FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Citation of dynamic data

William Norio Fukunaga

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Miguel Rocha da Silva, PhD

Second Supervisor: Gabriel Torcato David, PhD

July 25, 2019

Citation of dynamic data

William Norio Fukunaga

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Maria Cristina de Carvalho Alves Ribeiro
External Examiner: Prof. Pedro Manuel Rangel Santos Henriques
Supervisor: Prof. João Miguel Rocha da Silva

July 25, 2019

Abstract

Citation is a vital part of scientific research. However, citing datasets is not so valued as citing an article, impeding the lifecycle of research data. We will narrow that gap with this study.

In repositories without data versioning capabilities, any updates to a dataset overwrite the existing record without any trace of the changes made. This creates a problem when citing those records since there is no guarantee that, in the future, the citation will refer to the same data. Data versioning allows datasets to remain dynamic after publication while ensuring the integrity of any citations, by maintaining the entire history of changes.

Adding data versioning improves the research data lifecycle, since it enables adequate data citation of those versioned datasets. Data citation, in turn, enables the validation of research findings and the reuse of the datasets that support research articles.

This work will provide an easy way for developers to implement data versioning in their applications.

This work has the goal of designing and implementing an efficient database model capable of supporting subsets of versionable data. This database will then be packaged into a NPM (Node Package Manager) module.

This group has developed a NPM package that is easily available for users to integrate it with their own projects. It can also be used as a server although with reduced features. This is composed by a database that is capable of storing different versions of a file and has the ability of citing subsets from those file versions.

A comparison of this project against a production-ready solution has put this project a little slower than another. However, there are no production-ready solutions that support data subset citation. **Keywords**: Databases, MongoDb, NodeJS, citation, datasets.

Resumo

A citação é uma parte vital da pesquisa científica. Contudo, a citação de datasets não é tão valorizada como a citação de um artigo, impedindo o ciclo de vida de dados de pesquisa. Este estudo irá melhorar a valorização da citação de dados.

Em repositórios sem capacidades de versionamento de dados, qualquer atualização de um dataset atualiza os dados existente sem qualquer registo das modificações feitas. Isto cria um problema quando se cita aqueles registos visto que, não há garantia que, no futuro, a citação irá se referir aos mesmos dados. Dados versionados permitem aos datasets permanecer dinâmicos depois da publicação, assegurando a integridade de qualquer citação mantendo um histórico inteiro de modificações.

Adicionar versionamento de dados melhora o ciclo de vida dos dados de pesquisa, porque permite a citação de dados adequada de datasets versionados. A citação de dados, por sua vez, permite a validação de resultados de pesquisa e a reutilização dos datasets que suportam os artigos.

Este trabalho fornecerá um modo fácil de os programadores implementarem dados versionados nas suas aplicações.

Este trabalho tem o objetivo de criar e implementar um modelo de dados eficiente capaz de suportar subsets de dados versionáveis. Esta base de dados será depois compilada num módulo NPM (Node Package Manager).

Este grupo desenvolveu um módulo NPM facilmente disponível para os utilizadores integrarem com os seus próprios projetos. Também pode ser usado como um servidor isolado, apesar de não possuir tantas capacidades como quando utilizado como um módulo. Este projeto é composto por uma base de dados capaz de armazenar diferentes versões de um ficheiro e tem a capacidade de citar subconjuntos de dados.

This is composed by a database that is capable of storing different versions of a file and has the ability of citing subsets from those file versions.

Uma comparação feita deste projeto contra uma solução pronta para produção conclui que este projeto é um pouco mais lento. No entanto, uma solução que suporte citação de subconjuntos de dados não está facilmente disponível.

Acknowledgements

I want to thank my supervisor João Rocha for his dedication and help. He has been a great mentor throughout this entire process and also during my second half of my time in this course.

To my family and friends, for supporting me during the entire time and giving me the moral support to withstand the hard times, I have no words to express my gratitude towards you.

To all the people I worked with in these years, thank you for helping me develop my skills and showing me how to appreciate my work. I hope we can work together once more.

To everyone in *random.org #pt Long Live Tugalândia E Os Preços Baixos, thank you for never leaving me alone and procrastinating with me right until the last second of the deadline, sometimes even after that.

A special thanks to Daniela Sá and Vasco Pereira for accompanying me during those train rides that were delayed or cancelled every single time. Really, CP sucks. But I enjoyed those talks we had and hope we can continue having these talks.

Funding acknowledgements

The author thanks the support from FEUP in the form of a bachelor's scholarship lasting 3 months financed by the project Laboratório Sapo - Máquina do Tempo.

William Norio Fukunaga

"Just as the Hare was overconfident in its speed,
so the developers are overconfident in their ability to remain productive."

Robert C. Martin

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abreviations and Symbols

ACID	Atomicity, Consistency, Isolation, and Durability
API	Application Programming Interface
ARK	Archival Resource Keys
CODATA	Committee on Data of the International Council for Science
DC	Data Citation
DOI	Digital Object Identifier
NBM	National Bibliographic Numbers
NPM	Node Package Manager
PID	Persistent Identifier
PURL	Persistent URL
RDA	Research Data Alliance
URL	Uniform Resource Locator
URN	Uniform Resource Names
WG	Working Group
WWW	*World Wide Web*

Chapter 1

Introduction

Data derived from research results should be managed correctly, specially when handling dynamic data, since it can be updated continuously. Research data management ranges from the point in time when it enters the research lifecycle until it is analyzed and results are extracted. Researchers can then share their results to other researchers, who can later reuse them in their own research [WT11].

This project benefits the domain of research data management, by improving the data lifecycle. More specifically, it will promote data sharing and re-use.

1.1 Problem Statement and Motivation

Nowadays, there is a data life cycle that guides researchers on managing data. A successful management of the data permits data to be preserved for later reuse and creates a synergistic effect, as preserved data benefits new research. That research will, in turn, provide data that enables further research. And it is through the use of citations, that it is possible to credit researchers for their data and maintain reproducibility and provenance of said data. Citations can promote the data in the long-term using persistent identifiers (PIDs), which have a long lifetime.

Versioning is a data management process that allows a group to keep track of changes made over time and brings benefits to several areas of work. For example, a software development team uses a version control tool, such as git[1], to prevent developers from altering source code that overwrites changes made by other developers. If a mistake is made and previous changes were accidentally deleted, version management makes it possible to return to an earlier version and fix this mistake.

Data citation allows users to replicate and verify the data that studies are based on. However, some researchers still do not share their data, denying them the possibility of getting credit for the data obtained. This lack of data citation by the researchers is mainly due to priority concerns and a lack of rewards [MN12]. In some cases, data repositories don't possess any data versioning capabilities, i.e. updates to a dataset overwrite the existing record without any trace of the changes

[1]https://git-scm.com/

made. This creates a problem when citing those records since there is no guarantee that, in the future, the citation will refer to the same data.

While data citation is important, in some cases, scholars and/or researchers do not use the entire dataset, selecting only specific subsets of data that suit their needs. And, to cite the data that was used, researchers need to identify the exact version of the subset that was used for their research, in an continuously changing data environment. [RAVP16]

This project will offer a solution that easily and correctly cites subsets of data from researchers' data, improving the lifecycle of research data.

1.2 Goals

In order to improve the research data lifecycle, this project will take the following steps:

1. **Study the evolution of dynamic datasets** - Gather research information related to dynamic datasets
   - (a) **Compare existing approaches** - Analyze existing implementations through a systematic literature review
   - (b) **Contact the RDA-WG-DC group** - Ask for information on this area and research conducted by the group
2. **Allow citation of dynamic data in a repository** - Create a data system capable of citing subsets of dynamic data
   - (a) **Identify subset** - Define the granularity of the data model
     - i. **Select an identification logic** - Choose a possible representation of a unique identifier
     - ii. **Integrate a persistent identification provider with the versioning system** - Use the PIDs from the provider to enable identification of subsets
   - (b) **Capture different versions of datasets** - Capacity to retrieve previous states of the dataset before they were altered
     - i. **Design data model** - Create a data model capable of supporting a history of modifications to the dataset
     - ii. **Implementation** - Implement the data model capable of supporting versioning
   - (c) **Associate metadata to each subset** - Add unique information to the subset to identify it uniquely
     - i. **Enable versioning and provenance** - Define necessary metadata to provide versioning and provenance
     - ii. **Implementation** - Provide the system with the ability to assign metadata to a subset and to be able to identify the subset based on that same metadata
3. **Publish versioning module** - Compile the project into a NPM package

1.3 Document Structure

This report has 6 chapters in total. Aside from chapter 1, chapter 2 describes research about open science concepts, citation of dynamic data and also some approaches to obtain versioning in different databases. Chapter 3 outlines the approach taken to this problem of dynamic data citation. Chapter 4 describes how this project was made. Chapter 5 describes how the final project will be tested. Finally, this report concludes with some notes in chapter 6.

Chapter 2

Literature Review

2.1 Open Data for Open Science

Often, researchers do not publish their research data when publishing a research paper. This raises some issues since replication and verification of the research claims will be very difficult without it [MN12]. One reason why researchers do not generally publish their research data is due to the additional work necessary to prepare it for publishing. Sharing information means that it is necessary to associate detailed metadata regarding it, since when compared to publications, data depends even more on metadata for their correct interpretation, as it is often purely numeric.

There is a program pushing towards public access for scientific publications or research data named the *Horizon 2020*. Every project funded or co-funded under this program should provide its datasets under *Open Data* terms [Com17].

2.1.1 Open Science, Open Data and Open Access

Presently, due to the progress of technology, it has become easier to share information with one another. People can share their data to others for re-utilization without any charges, provided they attribute its origin. This is defined as *Open Data* [ope]. A different, related term, *Open Access*, refers to "online, free of cost access to peer reviewed scientific content with limited copyright and licensing restrictions" [Fos].

These two concepts, *Open Data* and *Open Access*, are connected to *Open Science*. The latter states that there should be free access to publicly funded research results with minimal to no cost. Complying with the Open Science principles brings benefits with it [Ope14], such as:

- Increased efficiency by re-utilizing materials and processes to update processes;
- Better quality search results through faster data validation, which in turn improves repeatability of results;
- Improved scientific literacy since the public can more easily obtain access to scientific results and methods

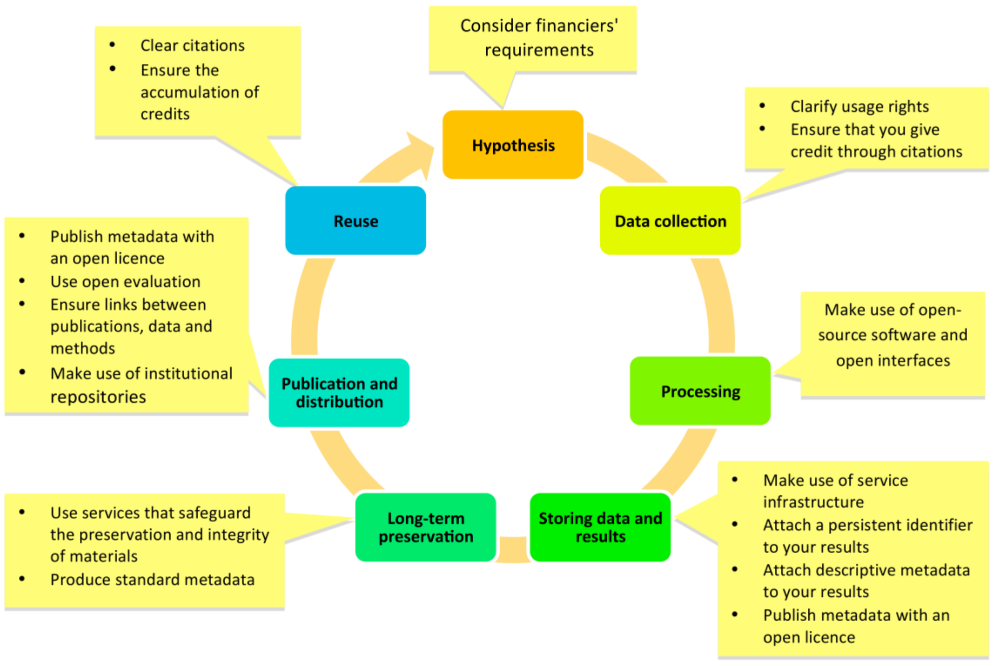
Consider financiers' requirements
Hypothesis
Clear citations
Ensure the accumulation of credits
Clarify usage rights
Ensure that you give credit through citations
Reuse
Data collection
Publish metadata with an open licence
Use open evaluation
Ensure links between publications, data and methods
Make use of institutional repositories
Make use of open-source software and open interfaces
Publication and distribution
Processing
Long-term preservation
Storing data and results
Use services that safeguard the preservation and integrity of materials
Produce standard metadata
Make use of service infrastructure
Attach a persistent identifier to your results
Attach descriptive metadata to your results
Publish metadata with an open licence

Figure 2.1: Stages of the research process[Ope14]

To truly obtain openness, it is necessary to adopt an open approach in all stages of the research process, as shown in Figure 2.1.

2.1.2 Research data lifecycle

The research data lifecycle guides researchers on managing data, as it enables researchers to dispose of the data or preserve it in the long-term and allowing to reuse it in the future [Hig08].

This creates a synergistic effect, as preserved data benefits new research. That research will, in turn, provide data enabling further research. And it is through the use of citations, that it is possible to utilize others' data to improve the research, as shown in Figure 2.2. Data citation is an important part of data search/reuse, which, in turn, is vital to this lifecycle [PV13].

The European Commission outlined a road map [CSH$^+$18] for compliance with the FAIR data principles, which state that data should be Findable, Accessible, Interoperable and Reusable. By applying these principles, researchers grant their data transparency, reproducibility, and reusability [Wea16].

2.1.3 Persistent Identifiers

Persistent identifiers (PIDs) are used to reference objects of interest for a long time. They were created to solve the issue of untrustworthy internet links, which could suddenly disappear, i.e. their persistence was not guaranteed. And along the years, they have been used in several situations, not only literature and data [KMWP17].

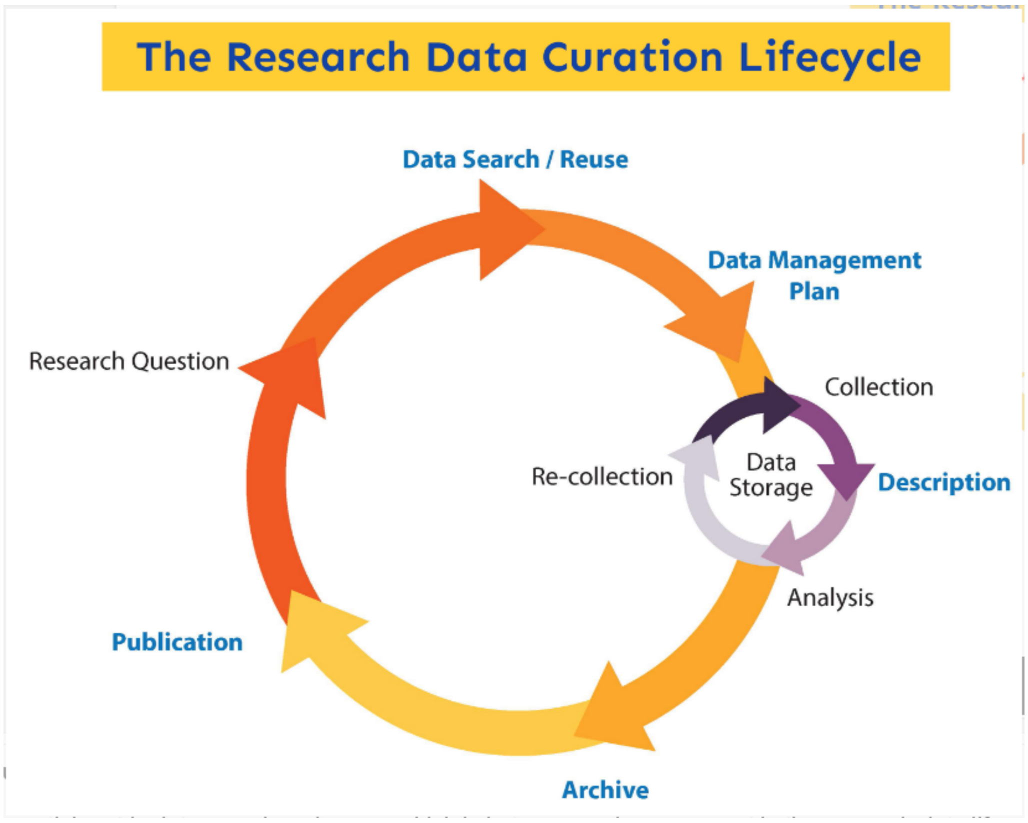

Figure 2.2: Research Data Curation Lifecycle[Hub]

Using PIDs to aid in the development of dynamic data citation has consequences, such as solving the issue of persistence. However, in dynamic data citation, subsets need to be identified and have their own metadata. This is not compatible with PIDs since it could require large amounts of subsets to have each their own PID. Instead, they are better suited for static data and serve only as reference point such as search queries[PR13a].

Some examples of persistent identifiers are: [HK06]

- **Uniform Resource Names(URN)** - concept with a common namespace for several different types of identifiers. Persistence and their resolution are key principles in their design
- **National Bibliographic Numbers (NBN)** - URN namespace used in national libraries. Focused in resource naming, both in print or electronic format
- **Handle System** - "...concept for a DNS-independent naming and resolving mechanism". Can resolve names to URLs and also convert them to different identifier formats and arbitrary data
- **Digital Object Identifier (DOI)** - framework to ensure common standards and practices. Uses the Handle System to name and resolve components
- **Archival Resource Keys (ARK)** - Combines necessary PID features with a framework. Focuses on resolving and delivering metadata

- **Persistent URL (PURL)** - focuses on the location of a resource in a persistent manner. Capable of relocation services and access to the resource's history of locations

- **OpenURL** - makes use of persistent identifiers but is not a scheme of persistent identification itself. It is a metadata transport protocol.

It should be pointed out that every case is unique and therefore, when implementing PIDs, the best solution for one situation may not be suitable for another.

2.1.4 Metadata for dynamic datasets

To cite an object, some information is necessary to identify the origin of the object. To this end, a working group on metadata, Metadata Working Group, determines and maintains the most appropriate standards for data citation in Datacite [Dat11]. They state that for data citation the following metadata are mandatory:

- Identifier (with mandatory type sub-property)

- Creator (with optional given name, family name, name identifier and affiliation sub-properties)

- Title (with optional type sub-properties)

- Publisher

- Publication year

- Resource type (with mandatory general type description sub-property)

According to Pröll and Rauber [PR13a], versioning support is needed to support dynamic citation but identifiers are inefficient when assigned to subsets due to the finer granularity of the system. Should the granularity be very fine, then a large amount of PIDs will be necessary, resulting in difficult and expensive (in terms of storage and computation) citations.

With these elements, a system can guarantee the following properties: **verifiability**, **provenance** and **persistence**. These properties and others will be defined in the next chapter.

2.2 Existing solutions

To handle dynamic data, the database needs to be capable of supporting versioning. To this end, there are several possible data models. This section presents various data models and implementations made using them.

2.2.1 Basic concepts

To design a system capable of supporting dynamic data citation, there are some properties of the data model that need to be defined beforehand.

A **temporal database** contains data that can suffer modifications over time [Sno86]. In it, reverting or applying changes are easier to express, compared to a conventional database. Conversely, in a conventional database, changes are seen as modifications to the state, having the previous state deleted from the database.

The **granularity** defines the smallest possible unit of data that is represented in a data model. In terms of datasets, this could be an entire directory structure, folder, file, table, or line. With each case, the complexity of the problem changes. On one hand, at a dataset level, not every changes made to its files may be recorded. On the other, keeping the granularity to a line-level implies keeping track of every line, with each line needing an identifier to track it. In a research paper published by Pröll and Rauber [PR13b], they kept the granularity at the level of a single record, keeping track of every minor detail.

A **dataset** is a collection of data. It can have different connotations regarding the data it represents. It can represent a group of folders containing resources with relations between them or it can represent a set of tables.

2.2.2 Relational approaches

Usually, a relational model only holds a state pertaining to a single point in time. Whenever an update to the database occurs, the existing data is overwritten by the new data. To use it with a system involving time, these attributes related to time are handled solely by the application program. In a temporal database however, querying through previous states is simpler and it is easier to represent changes to previous states [Sno86, JS99].

The temporal data model should be a consistent extension of an existing relational model. The relational model itself is a representation of the database as a collection of relations. A relation can be compared to a *table*, where each row represents a collection of related data values, also known as a *tuple*. A column header is called an *attribute* and every value in the same column possesses the same data type [EN16].

2.2.3 Document-based, noSQL approaches (time-varying in MongoDB)

Besides relational models for time-dependent data, one can also use NoSQL databases. NoSQL normally has six key features: [CR11]

1. capacity to horizontally scale simple operation throughput

2. replicate and distribute data

3. simple interface or protocol

4. a weaker concurrency model than the ACID (Atomicity, Consistency, Isolation, and Durability) transactions

5. efficient use of distributed indexes for data storage

6. capacity to dynamically record new attributes to data records

NoSQL can provide better versioning support capabilities than relational databases since it is more adaptable in its data model and provides better scalability, but only when there are gains to be obtained from de-normalization and this process is easy.

Relational models on the other hand, can allow for more sophisticated querying, but are more complex when managing temporal data, since the time-varying nature of the data must be respected and cannot scale as well [JS99, Mon12].

For this problem, scalability and query performance can, arguably, be more important than ACID features due to the simplicity of the existing relations.

2.2.4 Graph versioning

Graph-oriented technologies have been getting more popular recently, with the support of large companies, such as Facebook[1] with the GraphQL tool.

This popularity is due to their ability to efficiently handle data, by focusing on the relationships between the objects instead of the actual objects.

In 2013, Castelltort and Laurent [CL13] proposed a novel representation of historical graph data and tools to implement it as a plug-in for existing NoSQL graph systems, defining some mandatory requisites in their solution. They stated that, to support versioning, a graph database needed to be capable of providing methods to:

- Access the history of a node or a relationship,
- Obtain the differences between two different versions of a graph in time
- Track metadata about every version

2.2.5 Summary

In this section we have studied the basic properties of a data model essential for supporting dynamic data citation and different approaches to modelling the database so as to have version control capabilities. The relational database approach provides more advantages when the data model is normalized and de-normalizing it would not be beneficial. The graph versioning method is more adaptable than the relational model, but managing versions of a dataset can be complicated if not using supporting software. Taking into account these characteristics, we have decided to implement our data model in a noSQL approach due to its adaptability and versioning capabilities and also its good query performance.

[1]https://www.facebook.com/

2.3 Version control systems

Version control systems are tools that help manage information over time. They keep track of every change made to the information, giving the user the possibility to revert these changes. Version-control systems can be divided in three different types [git]:

- **Local**—users stores all the changes to the covered files in a simple local database. This allows them to backtrack in case of errors or accidents.
- **Centralized**—all file revisions are stored in a remote server, from which several users can fetch its contents, simplifying the organization and workflow of team projects. This allows users to be aware of what everyone related to the project is working on. However, it has a drawback: the single point of failure. All the information is stored on the server so, if the server is unresponsive for a certain amount of time, then collaboration between the users is impossible and it is not possible to save changes to the server. In case the server loses its contents and there are no backups, then all the information will be lost.
- **Decentralized**—There is no main server. All the users act as as servers and clients, storing all file revisions therefore, in case of a server failure, the others can restore the full data. One issue with this method is synchronization, since it is difficult to be certain of which server is the most up-to-date.

Based on these techniques, an implementation was made by Pröll and Rauber, where they designed a file-system based versioning with scriptable queries based on Git, a centralized version-control system. This enabled users to work with the versioned datasets simultaneously [PMR16].

2.3.1 UNIX version control systems

There have been several source control systems in the UNIX operating systems. However, only a few of them have brought significant changes [lin]:

- Source Code Control System (SCCS) - was the first UNIX version management system developed between 1970-1980. However, currently the system is obsolete and only old software uses it [Roc75, scc].
- Revision Control Systems (RCS)—was published shortly after its predecessor (SCCS), in the early 1980s. It improves the SCCS by providing an easier user interface, and improved storage of versions for faster retrieval [Tic85, rcs].
- Concurrent Version Systems (CVS)—released firstly in 1990 and then updated until 2008 as a front end for the RCS system. Although the individual file history is similar to RCS, it provides the ability to run scripts to log CVS operations and it enables developers located in different places or hindered by slow modems to function as a single team. [cvsa, cvsb]

	Intrusiveness	Storage demand	Query complexity
Integrated	high	low	low
Hybrid	low	medium	medium
Separated	low	high	low

Table 2.1: Comparison of table designs implementations [PR13b]

2.3.2 Implementations

Pröll and Rauber published a research paper in 2013 where they implement three different approaches for designing a database capable of retrieving versioned subsets in an efficient manner [PR13b].

The first approach, **integrated**, modifies all existing tables to include temporal metadata and a column to identify the version. This method is very intrusive, but it does not increase the storage space significantly, nor the queries' complexity.

The second approach, **hybrid**, added a table to the database that recorded every insertion, modification or deletion to the database. This approach is not very intrusive since it does not alter the existing tables, and it also does not demand large storage space. However, it possesses the disadvantage that the queries become very complex.

The third approach, **separated**, creates a copy of each record into a separate table, in addition to modification events. This method is also not very intrusive and the queries are simple to make. The drawback is the storage space required, that increases considerably. A summary of each method and its advantages/disadvantages are shown in Table 2.1.

The RDA Data Citation WG published a research paper [PR13a] proposing a framework model for dynamic data citation, capable of citing subsets. The proposed model can be applied to relational database management systems, but the authors also outline how to improve it to generic data sources. Figure 2.3 provides the interaction of the components of the network.

In another case, the same group used these recommendations 3.1 to develop a data citation system that handles large datasets while maintaining data privacy throughout the entire data life cycle. The solution proposed is not bound to any database vendor, instead focusing on being independent from specific software systems [PMR15].

The authors also published a research paper where they implemented three different approaches to a system capable of supporting versioning of subsets of data.

The first and most simple of the three was derived from a file system-based versioning with scriptable queries. The second was similar to the previous implementation, however it was inspired in Git branching, enabling different researchers to work on the same datasets without risk of conflicts. The last implementation made use of a relational database system, allowing efficient versioning and more flexible subset generation.

The first two solutions were deemed to be more efficient for small scale files instead of large scale. The third implementation covered the limitations from the other two by providing additional flexibility at the cost of complexity [PMR16].

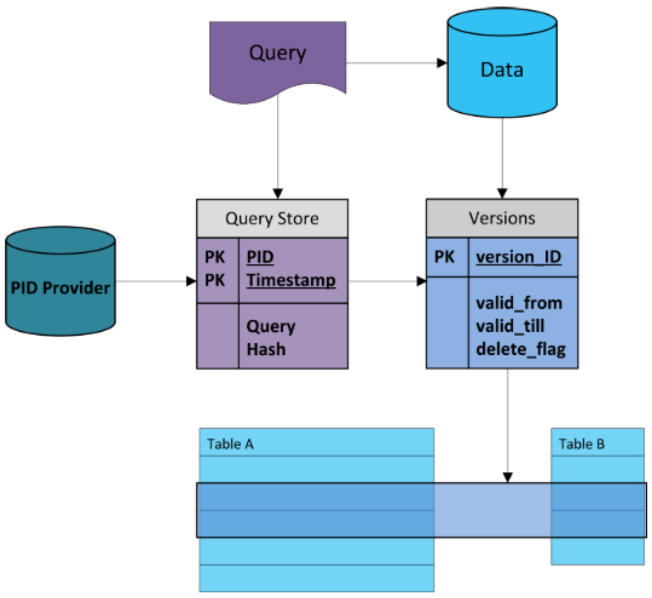

Figure 2.3: Comparison of table designs [PR13a]

Another group of researchers from the Washington University used the recommendations proposed by the RDA DC WG 3.1 to provide data versioning capabilities. More specifically, from those recommendations, they enabled data versioning and issued timestamps to the operations executed. They also implemented the query store, assigning timestamp to the queries and persistent identifiers to the queries. Finally they store the queries and provide a citation texts for the user. By adapting their database based on these recommendations, they also reduced the bottleneck of reproducibility by eliminating the time constraint of communication [GZR+17].

Not directly related to the work done by the RDA DC WG, Jonas Tappolet and Abraham Bernstein from the University of Zurich proposed a syntax and storage format based on graphs [TB09].

his approach, depending on the nature of the data, could significantly reduce the number of triples by removing redundancies which could lead to an increase in performance. A comparison between a normal query, a query using their proposed format and a query using their proposed format on top of a key-tree index structure is shown in Figure 2.4. These results show that in some cases their proposed query format may not provide better results than normal queries—however, the use of the key-tree index structure can compensate for that.

Another graph-based implementation was done by Arnaud Castelltort and Anne Laurent from the University of Montpellier [CL13]. They propose an innovative representation of historical graph data by creating an additional graph for tracking the version history. This could be done either by deploying the graph in a separate graph database or as a subgraph of the original graph database.

A drawback from the first representation is the need for consistency, managed possibly by synchronous methods, while in the second representation, queries related to the version have a consequence on the load of the overall system.

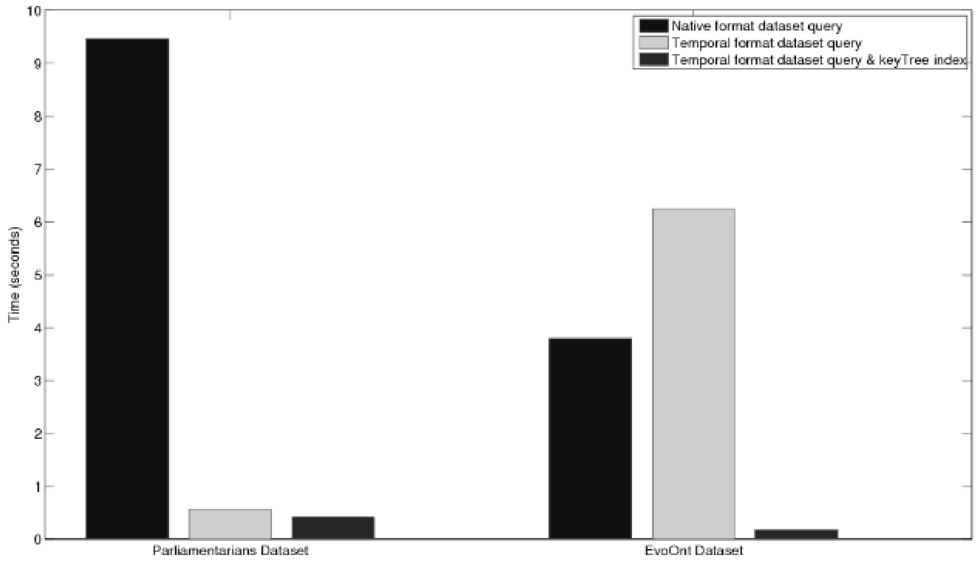

Figure 2.4: Execution times of time point queries using different datasets and query strategies [TB09]

2.3.3 Summary

In this section we analyzed different version control systems. Given the possibilities, we have opted for a separate database. This option, allows developers to use an existing database and complexity of the queries is not so high when compared to other alternatives. Aside from this approach, we will take into account the works made in this area, more specifically, the implementations from the RDA-WG-DC and their recommendations to building a system capable of providing data versioning and citation, and use that to develop our project.

Chapter 3

Approach

This chapter focuses on describing the problem and presenting the solution obtained to this problem. First, there are some basic concepts which need to be stated before the problem and solution are explained. This is described in 3.1. Afterwards, the problem is defined in 3.2. Then, the requirements implied by the problem are stated in 3.3. The chapter ends with the explanation of the proposed solution.

3.1 Basic Concepts

Before a solution can be presented, there are some concepts that need to be stated for clarity. A task group from the Committee on Data of the International Council for Science (CODATA) have carried out research on processes to enable data citation. A task group released a research paper on existing practices on data citation, suggesting also a set of guiding principles for implementing data citation [Tas13]:

1. **Status of Data**: Data citations should be given the same importance as the citation of other objects.
2. **Attribution**: Citations should facilitate giving scholarly credit and legal attribution to all parties responsible for those data.
3. **Persistence**: Citations should be as durable as the cited objects.
4. **Access**: Citations should provide the data and the associated metadata and documentation as much as necessary for both humans and machines to make informed use of the referenced data.
5. **Discovery**: Citations should support the finding of data and their documentation.
6. **Provenance**: Citations should facilitate the discovery of the data origin.
7. **Granularity**: Citations should support the finest-grained description necessary to identify the data.

8. **Verifiability**: Citations should contain information sufficient to identify the data unambiguously.

9. **Metadata Standards**: Citations should employ widely accepted metadata standards.

10. **Flexibility**: Citation methods should be sufficiently flexible to accommodate the varying practices among communities but should not differ so much as to compromise interoperability of data across communities.

Regarding the benefits of using data citations, it is the **verifiability** of the dataset, the most important benefit. With this property, the integrity of the dataset is maintained and therefore, researchers may compare and verify the data.

Another research group, the Research Data Alliance Working Group on Data Citation (RDA DC WG) has investigated how to efficiently cite subsets of data, concluding their research with 14 recommendations on how to build a system that supports dynamic citation of data subsets [RAVP16]:

1. **Data Versioning** - implementing versioning to ensure previous states are retrievable

2. **Timestamping** - apply timestamp to every operation (insertion, modification, deletion)

3. **Query Store Facilities** - store queries and necessary metadata to re-run them in the future

4. **Query Uniqueness** - normalize the query so identical queries can be detected.

5. **Stable Sorting** - record sorting should be unambiguous and reproducible

6. **Result Set Verification** - compute the checksum of the query result set to verify the correctness of a result

7. **Query Timestamping** - assigning a timestamp to the query

8. **Query PID** - assign a new persistent identifier (PID) to a query if it is new or the result set is different from previous results

9. **Store the Query** - store the query and necessary metadata in the query store

10. **Automated Citation Texts** - automatically generate citation texts, PID included, to simplify data sharing and citation

11. **Landing Page** - provide a human-readable page, given the PID, including the data, metadata, a link to the superset and the citation snippet

12. **Machine Actionability** - implement an API/ machine actionable landing page that allows users to access the data and metadata, given the query

13. **Technology Migration** - migrate the queries and fixity information when the data is migrated to a new representation

14. **Migration Verification** - verify a successful migration, confirming the queries can be re-executed correctly.

These recommendations can be divided in categories. The steps 1 to 3 have the goal of preparing the system for the posterior steps and to adapt existing systems which do not have them in a non-intrusive manner. After preparing the system, it is ready to identify specific datasets in the long-term with steps 4 to 10. For the user to retrieve the data and analyze it, it is necessary to have steps 11 and 12. Steps 14 and 15 provide recommendations on accommodating the data infrastructure to ensure sustainable preservation of the dynamically generated subsets [RAVP16].

From the mentioned recommendations, preparing the database for versioning and retrieving and showing the data are the first, more essential tasks. On the other hand, identifying the datasets in the long-term is a more complex and time-consuming task. Migration tasks can be very simple or very complex, depending on the organization of the solution.

3.2 Problem statement

Extra care is essential when citing data when compared to citing a research paper. In some cases, data repositories don't possess any data versioning capabilities, i.e. updates to a dataset overwrite the existing record without any trace of the changes made. This creates a problem when citing those records since there is no guarantee that, in the future, the citation will refer to the same data.

3.3 System requirements

Below is a list of requirements necessary for this project to be capable of citing subsets of dynamic data:

- Retrieve the current version of a file - To retrieve the current file version, the system needs to keep track of the most recent version document. This is done by assigning a date of the modification of the document
- Retrieve past versions of a file - To see past versions, it is necessary to store the different versions of the same file and keeping a timestamp of when it was saved in the database to be able to distinguish and order them accordingly. This is according to the recommendations from the RDA-WG-DC in 2.2.1
- Query one or several versions of a file - Allowing users to query the database for versions of a file is not complicated, but the main issue here is speed. The query should be deliver the expected results as quickly as possible. More details are explained in 3.5
- Store the query and its metadata - Storing the query allows the user to reuse it afterwards for validation of the results. Regarding the RDA-WG-DC recommendation number seven (assigning a timestamp to the query), there is more than one possible timestamp to assign to the query:

1. The timestamp is the actual execution time of the original query
2. Assigning the time of the last update to the database prior to the query
3. Assigning the time of the last update to the subset of data affected by the query

The first option is the most obvious and simple to realize. However, some issues arose after some discussions. Assigning it in this manner would reveal the moment the query was made by a user, possibly divulging private information. The second option is recommended by the RDA-WG-DC, as it eliminates the privacy issue. The third alternative also covers the problem from the first option however, it is more complex comparing it to the second option [RAVP16].

Assigning a PID and hashes to a query document conforms to the recommendation numbers four, six and eight. Computing a hash of the normalized query allows for efficient detection of identical queries while the hash of the result set allows for verification of the correctness of the result of a posterior similar search. Appointing a persistent identifier on the query should the query be new or the result set from a similar query is different allows the retrieval of the query and also the result set enabling provenance.

Lastly, in order to cite a subset, it is necessary to choose a citation format. This is explained in the next section.

- Assigning a PID to a query and hashes to it and its respective result set - appointing a PID preserves the data in the long term. On the other hand, a hash of the query can later be used to identify similar queries while a hash of the resulting set confirms the correctness of the information of the result
- Retrieve a subset data based on a query - The information stored in the database related to a query should also allow the user to retrieve the resulting information from that query
- Cite a subset of data - The system should be able to generate a citation pointing to the query and its respective result

3.3.1 Citation alternatives

When citing versioned data, there is no general consensus on the citation format. Researchers can use a citation method which suits them best, but is not acceptable to others. This section presents four approaches that can cite versioned data [Dat11].

Rauber et al. recommend citing a timestamped query containing the previous search so it is possible to re-run it on a versioned database. Figures 3.1 and 3.2 show examples of possible automatically generated citation texts.

In Figure 3.1, the PID refers to the dataset, while in Figure 3.2, it refers to the query that was executed, which in turn references the entire dataset. One important aspect to take into account is the date designation of the query. A possible solution would be to assign the actual date of the

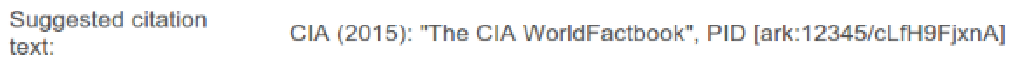
Suggested citation text: CIA (2015): "The CIA WorldFactbook", PID [ark:12345/cLfH9FjxnA]

Figure 3.1: Citing an entire dataset [RAVP16]

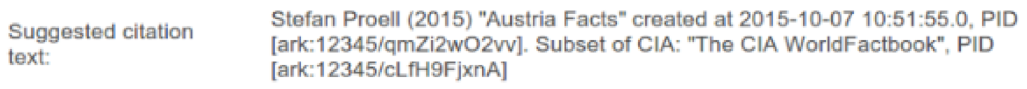
Suggested citation text: Stefan Proell (2015) "Austria Facts" created at 2015-10-07 10:51:55.0, PID [ark:12345/qmZi2wO2vv]. Subset of CIA: "The CIA WorldFactbook", PID [ark:12345/cLfH9FjxnA]

Figure 3.2: Citing a subset of the Data Set [RAVP16]

execution of the query to the metadata, but this raises privacy concerns, since it reveals the instant when the query was made. Instead, the authors recommend using the last update of the data store as the timestamp. Another possibility would be to assign the timestamp of the last update on the data to the affected data subset. This solution however, is more complex than the second since it requires the retrieval of the subset and then calculating the most recent updates to that same subset [RAVP16].

Aside from citing the timestamped query, the other alternatives are *time slice*, *full snapshot* and *continuously updated* approaches. *Time slice* cites a set of updates that were made during a time interval, as shown in Figure 3.3. The *full snapshot* involves making a copy of the entire dataset, as seen in Figure 3.4. The last option, *continuously updated*, cites the dataset as normal but inserts additional metadata, access date and time, to the citation; this can be seen in Figure 3.5 [BD15].

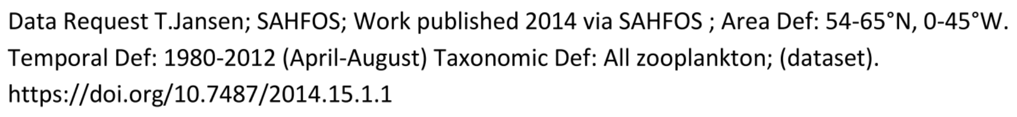
Data Request T.Jansen; SAHFOS; Work published 2014 via SAHFOS ; Area Def: 54-65°N, 0-45°W. Temporal Def: 1980-2012 (April-August) Taxonomic Def: All zooplankton; (dataset). https://doi.org/10.7487/2014.15.1.1

Figure 3.3: Citation example based on time slice [Dat11]

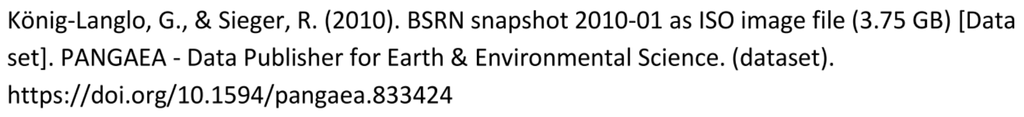
König-Langlo, G., & Sieger, R. (2010). BSRN snapshot 2010-01 as ISO image file (3.75 GB) [Data set]. PANGAEA - Data Publisher for Earth & Environmental Science. (dataset). https://doi.org/10.1594/pangaea.833424

Figure 3.4: Citation example based on full snapshot [Dat11]

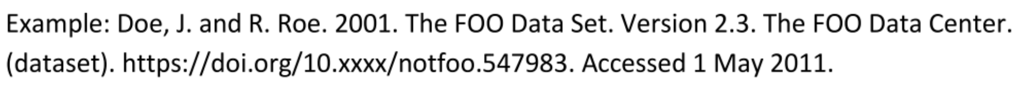
Example: Doe, J. and R. Roe. 2001. The FOO Data Set. Version 2.3. The FOO Data Center. (dataset). https://doi.org/10.xxxx/notfoo.547983. Accessed 1 May 2011.

Figure 3.5: Citation example based on continuously updated dataset [Dat11]

3.4 Maturity of the versioning process

Given the recommendations analyzed relatively to data citation in 3.1, it is possible to establish levels of maturity regarding data versioning support.

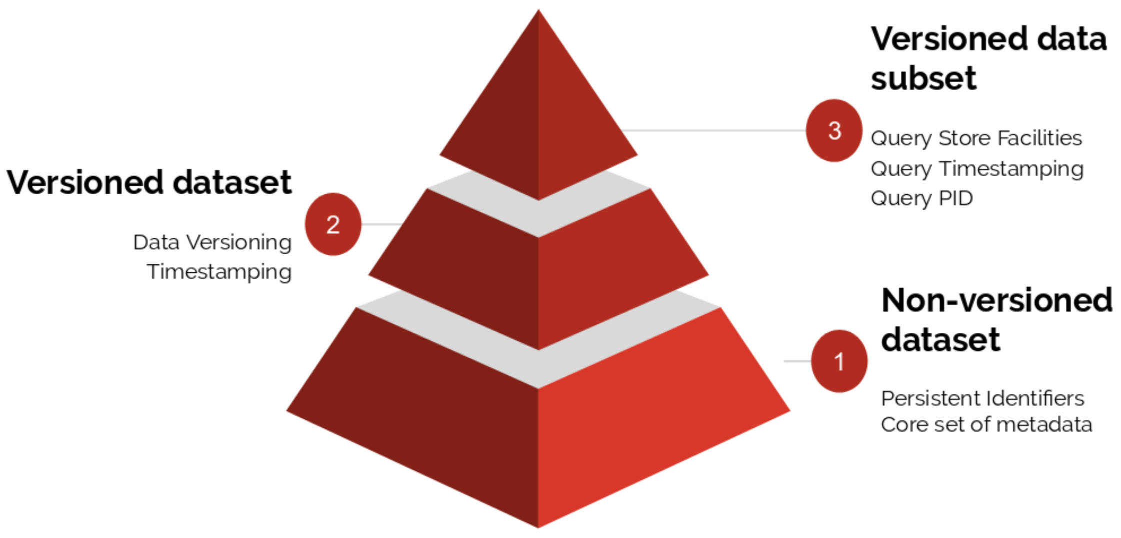

Figure 3.6: Levels of maturity for data citation

Figure 3.6 shows a high-level overview of the different levels of maturity for data citation support.

Starting from the bottom, each level adds capabilities that support the level above. On the first level, **non-versioned dataset**, the user can cite only datasets, as in most cases in current data management solutions. This is made possible by having persistent identifiers pointing to the dataset and core metadata to provide information related to the dataset.

Some existing platforms in this stage of maturity are DSpace[1] and CKAN[2]. CKAN itself does not have versioning capabilities built-in, so extensions that provide that functionality must be installed a posteriori. There is one plugin for the CKAN software which meets several of the recommendations provided by the RDA-WG-DC[3].

The next step is to enable a dataset to present information from a version that was modified afterwards, i.e. versioned data. This can be achieved by making use of timestamps; every change to the dataset is recorded without overwriting the original data, every delete is also marked with a timestamp but not erased. This allows the user to backtrack to a previous version.

[1]https://duraspace.org/dspace/
[2]https://ckan.org/
[3]https://github.com/fwoerister/ckanext-mongodatastore

Most platforms are in this stage, in which they fully support dataset versioning. Some platforms are EPrints[4], B2Share[5] and Zenodo[6].

And the last step in dynamic data citation is presenting subsets of versioned data. Following the recommendations from a previous study [RAVP16], this could be implemented with a query store that keeps track of every search made to the database and storing the corresponding core metadata.

This stage is very recent, with only some systems in this stage such as the Distributed Infrastructure of the Virtual and Atomic Molecular Data Center(VAMDC) and also the Washington University[PR13c, GZR+17]. These systems were mostly all adapted by the RDA-DC-WG. However, open access repositories with these capabilities were not found.

Considering each of these steps, the solution we propose would be at this last stage of this maturity phase.

3.5 Proposed solution

Taking into account the existing solutions provided by the RDA-WG-DC and the issues with versioned data citation, the proposal is a flexible database capable of supporting versionable data citation and that can be easily integrated in different projects. Several existing solutions are capable of supporting citation of versioned data. However, these are specific cases only, so they are optimized to handle the specific needs of each case. This benefits the product in itself, however, due to its nature, each case requires a unique approach and therefore has a lengthier development phase. Conversely, our solution may not be as optimized for each case, but allows for an easier and faster development by providing a database capable of supporting versionable data citation that is useful across diverse domains.

3.5.1 Data model design considerations

After gathering requirements, the next step in developing such a solution is to create a data model, which conforms to the recommendations stated in 3.1 and be as efficient as possible.

The data model can be optimized towards one of two aspects: memory usage or speed. For this solution, we chose speed. This meant that the database schema would not be as normalized as it could be, since it implied separating the data into different components. Separation of components benefits memory usage and reduces redundancy but has a negative impact on speed when retrieving information on more than one component at the same time. This is described later in 4.1.1.

The next step is to allow the user to upload files in a simple manner. For this, the CSV (comma separated values) file format was selected. Its simplicity and existence of multiple tools that export and/or import the data into different file formats constitutes a good choice. Although uploading a CSV file should be the easiest method to add its contents to the database, there should also

[4] https://www.eprints.org
[5] https://b2share.eudat.eu/
[6] https://zenodo.org/

be a method which allows users to manually upload content as they so choose. This alternative however, is costly in terms of manual labour when dealing with large amounts of content.

After uploading the content into the database, the user should be able to retrieve its contents from the database as needed.

To achieve this, there was a possibility of creating middleware acting between the user calls and the database interaction. Through a simple client API, the user would choose the operators and this middle man would convert it to a readable query by the database. This option increases simplicity at the cost of a necessity of awareness, from the side of the user, regarding the internal database structure.

This architecture was not chosen due to its difficulty. There would need to be an extensive operator conversion to deal with most if not all the operators specific to the database. Adding to that, the user would need to learn how this custom made solution works, in order to correctly use them. Given these challenges, we have decided to not create any conversion middleware, giving the user the single task of knowing the underlying database language.

After acquiring the necessary knowledge, the user is capable of performing queries unto the database. When a query is made, the system provides the opportunity of storing it and the respective result. This allows the user to later reuse it, by citing it or for comparison purposes. Regarding the citation, by reflecting on the different citation methods described in 3.3.1, we have chosen the representation of the subset of the data set in Figure 3.2. This method does not provide any computational advantages compared to the other methods and so, it is not better or worse than the others. As for timestamps, the system assigns by default the time of the last update to the database as the time of the query. If users do not find this action appealing, they may opt to insert their own date.

One issue with this project is assigning a PID to the query for citation purposes. This system is designed with integration with other projects in mind. Thus, this solution does not generate the persistent identifiers itself. Instead, we believe it is best for this system to entrust the task of PID generation to the projects that implement this system, as well as maintaining the links between PIDs and versioned resources.

Another possible cause for concern is related to the security of the database. As stated earlier, the main goal is to have this system integrated in other projects, instead of functioning as a standalone solution. Therefore, we believe that every access should be allowed and permissions management be managed by the external system that integrates this implementation. The developers using this system are responsible for handling the security side. The user should decide how to best proceed according to their own project.

Chapter 4

Implementation

This section details the implementation of the features that make up the project. It describes how the system works to support its use cases. It also states the technologies used and the reasoning for using them.

4.1 System implementation

In order to get a better understanding of how the system works, a diagram was made. It can be seen in Figure 4.1. An outline is explained below.

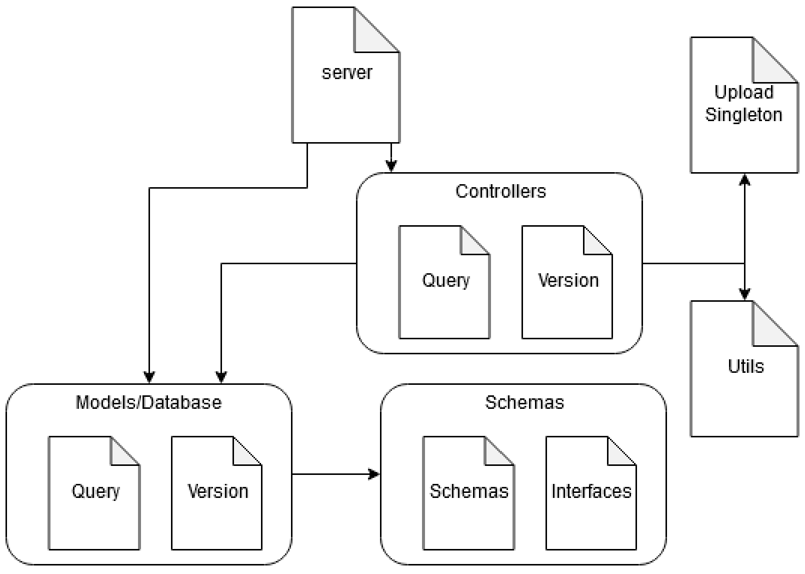

Figure 4.1: Components Diagram

The implementation is composed of separate files, each with their own purpose and groups of services, with a common purpose although with different intended targets. The project was developed using NodeJS[1] as a base and using the different packages it contains.

[1]https://nodejs.org

The **Server** file has the purpose of serving as an endpoint for user calls. This endpoint has several features which will be described later in 4.2. This was developed using *Express*[2].

The **Controllers** group of files handles the requests that need preprocessing before actually interacting with the database. This includes tasks such as uploading CSV files into the database or altering a query, maintaining the end result but changing its structure, to compare it later for similar queries. It it composed of the Query file which handles the retrieval and storage of queries in the database, and the Version file that manages the data that users insert into the database This group uses additional files: uploadSingleton and utils. While the uploadSingleton file is used for assisting the CSV file upload, the utils file is a set of methods that do not belong in the other groups.

The **Models** group corresponds to the database handlers. The database itself is run in MongoDB[3], a document based distributed database, which stores the data in a format similar to that of the JSON file format(Javascript object notation). The database is manipulated using the mongoose module[4], simplifying the process of interacting with the database. This group deals with executing searches in the database or adding data to it.

The **Schema** group corresponds to the database schema and is necessary for the mongoose software to parameterize the models interaction with the different collections in the database. It represents the manner in which the database is modelled after. This is presented in the subsection 4.1.1.

As for the technologies used, NodeJS is widely used (at the time this was written, there were 66533 projects in GitHub with the tag NodeJS[5]) and its export and importing methods are very simple and easy to use. Therefore, integration with other projects should not cause any major issues. As for the Server, using *Express* was the best choice since we already had experience with this technology and there are a large amount of tutorials to help guide through this technology. Since this project dealt with CSV files of variable properties, a No-SQL would be best suited to dealing with this type of data. Therefore, MongoDB was chosen. Lastly, Mongoose was chosen to simplify the usage of the MongoDB database and hasten the setup process.

[2]https://expressjs.com/
[3]https://www.mongodb.com/
[4]https://mongoosejs.com/
[5]https://github.com/topics/nodejs

4.1.1 Data model

Keeping in mind the recommendations mentioned previously in 3.1, the data model should be created in a manner that supports, among several properties, versioning and provenance especially. Therefore, the following excerpt represents the proposed data model.

```
Row : {
    creator:        String,
    date:           Date,
    resourceId:     String,
    tag:            String,
    hidden:         Boolean,
    lineIndex:       Number,
    lineContent:     [Line]
}

Line : {
  index:    Number,
  content:  String,
  header:   String
}

Query: {
  hashQuery:      String,
  hashResult:     String,
  pidData:        String,
  pidQuery:       String,
  queryText:      String,
  date:           Date,
  creator:        String,
  resourceId:     String,
  tag:            String,
}
```

Listing 4.1: Database schema

The data model is composed of the Row, Line and Query collections. Regarding the *Row* collection, the *resourceId*, *tag* and *lineIndex* properties serve as indexes (resourceId points to the file this document belongs to, the tag serves as a method of identifying the version of the file and the lineIndex indicates the line of the original content it holds). The hidden property is included in the data model for versioning purposes. Should a user attempt to delete a *Row* document from the database, the system should not actually delete that content but instead mark it as deleted.

The purpose of the *Line* collection is to represent the details of a single column of a *Row* document. It holds little value as a single document and is used only as a nested document. One important aspect regarding this collection when querying is that it is a nested document. Therefore, the MongoDB engine needs the order of the Line content to be ordered the same way as defined or the query will not function correctly. This is a particularity of the MongoDB technology itself. The implementation is not responsible for this characteristic.[Mon]

The *Query* collection holds properties used to verify the correctness of its contents. These are the `hash` and `pid` properties. The date property, as stated in 3.5, is given the value of the last time the database had a modification.

The PID is not generated by this project. Therefore, there is a route which allows users to alter the database entry, inserting the PID. This corresponds to the */updateQuery* route. More details are explained in 4.2.1.

Aside from these notes, there were some aspects that were fundamental in creating and are necessary to understand this data model:

- Each line of a CSV file corresponds to a single document, since there is a restriction on the document size in MongoDB - 16 megabytes [Mon19]. This limit would be reached for larger files, which would prohibit them to be uploaded to the database. While this limit can theoretically also be reached with just a single line, it is an unlikely situation and requires an extremely large amount of data in that line of the CSV file.

- Each row is organized in an array instead of an object (where the key would be the column header), due to possible repetitions in the column name, which would overwrite the values.There is no issue regarding storing the contents of a line as an array since, in JavaScript, the order of an array is preserved.

- As stated earlier, the focus was on query speed rather than memory storage optimization. By increasing redundancy with the metadata, repeating a dataset details, *join* operations are limited to complex queries. For simpler searches through the database, the query should theoretically be much faster.

4.2 Usage

It is possible to use this project in two different ways. One is to run it as a server, setting up an endpoint, in which it is possible to make use of the defined routes in a simple way. This was developed for running in *localhost* or as a container using *docker*[6]. The other option, is to integrate as a NPM package into another project, enabling the user to take full advantage of the features of this project.

In each case, the system alters its behaviour slightly. The following subsections describe each method.

[6]https://www.docker.com/

4.2.1 Server environment

One of the ways to make use of this project, is by setting it up as a server. This is made possible by using the Docker set of tools to create an isolated container or by using the node command and creating a local HTTP server. As stated earlier, it can be initialized by running the terminal command *npm start* for the localhost alternative while the command *docker-compose up* runs the program in a docker container.

A diagram presenting its behaviour is shown in 4.2

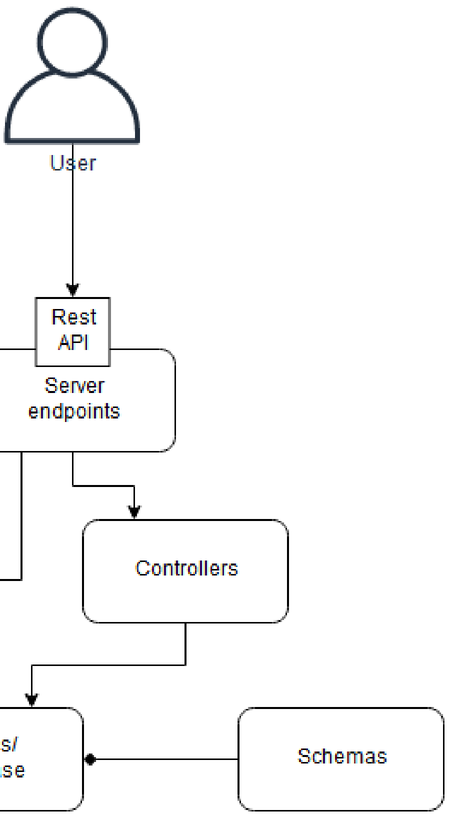

Figure 4.2: Server Behaviour

In this case the client transmits data to the system through a REST API. As stated before, this method is simple and easy to setup. It cannot, however, make full use of the database capabilities and instead uses the defined routes. The defined routes are stated in A.

In each route, the server returns an error should the request be missing a parameter.

4.2.2 NPM package integration

The other method of taking advantage of this project is to import it as a NPM package. Figure 4.3 shows a simple diagram representing its behaviour.

Through this, the user has the possibility of using the features implemented and in addition, the user can also make use of the MongoDB capabilities by simply calling the corresponding methods.

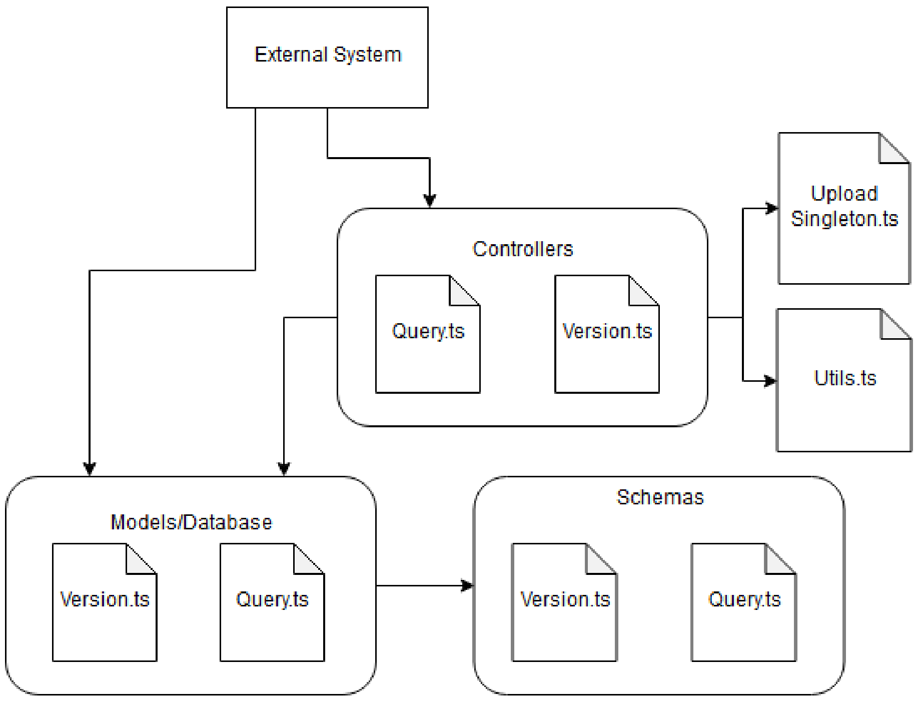

Figure 4.3: NPM Package Behaviour

4.3 Implementation of Use Cases

This section presents some examples of using this project. In these situations we will be showing the system in a server environment, presenting the requests and respective responses. The client was developed using the Python language due to its simple and easy methods of emitting requests.

4.3.1 Uploading data through a CSV file

This case involves uploading a file into the database. The user sends a POST request to the server with a file in its body. The server calls the controller responsible for the *Row* functions, which in turn inserts it in the database through the *Row* model functions. Afterwards, the server returns a response to the client whether it was successful or not. Figure 4.4 shows a sequence diagram of the process of uploading a file into the database.

The user sends a POST request to the */addFile* route. The request contains the CSV file to upload to the database and some metadata necessary for identification and retrieval. The metadata required is the *resourceId* to identify the file, the tag to distinguish between versions of the same file and the user who sent the request. An example of the request body is shown in Listing 4.2. These properties stated are all *string* types. The file is sent using a stream multipart form-data object. This is necessary for files that exceed the limit on http length.

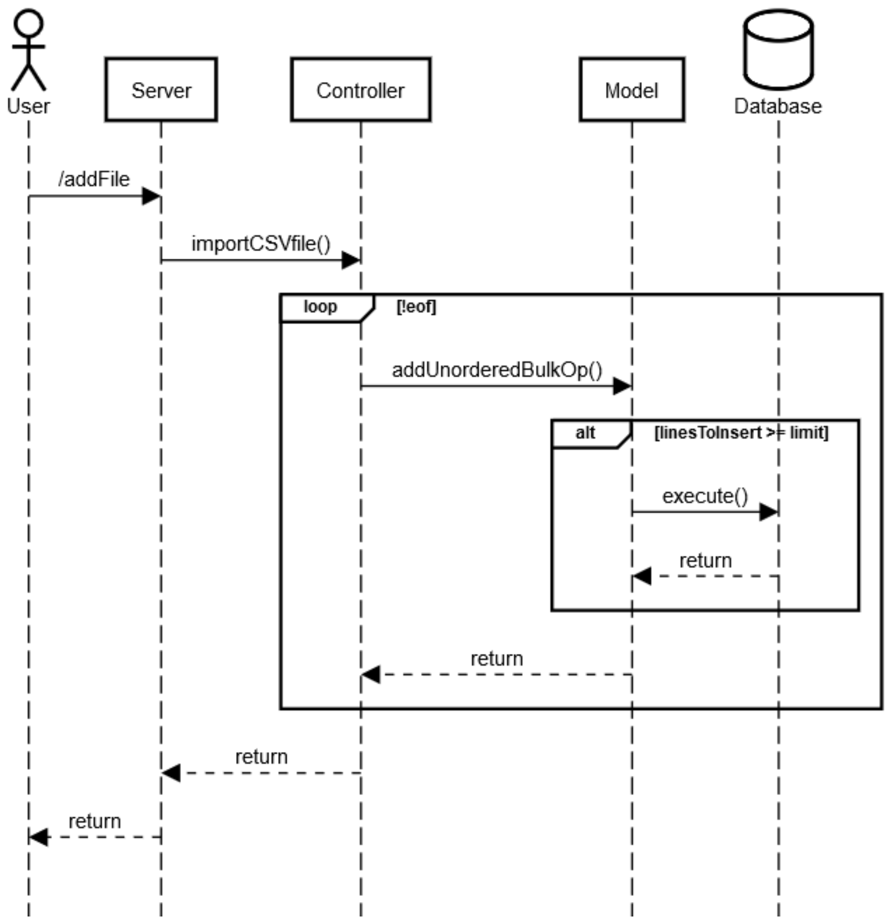

Figure 4.4: CSV file upload behavior

```
{
    resourceId: "test file",
    tag: "First version",
    user: "demouser"
}
```

Listing 4.2: Request body

After the server receives the request, it stores the content received in a temporary file before processing it. This is another measure for large sized files, which would otherwise occupy memory space in the RAM (random access memory) and could slow down processes in the server. The server then analyzes the file content by line by calling *importCSVfile()* from the controller domain, dividing each column through its separator. Then, each line is stored in a buffer which will afterwards be inserted in bulk, so as to optimize insertion time, if there is no document with the same index in the database. Should there be no problems the server will respond to the client with a success message.

4.3.2 Querying the database

This example demonstrates a user searching the database through content of a version of a file.

The user sends a GET request to the server. It, in turn, processes the data received and queries it through the database. Should the user want to add the query data into the database, then it is only necessary to add a parameter to the request stating that wish. If true, the controller responsible for the query methods processes the query and adds it to the database. The server then responds whether or not the operation was successful. Figure 4.5 shows a sequence diagram of this process. Another example described below provides a more detailed description regarding the creation of a query.

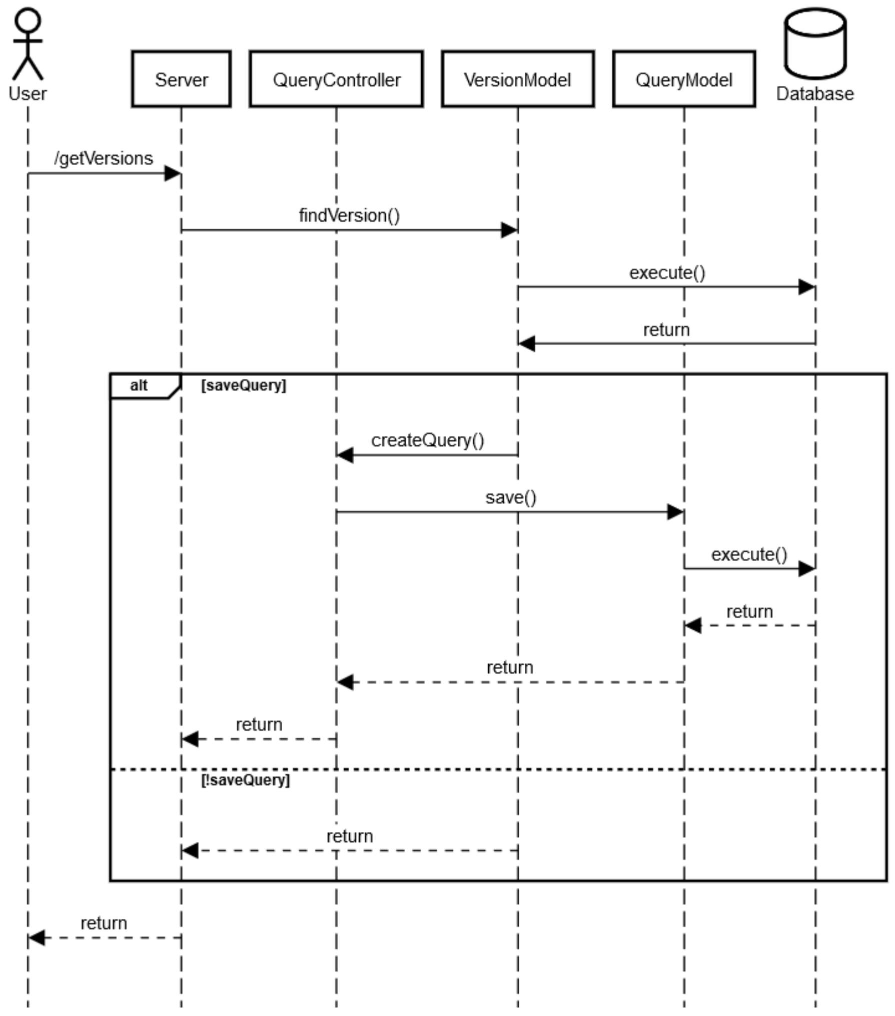

Figure 4.5: Query search behavior

First, the user sends a GET request to the */getVersions* route. This request's body contains only the JSON object required to perform the query and possibly other parameters to alter the return of the database content for example, to change the amount of entries the database should return or the order of the entries retrieved from the database. Listing 4.3 provides an example of a client request. The *query* parameter is a JSON type object that may contain special operators related to the MongoDB syntax and must be valid according to that same syntax. The other optional parameters are, in this example, a *string* and an *integer*. And the *create* is a boolean type that states whether or not the user wishes to store the query into the database.

```
{
    query: {
        lineContent: {
            $elemMatch: {
                header:'id',
                content:{
                    $gt: 4,
                    $lt:10
                }
            }
        }
    },
    sortBy: "-date",
    limit: 1,
    create:false
},
```

Listing 4.3: Query JSON 1

After the server receives the request from the client, it calls the *VersionModel* which, before executing it in the database, runs a quick validation test, to see if the received query object can be executed in the MongoDB database. Afterwards, if there are no problems detected, the database runs the query and returns its contents. As stated above in 4.1.1, the query of a document inside a document must be correctly ordered as defined or the database will not be able to retrieve the information requested. Then, if the user sent in the request whether the query should be stored the system will check for anomalies and store it if there are no problems. The process of inserting a query is explained in more detail in the following subsection.

4.3.3 Inserting a query

This last example explains the action of creating a new query into the database. The client sends a POST request asking to insert a query into the database. Then the server calls the Controller to process the data received and then store it into the database through the Model. The server then returns a message telling whether or not the operation was a success.

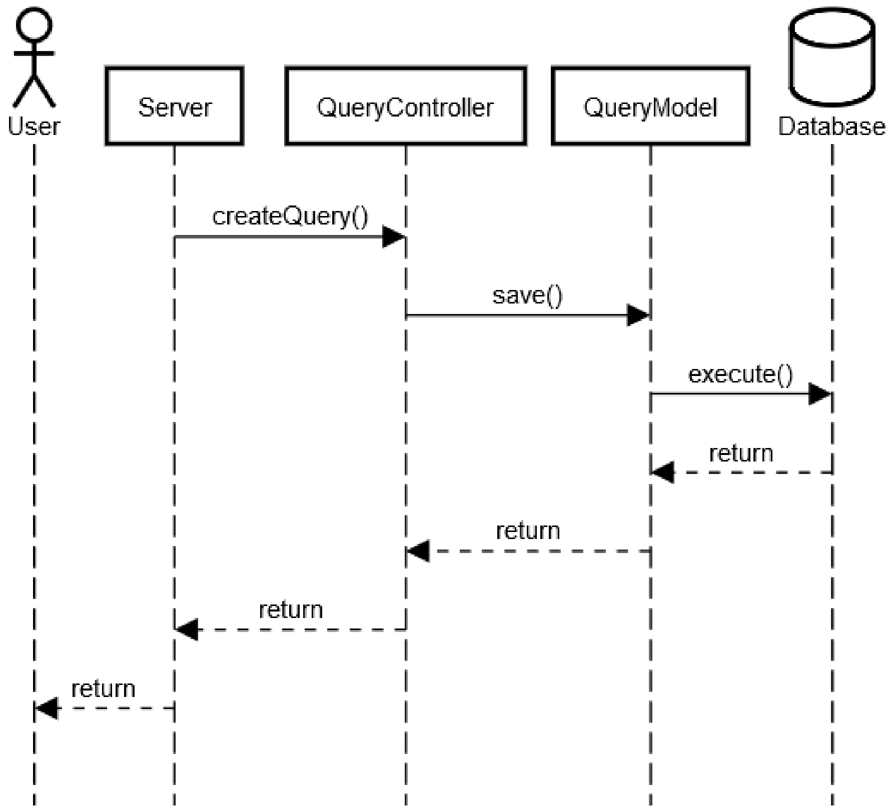

Figure 4.6: Query search behavior

First, the client sends a request to the */addQuery* route. This request contains the query the user wants to store, and a parameter stating whether or not the query should be rearranged. Listing 4.4 provides an example of the request body that is sent to the server. The *query* parameter is a JSON type object that may contain special operators related to the MongoDB syntax and should be valid according to that same syntax. The *order* indicates whether or not the query need to be rearranged. By default, the system assumes that it is necessary to rearrange unless the user also sends that parameter.

```
{
    query: {
        lineContent: {
            $elemMatch: {
                header:'id',
                content:{
                    $gt: 4,
                    $lt:10
                }
            }
        }
    },
    order: false
},
```

Listing 4.4: Query JSON 2

After the server receives the request, it calls the Controller to handle the processing through the *createQuery()* method. The processing is a rearrangement of the order of the query object. The rearrangement of the query starts by putting the properties included in the schema first and then the other properties are sorted alphabetically. This is needed to create a hash of the similar, rearranged query to search for similar queries. The query will only be inserted into the database if there are no entries in the database with the same rearranged query or, if there is, it must not have the same result set. As explained previously in 3.5, the `date` property (timestamp) will not be the time of the execution of the query, but instead the time the database was last updated will be used. This choice, as explained is due to privacy issues. The server, finally, responds to the user accordingly whether or not the query could be inserted into the database.

Chapter 5

Evaluation

To analyze this project, some metrics have been taken and compared to existing platforms and/or software. These experiments were run on a laptop with an Ubuntu operating system and an Intel i7-4800U CPU and 8GB of RAM.

It should be pointed out that this version of the Dataverse software, 4.9.4, does not posses querying capabilities, and cannot visualize the contents of the CSV file. However, more recent versions (tested with 4.12), can display the contents of the file but, it was not possible to recreate it in a Docker image. As such, it was not possible to ensure similar testing environments for a fair comparison with Dataverse.

5.1 CSV Upload

To test the performance when uploading a CSV file into the database, a comparison was made against Dataverse [1], an open source research data repository software. Another comparison was made using the `read.csv` package for the R language[2].

To ensure equal conditions in experimental scenarios, we used packaged docker images for our prototype and Dataverse (the Dataverse-docker project was used in this case[3]. R was run directly on the host machine, without any virtualization solution. Afterwards, through a simple client application, HTTP requests were sent and the results measured in response times, with CSV files of different sizes. The different times can be seen in 5.1.

As table 5.1 shows, our solution was slower than the Dataverse software as the file size got considerably larger. And the R language could not handle the large CSV files, due to not having enough memory available to store it all.

[1]https://dataverse.org/
[2]`https://stat.ethz.ch/R-manual/R-devel/library/utils/html/read.table.html`
[3]https://github.com/IQSS/dataverse-docker

col x lines	Our Prototype	Dataverse(4.9.4)	R read_csv	file size
10^2 x 13 (24KB)	0,01	0,91	0,016	24,58 KB
13 x 10^2 (24KB)	0,05	1,37	0,003	24,58 KB
10^2 x 10^2 (164KB)	0,01	0,92	0,019	167,94 KB
13 x 10^3 (216KB)	0,08	1,08	0,02	221,18 KB
10^3 x 13 (220KB)	0,02	0,93	0,095	225,28 KB
13 x 10^4 (2.1MB)	0,36	1,02	0,2	2,20 MB
10^4 x 13 (2.2MB)	0,1	0,92	4,05	2,31 MB
10^3 x 10^3 (16MB)	0,6	1,18	2,55	16,78 MB
10^4 x 10^4 (1.6GB)	52,52	41,91	1072,363	1,72 GB
10^3 x 10^6 (16GB)	552,49	449,45	-1	17,18 GB

Table 5.1: Caption

5.2 Query times

To analyze the performance of this system, we measured the time it took of some queries to retrieve information from the database. The program was put in a docker image as the previous analysis. In this situation however, there were no production-ready systems that had versioning and querying capabilities in a docker container that allowed us to perform a fair measurement method. The most recent versions of the Dataverse system had querying capabilities but there was no official container available with these new versions. In the end of this section, we present the query times for each objective and the difference in needing to arrange the query beforehand and sorting it previously.

5.2.1 Example 1

The purpose of this query was to ask the database for values between 4 and 10 belonging to the *id* header column. In this situation, due to the characteristics of the MongoDB database and syntax, it is not necessary to order the nested contents of the query since it is inside a MongoDB operator, *$elemMatch*. The query can be seen in Listing 5.1.

```
{
    query: {
        'resourceId': '/file/a234542',
        'tag': 'version-example!',
        lineContent: {
            $elemMatch: {
                header:'id',
                content:{
                    $gt: 4,
                    $lt:10
                }
            }
        },
    },
    order: false,
    projection:{lineContent:1, resourceId:1, tag:1, \_id:0}
},
```

Listing 5.1: Query request 1

5.2.2 Example 2

This query interrogates the database for documents that contain the author name 'Alex Smith' or 'Colas Camelli' and it also excludes documents that the system marked as hidden. As the previous case, by having the nested document as a property of a MongoDB operator, there is no need to sort the query and therefore, arranging the query provide no significant benefits. The query to send to the database is listed in Listing 5.3

```
{
    query: {
        '$elemMatch':{
            'lineContent.header':'author',
            'lineContent.content':{'$in':['Alex Smith', 'Colas Camelli']}
        },
        'hidden':False
    },
},
```

Listing 5.2: Query request 2

5.2.3 Example 3

This query tries to fetch documents that store the 'Tizanidine' value in the 'title' column, which is the second column of the imported CSV file(the indices start at 0). In this situation, there is a nested document. Therefore the order of the properties inside it must be the same as the schema. In this case there will be a sorted and unsorted query to send to the server. The server will rearrange the unsorted one to compare if there is significant difference in performance.

```
{
    query: {
        'lineContent': {
            'index': 1,
            'content': 'Tizanidine',
            'header':'title'
        }
    },
}
```

Listing 5.3: Query request 3 sorted

```
{
    query: {
        'lineContent': {
            'header':'title',
            'content': 'Tizanidine',
            'index': 1,
        }
    },
},
```

Listing 5.4: Query request 3 unsorted

5.2.4 Example 4

This use case stores the query into the database for later reuse. We used the query from 5.2.1 and another similar one but with the properties in a different order.

```
{
   'resourceId': '/file/a234542',
   'tag': 'version-example!',
   'lineContent': {
       '$elemMatch': {
           'header':'id',
           'content':{
               '$gt':4,
               '$lt':8
           }
       }
   }
}
```

Listing 5.5: Query request 4 sorted

```
{
    'lineContent': {
       '$elemMatch': {
           'content':{
               '$lt':8
               '$gt':4,
           },
           'header':'id',

       }
    },
    'resourceId': '/file/a234542',
    'tag': 'version-example!',
}
```

Listing 5.6: Query request 4 unsorted

5.2.5 Comparing query execution times

After describing the queries that were run in the program, we now present, in Table 5.2, the execution times of these queries, from the moment the client sent the request, to the moment the server responds the client.

Having measured these times, it is safe to assume that rearranging a query before sending it to the server has no significant impact, the reason most likely being the small number of lines to sort. The time difference should therefore be more noticeable with larger queries. However, given

Query	Performance(seconds)	
	Sorted	Unsorted
1	-	0.045
2	-	0.013
3	0.008	0.010
4	0.010	0.013

Table 5.2: Query performance times

the organization of the database schema (not normalizing as much as possible), the need for large queries should be limited to extreme situations.

Chapter 6

Conclusion

This chapter presents closing remarks about this entire project, presenting some notes about some issues encountered, the main contributions this project has done for the scientific community and areas in which this project could be further developed in the future.

6.1 Conclusion

The focus of this project is the development of a software solution capable of keeping a record of all versions of the data contained in a tabular file. It also allows the user to query the data and, if desired, to cite a subset.

Regarding the evaluation, when comparing to similar or related software we conclude that our solution is worse than the Dataverse software regarding file upload speed for large files, but better than the `read.csv2` function of the R programming language, commonly used for for reading CSV files.

With this project, we believe that we have improved the technology related to this area, if only by a little. The major contribution is a software that provides an easy integration with other projects (provided they use NodeJS) and has versioning and subset citing capabilities.

6.2 Further developments

This project has fulfilled the requirements presented in 3.3. As future work, we propose several aspects that could further improve this research:

- Developing the data model - the data model can be further developed to increase the usefulness of this software. For example, adding a *folder* collection to better organize the datasets.
- Adding more server routes - another possible improvement is increasing the functionality available when using this software as an HTTP server.
- Connecting to PID generation services - although this software requests that users add the PID by themselves, there is the possibility of generating the PIDs itself

- Supporting additional file types - this system has the capability of importing various file types, not just the CSV format
- Client interface - an easy to use user interface to use this project when setting it up as a server

6.3 Difficulties and issues encountered

One of the issues first met in this project was the inception of an efficient data model that both minimized memory and storage use and maximized execution speed of its services. It proved difficult to develop a model capable of satisfying both requirements simultaneously, so we designed a model that would prioritize speed over memory. The resulting model can be seen in 2.3.

Another challenge was evaluating our solution. Existing solutions that are capable of providing versioning and citing subsets of datasets are very few and we could not find one to compare it with.

There was a solution which we managed to compare against, *Dataverse*. We managed to compare data loading speeds but we could not compare query operation efficiency on all platforms. Although Dataverse does possess querying capabilities, this feature appeared in recent versions of the software and therefore, an updated Docker image of Dataverse with which a fair comparison could be made did not exist yet.

References

[BD15] Alex Ball and Monica Duke. How to Cite Datasets and Link to Publications, 2015.

[CL13] Arnaud Castelltort and Anne Laurent. Representing history in graph-oriented NoSQL databases: A versioning system. In *Eighth International Conference on Digital Information Management (ICDIM 2013)*, pages 228–234. IEEE, sep 2013.

[Com17] European Commission. H2020 Programme - Guidelines to the Rules on Open Access to Scientific Publications and Open Access to Research Data in Horizon 2020. (March), 2017.

[CR11] Rick Cattell and Rick. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12, may 2011.

[CSH$^+$18] Sandra Collins, Observatoire Astronomique De Strasbourg, Natalie Harrower, Simon Hodson, Sarah Jones, Digital Curation Centre, Leif Laaksonen, and Daniel Mietchen. FAIR Data Action Plan. *Interim recommendations and actions from the European Commission Expert Group on FAIR data*, (June):1–21, 2018.

[cvsa] Concurrent Versions System. https://en.wikipedia.org/wiki/Concurrent{_}Versions{_}System. Accessed on 2019-07-31.

[cvsb] CVS - Open Source Version Control. https://www.nongnu.org/cvs/. Accessed on 2019-07-31.

[Dat11] DataCite. DataCite Metadata Schema for the Publication and Citation of Research Data. page 29, 2011.

[EN16] Ramez Elmasri and Sham Navathe. *Fundamentals of database systems*. Pearson London, 2016.

[Fos] What is Open Science? Introduction | FOSTER. https://www.fosteropenscience.eu/content/what-open-science-introduction. Accessed on 2019-01-26.

[git] Git - About Version Control. https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control.

[GZR$^+$17] 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(Table 1):131–138, 2017.

[Hig08] Sarah Higgins. The International Journal of Digital Curation. 3(1):134–140, 2008.

[HK06] Hans-Werner Hilse and Jochen Kothe. *Implementing persistent identifiers*. 2006.

[Hub] Ann Hubble. Library Guides: Research Data Management: Home.

[JS99] C.S. Jensen and R.T. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.

[KMWP17] Jens Klump, Fiona Murphy, Tobias Weigel, and Mark Parsons. Editorial: 20 Years of Persistent Identifiers – Applications and Future Directions. *Data Science Journal*, 16(0):1–7, 2017.

[lin] Unix Programming - Version-Control Systems - Unix Tools for Version Control. https://www.linuxtopia.org/online_books/programming_books/art_of_unix_programming/ch15s05_3.html. Accessed on 2019-07-31.

[MN12] Hailey Mooney and Mark Newton. The Anatomy of a Data Citation: Discovery, Reuse, and Credit. *Journal of Librarianship and Scholarly Communication*, 1(1):eP1035, may 2012.

[Mon] Query an Array of Embedded Documents — MongoDB Manual. https://docs.mongodb.com/manual/tutorial/query-array-of-documents/. Accessed on 2019-06-03.

[Mon12] Morgan D Monger. Temporal data management in nosql databases. *Journal of Information Systems & Operations Management*, 6(2):237–243, 2012.

[Mon19] MongoDB. MongoDB Limits and Thresholds — MongoDB Manual. https://docs.mongodb.com/manual/reference/limits/{#}bson-documents, 2019. Accessed on 2019-04-16.

[ope] What is Open Data? http://opendatahandbook.org/guide/en/what-is-open-data/. Accessed on 2019-01-09.

[Ope14] Open Science and Research Initiative (ATT). Open Science and Research: The Open Science and Research Handbook. (December):16, 2014.

[PMR15] Stefan Pröll, Rudolf Mayer, and Andreas Rauber. Reproducible database queries in privacy sensitive applications. *IFAC-PapersOnLine*, 28(1):113–114, 2015.

[PMR16] Stefan Pröll, Kristof Meixner, and Andreas Rauber. Precise Data Identification Services for Long Tail Research Data. 2016.

[PR13a] Stefan Pröll and Andreas Rauber. Citable by Design - A Model for Making Data in Dynamic Environments Citable. *2nd International Conference on Data Management Technologies and Applications (DATA2013)*, pages 206–210, 2013.

[PR13b] Stefan Pröll and Andreas Rauber. Scalable data citation in dynamic, large databases: Model and reference implementation. *Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013*, pages 307–312, 2013.

[PR13c] Stefan Pröll and Andreas Rauber. Scalable data citation in dynamic, large databases: Model and reference implementation. *Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013*, pages 307–312, 2013.

[PV13] Heather A. Piwowar and Todd J. Vision. Data reuse and the open data citation advantage. *PeerJ*, 1:e175, oct 2013.

[RAVP16] Andreas Rauber, Ari Asmi, Dieter Van Uytvanck, and Stefan Pröll. Identification of Reproducible Subsets for Data Citation, Sharing and Re-Use. *Bulletin of IEEE Technical Committee on Digital Libraries, Special Issue on Data Citation*, pages 6–15, 2016.

[rcs] Gnu rcs. https://www.gnu.org/software/rcs/. Accessed on 2019-07-31.

[Roc75] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, Dec 1975.

[scc] Gnu sccs. https://www.gnu.org/software/cssc/. Accessed on 2019-07-31.

[Sno86] Richard Thomas Snodgrass. Temporal Databases. pages 1–8, 1986.

[Tas13] CODATA-ICSTI Task Group on Data Citation Standards and Practices. Out of Cite, Out of Mind: The Current State of Practice, Policy, and Technology for the Citation of Data. *Data Science Journal*, 12(September):1–75, 2013.

[TB09] Jonas Tappolet and Abraham Bernstein. Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, pages 308–322. Springer-Verlag, 2009.

[Tic85] Walter F. Tichy. Rcs — a system for version control. *Software: Practice and Experience*, 15(7):637–654, 1985.

[Wea16] Mark D. Wilkinson and Dumontier et al. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, 3:160018, mar 2016.

[WT11] Angus Whyte and Jonathan Tedds. Making the Case for Research Data Management Introduction – Doing More with Less Funders ' Data Policies. *Preservation*, 2011.

REFERENCES

Appendix A

Server environment API

The following segments describe the possible features that the user can utilize when setting up the project as a server.

Get information about the database

Get all details about the Row collection. Calls the MongoDB function *Collection.stats()*

- Route - http://$SERVER/
- Request type - GET
- Request body - None
- Returns a JSON object containing information on the database collections

Get Row content

Queries the database for data from the Row collection

- Route - http://$SERVER/getVersions
- Request type - GET
- Request body

```
    query :      {}
    amount?:     int
    sortBy?:     date
    order?:      boolean
    projection?:{}
    saveQuery?: boolean
    rearrange?: boolean
```

- Returns a JSON object of the call

Upload content through a CSV file

Receives a csv file and uploads its contents to the database

- Route - http://$SERVER/addFile
- Request type - POST
- Request body

```
    file
    creator:    string
    resourceId: string
    tag:        string
```

- Returns a csv file and the appropriate metadata and uploads it to the database

Get queries stored in the database

Retrieves information regarding the stored queries from the database

- Route - http://$SERVER/getQueries
- Request type - GET

- Request body

```
    query:      {}
```

- Returns documents that fit the given parameters

Add a query

Manually stores a query to the database

- Route - http://$SERVER/addQuery
- Request type - POST
- Request body

```
    query:      {}
    result:     {}
    rearrange:  boolean
    userId:     user
```

- Returns queries that fit the parameters

Update a query

Alters information from one or more query stored in the database

- Route - http://$SERVER/updateQuery
- Request type - POST
- Request body

```
    query:      {}
    result:     {}
    rearrange:  boolean
    userId:     user
```

- Returns a message if the update was successful

Delete a query

Deletes one or more queries that fit the user parameters from the database

- Route - http://$SERVER/deleteQuery
- Request type - POST
- Request body

```
query:      {}
result:     {}
rearrange:  boolean
userId:     user
```

- Returns a message if the delete was successful

Cite a query

Generates a citation and returns it to the user

- Route - http://$SERVER/getCitation
- Request type - POST
- Request body

```
query:      {}
result:     {}
rearrange:  boolean
userId:     user
```

- Returns the subset citation

Appendix B

NPM package environment API

The following segments describe the possible features that the user can utilize when setting up the project as a NPM package environment.

Query controller API

Describes the methods available to use from the query controller

B.0.1 Creating a Query document

Creates a query if it does not exist in the database

```
async function createQuery({queryObj, result, user="defaultUser", arrangeQuery=true
    })
```

- Parameters
  - queryObj - Query in JSON format to add to the database
  - result - JSON of the received result from the query
  - user User that created the query
  - arrangeQuery - if it is necessary to arrange it for later comparison
- Returns mongodb output if successful or null if there was an error

B.0.2 Sort the query properties

Sorts the query by placing the properties according to the schema first and additional parameters afterwards alphabetically

```
function orderQueryFull(queryText:{})
```

- Parameters
  - queryText - query to be made to the database
- Returns ordered query

B.0.3 Sorts the schemaless query properties

Sorts the schemaless query properties using the merge sort algorithm

```
function sortQuery(queryObj)
```

- Parameters
  - queryObj - JSON Object to sort
- Returns ordered query

B.0.4 Check if query exists

Checks the database whether the query exists or not

```
function sortQuery(queryObj)
```

- Parameters
  - queryObj - JSON object to check
  - arrangeQuery - if it is necessary to rearrange it in order to search for it
- Returns true if it exists, false if not

Row controller API

Describes the methods available to use from the query controller

B.0.5 Import a CSV file

Uploads the content from a csv file to the database

```
async function importCSVfile({filepath, printMetrics=false, data, deleteFile})
```

- Parameters

  - filepath - path of the temporary file holding the csv content
  - printMetrics - option to print upload time to the console
  - data - metadata to store alongside the content
  - deleteFile - option to delete the file located on the filepath in the end of this method

- Returns true if it exists, false if not

B.0.6 Add a Row document

Adds a row document manually into the database

```
async function addVersion(versionData)
```

- Parameters
  - versionData - data necessary to store into the database
- Returns stored object

Query model API

Describes the methods available to use from the query model

B.0.7 Verify result correctness

Verifies the correctness of the result query by comparing the hashes

```
async function isResultCorrect({result, arrangeObj = true} = {result:{}})
```

- Parameters
  - result - data received from a query
  - arrangeObj - whether the object needs to be rearranged
- Returns true if result is correct, false if not

B.0.8 Empty the database

Deletes all the data related to the Query collection

```
function dropData()
```

B.0.9 Get Queries related to a data version

Gets all queries related to a specific file version

```
async function getFromVersion({resourceId, tag}: {resourceId: String, tag: String})
```

- Parameters
  - resourceId - id to identify the file
  - tag - string to identify the version of a file
- Returns queries found

Row model API

Describes the methods available to use from the row model

B.0.10 Empty the database

Deletes all the data related to the Row collection

```
function dropVersion()
```

B.0.11 Get latest content version from the given id

Gets the latest row content from the given id

```
async function findLatestById({resourceId})
```

- Parameters
  - resourceId - id to identify the file
- Returns all the data found

B.0.12 Get Row data

Queries the database for row documents using the given parameters

```
async function findVersion({amount=20, query, sortBy="-date", order=true,
    projection = {}} = {query:""})
```

- Parameters
  - amount - maximum number of documents to return
  - query - query to interrogate the database with
  - sortBy - property to organize the result with
  - order - if necessary to arrange the query before interrogating the database
  - projection - parameters to receive from the database
- Returns row data found

B.0.13 Get last update time

Gets the time of the last update to the row collection

```
async function getLastModificationDate()
```

- Returns the date

B.0.14 Get schema

Gets schema from the row collection

```
function getSchema()
```

- Returns the schema

B.0.15 Get statistics

Get statistics related to the row collection

```
function getStats()
```

- Returns the statistics

B.0.16 Order schema properties

Orders the schema properties of the query

```
function orderQuery(queryText)
```

- Parameters
  - queryText - Query to order
- Returns the sorted query