Measure Query Performance
4. For each network
   4.1 Measure Update Time
   4.2 Measure Size & Tuple Increase
   4.3 Measure Query Performance
   4.4 Verify Query Correctness

We measured the time for each update, but as the data was not representative, as it varied heavily based on the usage of the server, we could not obtain reliable update times for the database and will not report these values. Similarly, we obtained heavily affected query execution times, which differed up to 4 minutes between updates, and are therefore only reporting the average results over all runs, to allow the comparison of runtimes between the different versioning approaches, but will not show a trend analysis for the ISMN database.

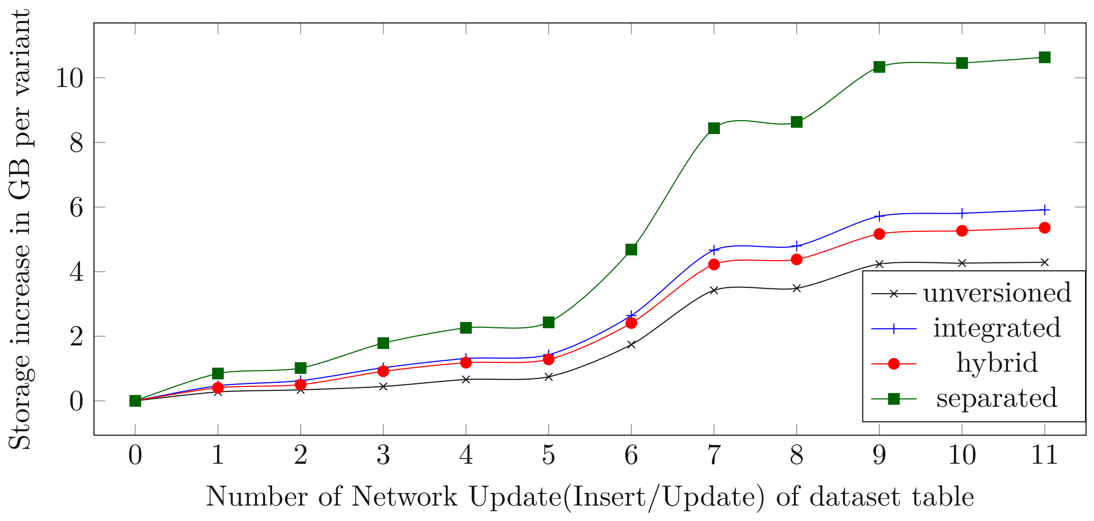


Figure 6.7: Storage increase in GB per variant, relative to starting size of variant

6.3.1 Storage Performance

To measure the storage overhead of versioning for the ISMN, we measured the database after each network update. After applying the versioning to the database, we removed storage occupied by dead tuples, by using *VACUUM FULL;*. This reduced the storage overhead of the integrated and hybrid versioning approaches by 100% of the storage consumption, as the tables are copied to a new location on the hard drive, when altered but the old space is not freed up and counts as used disk space. As the minimization process takes between one and two hours, depending on server load, and further locks all tables exclusively it is not feasible for a real-world scenarios. We measured the storage space once for each table, and once for all database objects in total.

	unversioned	**integrated**	**hybrid**	**separated**
database size	273.334	296.665	284.679	473.556
percentage	100	108.53	104.15	173.25

Table 6.24: Database storage comparison of the different versioned databases in GB and percent

In Table 6.24 we can see the database sizes after the final network update in GB and relative to the unversioned approach. The results differ from the anticipated results, as the separated approach does not need 200% of the unversioned database, and the integrated and separated approach show better values than in the optimized example for one table. This is most likely to database artifacts, such as indexes and primary keys, which seem not to be shrunken to the minimal size and are not mirrored in the history table if not necessary.

In Figure 6.7 we see the cumulative storage increase overall 11 network updates. The unversioned database increases in total by 4.29 GB, while the integrated databases increase by 5.913 GB, the hybrid for 5.36 GB and the separated for 10.636 GB. These measurements are all similar to the results we obtained in Section 6.2, but most likely need more storage for saving additional information, as updated indexes and not optimized tables.

6.3.2 Runtime Performance

As mentioned before the reported runtimes are gathered over different runs on the ISMN database server, but are heavily dependent on the server usage and server configuration. Therefore, we are not reporting the values for each run, but instead are reporting the minimum, maximum and average runtimes.

In Table 6.25 we can see the runtimes for one query, that consists of 13,670 subqueries, which represents a query for all available data for the networks displayed in Table 6.23. We can see that the hybrid versioning approach has a similar performance as the unversioned database with around 15 minutes per query, while the integrated and the separated approaches are both slower and are at 17:37 minutes and 17:11 minutes respectively. The performance impact on the separated query, which executes the same subqueries as the unversioned and the hybrid database might be due to the increased database size, which could increase the runtime.

	unversioned	**integrated**	**hybrid**	**separated**
average runtime	15:09 min	17:37 min	15:06 min	17:11 min
min runtime	14:31 min	17:05 min	14:32 min	16:41 min
max runtime	15:50 min	18:31 min	15:37 min	17:38 min

Table 6.25: Runtime for query execution with 10 parallel threads with 25 measurements

	integrated	**hybrid**	**separated**
average runtime	1:23 min	19:02 min	1:23 min
min runtime	1:22 min	17:29 min	1:26 min
max runtime	1:26 min	22:41 min	1:30 min

Table 6.26: Runtime for query re-execution with 200 parallel threads with 25 measurements

In Table 6.26 we can see the runtime for re-execution queries. These queries run with 200 threads, to decrease the evaluation time. As the subqueries in the execution process are executed per network, as the query needs to be translated into different subqueries, the execution process cannot run as parallelized as the re-execution. For the re-execution no translation of the query to subqueries is necessary, as all are already stored in the *subquery* table and they can be executed without constraints. Therefore these values cannot be compared with each other.

We can see that the runtimes for the integrated and the separated approach are nearly equal, with just a few seconds difference between these two. On the other hand is the enormously long re-execution process for the hybrid approach, which is most likely due to the computationally heavy combination of results of two distinct tables. This is because for each table used in the original query, the re-executable query needs to query 2^n tables, with n being the number of tables used in the select query, as the joins need to be performed for the original table and the history table.

6.3.3 Querystore Size

As the International Soil Moisture Network did not keep track of downloads until recently, we will assume that each user downloads the full database once a year. This would lead to 1600 download queries a year, or around 150 queries a month. For each of these queries 30,000 subqueries would be stored as well, with the associated meta information. As the size of both the original and the re-executed query is around 1 KB each, we estimate the size for one query with all corresponding subqueries at 60 MB. We then created the queries for 100 downloads and extrapolated the download size to measure the yearly impact.

In Table 6.27, we see that the storage per query is in the simple approach at 58.4 MB and at 10.2 MB for the advanced querystore. Which would lead to respective yearly storage needs of 93.7 GB and 16.2 GB. Here it is important to note, that the advanced querystore is computationally expensive, as the queries need to be reconstructed from two tables with joins and string replacements, whereas the queries for the simple querystore can just retrieve the queries without processing from the querystore.

	simple	**advanced**
max. size per download	58.4 MB	10.2 MB
size per year	93.7 GB	16.2 GB

Table 6.27: Download Size for the simple and advanced query store per query and per year

We advised the ISMN team to delete all queries without an issued DOI after a year, as the ISMN would store 48 million subqueries, if each user downloaded the full dataset once.

6.3.4 Query Correctness

To evaluate the query correctness, we conducted a test, displayed in Table 6.15, in which each executed download for each iteration was stored in the query store and one download of a previous network update with 13,670 subqueries was re-executed per network update. For each query in the download, the hash was generated twice, once on the execution and once on the re-execution, to allow the comparison between both query-results. Furthermore, to investigate if the results are different, also the number of results for both queries is returned.

Overall all 11 iterations for all 3 different versioning approaches, not a single query result has differed from the original result, when calculating the hash with the python library *hashlib* overall results without column names.

CHAPTER 7

Conclusion

We presented a novel Dynamic Data Citation framework for PostgreSQL according to the RDA Dynamic Data Citation Recommendations, which supports three different versioning approaches, storing queries with associated metadata and supporting the most common schema adaptions automatically. This framework has been adapted for the International Soil Moisture Network, to support their unique use case of executing and re-executing large numbers of SQL queries for one download request and has also been evaluated on their database. In the evaluation, we compared the three different versioning approaches on a small sample database and on the whole ISMN network with over $1.4 \cdot 10^9$ tuples.

We especially investigated the impact on the performance and the storage space of a database, when allowing the precise and persistent identification of arbitrary subsets. From the results of our evaluation, the execution time for queries executed with applied integrated versioning in the International Soil Moisture Network increased by around 17%, while it increased by 14% for the separated versioning approach and not at all for the hybrid versioning approach. The additional storage space increased by 8.6% for the integrated versioning, 73.25% for the separated versioning and 4.15% for the hybrid versioning.

The results of this evaluation indicate that hybrid versioning is not fit for real-world scenarios, as the re-execution of queries takes significantly longer than for the integrated and separated approach. Both other versioning approaches can be used in real-world settings, as the evaluation indicates. The implementation of the integrated approach is simpler and allows to use a wider variety of methods to avoid race conditions than the separated approach. The separated approach on the other hand has lower querying times and allows to keep the original tables unchanged.

Based on our analysis of the different versioning approaches and an analysis of their needs, we recommend using the separated approach to the ISMN team. This is because the

International Soil Moisture Network is continuously growing with over 100 new stations worldwide between November 2022 and April 2023 and a separation between the original and the history table is favorable to reduce the impact on query runtimes. Furthermore, the intrusiveness of the integrated versioning approach would require many changes to the backend, which are not connected to the download process. We further provided a list of necessary adaptions, to embed our framework into their platform, to provide dynamic data citation.

7.1 Future Work

As our implementation has many more facets, which can be explored, we would love to see progress in different areas.

For the implementation for the ISMN portal, there are several improvements, which would be great to evaluate and adapt to decrease the impact of implementing the WGDC recommendations. First of all, would it be great to evaluate the implementation on a fully functional test instance, which mirrors all operations occurring in the main system. This would help to gain a clearer picture of the limitations of our implementation and necessary adaptions to the system. It is further necessary to adapt the data integration workflow to not update the data at insertion, to enrich it with the correct quality flags, but do this in a combined step to decrease the impact of data versioning, by just storing two versions of the data instead of three versions. Further, it would be interesting to explore the runtime impact of the advanced querystore with a separated parameters table, as discussed in Section 4.2.2. We currently only evaluated the storage consumption of both querystore approaches, and found that the advanced querystore only requires a sixth of the storage, which is necessary for the simple approach.

Further work in dynamic data citation can include the extension of DBRepo [21] PostgreSQL containers, to increase the offered databases by the database repository and proof that multiple different databases can be combined with different underlying versioning implementations. Furthermore, it would also be interesting to port dynamic data citation to noSQL databases, which have not yet used it, as for example graph databases and document stores.

List of Figures

List of Tables

List of Listings

Bibliography

[1] D. Albert, B.K. Antony, Y.A. Ba, Y.L. Babikov, P. Bollard, V. Boudon, F. Delahaye, G. Del Zanna, M.S. Dimitrijević, B.J. Drouin, M.-L. Dubernet, F. Duensing, M. Emoto, C.P. Endres, A.Z. Fazliev, J.-M. Glorian, I.E. Gordon, P. Gratier, C. Hill, D. Jevremović, C. Joblin, D.-H. Kwon, R.V. Kochanov, E. Krishnakumar, G. Leto, P.A. Loboda, A.A. Lukashevskaya, O.M. Lyulin, B.P. Marinković, A. Markwick, T. Marquart, N.J. Mason, C. Mendoza, T.J. Millar, N. Moreau, S.V. Morozov, T. Möller, H.S.P. Müller, G. Mulas, I. Murakami, Y. Pakhomov, P. Palmeri, J. Penguen, V.I. Perevalov, N. Piskunov, J. Postler, A.I. Privezentsev, P. Quinet, Y. Ralchenko, Y.-J. Rhee, C. Richard, G. Rixon, L.S. Rothman, E. Roueff, T. Ryabchikova, S. Sahal-Bréchot, P. Scheier, P. Schilke, S. Schlemmer, K.W. Smith, B. Schmitt, I.Yu. Skobelev, V.A. Srecković, E. Stempels, S.A. Tashkun, J. Tennyson, V.G. Tyuterev, C. Vastel, V. Vujčić, V. Wakelam, N.A. Walton, C. Zeippen, and C.M. Zwölf. A decade with VAMDC: Results and ambitions. *Atoms*, 8(4):1–45, 2020.

[2] Peter Buneman, Greig Christie, Jamie A Davies, Roza Dimitrellou, Simon D Harding, Adam J Pawson, Joanna L Sharman, and Yinjun Wu. Why data citation isn't working, and what to do about it. *Database*, 2020:baaa022, January 2020.

[3] Peter Buneman, Susan Davidson, and James Frew. Why data citation is a computational problem. *Communications of the ACM*, 59(9):50–57, August 2016.

[4] Mark J. Costello. Motivating Online Publication of Data. *BioScience*, 59(5):418–427, May 2009.

[5] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. *The VLDB Journal — The International Journal on Very Large Data Bases*, 22(1):73–98, February 2013.

[6] Carlo Curino, Hyun Jin Moon, and Carlo Zaniolo. Graceful database schema evolution: The PRISM workbench. *Proceedings of the VLDB Endowment*, 1(1):761–772, 2008.

[7] Wouter Dorigo, Irene Himmelbauer, Daniel Aberer, Lukas Schremmer, Ivana Petrakovic, Luca Zappa, Wolfgang Preimesberger, Angelika Xaver, Frank Annor, Jonas

Ardö, Dennis Baldocchi, Marco Bitelli, Günter Blöschl, Heye Bogena, Luca Brocca, Jean-Christophe Calvet, J. Julio Camarero, Giorgio Capello, Minha Choi, Michael C. Cosh, Nick van de Giesen, Istvan Hajdu, Jaakko Ikonen, Karsten H. Jensen, Kasturi Devi Kanniah, Ileen de Kat, Gottfried Kirchengast, Pankaj Kumar Rai, Jenni Kyrouac, Kristine Larson, Suxia Liu, Alexander Loew, Mahta Moghaddam, José Martínez Fernández, Cristian Mattar Bader, Renato Morbidelli, Jan P. Musial, Elise Osenga, Michael A. Palecki, Thierry Pellarin, George P. Petropoulos, Isabella Pfeil, Jarrett Powers, Alan Robock, Christoph Rüdiger, Udo Rummel, Michael Strobel, Zhongbo Su, Ryan Sullivan, Torbern Tagesson, Andrej Varlagin, Mariette Vreugdenhil, Jeffrey Walker, Jun Wen, Fred Wenger, Jean Pierre Wigneron, Mel Woods, Kun Yang, Yijian Zeng, Xiang Zhang, Marek Zreda, Stephan Dietrich, Alexander Gruber, Peter van Oevelen, Wolfgang Wagner, Klaus Scipal, Matthias Drusch, and Roberto Sabia. The International Soil Moisture Network: serving Earth system science for over a decade. *Hydrology and Earth System Sciences*, 25(11):5749–5804, November 2021. Publisher: Copernicus GmbH.

[8] M. L. Dubernet, V. Boudon, J. L. Culhane, M. S. Dimitrijevic, A. Z. Fazliev, C. Joblin, F. Kupka, G. Leto, P. Le Sidaner, P. A. Loboda, H. E. Mason, N. J. Mason, C. Mendoza, G. Mulas, T. J. Millar, L. A. Nuñez, V. I. Perevalov, N. Piskunov, Y. Ralchenko, G. Rixon, L. S. Rothman, E. Roueff, T. A. Ryabchikova, A. Ryabtsev, S. Sahal-Bréchot, B. Schmitt, S. Schlemmer, J. Tennyson, V. G. Tyuterev, N. A. Walton, V. Wakelam, and C. J. Zeippen. Virtual atomic and molecular data centre. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 111(15):2151–2159, October 2010.

[9] Snehil Gupta, Connie Zabarovskaya, Brian Romine, Daniel A. Vianello, Cynthia Hudson Vitale, and Leslie D. McIntosh. Incorporating Data Citation in a Biomedical Repository: An Implementation Use Case. *AMIA Summits on Translational Science Proceedings*, 2017:131, 2017. Publisher: American Medical Informatics Association.

[10] Patrick Heidorn. Shedding Light on the Dark Data in the Long Tail of Science. *Library Trends*, 57:280–299, September 2008.

[11] Barend Mons, Herman van Haagen, Christine Chichester, Peter-Bram 't Hoen, Johan T. den Dunnen, Gertjan van Ommen, Erik van Mulligen, Bharat Singh, Rob Hooft, Marco Roos, Joel Hammond, Bruce Kiesel, Belinda Giardine, Jan Velterop, Paul Groth, and Erik Schultes. The value of data. *Nature Genetics*, 43(4):281–283, April 2011. Number: 4 Publisher: Nature Publishing Group.

[12] Heather A. Piwowar and Todd J. Vision. Data reuse and the open data citation advantage. *PeerJ*, October 2013. Publisher: PeerJ Inc.

[13] Stefan Pröll, Kristof Meixner, and Andreas Rauber. Precise Data Identification Services for Long Tail Research Data. *iPRES*, 2016.

[14] Stefan Pröll and Andreas Rauber. Scalable data citation in dynamic, large databases: Model and reference implementation. *2013 IEEE International Conference on Big Data*, pages 307–312, October 2013.

[15] Stefan Pröll and Andreas Rauber. A Scalable Framework for Dynamic Data Citation of Arbitrary Structured Data. *DATA 2014 - Proceedings of 3rd International Conference on Data Management Technologies and Applications*, pages 223–230, January 2014.

[16] Andreas Rauber, Ari Asmi, Dieter van Uytvanck, and Stefan Pröll. Data Citation of Evolving Data. *Research Data Alliance*, 2015. http://dx.doi.org/10.15497/RDA00016.

[17] Andreas Rauber, Bernhard Gößwein, Carlo Maria Zwölf, Chris Schubert, Florian Wörister, James Duncan, Katharina Flicker, Koji Zettsu, Kristof Meixner, Leslie D. McIntosh, Reyna Jenkyns, Stefan Pröll, Tomasz Miksa, and Mark A. Parsons. Precisely and Persistently Identifying and Citing Arbitrary Subsets of Dynamic Data. *Harvard Data Science Review*, 3(4), October 2021.

[18] John F Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, January 1995.

[19] Robert Schuler and Carl Kesselman. CHiSEL: a user-oriented framework for simplifing database evolution. *Distributed and Parallel Databases*, 39(2):483–543, June 2021.

[20] Patrick Säuerl. *Datenzitierbarkeit bei Schemaevolution in relationalen Datenbanken.* Thesis, TU Wien, 2018.

[21] Martin Weise, Moritz Staudinger, Cornelia Michlits, Eva Gergely, Kirill Stytsenko, Raman Ganguly, and Andreas Rauber. DBRepo: a Semantic Digital Repository for Relational Databases. *International Journal of Digital Curation*, 17(1):11, 2022.

[22] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J. G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A. C. 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, 3(1):160018, March 2016.

[23] Xin Zhao, Yigge Wan, and Yingbo Liu. Tracking and Querying over Timeseries Data with Schema Evolution. *Jisuanji Yanjiu yu Fazhan/Computer Research and Development*, 59(9):1869–1886, 2022.

[24] Carlo Maria Zwölf, Nicolas Moreau, Yaye-Awa Ba, and Marie-Lise Dubernet. Implementing in the VAMDC the new paradigms for data citation from the research data alliance. *Data Science Journal*, 18(1), 2019.

[25] Carlo Maria Zwölf, Nicolas Moreau, and Marie-Lise Dubernet. New model for datasets citation and extraction reproducibility in VAMDC. *Journal of Molecular Spectroscopy*, 327:122–137, September 2016.