TU WIEN Informatics

Entwicklung eines Data Citation Frameworks für RDF* Stores

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

066 926 Wirtschaftsinformatik

eingereicht von

Filip Kovacevic, BSc
Matrikelnummer 01227213

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 31. August 2021

Filip Kovacevic	Andreas Rauber

TU WIEN Informatics

Designing a Data Citation Framework for RDF* stores

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

066 926 Business Informatics

by

Filip Kovacevic, BSc
Registration Number 01227213

to the Faculty of Informatics

at the TU Wien

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

Vienna, 31st August, 2021

Filip Kovacevic	Andreas Rauber

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Filip Kovacevic, BSc

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

Wien, 31. August 2021

Filip Kovacevic

Acknowledgements

To my family: thank you for providing me with your love and support during my studies and thank your for believing in me.

To my friend Alek: Thank your for all your helpful tips and suggestions for my master thesis and for your overwhelming encouragement when I was about to give up.

To Tomasz: Thank you for offering me such an interesting topic and for all your good deeds including the arrangement of workspace and notebook, helping me to structure my thoughts, your tips, suggestions and guidance during my thesis even outside of office hours.

Last but not least to Andreas Rauber: Thank you for pushing me to finish my thesis and giving me a lifetime opportunity to continue my studies as a PHD student.

Kurzfassung

Um zitierbare Daten und reproduzierbare Ergebnisse zu ermöglichen hat die RDA Data Citation Working Group 14 Empfehlungen veröffentlicht. Diese Empfehlungen wurden von Datenzentren für verschiedene Backend-Technologien übernommen. Bisher befinden sich RDF*-Stores bzw. Triple-Stores nicht unter diesen Backend-Technologien. In dieser Masterarbeit behandeln wir die Empfehlungen im Bezug auf RDF*- und Triple-Stores, designen ein RDF* Data Citation Framework, implementieren einen Prototypen des vorgestellten Frameworks und evaluieren ihn. Um Versionierung & Timestamping auf Triple-Ebene, mit dem Ziel die Anzahl der zusätzlich notwendigen Triples gering zu halten, zu implementieren, verwenden wir RDF* und SPARQL*. Mit Funktionstests zeigen wir, dass unser Prototyp den Empfehlungen genügt. Zusätzlich stellen wir Tests und Ergebnisse zur Laufzeit-Performance und zum Speicherverbrauch zu Abfragen von Live- und historischen Daten zur Verfügung, welche auf zwei versionierte RDF*-Datenbasen (FHIR und DBPedia) ausgeführt werden. Als RDF*-Store und Ablage für die zwei Datenbasen verwenden wir GraphDB. Die Ergebnisse deuten darauf hin, dass RDF* and SPARQL* für Versionierung & Timestamping verwendet werden können und dass Datenbasen, die über die Zeit unterschiedlich angereichert wurden (Insert vs Update) sich auch unterschiedlich auf die Performance der Abfragen auswirken. Zu weiteren Einflüssen zählen die Implementierung der Filter und Joins in der Abfrage (engl. Timestamped Query) und die Größe der Daten- und Ergebnismenge.
Um die "Query Uniqueness"-Empfehlung zu implementieren, welche darauf abzielt semantisch identische Abfragen mittels Normalisierung der Abfrage zu entdecken, zeigen wir, die SPARQL-Query-Algebra vom W3C im Normalisierungsprozess eingesetzt werden kann. Wir behandeln und erwägen "Query Containment Solver" vom Stand der Kunst als Alternative, um semantisch identische Abfragen zu erkennen. Wir evaluieren zwei Query Containment Solver, JSAC und SpeCS, und vergleichen diese mit unserer SPARQL-Query-Algebra-basierten Implementierung. Die Ergebnisse deuten darauf hin, dass unsere Implementierung die höchste Abdeckung für SPARQL Abfragen der Version 1.1 hat und dass JSAC ein potenzieller Kandidat für die Implementierung von "Query Uniqueness" ist, sollte dieser "SPARQL 1.1"-konform werden.

Unsere Implementierung ist auf Github verfügbar:
`https://github.com/GreenfishK/DataCitation`

Abstract

To facilitate citable data and reproducible results The RDA Data Citation Working Group published 14 recommendations. These recommendations were adopted by data centers for different back-end technologies. So far, RDF* stores or triple stores are not among these back-end technologies. In this thesis we discuss the recommendations for RDF* and triple stores, design an RDF* Data Citation Framework, implement a prototype of the proposed framework and evaluate it. To implement the versioning & timestamping recommendations on triple-level, with the aim to keep the number of additionally required triples low, we employ RDF* and SPARQL*. With functional tests we show that our prototype is in accordance with the recommendations. We furthermore provide runtime performance and memory demand tests and results on querying live & historical data from two versioned datasets, namely FHIR and a DBPedia dataset. We imported the datasets into GraphDB, which we use as RDF* store. The results suggest that RDF* and SPARQL* can be used for versioning & timestamping and that the perfomance differs for datasets that were enriched with insert statements from those were update statements were used, even though the number of additionally added triples is the same for both. The performance furthermore depends on the way filters and joins are used in the timestamped query and on the dataset & result set size.
To implement the Query Uniqueness recommendation, which aims to detect semantically identical queries by means of query normalization, we show how W3C's SPARQL Query Algebra can be used in the normalization process. We consider and discuss state-of-the-art Query Containment Solvers as alternative approach to detecting semantically equivalent queries. We evaluate two of them, namely JSAC and SpeCS, and compare them with our SPARQL Query Algebra based implementation. The results suggest that our implementation has the highest coverage for SPARQL 1.1 queries and that JSAC has the potential to be used for implementing Query Uniqueness, once it becomes SPARQL 1.1 compliant.

Our implementation is available on Github:
`https://github.com/GreenfishK/DataCitation`

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Aim and Scope . . . 2
1.4 Methodological Approach . . . 3
1.5 Thesis Structure . . . 3

2 Related Work **5**
2.1 RDA Data Citation Recommendations . . . 5
2.2 RDA Data Citation Implementations . . . 6
2.3 RDF Metadata Representations . . . 9
2.3.1 Standard Reification . . . 9
2.3.2 Named Graphs . . . 10
2.3.3 Singleton Property . . . 11
2.4 Statement-level metadata with RDF* . . . 11
2.5 Query Containment Solvers for SPARQL . . . 13
2.5.1 AFMU . . . 14
2.5.2 SPARQL Algebra . . . 15
2.5.3 JSAC . . . 15
2.5.4 SpeCS . . . 15
2.6 Creating automated citation snippets . . . 16
2.7 Dataset Identification . . . 18

3 Design **19**
3.1 Requirements and Constraints . . . 19
3.1.1 Product requirements . . . 20
3.1.2 Constraints . . . 26
3.1.3 Non-functional requirements . . . 27

3.2 RDF* Data Citation Framework . . . 28
3.2.1 RDF* Store Utilities . . . 29
3.2.2 Persistent Identification Utilities . . . 31
3.2.3 Query Store Utilities . . . 32
3.2.4 Prologue Handler . . . 33
3.2.5 Query Handler . . . 34
3.2.6 Query Builder UI . . . 36
3.2.7 Landing Page . . . 37
3.2.8 Data Management . . . 38
3.3 Compliance with RDA Data Citation Recommendations . . . 38
3.4 Summary . . . 39

4 Implementation **41**
4.1 RDF* Data Citation API . . . 41
4.1.1 rdf_star module . . . 41
4.1.2 query_store module . . . 48
4.1.3 persistent_id_utils module . . . 48
4.1.4 Query_handler module . . . 58
4.1.5 Prefixes module . . . 62
4.2 Build and Distribution . . . 62
4.3 Summary . . . 63

5 Evaluation **65**
5.1 RDF* Data Citation Framework . . . 65
5.1.1 Functional Tests . . . 66
5.1.2 Non-functional Tests . . . 70
5.2 Detecting semantically equivalent queries . . . 76
5.2.1 SpeCS . . . 77
5.2.2 JSAC . . . 79
5.2.3 Results . . . 81
5.3 Summary . . . 83

6 Conclusion and Future Work **89**
6.1 Conclusion . . . 89
6.2 Future Work . . . 91

List of Figures **93**

List of Tables **95**

Listings **95**

Bibliography **99**

CHAPTER 1

Introduction

1.1 Motivation

Over the last years data has become increasingly central and critical in both research and in application. With the rapid transition towards the fourth paradigm of science (i.e., data-intensive scientific discovery) [HTT+09], where data is as vital to scientific progress as traditional publications are, challenges like data provenance, identifying subsets, authorship of data, evolution of data over time and long-term data preservation become apparent. Overcoming these challenges is an important cornerstone to reproducibility and verification of research results.
Many contributions have already been made to citing data coming from relational models, such as using well-established approaches from data-warehousing [Aro07], and XML hierarchical models [Sil17]. However, addressing citation challenges like versioning and timestamping with linked data comes with additional problems due to the means that are used to represent such data. The most common means are RDF (=Resource Description Framework) stores, which use simple triples (subject, predicate, object) that can be linked with each other for knowledge representation. To extract knowledge a query language known as SPARQL is commonly used in conjunction with RDF stores. Some solutions like the annotation approach had been proposed in the literature [NDP15]. Important preliminaries to data citation had been covered by the *data citation working group* in form of recommendations [vUSP16] and also been endorsed and adopted by various data centers. So far, no implementation of aforementioned recommendations has been devised for linked data and triple stores [dat19]. To address the challenges of data citation for triple stores (RDF stores) we propose a data citation framework which covers the first twelve *recommendations* by making use of extensions to RDF and SPARQL, which have recently been introduced, and other concepts from the literature [Har17].
For the remainder of this document we write *recommendations* when we refer to the recommendations in [RAVUP16a]. In addition, RDF stores and triple stores are used interchangeably.

1.2 Problem Statement

The RDA Data Citation Working Group published a set of 14 recommendations [RAVUP16a] to facilitate citable data and reproducible results. Different data centers [dat19] adopted the recommendations in very individual and heterogeneous ways. Used storage technologies include relational databases but also NetCDF files, Git and others. As the recommendations do not prescribe a specific type of data storage they can also be applied to RDF stores. As opposed to relational databases, citing linked open data originating from triple stores comes with a set of difficulties not met in the former storage systems, such as referencing whole triples, separating data from metadata or identifying subgraphs. There are various RDF-based proposed solutions [PFF+16], [WCEGL13], [Har17] to tackle the recommendations' underlying data-specific problem, such as data versioning, timestamping query and normalization (query containment solvers).
Even though solutions to these individual challenges exist, adopters would still need to lump together available methods and techniques on their own and make sure that they are compatible with each other. Moreover, one has to deliberate on an algorithm and thereby consider the control flow for these *recommendations*. As of today, there exists no compiled set of routines nor process model to implement these *recommendations* for triple stores.

1.3 Aim and Scope

The main goal of this work is to enable data citation within triple stores by creating a conceptual framework with methods and techniques on lessons learned from linked open data and relational data citation solutions. Furthermore, we want to learn from adopters that are targeting data citation problems underlying the *recommendations* by the *Data Citation Working Group*. At this point, the distinction between *recommendations* and *recommendation problems* needs to be pointed out. While *recommendations* answer the question *what* needs to be done, they do not always imply *how* it should be done. Therefore, the theoretical part elaborates aforementioned methods and techniques by focusing on the *how* in the context of RDF stores. It also discusses different approaches, their benefits and potential shortcomings of proposed solutions that can be mapped to individual *recommendations* (the *what*). The practical work builds on this elaboration and results into a prototype of the conceptual framework which defines routines and functions to be followed and used to implement the *recommendations* for RDF stores. This framework enables citing data sets by generating a citation snippet with a single function call given a SPARQL query. Moreover, we discuss and evaluate two means to detecting semantically equivalent queries as needed for the R4 - Query Uniqueness recommendation. One of them is W3Cs Query Algebra [1] and the other are Query Containment Solvers. We therefore formulate following research questions which we address in this work:

RQ1: What is the best way to use SPARQL* and RDF* to implement *Data Versioning*

[1] https://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html

and *Operation Timestamping* with the aim to keep the number of additionally required triples low?

RQ2: Which of the methods for detecting semantically equivalent queries yields the highest coverage?

RQ3: Which of the *recommendations* can be covered by the framework and which ones remain specific to the target system?

1.4 Methodological Approach

We follow the approach from [Hev07] which is specific to design sciences and defines three cycles - the relevancy cylce, the rigor cycle and the design cycle. During the relevancy cycle we research and collect technical requirements of triple store vendors and also requirements related to metadata collection and representation. Moreover, we seek out problems and opportunities from the target environment, which might encompass people like data operators and researchers but also technical interfaces like landing pages and triple store query editors. As the relevancy cycle also stipulates measuring improvement we define evaluation scenarios, measures and functional tests to be used during the field testing and evaluation processes.
In the rigor cycle we draw knowledge from state of the art theories and solutions that are either related to specific *recommendations* problems or were already used to solve them. This endows us with the right tools, techniques and methods not only to answer RQ1 but also to incrementally build our prototype. New methods to implement the *recommendations* that arise from our research, as a result, is added to the scientific knowledge base.
With the requirements from the relevancy cycle and the grounding from the rigor cycle we enter the design cycle where we incrementally build our RDF Data Citation Framework and furthermore an API (=Application Programming Interface) as an executable instance. Upon every evaluation of the API we refine the framework and repeat this feedback loop until all defined requirements from the field studies are fulfilled. While the evaluation during the feedback loop focuses on functionality and features the subsequent evaluation measures the performance of the API. During the design cycle and more particular evaluation process we dedicate our research to questions RQ2 and RQ3.

1.5 Thesis Structure

We introduce the RDA data citation recommendations and relevant concepts revolving around them in Chapter 2. These include RDF* and SPARQL* as well as query containment solvers for SPARQL which are fundamental for our main contribution of this work and part of the design cycle. In Chapter 3 we start with setting the scope by defining use cases and setting requirements and constraints to our RDF* Data Citation Framework. Right after we give an overall view of the framework's design and subsequently elaborate its components, thereby explaining its required inputs and offered features & functions. At the end of this chapter we exhibit how our framework fits into the RDA data citation recommendations. In Chapter 4

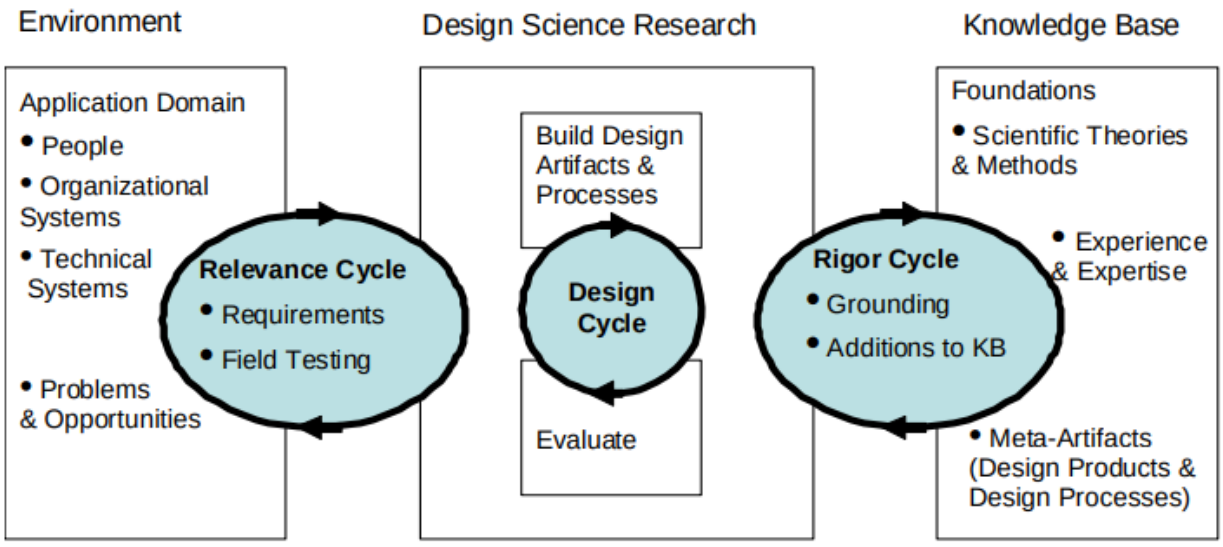


Figure 1.1: The Design Science Research Cycles [Hev07]

we instantiate the RDF* Data Citation Framework by means of rigor and concept from the previous respective chapters. We specifically show how we implemented the interfaces from Chapter 3 and how we solved different *recommendations* underlying problems for RDF* stores. This implementation we evaluate in 5 in two steps. First, we provide functional tests to lay out its core functionalities. Second, we outline evaluation scenarios we used for our runtime and memory performance evaluation and also present the evaluation results. We furthermore evaluate two Query Containment Solvers as means to R4 - Query Uniqueness. Finally, we revisit the research questions and give an outlook of our future work, which contain new ideas related RDF* Data Citation and also improvement possibilities of our current artefacts.

CHAPTER 2

Related Work

2.1 RDA Data Citation Recommendations

The RDA Data Citation Recommendations [RAVUP16b] are the backbone of our work. They were introduced in 2016 by the RDA Data Citation Working Group with the goal to keep research experiments reproducible even if the data source is continuously evolving. Reproducing and verifying research results is necessary for the research method in general and it is inevitable for assessing the validity of the experiment. The recommendations are based upon versioned data, timestamping and a query subsetting mechanism and are grouped into four areas which we show in Table 2.1.
The first area prepares the data for identification which is achieved by means of versioning and operation timestamping, latter comprising create, update and delete operations. It moreover sets up a query store for storing query and associated metadata. These metadata data are among others results from operations included in the next two areas.
The next area can be considered as a subgoal of reproducible research results, namely Persistently Identifying Specific Datasets. This area comprises several means which take query or dataset as input and produce a piece of information which helps in achieving this subgoal. We found that some of the recommendations from this are can be related to specific studies or research fields. R4 - Query Uniqueness is closely related to Query Containment Solvers, which we elaborate in 2.5. Assigning a PID to the query (R8) is the RDA DCWG's approach to the dataset identification problem. In the literature other proposals can be found, such as Research Resource Identifiers (RRID) [BBG+16] or named graphs [C+14]. Latter we discuss in Section 2.3.2. Creating automated citation snippets (R10) has been defined as a computation problem [BDF16] and been solved using an approach called *citation views* [ACD+17].
In the third area - Resolving PIDs and Retrieving the Data, we find ourselves on the presentation layer. The idea is to present data and metadata to both - humans (R11) and machines (R12) in a human-readable and machine-actionable way, respectively. Usually, the metadata comes from the query store while the data is retrieved upon query re-execution. The re-execution is

triggered by a download button on the landing page. Data and metadata need to be further processable, e.g. by workflow engines, without the interaction of humans, hence the machine-actionable representation.
The fourth area completes the change management as it considers that next to data, technology can change, too. This includes rewriting the queries and re-computing fixity information if the data gets migrated to a new environment. Each migration, of course, needs to be verified, thus it must be ensured that the queries are re-executable and yield the same result as before the migration.
By now, we should have a solid understanding of how the *recommendations* contribute to making and keeping research experiments reproducible and can proceed with specific implementations by adopters in the next section.

Table 2.1: RDA Data Citation recommendations[vUSP16]

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

2.2 RDA Data Citation Implementations

The RDA Data Citation Recommendations are since their publication in 2016 being adopted by various data centers across different science domains ranging from the geo science, over biomedical science to forest ecosystems. We also found distinct data and dataset types by each of the adopters which might be drivers to their heterogeneous implementation approaches. In the following we are going to outline the most noticeable characteristics of the respective solutions and at the end of this section we show a table of addressed *recommendations* by the individual adopters.

The Washington University Center for Biomedical Informatics implemented Dynamic Data Citation for their Electronic Health Records (EHR) datasets. They approached Versioning and Timestamping by employing Postgre's temporal_tables extension. For each live data table that should be versioned there is a paired history table where records are moved to when newer versions of these records are created. The period as of which the record is valid is given by a timestamp range which is stored in a single column [GZR+17].
The Vermont Monitoring Cooperative used two forms of versioning for their Ecosystem Monitoring Collaborator Network, namely Dynamic Subsetting and Provenance Tracking. While they apply former method only to tables, latter form of versioning is applied to all types of data (tables, binary files, images, ...). They use a step tracking table, which can be related to the query store, to record every state of a table. The way this table is implemented is reminiscent of a doubly linked list. Each record in the step tracking table has two DML statements attached in separate columns - one for moving to the previous state and one for moving to the next state. E.g. if the difference between $state_1$ and $state_2$ is one additional record in $state_2$ then a delete statement is attached to $state_2$ deleting that record and an insert statement is attached to $state_1$ to add that additional record and restore $state_2$. Their solution also allows to switch off versioning allowing for more changes to the dataset without tracking them [JD].
The Climate Change Centre Austria adopted the *recommendations* for their weather and climate data services. They use HTTP GET requests as queries to retrieve NetCDF files as geospatial datasets of high resolution climate scenarios. The queries have a small fixed set of required arguments and these are also reflected in the query's persistent identifier (query PID). In their solution it is also possible to create a new semantically identical subset version based on a different version of the original data. On the landing page, they display the citation snippet, metadata the subset version history and a link to the superset [SB19].
Gößwein *et al.* managed to introduce and implement Dynamic Data Citation at the Earth Observation Data Centre. The main characteristic is that versioning and timestamping was not only implemented for data but also for the execution environment where certain jobs are run. These jobs might produce different results when the environment changes, even though the input data stays the same. They achieved this by employing VFramework's context model to store static and dynamic data collected on the environment and job execution [GMRW19].
The Ocean Network Canada offer a GUI to issue queries with the aim to extract information about their deployed sensing devices. Similar to CCCA, there is a fixed set of arguments, such as sensor, time range and file format, that guides the user through the query. Every data search (query) is saved in the database. It is, however, not clear whether they use a separate query store or not. They define so called versioning tasks to reprocess or replace faulty data after certain fixes or parameter changes. Every-time such a task is executed a new version is minted along with a new DOI and linked to the previous version of the dataset. The version history is then shown on the landing page, along with metadata and a citation snippet. Their metadata attributes are a subset of attributes from DataCite's Metadata Schema[1] [RJ].
The Virtual Atomic and Molecular Data Centre extracts data as so-called VAMDC-XSAMS files. VAMDC-XSAMS is an extension of the XSAMS format (XML for atoms). It carries the *Version*

[1] https://schema.datacite.org/meta/kernel-4.3/doc/DataCite-MetadataKernel_v4.3.pdf

element as additional information which stores changes between two *data nodes* versions. Latter are federated databases used within the VAMDC infrastructure. The VAMDC-XSAMS files do not only carry data but also metadata, such as links to datasets that were used to compose the result, bibliographic information as well as other metadata. These metadata are stored in a query store using a query store service and associated with an Universal Unique Identifier (UUID). This service might also delete these metadata after the query has not been executed for an arbitrarily defined period of time (E.g. 5 years). This is because not every query that is stored in the query store has been used in published work and therefore has no DOI assigned. The user needs to manually assign a Digital Object Identifier (DOI) by clicking a "Get a DOI" button on the query's landing page and thereby triggering an upload of data and metadata to ZENODO, a service which the query store is interconnected with. The deletion process is suspended for all queries that have a DOI assigned [ZMBD19].

To provide a tangible summary of the covered recommendations by the mentioned adopters we created Table 2.2. An "x" in this matrix suggests that the adopter/implementation either directly addresses the corresponding recommendation or shows and discusses a concept that is closely related to the recommendation or can be easily mapped to it (e.g. VMC's step tracking table, naming it UUID instead of query PID). A empty cell means that to the best of our knowledge we could not find any evidence for the coverage of this recommendation. It is noticeable that none of the adopters addressed R13 and R14. The reason might be that technology migration has not been their main concern when firstly implementing Dynamic Data Citation because the implementations where carried out with fixed technologies. Though, it is something they might consider in future either because of preventive measures in place or because technology migration is due.

Table 2.2: Recommendations and adopters/implementations

RDA Recommendation	WU centre	VMC	CCCA	EODC	ONC	VAMDC
R1	x	x	x	x	x	x
R2	x	x	x	x		x
R3	x	x	x	x	x	x
R4		x	x	x		x
R5				x		
R6		x		x		
R7	x		x	x	x	x
R8	x	x	x	x	x	x
R9	x	x	x	x	x	x
R10	x		x	x	x	x
R11		x	x	x	x	x
R12			x	x		x
R13						
R14						

As we see, the Data Citation Recommendations are not limited to specific storage systems or datasets and could thus also be applied to RDF stores. However, when it comes to versioning and timestamping on statement-level we face challenges to optimize measures like query performance and required storage as we see in the next section.

2.3 RDF Metadata Representations

When it comes to representing metadata in RDF a few prominent approaches have been discussed and used so far, which include standard reification [2], named graphs [nam14], singleton properties [NBS14] and others. While standard reification and named graphs exploit properties of RDF and do not need the data triples to be stored in a certain way, the singleton property relies on a certain data representation. Experiments and evaluations have been conducted on different graphs, such as Wikidata, and different triple store vendors, such as GraphDB, Jena, Virtuoso and others [HHK15][OGO21] to compare their performances in terms of data retrieval but also memory consumption. Due to the need of additional triples and verbose query constructs to match the metadata none of the approaches are suited for real world scenarios where big datasets are used. In the following subsections we discuss their concepts, benefits and shortcomings before moving on to an alternative and more recent approach which offers significant benefits over these three standard methods (See in Section 2.4).
For the remainder of this chapter we use following running example and show how the different approaches could be used to represent facts about data triples:

> Natural language statement: "*Obama was a president between 20.01.2009 and 20.01.2017*"
> Data triple: :Obama :occupation :president
> Facts about triple representation:?

We refer to this data triple also as our example dataset. We use the term *entities* when referring to single triple parts independently of what those parts actually are. E.g. ":Obama" is an entity. When we write about the grammatical constructs subject, predicate and object in an abstract way we refer to them as *components*. E.g. ":Obama" is an entity and this entity is a subject. A subject is a component of a triple. Usually, entities are represented using IRIs (Internationalized Resource Identifiers) [3]. One example would be http://example.com/Obama. It is common practice to abbreviate IRIs by using a prefix that resolves to the IRI's namespace, for example: ex:http://example.com/. "ex" is the prefix and the http://example.com the namespace. We can then abbreviate our example IRI with: ex:Obama. To explain certain concepts we do not need to outline the namespace. That is why we leave out the prefix in our running example.

2.3.1 Standard Reification

Standard reification is a way to "zoom out" of triples and represent each fact about them along with other facts on the same level. This approach exploits the blank node property of RDF

[2] https://www.w3.org/TR/rdf-schema/#ch_reificationvocab
[3] https://www.w3.org/TR/rdf11-concepts/#section-IRIs

graphs where blank nodes (= nodes without an URI or literal) are used as subjects to refer to the triples' entities literally using their components (subject, predicate, object) as predicates. To add metadata to a triple these blank nodes can then be stated together with key-value pairs to form additional triples. We say that a triple is reified when it is rewritten using four additional triples as follows:

```
_:x1 rdf:type rdf:Statement .
_:x1 rdf:subject :Obama .
_:x1 rdf:predicate :occupation .
_:x1 rdf:object :president .
```

Now, we can add the presidential term as metadata to the reified triple:

```
_:x1 :from "2009-01-20"^^xsd:date .
_:x1 :until "2017-01-20"^^xsd:date .
```

If we would like to reify a second triple we would use the same procedure and just choose another blank node, e.g. _:x2.
We see that we did not need to introduce any new syntax to reify the triple and add metadata to it, which is what makes this approach compliant to the RDF standard. However, we create inefficiencies in terms of storage which is why this approach is discouraged in Linked Open Data [KCG16] [Har17] [met] [OGO21] [HHK15].

2.3.2 Named Graphs

A named graph is an RDF graph that has a name assigned in the form of an Internationalized Resource Identifier (IRIs). IRI is an extension of URI where characters from an universal character set are allowed instead of being limited to ASCII. This name or IRI can then be referenced in other graphs, e.g to add metadata onto it. Named graphs are closely related to Quad Semantics which use a fourth component to add context to triples. In fact, named graphs can be seen as a reformulation of Quad Semantics where the fourth component is moved to a distinct name property [CBHS05a]. This allows for a clearer distinction between the data and the context. Though, either formulations have the same expressiveness when IRIs are used. Using named graphs or quad semantics to express our running example would look like the following:

```
//Named graphs
ex:x1 {
:Obama :occupation :president.
}
//Adding metadata
ex:x1 :from "2009-01-20"^^xsd:date .
ex:x1 :until "2017-01-20"^^xsd:date .
```

```
//Quad semantics
:Obama :occupation :president ex:x1
//Adding metadata
ex:x1 :from "2009-01-20"^^xsd:date .
ex:x1 :until "2017-01-20"^^xsd:date .
```

Compared to standard reification we see a reduction in the number of triples and improvement

in readability as the data triple does not need to be re-expressed. The benefit becomes more visible if we add more data triples to the named graph and then use the graph's IRI to describe all triples within that graph at once, instead of describing every reified triple individually. One disadvantage that named graphs might have is the potential confusion due to different use cases of the name property. It can be used as a graph identifier, as we did it, but also as a means of identifying the document and as a retrieval URL or source of the triples [CBHS05b]. Named graphs have also been proposed for Data Citation to uniquely identify data sets by means of so called "Citation meta-graphs". Figure 2.1 illustrates this idea in a three-step approach using Quad Semantics. In step 1 we assign distinct names to all triples, that belong to a graph we want to cite using the fourth component, thus the name property, labeled "Name" in this figure. As a consequence, named graphs are created, each consisting of exactly one triple. In the second step we create the citation meta-graph, an directed cycle graph with the named graphs from step 1 being connected by a predicate p_α in arbitrary order so that each named graph has exactly one incoming and one outgoing edge. This citation meta-graph is also a named graph and gets the name $citX$ assigned. Finally, in step 3, we can associate metadata to the citation meta-graph $citX$ and thereby create a reference graph named $refA$. $refA$ is an IRI which makes it possible to access the machine-readable metadata [Sil15].

2.3.3 Singleton Property

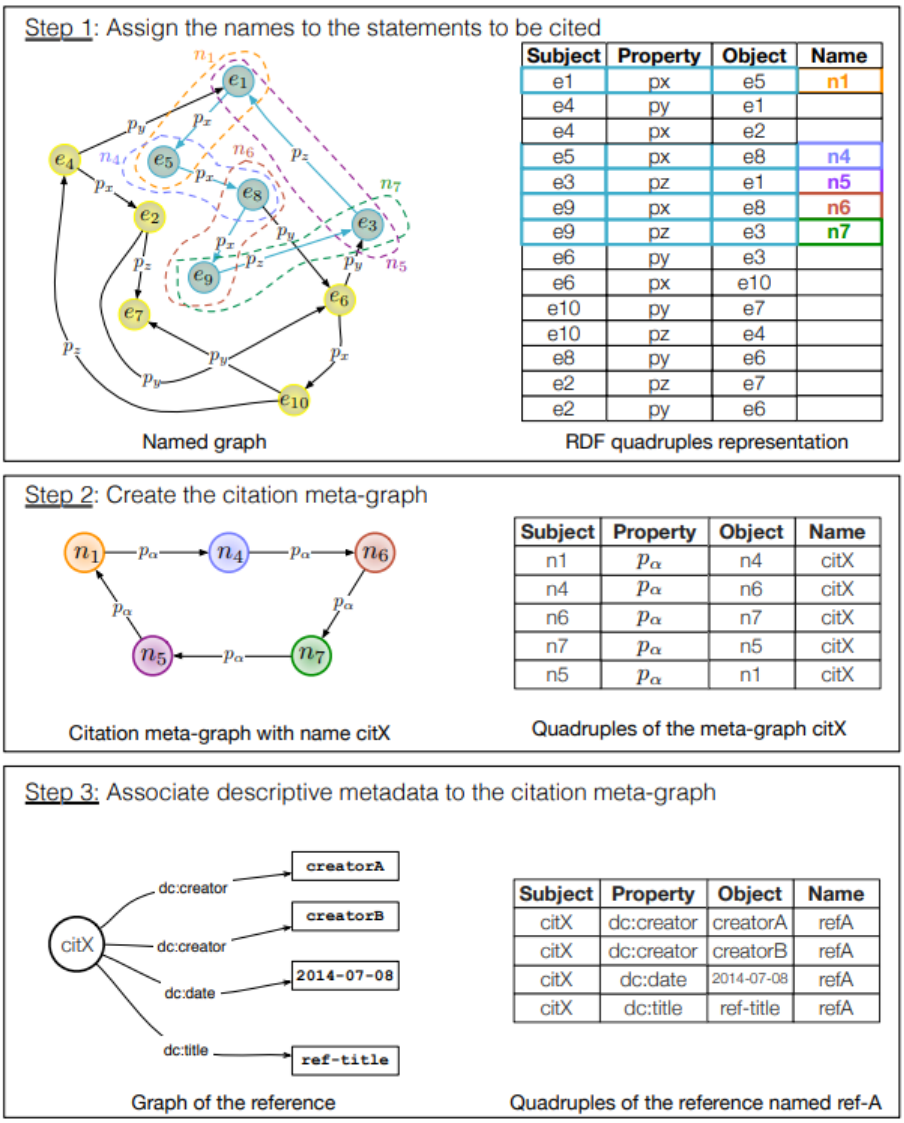
As in Standard Reification, the Singleton Property also uses formal semantics to define a model for representing data in triple stores that allows for a simple metadata representation. The idea is to have unique relationships between entities that represent each statement. "A singleton property is a property instance representing one specific relationship between two particular entities under one specific context"[NBS14]. Similar to blank nodes in standard reification, these instances can then be used for metadata annotations[NBT+15][NBS14]. With every instance one metadata attribute needs to be added that links the instances to their generic properties. In [NBS14] this metadata attribute or link is called *singletonPropertyOf*. Our running example modelled with the Singleton Property would look like this:

:Obama :occupation#1 :president .
:occupation#1 rdf:singletonPropertyOf :occupation .
:occupation#1 :from "2009-01-20"^^xsd:date .
:occupation#1 :until "2017-01-20"^^xsd:date .

Even though singleton properties are easy to query, they do not perform well for known graph databases, such as Virtuoso as evaluated in [HHR+16]. One reason is because it is untypical for RDF to have "a large number of unique predicates" and thus "disadvantageous for commonly-used SPARQL optimization heuristics" [Har17].

2.4 Statement-level metadata with RDF*

Statement-level metadata refers to metadata that is associated with a single statement or triple, e.g. :Obama :occupation :president. These metadata can be added in the form of key-value



Subject	Property	Object	Name
e1	px	e5	n1
e4	py	e1	
e4	px	e2	
e5	px	e8	n4
e3	pz	e1	n5
e9	px	e8	n6
e9	pz	e3	n7
e6	py	e3	
e6	px	e10	
e10	py	e7	
e10	pz	e4	
e8	py	e6	
e2	pz	e7	
e2	py	e6	

Subject	Property	Object	Name
n1	p_α	n4	citX
n4	p_α	n6	citX
n6	p_α	n7	citX
n7	p_α	n5	citX
n5	p_α	n1	citX

Subject	Property	Object	Name
citX	dc:creator	creatorA	refA
citX	dc:creator	creatorB	refA
citX	dc:date	2014-07-08	refA
citX	dc:title	ref-title	refA

Figure 2.1: The three-step methodology for citing LOD subsets

pairs and linked to statements. While graph databases have edge properties as means for statement-level metadata, RDF or triple stores, which present another graph-based approach, do not natively support this. As outlined in the previous section one can either define formal models based on existing RDF 1.1 syntax that require data and metadata statements to follow certain semantics or exploit RDF properties to add this feature to triple stores. On top, queries also need to be designed and written in a cumbersome way in order to match data and corresponding metadata triples for the respective metadata representation models [Har17]. RDF* and SPARQL*, as proposed in [Har17], work with a paradigm that we can encounter in various concepts from the informatics and mathematics domain, such as SQL & SPARQL

queries, XML and first-class functions. This paradigm are nested triples which allow for nesting a whole triple into the subject or object of another triple. Moreover, it introduces a new syntax which we show below:

«:Obama :occupation :president» :from "2009-01-20"^^xsd:date .
«:Obama :occupation :president» :until "2017-01-20"^^xsd:date .

In order to query data and metadata from our example graph above with SPARQL* we can follow the same semantics as in SPARQL just that the subject is now a nested triple written in the same syntax as in RDF*:

```
select ?name ?occupation ?valid_from ?valid_until where {
    <<?name ?p ?occupation>> :valid_from ?valid_from;
                             :valid_until ?valid_until.
}
```

An RDF* graph G* can be both - redundant and redundant free. In the former version a data triple t' that is contained in a metadata triple t would also be directly contained in G*. Let us first illustrate this below with the triple from our running example as t':

//Redundant graph	//Redundant-free graph
:Obama :occupation#1 :president	«:Obama :occupation :president» :from
«:Obama :occupation :president» :from	"2009-01-20"^^xsd:date .
"2009-01-20"^^xsd:date .	«:Obama :occupation :president» :until
«:Obama :occupation :president» :until	"2017-01-20"^^xsd:date .
"2017-01-20"^^xsd:date .	

Regardless of whether t' is directly contained in G* or not, there is no difference in terms of information content. Hence, G* is redundant if t' and t are both contained in G* [Har17]. SPARQL* and RDF* can be reduced to SPARQL and RDF respectively. More importantly, it is possible to map RDF* to Standard RDF Reification. Casting our eyes over to R13 - Technology Migration, it could be desired to automatize reducing and mapping of SPARQL* and RDF* when migrating to a technology where "the star extension" is not supported [Har17].

2.5 Query Containment Solvers for SPARQL

Query containment solvers answer the question whether one query is subsumed by another. If this is reciprocally true, we can say that two queries are equivalent. So, query containment solvers are a fitting means to tackle R4 – Query Uniqueness, next to their other purpose which is optimizing queries. To the best of our knowledge, five SPARQL query containment solvers have been proposed in the literature. Four of them deal with SPARQL queries [SVJ20][CEGL18][LPPS13][4] whereas one deals with XPath. Latter we will not discuss as XML is out of scope of this work. queries[GL06]. The query containment solver based on *AFMU*

[4] https://github.com/SmartDataAnalytics/jena-sparql-api/tree/develop/jena-sparql-api-query-containment

Table 2.3: Query Containment Solvers and supported features for SPARQL-Algebra, AFMU, TreeSolver [WCEGL13] and two more recent Solvers

Solver	projection	union	optional	blanks	cycles	RDFS
SPARQL-Algebra			x		x	
AFMU	x	x		x		x
TreeSolver	x	x		x		x
JSAC	x	x	?	x	?	
SpeCS	x	x	?	x	x	x

[CEGL18] is only a theoretical foundation but there exists no implementation so far. *SPARQL Algebra*, a query solver using trees that resemble a query execution plan, was also realized and published. A link to this solver is available but broken [SVJ20]. This solver stays inaccessible at the time of writing.
An evaluation of supported features by the different solvers has been done in 2013 for solvers available at that time using a query containment benchmark [WCEGL13]. The features included in the benchmark were picked with respect to the capability of these solvers, namely: projection, union, blank nodes, andquery containment under RDFS reasoning. Within this benchmark suite three test suits were designed with each containing multiple queries characterized by these features. OPTIONAL and cycles were not included in the benchmark design as SPARQL-Algebra was the only solver able to deal with these paradigms at time of evaluation. In Table 2.3 we extended the original table presented in the evaluation [WCEGL13] by two, more recent, solvers, JSAC and SpeCS. Both solvers were evaluated using aforementioned benchmark suite. For neither of them we could not conclude whether OPTIONAL is supported. However, Spasić *et al* argue that the SMT (=Satisfiability modulo theories) solver they employ for SpeCS can deal with cycles by nature [SVJ20].

2.5.1 AFMU

The "AFMU Query Containtment Solver" is based on the alternation-free (AF) fragment of the modal μ-calculus introduced in 1982 which can be used to describe properties of transition systems [HKP82]. Chekol *et al* turned an RDF graph into a transition system (p is the transition between s and o) and encoded queries and schema axioms as μ-calculus formulae. These formulas are then interpreted over the transition system. Overall, they reduced the containment problem to a satisfiability and validity problem [CEGL18]. To solve the satisfiability problem of the μ-calculus (nominal, functional and backwards) they employed the decision procedure developed by Tanabe *et al* and thereby mainly focused on nominal and backwards modalities [TTY+05]. The query containment solver cannot solve any given two queries but does support

only following features: blank nodes, acyclic union conjunctive queries, projection and RDFs. It is also just a theoretical work with no accessible online tool or prototype at the time of writing [CEGL18].

2.5.2 SPARQL Algebra

While the *SPARQL Algebra* is already part of the SPARQL 1.1. W3C recommendation [5] there is another work [LPPS13] also describing an algebra for SPARQL which is seemingly not connected to W3C's algebra. However, the ideas are closely related as both represent graph patterns as trees. One important use-case that the authors in [LPPS13] outline is the construction of query execution plans. The tree patterns they design are reminiscent of such execution plans known from SQL and should form a basis for query optimization in SPARQL. The authors establish syntactic an semantic relationships between pattern trees and SPARQL graph patterns. They furthermore define rules which are applicable on query trees to e.g. "eliminate redundancy". Hence, these rules help to normalize these trees. They also discuss containment and equivalency and bring out that containment can be used to test for equivalency but is not recommended as containment problems are more complex (π_2^P-complete vs NP-complete) [LPPS13]. SPARQL-Algebra supports OPTIONAL and cycles but does not feature UCQs, blank nodes and RDFS. A link is available to an online implementation of SPARQL-Algebra, however, the web-page is not accessible anymore.

2.5.3 JSAC

JSAC Query Containment Solver is a subgraph isomorphism checker. It uses a "bottom-up algebra-tree matching approach" to solve the problem of query containment. It thereby takes two trees, the "normalized algebra expression tree" and "a subtree of a superquery" to check for a subgraph isomorphism [6]. A tool is available on github [7] and it includes a feature which can be seen as an approach to R4 - Query Uniqueness. Instead of normalizing queries and computing checksums one could use the tool's query containment solver in combination with a feature called transparent sub graph isomorphy cache which detects whether prior result sets fit into a current query. On their github page there is no information about which SPARQL features the JSAC query containment solver supports. Though, we can see from the benchmark test in [SVJ20] that it does support union, projections and blank nodes but does not support RDFS.

2.5.4 SpeCS

The SpeCS query containment solver[SVJ20] builds on similar ideas as AFMU as it also uses logic and reduces the containment problem to a satisfiability problem. In contrast to the

[5] https://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html

[6] https://github.com/SmartDataAnalytics/jena-sparql-api/tree/develop/jena-sparql-api-query-containment

[7] https://github.com/SmartDataAnalytics/jena-sparql-api/tree/develop/jena-sparql-api-query-containment

propositional modal logic used with the μ-calculus SpeCS relies on theories in first order logic. It transforms SPARQL queries to a first order logic formula and then runs a SMT (satisfiability modulo theories) solver to check for containment. The authors in [SVJ20] also see the equivalency problem as a containment problem where reciprocal containment means equivalency. However, they do not make mention of the unequal complexities between containment and equivalency as the SPARQL Algebra authors do [LPPS13]. Their own evaluation showed that SpeCS performs better then the previously discussed solvers in terms of execution time and number of solved test cases. The test cases include following features: projection, union, blank nodes, cycles and RDFS. Their solver [8] is also available online for download.

2.6 Creating automated citation snippets

A citation snippet in the sense of Data Citation is a collection of information that identify the dataset (e.g. title, query PID [vUSP16]) and give attribution to contributors (e.g. authorship, ownership, date, ...) [BDF16]. Researcher did on the one side investigate the dependencies of a citation snippet and argued that the query is a means of identifying the dataset and also provides context information, which is why the citation pre-dominantly depends on it. The dataset can be empty or a bunch of image files, which is why it is not a reliable source of information for the citation snippet [BDF16][ACD+17].
On the other side, the creation of citation snippet templates and citation formats for specific subsets has been investigated. As subsets (or subgraphs in RDF) do vary in their granularity and contained information there is no "one fits it all" citation snippet template which always uses the same attributes, like we know it from citing books or articles. Buneman *et al* [BDF16] have demonstrated how to specify citable units, such as certain subsets, using databse views and create citations using a rule-based language on two databases, namely GtoPdb [gto] and MODIS [SBM06]. Alawini *et al* propose a similar solution with a citation framework which they applied to the eagle-i RDF dataset to proof their concept. They define *view queries* which predefine the dataset to be cited, *citation queries* which query information that are associated with the dataset and *citation functions* which use the output from *citation queries* and format it for the citation snippet [ACD+17].
Both solutions are similar in the way that they use views to define subsets and functions or rules to create citation snippets. However, Alawini *et al* also formalizes the extraction of citation data with the use of *citation queries*. We prepared an example to illustrate this idea. Instead of using an abstract language for the view and citation queries as Alawini *et al* did, we used SPARQL*. In Listing 2.1 and 2.2 we have two view queries. Both represent a specific subset and these subsets are also disjoint. Our example dataset, which by now also includes metadata, is now matched against these views (we do not show the matching process here). Only the view query from 2.1 would return a non-empty dataset. Therefore, the corresponding citation query, which we show in Listing 2.3, fires to extract the citation data. Note how the aliases or citation attributes from the other citation query in Listing 2.4 would differ from the first one. Lastly, a function would take these citation data as an input and produce an output

[8] www.math.rs/~mirko/SpeCS.tar.gz

as shown in Listing 2.5. For comparison, the citation snippet for the second view query would look slightly different, as we can see in Listing 2.6.

Listing 2.1: View Query1 example

```
Select ?s {
    ?s :occupation "president".
}
```

Listing 2.2: View Query2 example

```
Select ?s {
    ?s :occupation "formula-one-driver".
}
```

Listing 2.3: Citation Query1 example

```
Select (?s as ?president)
       (?x as ?president_from)
       (?y as ?president_until) {
    <<?s :occupation "president">>
        :valid_from ?x.
    <<?s :occupation "president">>
        :valid_until ?y.
}
```

Listing 2.4: Citation Query2 example

```
Select (?s as ?driver)
       (?x as ?career_from)
       (?y as ?career_until) {
    <<?s :occupation "formula-one-driver">>
        :valid_from ?x.
    <<?s :occupation "formula-one-driver">>
        :valid_until ?y.
}
```

Listing 2.5: Citation Snippet formatted with a Citation Function example1

```
{presidet: "Obama",
 president_from: 2009-01-20,
 president_until: 2017-01-20}
```

Listing 2.6: Citation Snippet formatted with a Citation Function example2

```
{driver: "Alonso", career_from: 2001-01-01}
```

Citation snippets might sometimes carry too little information or be underspeciefied. One example for underspecification is when different versions of the dataset were used but it is not clear which one. To automatically include all versions or linked datasets [MB15] proposed an ontology where citation snippets, subsets and supersets can all be linked. Links between subsets and supersets can then be inferred and underspecified citation snippets could then be enriched with information that is now available due to the network of datasets and citation snippets.

Sometimes data citation snippets would get too long and a trade-off would need to be made between completeness and length. If there are over 100 contributors to one dataset the citation snippet cannot include all of them. How does the algorithm decide where to make the cut? [FKS19] addressed this problem in the course of creating a data citation framework for nanopublications. They argues that metadata is not the only input needed for the citation snippets but "nanopubs curators" must also provide policies which define selection & ordering and operations on the data citation snippets. Moreover, they also define via the *presentation* component how the citation snippets are going to be presented which is reminiscent of *citation functions* in [ACD+17].

2.7 Dataset Identification

Most solutions from the literature propose a persistent identifier (PID) to identify datasets. The term PID is used as an umbrella term for all kinds of concrete implementations of persistent identifiers, such as DOIs, Digital Unique Identifiers (DUI), "named graph URI", Universal Numeric Fingerprint (UNF) and Research Resource Identifiers (RRID). Gianmaria Silvello outlines literature and solutions in [Sil18] in which these PIDs are used.
One option to use PIDs is to assign them to digital objects to be identified themselves. This includes datasets but also segments of data, XML nodes, subgraphs or single triples [Sil18]. Latter is the method used in the nanopublication model [GGV10]. Another option is to assign PIDs to queries associated with the digital object. More specific, we can assign a PID to a query that is used to retrieve the dataset. This is the solution preferred by [RAVUP16a]. The authors do not suggest any specific PID system, however, they do mention DOI and ARK as examples.
In Section 2.3.2 we have discussed named graphs in the context of statement-level annotations with the aim to enrich triples with metadata. As mentioned in this section, named graphs have also been proposed as a means of identifying RDF graphs. At this point, we need to outline the distinction between RDF datasets and RDF graphs and named RDF graphs.

> "An RDF graph is defined as a set of triples" [CBHS05c]
>
> "A Named Graph is an RDF graph which is assigneda name in the form of a URIref." [CBHS05c]
>
> "An RDF dataset is a $set = G,(u1,G1),(u2,G2),...(un,Gn)$ where G and each G_i are graphs, and each u_i is a URI. Each u_i is distinct. G is a called the background graph. G_i are namedgraphs." [PS05]

These distinctions are necessary to see that an RDF dataset can consist of multiple graphs, hence, we cannot rely on a single named graph URI to identify RDF datasets. Besides identification, other uses cases relatable to Data Citation, such as "ontology versioning and evolution", "Signing RDF graphs" and "Data syndication", have been outlined in [CBHS05c].

CHAPTER 3

Design

In this chapter we outline components of the proposed RDF* Data Citation Framework and their relationships to each other with respect to RDF* requirements and constraints. We start with examining use cases that are driving the motivations of this work. Next, we set product requirements to a Data Citation system where we already get a first indication of what a Data Citation Framework must cover. These requirements are accompanied by constraints in order to exclude potentially bad design choices and to ensure the coverage of the RDA Data Citation recommendations. We continue with the definition of three quality goals, which can be seen as non-functional requirements, that narrow the scope for design choices later to come. After having discussed the prerequisites, we illustrate our Data Citation Framework as a component diagram and afterwards explicate its components in detail. Last, we revisit our Data Citation Framework components and assess their compliance with the RDA Data Citation Recommendations.

In the following we use the terms *Data Citation Framework* and *Data Citation System* where latter should be understood as an implementation of the former which uses the framework's components and fulfills its requirements. By implementation we do not think of a single piece of software or API but the overall system including stakeholders that is needed for the functioning of Data Citation for RDF stores.

3.1 Requirements and Constraints

As the main motivation of this work goes, we want to foster reproducible results and therefore grant the researchers access to data that other researchers used in their experiments at a certain point in time. One approach would be to provide a link to datasets which lie in (public) repositories where researchers have uploaded them. This comes with some caveats, such as broken links after the repository moves to another domain or evolved datasets. Based on the *recommendations* we created a use case diagram (see Figure 3.1), that depicts typical use cases and involved actors. The diagram shows that publishing a dataset is not necessary because

the dataset should already be (publicly) available and retrievable by means of queries. This also involves data curators or operators to offer an access point to data that researches might want to use. In terms of RDF* stores a SPARQL query must be used for the purpose of retrieval. Researchers must then be able to retrieve these datasets using information provided in the citation snippet. This implies that the system must be able to query historical data to extract the dataset as it existed at the time of citation snippet creation. In addition to researchers, (RDF) Data operators are involved who are required to update their triple stores, e.g. when new data is available or errors in the data need to be corrected. This might also require them to query live and historical data, compare different dataset versions and make the right change decisions.

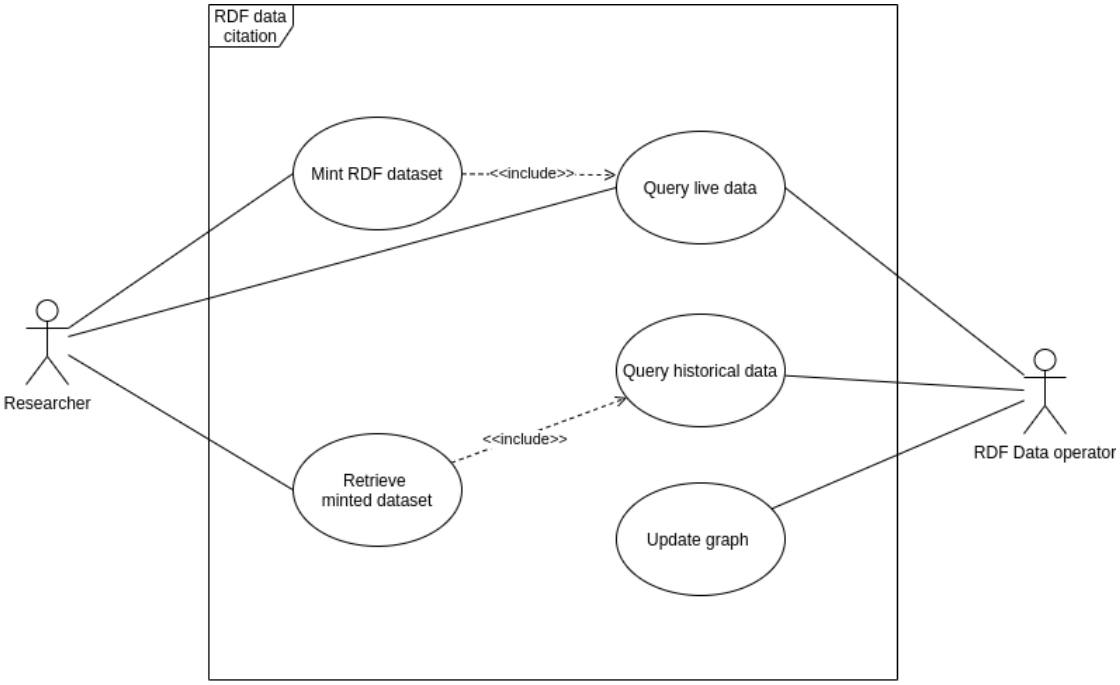


Figure 3.1: Data Citation Framework Use Case Diagram

3.1.1 Product requirements

Now that we covered the depicted use cases we should have a high-level view of the requirements. This brings us to the next step of breaking down the use cases into smaller pieces. In the following we link the Use Case diagram in Figure 3.1 to the RDA Data Citation Recommendations and discuss which of these recommendations can be turned into product requirements to our RDF Data Citation Framework (=product). Furthermore, we argue why they are required in order to meet our use case goals. On top, we add additional requirements, not covered by the RDA Data Citation Recommendations, quality goals (non-functional requirements) where some of them arise due to the nature of RDF* stores.

In Table 3.1[vUSP16] we see that all of the first three recommendations are required. Arguably, they are the foundation for enabling data citation by allowing for querying a specific dataset version (live or historical) with timestamp-based queries.
In Tables 3.2 and 3.3[vUSP16] the recommendations mainly revolve around the use case *Retrieve minted dataset*. Only one of them, namely R4 - Query Uniqueness, is not required to satisfy this use case.
Table 3.4 again focuses on retrieving PID-minted datasets. We argue that retrieving PID-minted datasets in one or another way is a minimum requirement to satisfy use case *Retrieve PID-minted dataset*.
The requirements in Table 3.5 are not the focus of our framework design as we build on specific data infrastructure (RDF* and SPARQL*). However, we needed to discuss what modifications to the data structure could possibly mean with regards to RDF and SPARQL. We see that migration within a certain frame of technologies (e.g. data representations) does not always require actions recommended in R13 and R14. It goes even so far, that SPARQL query migration might not even be wanted upon an ontology update (new schema) as it would yield different datasets, which is against the idea of reproducible results.

Table 3.1: Preparing the Data and the Query Store[vUSP16]

RDA Recommendation	Degree of requirement	Argument
R1 – Data Versioning: Apply versioning to ensure earlier states of data sets can be retrieved.	Required	Use Cases *Query live data* and *Query historical data* would not be met without versioning. In addition to this recommendation/requirement we ask for statement-level versioning. Thus, every triple in the RDF store must be versioned.
R2 – Timestamping: Ensure that operations on data are timestamped, i.e. any additions, deletions are marked with a timestamp.	Required	Closely coupled with versioning (R1): Each versioning activity must be documented with a timestamp.
R3 – Query Store Facilities : Provide means for storing queries and the associated metadata in order to re-execute them in the future.	Required	Without a query store, PID-minted datasets could not be retrieved as the query and execution timestamp would be missing. These are necessary information to generate a timestamped query.

Next to the RDA Data Citation Recommendations, where most of them are required for our Framework, we have following additional functional and product requirements.

Table 3.2: Persistently Identify Specific Data Sets[vUSP16]

RDA Recommendation	Degree of requirement	Argument
R 4 – Query Uniqueness : Re-write the query to a normalised form so that identical queries can be detected. Compute a checksum of the normalized query to efficiently detect identical queries.	Recommended	This recommendation does not match any of our use cases. Its absence does, however, imply redundancy within the query store, thus consuming more memory. On top, researchers might gain less credit as their contribution to the dataset does get less likely discovered. The user, who executes the new query, does not get notified about the existence of an identical query and its contributer/publisher.
R5 – Stable Sorting: Ensure that the sorting of the records in the data set is unambiguous and reproducible	Required	The use case "Retrieve minted dataset" would not be met if a query produces ambiguous sorts and hence yields a differing dataset from the minted one. This would also violate the reproducibility principle.
R6 – Result Set Verification: Compute fixity information (checksum) of the query result set to enable verification of the correctness of a result upon re-execution.	Required	To check whether the retrieved minted dataset is the same as when it was minted.
R7 – Query Timestamping: Assign a timestamp to the query based on the last update to the entire database (or the last update to the selection of data affected by the query or the query execution time). This allows retrieving the data as it existed at the time a user issued a query.	Required	This is trivially required in order to *Retrieve minted datasets*, *Query live data* and *Query historical data*. The easiest and most efficient way is to use the execution timestamp as the other two options would require additional queries to find the last update.

Verify unique sort index

A primary key is needed for sorting the result set. This constraint, however, does not exist in RDF stores. Therefore, the user needs to provide a primary key or unique sort index. The Data Citation System then needs to verify its uniqueness. This can be done either on the client side, e.g. when issuing the query to mint a new PID or centrally checked by a central framework component.

Table 3.3: Persistently Identify Specific Data Sets[vUSP16]

RDA Recommendation	Degree of requirement	Argument
R8 – Query PID: Assign a new PID to the query if either the query is new or if the result set returned from an earlier identical query is different due to changes in the data. Otherwise, return the existing PID.	Required	Assigning query PIDs is crucial for persistently identifying and retrieving specific datasets.
R9 – Store Query: Store query and metadata (e.g. PID, original and normalized query, query and result set checksum, timestamp, superset PID, data set description, and other) in the query store.	Required	Without metadata, such as query, query PID and timestamp, minted datasets could not be identified.
R10 – Automated Citation Texts: Generate citation texts in the format prevalent in the designated community for lowering the barrier for citing the data. Include the PID into the citation text snippet.	Required	To fulfill the goals of the use case "Mint RDF Dataset" we do need a citation text snippet. Moreover, to retrieve the dataset we need its corresponding query PID.

RDA Recommendation	Degree of requirement	Argument
R11 – Landing Page: Make the PIDs resolve to a human readable landing page that provides the data (via query re-execution) and metadata, including a link to the superset (PID of the data source) and citation text snippet.	Required	Minted datasets must be somewhere accessible in order to retrieve them. However, none of the use cases would require a human readable landing page, as long as the dataset can be accessed or downloaded.
R12 – Machine Actionability: Provide an API / machine actionable landing page to access metadata and data via query re-execution	Required	This is trivially required in order to *retrieve minted datasets.*

Table 3.4: Resolving PIDs and Retrieving the Data[vUSP16]

Query live and historical data

There must be a way for the user (Data Operator, Researcher, Publisher) to query data by passing a timestamp in addition to the query. A query processor must then return the version of the data that corresponds to the timestamp. If no timestamp is provided, the processor

RDA Recommendation	Degree of requirement	Argument
R13 – Technology Migration: When data is migrated to a new representation (e.g. new database system, a new schema or a completely different technology), migrate also the queries and associated fixity information.	Partially Required	A migration would be needed if another RDF query language, such as DQL or XQuery, or a completely different technology, such as NoSQL or SQL, is used. When migrating from one RDF representation to another, e.g. from n3 to json, there is no need to migrate the SPARQL query as it is independent of the underlying representation. A query migration is also not need across different Triple Store Vendors. This holds true as long as RDF is used for data representation and SPARQL to query data. Another case when queries do not need to be migrated is a schema migration. As data and metadata (semantics) are stored within the same physical storage a schema migration simply means updating the metadata triples and timestamping them. As a consequence, the old schema is still kept but marked as outdated. Thus, the historical queries still returns the minted datasets.
R14 – Migration Verification: Verify successful data and query migration, ensuring that queries can be re-executed correctly.	Required	SPARQL queries will always be re-executable as long as the RDF store supports the same SPARQL version as before or is backward compatible. Result sets of old and new system need to be compared and assured to be identical.

Table 3.5: Upon Modifications to the Data Infrastructure [vUSP16]

should extract live data (most recent version) from the RDF store.

Insert triples, Update triples, Outdate triples

RDF Data Operators must be able to perform write operations against triple stores which links to the use case *Update Graph*. In compliance with *R1* and *R2* recommendations every

write operation must come with additional metadata triples (Statement-level versioning) to ensure that the operations are versioned. The *Insert Triples* operation simply inserts one or more triples into the RDF* store, either serially or as bulk insert. The triple, of course, must be compliant with the underlying RDF representation (e.g. n3 or turtle). The *Update Triples* operation must update the object of the triple. We chose the term *Outdate triples* to make it clear that no actual deletions may occur under normal circumstances. Unusual circumstances would be if the RDF data operator accidentally inserted an unusually great number of corrupt records which noticeably compromise read and write operations which also occur when a user cites a dataset. Another reason would be deletions that are required by law, which need to be documented and traceable as source for irreproducibility.

Flexible metadata interface

The use case *Mint RDF dataset* requires the *Publisher* to provide metadata such as provenance information. There are a couple of Metadata schemata to consider as a guideline, like DataCite's Metadata Schema [1] or the ones proposed in [FCG+19], [BCDC+15]. A minimum set of metadata should be recommended to the user but over and above that the user should not be restricted in providing further metadata.

Mint RDF dataset

An interface must be in place which lets researchers mint datasets and thereby receive a citation snippet including a resolvable query PID that points to a landing page. The dataset must have been extracted in a previous step using a SPARQL query. The query must be linkable to the dataset and vice versa. When researchers mint datasets they must also provide metadata, as described in subsection 3.1.1 - Flexible metadata interface.

Describe dataset

A data set description must be available for every query PID. It is either required that a central component of the system derives a description using query text and dataset or the user simply provides a description on his own. Both methods can also be used in conjunction.

Handle SPARQL query prefixes

A SPARQL query most often comes with a prologue at the beginning of the query, which includes IRI prefixes that are used in the query. These prefixes must be resolved when normalizing the query. Furthermore, there might be the need to extended the prologue with custom prefixes or split prologue and query for easier query parsing and enrichment, e.g. when building a timestamped query.

[1] https://schema.datacite.org/meta/kernel-4.3/doc/DataCite-MetadataKernel_v4.3.pdf

3.1.2 Constraints

In this section we want to explicitly state constraints to the design of the *RDF Data Citation Framework*. We already mentioned some in the product requirements, such as that actual deletion of records is not allowed.

Dataset source

The source of the dataset must be an RDF store. A user may only mint datasets that he/she extracted from the triple store that is used within the Data Citation System using an RDF query (e.g. SPARQL). This is to avoid system designers to create environments where RDF datasets, which have been extracted from other sources, can be uploaded/published and cited where a query is either not available or it cannot be verified that the query returns the published result set because the user could not issue the query him/herself.

SPARQL as query language

The RDF query language must be SPARQL. The first reason for this restriction is that SPARQL is a W3C standard [2] and seemingly the most used language to query RDF stores. Based on a search using the keywords of eight known RDF query languages we found that SPARQL has the most entries in Google scholar (see Table 3.6) The second and more important reason is that we are focusing our research on RDF* in combination with SPARQL*. Though, we might open up this restriction in future and aim for a more generic RDF Data Citation design.

RDF query language	Number of results
SPARQL	67.300
XQUery	30.600
SquishQL	472
RDQL	4.140
TriQL	522
RQL	24.000
SeRQL	1.820
eRQL	450

Table 3.6: Search results from Google Scholar for individual RDF query languages on 16.06.2021 13:10 CEST

Query Handler algorithm

There must be an algorithm which implements R4-R10 of the RDA Data Citation Recommendations in a workflow manner and handles following distinct cases:

[2] https://www.w3.org/TR/sparql11-query/

1. The dataset's query already exists in the Query Store and the result set has not changed since the last execution time.

2. The dataset's query is new and a query PID has just been minted for it.

3. The dataset's query already exists in the Query Store but the result set has changed since the last execution time.

Updating only objects

Alternatively to updating the object, one could want to update the predicate or subject instead. To show what this could possible mean, we first transcend into the domain of relational databases and tables and construct an example, similar to the one in [PB08] which illustrates a mapping between a SQL database table and RDF triples. The Table in 3.7 shows a database table on the left and its RDF representation on the right. The transformation starts with defining two prefix URIs, one to be used for subjects and one for predicates. Either prefix URI contains the table name. The one for the subject is used exclusively for the primary key (assuming there are no composite keys). We see how id 123 becomes celebN:123 in the RDF representation and this way preserving the link between table and primary key. Next to the primary key, we have two other columns *lastNAme* and *occupation*. These are represented as predicates in RDF using the second prefix and thereby also preserving the association between the same table and its non-key attributes. What is left from the table on the left are the non-key attribute values and these are simply objects linked to the right subject/IDs and predicates/attributes in the RDF representation.
We can therefore conclude that updating the subject would either be a table name change, if we update the prefix URI or primary key value change if we update the part of the URI which comes after the prefix. In this example we could e.g update the prefix URI of celebN, which is <http://example.com/DB/Celebrity/>, to <http://example.com/DB/People/>. This would mean that the table is no longer called *Celebrity* but *People*. Both, table name and ID changes is something that rarely occurs. In the RDF domain former could mean a change of context or resource.
As we saw that predicates are used to represent table name + attribute we can say that a predicate update in RDF refers to Data Definition Language (DDL) in the context of relational tables. In RDF this might be wanted if one wants to switch to another ontology that uses different terms.
While both component updates might be desired we restrict ourselves to updates on objects only as this is in our opinion the most common use case and changing the context or ontologies needs further research which e.g. includes assurance that all triples with a common prefix URI are updated and not just a subset.

3.1.3 Non-functional requirements

In this section we set three quality goals (this term is taken from the arc42 documentation) or non-functional requirements to the *RDF Data Citation System*.

id	lastName	occupation
123	Obama	author
456	Alonso	formula-1-driver

```
@prefix celebN <http://example.com/DB/Celebrity/>
@prefix celebP <http://example.com/DB/Celebrity#/>
celebN:123    celebP:lastName    "Obama"
celebN:123    celebP:occupation  "president"
celebN:456    celebP:lastName    "Alonso"
celebN:456    celebP:occupation  "formula-1-driver"
```

Table 3.7: Database table *Celebrity* (left) and its RDF representation (right)

Compatibility - Replaceability

Alternative algorithms such as checksum computation, normalization approaches, minting new query PIDs or citation snippet generation must be implementable without limiting or hazarding the solutions functionalities. System designers may exchange whole system components, such as RDF store technologies, landing pages and data management tools as long as the requirements for the components themselves are met.

Efficiency in terms of query performance and triple store memory usage

Adding metadata with respect to versioning to the triple store when performing write operations must be as efficient as possible in terms of storage. This can be measured in the number of additionally added triples. As saving store might be inversely related with query performance we want to find a trade-off in terms of efficiency. Whether to aim for better query performance or less storage consumption is dependent on external factors like storage capacity and number of citations per hour and should be decided by *Data Citation System* designers.

Operability

The designed *RDF Data Citation System* should be easy to use for all involved stakeholders. Automation helps improving operability and should be applied wherever possible and thereby minimize the required user input. Possible applications are automatically collecting the user's provenance information and other metadata when minting a dataset, deriving a unique sort index and sparing the user from providing a primary key or deriving a dataset description from the dataset and query text.

3.2 RDF* Data Citation Framework

In this section we illustrate and elaborate our RDF* Data Citation Framework. The categorical building blocks we used to compose our framework do resemble a classical software architecture stack, that is, a persistence layer, a business logic and a (graphical) user interface. The modular composition allows system engineers and developers to focus on each module separately and enhance them or replace them without corrupting other modules/components. This framework should fulfill the product requirements from the previous section and thus be

compliant with the *recommendations* and also promote the achievement of non-functional requirements, which, however, in the very end depend on the implementation of the data citation system. The overall idea is to have user interfaces for our stakeholders (researchers and data operators) where they can publish and access data & metadata or operate on them, a central system or API comprised of several components that enable persisting and retrieving specific datasets by several means (versioning, checksums, normalization, ...) and foster machine-actionability by the use of prevalent formats. In the persistence layer we need two stores - RDF* store for data and a query store for metadata. The RDF* store as data store is not an arbitrary choice but is, in fact, the main contributor to versioning RDF data. In Figure 3.2 we give a high-level view of our framework and the components, which we breakdown in the subsequent sections.

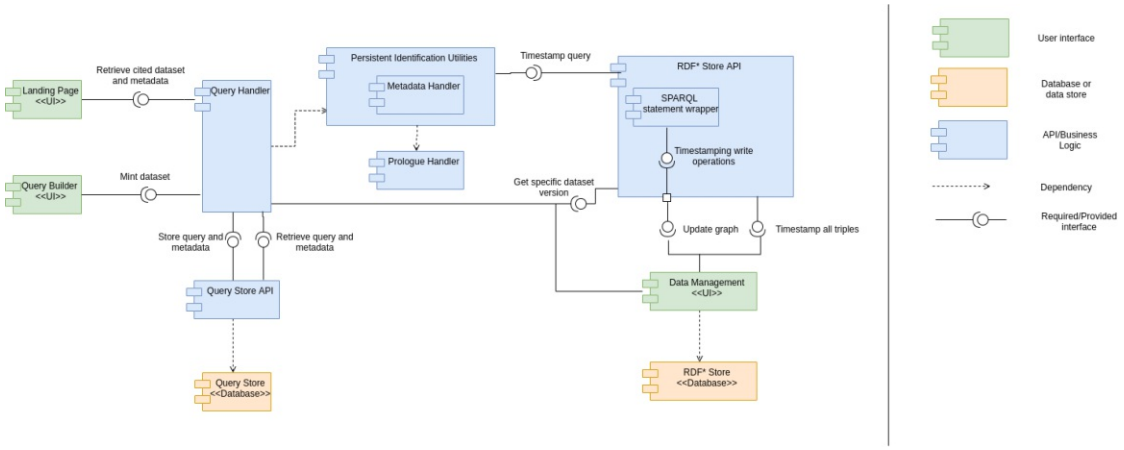


Figure 3.2: RDF Data Citation Framework and its components

3.2.1 RDF* Store Utilities

The RDF* Store Utilities define basic read and write operations to the triple store and a possible way to implement them to be compliant with R1 and R2 RDA Data Citation Recommendations. Every write operation must come with additionally added nested metadata triples where a versioning timestamp (object) is linked to the data triple (subject). The predicate, which does the linking between subject and object, can be one of following two conceptual properties: *valid_from* which defines the start date from which on the new triple is valid and *valid_until* which sets an expiration date to the triple. Both terms should preferably be IRIs from the semantic web. However, further research is needed here and an IRI might be proposed in future (see 6.2). This kind of statement-level annotation is possible with the use of RDF* to represent nested triples and SPARQL* to query them, which we put into service in our design. The example in Table 3.8 shows an RDF* triple set after insert, update and outdate have been consecutively performed.

Given a timestamp, the read operation (query) uses both aforementioned properties to filter for triples that lie in between two dates. This implies that all triples must have both versioning

Table 3.8: Example of a dynamic RDF* dataset that changes over time and is versioned using RDF*'s nested triples

RDF* set	Version
:Obama :occupation "president". «:Obama :occupation "president"» :valid_from "2009-01-20 12:00" «:Obama :occupation "president"» :valid_until "2017-01-20 12:00"	Initial Version
:Obama :occupation "president". «:Obama :occupation "president"» :valid_from "2009-01-20 12:00" «:Obama :occupation "president"» :valid_until "2017-01-20 12:00" :Obama :occupation "author". «:Obama :occupation "author"» vers:valid_from "2018-01-01 14:53" «:Obama :occupation "author"» vers:valid_until "9999-12-31 12:00"	Version 1 (after insert)
:Obama :occupation "president". «:Obama :occupation "president"» :valid_from "2009-01-20 12:00" «:Obama :occupation "president"» :valid_until "2017-01-20 12:00" :Obama :occupation "author". «:Obama :occupation "author"» vers:valid_from "2018-01-01 14:53" # «:Obama :occupation "author"» vers:valid_until "9999-12-31 12:00" «:Obama :occupation "author"» vers:valid_until "2019-05-06 17:53" :Obama :occupation "film_producer". «:Obama :occupation "film_producer"» vers:valid_from "2019-05-06 17:53" «:Obama :occupation "film_producer"» vers:valid_until "9999-12-31 12:00"	Version 2 (after update)
:Obama :occupation "president". «:Obama :occupation "president"» :valid_from "2009-01-20 12:00" «:Obama :occupation "president"» :valid_until "2017-01-20 12:00" :Obama :occupation "author". «:Obama :occupation "author"» vers:valid_from "2018-01-01 14:53" # «:Obama :occupation "author"» vers:valid_until "9999-12-31 12:00" «:Obama :occupation "author"» vers:valid_until "2019-05-06 17:53" :Obama :occupation "film_producer". «:Obama :occupation "film_producer"» vers:valid_from "2019-05-06 17:53" # «:Obama :occupation "film_producer"» vers:valid_until "9999-12-31 12:00" «:Obama :occupation "film_producer"» vers:valid_until "2021-05-06 11:12"	Version 3 (after outdate)

properties attached to them in the triple store. However, if one starts to version a non-empty triple store, only the newly added triples are annotated with these properties. This is why the RDF* Store Utilities employs one more function, namely *version_all_rows()*, which has the purpose to also version existing triples, before inserting any new ones. We propose two ways to do this. The first one is to attach a start and an end date to all existing triples, meaning that two additional metadata triples per data triple are added. For the start date the current system timestamp could be used and the end date should be an end date that lies far in the

future. The second approach is to only set the end date and leave out the start date, thus, one additional metadata triple per data triple. Figure 3.3 illustrates the interface we just discussed.

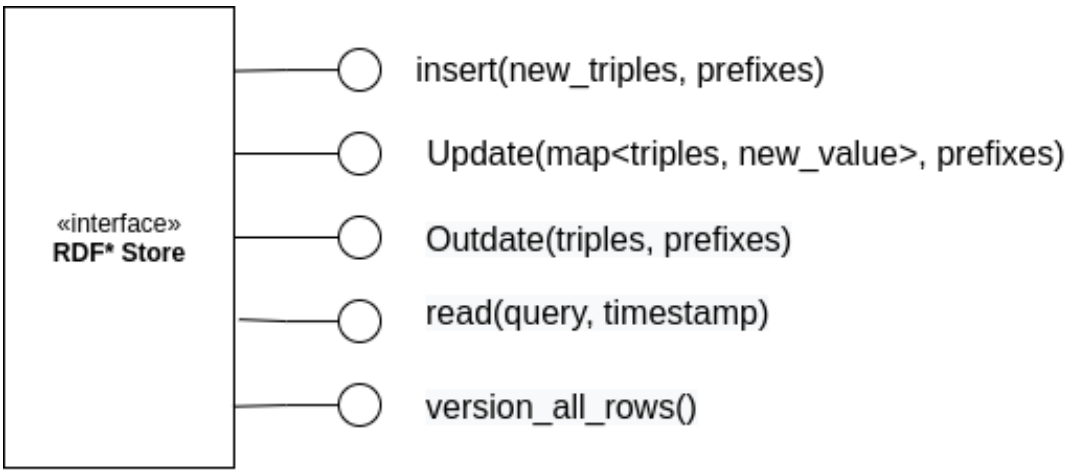


Figure 3.3: RDF* Store Utilities to enable communication between triple store and user.

3.2.2 Persistent Identification Utilities

The *Persistent Identification Utilities* component bundles all functions that are discussed in Section "Persistently Identify Specific Data Sets" in [vUSP16] except for R9 - store query. Moreover, we added two additional functions, namely *describe* and *create_sort_index* to foster operability and minimize the needed input from users. *Describe* takes both - dataset and query - as arguments. While it might be intuitive to take the dataset for knowledge extraction the query itself can be seen as an description of the dataset already. E.g. if we use a filer $?name = "Obama"$ one could suggest that this dataset "is about" the former president of the U.S.A. *Create_sort_index* only takes the dataset as an argument and ideally returns one unique sort index. However, there are cases where more than one unique sort indexes are possible and it is not easily decidable which one to take. An example can be found in Table 3.9. We see that possible unique sort indexes, which require a minimum number of columns, are ('column2', 'column4') and ('column3', 'column4'). In such cases, any of these offered indexes can be chosen by the algorithm to minimize user input. Last we have a flexible Metadata constructor that offers optional arguments from the mandatory fields of DataCite's Metadata Schema [3] plus a dataset description and further arguments to be used for additional Metadata. These are all utilities that can and should be employed in the query builder algorithm and they are summarized in Figure 3.4.

[3] https://schema.datacite.org/meta/kernel-4.3/doc/DataCite-MetadataKernel_v4.3.pdf

Table 3.9: Theoretical example of a dataset where multiple unique sort indexes are possible.

column1	column2	column3	column4
1	1	1	1
1	1	1	2
1	2	1	3
1	1	2	3
1	1	1	5

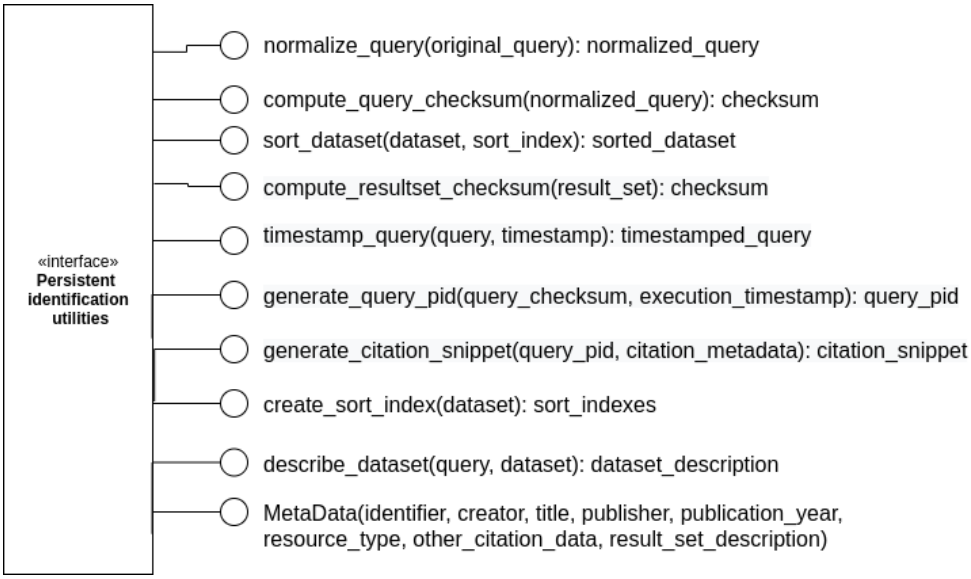


Figure 3.4: RDF Data Persistent Identification Utilities to be used in the query builder algorithm.

3.2.3 Query Store Utilities

The *Query Store Utilities*, similar to *RDF* Store Utilities* provide functions for read and write operations that can be issued against the Query Store. However, the difference is that the read and write statements are predefined with certain parameters being the only variables within the statements. Functions and parameters are shown in Figure 3.5. *get_query* retrieves a query and its metadata by the query PID. Subsequently, this information can be presented on a landing page, as recommended in R12 and the query can be used for re-execution. *get_last_execution* retrieves a query and its metadata by the query checksum. The purpose of this function is to compare a new query and the returned live dataset, that a researcher is about to mint, with the latest version of it (the last execution), as one query_checksum can have multiple PIDs. *store* stores query and metadata. If the query is new, all the associated metadata (query-specific & dataset-specific metadata, provenance metadata, ...) is stored. If the query is found in the query store but the result set changed, only the result set relevant metadata is

stored and attached to a new query PID. The schema in Figure 3.6 depicts a normalized query store schema where information that change more frequently, which are result set related information, citation metadata and the timestamped query, are found in a separate table *query_satellite*. Query related metadata are less prone to changes, which we collect in the *query_hub* table.

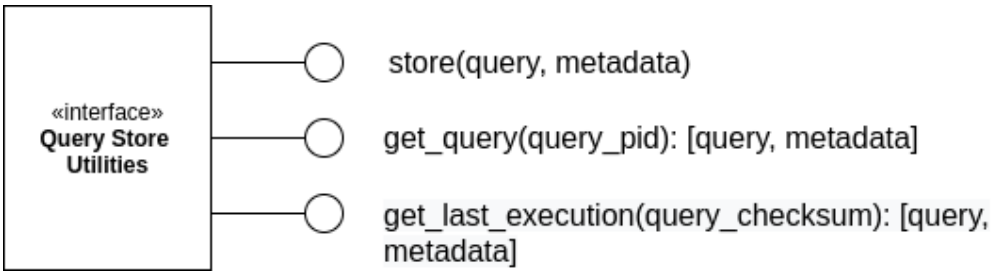


Figure 3.5: Query Store Utilities

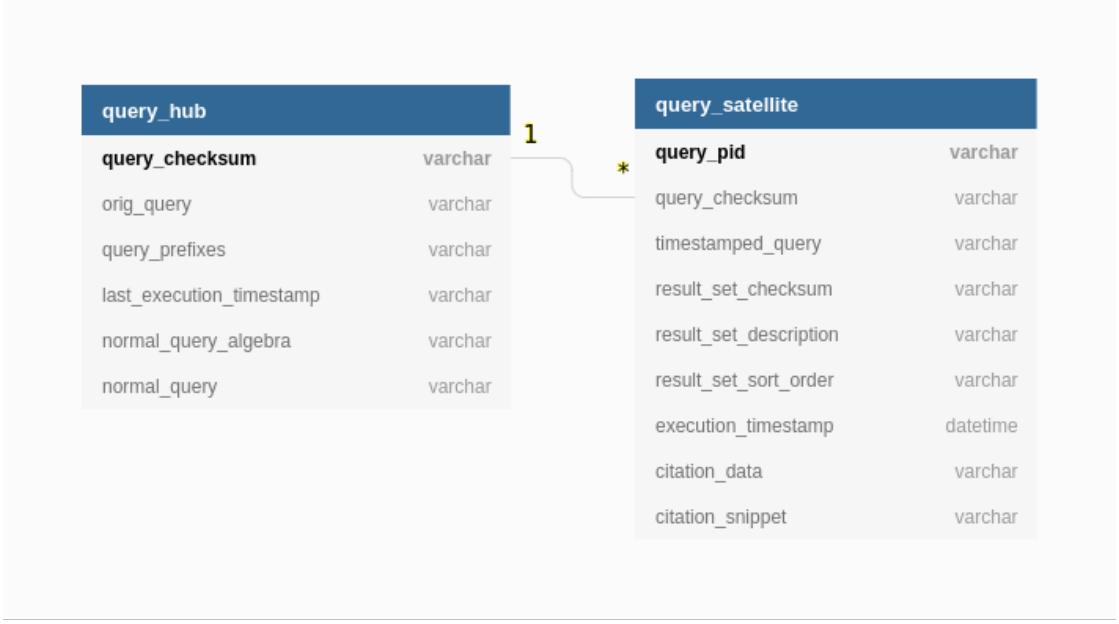


Figure 3.6: Normalized Query Store Schema

3.2.4 Prologue Handler

The *Prologue Handler* is an optional component which provides useful functions to accommodate for the usage of namespaces (base and prefix declarations) in SPARQL. This component is not designed for end users, thus, actors we saw in Figure 3.1 but rather for system engineers

and developers which might want to use this framework to implement an *RDF Data Citation System*. Prologues in SPARQL queries are optional and each query which uses them can be rewritten into a "prologue-free" form. This comes in handy when we want to normalize queries and thereby resolve prefixes. Resolving prefixes can be seen as a normalization step. This is because prefixes are just arbitrary labels which can be different between two queries but yield the same IRIs when resolved. Another use case developers might encounter is to split prologue and query body in order to operate on them separately. Prefixes could then be extended with additional versioning-related prefixes and re-attached to the query. Also note that we store query prefixes in the Query Store (see 3.6). If migrating to a new ontology it could suffice to just modify the prefix IRIs, which is also why we have them separated from the query body. Alternatively, custom query parsers or regular expressions could be used to inject code either in the prologue or query body section. We cast the discussion above into an interface displayed in Figure 3.7 and provide an example in Figure 3.8 of how these functions could be used to add prefixes that might be used for Versioning and Timestamping. Briefly, the example we show first splits a query (upper left) into two parts - Prologue and Query Body (upper rights) whereas the query is stored as string and the prefixes and corresponding IRISs in a dictionary. Then we extend the dictionary by two prefixes vers and xsd (lower left) and attach them to the original query (lower right). What would be left as a further step is to transform the query body into a timestamped query body where the newly defined prefixes would actually be used. This, however, we leave for the example in Section 4.1.3 where we treat timestamped queries in more detail.

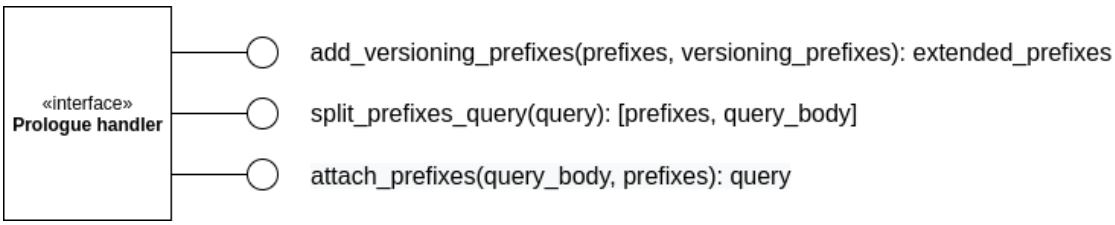


Figure 3.7: Prologue handler interface

3.2.5 Query Handler

The *Query Handler* can be considered as the core component of our framework which directly relates to the use cases *Mint RDF Dataset* and *Retrieve Minted Dataset*. It makes use of *Query Store Utilities*, and *Persistent Identification Utilities* and *RDF* Store Utilities* to return citation snippets, metadata and minted datasets via re-execution. As the *recommendations* go, citing datasets means persistently identifying them. That is why *recommendations* R4-R10 have to be part of the query handler algorithm. To include them, we can use the component *Query Store Utilities* for R9 and *Persistent Identification Utilities* for remaining ones. Dataset utilities like *Sort Dataset*, *Compute Result Set Checksum*, *Create Sort Index* and *Describe Dataset* need the dataset as input, which we get by executing *read(query, timestamp)* from the *RDF* Store Utilities*. The current system timestamp can be passed to the timestamp parameter which, in fact, means querying live data. Theoretically, there could be updates made to the triple

```
query = "
PREFIX ex: <http://example.com/>
select ?s ?p ?o
where {
    ?s ?p ?o .
    filter(?s = ex:Obama)
}"
```

```
prefixes = {ex: "<http://example.com/>"}

query_body = "
select ?s ?p ?o
where {
    ?s ?p ?o .
    filter(?s = ex:Obama)
}"
```

```
prefixes =
{ex: "<http://example.com/>",
vers: "<http://example.com/versioning/>",
xsd: "<http://www.w3.org/2001/XMLSchema#>"}

query_body = "
select ?s ?p ?o
where {
    ?s ?p ?o .
    filter(?s = ex:Obama)
}"
```

```
query = "
PREFIX vers: <http://example.com/versioning/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ex: <http://example.com/>

select ?s ?p ?o
where {
    ?s ?p ?o .
    filter(?s = ex:Obama)
}"
```

Figure 3.8: Prologue handler example usage to add Versioning and Timestamping prefixes

store between the point the system timestamp was returned and the query execution. This is, however, not a problem as the updates receive a later timestamp. The utilities we discussed so far can be fit into a procedure. But when it comes to storing the query and its metadata into the query store (R9) a decision logic needs to be set up which matches the requirements stated in Section 3.1.2. Table 3.10 describes cases and actions for R9 - store query. With *generate_query_pid* from *Persistent Identification Utilities* we only compute a technical query PID which needs to be further processed and turned into an URL that resolves to a landing page. To do this we included the parameter *create_identifier(query_pid)* whereas we did not set any constraints on how to create the URL. This URL should be assigned to DataCite's *identifier* (see 4.1.3).

Table 3.10: Cases as describe in Section 3.1.2

Case	Tables with a new entry
The dataset's query already exists in the Query Store and the result set has not changed since the last time.	{}
The dataset's query is new and a PID has just been minted.	{query_hub, query_satellite}
The dataset's query already exists in the Query Store but the result set has changed since the last time.	{query_satellite}

Once a query PID has been minted for the dataset it can be retrieved using the very same

query PID. To achieve this we can use two components we already outlined, namely *Query Store Utilities* for retrieving metadata including the query and *RDF* Store Utilities* for retrieving the dataset via re-execution. The functions should return both - dataset and metadata - to the caller in a machine-actionable format, such as json. As we see in Figure 3.9 the query handler component is rather short in terms of offered functions but vital for the RDF Data Citation Framework.

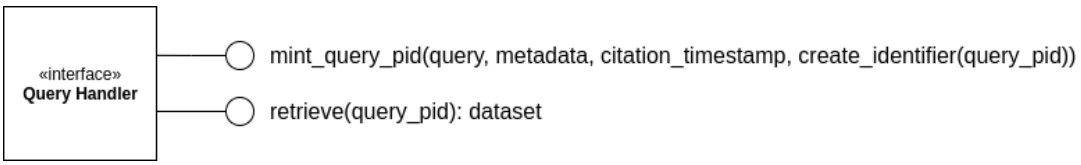


Figure 3.9: Query Handler interface

3.2.6 Query Builder UI

In our design the term *Query Builder UI* refers to a user interface which researchers use to create datasets and mint a resolvable query PID for it. In this section we define obligatory and optional features this component must include. The first and most important feature is the query editor which allows for writing queries and executing them. As described in Section 3.1.2 - Dataset source, we must constrain our dataset to come from an accessible and queryable RDF store. This query editor does not necessarily need to be a text editor for SPARQL queries but can also be implemented as a graphical query builder. In this case, the graphical query must be translated into a SPARQL query nonetheless. In conjunction with a query editor we need either a result set viewer or the possibility to download the result set so that the user can verify the result sets correctness. Only then a user can issue a command to mint a query PID for the dataset which triggers *mint_query_pid* from the *Citation* component and returns a citation snippet. As seen in Section 3.2.5 the *mint_query_pid* function also requires metadata as input and therefore the *Query Builder UI* needs a metadata interface. Here the system engineers have the option to either let the user populate required metadata field or automatically derive them (desired). Latter can be tackled by first grouping metadata into a way that attributes that share common or similar sources are assigned to the same group and then treating each group separately. Table 3.11 shows our choice of metadata categories & mapped metadata attributes from the MetaData constructor of the *Persistent Identification Utilities* component and possible sources from where these attributes can be derived. The last mandatory feature is a viewer for the returned citation snippet.

A well-engineered system should inform users about their actions and wrong inputs. That is why we highly recommend to return following additional information to the user:

- Did I provide a unique sort index in the "order by"-clause of the query?
- Did I, by accident, use multiple order-by clauses? (In SPARQL this is allowed compared to most known SQL languages)

Table 3.11: Metadata attributes mapped to categories and sources they could be automatically collected from

Metadata category	Attribute	Metadata source
User-specific	creator	Not considered in our design. We cannot know who initially wrote the SPARQL query.
	publisher	From the publisher's account
Dataset-specific	title	Inferred from the dataset and query text
	resource type	"Dataset" is a resourceTypeGeneral in DataCite's Metadata schema. The subtype could be inferred from the dataset and query text.
	identifier	Returned by a function that creates a persistent landing page URL from the query PID.
	result set description	Could be inferred from the dataset and query text.
Citation-specific	Publication year	Execution timestamp.

- Did I provide an "order-by"-clause at all?
- Is this a new query PID?
- If the query is not new: Did the dataset change since the last execution?

3.2.7 Landing Page

The landing page is a component which we can directly link to the use case *retrieve minted dataset* and the R11 recommendation. We already expressed our view in Table 3.4 that a landing page is recommended but not necessary if we look at the use case *Retrieve minted dataset* in a narrow way, thus, if we assume that researchers just want to access data and metadata in any possible way by following a PID from a citation snippet. This PID could therefore resolve to a repository where the dataset and metadata have been uploaded or trigger a download. However, there are a few issues with such alternative approaches. Former approach generates a high workload as possibly terabytes of data must be uploaded. It also increases redundancy as the dataset would be available twice - once in the Triple Store and once in a new directory. Latter one again forces the user to wait until the download is finished, which can take unnecessarily long if the dataset is big enough. This is why we also stick to the data retrieval via query re-execution and presentation on the landing page as this seems to be the most efficient way to access data and metadata.
This information can be retrieved using the *retrieve* function from the *Citation* component (see Section 3.2.5). In our framework we do not set any restrictions on the landing page visual

design. We, though, suggest it to be a web page that can be accessed through a resolvable query PID (e.g. DOI) via any browser.
The Landing page component is also the right place to include access control. Data citation is not restricted to research data but this idea can also be implemented by corporations with sensitive data where datasets need to be persistently identifiable, e.g. to reproduce certain reports, but should only be accessible by certain user groups. DataCite's metadata attributes publisher or creator can be used for this purpose and mapped to these user groups. The component diagram in Figure 3.10

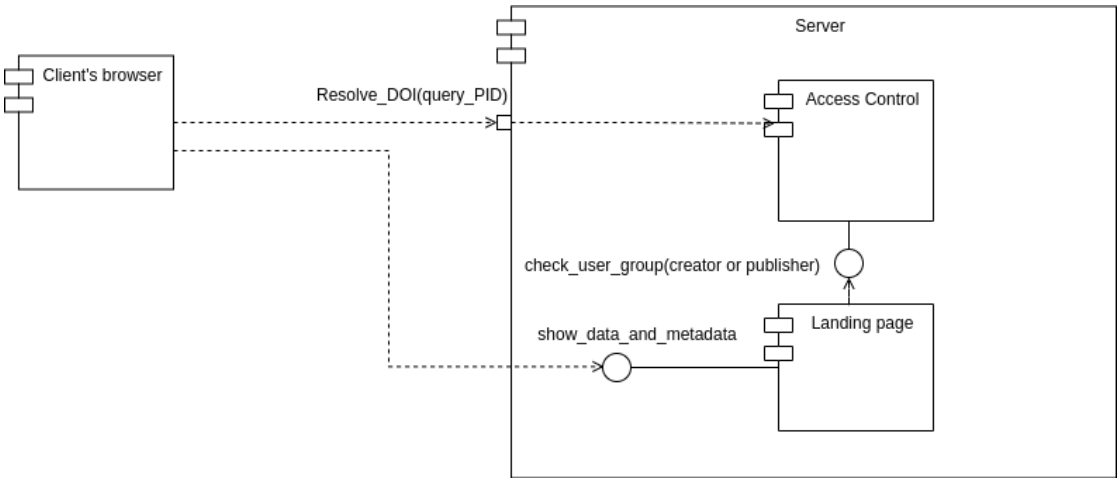


Figure 3.10: Landing page and access control

3.2.8 Data Management

Data management is a component that comes on top of *RDF* Store Utilities* and can be seen as a UI for data operators. It can be implemented either as a GUI or CLI. System engineers are free to add more functionality to the *Data Management* interface than what is provided in *RDF* Store Utilities*. This extra functionality can encompass job schedulers, batch and stream processing or schema and ontology validations. Data operators use this component to curate the RDF* store whereas versioning and timestamping operations are abstracted from them as they are handled within the *RDF* Store Utilities* component.

3.3 Compliance with RDA Data Citation Recommendations

In Section 3.1.1 we expressed *recommendations* as product requirements to the RDF* Data Citation Framework. In Table 3.12 we enumerated all recommendations in the first column and explain in the second column how they were considered in the framework. We employed RDF* as data representation model together with SPARQL* as query language in *RDF* Store Utilities* as a means to R1 and R2. To store and retrieve query & metadata (R3, R9) we designed the *Query Store Utilities* component. To cover recommendations R4-R8 we packed various

functions into *Persistent Identification Utilities*. These functions are then used by the *Citation* component to persistently identify a specific dataset and by the end generate a new or return an existing Citation Snippet (R10). Moreover, this component offers users a function to retrieve data & metadata including query from the RDF* store and query store respectively in a machine-actionable way (R12). We fulfill R11 - Landing Page using a component of the same title. R13 & R14 are not covered by our framework.

3.4 Summary

In this chapter we designed an RDF* Data Citation Framework that is compliant with R1-R12 of the *recommendations*. We turned these *recommendations* into product requirements and designed functions and features to meet them. Furthermore, we outlined additional product and non-functional requirements (quality goals) to aim for a comprehensive system where not only researchers but also data operators are involved. We broke down necessary components from the high-level view and used depictions of interfaces and component diagrams to show their functionalities. Last we assured that the framework design is compliant with the *recommendations*.

Table 3.12: RDA Data Citation Recommendations and how they were fit into the RDF* Data Citation Framework

Recommendation	Fit within the framework
R1 – Data Versioning	We design the *RDF* Store Utilities* component in a way to enable versioning via statement-level annotations using RDF* and SPARQL*.
R2 – Timestamping	The Write operations in *RDF* Store Utilities* use the metadata attributes valid_from and valid_until to timestamp any write operation. The read operation in *RDF* Store Utilities* queries data where the provided timestamp lies between valid_from and valid_until.
R3 – Query Store Facilities	*Query Store Utilities* offers functions to store and retrieve queries & associated metadata.
R4 – Query Uniqueness	We design functions *normalize_query(original_query)* and *compute_query_checksum(normalized_query)* as *Persistent Identification Utilities* to ensure query uniqueness. These utilities can be used within the query handler algorithm.
R5 – Stable Sorting	We argued that stable sorting needs a unique sort index either provided by the user or, if possible, derived from the dataset. Therefore, we define two functions, namely, *create_sort_index* and *sort_dataset*, which our found in *Persistent Identification Utilities*. Ideally, the publisher should provide a unique sort index already in the query's "order-by" clause.
R6 – Result Set Verification	To compute a result set checksum we simply use *compute_resultset_checksum* from *Persistent Identification Utilities*.
R7 – Query Timestamping	We can generate a timestamped query by simply providing the query and the timestamp to function *timestamp_query(query, timestamp* from *Persistent Identification Utilities*.
R8 – Query PID	To compute a technical query PID, we make use of function *generate_query_pid(query_checksum, timestamp)* from *Persistent Identification Utilities*.
R9 – Store Query	We design the component *Query Store Utilities* which offers the function *store(query, metadata)* to be compliant with R9.
R10 – Automated Citation Texts	To print out a citation text we design the *mint_query_pid* function that among others takes metadata as an argument, whereas it is the user's choice which citation metadata should result into the citation snippet.
R11 – Landing Page	In our design, the landing page sticks to the recommendation R11. We do not set any restrictions on the visuals. However, we did include an access control component targeting corporations so that only certain user groups can access the data and metadata.
R12 – Machine Actionability	The machine actionability is reached through the output of function *retrieve* from the *Citation* component which next to the dataset returns metadata in a machine-readable format, such as JSON. This very same data can be parsed and presented in a human-readable way.
R13 and R14 – Technology Migration and Verification	We discussed Technology Migration and Verification in 3.3 and covered some possible migration cases. However, in Section 3.1.1 we mentioned that we do not incorporate these recommendations into our framework design by means of functions or features of specific components.

CHAPTER 4

Implementation

We implemented the prototype in such a way that each of the components described in Chapter 3 has its corresponding python module. Hence, in this chapter, we discuss specific modules and use the term "modules" instead of components as it aligns better with the terms used in the language of our choice, namely Python. For each module in Section 4.1 we give an overview, provide implementation details in form of code snippets, lay out dependencies to other modules and show how it conforms to the proposed RDF* Data Citation Framework. In Section 4.2 we write about how we built and distributed our python API via Anaconda and Github.

4.1 RDF* Data Citation API

Instead of component, we are going to use the term *module* hereinafter which comes from the programming language that we used, namely Python. In general, the components from the high level view in Figure 3.2 match our API modules from this chapter. However, the nested components "Metadata Handler" and "SPARQL statement wrapper" were not directly translated into modules. We used a python class inside a module with a constructor as interface for former and text templates together with replace functions for latter component.

4.1.1 rdf_star module

The *rdf_star* module implements all functions from the *RDF* Store API* component and two distinct implementations of versioning where one has runtime performance and the other has low storage space consumption benefits. For convenience, we also included two additional functions, namely *reset_all_versions* and *_delete_triples* which are helpful when testing the solution to reset the database to an initial state. For each write operation (insert, update, outdate) we use SPARQL* templates that ensure that operations are being timestamped on statement-level. Each template has placeholders where the actual write operation is fit. The

read operation makes use of the *persistent_id_utils* module to create a timestamped query which is then sent to GraphDB's (post) endpoint to retrieve the data. In the following we illustrate and reveal how we used SPARQL* to implement each of the read and write operations and also explain the two different versioning modes, namely Q_PERF and SAVE_MEM. We also answer the question how many parameters are needed to connect this module to the examined RDF stores thereby get a fully operational RDF* interface for read and write operations.

SPARQL service endpoints

Before we dive into the implementations of individual read and write statements from our framework, we want to outline a fundamental concept that is used within every of these operations and which makes this implementation work with all RDF stores. The goal is to make read and write statements executable against any RDF store by finding an interface which is common for every RDF store. The interfaces that we found are usually referred to as SPARQL service endpoints and come from the idea of federated queries [1] In Listing 4.1 we present such a query which uses the SERVICE keyword to fetch triples from a remote graph. The argument that SERVICE takes is the location of the remote graph. If we break down the location we provided as an argument, we can easily read out that the remote graph comes from a respository named DataCitation_FHIR and runs on localhost with port 7200. In fact, this is the read-endpoint returned by GraphDB for this repository. We tested this query by executing it from Stardog's[2] query editor to get data from GraphDB [3]. The same principle can be applied to make write operations, just in this case, there is usually a second endpoint. Moreover, authentication can be required for some RDF stores. *SPARQLWrapper* is a library that provides the functionality to execute queries against remote triple stores with or without basic or digest authentication. This is the core library that we use in our implementation.

In practice, we tested our prototype with three data backends [4,5,6] and were successful along the connectivity dimension. However, we were not successful with Stardog in terms of retrieving data. This might be due to incompatible interfaces between SPARQLWrapper and Stardog. Further investigation is needed here. In the following sections we discuss read and write operations that can be executed once the service endpoints (and credentials) are set.

Listing 4.1: Federated query

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX fhir: <http://hl7.org/fhir/>
select ?s ?p ?o
where {
        SERVICE <http://128.131.205.115:7200/repositories/DataCitation_FHIR> {
        ?s ?p ?o .
        filter(?s = fhir:Identifier)
```

[1] https://www.w3.org/TR/sparql11-federated-query/
[2] https://www.stardog.com/
[3] https://www.ontotext.com/products/graphdb/
[4] https://www.ontotext.com/products/graphdb/
[5] https://jena.apache.org/
[6] https://www.stardog.com/

```
    }
} order by ?s ?p ?o
```

Insert

As mentioned in Section 3.2.1 we make use of the two attributes valid_from and valid_until to mark a start and end date of a triple. When we insert a triple we know what the start date is but we do not know the end date. It is valid until further notice. That is why we choose an artificial end date. The two attributes need to be enhanced with a prefix as in RDF stores we are working with IRIs. We constructed the unique prefix IRI <https://github.com/GreenfishK/DataCitation/versioning/> which is our Github repository for this project plus "/versioning/".
Listing 4.2 shows the template we use for insert with the placeholders marked as *{number}*. This snippet demonstrates the employment of SPARQL* and its statement-level annotation capability. SPARQL's function *now()* creates a valid_from date as of execution time. The valid_until date is an artificial date and is set for all triples using *insert_all_rows* function (See later in Section 4.1.1). Upon calling *insert(new_triples, prefixes)* the placeholders get replaced with the actual entities from the provided triple. For each triple one insert statement is executed against the triple store. In future we might consider bulk inserts.
In Listing 4.3 we demonstrate how the template is populated if we execute the insert function and thereby plug in a new triple ex:Obama ex:occupation "author" with ex:<http://example.com/> as prefix. Notice how two additional prefixes *vers* and *xsd* were inserted by the use of the prefix handler (see later in Section 4.1.5).

Listing 4.2: Insert template for versioned triples

```
# Prefixes
{0}

# Insert statement
insert {
{1} {2} {3}.
<<{1} {2} {3}>  vers:valid_from ?newVersion.
<<{1} {2} {3}>>  vers:valid_until "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime.
}
where {
    BIND(xsd:dateTime(NOW()) AS ?newVersion).
}
```

Listing 4.3: Filled out insert template example for versioned triples

```
# Prefixes
PREFIX vers: <https://github.com/GreenfishK/DataCitation/versioning/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ex: <http://example.com/>

# Insert statement
insert {
    ex:Obama ex:occupation "author".
    <<ex:Obama ex:occupation "author">>  vers:valid_from
        ?newVersion.
    <<ex:Obama ex:occupation "author">>  vers:valid_until
        "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime.
}
```

```
where {
    BIND(xsd:dateTime(NOW()) AS ?newVersion).
}
```

Update

We implemented the update function to update the object of a provided triple and in fact replace it with a new object. The old triple does not get deleted, as this would be against the versioning principle, but gets outdated. Simultaneously, a triple with the new object gets inserted and versioned. So, updated can be seen as a combination of insert and outdate where it does not matter which one gets executed first. The template that incorporates this logic can be inspected in Figure 4.4.
We continue our example from the previous section in Figure 4.5. Obama now has a new job as a film producer as he is tired of writing books. That is why we need to update his occupation "author" by replacing the artificial end date of this triple with the current timestamp, inserting a new triple with "film_producer" as object, and versioning this new triple with by setting *valid_from* to the current timestamp and *valid_until* to the artificial end date.

Listing 4.4: Update template for versioned triples

```
# prefixes
{0}

delete {
    <<?subjectToUpdate ?predicateToUpdate ?objectToUpdate>> vers:valid_until
        "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime
}
insert {
    # outdate old triple with date as of now()
    <<?subjectToUpdate ?predicateToUpdate ?objectToUpdate>> vers:valid_until ?newVersion.
    # update new row with value and timestamp as of now()
    ?subjectToUpdate ?predicateToUpdate ?newValue. # new value
    # assign new version.
    <<?subjectToUpdate ?predicateToUpdate ?newValue>> vers:valid_from ?newVersion
        ;vers:valid_until "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime.
}
where {
    bind({1} as ?subjectToUpdate)
    bind({2} as ?predicateToUpdate)
    bind({3} as ?objectToUpdate)
    bind({4} as ?newValue). #new Value
    # versioning
    <<?subjectToUpdate ?predicateToUpdate ?objectToUpdate>> vers:valid_until ?valid_until.
    filter(?valid_until = "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime)
    BIND(xsd:dateTime(NOW()) AS ?newVersion).
    # nothing should be changed if old and new value are the same
    filter(?newValue != ?objectToUpdate)
}
```

Listing 4.5: Filled out update template example for versioned triples

```
# prefixes
PREFIX vers: <https://github.com/GreenfishK/DataCitation/versioning/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ex: <http://example.com/>
```

```
delete {
    <<?subjectToUpdate ?predicateToUpdate ?objectToUpdate>> vers:valid_until
        "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime
}
insert {
    # outdate old triple with date as of now()
    <<?subjectToUpdate ?predicateToUpdate ?objectToUpdate>> vers:valid_until ?newVersion.
    # update new row with value and timestamp as of now()
    ?subjectToUpdate ?predicateToUpdate ?newValue. # new value
    # assign new version.
    <<?subjectToUpdate ?predicateToUpdate ?newValue>> vers:valid_from ?newVersion
        ;vers:valid_until "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime.
}
where {
    bind(ex:Obama as ?subjectToUpdate)
    bind(ex:occupation as ?predicateToUpdate)
    bind("author" as ?objectToUpdate)
    bind("film_producer" as ?newValue). #new Value
    # versioning
    <<?subjectToUpdate ?predicateToUpdate ?objectToUpdate>> vers:valid_until ?valid_until .
    filter(?valid_until = "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime)
    BIND(xsd:dateTime(NOW()) AS ?newVersion).
    # nothing should be changed if old and new value are the same
    filter(?newValue != ?objectToUpdate)
}
```

Outdate

To outdate a triple means that it is not be returned anymore when we query live data or data that is past the outdate event. This is established by simply changing the artificial end date to a system timestamp as of execution time. "Changing" means deleting the annotation and appending a new annotation to the same triple. Trivially, only triples from the live data subset can be outdated. Live data can be easily filtered as all triples that are marked with an artificial end date correspond to it.
In Listings 4.6 and 4.7 we show the template and its population at runtime respectively and thereby finish our story line with Obama who retires at this point. Therefore, we outdate the inserted triple with "film_producer" as the object by actualy deleting the metadata triple with the artificial end date and inserting a new metadata triple with the current timestamp as end date. This means, that Obama currently has no occupation and querying live data to find out Obama's current occupation would return an empty dataset.

Listing 4.6: Outdate template for versioned triples

```
# prefixes
{0}

# Delete and insert statements
delete {
    <<?subjectToOutdate ?predicateToOutdate ?objectToOutdate>> vers:valid_until
    "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime
}
insert {
    # outdate old triples with date as of now()
    <<?subjectToOutdate ?predicateToOutdate ?objectToOutdate>> vers:valid_until ?newVersion.
```

```
}
where {
    bind({1} as ?subjectToOutdate)
    bind({2}as ?predicateToOutdate)
    bind({3} as ?objectToOutdate)
    # versioning
    <<?subjectToOutdate ?predicateToOutdate ?objectToOutdate>> vers:valid_until
    "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime .
    BIND(xsd:dateTime(NOW()) AS ?newVersion).
}
```

Listing 4.7: Filled out update template example for versioned triples

```
# prefixes
PREFIX vers: <https://github.com/GreenfishK/DataCitation/versioning/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ex: <http://example.com/>

# Delete and insert statements
delete {
    <<?subjectToOutdate ?predicateToOutdate ?objectToOutdate>> vers:valid_until
    "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime
}
insert {
    # outdate old triples with date as of now()
    <<?subjectToOutdate ?predicateToOutdate ?objectToOutdate>> vers:valid_until ?newVersion.
}
where {
    bind(ex:Obama as ?subjectToOutdate)
    bind(ex:occupation as ?predicateToOutdate)
    bind("film_producer" as ?objectToOutdate)
    # versioning
    <<?subjectToOutdate ?predicateToOutdate ?objectToOutdate>> vers:valid_until
    "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime .
    BIND(xsd:dateTime(NOW()) AS ?newVersion).
}
```

Versioning Modes

In Section 3.2.1 we already introduced two possible ways how to initialize versioning in a triple store. The question boils down to whether to initially include a start date on the triples that existed prior to versioning where we potentially do not know the real start date. Including both, start and end date makes queries retrieving a specific dataset version simpler compared to the second approach, which is to initially only have an end date on triple level. Latter approach, however, saves memory as only one additional metadata triple per data triple is added. We included both approaches in the *rdf_star* module and called them *Versioning Modes*. The function *version_all_rows* asks for either *Q_PERF MEM_SAVE* which correspond to the first and second approach, respectively. Depending on the versioning mode this function either sends the insert statement in Listing 4.8 or the one in Listing 4.9 to the RDF store update endpoint to version all triples in the triple store.

Listing 4.8: Query performance version_all_rows

```
{prefixes}
insert
{
    <<?s ?p ?o>> vers:valid_from
    ?currentTimestamp.
    <<?s ?p ?o>> vers:valid_until
    "9999-12-31T00:00:00.000+02:00"
    ^^xsd:dateTime.
}
where
{
   ?s ?p ?o .
   BIND(xsd:dateTime("{1}"^^xsd:dateTime)
   AS ?currentTimestamp).
}
```

Listing 4.9: Memory saving version_all_rows

```
{prefixes}
insert
{
    <<?s ?p ?o>> vers:valid_until
    "9999-12-31T00:00:00.000+02:00"
    ^^xsd:dateTime.
}
where
{
   ?s ?p ?o .
}
```

Next we explain how queries that employ versioning (also called timestamped queries) are affected by these two versioning modes. Let us first describe the basic idea of our timestamping queries approach. Each SPARQL query has a *basic graph pattern (BGP)* which consists of triple statements. These triple statements can be considered as joins like in relational SQL. If we think of joining tables with versioned data where the timestamp is part of the primary key we need to include the timestamp attribute in every JOIN-condition where tables are joined. This effectively means for SPARQL that we need to narrow down the set of data triples for each triple statement by using their versioning labels, *valid_from* and *valid_until*, as additional triple statements combined with a filter to select only those triples where the provided timestamp lies in between the start and end date. This must be done inside every *basic graph pattern*, which can occur multiple times and be nested.
Code snippets in Listings 4.11 and 4.10 show the extensions templates that we need to attach to every BGP. We see that in the Q_PERF mode we need to provide two nested triple statements for the versioning and one filter to state that the time of execution must lie somewhere between the start and end date. In the MEM_SAVE mode we need to lower the restrictions on the nested triple statement which carries the *valid_from* by saying that it is optional. The filter operation splits into two filter operations: One that check whether execution timestamp lies before the end date and one that goes hand in hand with the optional statement, asking for the execution timestamp to be after the *valid_from* date but only if such a date exists. As the MEM_SAVE mode uses a left join we assume that the Q_PERF mode is faster, hence the name. In the Section 4.1.3 we show an example of a timestamped query where these versioning extensions are put into use.

Listing 4.10: Query performance SPARQL template

```
<<{datatriple}>> vers:valid_from
{valid_from}.
<<datatriple>> vers:valid_until
{valid_until}.
filter(?valid_from <= ?TimeOfExecution &&
?TimeOfExecution_{BGP_index} < ?valid_until)
```

Listing 4.11: Memory saving SPARQL template

```
<<{datatriple}>> vers:valid_until
{valid_until}.
filter(?TimeOfExecution_<BGP_index> <
{valid_until})
optional {<<{datatriple}>>>
vers:valid_from {valid_from}.}
filter (!bound({valid_from}) ||
{valid_from} <= ?TimeOfExecution_{BGP_index})
```

4.1.2 query_store module

The *query_store* module implements all functions and the database schema described in the design chapter. We use python's sqlite3 as lightweight query storage and sqlalchemy as database API. Let us briefly describe each of these functions.
The *store* function, next to storing query and metadata, implements a case distinction for queries that are new and those that are not. The decision whether a query is new or not happens outside of this function, thus, it is a flag that needs to be set by the caller. As already mentioned in Section 3.2.3 we use the table *query_hub* to collect information that are valid for the query even if the dataset changes after a query PID has been minted. We write to this table only if the query is new. Additionally, we insert a row into *query_satellite* if the query is not new to update dynamic metadata (result set checksum, query PID, timestamped query, publisher, ...).
get_query retrieves query and metadata by the query's PID. Here we get all the information from both query store tables.
get_last_execution is what we need to return the latest available information to a stored query. Here we use the query's checksum to find the query in *query_hub* and then retrieve the latest metadata for that query by *last_execution_timestamp*. We use latter attribute to retrieve the associated row from the *query_satellite* table.

4.1.3 persistent_id_utils module

Next to the functions from the design chapter we employ, among other helper functions, one that is crucial for query normalization and query timestamping as it revolves around SPARQL's query algebra [7] which was a huge research focus during the implementation. We outline its usage in the subsequent Sections *Timestamp query* and *Normalize query* and In the remaining sections we shortly explain basic ideas of the other utilities and provide code snippets where needed.

[7] https://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html

Normalize query

To normalize a query one first needs to identify equivalent semantics, thus, different ways to write a query without changing the outcome. Common examples are labels like variable names and prefixes, optional syntax like brackets or specific keywords and abbreviations. Specifically in SPARQL we have different types of paths, such as sequence paths, inverted paths and alternative paths. These can all be rewritten using only triple statements and logical operators. Then we also have a few options to filter for data where equal filter expressions are seemingly hard to check for equality. For latter, consider the two equivalent queries in Listings 4.12 and 4.13. To normalize what is inside the concat function we would need to employ different techniques that actually compute the results of these expressions and then compare them. That is why we limit the query normalization function to the equivalent expressions described Table 4.1. This table aims to capture semantics and structures that are commonly used in general but more specifically to simplify queries, e.g. by leaving out the where clause, writing "a" instead of "rdf:type" and using sequence paths. To each equivalent expression we add normalization measures in Table the second column which we perform when we call the function *normalize_query_tree*. The suffix "tree" in the function name refers to the query's "SPARQL algebra tree" which is the main tool we use for normalization and we are going to discuss it in the following lines.

Listing 4.12: Example query

```
select ?x ?y  {
    ?x ?y ?o
    filter(?o = concat(str(12 + 4),
                       "abc"))
}
```

Listing 4.13: Equivalent example query

```
select ?x ?y  {
    ?x ?y ?o
    filter(?o = concat(str(1 + 15),
                       "ab",
                       "c"))
}
```

SPARQL queries can be broken down into individual elements and these elements can be fit into an n-ary tree, where the number n could be defined by the number of triples or variables as either are nodes in such a tree. Once we have such a structure we can traverse the tree and perform operations on existing nodes or insert new nodes into the tree. In *SPARQL algebra* there is a well-defined grammar in EBNF notation[8]. We can therefore set conditions for specific nodes while traversing the tree and perform our normalization measures. To get such a tree we translate the query using python's RDF library *rdflib*. After normalization we translate the query tree back into query text and save either in the query store.
Let us look at a simple example query 4.14 and its query algebra in 4.15. The query tree reveals how all the query elements are nested into the tree. Usually, in the leaf nodes we find a *basic graph pattern* with triple statements which also might encompass paths. We can find it notated as p = BGP in the tree. On the same level we have a *RelationalExpression* node, which is the expression in our filter. Both, the BGP and the expression are filter inputs in the tree representation. The BGP being an input to the filter expression might be a bit counter-intuitive

[8] https://www.w3.org/TR/sparql11-query/#grammar

Table 4.1: Equivalent SPARQL expressions and normalization measures

Semantically equivalent SPARQL phrases	Normalization measures
The WHERE clause is optional.	A where clause is always inserted.
rdf:type" predicate can be replaced by "a".	rdf:type is always replaced by "a".
If the same subject is used multiple times in subsequent triple statements (separated by a dot) it can be left out in all the subsequent triple statements where the subject occurs. Instead of the subject variable name a semicolon is written in subsequent triple statements where the same subject as in the first statement should be used.	Triples are always stated explicitly without leaving out the subject.
The order of triple statements has no effect on the query semantics.	Triple statements are ordered by the number of bindings. In cases where triple statements have equal number of bindings this might yield two different permutations of triple statements between two queries.
Aliases via BIND keyword just rename variables but they do not affect the query result.	Aliases are removed if they are used to rename variables. They are not removed if they are used to label more complex expressions.
Variable names have no effect on the query semantics.	Variables in the query tree are replaced by letters from the alphabet. Only up to 26 variables are supported.
Finding variables that are not bound can be written in two ways: 1. with the OPTIONAL keyword adding the optional triple combined with filter condition !bound(?var); 2. with "filter not exists (triple statement)".	"filter not exists triple statement" expressions are converted into the "filter + !bound" expression.
Inverting paths can be achieved in following ways: 1. Explicitly stating two or more triple statements. 2. Using the following syntax: ?x prefix:predicate / ^prefix:predicate ?y	Inverted paths are resolved to explicit triple statements.
Sequence paths can reduce the number of triples in the query statement and are commonly used. They can also be rewritten to common triple statements.	Sequence paths are resolved to explicit triple statements
Prefixes are just labels and they are replaceable by any allowed char sequence without impacting the query semantics.	Prefixes are resolved and full resource names are stated.

but we can think of it as the filter being applied on the BGP, which makes sense. On the same level as the *Filter* node we have our projection variables stored in the *PV* node. *Filter* and *PV* are passed over to *Project*, whereas *PV* is always part of the *Project* node. To reconstruct the query text we could stop at this level as the higher node *SelectQuery*, which redundantly includes the same *PV* set, is not necessary anymore. This is because only select queries can have project variables as the other SPARQL query types ASK, CONSTRUCT and DESCRIBE do not have any project variables at all. Note also that the node *_vars* is not needed as it also carries redundant information. So far, we have found this to be true for all our test queries (see on our Github page [9])

Listing 4.14: Example query

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>

select ?x ?y {
    ?x foaf:knows / ^foaf:knows ?y
    FILTER(?x != ?y)
}
```

Listing 4.15: Example query tree

```
SelectQuery(
    p = Project(
        p = Filter(
            expr = RelationalExpression(
                expr = x
                op = !=
                other = y
                _vars = set()
                )
            p = BGP(
                triples = [(rdflib.term.Variable('x'),
                Path(http://xmlns.com/foaf/0.1/knows /
                Path(~http://xmlns.com/foaf/0.1/knows)),
                rdflib.term.Variable('y'))]
                _vars = {rdflib.term.Variable('x'), rdflib.term.Variable('y')}
                )
            _vars = {rdflib.term.Variable('x'), rdflib.term.Variable('y')}
            )
        PV = [rdflib.term.Variable('x'), rdflib.term.Variable('y')]
        _vars = {rdflib.term.Variable('x'), rdflib.term.Variable('y')}
        )
    datasetClause = None
    PV = [rdflib.term.Variable('x'), rdflib.term.Variable('y')]
    _vars = {rdflib.term.Variable('x'), rdflib.term.Variable('y')}
    )
```

If we apply our normalization measures from Table 4.1 on our query tree, following steps are

[9] https://github.com/GreenfishK/DataCitation/tree/master/tests/algebra_to_text/test_data

executed:

1. The sequence path is resolved using a dummy variable ?dummy1.
2. The inverted path is resolved, thereby switching variables ?dummy1 and ?y.
3. Variables ?x, ?y and ?dummy1 in the query tree is replaced by ?a, ?b and ?c.
4. The prefix "foaf" is replaced by its IRI.
5. *_vars* nodes is cleared because we do not perform any normalization steps on this node.

In Listing 4.16 we can see the result of our normalization and furthermore the back-translation into the query text in Listing 4.17. Prior to this work translating an algebra tree back into a query text did not exist as a function in rdflib. We added this functionality by making a pull request to the official github repository of rdflib [10]. We, however, do not discuss this function in this thesis.

Listing 4.16: Normalized query tree

```
SelectQuery(
    p = Project(
        p = Filter(
            expr = RelationalExpression(
                expr = a
                op = !=
                other = b
                _vars = set()
                )
            p = BGP(
                triples = [(rdflib.term.Variable('a'),
                rdflib.term.URIRef('http://xmlns.com/foaf/0.1/knows'),
                rdflib.term.Variable('c')), (rdflib.term.Variable('b'),
                rdflib.term.URIRef('http://xmlns.com/foaf/0.1/knows'),
                rdflib.term.Variable('c'))]
                _vars = set()
                )
            _vars = set()
            )
        PV = [rdflib.term.Variable('a'), rdflib.term.Variable('b')]
        _vars = set()
        )
    datasetClause = None
    PV = [rdflib.term.Variable('a'), rdflib.term.Variable('b')]
    _vars = set()
    )
```

[10] https://github.com/RDFLib/rdflib

Table 4.2: Rough estimation of time complexity of query normalization

Step	Time complexity	Comment
query –> algebra tree	$O(2^n * k)$	Worst-case assumption. Assuming backtracking algorithm for regex. Variable k is the number of iterations needed to translate the query and n is the expression size.
normalizing algebra tree	$O(N * (j - l))$	We visit each node of the N nodes once and we do this j times where j is the number of normalization steps. There are some normalization steps, such as "optional where clause" and "rdf:type instead of 'a'", that are implicitly solved by the query algebra and therefore do not need to be performed.
algebra tree –> query	$O(N * m/2)$	We visit each node once and translate the node into query text. To translate the node we need to iterate through if-conditions until we hit the right one that matches the node. We assume an average time of m/2 where m is the number of if-conditions.

Listing 4.17: Normalized query

```
SELECT ?a ?b{
    FILTER(?a != ?b) ?a <http://xmlns.com/foaf/0.1/knows> ?c.
    ?b <http://xmlns.com/foaf/0.1/knows> ?c.
}
```

Let us roughly estimate the time complexity of this approach and then raise two questions. In total, we have three steps: 1. Build the query algebra from the query; 2. Normalize the query algebra; 3. Translate the query algebra into query text. In Table 4.2 we see a complexity estimation for each step. The two questions are now: Can we optimize this approach and are there any benefits of doing it? Instead of performing the normalization measures on the query algebra we could do it directly on the query text by means of regular expressions which were used to build the algebra tree in the first place. The reason why we decided against it is the much higher implementation effort. A partial optimization would be to omit the third step which we explain in Section 4.1.3.

Timestamp query

In Section 4.1.1 we saw how we need to adapt the SPARQL code not only to version all triples but also to read versioned triples. For either versioning mode we outlined certain additional triple statements and filters that need to be attached to the query for each triple statement. The function *timestamp_query* is the place where we actually implement this. The idea is simple:

For each BGP we need to insert as many dummy triples into the BGP as there are triple statements in the BGP as each triple statement represents another subset (We explained this Section 4.1.1 - Versioning Modes by comparing it to relational tables that are joined). These dummy triples then get replaced by either snippet in Listing 4.10 or the one in Listing 4.11 with simple string replacement tools. In listing 4.18 we highlighted the injected versioning extensions from the MEM_SAVE mode and versioning prefixes. We see that there are two triple statements and for each of them we have a matching versioning block, highlighted in blue. The variables for versioning in these blocks are enumerated, such as *?triple_statement_**0**_valid_from* and *?triple_statement_**0**_valid_until* for the first triple statement.
Additionally, we find such an enumeration in the *TimeOfExecutionBGP_0* variable. This is because there could be multiple BGPs and it is easier to insert the same execution timestamp for each BGP along with the versioning extensions than to define a globally accessible timestamp within the query. Although, this would be an alternative.

Listing 4.18: Example for a timestamped query in memory saving mode

```
PREFIX vers: <https://github.com/GreenfishK/DataCitation/citing/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

PREFIX ex: <http://example.come>

SELECT ?name ?occupation {
?person ex:name ?name.
?person ex:occupation ?occupation.
FILTER(?name = "Barack Obama"@en)

<<?person ex:name ?name>> vers:valid_until
?triple_statement_0_valid_until.
filter(?TimeOfExecutionBGP_0 < ?triple_statement_0_valid_until)
optional {
<<?person ex:name ?name>> vers:valid_from
?triple_statement_0_valid_from.
}
filter (!bound(?triple_statement_0_valid_from)
|| ?triple_statement_0_valid_from <= ?TimeOfExecutionBGP_0)

<<?person ex:occupation ?occupation>> vers:valid_until
?triple_statement_1_valid_until.
filter(?TimeOfExecutionBGP_0 < ?triple_statement_1_valid_until)
optional {
<<?person ex:occupation ?occupation>> vers:valid_from
?triple_statement_1_valid_from.
}
filter (!bound(?triple_statement_1_valid_from)
|| ?triple_statement_1_valid_from <= ?TimeOfExecutionBGP_0)

bind("2021-06-19T16:43:36.882847+01:00"^^xsd:dateTime as ?TimeOfExecutionBGP_0)

}ORDER BY ?mention
```

Compute query and result set checksum

To compute the checksum of the normalized query we employed a simple sha256 algorithm that computes a hash value out of the input string. Next to the normalized query we also allow for the normalized query algebra tree as an input. As the normalized query is not meant to be executed but only serves the purpose to compare a query that is about to be stored against queries from the query store, we could do this comparison also with the queries' algebra trees. This also answers why we would not need to back-translate the normalized query algebra tree into a query, as hinted in the last paragraph of Section 4.1.3 - Normalize query. Despite the fact, we still do reverse the algebra tree and thus compute a checksum from the normalized query. The main reason is that the implementation of SPARQL's query algebra might change in terms of node names, tree structure and due to bug fixes. This would make older query algebra trees incomparable with newer ones. On top, if there is a need to visually compare two normalized queries and e.g. find out why they are not equal we could easier do this with queries as they are more compact than query trees. We also believe that generating or retrieving citation snippets is not time critical and we can therefore trade off performance for stability.

We compute the result set checksum using pandas' *hash_pandas_object* function which return a series (or vector) of integer row hash sums. These we concatenate as strings and compute a hash sum of the whole string using the same algorithm as for the query checksum.

Generate query PID

As we saw in figure 3.4 we need two inputs to generate a query PID - query checksum and execution timestamp. Now there are various ways how to cast these two inputs into a PID, such as computing hash values of either inputs and then concatenating them or first concatenating them and then computing a hash value. However, we opted for the simplest one and that is to concatenate both inputs. This PID serves as a primary key in the *query_satellite* table to identify citations but is otherwise not used in this not resolvable "raw form". In 4.1.4 we show an interface that can be used by a *Query Builder UI* to transform PIDs into landing page URLs.

Describe dataset

Our implementation of *describe_dataset* uses simple descriptive statistics expressed in natural language together with one heuristic to create a dataset description from the dataset. The simple statistics for each column are:

1. number of non-empty values
2. frequencies
3. max values
4. unique values (opposite of frequencies)

If a column has one and the same value in every row, meaning that its frequency equals the number of rows we assume that the dataset "is about" this value. This does not have to be necessarily true because one value could occur in every row "by accident" and not because we filtered for it. This is why the query text comes in handy to derive such information, however, we have not implemented it yet. In Listing 4.19 we can see an example query from the news dataset where we filter for "Obama". The derived dataset description is shown in Listing 4.20. We see that our heuristic inferred that the dataset is not only about Obama but also about the democratic party because we find this value to be unique in column *party_label*. Whether this summary is representative is up to the creator or publisher to decide.

Listing 4.19: Example query from the news dataset where we filter for Obama

```
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX publishing: <http://ontology.ontotext.com/publishing#>

select ?personLabel ?party_label ?document ?mention where {
    ?mention publishing:hasInstance ?person .
    ?document publishing:containsMention ?mention .
    ?person pub:memberOfPoliticalParty ?party .
    ?person pub:preferredLabel ?personLabel .
    ?party pub:hasValue ?value .
    ?value pub:preferredLabel ?party_label .

    filter(?personLabel = "Barack Obama"@en)
} order by ?mention
```

Listing 4.20: Derived dataset description from the dataset

```
This dataset is about: Democratic Party@en, Barack Obama@en

Each column has following number of non-empty values:
document: 6,
mention: 6,
party_label: 6,
personLabel: 6

For each column following frequencies were observed:
document: 5,
mention: 1,
party_label: 6,
personLabel: 6

Each column has following max/top values:
document:  http://www.reuters.com/article/2014/10/10/
us-usa-california-mountains-idUSKCN0HZ0U720141010,
mention: http://data.ontotext.com/publishing#Mention
-96cd9530c126974107c405f240907337db267d369e851e904b21ad75955473af,
party_label: Democratic Party@en,
personLabel: Barack Obama@en

For each column following unique values were observed (opposite of frequencies):
document: 2,
mention: 6,
party_label: 1,
personLabel: 1
```

Create sort index and sort

In order to sort the dataset we first need a unique sort index. This can either be a primary key provided by the user or one that is derived from the dataset. To foster automation we created an algorithm that can accomplish latter option. The idea is to find the "simplest" possible unique (multi-)index. Simple hereby means that each (multi-)index is a minimum set of columns. E.g. only one column could be enough to serve as a unique index for sorting. The algorithm starts by iterating through every single column and checking whether it would make up a unique index or not. If no unique index is found it goes on by trying out every column pair combination. If still no unique indexes are found it goes on with triple combinations and so on.
One challenge we face with the index derivation is that there could be more possible solutions derived. E.g. there could be two columns where each has only one unique value or two possible compositions of columns where each composition would yield an unambiguous sort order. To select the combination that most likely remains valid as a primary key even if further triples are added we implemented a criteria to further distinguish between the composite keys. For each composite key the following steps are executed: 1. Count the number of distinct values for each key attribute (column); 2. Sum up all the counted distinct values. The composite keys with the maximum sum of distinct key attribute values is returned.

If the number of unique composite sort keys is still greater than one, an exception is raised to the caller (e.g. publisher) asking to confirm one from the suggested ones or provide his own unique sort index.

Metadata

The *Metadata* class comprises metadata attributes to be stored in the query store other than the ones (checksum, query PID, ...) related to the *recommendations*. Another distinction to the recommended metadata attributes is that these do not get automatically derived by the API. However, in a Data Citation system these can be automatically retrieved. We already provided possible sources in Table 3.11. We packed all parameters as described in the design Section 3.2.3 into a constructor (python's __init__ function) and set them to be optional. Even though every metadata attribute and value could be provided as a key-value-pair via the *other_citation_data* member we wanted to explicitly list DataCite's manadatory field as arguments in order to promote this metadata schema. Additionally, we put the data citation snippet text as member of the init function. The class resides within the *citaion_utils* module. We wanted to encapsulate all metadata related classes and functions in one module and therefore did not create a separate one for *Metadata*. Instances of this class can be found in the query handler and query store module where every function of either modules uses it to either store or retrieve the carried information.
This class furthermore provides two functions to encode and decode its members as JSON, namely *to_json* and *set_metadata* respectively. This is needed because the query store does not list every possible metadata attribute as column, so we need to use one column for all the additional attributes. Column *citation_data* in the *query_satellite* serves this purpose (see in Figure 3.6) and is used like in the example in Table 4.3.

One noticeable member of *Metadata* is *identifier*. We use query PIDs for identification of our datasets, however, the query PIDs we generate with *persistent_id_utils.generate_query_pid* are no URLs that point to a landing page. That is why we need to store the resolvable PID in here. In Section 4.1.4 we describe what we additionally do with this member during the query handler algorithm.

Generate citation snippet

To generate the citation snippet text we use one simple function with a keyworded, variable-length argument list (kwargs). The idea is to be flexible with what information should be inside the citation snippet. The parameter values get concatenated and separated by a comma in the same order they are passed. Currently, there is no validation on the arguments. Possible validations could be deployed through policies like we explained in Section 2.6.

4.1.4 Query_handler module

The query_handler module can be considered as the core module as it employs functions from all the modules we have discussed so far to handle queries and and retrieve minted datasets via query re-execution. It has two functions - *mint_query_pid* and *retrieve* - and also

Table 4.3: Query Satellite Example

query pid	query checksum	...	citation data
123202100607T12:00:00	123		{"identifier": "https://doi.example.com/10.1594/DataCitation.667386", "creator": "Filip Kovacevic", "title": "Running Example", "publisher": "Filip Kovacevic", "resource_type": "RDF/Dataset", "other_citation_data": { "contributer": "Tomasz Miksa" }, "result_set_description": "All news articles where Obama has been mentioned" }

a constructor (=__init__) where most importantly the caller must provide SPARQL endpoints and optionally credentials. On top we enriched this module with boolean attributes to let the user upon execution of the query handler algorithm know whether the query already exists and if so, whether the result set has changed.

Mint Query PID

Let us first break down the *mint_query_pid* function by its arguments. The *mint_query_pid* function takes the query as first argument. This is not only a design choice to ensure that the dataset is actually queryable but it is moreover needed to lookup the query in the query store, execute the query and then compare result and query checksums.
The metadata argument is of type *persistent_id_utils.Metadata* and encompasses all metadata that is provided outside of the API and not covered by the *recommendations*.
The *execution_timestamp* is used to timestamp the query as discussed in Section 4.1.3 and thereby retrieve specific dataset, such as live data or historical datasets with an older timestamp. We therefore do not set any restrictions on the timestamp but we do recommend to simply pass the current system timestamp as this guarantees that the newest version of the dataset is queried.
The *create_identifier* argument which is a function that takes the query PID as an argument should be implemented by the user to create a persistent URL. It is applied on the query PID and the resulting URL is stored in *metadata.identifier*. If this function is not provided, the identifier is set to the query PID.
Calling *mint_query_pid* with the arguments above triggers an algorithm like we illustrated in

Figure 4.1. The algorithm starts off with a sequential flow and thereby creates metadata as described in R4-R8 including a validation on the query's "order by"-clause. Then it steps into a control flow where the three cases as described in Section 3.1.2 are covered. Only if the query is new a persistent identifier is created by using *create_identifier* on the query PID followed by the creation of the citation snippet and the persistence of metadata into the query store. Finally the citation snippet is returned to the caller.

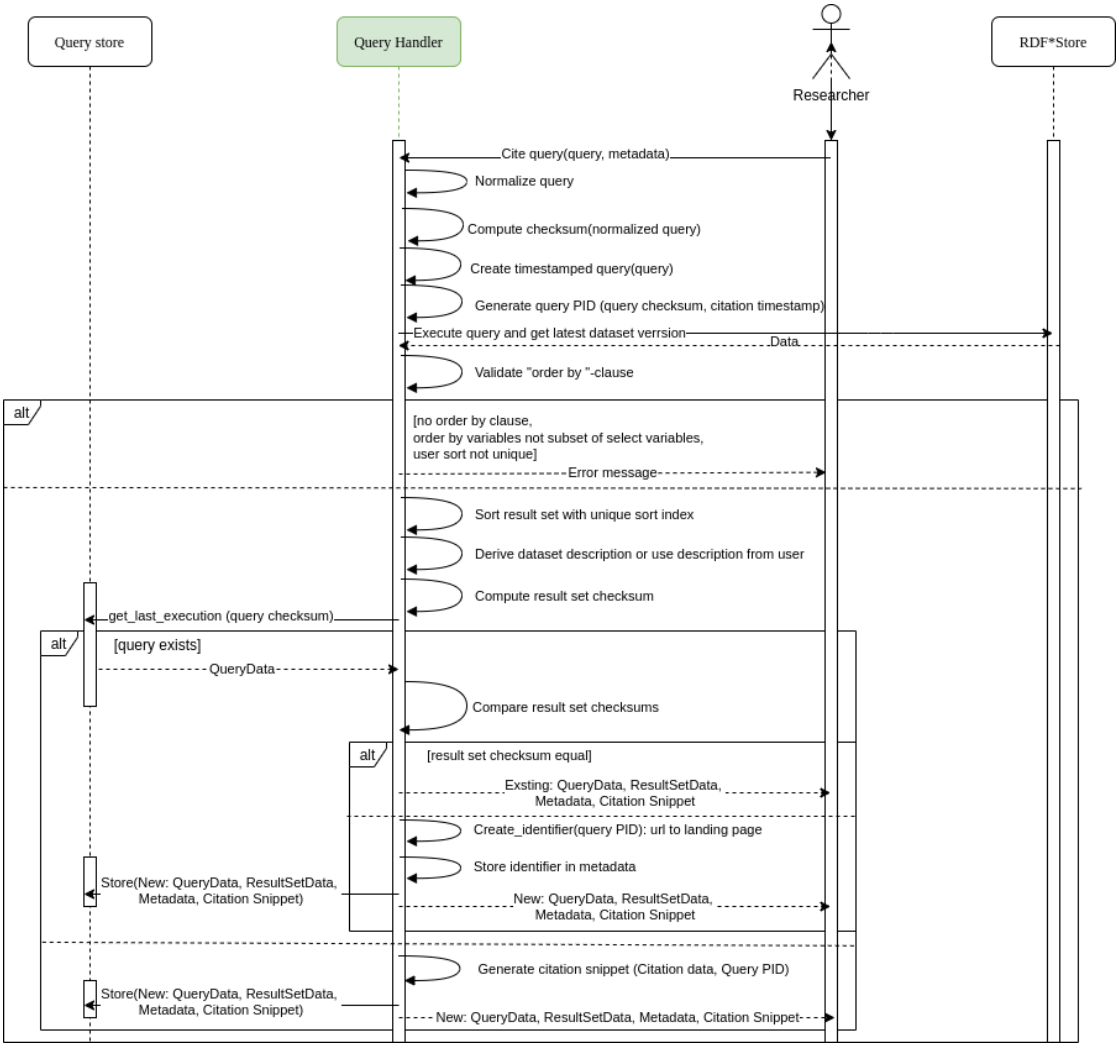


Figure 4.1: An algorithm for handling queries including recommendations R4-R10 and R12

In Listing 4.21 we show an example of how to use this function to get citation snippet and query store metadata. We start of by defining how to create a resolvable PID from the technical *query_pid*. Latter is computed by the API but it is up to the user to convert the query PID into an URL that points to a landing page. Next we provide metadata by using the MetaData constructor. In addition to the DataCite's obligatory attributes we also have *other_citation_data*

and *result_set_description*. In the third step, we define a query that we want to execute. All these data are now plugged into the *mint_query_pid* function. Once this function is executed, one can retrieve citation snippet and metadata from the *query_handler* object and additionally lookup whether the query was new and whether the dataset has changed since the last execution.

Listing 4.21: Mint dataset - example

```
# Step 1
def create_identifier(query_pid: str):
    # Write your own code to create an URL out of a query PID
    identifier = "http://www.mylandingpage.com/" + query_pid
    return identifier

# Step 2
citation_metadata \
= persistent_id_utils.MetaData(identifier="DOI_to_landing_page", creator="Filip Kovacevic",
                               title="Judy Chu occurences", publisher="Filip Kovacevic",
                               publication_year="2021", resource_type="Dataset/RDF data",
                               other_citation_data={"Contributor": "Tomasz Miksa"},
                               result_set_description = "All news articles where Obama
                               has been mentioned.")

# Step 3
query_test =
"""
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX publishing: <http://ontology.ontotext.com/publishing#>

select ?personLabel ?party_label ?document ?mention where {
    ?mention publishing:hasInstance ?person .
    ?document publishing:containsMention ?mention .
    ?person pub:memberOfPoliticalParty ?party .
    ?person pub:preferredLabel ?personLabel .
    ?party pub:hasValue ?value .
    ?value pub:preferredLabel ?party_label .
    filter(?personLabel = "Barack Obama"@en)
}
"""

# Step 4
citation_data = query_handler.mint_query_pid(
select_statement=query_text,
citation_metadata=citation_metadata,
create_identifier=create_identifier)

# Step 5 - Retrieve metadata and citation snippet
citation_snippet = query_handler.citation_metadata.citation_snippet
citation_metadata = query_handler.citation_metadata
dataset_metadata = query_handler.result_set_utils
query_metadata = query_handler.query_utils
yn_query_exists = query_handler.yn_query_exists
yn_result_set_changed = query_handler.yn_result_set_changed
```

Retrieve

Retrieve does nothing more than retrieving the query and metadata by the query PID from the query store, taking the timestamped query to re-execute it against the RDF* store and returning dataset and metadata to the caller. The metadata gets encoded as JSON object. We thereby categorize metadata into query metadata, dataset metadata, citation metadata and the citation snippet. The citation snippet can include metadata from every of the former three categories and therefore cannot be fit into one of these. Hence, it is on the same level as the other categories in the JSON tree. In 4.22 we continue the example from 4.21 and show the usage of *retrieve*.

Listing 4.22: Retrieve minted dataset - example

```
query_pid = citation_data.query_utils.pid
dataset, meta_data = query_handler.retrieve(query_pid)
```

4.1.5 Prefixes module

The *prefixes module* is an instance of the prologue handler and a helper module that provides useful functions to deal with query prefixes that are declared in the prologue of the query. During our research we implemented three functions but ended up using only one of them in the citation API, which we briefly describe. The function *versioning_prefixes* enriches a prologue with two additional prefixes & IRIs that are essential for each read and write function listed in the rdf_star module and moreover in the timestamp_query function.

> PREFIX vers: <https://github.com/GreenfishK/DataCitation/versioning
> PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

The prefix *vers* is used in front of valid_from and valid_until, which we use to version and read versioned data. The prefix *xsd* resolves to the datetime from W3C's XML schema and is again needed for aforementioned versioning predicates but also for the new version timestamp whenever data is written into the RDF* store.

4.2 Build and Distribution

To show our practical work we decided to make a build from the source and distribute it online. What we included in the build is a collection of python modules, which we described in Section 4.1, templates that are used by the rdf_star and query_utils module, a query_store.db and several prepared sql statement files that are used by the query_store module.

To build, publish and install the package we used conda:

Listing 4.23: Build and install RDF* Data Citation API

```
conda build <project folder name> --croot Distribution/build/conda
conda install -c greenfish rdf\_data\_citation
```

The project folder where the source code resides must be passed to the build command. The build is then available in Distribution/build/conda. The build command takes settings from

four files, namely meta.yaml, setup.py, conda_build_config.yaml and MANIFEST.in. These files can be inspected on our Github page [11].

The install command can be executed like it is shown above. However, the build has been only configured for and tested on Linux. For Windows one needs to download the package from `https://anaconda.org/Greenfish/rdf_data_citation` and install it manually. Our project is licensed under the GNU General Public License.

4.3 Summary

In this chapter we showed our practical work, which encompasses a python API that implements all business logic components from the design chapter. Moreover, a query store is implemented and embedded within the project structure. The implementation is in accordance with the RDA Data Citation recommendations and fulfills the product requirements we outlined in Section 3.1.1. We showed our contribution to versioning data on statement-level using RDF* and SPARQL* in 4.1.1 with two different versioning modes - query performance and memory saving. We collected all functions that can be interpreted from requirements R4-R8 and R10 in the persistent_id_utils module. There we also discussed query normalization and how we tackled it using SPARQL's query algebra where we performed normalization measures on the query tree. Furthermore, we explained how we extend queries query with additional joins and filters to make it a timestamped query and persistently identify datasets. To create a dataset description we used a mix of descriptive statistics and one heuristic that is applied on the dataset. As in RDF we do not use primary keys we discussed how we can derive a unique sort index from the dataset and how in general we can make the user aware of it. In Section 3.9 we described how to mint datasets and retrieve PID-minted datasets & metadata and showed example code snippets. Finally, we described the build and installation process and where our work can be accessed and downloaded.

[11] `https://github.com/GreenfishK/DataCitation`

CHAPTER 5

Evaluation

In this chapter we evaluate the performance of the RDF* Data Citation Framework Prototype and also evaluate two methods for detecting semantically equivalent queries.
We start off with listing functional tests in Section 5.1.1 including description and test results for the RDF* Data Citation Framework. The functional tests are structured into four packages. The first two, namely *SPARQL Query Algebra to Query Text Translation* and *Normalization* deal with our implementation of R4 – Query Uniqueness, which is the first method for detecting semantically equivalent queries. The *Versioning* test package revolves around timestamped write operations and queries on live and historical data for two different versioning modes (R1 and R2). The *Query Handler* package finally presents tests for minting RDF datasets and retrieving them via query PID (R1-R10). In Section 5.1.2 we present evaluation scenarios and evaluation metrics that we used to measure the runtime and memory performance of functions from the rdf_star module (R1 and R2). Subsequently, we present the results of this evaluation.
In Section 5.2 we evaluate two Query Containment Solvers, that we already discussed in Section 2.5 of Chapter 2, with regards to detecting semantically equivalent queries. The results from the evaluation of this second method is then compared to the results from the first method. Finally, we conclude what this means for R4 – Query Uniqueness.

5.1 RDF* Data Citation Framework

As we built the RDF* Data Citation Framework API with the intention to publish and offer it to users who are looking for a Dynamic Data Citation Solution for RDF* stores and graph databases we need to among other things provide a documentation of its functionality. Test Cases are a good means of documentation as they show conditions and scenarios under which the solution works and for which it does not. As SPARQL queries are the main input for this API and they can be written in many different ways we decided for most of our functional tests to revolve around queries and show the coverage for different constructs and keywords.

The non-functional tests on the other hand give insights to the performance of this API so that users can consider them when making decisions about hardware or if query executions are time critical.

5.1.1 Functional Tests

The functional tests we provide in this chapter reflect our research coverage and are motivated by the Use Cases that we discussed in Chapter 3 (cf. Figure 3.1). While the first three sections (= test packages) use different queries as test objects to test conversion between query and query algebra tree, normalization and versioning, the fourth section (Query handler) tests scenarios (see alternative scenarios in query handler algorithm 4.1) and erroneous user inputs when minting a dataset. We used the "News Dataset" for all functional tests, which is the default Graph that comes with GraphDB. In each section we provide examples of the executed tests from the corresponding test packages. All test cases can be found on our Github page [1]

SPARQL Query Algebra to Query Text Translation

This test package has the most tests as it covers all keywords from RDF 1.1 within 39 synthetically constructed queries. There is one test case per query and in each test case the corresponding query is translated into its algebraic form and then back translated using the *translate_algebra* function from the *persistent_id_utils* module, which we suggested via pull request to be included in rdflib module [2]. Maintainers of the rdflib module reviewed it and merged it. To put it more formal, we have following pipe of functions:

$$query' = translate\_algebra(translateQuery(query))$$

There are now two ideas how to test the translation. One is to execute $query$ and $query'$ and see if both yield the same result. The other is to apply the same procedure on $query'$ and make sure that the derived $query''$ is equal to $query'$, thus, that this function is idempontent. Our test cases currently only use the latter approach. The tests are divided into following categories:

- **functions**: date functions, numeric functions, string functions, ...
- **graph patterns**: BGP, Join, Union, Group, Having, ...
- **operators**: arithmetic, conditional, relational, ...
- **property paths**: alternative paths, inverse paths, sequence paths, ...
- **solution modifiers**: distinct, project, order by, reduced, ...
- **others**: SERVICE, VALUES

In Listing 5.1 we can see a query from the category *propert path*. In Listing 5.2 we see the output, $query'$ where two things happend: 1. OFFSET 0 was added and the prefixes where resolved, thus, the full resource names are stated. This is why we consider this procedure as

[1] https://github.com/GreenfishK/DataCitation/tree/master/tests
[2] https://github.com/RDFLib/rdflib

"pre-normalization" because some normalization measures like these are already applied. Computing $query''$ now from $query'$ yields the same query string. Further examples can be found on our Github page.

Listing 5.1: Property path test query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
select * where
  {
    ?x foaf:mbox <mailto:alice@example> .
    ?x foaf:knows/foaf:knows/foaf:name ?name .
  }
limit 100
```

Listing 5.2: Output of function translate_algebra(translateQuery(query))

```
SELECT ?x ?name{
    ?x
        <http://xmlns.com/foaf/0.1/mbox>
            <mailto:alice@example>.
    ?x <http://xmlns.com/foaf/0.1/knows>
        /<http://xmlns.com/foaf/0.1/knows>
            /<http://xmlns.com/foaf/0.1/name> ?name.
}OFFSET 0 LIMIT 100
```

Normalization

The test cases for normalization are structured as follows: For each entry in Table 4.1 we define two semantically identical queries. We then normalize both queries using the normalization function from the *persistent_id_utils* module and afterwards compute their checksums. The test passes if the checksums are equal, otherwise it fails. To show an example we exhibited a query pair in Listings 5.3 and 5.4. There we can see that the first query joins triple statements in the usual way while the second query uses a sequence path as a shortcut. The normalized form of both queries can be inspected in Listing 5.5. Note that letters are used as variables and the sequence path is resolved, which are both measures we defined in Table 4.1.

Listing 5.3: Sequence paths - alternative 1

```
PREFIX pub:
<http://ontology.ontotext.com/taxonomy/>
PREFIX publishing:
<http://ontology.ontotext.com/publishing#>

select ?personLabel ?party_label where {
    ?person pub:memberOfPoliticalParty ?party .
    ?party pub:hasValue ?value .
    ?value pub:preferredLabel ?party_label .
    ?person pub:preferredLabel ?personLabel .
    filter(?personLabel = "Barack Obama"@en)
}
```

Listing 5.4: Sequence paths - alternative 2

```
PREFIX pub:
<http://ontology.ontotext.com/taxonomy/>
PREFIX publishing:
<http://ontology.ontotext.com/publishing#>

select ?personLabel ?party_label where {
  ?person pub:memberOfPoliticalParty
      / pub:hasValue
          / pub:preferredLabel ?party_label.
  ?person pub:preferredLabel ?personLabel .
  filter(?personLabel = "Barack Obama"@en)
}
```

Listing 5.5: Sequence paths - normalized query

```
SELECT ?a ?b{
    FILTER(?a = "Barack Obama"@en)
    ?c <http://ontology.ontotext.com/taxonomy/hasValue> ?d.
    ?d <http://ontology.ontotext.com/taxonomy/preferredLabel> ?b.
    ?e <http://ontology.ontotext.com/taxonomy/memberOfPoliticalParty> ?c.
    ?e <http://ontology.ontotext.com/taxonomy/preferredLabel> ?a.
}
```

Our solution recognizes 9/10 query pairs as semantically identical. Later, in section 5.2 we use the same test cases to test the two query containment solvers and finally compare the results with our solution.

Versioning

In this package we wrote tests for read and write operations of the *rdf_star* module. The general idea for write operations is to define beforehand what changes to expect in the triple store. If a new triple is inserted, we expect that two additional nested triples come along. By querying the result before and after the changes we can easily make comparisons and assert our expectations. We have a relatively high number of test cases for update compared to the other functions. This is because update is the most complex operation and has a bigger impact on the existing triples, while insert does not affect existing triples.
The general idea for testing the read operation is to construct queries with different placements of BGPs, as the "versioning extensions" get inserted right into the BGPs and then check whether the timestamped queries return correct results. BGPs can occur multiple times in a query, at the same or different level as other BGPs (e.g. union of two BGPs or subselects).

Test name:	test_insert: two_consecutive_inserts
Test case description:	Make two consecutive inserts and retrieve the dataset as it was before, between and after the inserts. Check that the datasets reflect the right information as of each timestamp.
Expected result:	2
Actual result:	2
Test passed:	True

Test name:	test_update_single: add_new_triple
Test case description:	After a single triple update four additional triples must be added. One of them must be the new triple with the new object value.
Expected result:	1_http://ontology.ontotext.com/resource/tsk8e8v43mrk
Actual result:	1_http://ontology.ontotext.com/resource/tsk8e8v43mrk
Test passed:	True

Test name:	test_outdate: outdate_triples
Test case description:	Test if the number of triples in the RDF store after outdating a set of triples did not change. Moreover, test if the result set after the triples have been outdated is empty.
Expected result:	number of triples in db: 190718; number of rows in dataset after outdate: 0
Actual result:	number of triples in db: 190718; number of rows in dataset after outdate: 0
Test passed:	True

Query Handler

This test package contains tests that on the one hand describe and check different scenarios that can occur while handling a query, e.g. dataset has changed or semantically equivalent query exists already. Our general approach is to generate citation snippets "by hand" that we expect as an outcome and to compare them with the actual citation snippets that are returned by the query handler. On the other hand we are testing different error scenarios that may occur with the sort variables. The tests for now do not cover retrieving data and metadata by a given query PID for now but they can easily be added in the same manner later on. Below we show three example test cases.

Test name:	Empty dataset
Test case description:	Test if an empty dataset can be cited and a citation snippet is returned.
Expected result:	This is an empty dataset. We cannot infer any description from it. ea6af57430c9bfeb6d23cb9c70cbf5722bc3e5d9eba7ff0d44447fed917d879e2021-04-30T12:11:21.941000+02:00, Filip Kovacevic, Obama occurrences as Republican, Filip Kovacevic, Dataset/RDF data
Actual result:	This is an empty dataset. We cannot infer any description from it. ea6af57430c9bfeb6d23cb9c70cbf5722bc3e5d9eba7ff0d44447fed917d879e2021-04-30T12:11:21.941000+02:00, Filip Kovacevic, Obama occurrences as Republican, Filip Kovacevic, Dataset/RDF data
Test passed:	True

Test name:	Changed dataset
Test case description:	Test if a new query PID is created if the dataset has changed since the last execution and the query stayed the same (=same query checksum).
Expected result:	fdb137f830ad12f4641d755ca86c966a01288d80c586418ee7457cff69a81a662021-04-30T12:11:21.941000+02:00, Filip Kovacevic, Obama occurrences, new mention, Filip Kovacevic, Dataset/RDF data fdb137f830ad12f4641d755ca86c966a01288d80c586418ee7457cff69a81a662021-08-13T13:27:58.676957+02:00, Filip Kovacevic, Obama occurrences, new mention, Filip Kovacevic, Dataset/RDF data checksum:fdb137f830ad12f4641d755ca86c966a01288d80c586418ee7457cff69a81a66
Actual result:	fdb137f830ad12f4641d755ca86c966a01288d80c586418ee7457cff69a81a662021-04-30T12:11:21.941000+02:00, Filip Kovacevic, Obama occurrences, new mention, Filip Kovacevic, Dataset/RDF data fdb137f830ad12f4641d755ca86c966a01288d80c586418ee7457cff69a81a662021-08-13T13:27:58.676957+02:00, Filip Kovacevic, Obama occurrences, new mention, Filip Kovacevic, Dataset/RDF data checksum:fdb137f830ad12f4641d755ca86c966a01288d80c586418ee7457cff69a81a66
Test passed:	True

Test name:	Non-unique sort
Test case description:	Test if a query with a non-unique order by clause written by the user throws a NoUniqueSortIndexError exception.
Expected result:	The "order by"-clause in your query does not yield a uniquely sorted dataset. Please provide a primary key or another unique sort index
Actual result:	The "order by"-clause in your query does not yield a uniquely sorted dataset. Please provide a primary key or another unique sort index
Test passed:	True

5.1.2 Non-functional Tests

In this section we evaluate the most time and memory consuming operations of our RDF* Data Citation Framework Prototype, namely retrieving live and historical data from a versioned RDF* store. We are interested in comparing the two different versioning modes which we defined in 4.1.1 and test our hypothesis that in the query performance mode (Q_PERF) the

queries run faster while having more metadata triples loaded into the RDF store and the opposite statement is true for the memory saving mode (MEM_SAVE). This way we want to support decision-making when applying the API for the first time on a non-empty triple store and a versioning mode needs to be chosen. If e.g. a triple store is only rarely updated one would probably rather consider to use the memory saving mode. However, if a triple store is constantly updated, the set of triples without a start date, which is the case in aforementioned mode, would become relatively small over time and overhead due to left joins and additional filters would be generated in this mode. In this case the query performance mode would be considered under the assumption that our hypothesis is true.
In the following subsections we first outline and describe our evaluation setup and process. We thereby explain the parameters we used during the evaluation process and scenarios that we evaluated. Next, we define metrics that we used to measure the different scenarios. Last, we show the results for each scenario, present interesting findings and make conclusions about the performance and hypothesis.

Evaluation setup and process

Our evaluation process is shown in Figure 5.1. Referring to this figure, we define a scenario in our evaluation setup is an instance (parameter values) of the fixed parameters and parameter values in the upper left corner. One scenario can be seen as one non-functional test. For each scenario we incrementally increase the size of the database by adding new triples until 10 increments are reached. The triples that are added are randomly created. The data triples together with the metadata triples make up 30% of the initial number of triples of the corresponding dataset and are inserted either due to a timestamped insert or timestamped update operations. The parameters from the figure are describe as follows:

- **write operation**: A normal insert, timestamped insert or timestamped update. Timestamped means that triples are written to the triple store with a version timestamp, as discussed in Section 4.1.1.
- **dataset**: Can be either small or big. The small one is the FHIR ontology [3] and contains 74190 triples. The big one is a DBPedia dataset [4] and contains 1372410 triples.
- **versioning mode**: Can be the query performance mode (q_perf) or memory saving (mem_sav) as introduced in Section 4.1.1.
- **query**: Can be either a simple query or a complex query (see Listings 5.6 and 5.7). We used the same queries for both datasets to make the results more comparable. This is possible as the RDF schema triples are inferred by the inference engine for both datasets. The simple query contains the following elements: BGP, Projection, Filter, String Function.
  The complex query contains the following elements: BGP, Projection, Filter, String function, Order By, Join, Union, Aggregation.

[3] https://www.hl7.org/fhir/downloads.html
[4] http://downloads.dbpedia.org/2016-10/core-i18n/en/category_labels_wkd_uris_en.ttl.bz2

- **procedure to evaluate**: We defined two procedures "retrieve live data" and "retrieve history data" based on the get_data function from the python API where we provide a timestamp as of initial database state for the latter and the system timestamp for the former.

  - retrieve_live_data(query): Retrieves the life data by using get_data from the rdf_star module with the current system timestamp. The result set grows bigger upon every increment simulating a dynamic dataset.
  - retrieve_history_data(query, current_timestamp): Retrieves data with a timestamp that is created at the time of initial database state. The dataset stays unchanged, even if new triples are added.

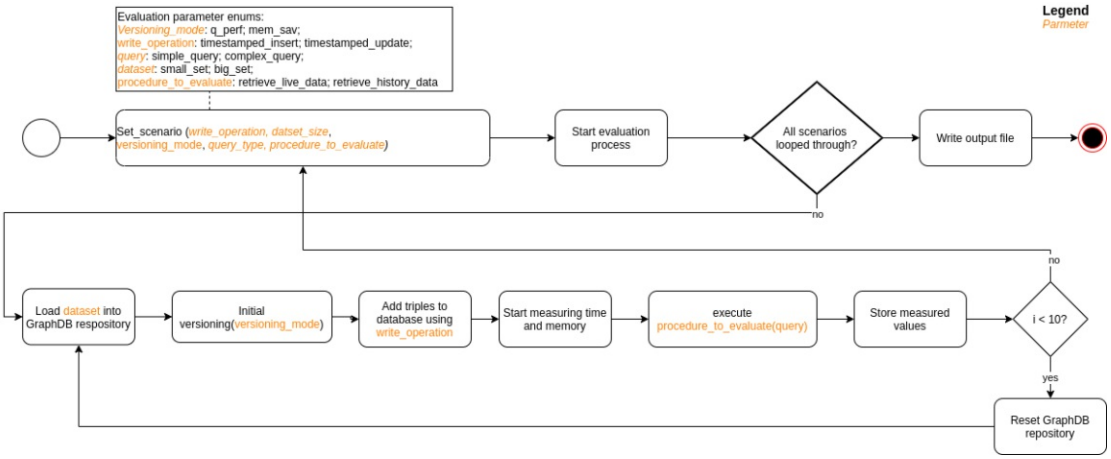


Figure 5.1: Evaluation process of the rdf_star module

Thus, we only evaluate the rdf_star_module. Further possible procedures we could evaluate would be minting a new PID and retrieving data and metadata from the query_handler module. However, these again include querying live and historical data which are the most time consuming operations. Querying metadata from the query store might be time consuming, too. However, we currently have no data about the typical size of a query store in practice, which is why we leave this evaluation for the future.

Listing 5.6: Simple query

```
select ?s ?p ?o
where {
        ?s ?p ?o .
    filter(?p = rdfs:label
        && strends(?o, "new_value"))
}
```

Listing 5.7: Complex query

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

select
?s ?p (count(?o) as ?cnt_new_values)
?y ?z ?m (count (?x) as ?cnt_x)
where {
    {
        ?s ?p ?o .
    filter(?p = rdfs:label && strends(?o, "new_value"))
    }
    union
    {
       ?x ?y ?z.
       ?x rdf:type ?m.

    }
} group by ?s ?p ?y ?z ?m
order by ?s ?p
```

We created the dataset to be added on every increment by using an insert statement combined with a SPARQL query. In Listing 5.8 we show the versioned insert statement we used for the FHIR dataset.
As a triple store we use GraphDB-free 9.5 [5], which runs on Java 11, for all evaluation scenarios. To import a dataset one first needs to create a repository, which we did separately for each dataset. We used the default settings. The most important ones are shown in Table 5.1.

Listing 5.8: Versioned random data that gets inserted on query increment

```
PREFIX rdfs:
<http://www.w3.org/2000/01/rdf-schema#>
PREFIX dc:
<http://purl.org/dc/elements/1.1/>
PREFIX xsd:
<http://www.w3.org/2001/XMLSchema#>
PREFIX vers:
<https://github.com/GreenfishK/DataCitation/
versioning/>

insert {
    ?s ?p ?new_label.
    <<?s ?p ?new_label>> vers:valid_from ?newVersion.
    <<?s ?p ?new_label>> vers:valid_until "9999-12-31T00:00:00.000+02:00"^^xsd:dateTime.
} where
{
    {select  ?s ?p  (concat(str(rand()), "_new_value") as ?new_label)
        where {
            ?s ?p ?o .
            filter (?p = rdfs:label
                || ?p = dc:title)
        }
        limit 7418
    }
    BIND(xsd:dateTime(NOW())
```

[5] https://www.ontotext.com/products/graphdb/graphdb-free/

```
        AS ?newVersion).
}
```

Table 5.1: GraphDB settings

Parameter	Value
Type	GRAPHDB-FREE
Ruleset	RDFS-Plus (Optimized)
Supports SHACL validation	false
Base URL	http://example.org/owlim#
Entity index size	10000000
Use predicate indices	true
Cache literal language tags	true
Use context index	false
Enable literal index	true
Check for inconsistencies	false
Throw exception on query time-out	false
Read-only	false
Entity ID bit-size	32

All the scripts and prepared statements we used for evaluation are available in a separate project on Github [6]

Evaluation Metrics

During our evaluation process we measure execution time and peak memory consumption of the isolated procedures (E.g. retrieve_live_data) for each scenario and during each increment. The code snippet in Listing 5.9 shows the employment of modules time and tracemalloc. We started recording before the execution of these procedures, which we see in the first code block. In the second code block the function is picked based on the passed "procedure_to_evaluate" parameter and the current parameter set (scenario) is passed to the function. In the third code block we stop the recording and save the results in simple variables, which are later on, together with the parameters, added to a dataframe and plotted (see results).

Listing 5.9: Random data that gets inserted on query increment

```
time_start = time.perf_counter()
tracemalloc.start()

func = procs[procedure_to_evaluate][0]
func_params = procs[procedure_to_evaluate][1]
func(*func_params)

time_elapsed = (time.perf_counter() - time_start)
memMB = tracemalloc.get_traced_memory()[1] / 1024.0 / 1024.0
tracemalloc.stop()
```

[6] https://github.com/GreenfishK/DataCitation_Evaluation

Table 5.2: Hardware specifications

Architecture:	x86_64
CPU(s):	8
CPU Model name:	Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
RAM:	2x 4096 MB; DDR3
SSD Model Number:	CT240BX500SSD1
SSD Interace:	SATA 6Gb/s
SSD Read:	540MB/s
SSD Write:	500MB/s

Hardware specifications

We executed the non-functional tests on our local machine which has hardware specifications as shown in Table 5.2. This list is not exhaustive and we only provide the specifications which we believe have the most impact on the performance. Also, we compare the scenarios relatively to each other and do not comment on specific performance figures/numbers.

Results

We first outline general characteristics of the small and big dataset with regards to runtime performance. Then we discuss peculiar observations and refer to figures for visual comparison. Afterwards we discuss the memory consumption. For each dataset we have eight figures. All figures have the same parameters and chart elements. The parameters that are present in the figures for each dataset are: procedure_to_evaluate, query (type) and write operation. In each figure we show the number of triples for both versioning modes as barplots and the runtime and memory consumption as lines. We additionally show the number of triples in the result set as grey barplots. They can be seen as a "there of"-position. The runtime performance is shown in the upper subplot and the memory consumption in the lower.
Runtime performance: For the small dataset and timestamped inserts we can spot a significantly higher runtime performance in the Q_PERF mode and increasing deviation between the two versioning modes when querying historical data. Querying live data, on the other hand, has only negligible performance differences. (see Figures 5.2, 5.3, 5.4 and 5.5). For the small dataset and timestamped updates the same is true for querying historical data as for timestamped inserts (see Figures5.6 and 5.8).
When only timestamped updates are done, live data queries in the MEM_SAVE mode perform better for the simple query when passing 100k triples and the runtime performance equalizes at the 400k triples mark for the complex query (see Figures 5.9 and 5.7). The observation about the better runtime performance and lower storage needs in the MEM_SAVE mode for live data queries when only timestamped updates are present is also made for the big dataset (see Figures 5.17 and 5.15). In latter Figure we can also see how complex live data queries in the MEM_SAVE mode surpass the Q_PERF mode after the 400k triples mark.
One obvious observation is the peak in runtime performance (=low runtime performance) in Figure 5.5. This happened during the very first query and we have one possible explanation for

this phenomenon. GraphDB had no optimized index structure before the very first execution and only then reworked the indices. While we cannot be sure whether this is true we only see it occurring once and therefore consider it as irrelevant for the overall picture of the runtime performance.

Memory consumption: The behavior of the memory consumption is mostly explained by the size of the result set (grey bars). The more triples we have in our result set the higher the memory consumption. We can see that the number of triples in the result set stays the same upon every increment when querying historical data, which is an expected behavior. For live data queries where the result set changes upon every increment we also see a linear increase in memory consumption. In some plots, such as in Figure 5.8, the lines look fuzzy, which is only due to the higher zoom on the memory consumption scale (right y-axis). We can conclude that memory consumption does not significantly differ between the Q_PERF and MEM_SAVE mode, which makes sense, as the modes do not affect the result sets.

Conclusion: There is no general answer to which versioning mode should be chosen as it depends on many factors, such as the frequency of inserts and updates, the result set size and whether it is affected by the dataset changes and the frequency of live and history data queries. If no estimations about these factors can be made, the MEM_SAVE mode should be preferred as it has clear benefits when only timestamped updates are present. However, if one can estimate the read and write frequencies, evolution of datasets & result sets and storage capacities our evaluation results can make it easy to decide about the versioning mode. More importantly, our evaluation results give an impression about how queries perform against versioned datasets inside an RDF* store. The results suggest that datasets, that have been enriched over time by insert statements, have a different impact on query performance than datasets where update statements were used, even though the number of inserted triples is the same for both operations. Further variables that influence the runtime performance are the dataset & result set size and the way joins and filters are used in the timestamped query (compare Q_PERF vs MEM_SAVE).

5.2 Detecting semantically equivalent queries

For this part of evaluation we take the queries as in Section 4.1.3 as input for the two Query Containment Solvers. These are also the queries we used to test the normalization function. We need to check the reciprocal containment of these queries in order to verify their equivalency. However, if for any reason the Query Containment Solvers report that one query is not contained in the other, the test fails immediately and we do not need to check the other direction. As for the metric, we use a simple metric which is either true if the a query pair gets recognized as semantically identical, false if it does not and "not compilable" if the solver cannot compile the query, e.g. because specific syntax is not supported. In the following two sections we use the test *rdf_type_predicate*, which conforms to the second row in Table 4.1, as example to show the different inputs and outputs. The remaining test cases are available in two, for each solver separately defined, Github repositories [7] [8]. For both solvers and for the

[7] https://github.com/GreenfishK/SpeCS_Evaluation
[8] https://github.com/GreenfishK/JSAC_Evaluation

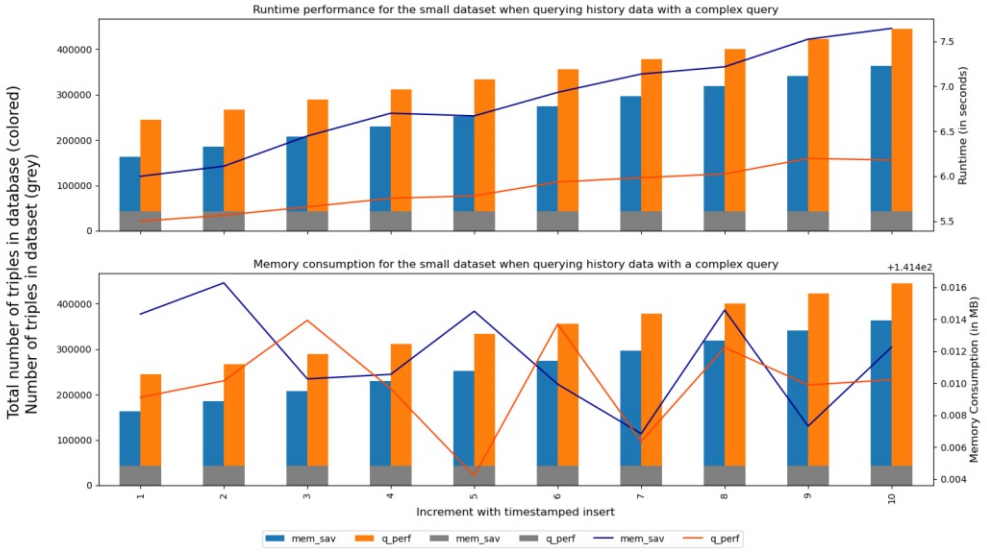


Figure 5.2: Measuring runtime and memory consumption of retrieving historical data with the complex query against the small dataset which was incremented with timestamped inserts in both versioning modes

query algebra approach from our Data Citation API we show and explain the results in Section 5.2.3.

5.2.1 SpeCS

To evaluate the SpeCS solver we executed six steps which we describe in the following. First, first downloaded it from `www.math.rs/~mirko/SpeCS.tar.gz`. Unfortunately, we discovered that the download package is not available anymore via this link. Second, we extracted the source files and built the application with the provided make files. Next, we created the input files for the solver where we placed the queries in a specific syntax. We took the same queries as for the algebra solver in our Data Citation API and for each query pair we created a pair of input files. In one file we state one query as superquery and the other as subquery and in the other file we switch the roles. In Listings 5.10 and 5.11 shows one test case as example of such pair of input files for the solver. Finally, we run the solver via CLI where we pass an input file that lies in a sub-directory with the following command:

```
./specs -file test_queries/<input_file.txt>
```

Some of the original test queries where not recognized by the solver, which is why we made following modifications to the queries:

1. The solver needs a where keyword in the query. Include the where-clause in every test query except for the *optional_where_clause* test.

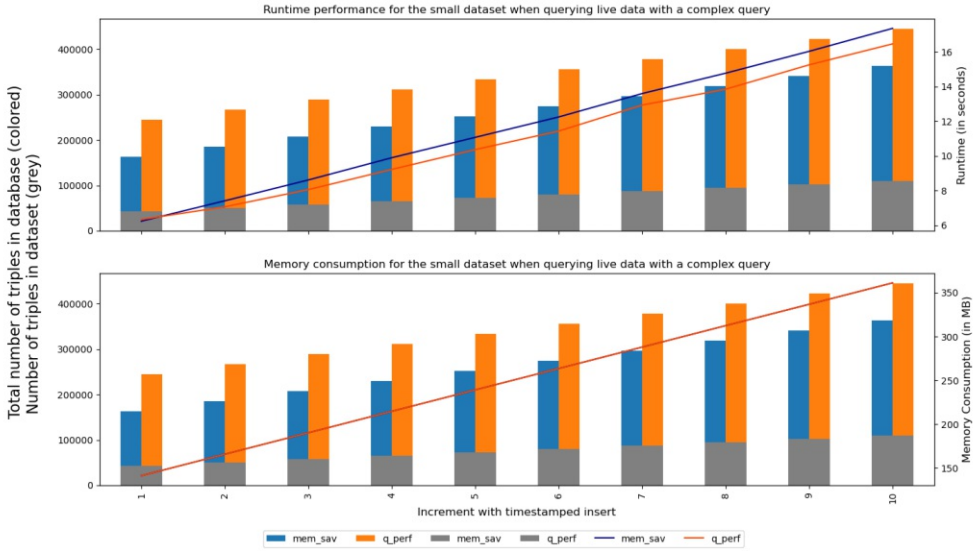


Figure 5.3: Measuring runtime and memory consumption of retrieving live data with the complex query against the small dataset which was incremented with timestamped inserts in both versioning modes

2. Remove language suffix @en such as is in "Obama"@en.

These modifications do not affect the test cases in their scenarios. Nevertheless, we can tell that the solver is not SPARQL 1.1 compliant as such modifications would not be needed otherwise.

Listing 5.10: SpeCS input file pair example - first file

```
Superquery:
PREFIX rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?s ?o where {
        ?s rdf:type ?o .
} order by ?s

Subquery:
select ?s ?o where {
        ?s a ?o .
} order by ?s
```

Listing 5.11: SpeCS input file pair example - second file

```
Superquery:
select ?s ?o where {
        ?s a ?o .
} order by ?s

Subquery:
PREFIX rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?s ?o where {
        ?s rdf:type ?o .
} order by ?s
```

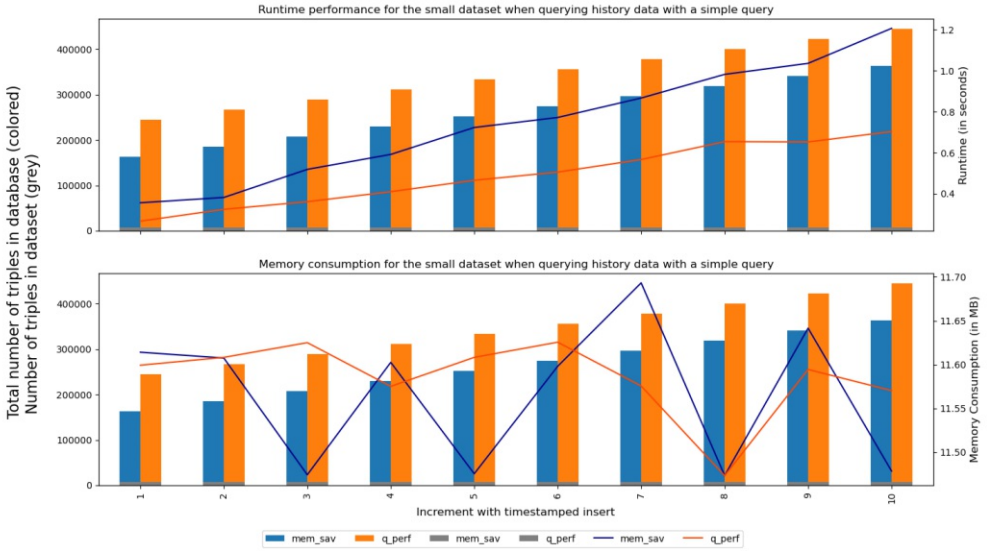


Figure 5.4: Measuring runtime and memory consumption of retrieving historical data with the simple query against the small dataset which was incremented with timestamped inserts in both versioning modes

5.2.2 JSAC

The evaluation on JSAC was performed using their very own test suite where one can create unit tests to check the containment of two queries. The project is available on Github [9] and is implemented in Java. We first forked the project into our own Github page and then wrote a test case for each query. A test case loads a pair of files, where the semantically equivalent queries are placed, into a string and checks the containment using the a function which takes two queries as input. Similar to the evaluation of SpeCS, this function is called twice where the queries are switched in the second call. Listing 5.12 shows an example with the same test case we showed for SpeCS.

Listing 5.12: SpeCS input file pair example - first file

```
@Test
public void testRDFTypePredicate() throws IOException {
    String vStr = Files.readString(Path.of(url_test_dir +
    "/test_normalization__rdf_type_predicate_alt1.txt"));
    String qStr = Files.readString(Path.of(url_test_dir +
    "/test_normalization__rdf_type_predicate_alt2.txt"));
    printOutQueryContainments(vStr, qStr);
    printOutQueryContainments(qStr, vStr);
}
```

[9] https://github.com/SmartDataAnalytics/jena-sparql-api/tree/develop/jena-sparql-api-query-containment

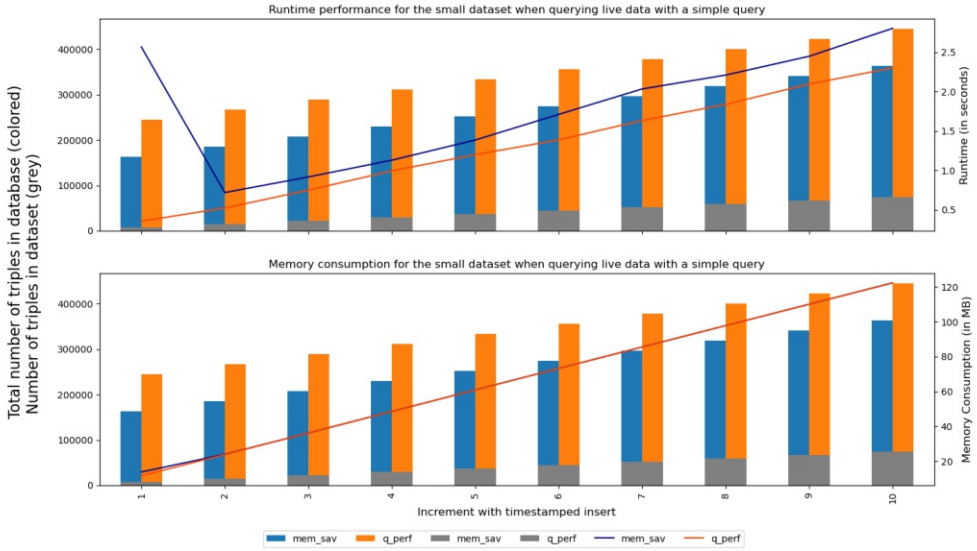


Figure 5.5: Measuring runtime and memory consumption of retrieving live data with the simple query against the small dataset which was incremented with timestamped inserts in both versioning modes

Again, we needed to modify some queries so that the test case does not fail because semantics that we are not testing. We also found the lack of support for following query semantics:

1. The solver does not support order-by clauses and throws a syntax error if there is one in the query. We removed all order-by clauses from the test queries. They were initially provided due to the R5 recommendation but then dropped for evaluationg JSAC and SpeCS as the order-by clause is not part of any of these tests (It is, however, part of some functional test).
2. Aliases are neither supported using BIND in the query body nor using AS in the select statement. Unfortunately, we have one test case where aliases are tested. Therefore, we could not remove it from this one as the test would miss its purpose.
3. (Inverted) sequence paths are not recognized.

JSAC also employs algebra trees, which are dissimilar to rdflib's implemenation of the W3C SPARQL query algebra, to solve the containment problem and prints it for each executed test case. In Listing 5.13 we show the algebra for the same super- and sub-query in Listing 5.10, which turns out to be the same. We can e.g. see next to *?v_2* that the predicates rdfs:type and "a" are resolved the same way, which is what we expect for that test case.

Listing 5.13: Normalized index entry algebra expression for the test case *rdf_type_predicate*

```
Normalized index entry algebra expression:
(OpDistinctExtendFilter (filter true
    (OpExtConjunctiveQuery ConjunctiveQuery
        [projection=VarInfo [projectVars=[?s, ?o],
```

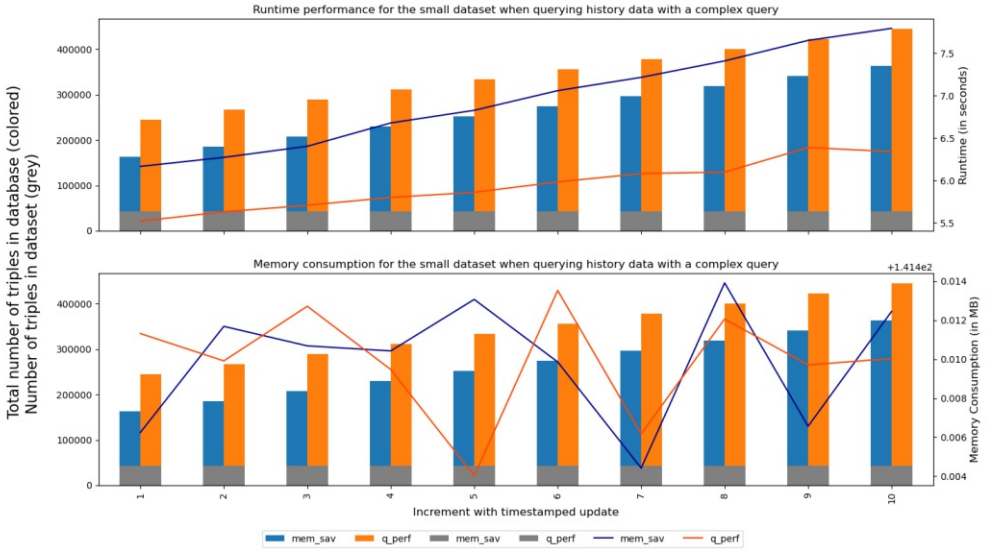


Figure 5.6: Measuring runtime and memory consumption of retrieving historical data with the complex query against the small dataset which was incremented with timestamped updates in both versioning modes

```
        distinctLevel=0],
        qfpc=<quads=[[?v_1 ?s ?v_2 ?o]],
        filterDnf=[[(= ?v_2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>)]]>]
    )
  )
)
```

5.2.3 Results

Table 5.3 presents the result for each solver and 10 test cases. We can see that SpeCS managed to recognize only one query pair as semantically identical. Moreover, some semantics are not known to this solver, like sequence paths and the optional where clause. JSAC managed to solve 6/10 query pairs whereas only one actually failed and three of them threw an Exception as they could not be compiled, which is the same issue as SpeCS had. Our solution from the RDF* Data Citation API passed 9/10 tests. None of the solutions could solve *variables_not_bound* and the reason for this is probably that the two equivalent expressions are too heterogeneous and hard to tell that they are equal. The SPARQL algebra tackles this problem by matching one of the two expressions and transforming it into the other (also see Table 4.1). The reason for the test to still fail is bug in the normalization procedure which we will manage to resolve in near future.

The conclusion we can draw for R4 - Query Uniqueness is that evidently JSAC would be an alternative as it uses a similar approach to check for query containment as we do. A possibility

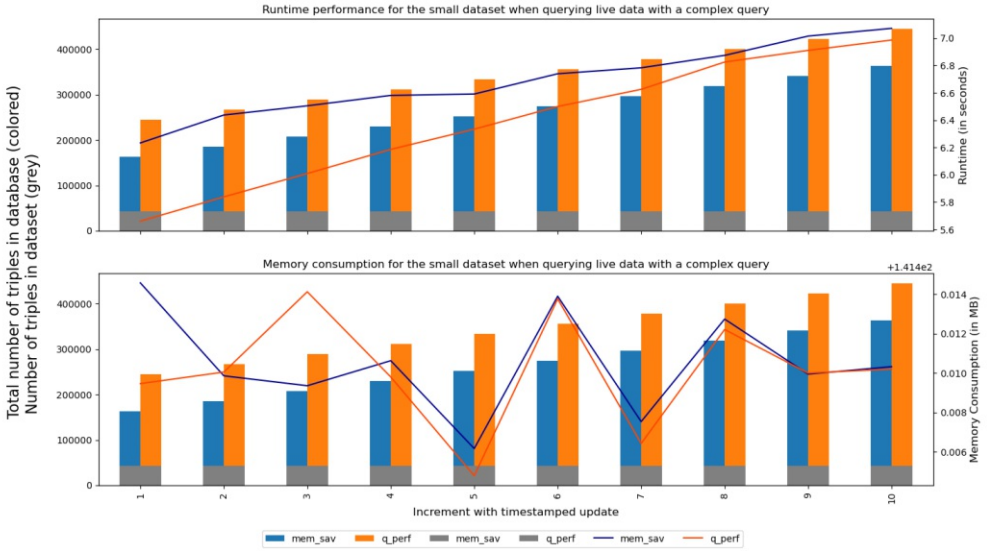


Figure 5.7: Measuring runtime and memory consumption of retrieving live data with the complex query against the small dataset which was incremented with timestamped updates in both versioning modes

Table 5.3: Detecting semantically identical queries - test results

Testcase	SpeCS	JSAC	SPARQL Algebra
optional_where_clause	⚠	✓	✓
rdf_type_predicate	✓	✓	✓
leave_out_subject_in_triple_statements	✗	✓	✓
order_of_triple_statements	✗	✓	✓
alias_via_bind	✗	⚠	✓
variable_names	✗	✓	✓
variables_not_bound	✗	✗	✗
inverted_paths	⚠	⚠	✓
sequence_paths	⚠	⚠	✓
prefix_alias	✗	✓	✓

for the future would be to employ this Java module in our python framework to compute normalized query algebras under the premise that it gets fully SPARQL 1.1 compliant. The benefit would also be that JSAC is already an established module and part of the jena-sparql-api project Github where it has an active community.

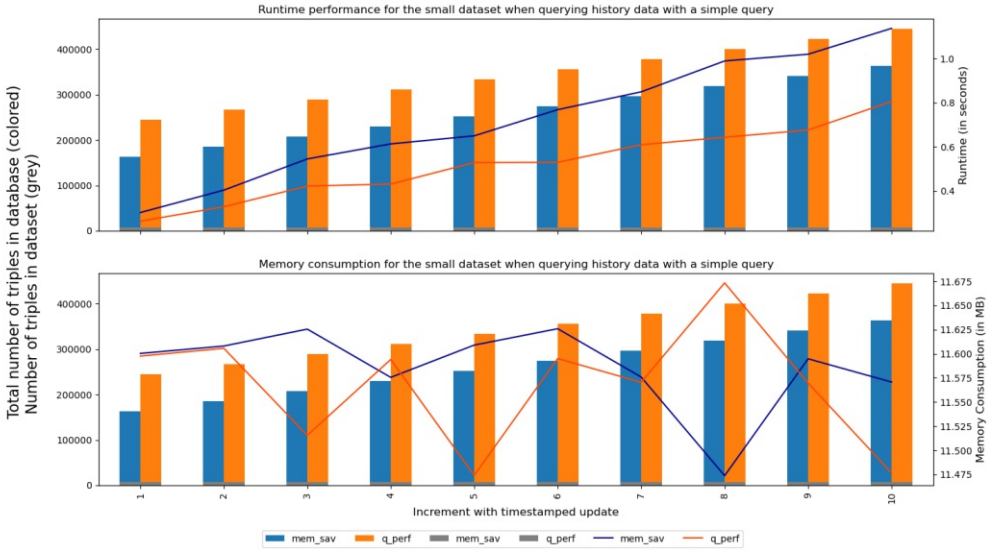


Figure 5.8: Measuring runtime and memory consumption of retrieving historical data with the simple query against the small dataset which was incremented with timestamped updates in both versioning modes

5.3 Summary

In this Chapter we evaluated the RDF* Data Citation Prototype, which is written in python, by means of functional and non-functional tests. We structured the functional tests into four test packages, namely, SPARQL Query Algebra to Query Text Translation, Normalization, Versioning and Query Handler and discussed them in Section 5.1.1. We provided examples for each test package and a Github link to the full test documentation. The test cases cover recommendations R1-R10 and R12 and thereby ensured the functioning of our citation-enabling Prototype. In Section 5.1.2 we evaluated runtime performance and memory consumption of a simple and a complex query that retrieves live and historical data from a small and a big versioned RDF* dataset, that we stored in GraphDB. We focused on comparing the two versioning modes Q_PERF and MEM_SAVE and gave an impression how these queries behave in such a versioned environment by visually analyzing different plots. We came to the conclusion that queries in the Q_PERF mode do not perform better for scenarios where timestamped updates are used and therefore triples should initially just be annotated with an end date (=MEM_SAVE), if there are no estimations about expected read & write frequencies and the evolution of result sets.
In Section 5.2 we evaluated two Query Containment Solvers by checking how many semantically equivalent queries based on the expressions in Table 4.1 can be detected. The test queries we used for the solvers we also used earlier in the *Normalization* test package to test our solution of R4 - Query Uniqueness. Finally, we compared these three solutions and

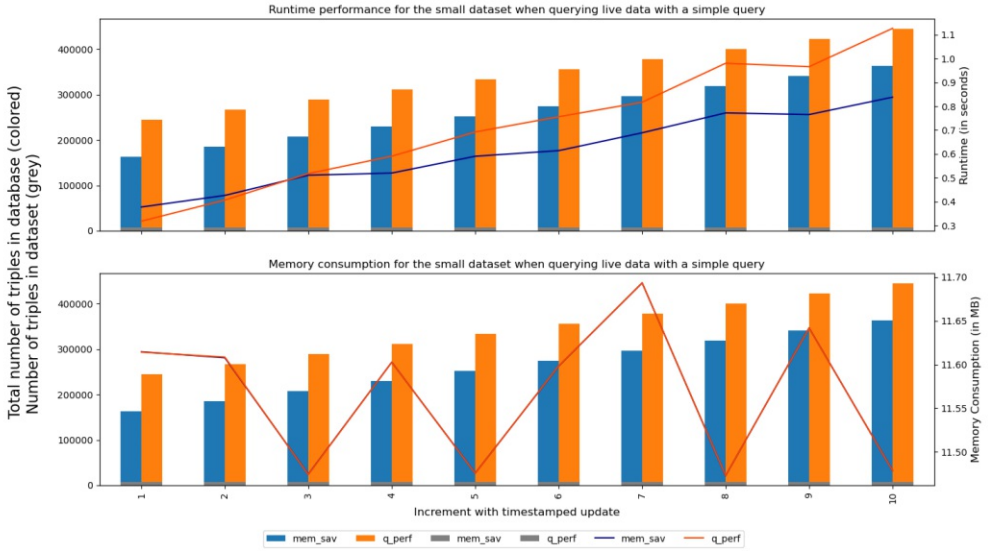


Figure 5.9: Measuring runtime and memory consumption of retrieving live data with the simple query against the small dataset which was incremented with timestamped updates in both versioning modes

concluded that our solution has the most coverage. However, we considered the JSAC Query Containment Solver for the future as it has a well-established project on Github, test suite and active community.

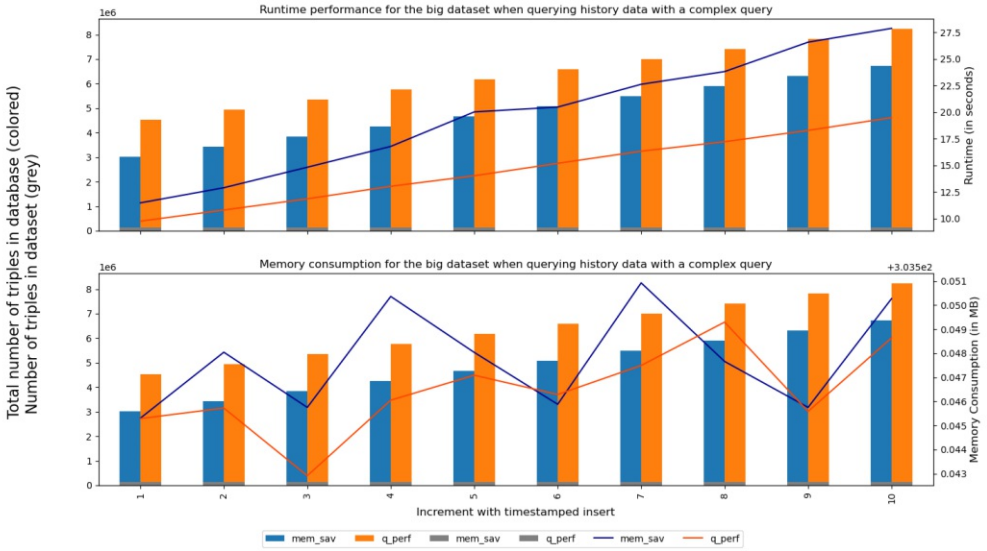


Figure 5.10: Measuring runtime and memory consumption of retrieving historical data with the complex query against the big dataset which was incremented with timestamped inserts in both versioning modes

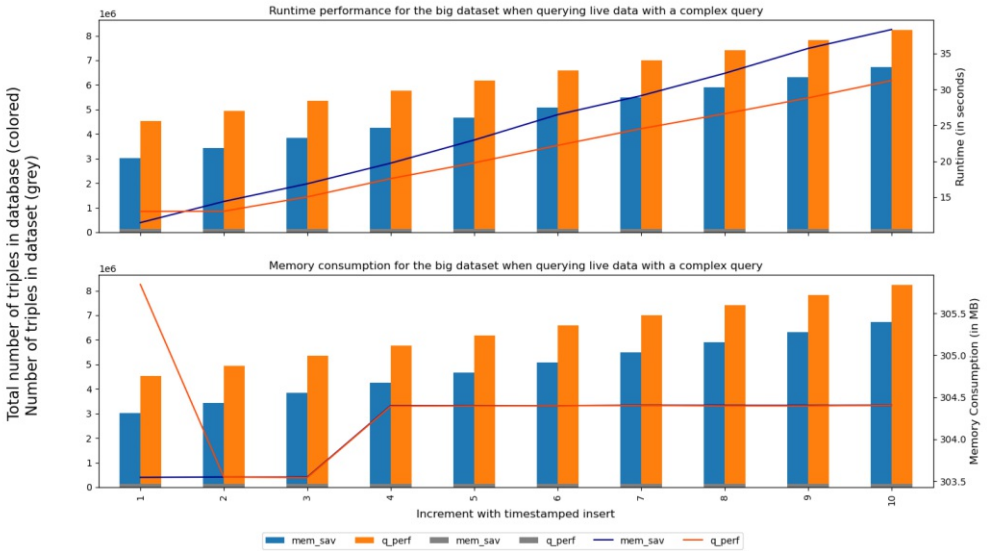


Figure 5.11: Measuring runtime and memory consumption of retrieving live data with the complex query against the big dataset which was incremented with timestamped inserts in both versioning modes

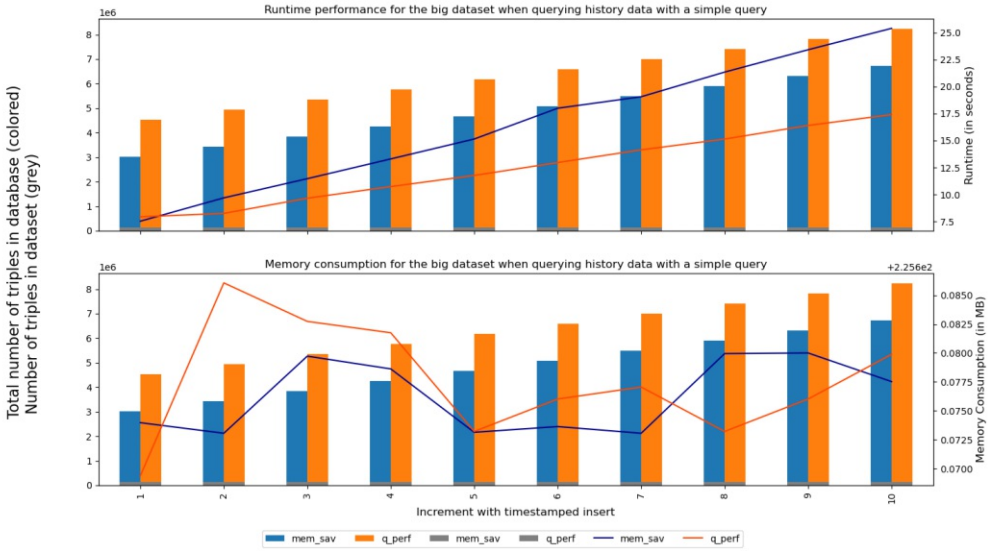


Figure 5.12: Measuring runtime and memory consumption of retrieving historical data with the simple query against the big dataset which was incremented with timestamped inserts in both versioning modes

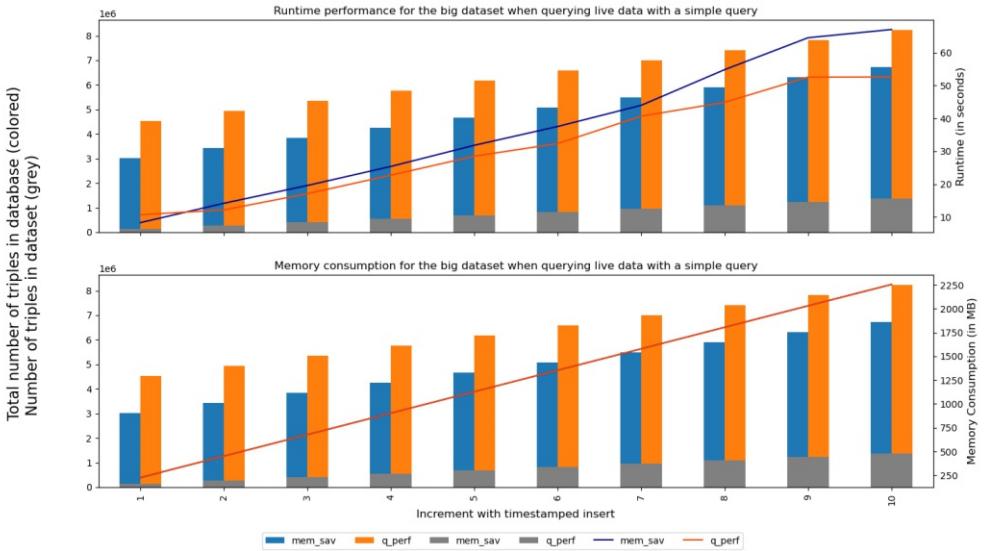


Figure 5.13: Measuring runtime and memory consumption of retrieving live data with the simple query against the big dataset which was incremented with timestamped inserts in both versioning modes

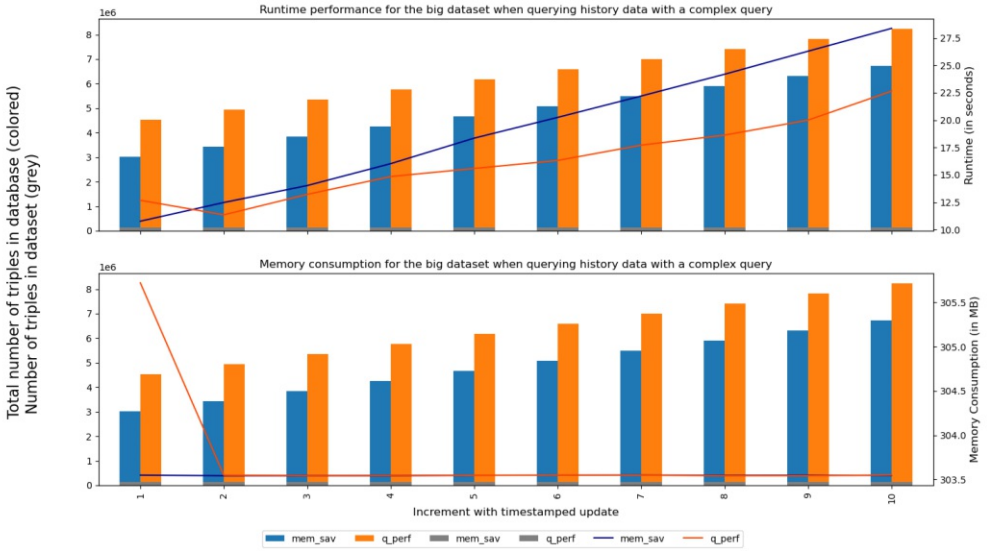


Figure 5.14: Measuring runtime and memory consumption of retrieving historical data with the complex query against the big dataset which was incremented with timestamped updates in both versioning modes

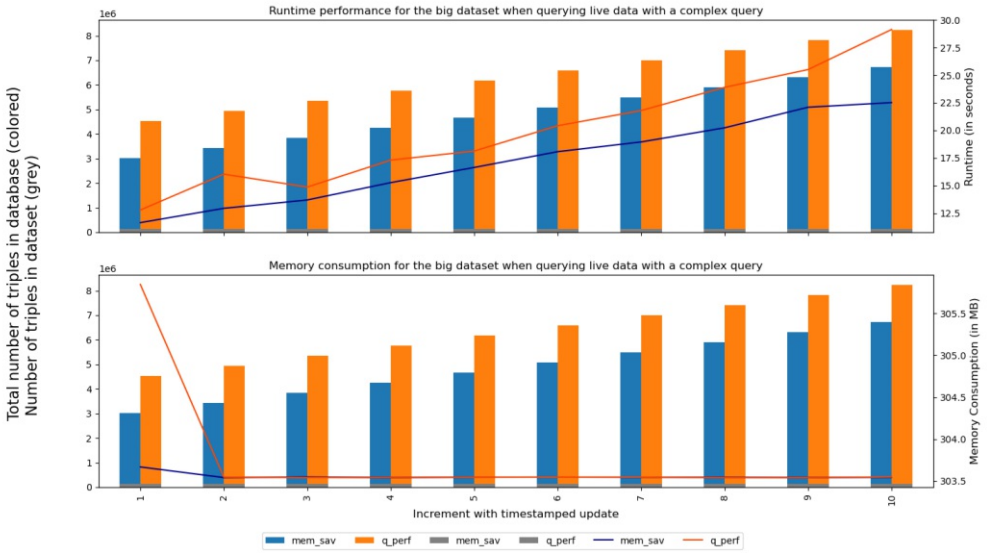


Figure 5.15: Measuring runtime and memory consumption of retrieving live data with the complex query against the big dataset which was incremented with timestamped updates in both versioning modes

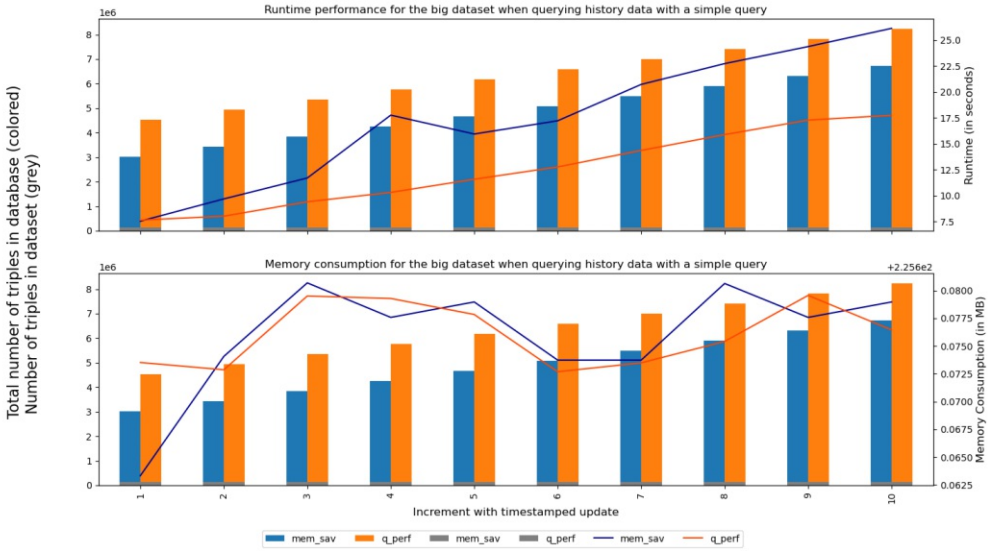


Figure 5.16: Measuring runtime and memory consumption of retrieving historical data with the simple query against the big dataset which was incremented with timestamped updates in both versioning modes

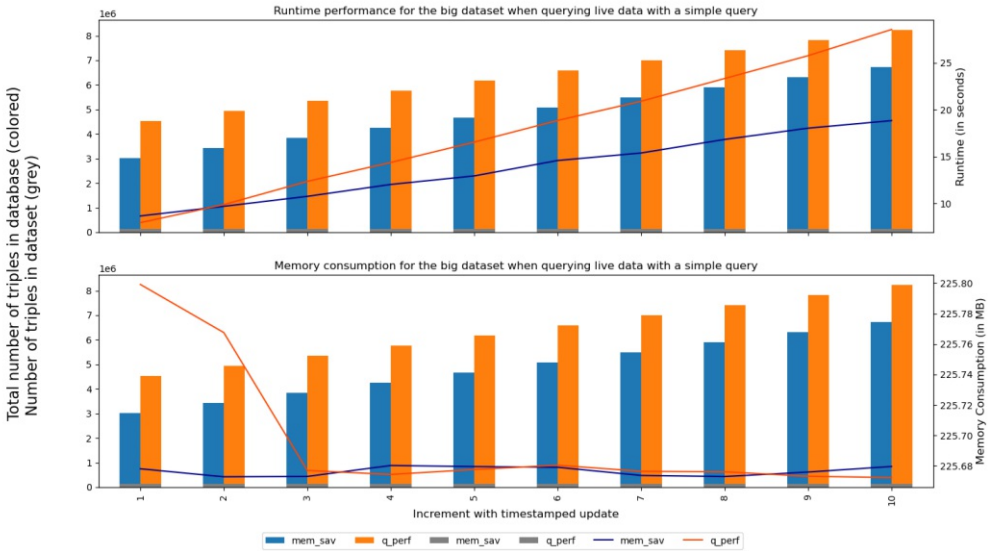


Figure 5.17: Measuring runtime and memory consumption of retrieving live data with the simple query against the big dataset which was incremented with timestamped updates in both versioning modes

CHAPTER 6

Conclusion and Future Work

In this final chapter we first conclude our work by revisiting the research questions from Section 1.3 of Chapter 1 and by summarizing our findings. We thereby refer to single chapters and sections where the research questions have been addressed. Next we lay out future work which for one part are new ideas for future contributions and for the other part possible improvements.

6.1 Conclusion

All our research questions where accompanied by the RDA Data Citation recommendations, which we introduced in Section 1.3 of Chapter 1. RQ1 and RQ2 addressed recommendations R1 & R2 and R4 more thoroughly by discussing and applying RDF* & SPARQL* and the W3C's SPARQL query algebra within a proposed RDF* Data Citation Framework and API. Later Framework and API are the results motivated by RQ3 which also subsume conceptual and practical results from RQ1 and RQ2.

RQ1

What is the best way to use SPARQL* and RDF* to implement *Data Versioning* and *Operation Timestamping* with the aim to keep the number of additionally required triples low?
We defined the predicates *vers:valid_from* and *vers:valid_until* in Section 3.2.1 of Chapter 3 to connect triple and timestamp and thereby enabling versioning by version timestamps. In Section 4.1.1 of Chapter 4 we illustrated templates of RDF* write operations, which employ these two predicates, and showed examples how they can be used to update an RDF* triple store. The number of additional triples during an insert or update is exactly two and three in total. To further lower the number of additionally required triples on database level we introduced the MEM_SAVE versioning mode in Section 4.1.1 of Chapter 4, which initializes a dataset by only attaching one additional (metadata) triple that carries an end date that lies far

in the future. We showed that this initial version of the dataset can be persistently identified, even if the start date is missing and further versioned triples are added to the triple store, by designing versioning extensions (=snippets) that are injected into a query and transforming it into a timestamped query (see Section 4.1.3 of Chapter 4). In a late stage of our research we realized that the number of additionally required triples can even be further lowered, which we will discuss in Section 6.2 of this Chapter.

RQ2

Which of the methods for detecting semantically equivalent queries yields the highest coverage?
One method of detecting semantically equivalent queries is to normalize the queries first, then compute a checksum and finally compare their checksums, as the R4 recommendation suggests. We approached this by first translating the query into an algebra tree, then operated on the tree nodes to normalize the tree and finally translated the tree back into a query to compute its checksum. We used python's rdflib package to translate the query into the W3C's SPARQL query algebra and implemented a translator that does the reverse job. We discussed this approach and showed an example in Section 4.1.3 of Chapter 4. Another method is to use Query Containment Solvers and to check whether two queries are contained in each other. The methods which the Query Containment Solvers use vary. We outlined and explained the solvers available at the time of writing in Section 2.5 of Chapter 2. In Section 5.2 of Chapter 5 we evaluated the solvers by using test queries that we created based on the semantically equivalent expressions in Table 4.1.3. We did the same evaluation for our implementation of the R4 recommendation as part of functional tests (see Section 5.1.1 of Chapter 5) and compared the results. While our solution yields the highest coverage (see Table 5.3) we acknowledged the work of the JSAC solver and considered to use it in the future, if it becomes fully SPARQL 1.1 compliant.

RQ3

Which of the *recommendations* can be covered by the framework and which ones remain specific to the target system?
With the proposed RDF* Data Citation Framework we designed in Chapter 3 we covered R1-R12 of the Data Citation recommendations. We implemented versioning and timestamping (R1 & R2) with the use of SPARQL* and RDF* (see Section 4.1.1 and 4.1.3 of Chapter 4). We designed a normalized query store in Section 3.2.3 of Chapter 3. We designed functions for R4-R8 & R10 in Section 4.1.3 of Chapter 3. The algorithm in Section 4.1.4 of Chapter 4 makes use of these functions and handles the query to return a new or existing citation snippet. In former case the query is stored into the query store (R9). In Section 3.2.7 of Chapter 3 we discussed the landing page and alternative methods of providing data and metadata. However, we argued and concluded that the landing page is the most efficient way to present data and metadata. The landing page (R11) is only covered in our framework design as a component but with no specific structure or design for data and metadata to be represented. We also did not include it in the implementation. To foster machine actionability (R12) we designed a

metadata interface which uses the obligatory attributes of DataCite's Metadata Schema as basis but leaves also room for other metadata that can be added. These metadata are encoded as JSON and stored in the query store (see Section 4.1.3 of Chapter 4) and can, together with the result set be retrieved by means of function *retrieve* from the Query Handler module (see Section 3.2.5 of Chapter 4).
We discussed recommendations R13 & and R14 in Section 3.1.1 of Chapter 3 and thereby explained what they could possibly mean in the domain of RDF and SPARQL. However, we did not include them in our framework design.

6.2 Future Work

Optimization of versioning and timestamping: We chose SPARQL* and RDF* as means for versioning and timestamping with the aim to increase to lower the storage consumption as metadata can be added in a more compact way than other prominent approaches that we introduced in Section 2.3 of Chapter 2. In our design three triples are required for the insert and update operations to represent data and metadata. However, this number can even be further decreased. The simplest and fastest way is to delete the data triple and thereby create a redundant-free graph. We provided a discussion about redundant and redundant-free graphs in Section 2.4 of Chapter 2. This way we need only two triples but there is still room for improvement. Consider the examples in Listings 6.1 and 6.2. The first examples shows a redundant-free graph where we do not have the data triple additionally like in our current design. The second example reduces redundancy even further by creating a hierarchy or having a data triple inside a meta triple inside a metameta triple. Our future work leads this way and we will further thrive for optimization of versioning and timestamping.
Evaluation of other triple stores: In our work we have only evaluated GraphDB. However, we were successful with connecting our Prototype to Apache Jena and Stardog. Yet, we do not know how these perform. By evaluating these in the same manner as GraphDB we might be able to compare the results and support decision-making when deciding about the triple store technology.
Deriving dataset descriptions: In our Prototype we used a heuristic and descriptive statistics to derive a dataset description from the dataset itself (see Section 4.1.3 of Chapter 4). The query provides further information about the dataset. These information could e.g. be extracted from the query's algebra tree, which is what we want to explore in near future.

Listing 6.1: Data and metadata representations - alternative approach 1

```
<<:Obama :occupation :president>>  :from "2009-01-20T00:00:00.000+02:00"^^xsd:date .
<<:Obama :occupation :president>>  :until "2017-01-20T00:00:00.000+02:00"^^xsd:date .
```

Listing 6.2: Data and metadata representations - alternative approach 2

```
<<<<:Obama :occupation :president>>
    :from "2009-01-20T00:00:00.000+02:00"^^xsd:date>>
        :until "2017-01-20T00:00:00.000+02:00"^^xsd:date .
```

List of Figures

1.1 The Design Science Research Cycles [Hev07] 4

2.1 The three-step methodology for citing LOD subsets 12

3.1 Data Citation Framework Use Case Diagram 20
3.2 RDF Data Citation Framework and its components 29
3.3 RDF* Store Utilities to enable communication between triple store and user. . . 31
3.4 RDF Data Persistent Identification Utilities to be used in the query builder algorithm. 32
3.5 Query Store Utilities 33
3.6 Normalized Query Store Schema 33
3.7 Prologue handler interface 34
3.8 Prologue handler example usage to add Versioning and Timestamping prefixes . 35
3.9 Query Handler interface 36
3.10 Landing page and access control 38

4.1 An algorithm for handling queries including recommendations R4-R10 and R12 60

5.1 Evaluation process of the rdf_star module 72
5.2 Measuring runtime and memory consumption of retrieving historical data with the complex query against the small dataset which was incremented with timestamped inserts in both versioning modes 77
5.3 Measuring runtime and memory consumption of retrieving live data with the complex query against the small dataset which was incremented with timestamped inserts in both versioning modes 78
5.4 Measuring runtime and memory consumption of retrieving historical data with the simple query against the small dataset which was incremented with timestamped inserts in both versioning modes 79
5.5 Measuring runtime and memory consumption of retrieving live data with the simple query against the small dataset which was incremented with timestamped inserts in both versioning modes 80
5.6 Measuring runtime and memory consumption of retrieving historical data with the complex query against the small dataset which was incremented with timestamped updates in both versioning modes 81

5.7 Measuring runtime and memory consumption of retrieving live data with the complex query against the small dataset which was incremented with timestamped updates in both versioning modes 82
5.8 Measuring runtime and memory consumption of retrieving historical data with the simple query against the small dataset which was incremented with timestamped updates in both versioning modes 83
5.9 Measuring runtime and memory consumption of retrieving live data with the simple query against the small dataset which was incremented with timestamped updates in both versioning modes 84
5.10 Measuring runtime and memory consumption of retrieving historical data with the complex query against the big dataset which was incremented with timestamped inserts in both versioning modes 85
5.11 Measuring runtime and memory consumption of retrieving live data with the complex query against the big dataset which was incremented with timestamped inserts in both versioning modes 85
5.12 Measuring runtime and memory consumption of retrieving historical data with the simple query against the big dataset which was incremented with timestamped inserts in both versioning modes 86
5.13 Measuring runtime and memory consumption of retrieving live data with the simple query against the big dataset which was incremented with timestamped inserts in both versioning modes 86
5.14 Measuring runtime and memory consumption of retrieving historical data with the complex query against the big dataset which was incremented with timestamped updates in both versioning modes 87
5.15 Measuring runtime and memory consumption of retrieving live data with the complex query against the big dataset which was incremented with timestamped updates in both versioning modes 87
5.16 Measuring runtime and memory consumption of retrieving historical data with the simple query against the big dataset which was incremented with timestamped updates in both versioning modes 88
5.17 Measuring runtime and memory consumption of retrieving live data with the simple query against the big dataset which was incremented with timestamped updates in both versioning modes 88

List of Tables

2.1 RDA Data Citation recommendations[vUSP16] 6
2.2 Recommendations and adopters/implementations 8
2.3 Query Containment Solvers and supported features for SPARQL-Algebra, AFMU, TreeSolver [WCEGL13] and two more recent Solvers 14

3.1 Preparing the Data and the Query Store[vUSP16] 21
3.2 Persistently Identify Specific Data Sets[vUSP16] 22
3.3 Persistently Identify Specific Data Sets[vUSP16] 23
3.4 Resolving PIDs and Retrieving the Data[vUSP16] 23
3.5 Upon Modifications to the Data Infrastructure [vUSP16] 24
3.6 Search results from Google Scholar for individual RDF query languages on 16.06.2021 13:10 CEST 26
3.7 Database table *Celebrity* (left) and its RDF representation (right) 28
3.8 Example of a dynamic RDF* dataset that changes over time and is versioned using RDF*'s nested triples 30
3.9 Theoretical example of a dataset where multiple unique sort indexes are possible. 32
3.10 Cases as describe in Section 3.1.2 35
3.11 Metadata attributes mapped to categories and sources they could be automatically collected from 37
3.12 RDA Data Citation Recommendations and how they were fit into the RDF* Data Citation Framework 40

4.1 Equivalent SPARQL expressions and normalization measures 50
4.2 Rough estimation of time complexity of query normalization 53
4.3 Query Satellite Example 59

5.1 GraphDB settings 74
5.2 Hardware specifications 75
5.3 Detecting semantically identical queries - test results 82

Listings

2.1 View Query1 example . . . 17
2.2 View Query2 example . . . 17
2.3 Citation Query1 example . . . 17
2.4 Citation Query2 example . . . 17
2.5 Citation Snippet formatted with a Citation Function example1 . . . 17
2.6 Citation Snippet formatted with a Citation Function example2 . . . 17
4.1 Federated query . . . 42
4.2 Insert template for versioned triples . . . 43
4.3 Filled out insert template example for versioned triples . . . 43
4.4 Update template for versioned triples . . . 44
4.5 Filled out update template example for versioned triples . . . 44
4.6 Outdate template for versioned triples . . . 45
4.7 Filled out update template example for versioned triples . . . 46
4.8 Query performance version_all_rows . . . 47
4.9 Memory saving version_all_rows . . . 47
4.10 Query performance SPARQL template . . . 48
4.11 Memory saving SPARQL template . . . 48
4.12 Example query . . . 49
4.13 Equivalent example query . . . 49
4.14 Example query . . . 51
4.15 Example query tree . . . 51
4.16 Normalized query tree . . . 52
4.17 Normalized query . . . 53
4.18 Example for a timestamped query in memory saving mode . . . 54
4.19 Example query from the news dataset where we filter for Obama . . . 56
4.20 Derived dataset description from the dataset . . . 57
4.21 Mint dataset - example . . . 61
4.22 Retrieve minted dataset - example . . . 62
4.23 Build and install RDF* Data Citation API . . . 62
5.1 Property path test query . . . 67
5.2 Output of function translate_algebra(translateQuery(query)) . . . 67
5.3 Sequence paths - alternative 1 . . . 67
5.4 Sequence paths - alternative 2 . . . 67
5.5 Sequence paths - normalized query . . . 67
5.6 Simple query . . . 73
5.7 Complex query . . . 73
5.8 Versioned random data that gets inserted on query increment . . . 73
5.9 Random data that gets inserted on query increment . . . 74
5.10 SpeCS input file pair example - first file . . . 78

5.11 SpeCS input file pair example - second file 78
5.12 SpeCS input file pair example - first file 79
5.13 Normalized index entry algebra expression for the test case *rdf_type_predicate* 80
6.1 Data and metadata representations - alternative approach 1 91
6.2 Data and metadata representations - alternative approach 2 91

Bibliography

[ACD+17] A. Alawini, L. Chen, S. B. Davidson, N. Portilho Da Silva, and G. Silvello. Automating data citation: The eagle-i experience. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 1–10, 2017.

[Aro07] Siddhartha Cingh Arora. System and method for database versioning, July 31 2007. US Patent 7,251,669.

[BBG+16] Anita Bandrowski, Matthew Brush, Jeffery S Grethe, Melissa A Haendel, David N Kennedy, Sean Hill, Patrick R Hof, Maryann E Martone, Maaike Pols, Serena S Tan, et al. The resource identification initiative: A cultural shift in publishing. *Neuroinformatics*, 14(2):169–182, 2016.

[BCDC+15] Elena Bravo, Alessia Calzolari, Paola De Castro, Laurence Mabile, Federica Napolitani, Anna Maria Rossi, and Anne Cambon-Thomsen. Developing a guideline to standardize the citation of bioresources in journal articles (cobra). *BMC medicine*, 13(1):33, 2015.

[BDF16] Peter Buneman, Susan Davidson, and James Frew. Why data citation is a computational problem. *Communications of the ACM*, 59(9):50–57, 2016.

[C+14] World Wide Web Consortium et al. Rdf 1.1 concepts and abstract syntax. 2014.

[CBHS05a] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs. *Journal of Web Semantics*, 3(4):256, 2005. World Wide Web Conference 2005——Semantic Web Track.

[CBHS05b] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, page 613–622, New York, NY, USA, 2005. Association for Computing Machinery.

[CBHS05c] Jeremy J Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *Proceedings of the 14th International Conference on World Wide Web*, pages 613–622, 2005.

[CEGL18] Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. Sparql query containment under schema. *Journal on Data Semantics*, 7(3):133–154, 2018.

[dat19] Webconference data citation wg. `https://www.rd-alliance.org/group/data-citation-wg/webconference/webconference-data-citation-wg.html`, 2019.

[FCG+19] Martin Fenner, Mercè Crosas, Jeffrey S Grethe, David Kennedy, Henning Hermjakob, Phillippe Rocca-Serra, Gustavo Durand, Robin Berjon, Sebastian Karcher, Maryann Martone, et al. A data citation roadmap for scholarly data repositories. *Scientific Data*, 6(1):1–9, 2019.

[FKS19] Erika Fabris, Tobias Kuhn, and Gianmaria Silvello. A framework for citing nanopublications. In Antoine Doucet, Antoine Isaac, Koraljka Golub, Trond Aalberg, and Adam Jatowt, editors, *Digital Libraries for Open Knowledge*, pages 70–83, Cham, 2019. Springer International Publishing.

[GGV10] Paul Groth, Andrew Gibson, and Jan Velterop. The anatomy of a nanopublication. *Information Services & Use*, 30(1-2):51–56, 2010.

[GL06] Pierre Genevès and Nabil Layaïda. A system for the static analysis of xpath. *ACM Transactions on Information Systems (TOIS)*, 24(4):475–502, 2006.

[GMRW19] Bernhard Gößwein, Tomasz Miksa, Andreas Rauber, and Wolfgang Wagner. Data identification and process monitoring for reproducible earth observation research. In *2019 15th International Conference on eScience (eScience)*, pages 28–38. IEEE, 2019.

[gto] Gtopdb guide to pharamacology. `https://www.guidetopharmacology.org/`. Accessed: 2021-08-06.

[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*, 2017:131, 2017.

[Har17] O. Hartig. Foundations of rdf* and sparql*: (an alternative approach to statement-level metadata in rdf). In *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web 2017 (Vol. 1912)*, 2017.

[Hev07] Alan R Hevner. A three cycle view of design science research. *Scandinavian journal of information systems*, 19(2):4, 2007.

[HHK15] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. Reifying rdf: What works well with wikidata? *SSWS@ ISWC*, 1457:32–47, 2015.

[HHR+16] Daniel Hernández, Aidan Hogan, Cristian Riveros, Carlos Rojas, and Enzo Zerega. Querying wikidata: Comparing sparql, relational and graph databases. In *International Semantic Web Conference*, pages 88–103. Springer, 2016.

[HKP82] David Harel, Dexter Kozen, and Rohit Parikh. Process logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144–170, 1982.

[HTT+09] Anthony JG Hey, Stewart Tansley, Kristin Michele Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA, 2009.

[JD] Jennifer Pontius James Duncan. Ecosystem monitoring collaborator network. `https://www.rd-alliance.org/system/files/documents/170213_RDA_WGDC_Webinar_James_Duncan_Adoption_VermontMonitoringCooperative.pdf`.

[KCG16] A Clara Kanmani, T Chockalingam, and N Guruprasad. Rdf data model and its multi reification approaches: A comprehensive comparitive analysis. In *2016 International Conference on Inventive Computation Technologies (ICICT)*, volume 1, pages 1–5, 2016.

[LPPS13] Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. Static analysis and optimization of semantic web queries. *ACM Transactions on Database Systems (TODS)*, 38(4):1–45, 2013.

[MB15] Brigitte Mathiak and Katarina Boland. Challenges in matching dataset citation strings to datasets in social science. *D-Lib Magazine*, 21(1/2):23–28, 2015.

[met] Graphdb. `https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-star/`.

[nam14] Rdf 1.1 concepts and abstract syntax. w3c recommendation. `https://www.w3.org/TR/rdf11-concepts/#dfn-named-graph`, 2014.

[NBS14] Vinh Nguyen, Olivier Bodenreider, and Amit Sheth. Don't like rdf reification? making statements about statements using singleton property. In *Proceedings of the 23rd International Conference on World wide web*, pages 759–770, 2014.

[NBT+15] Vinh Nguyen, Olivier Bodenreider, Krishnaprasad Thirunarayan, Gang Fu, Evan Bolton, Núria Queralt Rosinach, Laura I Furlong, Michel Dumontier, and Amit Sheth. On reasoning with rdf statements about statements using singleton property triples. *arXiv preprint arXiv:1509.04513*, 2015.

[NDP15] Dario De Nart, Dante Degl'Innocenti, and Marco Peressotti. Well-stratified linked data for well-behaved data citation. *CoRR*, abs/1512.02898, 2015.

[OGO21] Fabrizio Orlandi, Damien Graux, and Declan O'Sullivan. Benchmarking rdf metadata representations: Reification, singleton property and rdf*. In *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*, pages 233–240, 2021.

[PB08] Eric Prud'hommeaux and Alexandre Bertails. A mapping of sparql onto conventional sql. *World Wide Web Consortium (W3C)*, page 4, 2008.

[PFF+16] Vassilis Papakonstantinou, Giorgos Flouris, Irini Fundulaki, Kostas Stefanidis, and Giannis Roussakis. Versioning for linked data: Archiving systems and benchmarks. *Proceedings of the Workshop on Benchmarking Linked Data (BLINK 2016) co-located with the 15th International Semantic Web Conference (ISWC)*, 1700, 2016.

[PS05] E. Prud'hommeaux and A. Seaborne. Sparql querylanguage for rdf. http://www.w3.org/TR/2005/WD-rdf-sparql-query-20050217/, 02 2005.

[RAVUP16a] Andreas Rauber, Ari Asmi, Dieter Van Uytvanck, and Stefan Proell. Identification of reproducible subsets for data citation, sharing and re-use. *Bulletin of IEEE Technical Committee on Digital Libraries, Special Issue on Data Citation*, 12(1):6–15, 2016.

[RAVUP16b] Andreas Rauber, Ari Asmi, Dieter Van Uytvanck, and Stefan Proell. Identification of reproducible subsets for data citation, sharing and re-use. *Bulletin of IEEE Technical Committee on Digital Libraries, Special Issue on Data Citation*, 12(1):6–15, 2016.

[RJ] Chantel Ridsdale Reyna Jenkyns, Melissa Cuthill. Ocean network canada. `https://www.rd-alliance.org/system/files/documents/Portage_Webinar_2020Nov24_OceanNetworksCanada_DynamicData.pdf`.

[SB19] Chris Schubert and Harald Bamberger. Handling continuous streams for meteorological mapping. In *Service-Oriented Mapping*, pages 251–268. Springer, 2019.

[SBM06] Vincent V Salomonson, William Barnes, and Edward J Masuoka. Introduction to modis and an overview of associated activities. *Earth science satellite remote sensing*, pages 12–32, 2006.

[Sil15] Gianmaria Silvello. A methodology for citing linked open data subsets. *D-Lib Magazine*, 21(1/2):1505–1524, 2015.

[Sil17] Gianmaria Silvello. Learning to cite framework: How to automatically construct citations for hierarchical data. *Journal of the Association for Information Science and Technology*, 68(6):1505–1524, 2017.

[Sil18] Gianmaria Silvello. Theory and practice of data citation. *Journal of the Association for Information Science and Technology*, 69(1):13, 2018.

[SVJ20] Mirko Spasić and Milena Vujosevic Janicic. Specs — sparql query containment solver. pages 31–35, 05 2020.

[TTY+05] Yoshinori Tanabe, Koichi Takahashi, Mitsuharu Yamamoto, Akihiko Tozawa, and Masami Hagiya. A decision procedure for the alternation-free two-way modal μ-calculus. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 277–291. Springer, 2005.

[vUSP16] Andreas Rauber; Ari Asmi; Dieter van Uytvanck; Stefan Proell. Tdata citation of evolving data: Recommendations of the rda working group on data citation (wgdc)t. pages 1–2, 2016.

[WCEGL13] Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. Evaluating and benchmarking sparql query containment solvers. In Harith Alani, Lalana Kagal, Achille Fokoue, Paul Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha Noy, Chris Welty, and Krzysztof Janowicz, editors, *The Semantic Web – ISWC 2013*, pages 408–423, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[ZMBD19] Carlo Zwölf, N. Moreau, Yaye-Awa Ba, and M.L. Dubernet. Implementing in the vamdc the new paradigms for data citation from the research data alliance. *Data Science Journal*, 18, 01 2019.