DELFT UNIVERSITY OF TECHNOLOGY

MSc Computer Science - Data Science & Technology

Bi-VAKs: Bi-Temporal Versioning Approach for Knowledge Graphs

---

Author:
Lisa MEIJER

Supervisor:
Dr. Christoph LOFI

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology
Electrical Engineering, Mathematics and Computer Science

Defence date:	July 27, 2022
Project duration:	July, 2021 – July, 2022
Student number :	4591879
Thesis committee:	Dr. Avishek Anand (chair)
	Dr. Christoph Lofi (supervisor)
	Dr. Przemyslaw Pawelczak (external)

Abstract

Bi-VAKs: Bi-Temporal Versioning Approach for Knowledge Graphs

by Lisa MEIJER

Over time Linked Data collections are continuously subject to change because of numerous reasons. Users could insert new observations, or they could rectify erroneous statements in these knowledge graphs. In order not to lose historically import information, this trend of evolving Linked Data collections increases the need to version these collections. Furthermore, retrieving prior versions and their in-between changes could provide Linked Data users relevant information. However, for some changes to these collections we should record both their transaction time and their valid time. To address these two problems of versioning Linked Data collections and having bi-temporal changes, we introduce the Bi-Temporal Versioning Approach for Knowledge graphs (Bi-VAKs): a prototypical bi-temporal change-based Version Control System for an arbitrary RDF dataset. Bi-VAKs registers both the transaction time and the valid time of a set of modified quads, and therefore, it allows for coupled historical and retrospective SPARQL queries. In addition, in order to enhance collaboration between its users Bi-VAKs also keeps track on provenance data; it supports diverged states; and provide a standard data access interface. However, since the standard RDF data model is atemporal, defining such a set of modifications (update) in RDF poses difficult challenges. Firstly, to indicate this transaction time and the valid time of an update Bi-VAKs divides a revision or version into a transaction revision and a valid revision. And hence it directly separates the metadata from the actual data. Secondly, to denote and retrieve the modified triples/quads within a update Bi-VAKs uses RDF-star and SPARQL-star. In order to connect these revisions we develop three reference strategies: the explicit, the implicit, and the combined reference strategy. These strategies let a transaction revision either refer explicitly to its corresponding valid revision(s) or implicitly by the same revision number and branch index. Based on these strategies we initiate some different approaches to query the updates from the revision-store. And, we propose some different methods to construct a (prior) version. However, to evaluate these different design decision we cannot use the existing uni-temporal benchmarks for our bi-temporal versioning approach. Therefore, we expand the BEAR benchmark to a bi-temporal benchmark (Bi-BEAR). By means of this benchmark we demonstrate that all three reference strategies have about the same storage size. We notice that the usage of a snapshot and retrieval of all updates worsen the version materialisation (VM), delta materialisation (DM), and version (VQ) query performance. In addition, the VM query look up time considerably decreases if only the matching updates are queried. And modified updates, branches, and more updates in the revision-store slightly lower the VM, DM, and VQ query performance. In addition, for the implicit and combined reference strategy the query time is rather the same, and sometimes even better if we sort the updates instead of aggregating them directly. Overall, the implicit reference strategy is performing best, and is quickly followed by the combined reference strategy.

Acknowledgements

Finalising my master thesis marks the end of my study life at the TU Delft. I must confess that doing this thesis was the most challenging and most lonely time of all my time at the TU Delft. This journey started 5 years ago when I began my bridging program in Computer Science after studying two years of Econometrics and Operational Research at the Erasmus University. I did not expect that such a year could open my eyes in so many ways. It introduced me to a complete new world of computer science. It taught me how to enjoy studying, especially regarding data science and data engineering. In addition, it gave me the skills that helps me to realise my own ideas. Although 3 years of computer science brought me a lot, I still have my doubts whether I am good enough and know enough to call myself a real computer scientist. I really hope that after this thesis I have proven myself enough and that it gives me the confidence to allow myself to be a computer scientist.

There are many people I would like to acknowledge. First, I would like to thank my supervisor Christoph Lofi for guiding me throughout the thesis process. Although we did not talk a lot about the thesis itself, I really have enjoyed our conversation about other topics. I also want to thank him for his patience and his belief, but also for setting limits so I could finish this project. In addition, I would like to thank Joep Meindertsma for all our enjoyable meetings, for his enthusiastic ideas on Linked Data, on distributed systems and on Web3 technologies, and mostly for his support. I also would like to thank the rest of my committee, Avishek Anand and Przemyslaw Pawelczak for making time to evaluate my work. Furthermore, I would like to thank all my colleagues and patients at the Rijndam rehabilitation centre. During this year they always gave me a pleasant and funny Thursday morning.

Furthermore, I want to express my gratitude to my friends and family. In particular, I would like to thank my grandmother, to whom I could come every week with all my questions and uncertainties during this tough year. I want to thank Jacques for all our walks together and for his endless enthusiasm and discussing from economics to relationships. Writing a thesis during Covid times was not always easy. These two people, in particular, motivated me, and made the experience much more enjoyable. I also would like to thank my boyfriend, Dirk, for making my weekends more fun. Finally, I want to thank my parents for everything they have done for me, their unconditional love and support, and for encouraging me to finish this project.

Lisa Meijer
July 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The value of data can hardly be underestimated nowadays, especially, if it contains meaningful information. Data could gain more meaning by linking it. A way of structuring data so that it is interconnected is Linked Data. Linked Data forms a web of data, which it not only understandable and interchangeable between humans but also between machines (Berners-Lee, 2006). Besides, the data model representing Linked Data, the *Resource Description Framework (RDF)* (Cyganiak, Wood, and Lanthaler, 2014), has many advantages over other data models, such as its graph structure, and its interoperability. Therefore, Linked Data has many real-world applications in various industries, even in a small country as the Netherlands. Digital Network for Heritage (Netwerk Digitaal Erfgoed[1]), for instance, uses Linked Data to increase the usability of digital heritage collections. Linked Data Overheid[2] (LiDO) is an example of a Linked Data collection that provides insight into the connections between national and European regulations, judgements by Dutch and European judges, parliamentary documents and official announcements. And Kadaster[3] has made the Addresses and Buildings Key Register (BAG) available as Linked Data such that it can be easily connected with other (governmental) key registers.

However, over time these arising interconnected datasets are continuously subject to change due to numerous reasons, such as the insertion of new observations or rectifying erroneous conceptualisations (Umbrich et al., 2010). This trend of evolving Linked Data collections increases the need of versioning these knowledge graphs, particularly, not to lose historically important information. Retrieving prior Linked Data versions and their in-between changes could provide Linked Data users relevant information. If the BAG is archived over time, users of the BAG could, for instance, examine people's relocation behaviour. Nevertheless, the BAG faces a significant problem when an archive only records the time of changes on a single time line: the time that a change is either stored or valid. A change of address does not have to be valid yet at the time of recording. Unfortunately, adding time information to the RDF model raises difficult challenges, and therefore, these bi-temporal changes cannot be modelled straightforward in order to store and retrieve the RDF dataset versions. In addition, Linked Data users work collectively on these collections, which yet leads to new problems, primarily, in a distributed setting. In this thesis, we introduce a bi-temporal Linked Data versioning approach that records both time dimensions for each change and that supports the evolution of a dataset in a collaborative setup. We explain our approach in more detail, after we outline the problem of versioning Linked Data collections and the problem of temporal data modelling in section 1.1 and 1.2.

[1] https://netwerkdigitaalerfgoed.nl
[2] https://linkeddata.overheid.nl
[3] https://www.kadaster.nl

1.1 Versioning Linked Data

In addition to Kadaster, Digital Network for Heritage (NDE) could also benefit from versioning its data collections, since versioning prevents information loss, and hence contributes to one of NDE's main purposes: sustainability of its Linked Data collections. Versioning lets important heritage-related sources remain available, accessible and verifiable for its users in the long term. Nonetheless, apart from the arguments that querying prior versions, or identifying version differences provide valuable information, we can define many other reasons for the need of a Linked Data version control system, in particular, the following reasons:

Understanding the Evolution of Data 'Volatility' is an essential concept of the current web, as it is continuously evolving. By also storing the previous states of the dataset, data users could inspect which statements were added, removed or modified, and therefore could understand, and prove why the dataset has been changed accordingly. Besides, knowing the evolution of the data opens up opportunities for assessing the quality of the data. The amount of modified statements, for instance, may indicate how actively a dataset is maintained (Meinhardt, Knuth, and Sack, 2015).

Collaboration As people work collectively on RDF datasets, often, they may not immediately agree on the content of the dataset. Data users might differ in their motivation to contribute to the dataset, for instance because of their differences in perspective or understanding of the data (Arndt et al., 2019). In order to express dissent without damaging the common dataset, we need to keep track of multiple different dataset versions (Arndt et al., 2019). By having diverged versions, versions could be developed separately, while still sharing a common part of the dataset. When eventually these data users have reached consensus, these versions might either be synchronised by exchanging the changes.

Version References and Data Consistency In a collaborative setting usually data users work with a dataset fixed at a specific point in time. In order to let other users know which version they work with, they should explicitly refer to their specific static state of the dataset. Furthermore, version referencing could help to keep datasets uniform. As a version reference denotes a particular state of the dataset at a point in time, knowing the reference of two separate datasets might indicate whether one dataset is consistent with values in another dataset at the same point in time.

Provenance Recording and Exploration When we evaluate, or interchange datasets from different sources, it may be desired to record the provenance information of the data. Provenance information improves the determination of data reliability, and hence the data quality, since it gives data users the opportunity to assess not only what has been changed, but also who it has changed, and why and when the dataset has been changed.

Error Correction and Backup Although collaboration let data users contribute to a common dataset, it may also cause insufficiencies in the data. By tracking all the changes collaborators might be better at detecting, and subsequently correcting these failures in coming versions. Besides, versioning can allow to switch to an older version until erroneous changes are undone. In addition, the data might even get lost, if an attack or complete crash corrupt the dataset. Periodically performing version controlled data set backups might help to avoid content loss by restoring the data from the previous versions.

In summary, versioning Linked Data is not only about storing and hence retrieving versions, and their differences. It is also about maintaining data quality and collaborating between data users. However, these concepts add even more challenges to an already

complicated task: the *storage-recreation trade-off* (Bhattacherjee et al., 2015). By separately storing full versions a particular version can quickly be retrieved, but these versions may have a lot of overlap. By storing only the differences (deltas) between versions there is less overlap, but higher latencies due to a version's construction time.

1.2 Temporal Data Modelling in Linked Data

Just like Kadaster, LiDO also has a need to manage the validity of their Linked Data collection(s), especially, in order to obtain the part of the legislation that was effective at the requested time. A judge, for example, can refer to a law article in general or to the version in force on the day of the judgment. However, besides that regulations can enter into force proactively, it can also enter into force retroactively. Therefore, not only the date of entry into force matters, but also the view date: Could someone know at the time of acting on the legal text that it had been amended retroactively (Opijnen, 2018)? It allows users to view regulations as they were before they were changed by a retroactive amendment, and as they were when they were effective at a requested time. Thus, instead of a uni-temporal data model, which considers either the time the data is valid (*valid time*) or the time the data is stored (*transaction time*), we need a bi-temporal data model to consider both these time dimensions and to allow for coupled historical and retrospective queries.

1.3 Bi-Temporal Versioning Approach for Knowledge graphs (Bi-VAKs)

To address the two problems of versioning Linked Data collections and having bi-temporal changes, we introduce the Bi-Temporal Versioning Approach for Knowledge graphs (Bi-VAKs): a prototypical change-based version control system for an arbitrary RDF dataset, also known as knowledge graphs. Bi-VAKs registers both the transaction time and the valid time of a set of modified quads, and hence allows for bi-temporal SPARQL (Update) queries. By having a change-based storage strategy Bi-VAKs has a proper balance between the storage-recreation trade-off, because it only stores the set of inserted and deleted quads (change sets) between consecutive versions. In addition, this strategy offers Bi-VAKs considerable opportunities for collaboration between Linked Data users in a distributed environment, such as branching off a common dataset, keeping track of provenance information, and the possibility of exchanging changes. However, the RDF data model itself is atemporal. Therefore, adding two-dimensional time information to an in RDF defined change poses difficult challenges.

In the first place, a change must have both a transaction time as well as valid time. Therefore, our prototypical system Bi-VAKs distinguishes *transaction revisions* from *valid revisions* to denote respectively the transaction time and the valid time. Moreover, this division between *transaction revisions* and *valid revisions* respectively separates the metadata, such as the author, the modification type, or the creation time, from the actual changes, such as the collection of inserted and deleted RDF quads made to the Linked Data collection, the name of a branch, or the description of a tag.

In the second place, we need to detect and to model a RDF structured change set in such a way that we can refer to the actual modified data (triples, or quads). Therefore, Bi-VAKs utilises SPARQL Update queries and RDF-star (Hartig and Thompson, 2014) to respectively detect and model these change sets, which we call *Updates*. RDF-star and its

associated query language SPARQL-star are an extension on RDF and SPARQL to annotate RDF triples by a more compact representation of statement-level metadata than standard RDF reification (Brickley and Guha, 2014).

In the third place, we must retrieve these *Updates* to construct a (prior) version and return a response for a SPARQL query. However, by querying all changes over a certain time period we probably cannot see the wood for the trees because of the excessive number of *Updates*. Therefore, Bi-VAKs extracts the basic triple pattern from the incoming SPARQL queries. And it subsequently uses a triple pattern and SPARQL-star (Hartig and Thompson, 2014) to query these *Updates* more efficiently.

In addition to the fact that a well modelled change could enhance collaboration, for example through the exchangeability of changes, it may also be made easier when Linked Data users can continue using their existing RDF tools and stores. Therefore, Bi-VAKs is a middle-ware between the Linked Data users and SPARQL endpoints of some triple/quad-stores. It provides a standard SPARQL 1.1 endpoint supporting a read/write interface on a versioned RDF dataset. Therefore, Bi-VAKs connects to the revision-store - a triple/quad-store containing all transaction and valid revisions - and to data store(s) - triple/quad-store(s) containing a materialisation version of the RDF dataset. Furthermore, Bi-VAKs also supports some general versioning concepts, such as branching, tagging, and reverting. Figure 1.1 shows a sketch of Bi-VAKs, and demonstrates the main contributions of Bi-VAKs: the creation, retrieval, and management of Linked Dataset versions. Bi-VAKs is still a first step to a bi-temporal versioning approach. Storing uncompressed all changes in a single RDF dataset might not be space efficient. And, adding the valid time as object to a triple might not be query efficient.

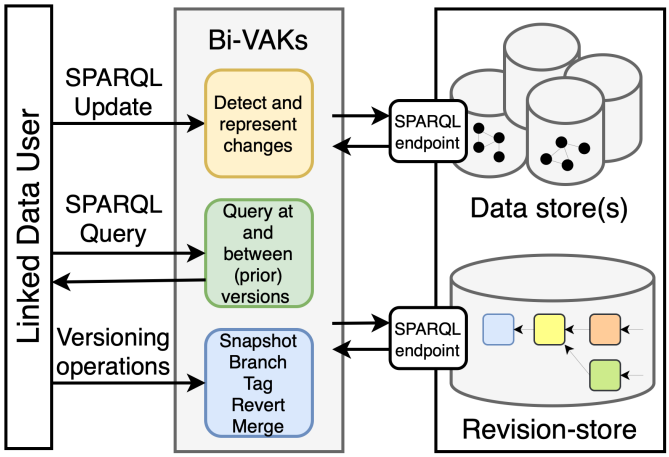


FIGURE 1.1: Outline of the prototypical Bi-Temporal Versioning Approach for Knowledge graphs (Bi-VAKs).

In thesis we focus on a generic solution for versioning Linked Data. Bi-VAKs does not make any assumptions on the application domain, and additional semantics such as OWL (McGuinness, Van Harmelen, et al., 2004) or SKOS (Miles and Bechhofer, 2009). It has to rely on the pure RDF data model. Besides, since blank nodes cannot uniquely be identified over different datasets, they pose an issue when it comes to referring to modified quads. Therefore, we decided not to consider blank nodes. Furthermore, due to time limitations we were not able to re-implement existing version control system for RDF datasets. In addition, all implemented existing RDF versioning approaches consider a uni-temporal data model, instead of a bi-temporal data model. Therefore, we do not

aim to compete with these existing systems. We entirely evaluate our system by self-predefined experiments, and on a self-created bi-temporal versioned RDF graph. Besides, we do not evaluate our system on complex SPARQL queries. We only assess our system on basic triple pattern queries: the basic elements of SPARQL. We refer to Chapter 9 for a more in-depth study on these issues.

1.4 Research Questions

In the previous section, we debated the need for a collaborative Linked Data Version Control System that can manage both the *valid time* and the *transaction time* of a collection of modified quads, and therefore, can support bi-temporal SPARQL (Update) queries. Such a change set (or update) is exchangeable, and allows users to efficiently distribute their changes among other collaborators. In addition, metadata can easily be added to these updates, and to further enhance collaboration an update can be in different diverging states simultaneously. Therefore, we opt for an in RDF defined update representations that decouples transaction revisions from valid revisions registering respectively the transaction time and the valid time, and the metadata and the actual data. This design choice compromises the first research question:

> **RQ1:** How can we design a Linked Data change-based collaborative version control system that can manage both the *valid time* and *transaction time* of a collection of changes made to a RDF dataset?

But since the RDF data model itself is atemporal, the inserted and deleted quads should be modelled otherwise to let an update refer to the modified quads. Therefore, we represents such an update by means of RDF-star. Due to its compact representation, RDF-star increases the efficiently of data exchange. Moreover, such a representation makes it possible to simply request only the changes that actually contain the triple pattern(s) specified in the user's query. And, thus it enables efficient version materialisation (VM), delta materialisation (DM), and version (VQ) basic triple pattern queries on a versioned RDF dataset. We formulate the second research question as follows:

> **RQ2:** How can we represent updates by using RDF-star, and enable efficient bi-temporal version materialisation, delta materialisation, and version basic triple pattern queries by using SPARQL-star?

In addition to a theoretical framework of our versioning approach, we also build a prototypical implementation. In order to evaluate its ingestion and query performance we need a bi-temporal versioned RDF dataset, and a collection of queries and their VM, DM and VQ results over the different bi-temporal versions. To the best of our knowledge, due to the limited research on bi-temporal RDF versioning approaches, thus far no work has been proposed to systematically benchmark bi-temporal RDF archives. Therefore, for convenience we extend the uni-temporal Benchmark for RDF archives (BEAR) proposed by Fernández et al. (2019) so that this new bi-temporal benchmark (Bi-BEAR) can also assess bi-temporal RDF versioning approaches. This leads to the third and final research question:

> **RQ3:** How can we expand the BEAR Benchmark to a bi-temporal benchmark, and what is the ingestion and query performance of our prototypical change-based version control system?

1.5 Contributions

We summarise the contributions of this thesis as follows:

- We present a prototypical change-based version control system for RDF datasets, Bi-VAKs, that records both the transaction time and the valid time of its dataset changes in order to provide for bi-temporal SPARQL (Update) queries.
  - Bi-VAKs creates for each versioning operation a transaction revision and (multiple) corresponding valid revision(s), which indicate respectively the transaction time and the valid time. In addition, this division makes it possible to query these revisions separately; to refer to the same or multiple revisions simultaneously; and to have these revisions serve different purposes.
  - Bi-VAKs represents its changes by using RDF-star in order to enable referring to the added, deleted, and modified quads within an RDF structured change.
  - Bi-VAKs has a standard SPARQL 1.1 endpoint supporting a read/write interface on a bi-temporal versioned RDF dataset. It detects and creates the RDF dataset changes from the SPARQL Update queries. And it supports version materialisation, delta materialisation, and version SPARQL SELECT basic triple pattern queries to allow for historical and retrospective queries.
  - Bi-VAKs allows its users to make changes over its existing changes, and it ensures the completeness and accuracy of all changes ever made to Bi-VAKs. Bi-VAKs even computes a cryptographically secure hash for each revision, and it includes the hash of its predecessors to guarantee the integrity of each revision.
  - Bi-VAKs supports branching, tagging, reverting and merging to support collaboration between Linked Data users.
- We expand the uni-temporal BEAR Benchmark to a benchmark for bi-temporal RDF version control systems (Bi-BEAR).
- We evaluate Bi-VAKs' ingestion and query performance on various input parameters, such as the number of branches or the number of updates.

1.6 Thesis Outline

The rest of the thesis is structured as follows. In Chapter 2 we outline the needed background such as an explanation of the resource description framework (RDF), RDF-star, and their query languages. We also created a overview of the already known version control systems for datasets in general and not RDF in particular. In Chapter 3 we give a literature overview of various RDF version control systems; how changes can be represented and detected; and an overview of RDF archiving benchmarks. In Chapter 4 we define Bi-VAKs requirements and concepts. And while we give in Chapter 5 a detailed description of Bi-VAKs' conceptual design, we describe in Chapter 6 some of its practical design choices. In Chapter 7 we briefly explain the actual implementation of Bi-VAKs in Python, and in Chapter 8 we conduct a set of experiments, evaluation Bi-VAKs ingestion and query performance from several angles. Following the experimental results, we discuss these in-depth in Chapter 9, even as the limitations of Bi-VAKs itself. We end this work with some concluding remarks and a number of proposals for future work in Chapter 10.

Chapter 2

Background

In this thesis we develop an approach to version a bi-temporal RDF dataset using RDF-star. In this chapter we establish background knowledge, and explain the main concepts and terminology which form the background of our work. We begin this chapter by introducing the terminology of Linked Data, RDF, and SPARQL in Section 2.1. In Section 2.2, we explain the triple annotation approach RDF-star, and SPARQL-star. Subsequently, in Section 2.3 we present some terminology of temporal data. Finally, in Section 2.4, we give an global overview of existing (data) version control systems which uses other data models than RDF.

2.1 Linked Data

Linking data may give data more meaning, because it establishes certain relations that were otherwise unknown. Thereby, it is important to have this data available in a standard format, as it could make the data more accessible, manageable and exchangeable to both humans and machines. *Linked Data* defines such a set of design principles for sharing machine-readable interlinked data using the international standards of the World Wide Web Consortium (W3C), for example the use of URIs (Uniform Resource Identifier) to name things. Another principle stated that one should use the open standards RDF and SPARQL, which we explain in more detail in the following sections.

2.1.1 The Resource Description Frame (RDF)

The Resource Description Frame (RDF) (Cyganiak, Wood, and Lanthaler, 2014) is a data model to represent interconnected data by making statements about resources. The core structure of RDF is a set of triples - each consisting of a *subject*, a *predicate*, and an *object* - which is called a *RDF graph*. A triple can be interpreted as a statement about a resource, the subject in essence. The objects describes its value, and the predicate explains their relationship or property. A triple can be represented as a node-arc-node link in a directed graph, which is illustrated in Figure 2.1. Such a node can be an IRI (Internationalized Resource Identifiers), literal, or a blank node. An IRI or a literal denotes a resource, which can be anything, including physical things, documents, abstract concepts, numbers, and strings. IRIs define a resource by a unique identifier, while, literals define it by a datatype, such as strings, number, and dates. A blank node, however, does not identify a specific resource, and is, therefore, called an anonymous resource. A *subject* of a triple is either represented by an IRI or blank node, a *predicate* only by an IRI, and an *object* by an IRI, blank node, or a literal.

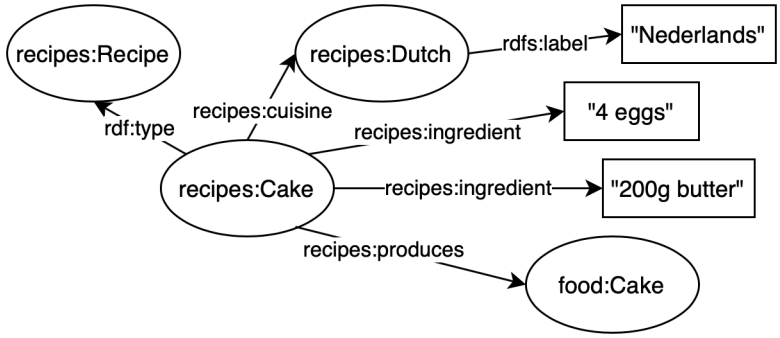


FIGURE 2.1: An example RDF graph describing the resource http://recipehub.nl/recipes#Cake, its properties, and relationships.

A collection of IRIs, which are intended for the same use, is called an RDF vocabulary. The IRIs in an RDF vocabulary often begin with a common sub-string known as a namespace IRI. For instance, the RDF Schema vocabulary contains the IRIs indicating the RDF Schema language, and its namespace IRI is `http://www.w3.org/2000/01/rdf-schema#`. In addition to a single RDF graph, multiple graphs may also be grouped together in a RDF dataset. An RDF dataset always consists of exactly one default graph and zero or more named graphs. These named graphs are named by either an IRI or a blank node. Therefore, a statement can also be represented as a quad (4-tuple), where the fourth component denotes the graph name. Triples or quads can be stored and retrieved respectively in a triple- or quad-store through semantic queries, such as SPARQL (Section 2.1.2). Nevertheless, to enable the exchange of RDF graphs and RDF datasets between systems, they should be serialized into RDF documents syntaxes to represent the RDF data. An RDF document is a document that encodes an RDF graph or RDF dataset in a concrete RDF syntax. A variety of RDF concrete syntaxes exists, such as Turtle, RDFa, JSON-LD, or TriG. Listing 2.1 shows the Turtle syntax of the RDF graph illustrated in Figure 2.1.

LISTING 2.1: An example of the Turtle syntax.

```
@prefix recipes: <http://www.recipehub.nl/recipes#> .
@prefix food: <http://www.recipehub.nl/food#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

recipes:Cake
    rdf:type recipes:Recipe ;
    recipes:cuisine recipes:Dutch ;
    recipes:ingredient "4 eggs" ;
    recipes:ingredient "200g butter" ;
    recipes:produces food:Cake ;
.
recipes:Dutch
    rdfs:label "Nederlands" ;
.
```

2.1.2 SPARQL

SPARQL 1.1 (Prud'hommeaux and Seaborne, 2008) is the W3C's recommended RDF query language. Generally, a SPARQL query consists of a set of triple patterns, which is called a graph pattern. A triple pattern $t := (s, p, o) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup L \cup B \cup V)$ is an RDF triple that allows a variable (V) at any position. A basic graph pattern will be matched with the RDF terms (IRI, literal, blank node) in the RDF graph such that these RDF terms can be substituted for the variables in order to return all possible query

solutions. In other words, graph pattern matching produces a solution sequence, where each solution has a set of bindings of variables to RDF terms. SPARQL has four query types (query forms):

- SELECT returns the variables and their bindings to RDF terms.
- CONSTRUCT returns a single RDF graph constructed by substituting the RDF terms for the variables in a graph template.
- ASK returns a boolean that indicates whether a query pattern has a solution.
- DESCRIBE returns a single RDF graph containing all triples that describe a resource.

Listing 2.2 demonstrates a SPARQL SELECT query to fetch all recipes, which have the Dutch cuisine, and Listing 2.3 demonstrates its result in JSON Format[1]. Nonetheless, all triple patterns must match to produce a solution. Therefore, if the triple pattern in the SPARQL OPTIONAL part does not match, no bindings are created, but the solution is not eliminated. Moreover, SPARQL UNION provides an approach of combining graph patterns so that one of the several alternative graph patterns may match. To restrict solutions, SPARQL offers the FILTER operator that only returns those bindings for which the filter expression evaluates to true. Furthermore, in order to create a specific sequence, any sequence modifier, such as LIMIT, ORDER, and DISTINCT, can be applied. However, many RDF data stores hold multiple RDF graphs. To specify the named graphs that are used for matching a basic graph pattern, there exist multiple options: the FROM clause to specify the default graph and the FROM NAMED clause to specify the named graph(s); the GRAPH operator; or the SPARQL protocol request.

LISTING 2.2: An example of a SPARQL SELECT query that fetches all recipes with the Dutch cuisine.

```
PREFIX recipes: <http://www.recipehub.nl/recipes#> .
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
SELECT ?recipe
WHERE {
    ?recipe recipes:cuisine ?cuisine .
    ?cuisine rdfs:label "Nederlands" .
}
```

LISTING 2.3: An example of a SPARQL 1.1 Query Results JSON Format from the query presented in Listing 2.2

```
{ "head": {
    "vars": ["recipe"]
    } ,
  "results": {
    "bindings": [
        {
            "recipe": { "type": "uri", "value": "http://recipehub.nl/recipes#Cake" }
        }
    ]
  }
}
```

SPARQL Update

In order to modify existing RDF graphs or datasets SPARQL 1.1 Update (Gearon, Passant, and Polleres, 2013) provides a number graph update operations and graph management operations. While the INSERT DATA or DELETE DATA graph update operation respectively directly inserts or deletes the triples stated in the query into a graph, the DELETE/INSERT

[1] https://www.w3.org/TR/sparql11-results-json/

operation inserts and deletes a group of triples which matches the query patterns stated in the update query. In addition, The LOAD operation respectively inserts the content of a document representing a RDF graph into a graph, and the CLEAR operations deletes the triples from a single or multiple graph(s). The graph management operations only concentrates on graph operations, such as the creation, deletion, copy and move of a full RDF graph.

2.2 Triple Annotation

A long standing issue for RDF is the lack of a convenient way to annotate RDF triples and query such annotations. Previous research already proposed a number of approaches in order to support statement-level metadata annotations. The approach that is supported by any RDF store is standard RDF reification (Hayes and Patel-Schneider, 2014). To express metadata about a given RDF triple, RDF reification include four additional RDF triples to refer to a reified triple. This is rather inefficient for exchanging as well as for managing RDF data, and its SPARQL syntax is also cumbersome (Hartig and Thompson, 2014). Another approach is via named graphs (Carroll et al., 2005). The identifier of the named graph can be considered as a node in the RDF graph, and it can be used to represent the provenance of a set of triples. A significant drawback is the overload of named graphs, and it hinders an application to use named graphs for other use cases.

An alternative to the named graph as identifier is Singleton Properties (Nguyen, Bodenreider, and Sheth, 2014). Singleton Properties uses the predicate (property) as an identifier to make a metadata annotation. However, it is highly inefficient for querying data, and it introduces a large number of unique predicates, which is untypical for RDF data and thus again disadvantageous for commonly-used SPARQL optimization techniques. Furthermore, it is much harder to share and combine different datasets, and moving data from one tool to another tool is not as simple. Therefore, Hartig (2017) proposed RDF-star (RDF*), which is an extension of the RDF 1.1 standard that allows making statements about other statements. To query these annotations, Hartig (2017) also extended the SPARQL query language. This extension is called SPARQL-star (SPARQL*).

2.2.1 RDF-star

RDF-star proposes a more efficient reification serialisation syntax. Because of its compact representations this syntax reduces the document size, and it shorten the SPARQL queries, which increases the efficiency of data exchange and improves comprehensibility. In addition, triple stores could be further optimised internally to handle RDF reification by using such a representation. The idea of RDF-star is to put a triple in the subject or object position of another triple that represent metadata about the embedded triple. These triples that include a triple as a subject or object are known as *RDF-star triples* (Hartig et al., 2021). In a given RDF-star graph, an identical triple may be both an *embedded triple*, which is RDF-star triple that is used as the subject or object of another RDF-star triple, and an *asserted triple*, which is just an element of an RDF-star graph. Thus, an embedded triple does not exist on its own as a triple in the RDF-star graph.

RDF-star extends a number of RDF concrete syntaxes to define the following new syntaxes: Turtle-star, TriG-star, N-Triples-star, and N-Quads-star. Turtle-star, for instance, uses double angle brackets to enclose a triple that serves as a subject or object of another triple, which is illustrated in Listing 2.4. In Listing 2.4, the triple ⟨ reciples:RecipeChowMein

recipes:cuisine attributes:ChineseCuisine ⟩ is only an embedded triple and not an asserted triple, like the triple ⟨ attributes:ChineseCuisine rdfs:label “Chinese keuken” ⟩.

LISTING 2.4: An example of the Turtle-star syntax.

```
@prefix recipes: <http://recipehub.nl/recipes#> .
@prefix attributes: <http://recipehub.nl/attributes#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://www.example.org/> .

attributes:ChineseCuisine rdfs:label "Chinese keuken" .
<< reciples:RecipeChowMein recipes:cuisine attributes:ChineseCuisine >> :accordingTo :
    Lisa .
```

2.2.2 SPARQL-star

SPARQL-star (SPARQL*) is an RDF-star-aware extension of SPARQL in order to query RDF-star graphs (Hartig, 2017). In addition to the standard solution mappings of SPARQL which bind variables only to IRIs, blank nodes, or literals, the solution mappings of SPARQL-star introduces the possibility to bind variables to RDF-star triples, and enables users to directly access metadata triples via queries. Therefore, in contrast to the corresponding queries for the other proposals (e.g., RDF reification, singleton properties), SPARQL-star queries are very concise. This compact syntax does not require users to write verbose patterns and other constructs. An example of a SPARQL-star is illustrated in Listing 2.5 that shows that SPARQL-star allows for BIND clauses with an embedded triple pattern.

As mentioned in Section 2.1.2, SPARQL queries can take four forms: SELECT, CONSTRUCT, DESCRIBE, and ASK. The introduction of RDF-star changes the result serialization format of CONSTRUCT and DESCRIBE, as they should now be represented by Turtle-star instead of Turtle for instance. The serialization format for representing SELECT query results should also be extended in order to represent the embedded triple RDF term by adding an extra RDF term “triple”.

LISTING 2.5: An example of a SPARQL query that fetches all recipes with the label “Chinese keuken”.

```
PREFIX recipes: <http://www.recipehub.nl/recipes#> .
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
PREFIX : <http://www.example.org/> .

SELECT ?recipe
WHERE {
    << ?recipe recipes:cuisine ?cuisine >> :accordingTo  :Lisa .
    ?cuisine rdfs:label "Chinese keuken" .
}
```

2.3 Temporal Data

Temporal data is data that varies over time, and that represents a state at a certain point in time. We could collect temporal data for weather forecasting, analysing food prices or studying demographic trends, and so on. Time values in the data can be described as a point time or as a duration of time (time interval). Whereas a time point refers to a single point in time, a time interval is a range of time points starting from a start time point and ending at an end time point. In general, temporal data can be further divided into two types: the transaction time, and the valid time. Although, sometimes one also considers other types, such as decision time, publication time or efficacy time.

The transaction time is the time period when the data are actually stored or available in the database (Jensen and Snodgrass, 1999). It primarily captures the time-varying states of the database (Zhang et al., 2021). The valid time is the time period during which a statement is true in the modelled or real world (Jensen and Snodgrass, 1999). It particularly captures the time-varying states of the modeled world independent of its recording time (Zhang et al., 2021). However, the valid time is not always known. We refer to these time points as *anonymous timestamps*. To handle data along these two different timelines, we need a bi-temporal data model instead of a uni-temporal model. With bi-temporal modelling it is possible to rewind the information to "how it actually was" in combination with "when it was actually recorded". Therefore, all data becomes immutable: information can never be overwritten or discarded. Hence, we can make retroactive changes to data points while retaining the history of what these data points believed to be in the past.

2.4 Dataset Version Control Systems

A Version Control System (VCS) keeps track of, manage and record changes to source code, documents, or other collections of information over time. Generally, VCS serves as a safety net to protect collections of information from irreversible harm. VCS allow users to revert datasets to their previous versions in order to recover lost data, or locate and fix problems. In addition, with VCS users are able to compare the changes made to the data over time, and exchange their changes in order to work simultaneously on the same dataset. Often, VCS does not only store the evolution of the data, but also the reasoning behind the change(s), the user who made the change(s), and its timestamp. The data file(s), its complete history of changes, and its metadata are stored in a repository. In the initial VCS these repositories were kept locally - the repository and its user were on the same machine - or were located on a shared folder allowing users to collaborate within a local area network. Subsequently, in order to enable geographically dispersed teams working on projects together, centralised VCS became popular. These client/server VCS store the entire repository on a central server, and its clients are able to read from and submit changes to it. But as clients must have a network connection to check in and out changes, in these recent years distributed VCS are turning out to be the standard VCS. These distributed systems allow collaboration without the need of a central repository, as clients store the full repository locally, and re-synchronize it with other repositories later on.

Since the repository holds all versions of the content, changes to the data need to be saved from the working directory/copy of the user to the repository in order to create a new version of the content. This act of recording changes to the repository creating a new version is called *commit*. Each commit might contain changes possibly to different parts of the dataset, its associated metadata, a unique identifier (version number), a message, a reference to the parent commit, a reference to the author of the changes, and a reference to the committer. For VCS it is advantageous to perform atomic commits. It ensures that if any part of the dataset, that has been committed, fails to be accepted by the repository, the whole commit should fail, and all of the changes completed in the atomic commit are reversed. And it ensures that only one atomic commit is processed at a time. Commits occur in a sequence over time, and thus can be arranged in order, either by a version number or a timestamp. In a linear order, with no branching or undoings, each commit is based on its immediate predecessor alone, such that they form a linear graph, called the main stream. As collaboration is a significant aspect of VCS, users can branch off from the main stream, such that they can work on several versions of the content in parallel

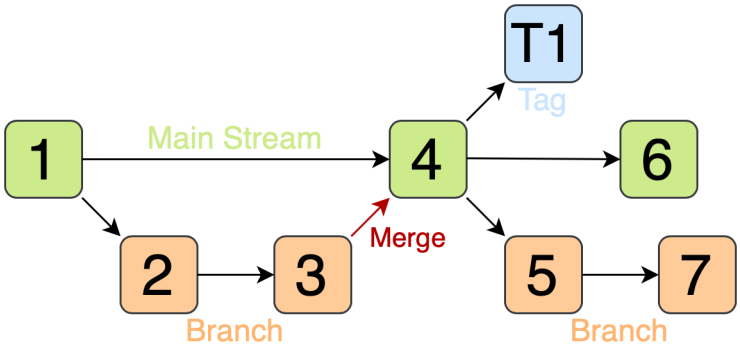


FIGURE 2.2: A rooted directed acyclic version graph. Trunk is in green, branches in orange, and tag in blue.

without interference. These branches can be merged later on into the main stream or into another branch by incorporating both changes. If these changes, however, overlap, it may be difficult or even impossible to merge. The main stream and all branches might form a rooted directed acyclic graph, because the commits' parents are always backwards in time, and the graph starts with the oldest version. The last element(s) of such a graph are called the head(s). Figure 2.2 illustrates a simple example of a version graph, for which (6) and (7) are the heads of this graph. In some VCS, users can annotate a commit as a tag. Tags are special types of branches that mark a milestone along the evolution of a dataset.

Datasets are rather varied in nature, ranging from small to large, from structured (tabular) to unstructured (text and arbitrary binary objects), and from largely complete to noise and incomplete. The most well known version control systems are for (plain text) files and unstructured datasets, such as Subversion, and Git. Most version control terminology, and techniques still are clearly inspired by these systems. Therefore, before we introduce in Section 2.4.2 the version control systems for structured datasets, as RDF datasets, we give in section 2.4.1 a convenient introduction of the systems for unstructured datasets.

2.4.1 Version Control Systems for (plain text) files and unstructured datasets

One of the first attempts of a version control system is the revision control system (RCS) (Tichy, 1982) that helps managing multiple versions of a single text file. RCS automates the basic storing, retrieval, logging, branching, and merging of revisions. RCS stores an entire copy (snapshot) of the most recent version of the file and its reverse differences (deltas) to get back to the older states. Such a reverse-delta schema allows for a very quick checkout of the current revision. However, the older the checkout revision, the longer the checkout takes since an increasing number of deltas need to be calculated against the current snapshot. In addition, the version history can be edited by the users, and only one user can work on a file simultaneously. Since RCS only manages individuals files, and not collections of files, the centralized version control system, Concurrent Versions System[2] (CVS), expands upon RCS and adds support for repository-level change tracking, and a client/server networking element. The first step is to set up a centralized repository on a remote server, and to import a project into the repository, which is called a module. Clients obtain copies of modules to their working directory on their local machines by checking them out from the remote server. Clients can modify their own

[2] https://www.gnu.org/software/trans-coord/manual/cvs/cvs.html

working copy concurrently, but before they can commit their changes, they need to keep their working copy up-to-date. Although RCS has been seen as a de facto standard during the nineties, it has some limitations, such as no atomic commits, no global version numbering, and no good support for binary files. Therefore, the centralised version control system Subversion[3] (SVN) is built with the intention of fixing some of these limitations. A Subversion repository on a remote server has a directory called db/revs in which all revision tracking information for the checked-in (committed) files and directories is stored. A commit includes changes to multiple files and directories and is stored in a new file in the revs directory. In order to be more space efficient, the full content of a file is stored, once it is committed for the first time. Upcoming commits of the same file will store only the deltas.

Even though centralised version control systems have been a common approach over many years, currently distributed version control systems have gathered widespread attention and adoption. These distributed systems, however, have some challenges, many of which are inherent in any distributed system. For instance, it is impossible to temporally order revisions in any given repository, as users can commit in parallel. In order to ease exchange of commits, these systems must keep track of the ancestry relations between commits. Both the recognized distributed version control systems Git and Mercurial[4] have opted to use SHA1 hashes of the contents of the commit as their unique identifier. By including the reference to the parental commit in the SHA1 hash, all history leading up to any revision can be verified using its hash. Furthermore, such an identifier can also be used to verify whether the contents of the commit have not been changed, or whether the contents of multiple commits are identical. Mercurial stores the history of a file in a firelog consisting of its individual file content and its deltas from the previous version. The manifest puts together the information about the files. If a file has not changed between two commits, the entry for that file in the two revisions of the manifest will point to the same revision of its filelog. The changelog contains the meta-information about each commit. Git[5] is by far the most widely used distributed version control systems, nowadays, especially in software development. Git provides rapid branching and merging strategies through smart referencing, and offers synchronizing with multiple remote repositories. Every user works locally on a working copy, which is a complete clone of a (remote) Git repository. A Git repository consists of commits that each represents the current state of the files in the working directory at a given time by pointing to a Git tree object. A git tree object consists of a group of pointers, pointing to other tree objects and binary large objects (blobs). A blob stores the contents of a file as binary data. Thus, instead of storing the changes to each file as the aforementioned systems do, Git stores the references to other references or to blobs containing the new created or changed files.

2.4.2 Version Control Systems for (semi-)structured datasets

Although there has been a long line of version control systems for unstructured datasets, they all are designed to deal with modest-sized files, and therefore, they have significant limitations when handling large files, (unordered) structured datasets, and large number of versions. When using such a version control system (e.g. Git) teams tend to store their data in files, often using highly ad hoc and manual version management techniques. It is not uncommon that directories contain thousands of files with names like data1-v1.csv, data1-v2.csv, data1-v1-after-applying-program-XYZ.txt, etc. (Bhardwaj

[3] https://subversion.apache.org
[4] https://www.mercurial-scm.org
[5] https://git-scm.com

et al., 2015). In general, these systems require each user to have a separate, complete, local copy of a dataset, which is impractical within large, multi-gigabyte or terabyte-scale databases (Maddox et al., 2016). In addition, they are designed to store unstructured data. Therefore, they use general-purpose differencing tools (like Unix diff) to determine deltas and compare versions, which are highly inefficient for structured data, and slow to handle the scale (Maddox et al., 2016). Furthermore, these systems do not support sufficient querying and retrieval functionalities, e.g., querying specific versions of datasets, performing joins across versions, or explore and query metadata.

The above-mentioned reasons urge the need of (semi-)structured data version control. Prior research (Bhattacherjee et al., 2015), however, demonstrated there is a natural trade-off between the storage requirements and the querying efficiency. All data version control systems must deal with this trade-off, despite they are designed for handling different types of datasets. There has already been plenty of research on time travel (or temporal) databases (e.g., relational (Ahn and Snodgrass, 1986; Salzberg and Tsotras, 1999; Tansel et al., 1993), graph (Khurana and Deshpande, 2013) and array (Seering et al., 2012; Soroush and Balazinska, 2013)) which is, essentially a linear chain of versions, that annotates each record with a time interval. This interval denotes the version it belongs to, and facilitates "time-travel" queries. However, many of these systems lack support for branching, and merging versions.

An example of a data VCS for relational database is proposed by Bhardwaj et al. (2015). This multi-version data management system, Dataset Version Control System (DVCS), provides similar version management functionalities to Git, such as branching, but DVCS has significantly more powerful versioning query language and scales to larger and more structured datasets. In addition, they built a hosted platform on top of DVCS, DataHub, which is analogous to Github[6]. Via DataHub, users can interact with datasets, either by directly issuing queries to DataHub servers or by checking out local copies of datasets. A key component of DataHub is the Relational Dataset Branching System, Decibel (Maddox et al., 2016), which is a relational storage system with built in version control to track, integrate, and query changes made by different users to the same dataset. Decibel uses a very flexible logical data model that treats datasets as unordered collections of records, where records are identified by primary keys. Users can interact with Decibel by opening a connection (or session) to the centralized Decibel server. In Decibel, every modification conceptually results in a new version. In update-heavy environments, however, this would lead to extra overhead. Consequently, Decibel allow users to classify some of these versions as committed versions, which can only be checkout, and queried. However, Decibel requires redesigning the entire database stack, and does not benefit from the querying capabilities that exist in current database systems. Therefore, Huang et al. (2017) proposed a dataset version control system, ORPHEUSDB, which is built on top of standard relational databases as a middleware layer.

Nowadays, there also exist a number of non-academic data or/and Machine Learning experiment management tools for data science applications. An example of a commercial tool is Data Version Control[7] (DVC) that works on top of Git repositories. In order to enable data versioning, it replaces large files, dataset directories, machine learning models, etc. with small meta-files, which point to the original files. Another example is Splitgraph[8], which is a tool for building, versioning, and querying datasets. It works on top of PostgreSQL and uses SQL for all versioning and internal operations.

[6] https://github.com
[7] https://dvc.org
[8] https://www.splitgraph.com

Chapter 3

State of the Art

In addition to research on Version Control Systems (VCS) for relational databases, there is also some research on VCS for Linked Data collections. A simple storage strategy to manage versions over Linked Data is Independent Copies (IC) (Fernández, Polleres, and Umbrich, 2015). IC stores and manages a separate copy of the dataset(s) for each version. Therefore, it is easy to query a specific version of the dataset(s). However, a disadvantage of IC is it scalability regarding its storage space, especially when users frequently make small changes. All versions should be stored separately, and corresponding triples are copied over several places. Besides, whenever information about the changes is requested, these changes must firstly be computed on the fly. These challenging tasks such as computing deltas and loading the appropriate versions require non-negligible processing efforts (Fernández et al., 2019). An example of an IC approach is SemVersion (Völkel and Groza, 2006). SemVersion is inspired by the classical version control systems (e.g., CVS or SVN), with support for multiple RDF graphs, branching and merging.

To address this scalability problem, Arndt et al. (2019) proposed another storage strategy: Fragment-based (FB). Instead of fully copying all triples across versions FB only stores snapshots of changed fragments such as resources, sub-graphs or individuals graphs. FB corresponds to the storage strategy that Git[1] has implemented. A Git repository only stores the snapshots of modified files instead of a snapshot of all files and stores the reference(s) to the unchanged files. This FB version control system, Quit Store (Arndt et al., 2019), is built on top of Git, and provides collaborative version control for multi-graph RDF datasets. In addition, it uses the PROV ontology (Lebo et al., 2013) for metadata management. Quit Store physically stores the history of the dataset in text files (N-Triples files) by considering each version to be a commit, and makes this history accessible via a SPARQL endpoint. Although Quit Store only stores snapshots of the modified files, it still requires a lot of memory whenever frequently small changes are made (Pelgrin, Galárraga, and Hose, 2021).

Another storage strategy that Fernández, Polleres, and Umbrich, 2015 defined is the Timestamp-based (TB) approach. TB associates to each triple its time- or version-related metadata such as temporal validity intervals or insertion/deletion timestamps. Hauptmann, Brocco, and Wörndl, 2015, for instance, stored each triple in a different named graph such that the context information can be used as identifier for the triple. Accessing random versions can explicitly be queried through virtual graphs. X-RDF-3X (Neumann and Weikum, 2010) is an extension of RDF-3X, and annotated each triple with a created and deleted timestamp. It reconstructed a version at time t by returning all triples for which t falls into the corresponding interval. x-RDF-3X does not support versioning for multiple graphs, neither branching nor tagging. RDF-TX (Gao, Gu, and Zaniolo, 2016)

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

is an in-memory query engine that efficiently manages temporal RDF data over single RDF graphs. It indexes triples and their time metadata using a compressed multi-version B+tree. RDF-TX achieves a fast evaluation of temporal queries by proposing an extension of SPARQL (SPARQL^T), which support queries over the temporal RDF graphs. RDF-TX outperforms similar systems as X-RDF-3X in terms of query efficiency.

v-RDFCSA (Cerdeira-Pena et al., 2016) is built on the RDF compressor RDFCSA (Brisaboa et al., 2015). It used RDFCSA to compress the RDF triples in the archive, and encoded the information about the versions using bitsequences. V-RDFCSA provides efficiently retrieval algorithms by utilizing RDFCSA self-indexing capabilities for accessing the compressed triples, and perform bit-based operations to enable versioning queries. However, its query functionalities are still limited since it only supports VM, DM, and VQ queries on basic triple patterns. Dydra (Anderson and Bendiken, 2016) is an RDF graph storage service that supports multi-graph dataset versioning. It indexes quads in six ways implemented as B+trees and link them to visibility maps. It used insertion and deletion timestamps to indicate the time the quad was present. Furthermore, they extended the query language by introducing the REVISION keyword, which is similar to the SPARQL keyword GRAPH for referring to different dataset versions. Moreover, another work that annotates to each triple its time metadata is StarVers (Kovacevic et al., 2022). StarVers employed RDF-star to annotate each triple with temporal metadata.

Although these systems seem rather storage efficient because of their indexing, and compression techniques, not all existing Linked Data tools can directly utilise them, as these techniques need special triple/quad-stores. In addition, TB approaches are not so suitable for a proper collaboration. It is hard to exchange these indexed and compressed version datasets between different systems, and adding diverging states is complex. Furthermore, these approaches are so far designed for either the valid time or the transaction time of a triple. It would add an extra dimension of complexity to design them for both transaction and valid time.

The storage strategy that addresses the aforementioned issues is the Change-Based (CB) approach (Fernández, Polleres, and Umbrich, 2015). CB only stores the differences (deltas) between versions. These deltas are much easier to exchange between systems, and can typically be stored in any triple/quad-store. Furthermore, both the transaction and the valid time can simply be attached to such a change. Therefore, this storage strategy is suitable for our approach. However, its storage-efficiency still depends on the ratio of changes occurring between consecutive versions. The IC approach is still more storage-efficient than CB, if large changes between consecutive versions are made. Furthermore, the CB approach has a high materialization cost for version materialisation (VM) queries, particularly for long sequences of deltas between two snapshots (delta chain). Hence, the fundamental challenge of RDF archiving is the storage-recreation trade-off: the more storage we use, the faster it is to recreate or retrieve versions, while the less storage we use, the slower it is to recreate or retrieve versions (Bhattacherjee et al., 2015). In section 3.3 we further outline the existing research on the change-based and hybrid RDF version control systems.

However, before the difference between versions can be stored and retrieved, they first have to be detected in a manual or automatic fashion or a posteriori using some change detection tool. Therefore, we first give in Section 3.1 a brief overview of change detection approaches. In addition, in order to exchange the changes between collaborators, we require a proper change representation which is understandable between humans and machines. Subsequently, in Section 3.2 we discuss which solutions have already been put forward in the literature to represent changes. Furthermore, we also looked into the

limited research on bi-temporal RDF version control systems in Section 3.4. Finally, we end this chapter by outlining the general different queries, and the some RDF archiving benchmark, which can be used to evaluate the ingestion and query performance of RDF versioning systems.

3.1 Change Detection

As a change-based storage approach only stores the changes (or deltas) between two consecutive versions, it requires a change detection approach to identify the changes between two (subsequent) versions. One approach would be to use a differencing algorithm. A differencing algorithm calculates the differences between two versions of a RDF dataset or a RDF graph. OntoView Klein et al. (2002) compared two versions of an ontology by using a UNIX diff inspired comparison tool. It is a lined based tool, and thus the ontology is first canonicalized at the syntactic level before being given to the diff tool. Promptdiff (Noy and Musen, 2002) is an ontology-versioning tools that determines what has changed between two versions by using a set of heuristics matchers, IF-THEN rules, that compares the two ontologies on different triple conditions. And COnto-Diff (Hartung, Groß, and Rahm, 2013) is a generic diff algorithm that automatically determine expressive (compact) diffs between given ontology versions. This approach is based on the result of a (semi-) automatic match operation. A problem of such algorithms is that they are either based on lines that always requires a canonicalisation algorithm or on a collection of well defined definitions that describes the semantics of different change operations (Papavassiliou et al., 2009). Another approach is to detect changes using temporal queries over a version log that is maintained during updates (Plessers and Troyer, 2005). The downside is that the detection process requires a version log to be maintained, so it essentially requires recording information on the changes as they happen. Moreover, many existing triple/quad-stores do not support a version log. Therefore, another approach is to detect the changes from a SPARQL (Update) query. Roussakis et al. (2015) and Singh, Brennan, and O'Sullivan (2018), for instance, relied on plain SPARQL queries to query the two RDF dataset versions, and they extracts the changes from the results. Frommhold et al. (2016) detects its changes from SPARQL Update Query itself. Namely, a SPARQL Update Query is a quite prominent way to gather information about a change, because it might literally contain the changes.

3.2 Change Representation

Changes (or deltas) between two RDF versions can be seen as changes between two RDF datasets. As a RDF statement is the smallest directly manageable piece of knowledge, a change can be represented as a set of triple/quad insertions and deletions. Therefore, a triple update in a RDF dataset could be modeled as a deletion of the old triple, followed by an insertion of the new value of the triple. A RDF change model generally has two semantic layers of change (Roussakis et al., 2015): simple (low-level) and complex (high-level) changes. Whereas, simple changes only distinguish added and deleted triples, complex changes define human-readable changes with the purpose of obtaining a more concise explication on the whys and hows of the change. These data changes can be described via RDF using specialized vocabularies and change ontologies, or through patch file formats. A patch file format is a (text) file containing a list of difference between two versions, mainly produced by running a related difference algorithm.

3.2.1 Change Vocabularies and Ontologies

To represent changes to RDF datasets by means of RDF itself a variety of vocabularies and ontologies exists. Haase and Stojanovic (2005), for instance, formalized the semantics of changes with respect to an individual OWL dataset by defining arbitrary consistency conditions: structural, logical, and user-defined consistency. They assumed that an important aspect in the evolution process of an ontology is to guarantee the consistency of the ontology when it changes. Therefore, they ensured by means of defined resolution strategies that these conditions are maintained as the ontology evolves. Watkins and Nicole (2006) developed a document provenance ontology as the basis for a semantic version control system. They used named graphs to record the class instance in the provenance ontology and signed them with a digital signature (SHA-1) to verify the integrity of the provenance data.

Changeset[2] is a vocabulary that describes changes to resource descriptions using RDF reification. The class ChangeSet encapsulates the delta between two versions of a single resource description, and it uses RDF statement(s) to refer to the added and deleted triple(s). In the Log Ontology (Auer and Herre, 2006) changes are also represented as reified statements. The main concept introduced in this work is the concept of atomic graphs, which provides a practical approach to deal with blank nodes in change sets. Additionally they introduced a formal hierarchical system to structure a set of changes and evolution patterns that lead to the changes of a knowledge base.

Instead of using RDF reification the Graph Update Ontology[3] (GUO) used a different approach. GEO also described a change of a single resource description in a specified graph, but a change is encoded as a graph node with properties that point to the actual target node. However many of these vocabularies only included triple level change information, and they did not include resource level change information as well. Therefore, Singh, Brennan, and O'Sullivan (2018) proposed the Delta-LD Change model. The Delta-LD Change model models the atomic level change operations, such as the added and deleted triples (triple level change information). In addition, it also models the objective of the atomic changes, such as the change type and the subject (resource IRI) of the change in the older and newer version (resource level change information). Namely, a change in the IRI of a resource can lead to structurally broken interlinks.

Apart from using change vocabulary to model changes, provenance vocabularies are also frequently used. The Provenance Vocabulary Core Ontology (PROV-O) (Lebo et al., 2013), for instance, is not designed to describe changes in particular, but many of its properties can be reused. As a World Wide Web Consortium (W3C) Recommendation PROV-O is the de facto standard for the representation, exchanging, and integrating of domain-independent provenance information generated in different applications and under different contexts. Despite of the usefulness of change vocabularies due to the allowance of communication via Semantic Web technologies, these change representations are generally not very compact, and for each atomic change, new RDF statements must be created. Furthermore, none of these vocabularies are accepted as a real standard, and it is still not commonly adopted to describe changes in RDF.

[2] https://vocab.org/changeset/schema.html
[3] http://purl.org/hpi/guo

3.2.2 Patch File Formats

While patch file formats are not as expressive as RDF change vocabularies, they incline to be more compact making them suitable for practical space-conscious applications. Although patch files are mainly text files, they are mostly derived from RDF syntaxes. Berners-Lee and Connolly (2004) defined that such a format needs to uniquely identify what is changing and to distinguish between the pieces added and those subtracted. Based on these needs they proposed a patch format for RDF deltas by using the property "replacement" to express any change. Listing 3.1 illustrates an example of a triple update in that format.

LISTING 3.1: RDF Delta

```
@prefix diff: <http://www.w3.org/2004/delta#>.
{ ?x  bank:accountNo "1234578"; bank:balance 4000}
 diff:replacement
{ ?x  bank:accountNo "1234578"; bank:balance 3575}.
```

Another patch format is PatchR (Knuth, Hercher, and Sack, 2012). PatchR is a bit of a combination between a patch file and a RDF change vocabulary. It expresses change requests (patches) within a Linked Data collection to propose changes for correcting incorrect data. The Patch Request ontology describes various kinds of these patch requests by adopting concepts from the Graph Update Ontology (GUO) and includes the provenance information based on PROV ontology (Lebo et al., 2013). Listing 3.2 illustrates an example of a single patch.

LISTING 3.2: PatchR

```
@prefix : <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix pat: <http://purl.org/hpi/patchr#> .
@prefix guo: <http://webr3.org/owl/guo#> .
@prefix prov: <http://purl.org/net/provenance/ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
@prefix dbp: <http://dbpedia.org/resource/> .
@prefix dbo: <http://dbpedia.org/ontology/> .

:Patch_15 a pat:Patch ;
    pat:appliesTo <http://dbpedia.org/void.ttl#DBpedia_3.5> ;
    pat:status pat:Open ;
    pat:update [
        a guo:UpdateInstruction ;
        guo:target_graph <http://dbpedia.org/> ;
        guo:target_subject dbp:Oregon ;
        guo:delete [
         dbo:language dbp:De_jure ] ;
        guo:insert [
            dbo:language dbp:English_language ] ];
    prov:wasGeneratedBy [
        a prov:Activity ;
        pat:confidence "0.5"^^xsd:decimal ;
        prov:wasAssociatedWith :WhoKnows ;
        prov:actedOnBehalfOf :WhoKnows#Player_25 ;
        prov:performedAt "..."^^xsd:dateTime ] .
```

TurtlePatch[4] is a patch format that is inspired by SPARQL 1.1 UPDATE language, but it is easier to parse and to process than SPARQL 1.1 UPDATE. TurtlePatch only describes the inserted and deleted triples, and it does not add provenance information to the patch, as is illustrated in Listing 3.3.

[4] https://www.w3.org/2001/sw/wiki/TurtlePatch

LISTING 3.3: TurtlePatch

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX s: <http://www.w3.org/2000/01/rdf-schema#>
DELETE WHERE {
<http://www.w3.org/People/Berners-Lee/card#i> foaf:mbox <mailto:timbl@w3.org>.
}
INSERT DATA {
<http://www.w3.org/People/Berners-Lee/card#i> foaf:mbox <mailto:timbl@hushmail.com>.
<http://www.w3.org/People/Berners-Lee/card> s:comment "This is my general description of
     myself.\n\nI try to keep data here up to date and it should be considered
    authoritative.".
}
```

RDF Patch[5] is a file format for recording changes made to an RDF dataset. The text format is similar to N-Triples. Each line starts with an operation code. A means "add", D means "delete", PA means "add prefix", H means "header". TX and TC delimit a block of quad, triple, and prefix changes in order to process a patch faster. Listing 3.4 demonstrates an example of RDF Patch.

LISTING 3.4: RDF Patch

```
H id <uuid:0686c69d-8f89-4496-acb5-744f0157a8db> .
H prev <uuid:3ee0eca0-6d5f-4b4d-85db-f69ab1167eb1> .
TX .
PA "rdf" "http://www.w3.org/1999/02/22-rdf-syntax-ns#" .
PA "owl" "http://www.w3.org/2002/07/owl#" .
PA "rdfs" "http://www.w3.org/2000/01/rdf-schema#" .
A <http://example/SubClass> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
    www.w3.org/2002/07/owl#Class> .
A <http://example/SubClass> <http://www.w3.org/2000/01/rdf-schema#subClassOf> <http://
    example/SUPER_CLASS> .
A <http://example/SubClass> <http://www.w3.org/2000/01/rdf-schema#label> "SubClass" .
TC .
```

JSON LD PATCH[6] defines the inserted and deleted triples via a JSON document structure that contains an array of single addition or deletion operation objects. These objects consist of an operation (addition or deletion) and the subject, predicate and object of the added or deleted triple, which is presented in Listing 3.5.

LISTING 3.5: JSON LD PATCH

```
[{
    "op": "add",
    "s": "http://example.org/my/resource",
    "p": "http://example.org/ontology#title",
    "o": {
      "value": "New Title",
      "type": "http://www.w3.org/2001/XMLSchema#string"
    }
  },
  {
    "op": "del",
    "s": "http://example.org/my/resource",
    "p": "http://example.org/ontology#publicationYear",
    "o": {
      "value": "2013",
      "type": "http://www.w3.org/2001/XMLSchema#gYear"
    }
}]
```

The Linked Data Patch Format (LD Patch) (Bertails, Champin, and Sambra, 2015) is a format for describing changes, which could be applied to Linked Data, and it should be

[5] https://afs.github.io/rdf-delta/

[6] https://github.com/digibib/ls.ext/wiki/JSON-LD-PATCH

seen as a language for updating RDF graphs in a resource-centric fashion. The "Add", and the "Delete" operation respectively add and remove RDF triples from the target graph. The "Bind" operation binds an RDF Term to a variable. The "Cut: operation removes one or more triples connected to a specific blank node, and the "UpdateList" operation updates members of an RDF collection. Listing 3.6 illustrates an example of LD Patch.

LISTING 3.6: LD Patch

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix schema: <http://schema.org/> .
@prefix profile: <http://ogp.me/ns/profile#> .
@prefix ex: <http://example.org/vocab#> .

Delete { <#> profile:first_name "Tim" } .
Add {
  <#> profile:first_name "Timothy" ;
    profile:image <https://example.org/timbl.jpg> .
} .

Bind ?workLocation <#> / schema:workLocation .
Cut ?workLocation .

UpdateList <#> ex:preferredLanguages 1..2 ( "fr-CH" ) .

Bind ?event <#> / schema:performerIn [ / schema:url = <https://www.w3.org/2012/ldp/wiki/
    F2F5> ]  .
Add { ?event rdf:type schema:Event } .

Bind ?ted <http://conferences.ted.com/TED2009/> / ^schema:url ! .
Delete { ?ted schema:startDate "2009-02-04" } .
Add {
  ?ted schema:location [
    schema:name "Long Beach, California" ;
    schema:geo [
      schema:latitude "33.7817" ;
      schema:longitude "-118.2054"
    ]
  ]
} .
```

Although, these patches might be very compact and storage efficient, and thus easier to exchange. It is not possible to query the information within these patches effectively. Therefore, all patches must first be retrieved before a particular version can be constructed. Furthermore, these patches are not convenient in describing provenance information or other metadata of a change.

3.3 Change- and Hybrid-Based Storage Approaches

Most literature from in the previous sections only focuses on the representation and detection of changes, and not on the systems that utilise these representations and detection approaches, such as our approach. Therefore, we describe in this section the literature of the change-based and hybrid-based RDF versioning systems. Cassidy and Ballantine (2007) proposed a changed based version control system for RDF graphs based on the theory of patches as implemented in the Darcs version control system[7]. The theory of patches describes a version as a sequence of patches. Cassidy and Ballantine (2007) saved each patch as a named graph, which contain the reified added and deleted triple(s). Although their approach covers the versioning operations commute, revert, and merge, it only supports linear version tracking. Im, Lee, and Kim (2012) suggested a system based on a relation database. This database stores a snapshot of the latest RDF version and the

[7] http://darcs.net

deltas between the consecutive versions. The deltas are stored separately in a delete and an insert table. To access a specific version the version is constructed on the fly by joining the original version and the relevant delta tables. Furthermore, to avoid unnecessary computation due to duplicates in deltas they compressed the delta between two specific versions and stored them in advance.

R&Wbase (Sande et al., 2013) is a system based on the principles of distributed version control as it supports branching and merging. R&Wbase stores all triples internally as quads, where the context value identifies the version and indicates whether the triple was added or deleted. Aside from the triples, these named graphs also stores the provenance of the delta by using the PROV-O ontology (Lebo et al., 2013) such as its unique hash and reference to its predecessor. To access separately a random version R&Wbase provides a straightforward way through SPARQL by using virtual graphs. R43ples (Graube, Hensel, and Urbas, 2014) also stores the differences between revisions as separate named graphs, and it performs version control on a graph level. Therefore, for both systems it is not guaranteed to use named graphs for other purposes, and potentially an extensive number of named graphs can be created. R43ples works as a SPARQL proxy in front of an existing triple-store. To model a version as Linked Data R43ples uses the Revision Management Ontology (RMO), which is an extended and more domain-specific version of the PROV ontology. To access the revision named graph R43ples uses an extended SPARQL protocol language that adds the specified revision information and the commit message to the query. The R43ples approach also supports tags and branches, which can be queried by using the new SPARQL keywords BRANCH and TAG. Another change-based versioning approach is proposed by Frommhold et al. (2016). The Virtuoso server detects the changes within a single transaction. Each deleted or added instance is stored in a diff table referencing the author, change message, timestamp and transaction. Before they created a patch of all triples from the diff table and the meta-information, they first performed the Minimum Self-Contained Graph (MSG) calculation to deal with blank nodes. Furthermore, to ensure the integrity of the patch they added a unique hash value to the patch. Although this system is able to track changes for any possible RDF dataset, even when blank nodes are involved, it does not provide support for branches and for distributed collaboration.

TailR (Meinhardt, Knuth, and Sack, 2015) is a hybrid IC/CB approach since it stores the initial or some other version of the dataset as snapshot and the consecutive versions as deltas. TailR has been implemented as a Python web service on top of a data store. It comprises a Push API for submitting dataset change information, and a read-only Memento API for providing access to prior version information of linked data resources via memontoes (Sompel et al., 2009). Memento (Sompel et al., 2009) introduces time travel for the web. It uses standard HTTP capabilities in order to link and retrieve past states of web resource using datetime negotiation in HTTP. The most recent developed version control system is OSTRICH (Taelman et al., 2019). OSTRICH is a hybrid IC-CB-TB storage technique that efficiently resolves version materialisation, delta materialisation and version triple pattern queries with result offsets. OSTRICH stored the initial version of a dataset as a fully materialized snapshot in the HDT file format and all other versions as changesets. In order to reduce storage requirements and lookup times of any triple pattern query, each change set is stored in a six tree-based index structure, where values are dictionary-encoded and timestamped. The delta chain is constructed in an aggregated deltas fashion as proposed by Im, Lee, and Kim (2012). This aggregated deltas fashion means that the construction of any version only requires at most one delta and one snapshot since each delta inherits the changes of its preceding delta. Despite its good query

performance, OSTRICH does not support branching and merging yet, and it changes are not easily exchangeable.

3.4 Bi-Temporal Versioning Approaches

Although much research has been done on uni-temporal RDF versioning approaches, very little research has been done on bi-temporal or multiple-temporal RDF versioning approaches. Grandi (2011) proposed a multi-temporal RDF data model (Multi-tempRDF) in order to support RDF light-weight ontology versioning. They assigned a temporal pertinence to a RDF triple, which is a subset of the multidimensional time domain. Yang and Yan (2018) presented TRDFS that maps temporal XML document with Schema to the temporal RDF triples by introducing some rules and algorithms. Furthermore, they extended the temporal RDF graph model proposed by Gutiérrez, Hurtado, and Vaisman (2007) with both valid and transaction time. They used a classical RDF graph with temporal vocabularies to represent temporal RDF. Another model that records both the valid and transaction time is RDFt (Zhang et al., 2019). RDFt represents temporal data by adding the time information and the update count information to the predicate part of a triple, which respectively indicate the valid and transaction time. RDFt uses a RDF reification to refer to a triples, and refers to the time and update count information via the RDFt vocabulary. They also defined a query language called SPARQL[t] to query RDFt. A non-academic bi-temporal RDF database is TerminusDB[8]. TerminusDB is an opensource graph database that is focused on collaboration and versioning. It fully preserves data lineage and change history with built-in revision control (Otterdijk, Mendel-Gleason, and Feeney, 2020).

3.5 Retrieval Functionality

In order to evaluate to query performance of RDF versioning systems on different time perspectives, Fernández et al., 2019 introduced five fundamental query types, which are referred to as query atoms:

- Version Materialisation (VM) intends to query a specific (historical) dataset version. Example: *Which series were available on Netflix a year ago?*
- Delta materialization (DM) aims to retrieve the difference between two versions and for a certain query. Example: *Which series left or were new on Netflix between a year ago and now?*
- Version query (VQ) returns the versions for which a particular query gives a result. Example: *At what time was episode X present on Netflix?*
- Cross-version join (CV) joins the results of two queries (Q1 and Q2) respectively between versions Vi and Vj. Example: *Which series were available on Netflix a year ago and now?*
- Change materialization (CM) returns a list of versions in which a given query Q produces consecutively different results. *At what time was episode X new on Netflix or left it Netflix?*

[8] https://terminusdb.com

3.6 RDF archiving benchmarks

In addition to the need of query atoms, we also require a benchmark for evaluation purposes. A benchmark serves as a standard and allows us to easily compare systems by measuring and assess their performance against the benchmark's performance. One of the first proposed RDF archiving benchmark is EvoGen (Meimaris and Papastefanatos, 2016). EvoGen generates synthetic, evolving data represented in the RDF model and offers workload generation capabilities. It is an extension of the Lehigh University Benchmark (LUBM) (Guo, Pan, and Heflin, 2005) generator with additional classes and properties for enabling schema evolution. It allows the user to adjust parameters of the dataset and the query generation process, such as the amount of changes and the number of versions. EvoGen has adapted the LUBM's existing benchmark queries by new types of queries, such as temporal queries, queries over changes, and queries across versions. However, its functionality is restricted to the LUBM scenario, and the evolving RDF data data is rather synthetic.

SPBv (Papakonstantinou et al., 2017) is also a highly configurable and extensible benchmark. The data generator of SPBv extends the data generator of SPB (Kotsev et al., 2016), which produces RDF descriptions of instances of the BBC creative work core ontology. The SPBv data generator tries to simulate the evolution of these descriptions by storing them in different versions according to their creation date. Apart from the required storage space and the time a system needs for storing a new version, SPBv also evaluates the time required to answer a query. Therefore, SPBv can generate a set of SPARQL queries for a described set of versioning queries types. Similar to EvoGen, SPBv is also not really realistic. In addition, it only supports insertions, but no deletions and modifications, which are rather important for evaluating a change-based versioning approach.

BEAR (Fernández et al., 2019) is a benchmark for RDF archive systems which uses real-world data sets from different domains. BEAR consists of three main data sets, namely BEAR-A, BEAR-B, and BEAR-C. BEAR-A contains the first 58 weekly snapshots from the Dynamic Linked Data Observatory (Käfer et al., 2013). The Blank Nodes are replaced with Skolem IRIs and the context information is removed, which resulted in 58 versions having between 30M and 66M triples per version. BEAR-A provides triple pattern queries and their results for three different versioned query types. These triple pattern queries are split in low and high number of results and have the form (S??), (?P?), (??O), (SP?), (?PO), (S?O) and (SPO). BEAR-B contains the 100 most volatile resources from the DBpedia Live changesets (Morsey et al., 2012) over the course of three months (August to October 2015) at three different granularities: instant, hour and daily. BEAR-B provides realistic triple pattern queries, mixing (?P?) and (?PO), and their results, based on the most frequent triple patterns from the DBpedia query set. Finally, BEAR-C contains 32 weekly snapshots of the Open Data Portal Watch project (Umbrich, Neumaier, and Polleres, 2015), for which the Blank Nodes are replaced with Skolem IRIs. BEAR-C provides 10 complex SPARQL queries. While, these queries cannot be solved by the current archiving strategies, they could stimulate the development of new query resolution algorithms. BEAR provides a baseline for RDF archive implementation based on HDT and Jena's TDB store for IC, CB, TB, IC/CB, and TB/CB approaches.

Chapter 4

Requirements and Concepts for Bi-VAKs

This chapter give the requirements and concepts of our prototypical bi-temporal change-based Linked Data Version Control System: Bi-Temporal Versioning Approach for Knowledge graphs (Bi-VAKs). The aim of Bi-VAKs is to version a RDF dataset that is subject to small and frequently made bi-temporal changes in a highly collaborative setting, and to provide for historical and retrospective SPARQL (Update) queries. An example of a dataset with bi-temporal changes and many users working simultaneously is the basic key register of persons (Basisregistratie Personen (BRP)[1]). The BRP contains personal data of inhabitants of the Netherlands (residents) and of persons who have left the Netherlands (non-residents). Information from the BRP is used and maintained by a large number of organisations, including municipalities, the Tax and Customs Administration[2], UWV[3], etc. Since BRP both has current data (such as one's current address) and historical data (such as one's former addresses), a change has both a transaction time and a valid time. For instance, the registration date of a relocation might differ from the date the person actually moves. In addition, these changes could be extremely small, and many changes are made over time, as the BRP is continuously evolving due to deaths, childbirths, and relocations. In order to version such a dataset, and query a certain bi-temporal state in a collaborative environment, there is a need to represent, retrieve, and exchange these bi-temporal changes.

However, as discussed in Chapter 3 the storage strategy IC is fast for querying single versions, but not useful when many small changes are made. Furthermore, in a bi-temporal setting a version has both a transaction and valid time. To store each bi-temporal version separately we might end up having numerous copies of the dataset. The storage strategy FB faces the same problem, because it also stores its versions separately, and bi-temporally versioning would result in many N-Triples files. While, indexing triples and their time metadata makes versioning systems more query and space efficient, these timestamped triples are hard to exchange between Linked Data users, and hence not suitable in a very cooperative setting. Moreover, these TB approaches only associates to each triples its time or version-related metadata. Associating both time and version-related metadata could be complex. Therefore, we conclude that the change-based storage strategy is the most appropriate strategy for the following use cases. Since this strategy stores the differences between versions, *small and frequently made changes* (a) only result in a new change set (or update), which requires less storage space than a complete copy of the dataset, and does not lead to a version-related metadata for every modified triple. Moreover, both the

[1] https://www.rvig.nl/brp
[2] https://www.belastingdienst.nl
[3] https://www.uwv.nl

transaction time and the valid time (b) can simply be added to such an update without requiring a complex index technique. In addition, these updates remain exchangeable, while these index, and compressed version datasets are hard to exchange. Besides, a CB approach can also support other versioning operations more efficiently, such as branching or reversion of errors. These operations also enhance the *collaborative and decentralised design* (c). Although, doing *historical and retrospective SPARQL queries* (d) is not as efficient as for the IC strategy due to its version's construction time, still any state can be constructed by obtaining only the necessary updates. Lastly, the TB storage strategy requires a specialised triple-store to handle these indexing techniques(s). It would involve a higher complexity to attach these specialised version control functionalities to *existing Linked Data systems* (e) than it would for a change-based storage strategy.

Based on these use cases we have defined a number of requirements that Bi-VAKs should satisfy which we list in Section 4.1. Additionally, these requirements are related to existing studies on RDF versioning approaches, which we discuss in more detail in Section 4.2. In the subsequent sections we first define the basic versioning operations that Bi-VAKs supports, and then briefly introduce the basic functioning of Bi-VAKs.

4.1 Requirements of Bi-VAKs

In the following section, we describe the requirements that are relevant for our conceptual versioning approach Bi-VAKs. These requirements point out the main aspects of modelling bi-temporal revisions as well as aspects of supporting collaborating on a RDF dataset, and integrating the system with existing Linked Data tools. First we present the three major requirements that are directly implied by our aim to support bi-temporal change-based versioning (REQ 1 to 4). These major requirements are followed by eight requirements that define SPARQL query improvements (REQ 5 to 6); that are needed to support the collaboration process (REQ 7 to 9) respective and that are necessary to employ the system in a Linked Data context (REQ 10 to 12).

REQ1 *Bi-Temporal Change-based Versioning Model* For some scenarios accessing the historical values on a single time line is not enough. For these scenarios the time when it has been changed as well as the time it is valid are both relevant. For instance, in order to rectify an incorrectly received amount of salary both the incorrect and correct salary, and their time of recording are required. But, in addition to those changes to a RDF dataset, also the other versioning concepts, such as branching, merging, and tagging, might be dependent on a valid time and a transaction time simultaneously. Therefore, we need a bi-temporal versioning model for the changes to a RDF dataset and to the versioning system. However, the RDF data model itself is atemporal, and thus these bi-temporal changes should be modelled otherwise.

REQ2 *Triple Annotation Using RDF-star* Although it seems appealing to only store the changes made to a RDF dataset, there is still no standard structure to model these change sets. A change set consists of a collection of inserted and deleted quads, but RDF has no common approach to refer within another triple or quad to a modified triple/quad efficiently. Standard RDF reification, or named graphs are two common approaches, but in many ways these triple annotation approaches are not as efficient. Therefore, for describing these RDF dataset change sets RDF-star is a much better alternative. It improves comprehensibility, and it needs less storage space, because these modified triples can be described in less triple statements compared to the other approaches.

REQ3 *Historical and Retrospective SPARQL Queries* Assessing the Linked Data bi-temporally is of great importance to ensure the reliability of the dataset, and to derive valuable insights from it. For example, one wants to know which payment requests have already been recorded but which are not valid yet. In order to respond to these queries, it should be possible to access any version and the difference between versions of the dataset. Therefore, from all the historical and retrospective set of modifications, a temporary RDF dataset should be created to construct such a version. Specifically, we focus on querying bi-temporal version materialization (VM), delta materialization (DM), and version (VQ) basic triple patterns queries. These queries respectively concentrate on applying the query to a specific version; on obtaining the differences between versions for that query; or on returning the versions, for which the query gets a result.

REQ4 *Possibility of Modifying Change Sets* Unfortunately, no Linked Data user is free of making mistakes. It can often happen that a user made a change to the RDF dataset, but saw afterwards that this change is not correct, e.g. the valid time has been entered incorrectly, or a triple was deleted while it should not have been deleted. However, if this correction is considered as a new revision without creating a link between the two change sets, both change sets might be retrieved. Which may results in a wrong state of the RDF dataset. Therefore, it must be possible to modify change sets without both change sets ending up in a state incorrectly, and without the preceding change set being modified or removed.

REQ5 *Direct Access to Modified Triples* Obtaining a query result for the historical and retrospective SPARQL queries requires to construct (prior) version(s) by rewinding or fast-forwarding the bi-temporal change sets. Fortunately, we do not require all change sets to set up (prior) version(s), since we only need the change sets containing the triple pattern stated in the users' SPARQL query to return a query response. This approach speeds up the version construction time, and it improves the query process. Nonetheless, for this approach, we need a special algorithm that extracts the basic triple pattern from the users' query, and subsequently by means of SPARQL-star queries these change sets.

REQ6 *Interleaving Fully Materialised Versions* However, constructing every version by retrieving all the prior updates might be very time. By storing a interleaving fully materialised version (snapshot) of a state of the RDF dataset, Linked Data users could directly access this state without having to construct it first. Especially, if a particular state of the RDF dataset is queried often, it improves the performance.

REQ7 *Support Divergence* In a collaborative setting Linked Data users may differ in their motivation to contribute to a RDF dataset, for instance because of their different opinions or understandings of a certain subject. Therefore, it would be preferable to have multiple different versions of the dataset to express dissent, and diverge from the common dataset. Besides, organisational structures also affects the linear development of a Linked Data collection, for example if working groups discuss partial subjects. Therefore, the system needs to be able to support diverging states of a versioned RDF dataset. Contributors could divide from the current state of the dataset by creating a branch. In a branch they can freely modify the common dataset without actually altering it.

REQ8 *Track and Record Provenance* In addition to diverging states, also tracking provenance encourages collaboration. Data provenance (also referred to as data lineage) is one of the most important concepts for building trust in a dataset. Reliability is

necessary in a collaborative data engineering environment, because small changes to the data are performed by many different parties. Therefore, tracking and recording provenance is a basic requirements for any dataset version control system. To maintain the origin of the changes the system needs to track the predecessor relations for a versioned dataset; the contributing Linked Data user (author information); the change reason; and the date of change. Furthermore, the versioning system should also be extensible to include custom provenance information depending on the use case.

REQ9 *Ensure Exchangeability, Interoperability, and Integrity* In order to further enhance the collaboration, collaborators should be able to easily exchange their changes, primarily, in order to synchronise their datasets. The synchronisation of datasets is of great importance to keep datasets up to date, and prevent mistakes due to the use of an old dataset. To efficiently exchange changes, they require a common change format, because it let computer systems automatically interpret the information exchanged meaningfully and accurately. However, a common information exchange format does not directly ensure the integrity of a change against both accidental and malicious manipulation. Therefore, in order to verify the change's integrity, and to fully track its history, we should compute a cryptographically secure hash for each change, and also include the hash of their predecessors in order to prevent history manipulation.

REQ10 *Detect data changes through SPARQL Update Queries* The tracking of the changes provides the necessary basis for further analysis in versioning systems. Although many source code versioning approaches use a differencing algorithm to determine the changes between two consecutive versions, calculating changes between RDF datasets with these kind of algorithms has its limitations. It takes time to compare two datasets. The dataset should be presented in a RDF serialization format. And many differencing algorithms are line-based approaches. Therefore, a canonical representations of the data is needed to allow for a stable comparison (Arndt, 2020). Besides, in a bi-temporal setting against which state of the RDF dataset should the new version be compared to calculate the differences? Namely the previous version might have another valid time than the current version. To overcome these limitations, the data changes should be detected differently. For instance, directly through the SPARQL Update queries. Therefore, we should develop an algorithm that extracts the inserted and deleted triples/quads from an INSERT DATA or DELETE DATA update operation.

REQ11 *Standard Data Access Interface* Linked Data collaborators might already use tools to create and edit RDF knowledge graphs. They only may lack support for collaborative version control. However, it may involve a high complexity to attach these specialised version control functionalities to existing Linked Data tools, and, thus, users may consider not to use these systems (Arndt, 2020). In order to let users continue using their existing Linked Data tools our version control system should have a standard data access interface which can be accessed by these existing tools.

REQ12 *Support for multiple RDF Graphs* Handling multiple RDF graphs (i.e. RDF datasets) allows Linked Data users to organise their stored knowledge in separate structural parts. Therefore, multiple RDF graphs may solve various data management problems in RDF applications, such as tracking provenance of RDF data, access control, replication of RDF graphs, and separation of concerns. In addition, when existing Linked Data tools already use multiple RDF graphs, there is no need to transform the quad-store into a triple-store, and to merge multiple graphs into a single graph.

However, as it is not possible to annotate quads with RDF-star, we define another approach to support quads in Bi-VAKs.

4.2 Bi-VAKs' Relation with State of the Art

Although some of the requirements of Bi-VAKs are mainly based on our use cases, such as the historical and retrospective SPARQL queries. Most of these requirements are related to the state of the art versioning approaches discussed in Chapter 3. Zhang et al. (2019), for instance, added both the time information and the update count to a triple, and therefore they also recorded both the transaction and valid time. However, an update count and time information for each modified triple may be redundant, since all modifications in the same change set have the same values. In addition, Zhang et al. (2019) uses RDF reification, which is rather an inefficient way for triple annotation compared to RDF-star. Which is also the case for the change-based versioning approaches R&Wbase (Sande et al., 2013), R43ples (Graube, Hensel, and Urbas, 2014), and the approaches proposed by Frommhold et al. (2016) and Cassidy and Ballantine (2007). These systems used named graphs as triple annotation approach to store the added and deleted triples, which creates an extensive number of named graphs. Besides, to version a RDF dataset using named graphs may prevent using named graphs in the versioning approach itself.

In addition to representing bi-temporal changes, none of these state of the art approaches described an approach for doing historical and retrospective version materialisation (VM), delta materialisation (DM), and version (VQ) basic triple pattern queries. Most of these versioning systems, such as OSTRICH (Taelman et al., 2019), only supported historical VM, DM, and VQ basic triple pattern queries. Therefore, by supporting these queries, Bi-VAKs fills in an interesting gap. Moreover, RDF-star even makes it straightforward to directly retrieve the changes which are associated to the requested query, and hence might improve the version construction time. In addition, a different improvement technique that some of these approaches used is a snapshot (Meinhardt, Knuth, and Sack, 2015; Taelman et al., 2019). Therefore, only the change sets between the snapshot and the requested version are need to construct a version. To allow our users to take advantage of this improvement, we also decided to create an interleaving fully materialised version (snapshot). However, in Bi-VAKs a snapshot relies on both a valid time and transaction time. A requested version and a snapshot can still be far apart, even if their transaction time is about the same.

Besides, so far there is relatively more research on the collaboration aspect of RDF version control systems. Many of these systems already supported branching, and tagging, and track and recorded provenance information of a change. For example, R43ples (Graube, Hensel, and Urbas, 2014), and R&Base (Sande et al., 2013) both used or extended the PROV-O ontology to specify and to store the provenance information, such as the author and commit message. And they created diverging states by adding a branch identifier to the change set. Moreover, a realisation of these systems as a SPARQL proxy between the Linked Data users and an existing SPARQL endpoint also enhanced cooperation (Graube, Hensel, and Urbas, 2014; Frommhold et al., 2016; Arndt et al., 2019). A standard data access interface provides Linked Data users access, while they might use different RDF editors, and readers. Even some research (Frommhold et al., 2016) detected the changes from a standard SPARQL Update query. However, in many related work it is not common to modify an existing change, despite the fact that users could make mistakes. They probably solved such a mistake by creating a new change, but it does not establish a link between two changes. In addition, only some systems (Frommhold et al., 2016)

prevented a change from unperceived history manipulation by computing a hash over a change, and only some systems (Graube, Hensel, and Urbas, 2014; Frommhold et al., 2016) had quad support.

4.3 General Versioning Operations

In this section we give a formal definition of the RDF dataset versioning model, and we define the main versioning operations in general, which we will describe in-depth in Chapter 5. This foundational formal model is an extension on the formal model described by Arndt et al. (2019) and Pelgrin, Galárraga, and Hose (2021). Thus far Pelgrin, Galárraga, and Hose (2021) have defined a RDF dataset versioning model intuitively as a temporally-ordered collection of states a RDF dataset has gone through since its creation. A RDF dataset goes into a new state when a Linked Data contributor makes at least one change to at least one graph in the RDF dataset. While such a 'update' versioning operation always leads to a new state of the RDF dataset, other versioning operations do not. By applying these versioning operations our versioning systems creates a new revision (r) or version, but not specifically a new version of the RDF dataset. Nonetheless, these revisions are all stored in the so called *revision-store* (R), and thus applying a versioning operation (*Apl*) results in a new revision r_j, which leads to a new state j of the revision store (R_j). Apart from being temporally-ordered based on their transaction time j, these revisions might have a valid time, and thus should also be temporally-ordered based on their valid time i. In other words, each state of the revision-store ($R_{i,j}$) is on two time lines i and j simultaneously, and thus we define our RDF dataset versioning model as follows:

Definition 1 (RDF dataset versioning model)**.** *A RDF dataset versioning model is a bi-temporally-ordered collection of states of the revision-store ($R_{0,0}, \ldots, R_{i,j}, \ldots$) that a revision-store has gone through since its creation ($i = 0, j = 0$). A revision-store R goes from its ancestor state $R_{i-1,j-1}$ into a new state $R_{i,j}$ when a new revision $r_{i,j}$ is created by having a Linked Data user perform one of the following versioning operations $R_{i,j} = Apl(r_{i,j}, R_{i-1,j-1})$: update, revert, snapshot, merge, branch, and tag.*

Figure 4.1 illustrates three revisions (A, B, C) that are for convenience on a single time line. The first versioning operation Apl_1 changes both the state of the RDF dataset, and the state of the revision-store, but the second versioning operation Apl_2 only changes the state of the revision-store. As showed in Figure 4.1, a revision refers to its previous revision, which in turn also refers to its ancestor. Therefore, the revision-store is mainly a (non-)linear chain of revisions created by performing the following versioning operations: *update* to record the changes, *snapshot* to store a full state of the RDF dataset, *branch* to support diverging states, *tag* to specifically refer to a revision, and *revert* to undo revisions, and *merge* to synchronise updates and branches with one another.

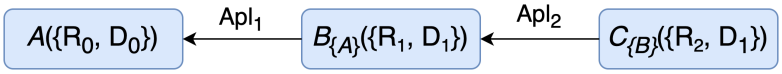


FIGURE 4.1: Illustration of bringing the revision-store into a new state.

4.3.1 Update Operation

The update operation is the fundamental versioning operation that applies an update to a state of the RDF dataset D_{j-1} in order to bring the dataset into a new state D_j, which we formally define as follows:

Definition 2 (Updating)**.** *Updating is the progress of moving an arbitrary state of the RDF dataset to a new state by changing at least one triple in at least one graph in the RDF dataset.*

In other words, a RDF triple is the smallest manageable piece of knowledge, which can only be added and deleted. Each addition and deletion turns the dataset into a new state, and thus the history of changes could be defined as a sequence of states, as well, as a sequence of updates. A RDF dataset in state i (D_i) always has a single default graph G_i^0, and 1 to m named graphs: $D_i = \{G_i^0, \ldots, G_i^m\}$. The dataset update $U_j = \{\hat{u}_j, u_j^0, \ldots, u_j^m\}$ consists of a collection of inserted and deleted triples for each graph, $u_j^i = \langle u_j^{i+}, u_j^{i-} \rangle$, and a graph update $\hat{u}$, which inserts or deletes complete RDF graph(s). Hence, as Pelgrin, Galárraga, and Hose (2021) also described, we can obtain a RDF dataset $D_j = U_j(D_{j-1})$ from D_{j-1} by (i) applying the individual updates $u_j^i(G_{j-1}^i) = (G_{j-1}^i \cup u_j^{i+}) \setminus u_j^{i-}$ for each graph $0 \leq i \leq m$, (ii) removing the graphs in $\hat{u}_j^-$, and (iii) adding the graphs in $\hat{u}_j^+$.

Besides, we can also rewind a dataset and obtain RDF dataset D_{j-1} from D_j by swapping the inserted and the deleted quads in the update. Hence, we get $D_{j-1} = U_j^{-1}(D_j)$ by (i) applying the individual updates $(G_j^i)u_j^i = (G_j^i \cup u_j^{i-}) \setminus u_j^{i+}$ for each graph $0 \leq i \leq m$, (ii) removing the graphs in $\hat{u}_j^+$, and (iii) adding the graphs in $\hat{u}_j^-$. However, in order to rewind a dataset, an update must be invertible (U_j^{-1}), and therefore the update U_j may only consist of the inserted quads, and added graphs which were not in the dataset at state $j-1$, and deleted quads, and graphs that were in the dataset at state $j-1$.

Figure 4.2 illustrates an example of two consecutive dataset states D_0 and D_1. D_0 is a dataset with graphs $\{G_0^0, G_0^1\}$. The dataset update U_1 generates a new dataset state D_1. U_1 consists of three changes. u_1^0 that modifies the default graph G^0. u_1^1 that modifies the named graph, and u_1^2 that inserts a second empty named graph. Note that U_1 is not invertible, because ⟨:Dinner, a, :Meal⟩ is not in D_0. Therefore, we cannot obtain D_0 from D_1, because it would insert ⟨:Dinner, a, :Meal⟩ to D_0.

D_0	U_1	$D_1 = U_1(D_0)$
$G_0^0 = \{\langle$:Vegan, a, :Diet $\rangle$,	$\hat{u}_1^2 = \{\hat{u}_1^{2+} = \{G^2\}, \hat{u}_1^{2-} = \emptyset\}$	$G_1^0 = \{\langle$:Vegan, a, :Diet $\rangle$,
$\langle$:GlutenFree, a, :Diet $\rangle\}$		$\langle$:LactoseFree, a, :Diet $\rangle\}$
$G_0^1 = \{\langle$:Lunch, a, :Meal $\rangle\}$	$\hat{u}_1^0 = \{\hat{u}_1^{0+} = \{\langle$:LactoseFree, a, :Diet$\rangle\}$,	$G_1^1 = \{\langle$:Lunch, a, :Meal$\rangle$,
	$\{\hat{u}_1^{0-} = \{\langle$:GlutenFree, a, :Diet$\rangle\}$	$\langle$:Snack, a, :Meal$\rangle\}$
	$\hat{u}_1^1 = \{\hat{u}_1^{1+} = \{\langle$:Snack, a, :Meal$\rangle\}$,	$G_1^2 = \emptyset$
	$\{\hat{u}_1^{1-} = \{\langle$:Dinner, a, :Meal$\rangle\}$	

FIGURE 4.2: A dataset with two dataset states D^0, D^1. The first state contains two graphs, the default graph G^0 and G^1. The dataset update U_1 (i) modifies G^0, (ii) modifies G^1, and (iii) creates a new empty graph G^2

Consequently, Pelgrin, Galárraga, and Hose (2021) has modeled these aforementioned dataset updates as a set of 5-tuples $\langle s, p, o, \rho, \zeta \rangle$. Here, $\langle s, o, p, \rho \rangle$ respectively corresponds to the subject, predicate, object, and context information of a quad, and ζ symbolises the state of the RDF dataset. However, our versioning approach should manage a bi-temporal RDF dataset. An update has both a valid and transaction time, and hence brings the RDF dataset into a new state $D_{l,j}$, which is in a two-dimensional space of time. Therefore, we model such an update as a set of 6-tuples $\langle s, p, o, \rho, \zeta, \psi \rangle$, where ζ and ψ together represent the state of the RDF dataset indicating the moment in the transaction time and valid time respectively. A dataset update ($U_{l,j}$) now consists of a collection of inserted and deleted quads, which has both a valid time l and transaction time j.

In order to obtain the bi-temporal RDF dataset $D_{l,j}$ from $D_{l-1,j-1}$, we (i) apply the bi-temporal individual updates $u^i_{l,j}(G^i_{l-1,j-1})$, (ii) remove the graph in $\hat{u}^-_{l,j}$, and (iii) add the graphs in $\hat{u}^+_{l,j}$.

4.3.2 Snapshot Operation

While a change-based versioning approach has many benefits over other approaches, it still requires to first construct a particular (prior) state of the revision-store R_{i-1}, before it can be queried. Therefore, to lower the construction time, Bi-VAKs stores some interleaving fully materialised version of the complete RDF dataset D_j. We call such as interleaving materialised version a *snapshot*. A database *snapshot* or state is a read-only copy or view of the database at a certain point in time. Since our revision-store considers both valid time and transaction time, our complete RDF dataset states are also two-dimensional. Therefore, we define the 'snapshot' versioning operation as follows:

Definition 3 (Snapshotting). *A snapshot represents a state of the RDF dataset at a particular moment in a two-dimensional space of time: $D_{i,j}$.*

In other words, a snapshot contains a read-only copy of the triple/quad-store, which has been made at time t_i and t_j. A snapshot only modifies the revision-store, but it does not change the RDF dataset itself.

4.3.3 Branch Operation

In order to let contributors diverge from the common state of the dataset Bi-VAKs supports the creation of branches. A branch is a split off either from the common state, also called the main stream, as depicted in Figure 2.2, or from another branch. As a revision only refers to its predecessor, we can simply create a branch by adding another revision, which refers to the same predecessor as the revision from the main stream. Figure 4.3 shows that both revision $E_{\{B\}}\{(R_3)\}$, and revision $C_{\{B\}}\{(R_2)\}$ refer to the same predecessor $B_{\{A\}}\{(R_1)\}$. Revision $E_{\{B\}}$ creates now a new branch or fork based on the revision $B_{\{A\}}$. Hence, letting revisions referring to similar ancestors results in a directed rooted in-tree, which is portrayed in Figure 4.3.

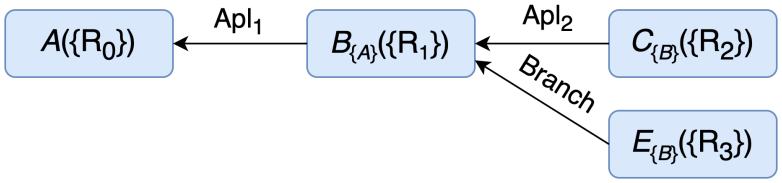


FIGURE 4.3: Two branches evolved from a common revision.

Although, we know that $R_1 \not\approx R_2$, and $R_1 \not\approx R_3$, we do not know anything about the relation between R_2 and R_3, and might regard them as independent. Based on the formal syntax described by Arndt et al. (2019), we define the 'branch' versioning operation as follows:

Definition 4 (Branching). *Branching is the (independent) evolution of the revision-store R with two revision-stores R_1 and R_2 as result, where $R_1 = Apl(r_1, R)$ and $R_2 = Apl(r_2, R)$. The revisions r_1 and r_2 might be unequal, but can be the same. The same applies for R_1 and R_2, they can be different after the independent evolution, but can be similar as well.*

We define a branch by adding the branches' IRI as a node to the head revision: the latest revision in the revision graph. The head revision of the main stream does not have the 'branch' attribute.

4.3.4 Merge Operation

As portrayed in Figure 4.3, after creating a second branch, the tree of revisions is diverged. In order to synchronise these diverged states we want to merge these branches, especially to put the changes from the branch into the main stream. Similar to Arndt et al. (2019), we define the notation of the merge as follows:

Definition 5 (Merging of two revision-stores)**.** Given the two revision $C_{\{B\}}(\{R_1\})$ and $E_{\{B\}}(\{R_2\})$, merging the two revision-stores R_1 and R_2 with respect to the revision history expressed by the revisions C and E is a function

$$Merge(C(\{R_1\}), E(\{R_2\})) \quad = \quad Merge_{C,E}(\{R_m\})$$

The *Merge* function takes two revision-stores as arguments and creates a new revision-store R_m. However, only the actual changes to the RDF dataset from the split-off are merged. The other versioning operations, such as tags, reverts, and snapshots, are not merged, and they cannot be retrieved from the main stream or branch from which the merge operation was performed. Therefore, the merge revision $M_{C,E}(\{R_m\})$ only refers to its own preceding revision, and not to the latest revision in the branch, as is demonstrated in Figure 4.4. In addition, our revision-store remains a directed rooted tree, instead of an acyclic directed graph as in other version control systems (Arndt et al., 2019).

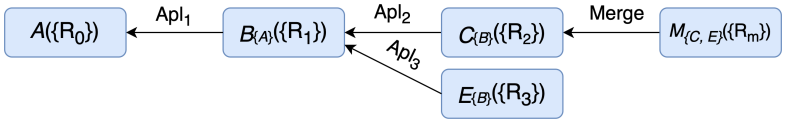


FIGURE 4.4: Merging a branch and the main stream.

Note that the definition does not make any assumptions about what the new revision-store R_m will look like. It depends on the actual implementation of the *Merge* function. Different merge strategies can produce different results, thus it is possible to have multiple merge revisions with different resulting revision-stores, but it merges the same revision-stores. We did not implement the merge function.

4.3.5 Tag Operation

In addition to branching and merging, Bi-VAKs also provide for the creation of a tag, which allows Linked Data users make explicit references to a specific state of the RDF dataset $D_{i,j}$. The 'tag' versioning operations is hence defined as follows:

Definition 6 (Tagging)**.** *Tagging is the operation of creating an explicit reference to a state of the RDF dataset* $D_{i,j}$.

Figure 4.5 gives an illustration of a tag, which refers to the revision-store state R_1. Similar to the 'snapshot' versioning operation, a tag only leads to a new state of the revision-store, and does not affect the RDF dataset.

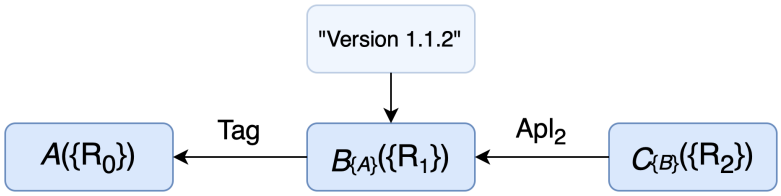


FIGURE 4.5: A revision creating a tag for state R_1.

4.3.6 Revert Operation

Although Linked Data contributors can express their disagreement via branches, it still possible that they make a mistake, and thus Bi-VAKs supports the reversal of revisions. Bi-VAKs reverts a revision r_i by creating a new revision r_{i+1} that removes revision r_i from the coming states of the revision-store, so r_i still exists in the preceding states of the revision-store ($\leq i$), but not in the succeeding states ($> i$). We formally explain the 'revert' versioning operation as follows:

Definition 7 (Reverting)**.** *Reverting a revision r_j in state j removes its existence in the subsequent states of the revision-store ($R_{j+1}, \ldots$) by calculating the inverse of revision r_j and applying it to the revision-store at state j.*

As illustrated in Figure 4.6, reverting revision $B_{\{A\}}(\{R_1\})$ is done by creating an inverse revision $B^{-1}_{\{B\}}(\{R_0\})$ such that revision $B_{\{A\}}$ does not exist in the next state of revision-store. Besides, since $B^{-1}_{\{B\}}$ reverts its parent revision, the state of the revision-store in the last revision R_0 is again equal to the state of the revision-store in the first revision (A). In order to obtain R_0, we need to compute the inverse revision $B^{-1}_{\{B\}}$, and apply it to R_1, which is in this example the inverse difference between R_0, and R_1. In Bi-VAKs, Linked Data contributors can revert all revisions in revision chain. However, (i) it depends on the versioning operation how the inverse revision is specified, and (ii) we do not take into account the inconsistencies a revert might create in RDF dataset itself.

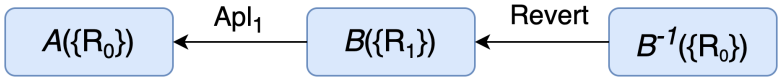


FIGURE 4.6: A revision reverting the previous revision.

4.4 Introduction to Bi-VAKs

In this section we introduce our prototypical Bi-Temporal Versioning Approach for Knowledge graphs (Bi-VAKs). Figure 4.7 gives an overview of Bi-VAKs. A Linked Data user could either send a SPARQL (Update) query or another versioning operation request to Bi-VAKs, which we have introduced in Section 4.3. Bi-VAKs extracts from both a SPARQL Update query and a versioning operation its information, and it forms a single transaction revision and (multiple) valid revision(s) depending on the users' request. This division of a change into a transaction revision and its corresponding valid revision(s) is a way of structuring revisions or versions in order to indicate their transaction time and their validation time respectively. These revisions are subsequently stored in a standard quad-store that can be reached by a SPARQL endpoint. As showed in Figure 4.7, we call this quad-store the *revision-store*, which forms a graph of revisions. In addition to a SPARQL update query,

a Linked Data user can also do historical and retrospective version materialisation (VM), delta materialisation (DM), and version (VQ) basic triple pattern queries by sending an SPARQL query with some corresponding time information to Bi-VAKs. Bi-VAKs returns a response by first constructing the requested version(s) either from its revisions in the revision-store or from an interleaving fully materialised version of the dataset in the data store(s). Therefore, Bi-VAKs consists of two main components.

The first component explains the conceptual design of our versioning approach that mainly focuses on how the revisions in revision-store are structured. It describes how these valid and transaction revisions are defined for different versioning operations. Furthermore, it explains three different strategies how a transaction revision is connected to its corresponding valid revision(s). The second component accounts for the interaction between the Linked Data users, the revision-store, and the data store(s) to allow for bi-temporal SPARQL (Update) queries. It explains how a change is detected from a SPARQL Update query; how these changes are represented in RDF-star; and how these changes are retrieved by means of SPARQL-star in order to constructed a version, and return a VM, DM, and VQ query result. We discuss these components in more detail in respectively Chapter 5 and 6. Subsequently in Chapter 7 we give a concise explanation of the actual implementation of these components.

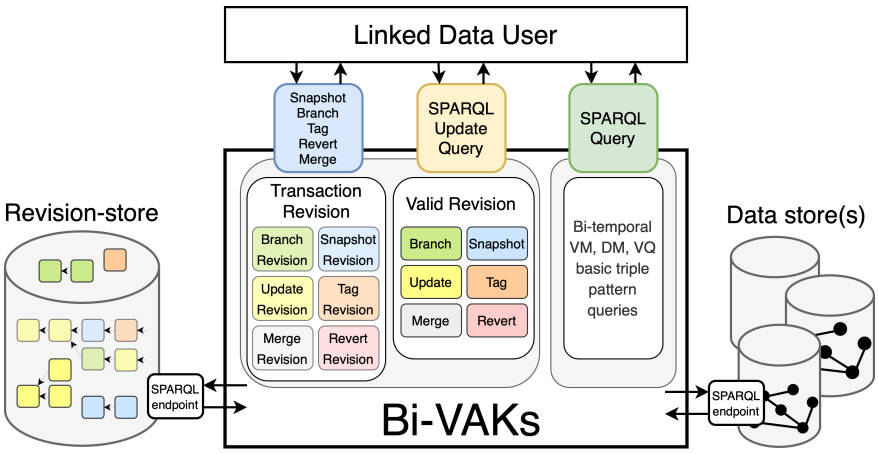


FIGURE 4.7: Overall architecture of the prototypical Bi-Temporal Versioning Approach for Knowledge graphs (Bi-VAKs).

Chapter 5

Representing Revisions in Bi-VAKs

In this chapter we elaborate on the conceptual design of our change-based versioning approach, Bi-VAKs, for which we have introduced its concepts and requirements in Chapter 4. Its conceptual design mainly concentrates on how the revision-store is structured, and the revisions are represented in order to fulfil the need for a bi-temporal change-based versioning model in a collaborative setting. A main aspect of our design is that we create for each versioning operation a single transaction revision and (multiple) corresponding valid revision(s). For example, by applying an 'update' versioning operation, Bi-VAKs creates a *Update* (valid revision) and a *Update Revision* (transaction revision). We primarily make this distinction between these two revisions to respectively express the valid time and the transaction time of a single version. Because, not only the dataset updates are bi-temporal, e.g. a change of address that is valid only at time i, but was entered at time j ($i > j$), also other versioning operations are bi-temporal, e.g. a tag represents a state of the RDF dataset both at time i and j ($i \neq j$). Hence each versioning operation brings the revision-store into a new bi-temporal state. However, in addition to this specification of the valid and transaction time, Bi-VAKs also has a number of other requirements that affects the representation of the revisions. In Bi-VAKs users should be able to modify an existing change, to diverge from the common RDF dataset, to track its provenance data, and to ensure changes their exchangeablity, interoperability, and integrity. We begin this chapter by first describing how these revisions are represented in general in order to avoid repetition (Section 5.1). Subsequently, in Section 5.2 we give a detailed description of the different types of transaction and valid revisions for the various versioning operations.

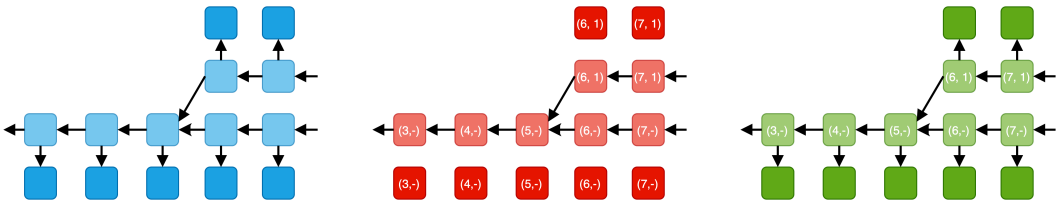
5.1 Transaction & Valid Revisions

In addition to expressing transaction time and valid time, the division between *Transaction Revisions* and *Valid Revisions* has many other advantages. First, (I) they could serve other purposes. The transaction revisions might contain the metadata or provenance information of a change, such as the author or the reason of the change, while the valid revisions may include the data that really had been changed. In other words, the transaction revisions only indicate when a change happened, whereas the valid revisions really indicate what happened. Second, (II) sometimes the transaction time of a valid revision must remain the same. A valid revision can refer to another transaction time than the time it was created. Besides, if we did not separate the revisions, (III) a modification to a valid revision is cumbersome. In addition to referring to their predecessor, the revisions must also refer to other revisions in the non-linear chain of revisions. Furthermore, (IV) user might want to create multiple valid revisions at the same transaction time. These valid revisions cannot be combined, because they could have a different valid time. And

we cannot put them in a row of transaction revisions, because that would imply a certain order. Finally, (V) it facilitates the retrieval of revisions separately. However, due to these reasons, the appearance of a valid revision differs from the appearance of a transaction revision. Therefore, in this section we give an overall and distinct description of the transaction revision (Sub-section 5.1.2), valid revision (Sub-section 5.1.3), their connection (Sub-section 5.1.1), and their unique identifier (Sub-section 5.1.4).

5.1.1 Connection between a Transaction Revision & their Valid Revision(s)

Although we separate these revisions, they must still be linked. To connect the transaction revision with its corresponding valid revision(s), we develop three referencing strategies, which are illustrated in Figure 5.1. The first naive reference strategy let the transaction revision explicitly refer to the valid revision(s), and it uses the direct references to the antecedent transaction revisions and to the valid revisions to obtain the requested revisions. We call this strategy the explicit reference strategy, and it is represented in Figure 5.1a. However, retrieving revisions via their antecedent reference may be slow in practise. The second strategy implicitly let the transaction revision refer to its valid revision(s) by an equal revision number and branch index. This strategies uses these numbers to obtain the requested revisions. We call this strategy the implicit reference strategy, and it is displayed in Figure 5.1b. Although, these numbers speed up the retrieval of revisions, the valid revisions now contain some kind of metadata. It might complicate the exchange of revisions, and two different transaction revisions can no longer point to the same valid revision. Therefore, we came up with a third strategy which is a combination of these two strategies. It let the transaction revision explicitly refer to its valid revision(s), but it obtains the requested revisions by using a revision number and branch index. We call this strategy the combined reference strategy, and it is showed in Figure 5.1c.



(A) Explicit (B) Implicit (C) Combined

FIGURE 5.1: The three different reference strategies.

5.1.2 Transaction Revisions

The *Transaction Revision* indicates the transaction time of a change to our versioning approach Bi-VAKs. Furthermore, it contains some metadata of each versioning operation, such as the author, the reason and the creation date. Figure 5.2 illustrates the key attributes of a *Transaction Revision*, but it is still a generic solution, and it can be extended for arbitrary domains. Whereas a square indicates a literal, an eclipse indicates an IRI. The text that is written on the arrow points to the predicate or property name of the transaction revision. The imaginary namespace IRI that is used for Bi-VAKs is `http://bi-vaks.org/vocab/`. Each *Transaction Revision* has its own unique IRI, and hence is a node in the RDF revision-store. Its IRI is a combination of the type of transaction revision, e.g. `http://bi-vaks.org/vocab/UpdateRevision_` and the hexadecimal of its hash. Depending on the applied versioning operation, it has one of the following

RDF types, which all are a subclass of the RDF class *Transaction Revision*: *Update Revision*, *Initial Revision*, *Branch Revision*, *Snapshot Revision*, *Merge Revision*, *Tag Revision*, and *Revert Revision*. We explain these different transaction revision types in more detail in Section 5.2.

As stated in Requirement 7, users should be able to diverge from the common RDF dataset (branching). To specify whether a *Transaction Revision* belongs to a certain branch, it depends on which reference strategy we use. For the explicit reference strategy only a branch revision and the head(s) of the graph of transaction revisions explicitly refer to a branch. For the implicit and combined reference strategy each transaction revision has a branch index that refers to a particular branch in the revision-store. These transaction revisions, therefore, form a directed rooted tree, which is illustrated in Figure 5.3. A *Transaction Revision* always refers to a single preceding transaction revision except the first revision, also called the *Initial Revision*, and multiple revisions can refer to the same preceding transaction revision when a branch is created. Such a non-linear chain of revisions shows in a glance which revisions were created before or after the others, and thus indicates the transaction time sequence. In addition to the IRI of its preceding transaction revision, a *Transaction Revision* always includes some provenance information. Adding provenance data is a main requirement for version control systems as stated in Requirement 8 in Section 4.1. Nevertheless, for simplicity, a transaction revision only contains the followings provenance information: the author, a description, and the creation date. They all are described as a RDF literal, as symbolised in Figure 5.2. But, due to its generality, it would be possible for Linked Data users to add more provenance data to it.

In addition to provenance information, Requirement 9 stated that we must ensure the integrity of a revision. Therefore, we include the SHA-256 value as an attribute to each revision, but we mainly use it to define the unique identifier of each revision. A hash of a revision is unique for the information included in the revision, as described in Section 5.1.4 in-depth. Finally, as demonstrated in Figure 5.2, the last attribute of a transaction revision depends on the revision referencing strategy as described in Sub-section 5.1.1. A transaction revision can explicitly refer to the valid revision by including its IRI to the transaction revision, or implicitly via its revision number and branch index. And, it can also combine these strategies. The property name of a valid revision for the explicit or combined reference strategy depends on the type of the valid revision. For instance, if a valid revision is an *Update*, the property is called 'update'. In summary, for the explicit strategy the transaction revision includes the valid revision(s); for the implicit strategy the transaction and valid revision include the revision number and branch index; and for the combined reference strategy, the transaction revision includes both the revision number and branch index, and the valid revision(s).

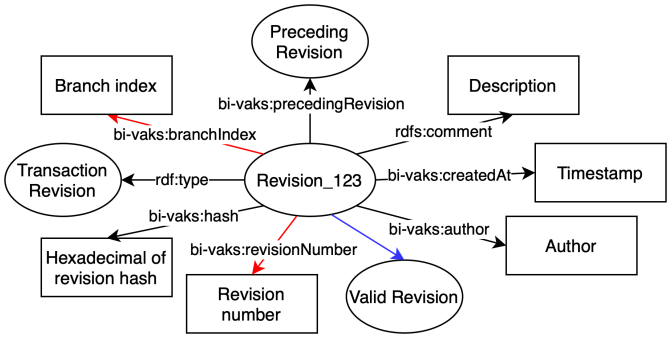


FIGURE 5.2: *Transaction Revision*

5.1.3 Valid Revisions

The *Valid Revision* indicates the valid time of a change to our versioning approach Bi-VAKs. And, it contains the actual modified data to Bi-VAKs. Similar to a *Transaction Revision* a *Valid Revision* is also a RDF resource in the revision-store with its own unique identifier. This IRI is a combination of the type of valid revision, e.g. `http://bi-vaks.org/vocab/Update_`, and the hexadecimal of the SHA-256 value of the revision, which is further described in Section 5.1.4. Therefore, if the revisions' SHA-256 value is completely the same as hash value of another valid revision, they contain exactly the same information. Additionally, it ensures that no information in the revision is changed, because once the hash is created it cannot be changed. Comparable to the *Transaction Revision*, for all different versioning operations we create a distinct *Valid Revision*. However, these types of *Valid Revisions* only have a few attributes in common. They all include their SHA-256 value, their RDF type, and their preceding valid revision. Furthermore, if a transaction revision implicitly refers to its valid revision(s), we also add the revision number and branch index to each valid revision in order to know to which transaction revision the valid revision belongs.

Nevertheless, a *Valid Revision* only has a preceding *Valid Revision*, if it modifies, reverts or merges another *Valid Revision*. To fulfil Requirement 4, Bi-VAKs offers the possibility of modifying *Valid Revisions* directly instead of removing them and subsequently adding them again. In this way we create less revisions, and we keep track of these revisions being related. Moreover, we do not require to change an existing valid revision, even when we revert it. Therefore, the SHA-256 value will never change, and we do not lose any information of the revisions. However, it is not allowed to modify all attribute in each valid revision. The RDF type, preceding revision, the SHA-256 value, the revision number, and branch index should never be modified. For the other attributes specific to the revision type it must always be possible that they could be altered, which we describe for each specific type in more detail in Section 5.2.

5.1.4 Unique Identifier

Since each change to the graph of revisions results in a new and different revision, we must uniquely define a revision, and thus specify a unique identifier for each revision. Therefore, we compute a cryptographically-strong hash identifier, SHA-256, over all information in a revision. Such a SHA-256 value is unique, when the data in a revision already differ in a single aspect from another revision. The author, description or branch, for example, may overlap, but their combination, the relation to its preceding revision, and revision number or reference(s) to the valid revision(s) most probably do not. Furthermore, since the hash value is completely unique due to the difference in information, a SHA-256 hash also ensures that nothing has been altered in the revision. It even make sure that the history of revisions do not change if the revisions are linked to their preceding revision(s). Namely, if something has been altered, e.g. its preceding revision, we would obtain another SHA-256 hash value for that revision. However, in order to verify that nothing in a revision has been altered, its SHA-256 hash has to be computed from a single, standard format. But the same information in a revision can already be expressed in a variety of different ways. Therefore, we need a canonicalization algorithm that transforms an input RDF dataset to a normalized RDF dataset, so if any two input datasets contain the same information regardless of their arrangement, they will be transformed into identical normalized dataset (Longley, 2021). Since the revisions only contain nodes which have globally-unique identifiers (no blank nodes), we can use a simple canonicalization algorithm. This algorithm first sets the temporal resource identifier of a revision

to http://bi-vaks.org/vocab/TemporalRevision. Then it serialises each triple or quad to N-Triples (Carothers and Seaborne, 2014) or N-Quads format (Carothers, 2014), and sort these ntriples, or nquads in lexicographical order. The hash value results from passing the sorted ntriples, or nquads through the hash algorithm. Listing 5.1 illustrates an example of a branch revision in N-Triples format sorted in lexicographical ordering.

LISTING 5.1: Representation of a *Branch Revision* in N-Triples

```
<http://bi-vaks.org/vocab/TemporalRevision> <http://www.w3.org/1999/02/22-rdf-syntax-ns#
    type> <http://bi-vaks.org/vocab/BranchRevision> .
<http://bi-vaks.org/vocab/TemporalRevision> <http://bi-vaks.org/vocab/author> "Lisa
    Meijer" .
<http://bi-vaks.org/vocab/TemporalRevision> <http://bi-vaks.org/vocab/branch> <http://bi
    -vaks.org/vocab/Branch_48ak32> .
<http://bi-vaks.org/vocab/TemporalRevision> <http://bi-vaks.org/vocab/createdAt>
    "2021-06-19T13:34:00+02:00"^^xsd:dateTimeStamp .
<http://bi-vaks.org/vocab/TemporalRevision> <http://bi-vaks.org/vocab/precedingRevision>
     <http://bi-vaks.org/vocab/UpdateRevision_3k8yqcd4k92> .
<http://bi-vaks.org/vocab/TemporalRevision> <http://www.w3.org/2000/01/rdf-schema#
    comment> "Create new branch called LisaBranch."@en-gb .
```

5.2 Types of Transaction & Valid Revisions

As we already mentioned in Section 4.4, we consider different types of transaction and valid revisions. In Figure 5.3 we give a conceptual visualisation of an extremely small example of a revision-store. Since these transaction revisions always refer to their preceding revision, they form a directed rooted tree of revisions. In Figure 5.3 every colored square indicates a particular transaction or valid revision. The dates beneath and above each square are the creation date of the transaction revision, and the start date and end date of a valid revision. The numbers within each square represent the revision number of each transaction revision, which corresponds in this case to the creation dates, and they indicate the transaction time sequence. Moreover, in this figure these numbers also indicate to which valid revision a transaction revision refers. As demonstrated in Figure 5.3, the chain of transaction revisions always starts with the *Initial Revision*. We refer to this start chain as the main stream. The *Initial Revision* can subsequently be followed by any other type of transaction revision except a *Merge Reversion*. When a Linked Data user wants to branch off from the common RDF dataset, as depicted in Figure 5.3 by *Branch Revisions* (4) and (8), Bi-VAKs creates a *Branch Revision* that refers to the transaction revision from which it derives. Thus, only the connection between *Branch Revisions* and their predecessor has any real meaning, while between other transaction revisions their link is purely for time ordering. In addition, the *Head Revisions* refer to the latest transaction revisions in the revision-store, which are illustrated by transaction revision (12), (14), and (16) in Figure 5.3.

As we stated above, each transaction revision is associated with a valid revision. We connect a transaction revision to a valid revision either by letting a transaction revision explicitly refer to the valid revision itself, or by adding the revision number and branch index to the valid revision. In the following sub-sections we discuss each type of transaction and valid revision in more detail. For each type of valid revision we give a similar schematic visualisation of its content as illustrated in Figure 5.2, and we visualise a schematic diagram how revisions are related. Figure 5.4a, for instance, illustrates such a diagram of a *Update (Revision)*. The arrow between two entities means that one entity relates to the other, but not vice versa. The line means that they both relate to one another. The two numbers (e.g. 0..1) represent their cardinality ratio. These numbers respectively specify the minimum and maximum number of relationship instances that an entity relates to.

As illustrated in these figures, each transaction revision always refers to a single preceding revisions except the initial revision. And it can be referred by multiple succeeding revisions to support diverging states. Although Git[1] let a revision also refer to multiple preceding revisions, especially to symbolise a merge of two branches, we only let a transaction revision refer to a single preceding revision. In this way we do not require to add the updates from the main stream to the merged branch. Thus, in Bi-VAKs it is not possible to construct a prior version in a merged branch backwards, and our transaction revision graph forms a directed rooted tree instead of a directed acyclic graph. Valid revisions, on the other hand, can form a directed acyclic graph.

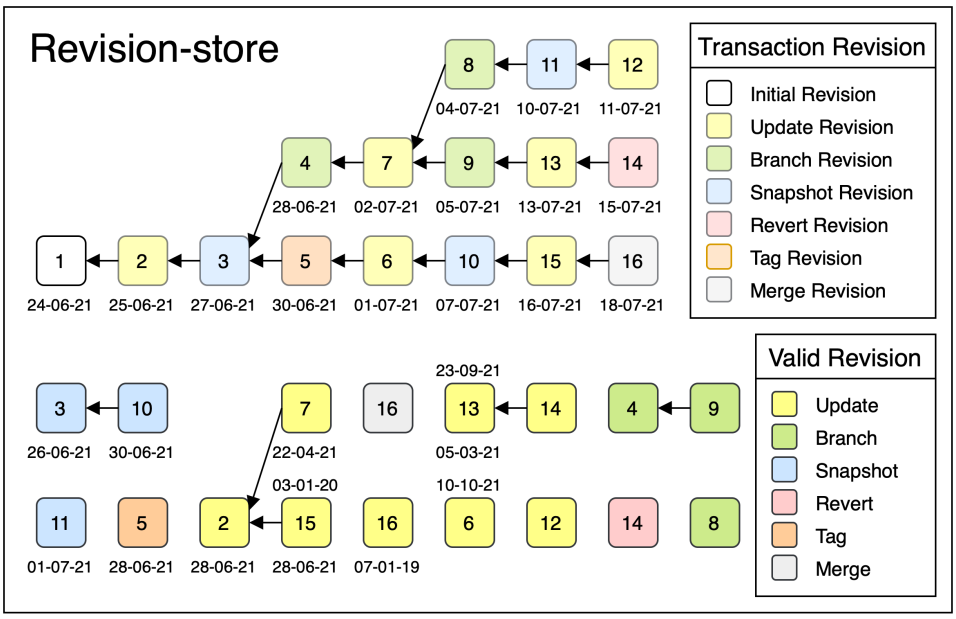


FIGURE 5.3: A conceptual illustration of a revision-store.

5.2.1 Initial Revision

The *Initial Revision* is the first transaction revision in the non-linear chain of revisions. If a Linked Data user already works with a RDF dataset, this revision initialises an *Update* containing all quads in this dataset. If a Linked Data user starts an empty revision-store, the *Initial Revision* does not refer to any *Valid Revision*. It only consists of the provenance data why, when, and who started this revision-store.

5.2.2 Update (Revision)

The *Update* and *Update Revision* are the most elemental revisions. They represent the actual change to the RDF dataset, and bring the RDF dataset into a new state $D_{i,j}$. In Bi-VAKs, a Linked Data user can modify the state of the RDF dataset in the followings ways:

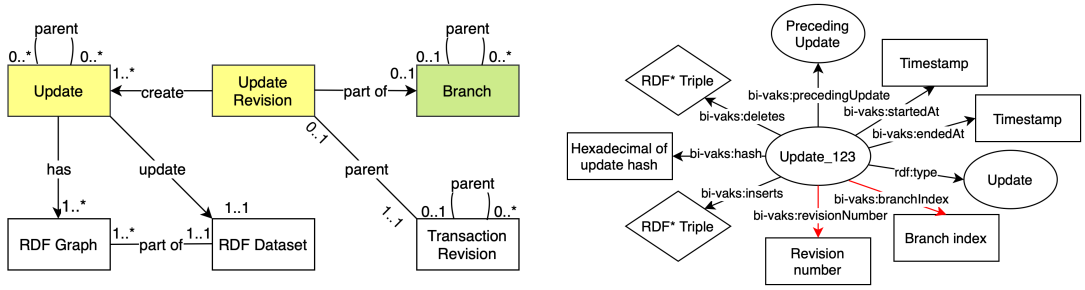
1. A Linked Data user can send a SPARQL update query to Bi-VAKs.
2. A Linked Data user can dump a RDF formatted file to Bi-VAKs.
3. A Linked Data user can modify an existing Update.

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

4. A Linked Data user can merge two or more Updates.
5. A Linked Data user can revert an Update Revision.
6. A Linked Data user can merge a branch into another branch.

In the first four cases Bi-VAKs creates both an *Update Revision* and one or multiple *Update(s)*. An *Update* always alters a single RDF dataset consisting of a default graph and zero or multiple named graphs, as illustrated in Figure 5.4a. And, it might contain inserted and deleted triples from different RDF graphs. An *Update Revision* refers to a single or multiple *Update(s)* either explicitly via the transaction revision or implicitly via the same revision number and branch index. An *Update* itself can be referred by multiple other updates in case of a branch, which is demonstrated by the update revision (7) in Figure 5.3. The update revision (7) modifies update (2), and therefore, update (7) refers to update (2). But an *Update* can also refer to multiple preceding updates in case of an update merge. Thus in comparison with the graph of transaction revisions, updates revisions may form an acyclic directed graph.

The *Update Revisions* and *Updates* are RDF resources with their own IRIs and various attributes in the RDF revision-store, as showed in Figure 5.4b. In order to define the valid time of a change, an *Update* could have an start date and an end date, or it might only have an start date, an end date, or no valid date at all (anonymous timestamp). In order to actually change the triples in the RDF dataset, the *Update* includes the inserted and deleted triples by using RDF-star, which we explain in more detail in Section 6.1.2.



(A) Update Diagram

(B) Update

FIGURE 5.4: (Provenance) diagram of the *Update (Revision)*.

In some situations it would be preferable to modify the *Update* itself. For instance, a Linked Data user has made a mistake in the end date of the *Update*. When we modify an *Update* at transaction time t_k, this *Update* should no longer exist in the RDF dataset after t_k. The new *Update* should be selected in the subsequent versions, and neither both nor the modified *Update*. But when we query a version back in time $(\ldots, t_{k-1})$, we must select the modified *Update*, because the new *Update* has not been created before t_k. Therefore, we let this new *Update* refer to the *Update* it is modifying, and we call this *Update* its preceding update. Hence, the latest *Updates* are the updates to which no other updates refer in the same sequence of transaction revisions. Figure 5.5 shows an example of two update modifications. Update (7) modifies the start date of update (2) and updates (15) adds an end date to update (2), and they refer to update (2) via the predicate 'precedingUpdate'. Now from transaction revision (14) update (7) is the latest update, while from transaction revision (3) update (2) is the most recent update.

Instead of modifying an *Update*, it is also possible to revert an *Update Revision*, and fully remove the update from the state of the dataset. The revert operation gives the *Update*

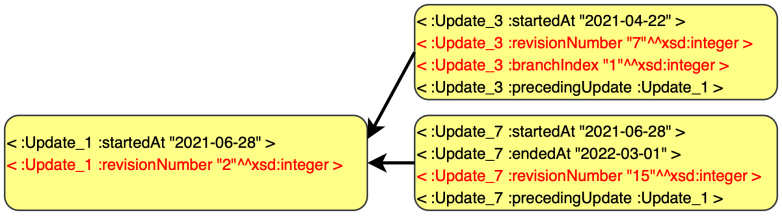
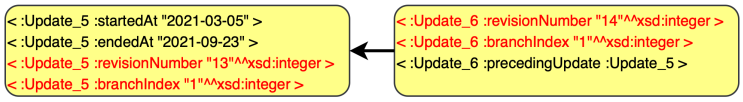
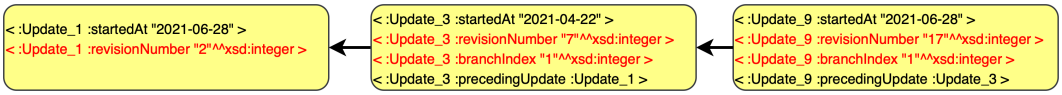


FIGURE 5.5: Example of modifying an *Update*.

Revision a start time as well an end time. The *Update Revision* should no longer exist, which affects the existence of the *Update*. All triples and quads stated in the *Update* should no longer be in that state of the RDF dataset, and thus the reverted *Update* should return to its previous state. This state is either empty, when this *Update* has not been modified before, or it contains the data from a modified *Update* earlier in time. Figure 5.6a shows an example of a reversion of an *Update*, for which its previous state is empty. Figure 5.6b demonstrates a reversion, for which its previous state is a modified *Update*.



(A) Revert an *Update* that has not been modified.



(B) Revert an *Update* that has been modified.

FIGURE 5.6: Example of reverting a *Update Revision* and a *Update*.

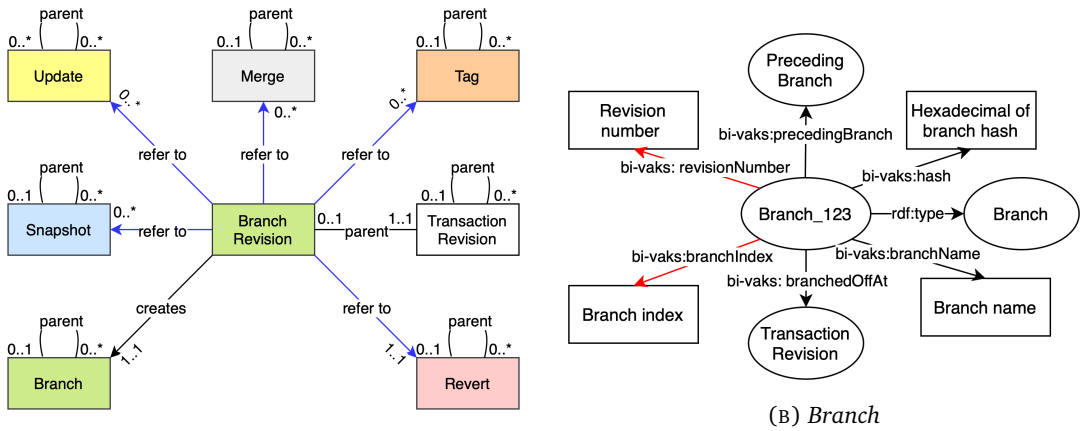
In addition to the fact that multiple *Updates* can refer to the same previous *Update*, a single *Update* can also refer to multiple previous *Updates* in order to merge them into a single *Update*. However, such a merge is only possible when the *Updates* belong to the same chain of transaction revisions (branch or main stream). Namely, it should always be possible to revert or modify such a *Update*. If a reverted merge *Update* refers to an *Update* in another branch, the *Revert Revision* would include an *Update* that belongs to another state of the revision-store, which was not in the other revision-store before. For example, suppose we would merge update (12), and update (6) in Figure 5.3, and then revert this newly created update revision. Now update (6) will end up in the main stream which was not there before. If an *Update* from another branch should be merged into the main stream or a branch, you should merge the complete branch.

5.2.3 Branch (Revision)

In order to improve the collaboration between Linked Data users, it is preferable to allow users to diverge from the common dataset without making a new copy. Therefore, Bi-VAKs allow users to branch off from the main stream or from other branches to create a separate dataset. In addition to the creation of a new branch, users could also modify an existing branch, such as its branch name. When a Linked Data user does such a branch version operation, Bi-VAKs always creates a *Branch Revision* and a *Branch*. A *Branch Revision* always has a single preceding transaction revision, and other transaction revisions can

refer to this branch revision. Figure 5.7a also illustrates that a *Branch Revision* always refers to a single *Branch*. A *Branch Revision* could also refer to other valid revisions, but it only does when a Linked Data user has made a mistake in the transaction revision the branch branches off from. When this happened, the branch requires to move to another place in the graph of revisions. However, the new branch must still refer to the already existing revisions in the modified branch without changing them. For example, if we would move branch (8) to transaction revision (4) instead of (7), it should still refer to transaction revisions (8), (11), and (12). We do not take into account that the semantics in these updates might conflict due to this change.

For simplicity a *Branch* only has a specific name. Although it would be possible to add more information to a branch resource, such as an extra description what this deviation from the dataset is about, some constraints of this deviation, or users which are allowed to modify this diverged dataset. This branch name is simply a RDF literal. As illustrated in Figure 5.7b, a *Branch* also contains the transaction revision it branches off from, primarily, in order to obtain all transaction revisions in the branch. For instance, to get all revisions from transaction revision (1) to (12), we need to know all places the branch (8) branches off from. And, as it should retrieve revisions (1), (2), (3), (4), (7), (8), (11), and (12), it needs to know that branch (8) branches off from branch (4) at revision (7), and from the main stream at revision (3). However, we do not want to retrieve the other transaction revisions in the branch and main stream.



(A) Branch Diagram

(B) *Branch*

FIGURE 5.7: (Provenance) diagram of the *Branch (Revision)*.

As we already mentioned, a Linked Data user can also modify an existing *Branch*. But, when a *Branch* is modified, it should no longer exist in the subsequent versions. Similar to an *Update*, we let this new *Branch* resource refer to the *Branch* resource it modifies. Again the latest *Branch* is the *Branch* to which no other *Branch* is referring. Figure 5.8 illustrates an example of a modification of branch (4) into branch (9) from the revision-store demonstrated in Figure 5.3. In this example the name has changed from "MyFirstBranch" to "MyChangedBranch".

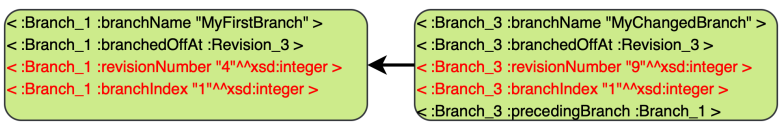
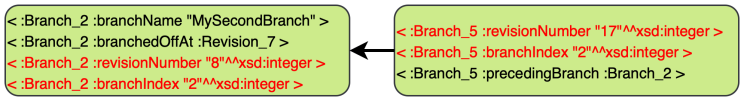
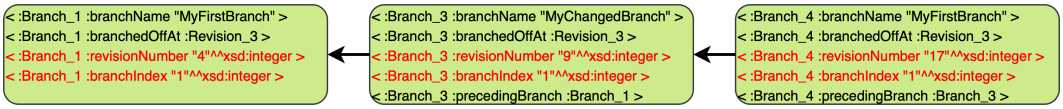


FIGURE 5.8: Example of modifying a *Branch*.

In order to ensure that the complete *Branch* no longer exists in the succeeding transaction revisions, a *Branch Revision* can be reverted. It should not be possible to query that branch at the following transaction time, but in previous transaction time we should still need to query this branch. Since we can modify a *Branch*, a reversion of a *Branch* does not always lead to a removal of a complete branch. We can also revert a modification of a branch such that we return to the initial branch. Figure 5.9a gives an illustration of Branch_2 fully removed, whereas Figure 5.9b gives an example that Branch_3 returns to its previous state. Nonetheless, if another branch already branches off from a branch, which is later reverted, this branch is not reverted and can still be queried in succeeding revisions. Suppose, if we revert branch (4) illustrated in Figure 5.3, branch (8) will still exist.



(A) Revert a *Branch*, which has not been modified.



(B) Revert a *Branch*, which has been modified.

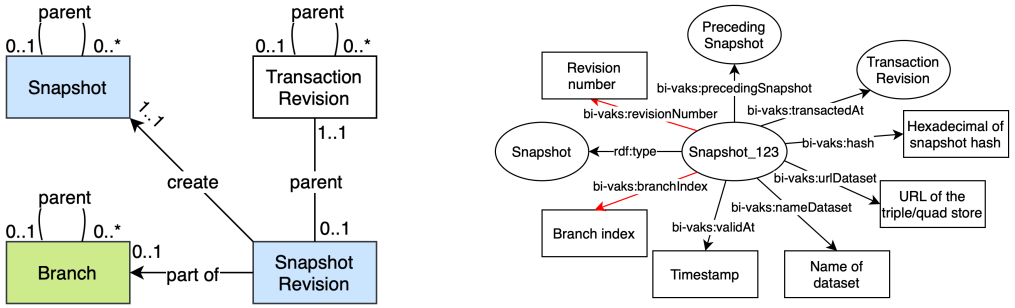
FIGURE 5.9: Example of reverting a *Branch Revision* and a *Branch*.

5.2.4 Snapshot (Revision)

In order to speed up the construction process of a particular version, Bi-VAKs allows it users to create a materialised version of a given state of the RDF dataset $D_{i,j}$ (snapshot). If this state is frequently used, it can be directly queried without having to construct a state first. Bi-VAKs creates a *Snapshot Revision* and a *Snapshot* when a Linked Data user either (i) does a snapshot versioning operation to create a complete new snapshot or (ii) modifies an existing *Snapshot*, for example, to change its effective date. A *Snapshot Revision* always refers to a single *Snapshot* and the *Snapshot* also refers to a single *Snapshot Revision*, which is illustrated in Figure 5.10a. In addition, a snapshot can also be part of a branch, and hence can include a materialised version of the diverged dataset, which is also illustrated in Figure 5.3 by snapshot (11).

A *Snapshot* should contain all the relevant information to query a RDF dataset via a HTTP endpoint, such as the name of the dataset and the URL of the triple/quad-store, which is demonstrated in Figure 5.10b. These attributes are both a literal. In addition, since the RDF dataset is bi-temporal ($D_{i,j}$), a *Snapshot* also contains an effective date (i), and a transaction revision. This effective date is RDF literal with a date timestamp as datatype, and can be in the past ($i < j$) as well as in the future ($i > j$). The transaction revision is an IRI that indicates the transaction time, and is either equal to or smaller than the latest transaction revision (head revision) in the revision-store.

In addition to the creation of a new *Snapshot*, an user could also modify an existing *Snapshot*. For instance, when a snapshot might be outdated. In order to query a snapshot, the previous snapshots should no longer be queried. They might be overwritten, or moved to another triple/quad-store. However, since no information may ever be deleted, we also refer to the preceding *Snapshot* within the new *Snapshot*. However, compared to the other valid revisions, we always want to query the latest snapshot, because this snapshot



(A) Snapshot diagram (B) *Snapshot*

FIGURE 5.10: (Provenance) diagram of the *Snapshot (Revision)*.

is actually referring to an existing and reachable RDF dataset. Figure 5.11 represents the example of snapshot (3) and (10) from Figure 5.3 that overwrite the same quad-store, such that the RDF dataset comes into a new state with another effective date.

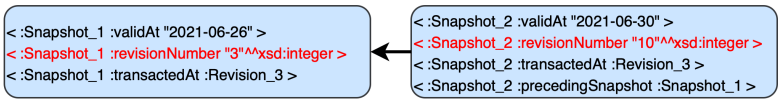
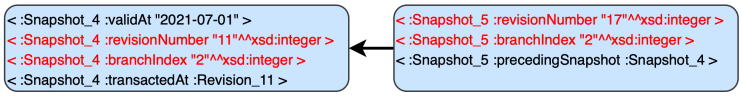
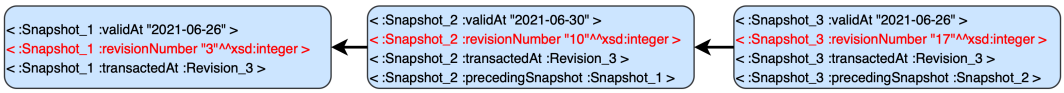


FIGURE 5.11: Example of modifying a *Snapshot*.

Similar to the update and the branch operations a *Snapshot Revision* can also be reverted. Reverting a *Snapshot Revision* leads either to a complete removal, or a reversal of a *Snapshot*. Figure 5.12a gives an illustration that Snapshot_2 is fully removed, whereas Figure 5.12b gives an example that Snapshot_3 returns to its previous state Snapshot_1.



(A) Revert an *Snapshot* that has not been modified.



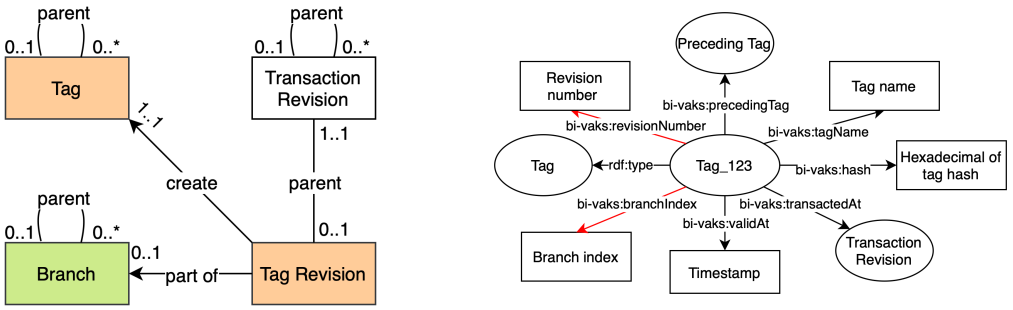
(B) Revert an *Snapshot* that has been modified.

FIGURE 5.12: Reverting a *Snapshot Revision* and a *Snapshot*.

5.2.5 Tag (Revision)

As our versioning approach will create enormous amount of revisions over time, it would be helpful if Linked Data users could explicitly refer to a state of the RDF dataset by a self-defined name, also called a tag. For example, to indicate a specific state of the dataset that users have used in their reports, and projects. When a Linked Data user does such a tag versioning operation or modifies an existing *Tag*, Bi-VAKs always creates a *Tag Revision*, and a single *Tag*, which is illustrated in Figure 5.13a.

As depicted in Figure 5.13b, a *Tag* resource always contains the self-defined tag name, which is a RDF literal. Additionally, it contains an effective date, and a transaction revision in order to refer to a state of the RDF dataset. We define its transaction time (j) by referring to a transaction revision. And we specify the valid time (i) by an effective date,: a literal with a date timestamp as datatype. This date can be in the past ($i < j$) as well as in the future ($i > j$). The referred transaction revision is either equal to or smaller than the most recent transaction revision (head revision) in the revision-store. For convenience, we only include these attributes, but it would be possible to add more attributes to a *Tag*, such as an extra description about this state of the dataset, users who worked on this dataset until this point, or a specific purpose of this state.



(A) Tag Diagram (B) Tag

FIGURE 5.13: (Provenance) diagram of the *Tag (Revision)*.

It is possible, however, that a Linked Data user is not satisfied with the current name of the tag. Therefore, it is required that we can modify a *Tag* without losing the data of all previous *Tag(s)*. Figure 5.14 demonstrates a modification in the name of tag (5) from Figure 5.3. The Tag_2 sill includes the same effective date and transaction revision.

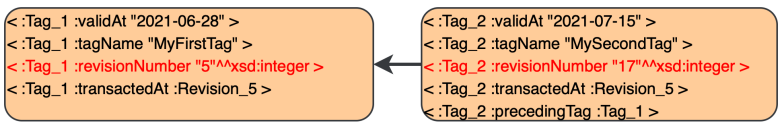
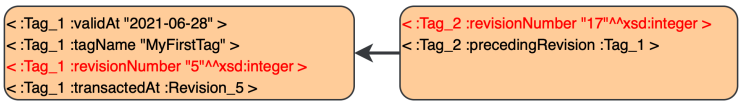


FIGURE 5.14: Example of modifying a *Tag*.

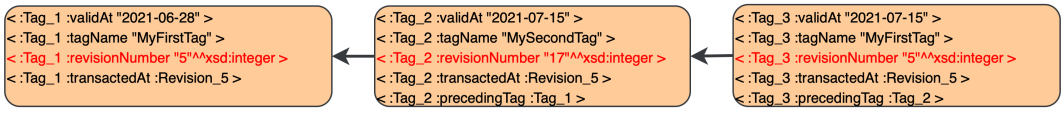
In some cases, it is even necessary that a *Tag* should no longer exist in the next states of the revision-store. Therefore, our approach supports the reversion of a *Tag Revision*, and its corresponding *Tag* by creating a new *Tag* that reverts its preceding *Tag*. This preceding *Tag* is either empty, or modified. These two options are illustrated respectively in Figure 5.15a and Figure 5.15b.

5.2.6 Revert (Revision)

As we already describe in the aforementioned sections, we can revert a *Transaction Revision*. When we revert such a branch, update, snapshot, merge and tag revision a *Revert Revision* is created that implicitly or explicitly refers to a single or multiple newly created update(s), snapshot(s), branch, merge(s), and tag(s), which is respectively illustrated in Figure 5.16a. Besides, we can revert a *Revert Revision* or modify a *Revert* itself. Both



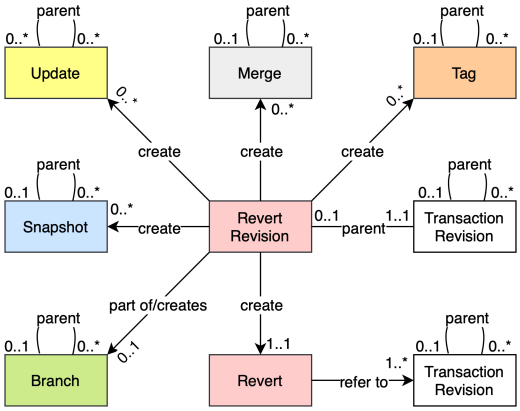
(A) Revert an *Tag* that has not been modified.



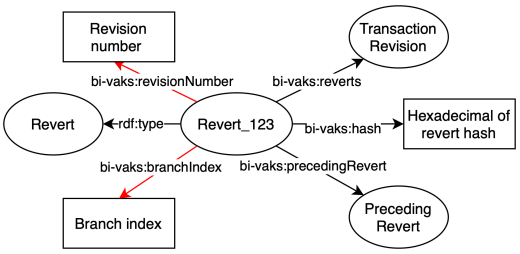
(B) Revert an *Tag* that has been modified.

FIGURE 5.15: Reverting a *Tag Revision* and a *Tag*.

actions create a new *Revert Revision* and *Revert*. The creation of a *Revert Revisions* always goes hand in hand with the creation of a single *Revert*. The *Revert* refers to the *Transaction Revision(s)* it reverts, which is visualised in Figure 5.16b. A *Revert* might also contain other attributes such as a keyword that denotes the reason why this revision should be reverted.



(A) Revert Diagram

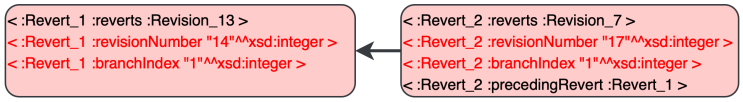


(B) *Revert*

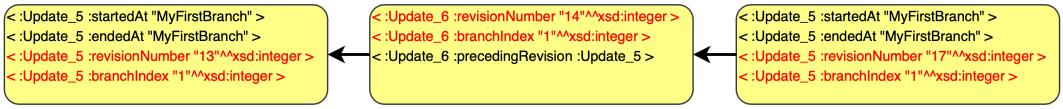
FIGURE 5.16: (Provenance) diagram of the *Revert (Revision)*.

Since our versioning approach should be able to modify all valid revisions, it should modify the *Revert* as well. However, not only the *Revert*, also the other *Transaction Revisions*, to which the corresponding *Revert* refers, should be reverted. For example, if transaction revision b should be reverted instead of transaction revision a, transaction revision a should have been brought back to its previous state. Therefore, the valid revisions corresponding to revision a should be reverted first, before the valid revisions corresponding to revision b can be reverted. The *Revert Revision* should refer to reverted valid revision from a, and reverted valid revisions from b. Figure 5.17a illustrates such an example. Suppose we modify revert (14) from Figure 5.3 such that it now reverts update (7) instead of update (13). Therefore, update (13) should be no longer reverted, and should be reverted to its previous update; update (7) should be reverted, and we should modify revert (14), as demonstrated in Figure 5.17b.

Just as we can modify a *Revert*, we can also revert a *Revert Revision*. Reverting a *Revert Revision* is rather similar to modifying a *Revert*. Again we should also revert all transaction



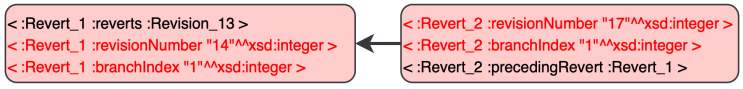
(A) Example of modifying a *Revert*



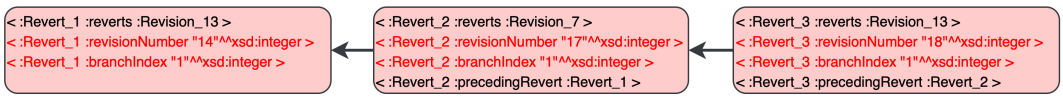
(B) Example of reverting a reverted *Update*.

FIGURE 5.17: Example of the modification process of a *Revert*.

revisions to which the corresponding *Revert* is referring. These newly created valid revisions are subsequently added to the *Revert Revision* that revert a *Revert Revision*. Figure 5.18a and 5.18b give an example when revert (14) would be reverted. They respectively illustrate that the reverted *Revert* is either empty or it returns back to its previous state.



(A) Revert an *Revert* that has been modified.



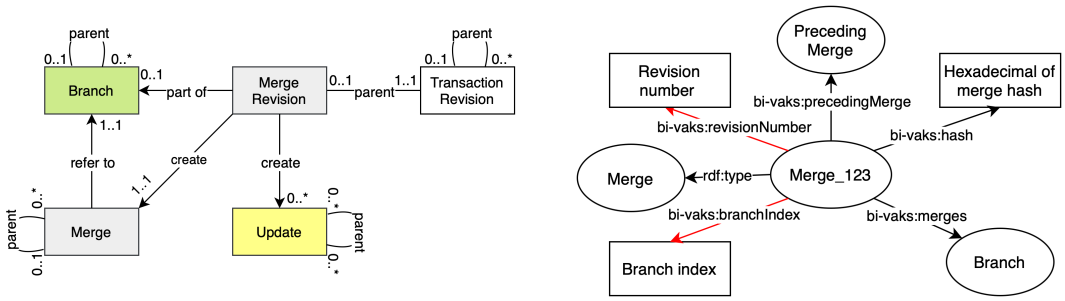
(B) Revert an *Revert* that has been modified.

FIGURE 5.18: Reverting a *Revert Revision* and a *Revert*.

5.2.7 Merge (Revision)

As Linked Data users can diverge from the common RDF dataset, they should also be able to conflate the diverged states, for instance, when they finally agree. Although in our prototypical versioning approach we do not implement the merge operation, in this section we give a small introduction of the merge operation for completeness. When a Linked Data user does a merge versioning operation, the user merges the given branch into a branch from which the user does the operation. When a branch B is merged into another branch A the updates belonging to branch B should be added to branch A. These updates from branch B are added to branch A by creating new or referring to existing *Update(s)* that contain the inserted and deleted quads in the diverged dataset. However, we refer to future research for a more in-depth study on the possible merge strategies and on the practical implementation of a merge.

As depicted in Figure 5.19a, a *Merge Revision* always creates a single *Merge* that again refers to a *Branch* in order to indicate the branch that is merged into the branch or main stream from which the user has done a merge operation. Figure 5.3 illustrates an example of a merge (16) that merges branch (4) into the main stream. Instances of the *Merge Revision* and *Merge* are both resources with their own IRIs and attributes, as showed in Figure 5.19b. It would be possible to add more attributes to a *Merge*, such as a merge strategy or some time constraints the updates must fulfill.



(A) Merge Diagram

(B) *Merge*

FIGURE 5.19: (Provenance) diagram of the *Merge (Revision)*.

In addition, a Linked Data user could also make a mistake in the branch it wanted to merge. When this happens, the updates, which were created or referred to in the modified merge revision, should be reverted, and should no longer exist in the new state of the revision-store. Thus, such a *Merge* modification does something similar as a *Revert* modification, because all *Updates* to which they refer must first be reverted. However, it must not revert to its previous state, because we might possibly add an update to the main stream or branch which was not there before. It always reverts an *Update* as if it had not been modified before. Figure 5.20 illustrates an example of a merge modification based on the revision-store presented in Figure 5.3. Instead of merging branch (4) into the main stream, branch (8) is merged into the main stream. The update (16) should thus be reverted.

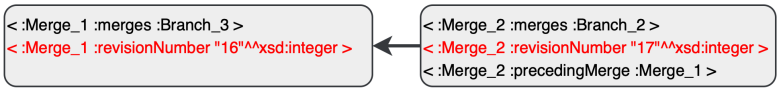
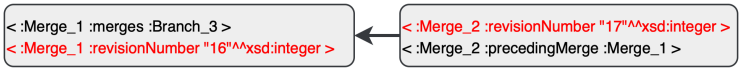
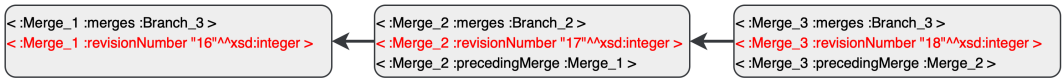


FIGURE 5.20: Example of modifying a *Merge*.

In addition to rectifying mistakes, a *Merge Revision* can also be completely discarded from the revision-store. The corresponding *Merge* should no longer exist in the coming states of the revision-store, and this reverted *Merge* should return to its previous state. Reverting a *Merge Revision* is rather similar to modifying a *Merge*. Again we should also revert all *Updates* to which the *Merge Revision* is referring. The previous state of a *Merge* is either empty, or contains some data, which is respectively illustrated in Figure 5.21a and Figure 5.21b.



(A) Revert an *Merge* that has not been modified.



(B) Revert an *Merge* that has been modified.

FIGURE 5.21: Example of reverting a *Merge Revision* and a *Merge*.

Chapter 6

Bi-Temporal SPARQL (Update) Query

In this chapter, we elaborate on the interaction between the Linked Data user and the revision-store to allow for bi-temporal SPARQL (Update) queries. We begin this chapter by describing how Bi-TR4Qs detects the bi-temporal changes made to a RDF dataset from a SPARQL Update query, and how it makes use of RDF-star to actually represent these changes in the revision-store (Section 6.1). Next, Section 6.2 explains how Bi-TR4Qs retrieves these changes from the revision-store, and how it subsequently constructs a particular (prior) state of the dataset by means of these updates. The last section 6.3 gives a description of the different SPARQL basic triple pattern queries, such as the version materialisation, delta materialisation, and version queries.

6.1 Bi-Temporal SPARQL Update Query

When a Linked Data user sends a SPARQL Update query to insert, delete, or modify some triples in the RDF dataset, the RDF dataset comes into a new state. In order to keep track of these changes, and to subsequently query prior states of the RDF dataset, Bi-VAKs must first detect these bi-temporal changes, then represent them in a *Update* and *Update Revision*, and finally store them in the revision graph $R_{i,j}$. In Section 6.1.1 we first describe how Bi-VAKs extracts the change(s) from the SPARQL Update Query instead of determining the change(s) between two consecutive dataset states in retrospect by using a difference algorithm. Once Bi-VAKs has identified the collection of bi-temporal changes from a single SPARQL update query, this collection must be represented and in addition its valid and transaction time must be taken into account. Section 6.1.2 gives an explanation how we model and store such a collection by using RDF-star.

6.1.1 Detecting Changes

In this section we describe how Bi-VAKs determines the collection of inserted and deleted quads from a SPARQL Update Query. The SPARQL 1.1 Update Language (Gearon, Passant, and Polleres, 2013) provides various graph update operations and graph management operations, as described in Section 6.1. These operations modifies triples in a RDF graph and complete graphs in a RDF dataset. But, due to time limitations we only focus on the INSERT DATA and DELETE DATA update operations. When a Linked Data user sends such an update operation, Bi-VAKs should create both an *Update Revision* and an *Update* by determining the inserted and deleted triples or quads from the query. As we stated in Section 4.3, for an update operations it is important that we can invert the corresponding *Update*. A change can be inverted if the inserted quad did not exist yet, and if the deleted

quad already exists in the RDF dataset at the time of deletion. However, our RDF dataset is bi-temporal. An *Update* has a valid time as well as a transaction time. The valid time t^v is a time interval $[t_1^v, t_2^v]$, an effective date $[t_1^v, \infty)$, a final date $(-\infty, t_2^v]$ or even no date $(-\infty, \infty)$. Therefore, a quad can exist multiple times in the dataset at different moments in time. A quad can only be inserted if it does not overlap with another update that inserts the same quad, and a quad can only be deleted if its time interval fully overlaps another update valid time that inserts the same quad. Nonetheless, it should also be possible to insert a quad, which was deleted before, or to delete a quad which was inserted again. Therefore, we define the definitions to insert and delete a quad as follows:

Definition 8 (Insertion of a Quad). *A quad Q with valid time t^v can be inserted if $N = M$, where N stands for the number of updates that inserts Q and somewhere overlaps t^v, and where M stands for the number of updates that deletes Q and fully overlaps t^v.*

Definition 9 (Deletion of a Quad). *A quad Q with valid time t^v can be deleted if $N = M+1$, where N stands for the number of updates that inserts Q and fully overlaps t^v and where M stands for the number of updates that deletes Q and somewhere overlaps t^v.*

Figure 6.1 illustrates an example of an update containing quad x with start time a and end time b ($[a, b]$), and four additional updates that either inserts (blue) or deletes (red) quad x. Suppose we would like to insert x in an update with time interval $[c, d]$. It is not possible to insert this quad if its time interval $[c, d]$ somewhere overlaps the interval $[a, b]$. And suppose we would like to delete x. Its interval $[e, f]$ must fully overlap with the interval $[a, b]$.

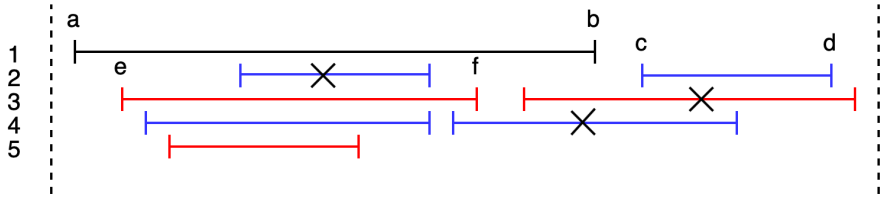


FIGURE 6.1: Example of 5 updates that all insert (blue) and delete (red) quad x in order to illustrate possible overlaps.

As depicted in Figure 6.1, the INSERT DATA and DELETE DATA update operation explicitly defines the quads, which should be inserted or deleted in the RDF dataset. Thus, Bi-VAKs only requires to determine whether these specified inserted and deleted quads satisfy the above mentioned definitions - the “invertibility check” - in order to create an update that can be inverted. If a quad does not satisfy these conditions, the whole change will be rejected.

LISTING 6.1: A INSERT/DELETE DATA graph update operation

```
PREFIX recipes: <http://www.recipehub.nl/recipes#> .

DELETE DATA
{ GRAPH recipes:GreatCurries { recipes:KatsuCurry recipes:meal recipes:MainCourse } } ;
INSERT DATA
{ GRAPH recipes:GreatCurries { recipes:KatsuCurry recipes:meal recipes:Starter } }
```

Although we only concentrate on INSERT DATA and DELETE DATA update operations in this thesis, it is also possible to detect the collection of modified quads from the DELETE/INSERT update operations. Since these operations do not specify the modified quads explicitly, we must first discover which quads correspond to the query at the

given particular time point or interval. However, to determine these modified quads from a bi-temporal RDF dataset is rather complex. If an update has a start s and an end date t, we need to query the RDF dataset both at state s, and at state t, as they might return different quads. These different quads must subsequently correspond, and they should satisfy the above mentioned definitions in order to add them to the collection of modified quads. Namely, we must ensure that they exist or do not exist in the dataset over the full time interval. If an update only has a start or end date, we need to query a single state, but we must still verify that these quads satisfy the above mentioned definitions. Figure 6.2 illustrates an example of 9 updates each containing a single inserted triple $\{\langle$ Car_x :color "red" $\rangle | x \in \{1,\ldots,9\}\}$. The bars indicate their time interval. Suppose a Linked Data users sends the SPARQL update query to change all red cars into blue cars, and it has a start s and end time t, which are illustrated by the green vertical lines. We construct the dataset state at $D_{s,j}$ and at $D_{t,j}$, and respectively obtain 4 red cars (2, 5, 7, 8) at state $D_{s,j}$, and 3 red cars (5, 7, 9) at state $D_{t,j}$. However, at the end time t car 2 and 8 do not exist in the dataset, and at the start date s car 9 does not exist. Therefore, only red car 5 and 7 can be changed into blue cars. Suppose the user's request only has a start date s or end date t the cars queried at state $D_{s,j}$ or at state $D_{t,j}$ should exist over the full interval in order to delete them.

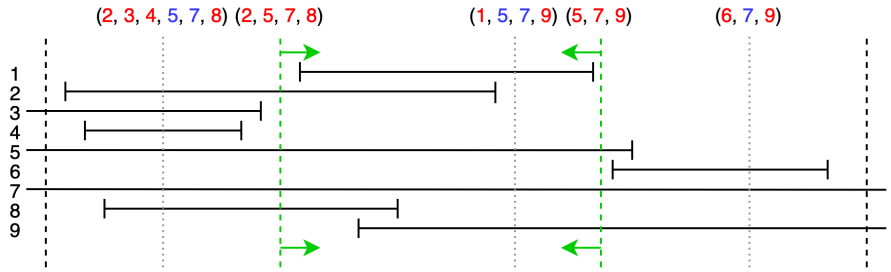


FIGURE 6.2: Example of 9 updates all containing information about red cars which are transformed into blue cars in a specific time interval.

Although it does not seem very efficient to determine for each quad whether it satisfies the above mentioned definitions, it is very important to avoid inconsistencies in the RDF dataset. Otherwise, it might happen that we delete a quad, which has never been inserted before, or insert a quad, which has already been inserted. Hence, during a version construction process we might end up in a state of the dataset containing the wrong quads. In addition to a SPARQL Update query requests, Bi-VAKs should also verify whether the quads in an update modification request, or update reversion request satisfy the above mentioned definitions.

6.1.2 Representing Changes

Once Bi-VAKs has extracted the collection of inserted and deleted triples or quads from the SPARQL Update Query, it should represent these triples in such a way that they can be stored in a triple/quad-store, and be queried to construct prior states of the RDF dataset. Instead of using either separate named graphs for the inserted triples and deleted triples, or standard RDF reification, Bi-VAKs puts them together in a *Update* and uses RDF-star. As we already described in section 5.2.2, a *Update* is a RDF resource that refers to the inserted triples by the predicate 'inserts' and to the deleted triples by the predicate 'deletes'. The object of these triples subsequently refers to the inserted and deleted triple by using the RDF-star notation. A complete inserted or deleted triple has now become a node in a

RDF graph, such that it is possible to query an update based on the triples it contains. We even do not need to know these triples in their entirety. They can also be queried from a basic triple pattern. In addition to the fact that Bi-VAKs should store and query the annotated triples, it should also consider the transaction and valid time of each collection of changes. Therefore, we distinguish *Update* from *Update Revisions*, which we already described thoroughly in Section 5.2.2. Listing 6.2 demonstrates an example of an *Update Revision* in the Turtle format. This *Update Revision* refers to the *Update* illustrated in Listing 6.3. This *Update* is in a diverged state of the dataset, and it inserts the triple ⟨ recipes:RecipeChocolateCake rdf:type recipes:Recipe ⟩ into the RDF dataset, and deletes the triple ⟨ recipes:RecipeLemonCake rdf:type recipes:Recipe ⟩.

LISTING 6.2: *Update Revision* representation in Turtle

```
@prefix recipes: <http://recipehub.nl/recipes#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://bi-vaks.org/vocab/> .

:Revision_f8dcab6ec0544c0c8287aee7f31a912a9ce572955b0d0e7ba50972d5b053e5a5
    a :UpdateRevision ;
    rdfs:comment "Inserts a chocolate cake recipe, and deletes a lemon cake recipe." ;
    :createdAt "2021-05-19T13:15:00+02:00"^^xsd:dateTimeStamp ;
    :author "Lisa Meijer" ;
    :hash "f8dcab6ec0544c0c8287aee7f31a912a9ce572955b0d0e7ba50972d5b053e5a5" ;
    :precedingRevision :Revision_99eff5bc911699bfa200c3296433b48e1481d2e5a7a4fef046 ;
    :update :Update_716bb121c3ce50d043d3fc7058f23d7e5160f3194948c114dcb ;
    :revisionNumber "1005"^^xsd:nonNegativeInteger ;
    :branchIndex "2"^^xsd:nonNegativeInteger ;
.
```

LISTING 6.3: *Update* representation in Turtle-star

```
@prefix recipes: <http://recipehub.nl/recipes#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <http://bi-vaks.org/vocab/> .

:Update_716bb121c3ce50d043d3fc7058f23d7e5160f3194948c114dcb
    a :Update ;
    :inserts << recipes:RecipeChocolateCake rdf:type recipes:Recipe >> ;
    :deletes << recipes:RecipeLemonCake rdf:type recipes:Recipe >> ;
    :startedAt "2021-04-10T17:45:00+02:00"^^xsd:dateTimeStamp ;
    :endedAt "2021-09-03T09:00:00+02:00"^^xsd:dateTimeStamp ;
    :hash "_716bb121c3ce50d043d3fc7058f23d7e5160f3194948c114dcb" ;
    :revisionNumber "1005"^^xsd:nonNegativeInteger ;
    :branchIndex "2"^^xsd:nonNegativeInteger ;
.
```

Representing named graphs

One of the main disadvantage of RDF-star for our version control system is that RDF-star is not able to represent quads. Therefore, we must define quads differently. We add the context information or graph name of the inserted and deleted quads to the context information the update. In this way an *Update* can contain quads from multiple named graph. Listing 6.4 shows an example of an *Update* that inserts the following quad ⟨ recipes:RecipeLambVindaloo rdf:type recipes:Recipe recipes:GreatCurries ⟩.

LISTING 6.4: *Update* representation in Turtle-star including an inserted quad.

```
@prefix recipes: <http://recipehub.nl/recipes#> .
@prefix : <http://bi-vaks.org/vocab/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

:Update_f544227dfd877dd995bc886b54e4b8c22120bd94379628753af5655ac29f6aa1
    a :Update ;
```

```
    :startedAt "2021-06-16T12:20:00+02:00"^^xsd:dateTimeStamp ;
    :endedAt "2021-12-07T10:30:00+02:00"^^xsd:dateTimeStamp ;
    :hash "f544227dfd877dd995bc886b54e4b8c22120bd94379628753af5655ac29f6aa1" ;
.
recipes:GreatCurries {
    :Update_f544227dfd877dd995bc886b54e4b8c22120bd94379628753af5655ac29f6aa1 :inserts <<
        recipes:RecipeLambVindaloo rdf:type recipes:Recipe >> ;
. }
```

Content strategies of an update

As we already described in Section 5.2.2, Linked Data users could make a mistake in an *Update*. For instance, its start date is incorrect, or some triples should not be included. Bi-VAKs gives users the possibility to modify an *Update*. However, a modification might cause overlap between the modified *Update* and the newly created *Update*. In Figure 6.3 we show two strategies how we can express these *Update* modifications. The first strategy (1) - the repeated update content strategy - creates a new *Update* which contains the triples present in the new state of the dataset. It fully copies the previous *Update* and only changes the quads, and dates which the user modifies. The second strategy (2) - the related update content strategy - creates a new *Update* that only contains the modified quads and dates, and refers to its modified *Update* for the unchanged quads.

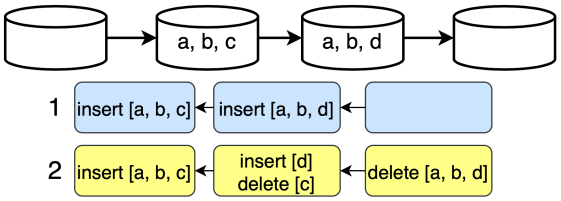


FIGURE 6.3: Visualisation of two update content strategies: (1) the repeated update content strategy and (2) the related update content strategy.

Suppose a Linked User would like to delete the 'RecipeAppleCake' instead of 'RecipeLemonCake' depicted in Listing 6.3. Listing 6.5 gives an illustration of strategy (1) that replaces ⟨ recipes:RecipeLemonCake rdf:type recipes:Recipe ⟩ with ⟨ recipes:RecipeAppleCake rdf:type recipes:Recipe ⟩. Listing 6.6 gives an illustration of strategy (2) that deletes ⟨ recipes:RecipeAppleCake rdf:type recipes:Recipe ⟩ and inserts the ⟨ recipes:RecipeLemonCake rdf:type recipes:Recipe ⟩.

LISTING 6.5: *Update* representation in Turtle-star using the repeated update content strategy

```
@prefix recipes: <http://recipehub.nl/recipes#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <http://bi-vaks.org/vocab/> .

:Update_ae338f95e454e75a2b913824ef270c7bc58894f30007d63a8f07acd9884d34f9
    a :Update ;
    :inserts << recipes:RecipeChocolateCake rdf:type recipes:Recipe >> ;
    :deletes << recipes:RecipeAppleCake rdf:type recipes:Recipe >> ;
    :precedingRevision :Update_848c4085c112cb3d822636feb91db8d8122e78dda76 ;
.
```

LISTING 6.6: *Update* representation in Turtle-star using the related update content strategy

```
@prefix recipes: <http://recipehub.nl/recipes#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <http://bi-vaks.org/vocab/> .
```

```
:Update_ae338f95e454e75a2b913824ef270c7bc58894f30007d63a8f07acd9884d34f9
    a :Update ;
    :deletes << recipes:RecipeAppleCake rdf:type recipes:Recipe >> ;
    :inserts << recipes:RecipeLemonCake rdf:type recipes:Recipe >> ;
    :precedingRevision :Update_848c4085c112cb3d822636feb91db8d8122e78dda76 ;
.
```

6.2 Version Retrieval

In this section we explain how we can construct a particular version or state of the dataset $D_{i,j}$ to allow for bi-temporal SPARQL queries. Such a specific version can be constructed by first obtaining the updates, then by sorting these updates based on their transaction time, and finally by fast forwarding them on an empty RDF graph. Suppose the state of the dataset has a single time dimension, and we would like to construct the blue dataset state, as illustrated in Figure 6.4. We must first obtain all the updates represented by the yellow blocks. Then we sort the updates, and subsequently fast forward them to construct the blue dataset state. The inserted triples are added, and the deleted triples are removed from every intermediate dataset state. Figure 6.4 directly shows that we must first sort the updates before we can fast forward the updates. Otherwise we end up with a different set of triples in the blue state. Besides, we cannot aggregate the updates to speed up this process, because we cannot invert the updates: g is already deleted in the second update, while it is just inserted in the fourth update. Aggregating takes all updates together, and it cancels out the same added and deleted triples. Hence, aggregating avoids sorting.

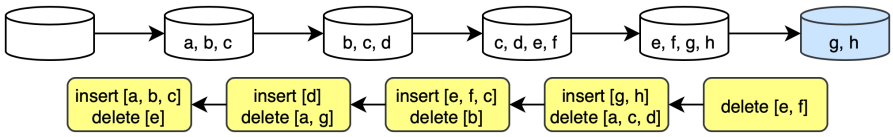


FIGURE 6.4: Visualisation of the influence of non-invertible updates to the states of the RDF dataset.

A bi-temporal state of the dataset or version $D_{i,j}$ depends on a valid time i and a transaction time j, where both i and j are a single point in time. In order to create a certain state of the dataset, we first need to obtain all required updates. An update, however, also has both a transaction time, and a valid time. The valid time t^v of an update can be a time interval $[t^v_s, t^v_e]$, an effective date $[t^v_s, \infty)$, a final date $(-\infty, t^v_e]$ or even no date $(-\infty, \infty)$. Therefore, it is possible that several updates might not exist in certain state of the RDF dataset, because they are just valid ahead ($t_s > i$) or back ($t_e < i$) in time. Besides, the update's transaction time t^t can be an effective date $[t^t_s, \infty)$ as well as a time interval $[t^t_s, t^t_e]$ due to a reversion or modification. Therefore, in order to obtain the right updates, the updates' transaction time t^t must enclose the transaction time j, and their valid time t^v must enclose the valid time i of the requested state $D_{i,j}$. The transaction time of update u encloses j if no other update with $t^t_s \leq j$ is referring to update u. The valid time encloses i if $t^v_s \leq i \leq t^v_e$.

However, the way we retrieve these updates and updates revision from the revision-store depends on the reference strategy (Sub-section 5.1.1). For the explicit reference strategy we use SPARQL property paths (Seaborne, 2010) to query all transaction revisions that meets the given time interval, and we use the explicit reference to the valid revision(s) to query the updates. Listing 6.7 presents an example of a SPARQL query for the explicit

reference strategy. With line 5 and 6 we query all updates having a transaction time equal to or smaller than j. With line 7 to 10 we query all updates having a valid time that encloses i, and with line 11 to 15 we remove all updates having a transaction time smaller than j.

LISTING 6.7: An example of a SPARQL query to obtain the request updates for the explicit reference strategy

```
1  PREFIX : <http://www.bi-vaks.org/vocab/> .
2
3  SELECT ?update
4  WHERE {
5      :Revision_f8dcab6ec0544c0c8287ae :precedingRevision* ?transactionRevision .
6      ?transactionRevision :update ?update .
7      OPTIONAL {{ {?update :startedAt ?startDate . }}
8      FILTER ( !bound(?startDate) || ?startDate <= "2015-07-02T06:01:23+00:00"^^<http://
           www.w3.org/2001/XMLSchema#dateTimeStamp> )
9      OPTIONAL {{ ?update :endedAt ?endDate . }}
10     FILTER ( !bound(?endDate) || ?endDate >= "2015-07-02T06:01:23+00:00"^^<http://www.w3
           .org/2001/XMLSchema#dateTimeStamp> )
11     MINUS {
12         :Revision_f8dcab6ec0544c0c8287ae :precedingRevision* ?otherTransactionRevision .
13         ?otherTransactionRevision :update ?otherValidRevision .
14         ?otherValidRevision :precedingUpdate ?update .
15     }
16 }
```

For the implicit reference strategy we use the revision numbers and the branch indices to query the updates. Listing 6.8 gives an example of such a SPARQL query for the implicit reference strategy. With line 6 to 9 we query all updates having a transaction time equal to or smaller than j. In this example, it means that we must query all updates in the main stream having a revision number smaller or equal to 31. With line 10 to 13 we query all updates having a valid time that encloses i, and with line 14 to 20 we remove all updates having a transaction time smaller than j.

LISTING 6.8: An example of a SPARQL query to obtain the request updates for the implicit reference strategy

```
1  PREFIX : <http://www.bi-vaks.org/vocab/> .
2  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
3
4  SELECT ?update
5  WHERE {
6      ?update rdf:type :Update .
7      ?update :revisionNumber ?revisionNumber .
8      OPTIONAL { ?update :branchIndex ?branchIndex }
9      FILTER ( ( !bound(?branchIndex) && ?revisionNumber <= 31 ) )
10     OPTIONAL {{ {?update :startedAt ?startDate . }}
11     FILTER ( !bound(?startDate) || ?startDate <= {0} )
12     OPTIONAL {{ ?update :endedAt ?endDate . }}
13     FILTER ( !bound(?endDate) || ?endDate >= {0} )
14     MINUS {
15         ?other rdf:type :Update .
16         ?other :revisionNumber ?otherRevisionNumber .
17         OPTIONAL { ?other :branchIndex ?otherBranchIndex }
18         FILTER ( ( !bound(?otherBranchIndex) && ?otherRevisionNumber <= 31 ) )
19         ?other :precedingUpdate ?update.
20     }
21 }
```

Finally, for the combined reference strategy we uses the revision numbers and branch indices to query the update revisions, and the explicit reference to the valid revision(s) to query the updates. Listing 6.9 gives an example of such a SPARQL query for the combined reference strategy. With line 5 to 7 we query all update revisions having a transaction time equal to or smaller than j. In this example, it means that we must query all update

revisions in the main stream having a revision number smaller or equal to 31. With line 8 we query the updates corresponding to the update revisions. With line 9 to 12 we query all updates having a valid time that encloses i, and with line 13 to 19 we remove all updates having a transaction time smaller than j.

LISTING 6.9: An example of a SPARQL query to obtain the request updates for the combined reference strategy

```
1  PREFIX : <http://www.bi-vaks.org/vocab/> .
2
3  SELECT ?update
4  WHERE {
5      ?transactionRevision :revisionNumber ?revisionNumber .
6      OPTIONAL { ?transactionRevision :branchIndex ?branchIndex }
7      FILTER ( ( !bound(?branchIndex) && ?revisionNumber <= 31 ) )
8      ?transactionRevision :update ?update.
9      OPTIONAL {{ {?update :startedAt ?startDate . }}
10     FILTER ( !bound(?startDate) || ?startDate <= {0} )
11     OPTIONAL {{ ?update :endedAt ?endDate . }}
12     FILTER ( !bound(?endDate) || ?endDate >= {0} )
13     MINUS {
14         ?otherTransactionRevision :revisionNumber ?otherRevisionNumber .
15         OPTIONAL { ?otherTransactionRevision :branchIndex ?otherBranchIndex }
16         FILTER ( ( !bound(?otherBranchIndex) && ?otherRevisionNumber <= 31 ) )
17         ?otherTransactionRevision :update ?other .
18         ?other :precedingUpdate ?update.
19     }
20 }
```

After we have obtained all updates fulfilling the transaction and valid time, we can construct the version: we first sort the updates by their transaction time and subsequently fast forward the updates by inserting and deleting their quads from a temporal dataset state. Figure 6.5 gives an illustration of various transaction and valid revisions between 1 June 2021 and 1 July 2021. The *Updates* and *Updates Revision* in yellow might have a different transaction and valid time which is indicated by their position in the figure. The *Updates* only have an effective date $[t_s^v, \infty)$, and some *Updates* are modified or reverted, because another *Update* is referring to them. Suppose we would like to know which updates belong to a state of the RDF dataset. For example, a state has a valid time of 20 June 2021 and a transaction time of 4 June 2021 (green). We only consider *Update Revisions* (25) and (26). Since *Update* (26) is not valid yet at 20 June 2021, this state only contains *Update* (25). Another example considers a state with a valid time of 12 June 2021 and a transaction time of 28 June 2021 (blue). We consider all *Update Revisions* in the main stream, but only *Updates* (25) (30), (27), and (29) are valid. Furthermore, we take *Update* (30) instead of (25), because *Update* (30) is referring to (25). The last example look at a state in a branch with a valid time of 3 June 2021 and a transaction time of 19 June 2021 (red). For this example, we consider *Update Revisions* (35), (33), (27), (26), and (25), but only *Update* (25) is valid yet.

Nonetheless, it can be very time consuming to first obtain, then order, and finally fast forward all the updates from the initial state onward in order to construct the requested bi-temporal state of the RDF dataset. Therefore, we introduce three improvements to reduce the retrieval time of a version. The first improvement let a Linked Data user set up a fully materialised version of a state of the dataset (*Snapshot*). We can now construct the requested state from a *Snapshot* with probably fewer updates than from the *Initial Revision*. In Section 6.2.1 we explain how we can construct a version or state from such a snapshot. In Section 6.2.2 we describe our second improvement. This improvement makes it possible to query the updates that specifically contain the quads corresponding to the quad pattern in the SPARQL query. Finally, Section 6.2.3 gives an explanation of the third improvement: aggregating the *Updates*. Since the *Updates* can be inverted, it is

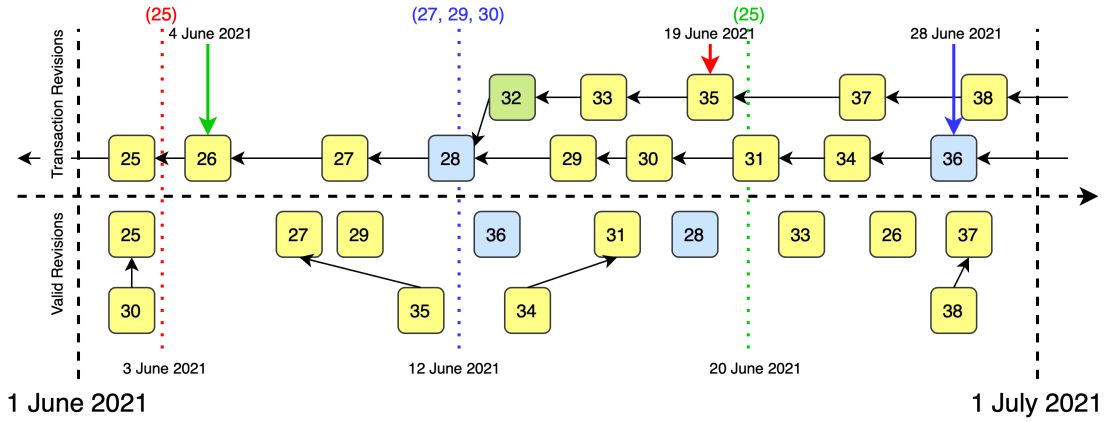


FIGURE 6.5: Visualisation of valid and transaction revisions between 1 June and 1 July 2021.

possible to take all updates together and cancel out the same added and deleted triples in order to leave out the sorting process. In the last Section 6.2.4 we give a brief summary of the complete version retrieval process.

6.2.1 Constructing a state of RDF dataset from snapshot

In order to construct a state of the RDF dataset $D_{i,j}$ from a *Snapshot* we first require to determine which *Snapshot* to take. The state of the *Snapshot* and of the request dataset might differ in their valid time as well as their transaction time, but to determine their difference in transaction time might be more complicated. Therefore, in our thesis we take for convenience the *Snapshot* that is closest to the requested state $D_{i,j}$ based on the valid time. Although, it is very easy to compute the difference in transaction time for the implicit and combined reference strategy. In addition to time, the closest *Snapshot* must also take into account whether the requested state and the snapshot are on the same chain of transaction revisions. For example, it is not possible to reach *Snapshot* (36) from *Update Revision* (37) in Figure 6.5.

After we have discovered the closest *Snapshot*, we query this *Snapshot* to establish the beginning of the version construction process by means of a CONSTRUCT query. This CONSTRUCT query contains the triple from the users' SPARQL query. And we add this CONSTRUCT query result to a temporal RDF dataset. Then we need to retrieve the updates between the closest snapshot and the requested state of the RDF dataset. Whether we should fast-forward or rewind the *Updates* depends on their transaction and valid time. Suppose the closest *Snapshot* has valid time l and transaction time k, $(D_{l,k})$, and the requested dataset state has a valid time i and transaction time j, $(D_{i,j})$. The *Updates* all have a start time t_s, and end time t_e. And, since the modified and reverted updates should be omitted, we retrieve the most recent updates: the updates with only an effective date t^t.

Nevertheless, we encounter a notable problem when we bring state $D_{l,k}$ into state $D_{i,j}$, because users could modify or revert an *Update*. Therefore, a *Snapshot* might contain the updated *Updates*, whereas the requested dataset still contains the old *Updates* ($j < k$). As illustrated in Figure 6.5, *Snapshot* (36) contains *Update* (30), while the dataset at revision number (27) still contains *Update* (25). Or a *Snapshot* still contain the old *Updates*, whereas the requested dataset already contains the updated *Updates* ($j > k$). As illustrated in Figure 6.5, *Snapshot* (28) still contains *Update* (25), while the dataset at

revision number (31) already includes *Update* (30). It means that we first must rewind or later fast-forwards these modified or reverted updates for constructing a state. The definitions below explain how we can convert one state to another state. While definition 10 describes how we can bring two states with the same valid time (l) from one to the other (from $D_{l,k}$ to $D_{l,j}$), definition 11 describes how we can bring two states with the same transaction time (k) from one to the other (from $D_{l,k}$ to $D_{i,k}$).

Definition 10 (From $D_{l,k}$ to $D_{l,j}$)**.** *To bring state $D_{l,k}$ into state $D_{l,j}$ we must either rewind or fast-forwards the latest updates having a difference in transaction time ($j \neq k$).*

- *($j < k$) We must first rewind all updates having $t_s^v \leq l \leq t_e^v$ and $j < t^t \leq k$. We must subsequently fast-forward the updates having $t_t \leq j$ and $t_s^v \leq l \leq t_e^v$ which are modifications of the updates having $j < t_t \leq k$ and $t_s^v \leq l \leq t_e^v$.*
- *($j > k$) We must first rewind the updates having $t_t \leq k$ and $t_s^v \leq l \leq t_e^v$ which are modifications of the updates having $k < t_t \leq j$ and $t_s^v \leq l \leq t_e^v$. We must subsequently fast-forwards all updates having $t_s^v \leq i \leq t_e^v$ and $k < t^t \leq j$.*

Definition 11 (From $D_{l,k}$ to $D_{i,k}$)**.** *To bring state $D_{l,k}$ into state $D_{i,k}$ we must either rewind or fast-forwards the latest updates having a difference in valid time ($i \neq l$).*

- *($i < l$) We must rewind all updates having $i < t_s^v \leq l$, $t_e^v > l$, and $t^t \leq k$, and fast forward all updates having $t_s^v < i$, $i \leq t_e^v < l$, and $t^t \leq k$.*
- *($i > l$) We must fast-forwards all updates having $l < t_s^v \leq i$, $t_e^v > i$, and $t^t \leq k$, and rewind all updates having $t_s^v < l$, $l \leq t_e^v < i$, and $t^t \leq k$.*

Below we give for each scenario an example to illustrate how we can covert the state of the closest snapshot $D_{l,k}$ in the state of requested RDF dataset $D_{i,j}$.

$j = k$ *Same Transaction Time*

$i = l$ Both the valid time and the transaction time have the same value. Therefore, we we can directly query the *Snapshot*, since the state of the *Snapshot* exactly corresponds to the state of the dataset.

$i < l$ The valid time of the dataset is smaller than the valid time of the snapshot, which is illustrated by the red dotted line in Figure 6.6. By following Definition 11, we can construct $D_{i,j}$ from $D_{l,k}$. We rewind update (1), (5), and (9), and fast-forward update (3), (4) and (8).

$i > l$ The valid time of the dataset is larger than the valid time of the snapshot, which is illustrated by the green dotted line in Figure 6.6. By following Definition 11, we can construct $D_{i,j}$ from $D_{l,k}$. We rewind update (1), (2), and (5), and fast-forward update (6).

$j < k$ *Smaller Transaction Time*

$i = l$ Although, both states have the same valid time, the transaction time of the dataset is smaller than the transaction time of the snapshot, which is illustrated in Figure 6.7 by the blue, and black color of $D_{l,k}$, and $D_{i,j}$ respectively. By following Definition 10, we can construct $D_{i,j}$ from $D_{l,k}$. Thus, we rewind update (1) and (5).

$i < l$ Since the state $D_{i,j}$ has a smaller transaction time than $D_{l,k}$, we first bring state $D_{l,k}$ into state $D_{l,j}$ by following Definition 10. Thus, state $D_{l,k} = \{1,2,5,7,9\}$ from Figure 6.7 becomes state $D_{l,j} = \{2,7,9\}$. However, the valid time of the dataset is smaller than the valid time of the snapshot, which is illustrated by

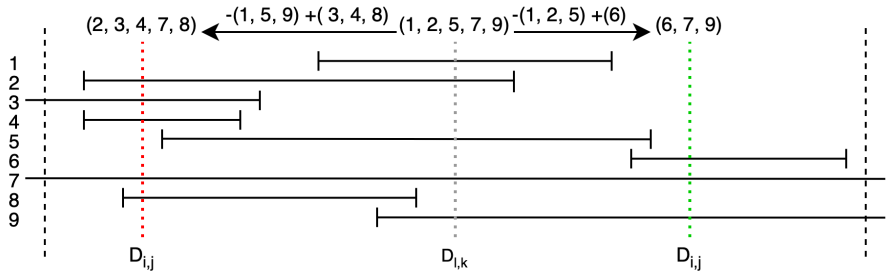


FIGURE 6.6: Constructing the dataset state $D_{i,j}$ from the snapshot state $D_{l,k}$, where $j = k$.

the red dotted line in Figure 6.7. By following Definition 11, we can construct $D_{i,j}$ from $D_{l,j}$. Hence, we rewind update (9), and fast-forward update (3) and (8)

$i > l$ Again since the state $D_{i,j}$ has a smaller transaction time than $D_{l,k}$, we first bring state $D_{l,k}$ into state $D_{l,j}$ by following Definition 10. However, the valid time of the dataset is larger than the valid time of the snapshot, which is illustrated by the green dotted line in Figure 6.7. By following Definition 11, we can construct $D_{i,j}$ from $D_{l,j}$. Thus, we only rewind update (2).

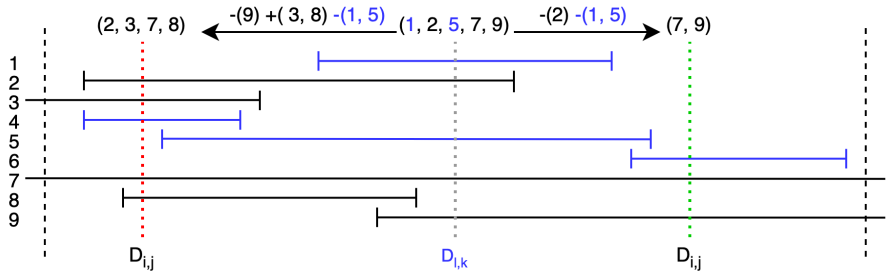


FIGURE 6.7: Constructing the dataset state $D_{i,j}$ from the snapshot state $D_{l,k}$, where $j < k$.

$j > k$ *Larger Transaction Time*

$i = l$ Although, both states have the same valid time, the transaction time of the dataset is larger than the transaction time of the snapshot, which is illustrated in Figure 6.8 by the blue, and black color of $D_{i,j}$, and $D_{l,k}$ respectively. By following Definition 10, we can construct $D_{i,j}$ from $D_{l,k}$. We fast-forward update (1) and (5).

$i < l$ We first bring state $D_{l,k}$ into state $D_{i,k}$ by following Definition 11, since the valid time i is smaller than valid time l, which is illustrated by the red dotted line in Figure 6.8. We rewind update (9), and fast-forward update (3) and (8), and thus, state $D_{l,k} = \{2,7,9\}$ now becomes state $D_{i,k} = \{2,3,7,8\}$. We subsequently bring state $D_{i,k}$ into state $D_{i,j}$ by following Definition 10. Thus, we fast-forward update (4).

$i > l$ Again we first bring state $D_{l,k}$ into state $D_{i,k}$ by following Definition 11, since the valid time i is larger than valid time l, which is illustrated by the green dotted line in Figure 6.8. We rewind update (2), and thus, state $D_{l,k} = \{2,7,9\}$

now becomes state $D_{i,k} = \{7,9\}$. We subsequently bring state $D_{i,k}$ into state $D_{i,j}$ by following Definition 10. Thus, we fast-forward update (6).

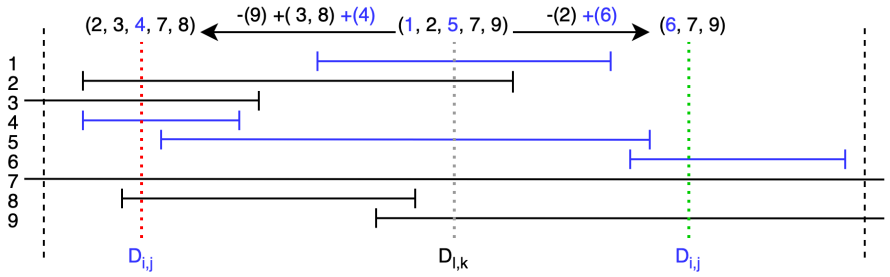


FIGURE 6.8: Constructing the dataset state $D_{i,j}$ from the snapshot state $D_{l,k}$, where $j > k$.

6.2.2 Retrieving updates based on a triple pattern

The easiest and most naive strategy to construct a specific version would be to query all updates enclosing both time intervals. For instance, suppose we would like to construct version F from version A, as showed in Figure 6.9. Since all updates have the same valid time, we must obtain all six updates to construct version F from A. However, only some updates actually include the triples that are needed to answer the users' basic triple pattern query. For instance, if a user would like to know all labels of the IRI `http://recipehub.nl/recipes#FrenchCuisine`, the grey and purple updates in Figure 6.9 do not include a triple that matches this triple pattern. Therefore, Bi-VAKs does not need to query all updates to return a query response to the user. It only obtains those updates that in fact match the triple pattern stated in the SPARQL query, which are most probable less updates than all updates.

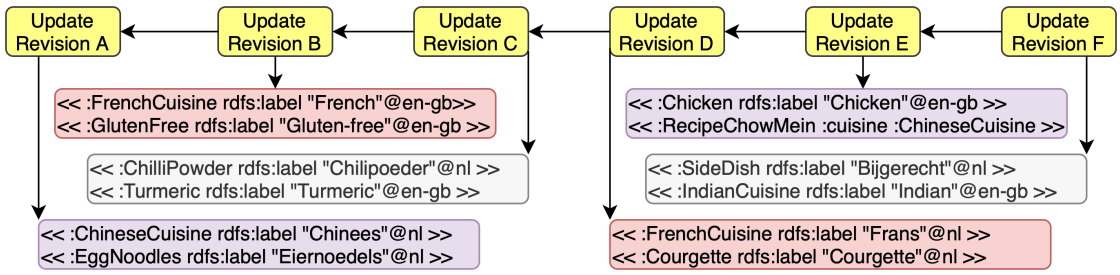


FIGURE 6.9: Representation of six different *Updates Revisions* and their corresponding *Updates*.

Due to the compact representation of SPARQL-star, we only need to add a single line to the SPARQL queries demonstrated in Listing 6.7, 6.8, and 6.9 in order to query the updates that match the basic triple pattern. Listing 6.10 presents an example of such a line of SPARQL. It selects all updates that either inserts or deletes a triple that matches ⟨ recipes:FrenchCuisine rdfs:label ?label ⟩.

LISTING 6.10: A SPARQL query line to query an update matching a basic triple pattern.

```
PREFIX recipes: <http://www.recipehub.nl/recipes#> .
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

?update ?p << recipes:FrenchCuisine rdfs:label ?label >>
```

Although we only consider basic triple pattern queries due to shortage of time, there also arises a significant problem when a users' query contains multiple triple patterns. Often, the variables in one triple pattern correspond to the variables in another triple pattern, so these triple patterns are interdependent. Furthermore, a single update might not include all triple patterns. These matching triples might be spread out over multiple updates. It means that we require to query over all updates, and we must define independent update variables in the SPARQL query. If we do not query over all updates, but over a select group of updates, some variables might stay undefined. For instance, suppose we would like to obtain all recipes having the label '"Chinees"@nl' between version D and F, as illustrated in Listing 6.11. The update that defines the cuisine having 'Chinees"@nl' as label ('ChineseCuisine') belongs to version C. Therefore, '?cuisine' will never be bound to 'ChineseCuisine' between version D and F. '?cuisine' stays undefined, and thus we are not able to determine the recipes with the 'ChineseCuisine' between version D and F. A simple strategy to solve these issues is to query all the updates which match one of the triple patterns, which is illustrated in Listing 6.12. This strategy is less specific, but it still filters updates, which are absolutely not relevant for the users' query.

LISTING 6.11: SELECT Query containing two triple patterns.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

SELECT ?recipe
WHERE {
    ?recipe :cuisine ?cuisine .
    ?cuisine rdfs:label "Chinees"@nl .
}
```

LISTING 6.12: A SPARQL query line to query an update matching a basic triple pattern.

```
PREFIX recipes: <http://www.recipehub.nl/recipes#> .
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

{ ?update ?p << ?recipe recipes:cuisine ?cuisine >> }
UNION
{ ?update ?p << ?cuisine rdfs:label "Chinees"@nl >> }
```

6.2.3 Aggregating updates

The last improvement we introduce is the aggregation of updates. Instead of first sorting the updates based on their transaction time, we take them together as a single collection of inserted and deleted quads, and remove all quads from the collection, which are both inserted and deleted. Thus, we insert quad q if $I > D$, and delete quad q if $i < D$, where I is the number of times q is inserted, and D the number of times q is deleted. However, in addition to rewinding updates we can only aggregate updates when they are invertable. Figure 6.10 gives a similar illustration as Figure 6.4, but now all updates can be inverted. It demonstrates that we only need to insert triples b, c, and g if we want to construct the blue state. The triples a, d, and e are canceled out.

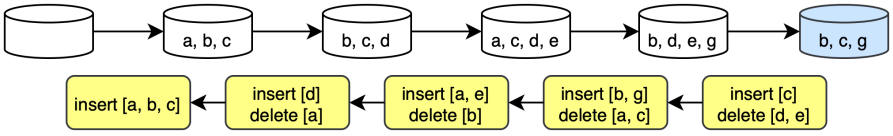


FIGURE 6.10: Visualisation of the influence of invertible updates to the states of the RDF dataset.

6.2.4 Constructing an arbitrary state of the RDF dataset (version)

In the previous sections we explained all steps to construct a version or state of the RDF dataset to allow for retrospective and historical SPARQL queries. In this section we bring everything together, and summarise the steps for constructing a version.

Step 1 *Latest Transaction Revision* When an user does not explicitly give a transaction time, we always take the most recent transaction time. This time always corresponds to the latest transaction revision: the revision to which no other revision is referring. Moreover, if a branch name is given we search for the latest transaction revision in that branch.

Step 2 *Closest Snapshot* Based on the transaction revision and the given valid time, we can now determine which *Snapshot* is the closest to the valid time. If we find a *Snapshot*, we query this *Snapshot* with a CONSTRUCT query, which is based on the user's SPARQL query, and we use its result as start point of our constructed version.

Step 3 *Query Updates* Based on the closest *Snapshot* we now retrieve the updates from the revision-store, as we have explained in section 6.2.1, and 6.2.2. Then we aggregate these updates, and we add the inserted quads to the constructed version.

6.3 Bi-Temporal SPARQL Query

As stated in the introduction retrieving both prior Linked Data versions, and the changes between them could provide Linked Data users relevant information, especially for couple historical and retrospective queries. Therefore, Bi-TR4Qs allows its Linked Data users to do different bi-temporal SPARQL queries. A Linked Data user can query a (prior) version, which we call a version materialisation (VM) query (Section 6.3.1). An user can also query the difference between two versions directly, which we call a delta materialisation (DM) query (Section 6.3.2). And, to determine which versions or states of the RDF dataset are able to return an answer to that SPARQL query, an user can do a version query (VQ) (Section 6.3.3). In this thesis we only focus on single triple pattern queries. Note that these queries do not cover the full spectrum of SPARQL queries. Nonetheless, triple patterns form the basis for more complex queries.

6.3.1 Version Materialisation (VM) Query

For version materialisation queries we would like to query a specific state or version of the RDF dataset $D_{i,j}$. Therefore, we must first retrieve this state by following the procedure described in Section 6.2; subsequently we must apply the user's query to this state of the RDF dataset, and finally we must return the result back to the user.

6.3.2 Delta Materialization (DM) Query

For delta materialisation queries we would like to obtain the difference in the query result between two different states of the RDF dataset. For instance, if state A returns $+1, +5, -6$, and state B return $+1, -8$, the difference between these states is $+5, -6, -8$. Since the user's SPARQL query only contain a single triple pattern, we can directly query the *Updates*, and we do not need to construct both states first. Therefore, we directly obtain the updates, which correspond to the triple pattern stated in the users' SPARQL query, and subsequently aggregate these updates in order to return this collection of inserted and deleted triples back to the user. However, as we already described in Section

6.2.1, two states can differ in both their transaction time and valid time. Therefore, by following the same proceeding described in Section 6.2.1 we can obtain the inserted and deleted triples between two different states of the RDF dataset.

6.3.3 Version Query (VQ)

For version query we would like to know at which state of the dataset $D_{i,j}$ we can answer the specified query. However, the valid time i could be a continuous unbounded time, and although the transaction time is discrete and bounded, the revision-store might consists of numerous transaction revisions. Therefore, for convenience we consider a *Tag* as a version, because a *Tag* is a reference to a specific state of the RDF dataset. Thus, a version query returns all the *Tags* that return an answer to the specified SPARQL query.

Chapter 7

Bi-VAKs Implementation

In this chapter we concisely describe the prototypical implementation of our change-based Version Control System, Bi-VAKs. This implementation is a realisation of the requirements and concepts described in Chapter 4, 5, and 6. However, due to time limitations Bi-VAKs does not support reversions and mergers yet. Figure 4.7 gives an overview of Bi-VAKs.

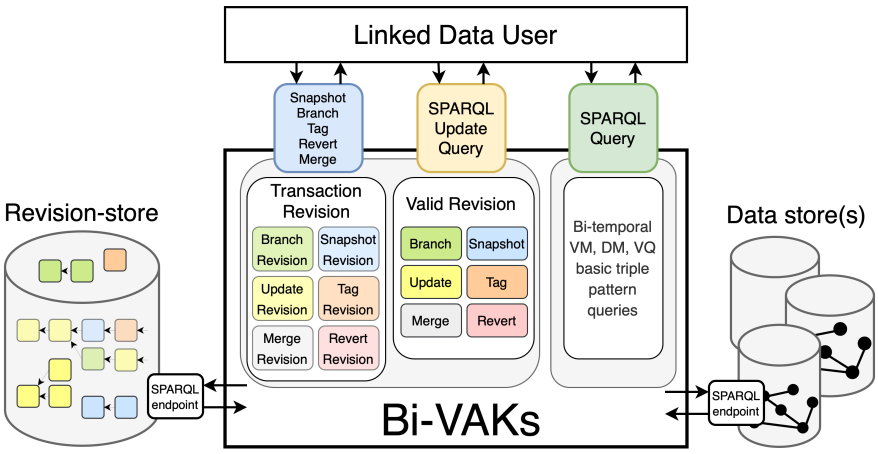


FIGURE 7.1: A general overview of the Bi-Temporal Versioning Approach for Knowledge graphs (Bi-VAKs).

As illustrated in Figure 7.1, Bi-VAKs is a middle-ware between Linked Data users and other RDF applications: the revision-store and some data stores. Bi-VAKs communicates with these triple/quad-store via a standard SPARQL 1.1 endpoint, and it respectively sends SPARQL queries and SPARQL Update queries to retrieve and store the data in these triple/quad-stores. A basic requirements for our middle-ware is that the tools, which Linked Data users already use, do not work on a file basis, but on a triple/quad-store with a SPARQL interface. Thus, to allow users to work with their own tools and Bi-VAKs simultaneously, Bi-VAKs provides a standard SPARQL 1.1 endpoint that currently only supports SPARQL 1.1 Select and Update Queries. These endpoints form the read/write interface on the versioned RDF dataset. In addition to this SPARQL 1.1 endpoint, Bi-VAKs also has endpoints to which Linked Data users can do other versioning requests, such as the creation of a branch, tag, or snapshot. In the subsequent sections we briefly discuss these endpoints in more detail. The prototypical implementation of Bi-VAKs is developed using Python[1]. It uses the Flask API[2] and RDFLib[3] to provide a SPARQL 1.1 interface

[1] https://www.python.org/
[2] http://flask.pocoo.org/
[3] https://rdflib.readthedocs.io/en/stable/

via HTTP. As the underlying quad-store to store the revisions and the snapshots, we use the in-memory store of Apache Jena-Fuseki 4.0.0[4]. Bi-VAKs' implementation code and the code for the experiments - which we further describe in Chapter 8 - can be found at https://github.com/lhmeijer/Bi-VAKs.

7.1 Update Requests

To provide write access to the versioned RDF dataset, Bi-VAKs supports a standard SPARQL 1.1 Update (Gearon, Passant, and Polleres, 2013) endpoint that thus far only supports INSERT DATA or DELETE DATA update operations. Nevertheless, in addition to a SPARQL Update query, in our implementation the user must also add a description and its name to its request in order to track some provenance information, as illustrated in Figure 7.2. In the figures below the bold lines around the input data indicate that this data is required. As explained in Section 5.2.2, an user can also add a valid time to a change by means of a start date, end date, or both dates. Furthermore, in order to modify a diverged versioned RDF dataset (also known as a branch), an user can add the corresponding branch name to its request. After Bi-VAKs has received this update request, it first extracts all modifications from the SPARQL Update query and all other remaining data from its request. Secondly, it checks for each modification whether we can insert or delete this modified quad into or from the RDF dataset. Then it constructs a *Update Revision* and *Update*, and finally, it stores these revisions in the revision-store by sending an SPARQL 1.1 Update query to the quad-store serving as the revision-store. Furthermore, it also deletes the previous *Head Revision* of the chain of transaction revisions, and it add a new *Head Revision* that refers to this newly added *Update Revision*: the latest added transaction revision. From the *Head Revision* we can determine the latest transaction revision for the main stream, and the other branches in the revision-store.

In addition to creating a new update, it is also possible to modify an existing update by sending a similar request to an additional endpoint with the update identifier in the URL, as illustrated in Figure 7.2. This existing update has not already been modified in the same chain of transaction revisions. From this request Bi-VAKs first extracts the included data. However, Bi-VAKs uses the data from the modified update if no data is included. Then, Bi-VAKs checks whether this modification is permitted by the "invertibility check", and then constructs both a *Update Revision* and *Update*. In addition, it let the new *Update* refer to the modified *Update*.

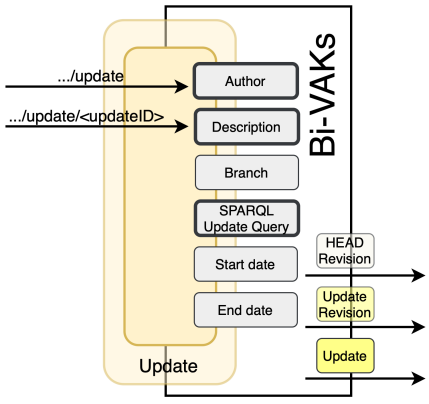


FIGURE 7.2: Illustration of an update request.

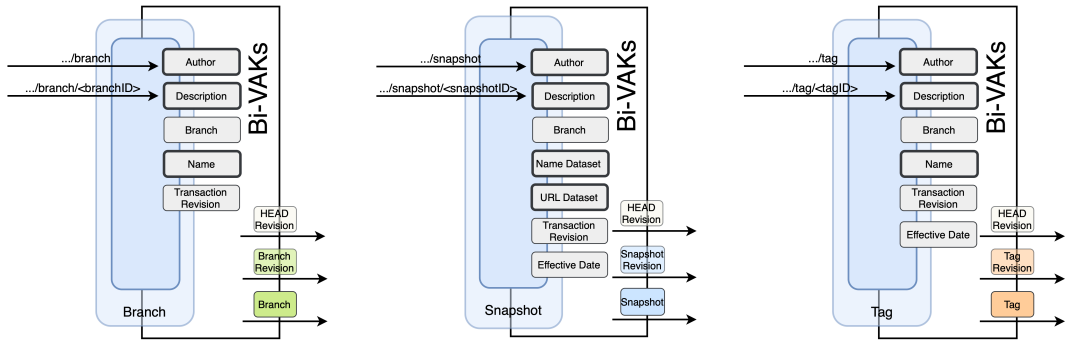
[4] https://jena.apache.org/documentation/fuseki2/

7.2 Other Versioning Requests

In addition to an update request, Linked Data users can also do other versioning requests to Bi-VAKs, as illustrated in Figure 7.1 and 7.3. These requests are rather similar to an update request, but they create a *Branch*, *Snapshot*, or *Tag* instead. In Figure 7.3a we first demonstrate the branch request, which Linked Data users can send in order to diverge from the common RDF dataset. This branch request must always contain a branch name, as showed in Figure 7.3a. In addition, they could add a transaction revision at which the new branch should branch off from. And, they could add a branch name of an existing branch, if the new branch should branches off from that particular branch. If a transaction revision is not given, the new branch always branches off from the head of the chain of transaction revisions. This chain depends on whether the new branch should branch off from the main stream or from another branch. In addition, an user could also send a branch request with a branch identifier in the URL in order to modify an existing branch. This branch must always be the latest branch. You cannot modify a branch which is already modified somewhere in the past. From both branch requests Bi-VAKs constructs a *Branch Revision* and a *Branch*, and it stores them in the revision-store. Furthermore, it also creates a new *Head revision*, which points to the this newly added *Branch Revision*. It only deletes the previous *Head Revision* if Bi-VAKs modifies an existing branch in order to prevent a new branch from being created.

The next versioning operation request is the snapshot request, as presented in Figure 7.3b. The snapshot requests creates a materialised version of a state of the RDF dataset based on the information in the requests. The effective date and the transaction revision respectively denote the valid time and the transaction time of this state. However, if the user does not include an effective date to its request, Bi-VAKs uses the current date to denote the valid time. And if no transaction revision is given, Bi-VAKs uses the head of the chain of transaction revisions to denote the transaction time. Bi-VAKs first creates this state of the RDF dataset by retrieving the corresponding update from the revision-store, and stores this state in a new triple/quad-store. This triple/quad-store has a standard SPARQL 1.1 endpoint, and it can be queried by a dataset name and URL included in the user's request. Similar to the other versioning operation requests, an user can also modify an existing snapshot by adding a snapshot identifier to the URL of the request. Bi-VAKs creates for both snapshot requests a *Snapshot*, and a *Snapshot Revision*. It deletes the previous *Head Revision*, and it inserts a new one referring to the latest *Snapshot Revision*.

The last versioning operation request that is implemented in Bi-VAKs is the tag request. As displayed in Figure 7.3c, a tag request always need a name to label a specific state of the dataset. A state is based on a valid time and a transaction time. Similar to a snapshot request, the valid time is either denoted by a given effective date or by the current date, and the transaction time is either indicated by a given transaction revision or the current *Head Revision*. Again, an user can modify an existing tag by adding the tag identifier to the URL of the request. Bi-VAKs creates for both tag requests a *Tag*, and a *Tag Revision*. It deletes the previous *Head Revision*, and it inserts a new one referring to the latest *Tag Revision*.



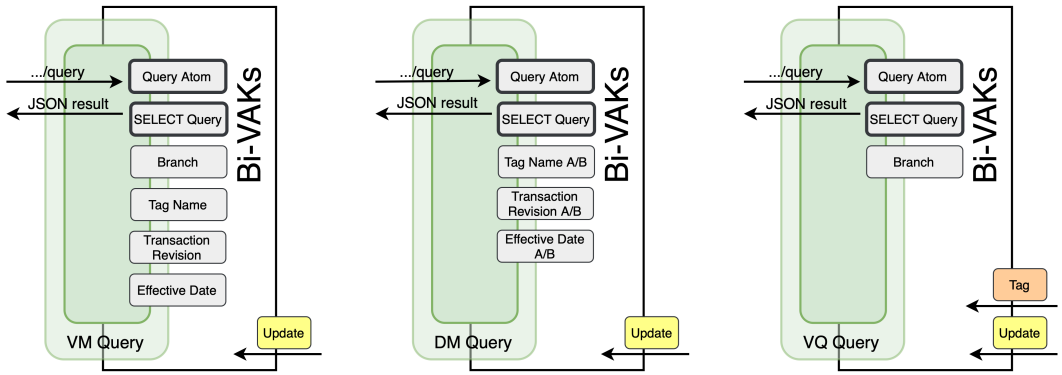
(A) Branch request (B) Snapshot request (C) Tag request

FIGURE 7.3: Illustration of three different versioning operation requests.

7.3 Version Materialisation, Delta Materialisation & Version Query Requests

To provide read access to a versioned RDF dataset Bi-VAKs has a standard SPARQL 1.1 (Prud'hommeaux and Seaborne, 2008) endpoint that so far supports version materialisation (VM), delta materialisation (DM), and version (VQ) basic triple pattern SELECT queries. A Linked Data user must send this query atom (VM, DM, and VQ) and the SELECT query via a GET request to Bi-VAKs. If an user sends a VM query request, Bi-VAKs first constructs the requested state of the RDF dataset; applies the SELECT query on this state; and sends its response back to the user in JSON Format. As explained in Section 6.2, this state can either be constructed sorted or aggregated, from the initial revision or from a snapshot, and all updates or only specific updates. The user can include the requested state in many ways to its query request. A user can sends a tag name that is referring to an existing *Tag*. A user can include a transaction revision and an effective date to directly indicate the transaction and valid time. Or the transaction and valid time are respectively based on the *Head Revision* and the current date.

For a DM query request, an user must send the state information of two states, as for this query it is required to determine the difference between the query results over two different states of the RDF dataset. Similar to the VM query request, these states are either based on a tag name, a transaction revision, or an effective date. After Bi-VAKs has extracted this state information, it either constructs both versions separately, and it compute their difference over the SELECT query results, or it directly determine their difference over the in-between updates. Subsequently, it returns the results of the inserted and deleted triples back to user in JSON format. The VQ query request works a little differently than the other two. The version query returns the versions for which the SELECT query gives a results. Instead of seeing each transaction revision as a version, we indicate a *Tag* as a version. A *Tag* consists of an effective date and a transaction revision to indicate the valid and transaction time. Therefore, Bi-VAKs first determine all *Tags* in the chain of transaction revisions, which, additionally, depends on a given branch name. Subsequently, it either constructs each state of the RDF dataset separately, or it uses the previous state and the in-between updates to construct a version. After applying the SELECT query on each version, it returns a result in JSON format back to its user, including the tag name and its corresponding query result if the state indicated by that tag gives a result for the user's query.



(A) VM query request (B) DM query request (C) VQ query request

FIGURE 7.4: Illustration of the three different query requests.

Chapter 8

Evaluation

In this chapter we evaluate the ingestion and query performance of our prototypical Linked Data Version Control System, Bi-VAKs, on different input settings. In other words, in this chapter we assess the ingestion and query performance of our prototypical implementation described in Chapter 7 on some design decisions we described in Chapter 5 and 6. In general, the performance of RDF versioning approaches can be assessed by the storage size, by the ingestion time of new changes, and by the look up time of different query types. These type of queries queries the (prior) versions (Version Materialisation (VM)), queries the differences between versions (Delta Materialisation (DM)), or queries all the versions returning a solution (Version Query (VQ)). However, the aim of our experiments is not to compare Bi-VAKs' performance with the performance of existing Linked Data versioning approaches, such as OSTERICH (Taelman et al., 2019) and QuitStore (Arndt et al., 2019). Because, these approaches only consider changes having a single time dimension. Our experiments serves as a benchmark to evaluate the performance of bi-temporal Linked Data versioning systems. This benchmark, Bi-BEAR, is an extension of the BEAR benchmark (Fernández et al., 2019) in order to evaluate bi-temporal ingestion and query performance. In Section 8.1 we outline in detail how we expand the BEAR benchmark. Besides, we also evaluate the effect of some design decisions of Bi-VAKs on the ingestion and VM, DM, VQ basic triple pattern query performance. Based on these design decisions described in chapter 5 and 6, we give in Section 8.2 an overview of them, the so-called evaluations points, for which we have evaluated Bi-VAKs' performance. In the subsequent sections we give the results of Bi-VAKs' ingestion performance derived from its ingestion time and storage size (Section 8.3), and Bi-VAKs' VM, DM and VQ basic triple pattern query performance based on the time needed to obtain a query result (Section 8.4) on these evaluation points. All these experiments were performed on a MacBook Pro with a 3,1 GHz Dual-Core Intel Core i7 processor and 16 GB of RAM on the local machine.

8.1 Expansion on the BEAR Benchmark (Bi-BEAR)

In this section we describe how we expand the Benchmark for RDF archives (BEAR) proposed by Fernández et al. (2019) from a uni-temporal benchmark to a bi-temporal benchmark (Bi-BEAR). BEAR is a test suite to evaluate the storage space efficiency and the performance of retrieval operations of RDF archiving approaches. BEAR is meant to serve as a baseline for RDF versioning systems, as they performed an empirical evaluation on various archiving strategies and RDF stores. BEAR consists of three main datasets and a corresponding set of SPARQL test queries and their results, namely, BEAR-A, BEAR-B and BEAR-C. Due to time and memory limitations we only use the BEAR-B-Daily. The dataset of BEAR-B contains the 100 most volatile resources from the DBpedia Live changesets (Morsey et al., 2012) over the course of three months (August to October 2015) at three

different granularities: instant (21,046 versions), hour (1,299 versions) and day (89 versions). These datasets only contain triples, and thus no quads. BEAR-B provides a batch of 62 realistic triple pattern queries, which is a mix of (?P?) and (?PO) queries and their VM, DM, and VQ results for the granularities: hour and day. The dataset of BEAR-B, and the queries, and can be found at https://aic.ai.wu.ac.at/qadlod/bear.html.

Nevertheless, the BEAR benchmark evaluates uni-temporal RDF archiving systems instead of bi-temporal systems, such as Bi-VAKs. The versioned triples and the queries in BEAR only have a transaction time, a version number. However, in Bi-VAKs both changes and queries have a transaction time as well as a valid time. Thus, we must alter the dataset to add an extra time dimension to the system's changes and to the queries. A simple solution would be to put all modifications between two consecutive BEAR versions into a single *Update* and add a time interval that overlaps the effective date of all queries. This time overlap is required because otherwise the Bi-VAKs might return other VM, DM and VQ results than the results of BEAR-B. However, in this solution all updates have the same valid time, and we still evaluate an uni-temporal versioning system. Therefore, we divide the changes between two consecutive BEAR versions into multiple *Update Revisions* and their corresponding *Updates*, and we add a different start and end date to these *Updates*. We do so by having an artificial Linked Data user send an HTTP request to the SPARQL Update endpoint of Bi-VAKs containing a SPARQL Update query and a randomly selected start and end date. Such a SPARQL Update query contains a set of inserted and deleted triples randomly taken from the changes between two consecutive BEAR versions. In order to ensure that the query results do not differ from the actual BEAR results, the *Updates* - which matches with one of the queries - must have a time interval that overlaps the effective date of that query. Although for convenience all queries have the same effective date for each version, we can now assess the effect of both the transaction and valid time. However, since a BEAR version now consists of multiple updates (revisions), we require another method to refer to a BEAR version. Therefore, we create for each BEAR version a *Tag*. As in Bi-VAKs each version is bi-temporal, these tags also have a different transaction and valid time. This valid time corresponds to the effective date of the queries.

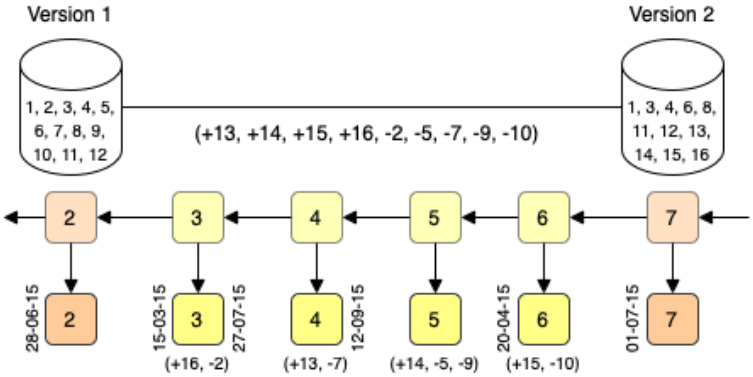


FIGURE 8.1: Example of transforming changes between two consecutive BEAR versions into *Update Revisions* and *Updates*.

Figure 8.1 illustrates an simplified example of extending the changes between version 1 and version 2 into *Update Revisions* and *Updates*, and creating a *Tag* for each version. Due to these adaptations we can no longer compare the Bi-VAKs' performance with the performance of BEAR and other versioning approaches. The number of *Updates* might

affect the speed of constructing a version, and a restriction on an effective date may slow down the retrieval of updates. In addition, we have made our versioning system more realistic by adding some metadata to each *Update Revision*, such as a description, an author and creation date. As a result, our revision-store contains more triples than other change-based versioning approaches, which only stores the changes between versions.

8.2 Evaluation Points to Assess Bi-VAKs

In the previous section we described how we have expanded the BEAR benchmark, and that we divided the changes between two consecutive BEAR versions into multiple *Update Revisions* and *Updates*. In addition to adjusting this number of modifications per *Update* and their time interval, we can vary various other input parameters in Bi-VAKs, which might affect Bi-VAKs' performance. These input parameters or so-called evaluation points (**EVA**) are based on certain design decision we have discussed in chapter 5 and chapter 6. In this section we outline these evaluations points, and demonstrate in the subsequent sections their impact on the Version Materialisation (VM), Delta Materialisation (DM) and Version (VQ) query performance, and storage costs of Bi-VAKs. However, to limit the number of experiments we did not consider all design decisions. One of the design decision we did not evaluate is named graphs. Although Bi-VAKs supports named graphs, since BEAR-B-Daily only contains triples, we does not include them in our experiments, and we refer to future work how we can extend our experiments to include them.

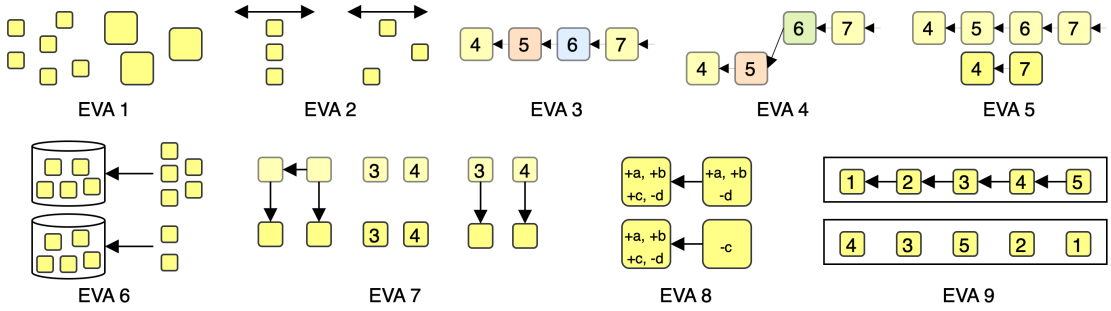


FIGURE 8.2: A global illustration of the different evaluation points.

Figure 8.2 gives a comprehensive illustration of the nine different evaluations points, which we further discuss below:

EVA1 *Modifications per Update* The number of modifications per update affects the total number of updates between two consecutive versions. The more updates in the revision-store the more updates Bi-VAKs requires to assess whether they are compatible with the transaction and valid time and the more storage space is needed. In our experiments, we either consider **50**, or **100** modifications per update.

EVA2 *Valid Time Interval* By varying the valid time interval of each update, we could influence the number of updates that are needed to a construct a version having a certain valid time. The more the updates' time intervals overlap the more updates must be queried for an overlapping valid time. We determine these time interval by first generating a date (t) in the middle of the interval from a normal distribution with a constant mean (μ_t =1 July 2015) and a variable standard deviation (σ_t). σ_t is either 1000000 (**LC**) or 5000000 seconds (**HC**). And then we generate a width (w) of the interval from a normal distribution with a constant standard deviation

(σ_w =1 day) and a variable mean (μ_w). μ_w is either 5 days (**LW**) or 50 days (**HW**). We now compute the start date by $t - w$ and the end date by $t + w$. Since we assume a maximum time interval of a year, we give updates no start date or no end date if respectively their start date is negative and end date is larger than 365 days.

EVA3 *Snapshot* Instead of storing only the changes we could also store a full materialised version of a state of the dataset (*Snapshot*). Therefore, we do not need to retrieve all updates ever made to the revision store, but only the differences between two states of the revision-store. In our experiments, we have created such a snapshot after the 45th version with an effective date of 1 July 2015 (**S45**).

EVA4 *Branches* In Bi-VAKs it is possible to branch off from the common dataset or from another branch. Branches might affect the query performance. Every branch branches off from the preceding branch, such that all transaction revisions still form a single chain. In our experiments, we have created a branch after each 3th version (**B3**).

EVA5 *Modified Updates* Linked Data users could also modify a valid revision in a revision-store. Such valid revision modifications affects the number of valid revisions in the revision-store and the references to their preceding valid revisions. Therefore, we modify some existing *Updates* in the revision-store by adding one second to their start and end date, and randomly modify an existing update after every 5 newly created updates (**M5**).

EVA6 *Fetching Strategy* As described in Section 6.2.2, in order to construct a previous version, we can either query all updates (**FA**) or we can only query the updates matching the basic triple pattern specified in the SPARQL query (**FS**).

EVA7 *Reference Strategy* In Chapter 5 we explained three reference strategies to connect a transaction revision with its corresponding valid revision(s). The implicit reference (**RI**) refers via revision numbers and branch indices to the valid revisions. The explicit (**RE**) and combined (**RC**) reference strategies directly refer via an attribute in the transaction revision.

EVA8 *Update Content Strategy* As described in Section 6.1.2, we can represent a *Update* modification in two different ways. We can either copy all modifications of a preceding update to its new update, or we let the new update refer to its predecessor. The update content strategies are respectively called the repeated (**CP**), and the related (**CL**) update content strategy.

EVA9 *Change Order* Constructing a RDF dataset state from its change sets requires to sort them in advance. Sorting (**SO**) might be very time-consuming. However, by having invertible changes sorting is not needed, and we can directly aggregate (**AG**) them. However, checking whether a modification can be invertible, is also time-consuming.

In order to evaluate these evaluations points we create a number of different experiments. These experiments vary in the evaluation points they evaluate. To indicate which evaluations points they consist of, we use their abbreviations as stated above. Suppose an experiment has 100 modifications per update (100), a little overlap (HC-LW), a single snapshot (S45), branches (B3), modified updates (M5), fetches all updates at once (FA), an explicit reference strategy (RE), a repeated update content strategy (CP), and aggregated changes (AG). We indicate to this experiments as 100-HC-LW-S45-B3-M5-FA-RE-CP-AG.

8.3 Ingestion Performance of Bi-VAKs

We evaluate the ingestion performance on storage size and ingestion time. Due to time and memory limitations we only use BEAR-B daily for our experiments. We ingest all its 89 versions, which has a total of 93,131 modifications. In order to assess the query performance of Bi-VAKs, we create a total of 21 different revision stores which vary in a number of storage evaluations points, such as the time interval, update content strategy, etc. We refer to Section 8.2 for the abbreviations of these evaluation points. These 21 revision stores are listed in the left column of Table 8.1. Besides, as we explained in Section 4.3, in order to be able to aggregate modifications and rewind updates an update must be invertible. Therefore, we need to check for each modification whether it can actually be inserted or deleted in the revision-store. This check has a substantial impact on the ingestion time, and hence we have measured the ingestion time both with and without the so-called "invertibility check".

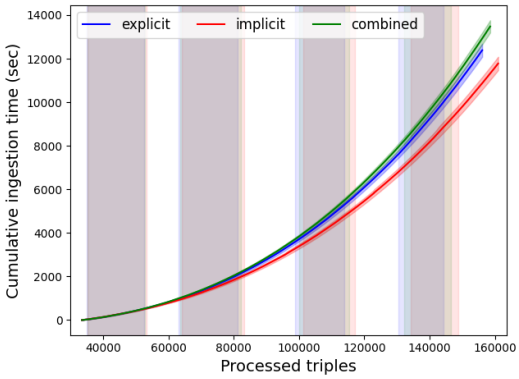
Table 8.1 displays the number of triples, and the storage size for the 21 revision stores in MB. The revision-store with 50 modification per update contains 1912 updates, and the revision-store with 100 modification per update contains 982 updates. Although the storage size of all three reference strategies do not differ much, Table 8.1 still demonstrates that the implicit and combined reference strategy have the highest storage size. In addition, the storage size increases when the modified updates contain all modifications from their preceding update (repeated update content strategy (**CP**)), and stays almost the same when the modified updates refer to its preceding updates for their modifications (related update content strategy (**CL**)). Furthermore, although the storage sizes for the implicit and combined reference strategy are almost the same, they slightly differ when we add branches to the revision-store.

TABLE 8.1: Number of triples and storage size in MB of a revision-store having different input settings

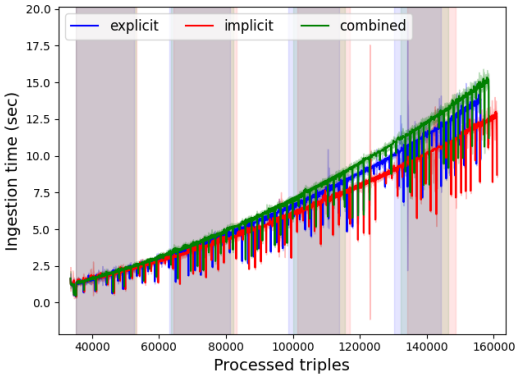
Approach	Number of Triples			Storage Size (MB)		
	Explicit	Implicit	Combined	Explicit	Implicit	Combined
50-LC-HW-CP	148744	150747	150747	44.17	44.53	44.60
50-HC-LW-CP	148562	150565	150565	44.13	44.49	44.56
100-LC-HW-CP	138514	139587	139587	42.03	42.22	42.26
50-LC-HW-CP-B3	149179	155258	153278	44.27	45.45	45.13
50-LC-HW-CP-M5	171468	173853	173853	50.89	51.31	51.40
50-LC-HW-CL-M5	153328	155713	155713	45.18	45.60	45.69
50-LC-HW-CP-S45-M5	171481	173867	173867	50.90	51.32	51.41

In Figure 8.3 we have plotted the median (cumulative) ingestion time and the 25th and 75th quantile time in seconds without and with the invertibility check in the revision-store 50-LC-HW-CL-S45-B3-M5. To calculate the median ingestion time we have repeated the process of constructing a revision-store 5 times. The vertical lines, which form the wide colored bars, indicate the 10th up to 80th version. These lines point out that some version contains more triples than others. Figure 8.3a puts the median cumulative ingestion time with the invertibility check on display, and it has a mean total ingestion time of around 4 hours for all three reference strategies. Figure 8.3c puts the median cumulative ingestion time without the invertibility check on display and it has a mean total ingestion time of around 3 minutes for all three reference strategies. Figure 8.3b shows the median individual ingestion time for each update in seconds with invertibility check, and Figure 8.3d shows the median individual ingestion time without invertibility check. These figures show that the ingestion time increases tremendously when we need to check for each

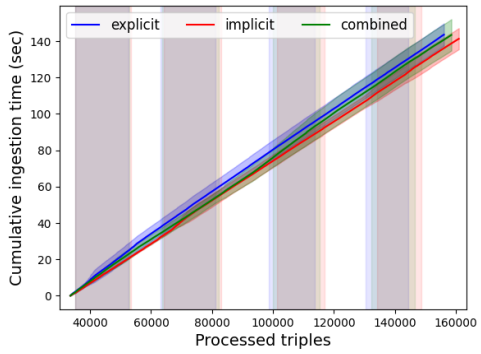
update whether it is invertible. It even further increases when the number of triples grows in the revision-store. As showed in Figure 8.3a, all three graphs have a convex shape. While with invertibility check it takes a bit less time for an implicit reference strategy than for an explicit and combined reference strategy, without invertibility check all three reference strategies almost have a similar ingestion time. In addition, they also do not increase when the number of triples grows in the revision-store, because the graphs in Figure 8.3c have a linear and an almost concave shape.



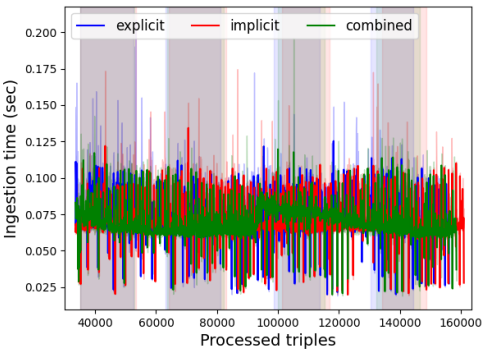
(A) Cumulative ingestion time with invertibility check



(B) Ingestion time with invertibility check



(C) Cumulative ingestion time without invertibility check



(D) Ingestion time without invertibility check

FIGURE 8.3: (Cumulative) ingestion time in seconds (sec) over 89 versions for revision-store 50-LC-HW-S45-B3-M5-CL.

8.4 Query Performance of Bi-VAKs

In this section we evaluate the query performance of Bi-VAKs over version materialisation (VM), delta materialisation (DM), and version (VQ) basic triple pattern queries. We evaluate the query performance by measuring the time it takes to obtain a query result. Due to time and memory limitations we only use BEAR-B daily for our experiments. We evaluate all its 89 versions. BEAR-B provides two query sets that contain ?P? and ?PO queries having respectively 49 and 20 queries. But because of time constrains we only use 20 queries of the ?P? query set for our experiments. Since we look at the median query time over all queries and not at a individual query level, 20 queries might enough to assess the overall query performance. We evaluate each query as VM query for all versions, as DM

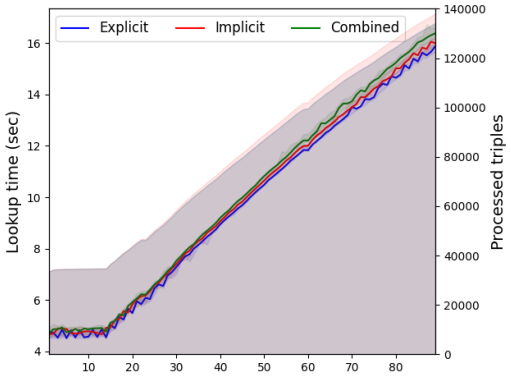
query between the first version and the next fifth version, and as VQ query. We have repeated every experiment 5 times to minimise outliers. To present the VM and DM results in Figures 8.4 and 8.6 we have plotted the median query duration in seconds and its 25th and 75th quantile. We plotted the median and quantiles instead of the mean, because they are less sensitive to outliers than the mean and standard deviation. In Figure 8.5 we plotted the median time and the difference between the 75th quantile and the median, because VQ queries return a single value for each query, and we plot these values in a bar chart instead of a line chart.

8.4.1 Performance of Version Materialisation (VM) Queries

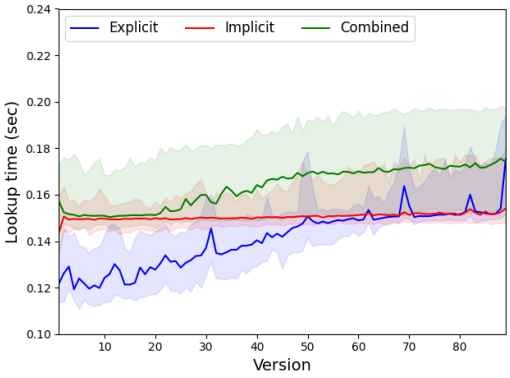
Figure 8.4 presents the median query time that it takes to obtain a result for a Version Materialsation (VM) query for different input settings. With a VM query an user can query a specific version or state of the RDF dataset. In Figure 8.4a we have plotted the median query duration when we query all updates to construct a version. The colored, almost grey, area in Figure 8.4a shows the number of triples needed to obtain a results. Since we query all updates, this number is the same for all 20 queries. For the subsequent query results this number differs, because they query only the specific updates, and these might vary between queries. Therefore, the remaining sub-figures only display the query look up time. Figure 8.4a shows that the more updates and triples we need to construct a version, the more time it takes to obtain a query result. In the subsequent sub-figures this trend is also slightly present, especially for the explicit reference strategy.

Figure 8.4b, 8.4c, and 8.4d present the query time when the number of modifications per update is either 50 or 100; or when the updates overlap a lot or a little. Figure 8.4b and 8.4d respectively show that it takes less time to query a specific version when there are less updates in the revision-store. Figure 8.4b and 8.4c demonstrate that it takes almost the same time to do a VM query when updates overlap a lot or a little. Figure 8.4b, 8.4d, 8.4e, and 8.4f present the time when the updates are either aggregated or sorted for both 50 and 100 modifications per update. For these sorted updates it is not needed to check their invertibility. These figures show that for the sorted updates the query duration mainly increases for the explicit reference strategy when a higher version is evaluated. And, the query time is about the same or even lower for the implicit and combined reference strategy. Although in general it is lower when there are 100 modifications per update instead of 50.

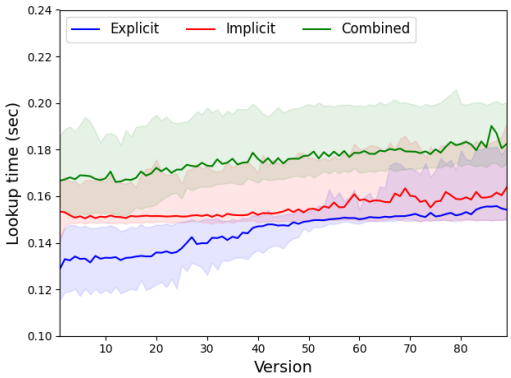
We have plotted in Figure 8.4g, 8.4h, 8.4i, and 8.4j the median query duration when users have created branches in the revision-store; when they have constructed a single snapshot; or when they have modified some updates. Figure 8.4b and 8.4h demonstrate that the query time slightly increases when some updates are modified. Figure 8.4g and 8.4h illustrates that there are small time differences between a related and a repeated update content strategy. For all three strategies the query time is a bit lower for the related content strategy. Figure 8.4b and 8.4i show that the query performance immensely declines when a single snapshot is created and when the versions are constructed using a single snapshot. Figure 8.4i shows that around version 45 the time drops, but starts growing substantially after version 45. Figure 8.4b and 8.4j show that branches considerably affect the query time for the implicit and combined reference strategy, in particular when the number of branches grows. And these figures demonstrates that the time for an explicit reference strategy stays more or less the same.



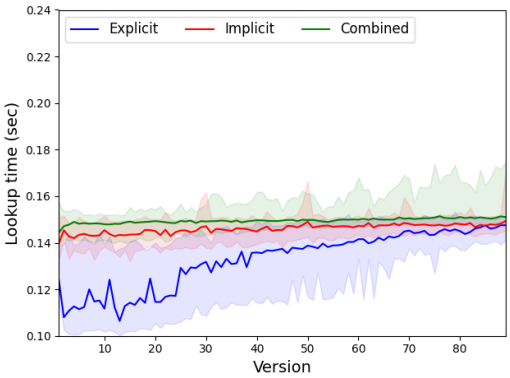
(A) 50-LC-HW-FA-CP-AG-VM



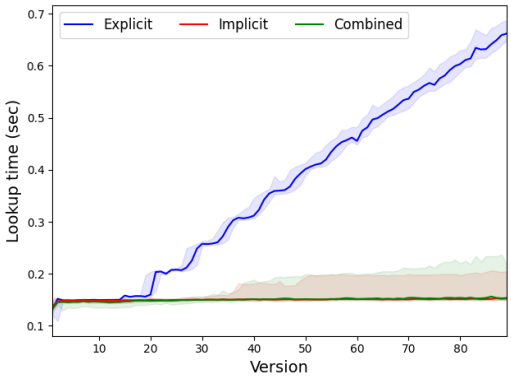
(B) 50-LC-HW-FS-CP-AG-VM



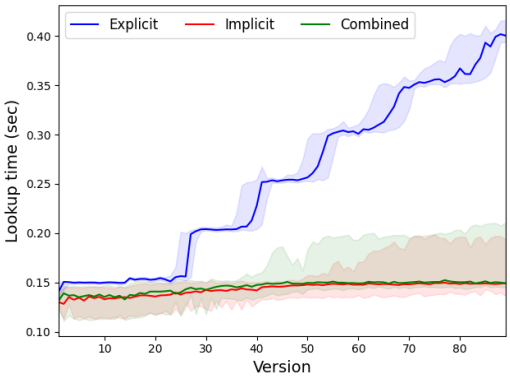
(C) 50-HC-LW-FS-CP-AG-VM



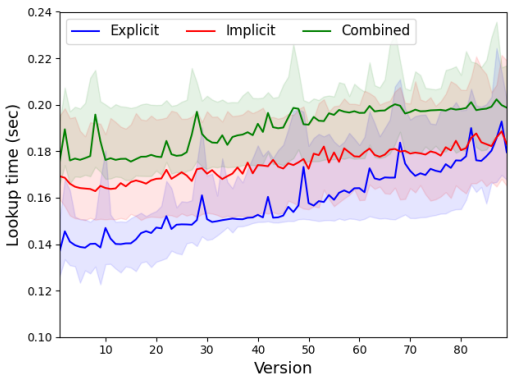
(D) 100-LC-HW-FS-CP-AG-VM



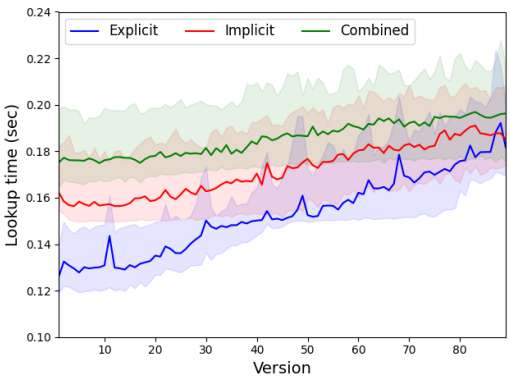
(E) 50-LC-HW-FS-CP-SO-VM



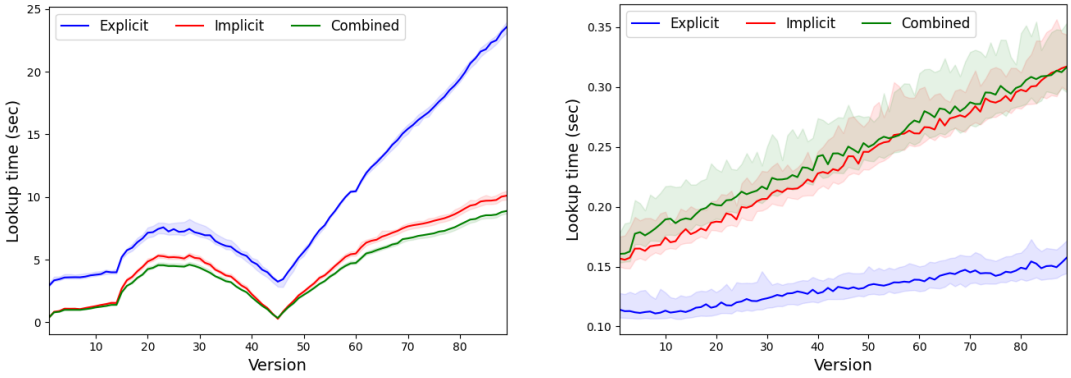
(F) 100-LC-HW-FS-CP-SO-VM



(G) 50-LC-HW-FS-CP-M5-AG-VM



(H) 50-LC-HW-FS-CL-M5-AG-VM



(I) 50-LC-HW-FS-CP-S45-M5-AG-VM (J) 50-LC-HW-FS-CP-B3-AG-VM

FIGURE 8.4: Version materialisation (VM) query look up time in seconds (sec) over 89 versions for different input settings.

8.4.2 Performance of Delta Materialisation (DM) Queries

Figure 8.6 presents the median query time it takes to obtain a result for a Delta Materialisation (DM) query for different input settings. With a DM query an user can query the difference in the query result between two different states of the RDF dataset. These DM queries are computed between the first version (state) and the subsequent fifth version (5, 10, etc.). We have only plotted the results with a fetching strategy of specific, because we determine a DM query result by only obtaining the modified quads between two consecutive versions, and from the previous section (8.4.1) we can conclude that its query look up time is significantly higher. The strategy's performance of querying directly the modifications between two version is displayed in Figure 8.6a, 8.6c, 8.6e, 8.6f, 8.6g, and 8.6i. For these figures we have put the y-axis in log scale to also present the difference between the implicit and combined reference strategy. Besides, instead of querying the modifications directly, in the remaining sub-figures we have plotted the results when we first retrieve both versions and subsequently compute their differences. These versions are retrieved by sorting the updates instead of aggregating the updates. Hence it is not required to check the updates their invertibility.

Figure 8.6a, 8.6b 8.6c, 8.6d, and 8.6e present the query duration when the number of modifications per update is either 50 or 100; when the modifications are either aggregated or sorted; or when the updates overlap a lot or a little. These figures show that for an explicit reference strategy it takes much more time to get a DM query result than for an implicit and combined reference strategy. Although the time goes down a bit if we have 100 modifications per update instead of 50. These figures also demonstrates that for an explicit reference strategy it is quicker to first determine each version separately and then their differences instead of directly obtaining their differences. For an implicit and combined reference strategy computing each version separately seems a bit faster, but not significantly. As illustrated in Figure 8.6a and 8.6e, between much overlap and a little overlap there is almost no difference.

In the remaining sub-figures we have plotted the median query duration when users have created branches in the revision-store, and when they have modified some updates. Again by comparing Figure 8.6a, and 8.6f we can see that branching has some influence on the query performance, especially for the implicit and combined reference strategy. The query time increases when the number of versions and thus branches grows instead of being constant. Figure 8.6g, 8.6h, 8.6i, and 8.6j demonstrates that for all three reference

strategies the query time increases when users have modified existing updates. Besides, whether we use a repeated or a related update content strategy, it hardly affect the query performance, primarily for the implicit and combined reference strategy.

8.4.3 Performance of Version Queries (VQ)

Figure 8.5 presents the median query time it takes to obtain a result for a Version Query (VQ) for different input settings. With a VQ Query an user can query the states of the RDF dataset for which it gets a response to its specified query, and thus we obtain a single query duration for each query. We determine a VQ query by first obtaining the tags and then by constructing the corresponding version based on its previous version. In general, Figure 8.5 shows that the look up time for the explicit reference strategy is always higher than for the implicit and combined reference strategy. These two strategies do not differ much. In addition, the query time increases when users have modified existing updates, or have created branches in the revision-store. The last group of bars in Figure 8.5 presents the query time when we construct each version independently, and sort the updates instead of aggregating the update. This corresponds to the sum of the query times shown in Figure 8.4e. For the explicit reference strategy this VQ query time is significantly lower, while for the other reference strategies it is somewhat lower.

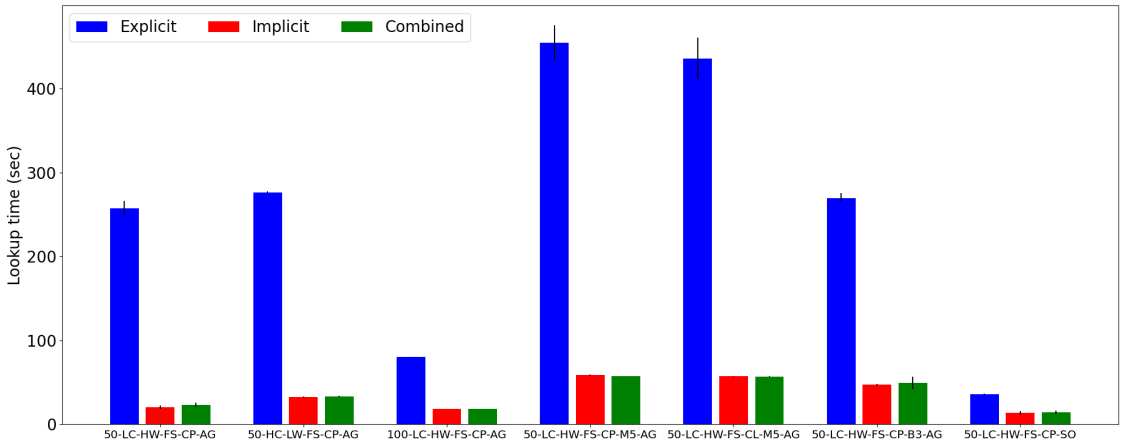
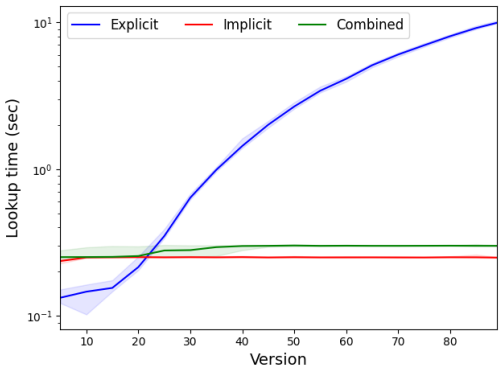
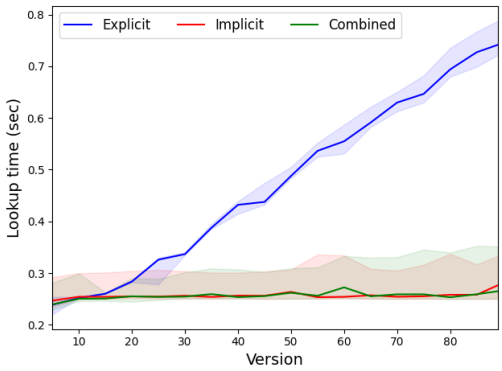


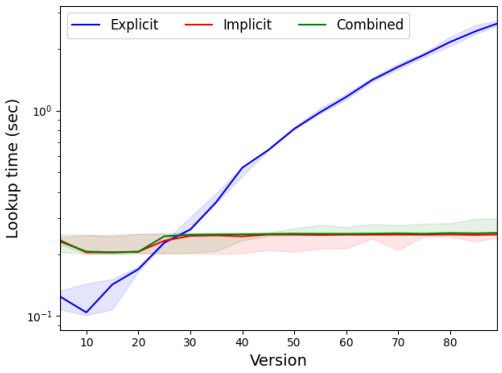
FIGURE 8.5: Version (VQ) query look up time in seconds (sec) over 89 versions for different input settings.



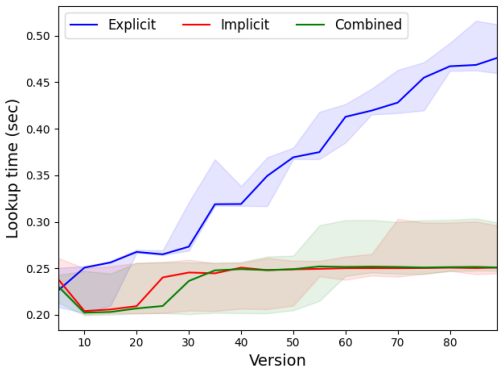
(A) 50-LC-HW-FS-CP-AG-DM-p



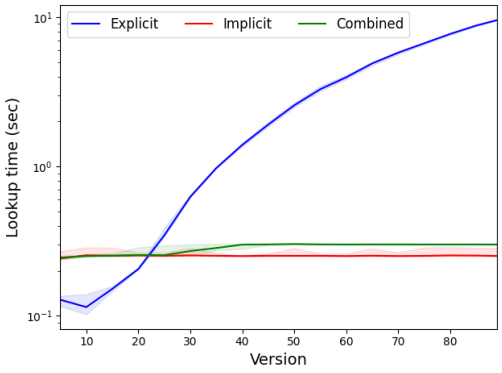
(B) 50-LC-HW-FS-CP-SO-DM-p



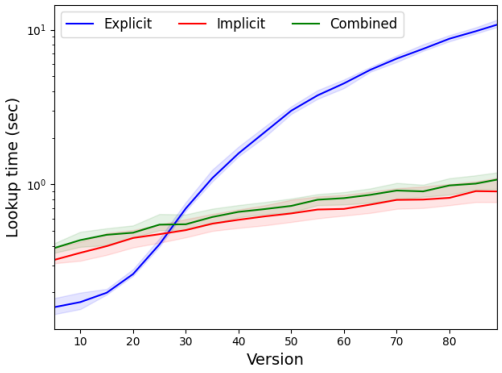
(C) 100-LC-HW-FS-CP-AG-DM



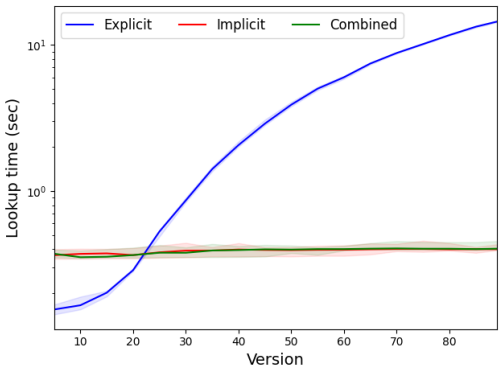
(D) 100-LC-HW-FS-CP-SO-DM



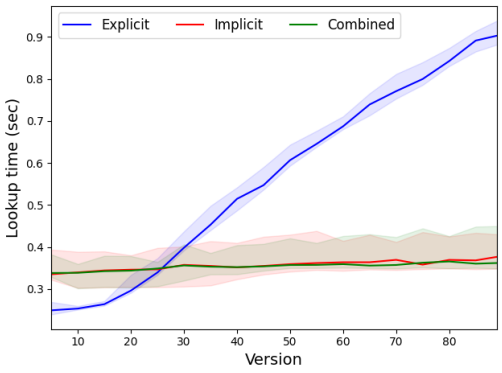
(E) 50-HC-LW-FS-CP-AG-DM



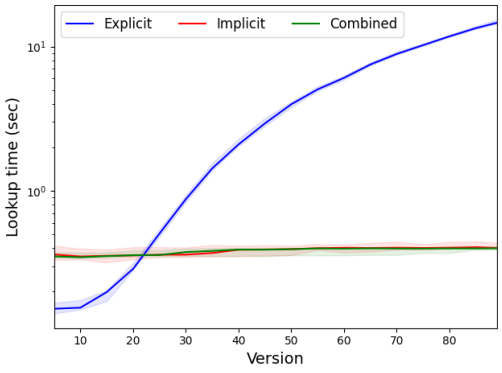
(F) 50-LC-HW-FS-CP-B3-AG-DM



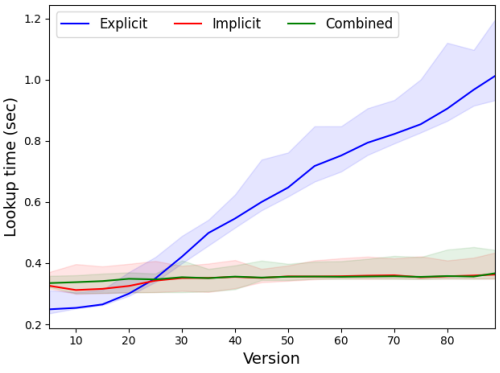
(G) 50-LC-HW-FS-CP-M5-AG-DM



(H) 50-LC-HW-FS-CP-M5-SO-DM



(I) 50-LC-HW-FS-CL-M5-AG-DM



(J) 50-LC-HW-FS-CL-M5-SO-DM

FIGURE 8.6: Delta materialisation (DM) query look up time in seconds (sec) over 89 versions for different input settings.

Chapter 9

Discussion

In this chapter we discuss the key findings, implications, and limitations of Bi-VAKs. We have divided this chapter into two parts. In Section 9.1 we discuss the experiments from Chapter 8 and their limitations. In addition, we interpret the evaluation results in more detail. In Section 9.2 we give an in-depth discussion of some limitations of Bi-VAKs.

9.1 Experimental results

In Chapter 8 we have presented a thorough evaluation of Bi-VAKs. In this evaluation, we focus on the ingestion performance and the version materialisation (VM), delta materialisation (DM), and version (VQ) basic triple pattern query performance. We acknowledge that this evaluation does not cover all aspects of Bi-VAKs. For example, we omit a performance analysis of modifications to a branch and tag, but such evaluation would be rather similar to the evaluation of update modifications. They may even have less influence on the performance than update modification, because the updates are queried more often than branches or tags. Nevertheless, our experiments have many other limitations. Therefore, we give a comprehensive overview of these limitations in section 9.1.3. But, we first elaborate on the key findings from the conducted experiments in Section 9.1.1 and 9.1.2.

9.1.1 Ingestion Performance

For the ingestion performance we look both at the storage size as well at the ingestion time of creating a revision-store with different input settings. We showed that revision-stores with an implicit and combined reference strategy and with a repeated content update strategy have the largest storage size, because they contain the highest number of triples. The implicit and the combined reference strategy contains the same amount of triples except for the revision-store including branches. The implicit reference strategy adds a revision number and branch index to a valid, and transaction revision, and the the combined reference strategy adds a revision number, branch index, and the valid revision to a transaction revision. The explicit reference strategy only adds the valid revision(s) to the transaction revision, which results in a smaller storage size. Furthermore, a revision-store with a repeated update content strategy consists of more triples than a related content strategy, because, as the name already implied, it repeated the modifications in the newly created update. Although, fortunately there is not a great difference in storage size between the various revision-stores, they still contain much more triples than the total number of modifications (93,131). The extra metadata, as the authors, creation dates, SHA-256 value, descriptions, and the preceding revision also affect their storage sizes.

In addition to the storage size, we also study the ingestion time. If we look at the ingestion time for a revision-store with an input setting of 50-LC-HW-CL-S45-B3-M5, we conclude that the ingestion time increases tremendously when we would check for every modification whether we can delete or insert it to the revision-store. The time even increases when the revision-store consists of more updates. It grows almost exponentially. It takes around 10 seconds to ingest a single update. It will not be feasible in practice if Linked Data users regularly make changes to the dataset. Moreover, in practise a single update might not have 50 modifications but perhaps as many as 1000. If we did not check whether a modification is invertible, Bi-VAKs has a much more realistic ingestion time of around 80 milliseconds. Its ingestion time also has a linear growth and is independent of the number of updates in the revision-store. In practise $\approx$ 80 milliseconds might be still high if numerous users make many adaptations in a short time. Computing sequentially a SHA-256 value for each transaction and valid revision may take too long for these systems. We may get concurrency errors, such as some updates might refer to the same update, while they should end up in sequence.

9.1.2 Query Performance

For the query performance we primarily examine the time to obtain a version materialisation (VM), delta materialisation (DM) or version (VQ) query result. We looked at the influence of various input settings, such as different fetching strategies, branches, a snapshot, and modified updates. We conclude that the overall performance of the implicit and of the combined reference strategy are roughly the same. Their median query duration lines in the various graphs overlap at most points, and their query time is low for most input settings. The performance of an explicit reference strategy, however, can considerably vary from graph to graph, and can sometimes be very bad. Sometimes, we even could not obtain a result when there are too many updates in the revision-store. The explicit reference strategy uses SPARQL property paths to obtain the required transaction revisions instead of revision numbers. These property paths from one transaction revision to all its preceding transaction revisions might profoundly affect the query performance. In addition, all revisions must be retrieved in order to sort the updates, whereas for these other strategies the updates are sorted according to their revision number. Besides, Fuseki got an stack overflow error after trying to query too many updates. The query paths have seemed to be too large. Below we further discuss the results for each query atom separately.

Version Materialisation (VM) From the query results presented in section 8.4.1 we conclude the following points. (i) Sorting and aggregating nearly give the same results for the implicit and combined reference strategy, while sorting certainly influence the performance of the explicit reference strategy. It means that the aggregation of updates does not directly lead to a better result, so regarding performance it not necessary if updates are invertible. (ii) Adding a snapshot considerably worsens the query performance for all three reference strategies. The retrieval of the updates and the process of fast-forwarding and rewinding these updates between two states take much longer than the retrieval and the process of fast-forwarding all updates from the initial revision. (iii) Adding a lot of branches affects the query duration for the implicit and combined reference strategy. The explicit reference strategy has the same procedure with as well as without branches. For the implicit and combined reference strategy we must first determine the branch indices and the revision numbers from which these branches branch off, and then based on these numbers we can retrieve the updates. (iv) Having modified updates slightly influences the performance, particularly, for the explicit reference strategy. When retrieving updates

we always want the updates with the latest transaction time. Therefore, we must omit the modified updates, which might affects the query performance. (v) The related update content strategy slightly improve the query performance. Thus, the related update content strategy has both a better ingestion performance and a query performance than the repeated update content strategy. (vi) More modifications per update improves the query performance, because we need to consider and retrieve less updates. (vii) Little and much overlap rather affects the performance. It mainly does for the explicit and combined reference strategy. Querying the updates with much overlap takes a little bit more time, because we must retrieve more of them at once. (viii) In conclusion, the explicit reference strategy has the best aggregated VM query performance, but its query time increases when we evaluate a later version. It means that property paths are performing well for a small number of versions, but worse or not all for a large number of versions. The implicit reference strategy performs better than the combined reference strategy. For the implicit reference strategy we can directly query the *Updates*, while for the combined reference strategy we first query the *Updates Revision* and then the *Updates*.

Delta Materialisation (DM) From the query results presented in section 8.4.2 we conclude the following points. (i) Although the aggregated DM performances are good for the implicit and combined reference strategy, their VM performances for a single snapshot are really bad. This is contradictory, because we retrieve the updates between two states in the same way. However, for the snapshot we compare the snapshot state with the request version, while for DM queries we compare the first version with the fifth, tenth, etc. version. Therefore, there are no or hardly any updates to compare with whether they are modifications or reversions of each other. The retrieval of modified updates by previous or following updates takes a lot of time. (ii) Constructing each version separately by sorting them first and then computing their difference has an equal or even lower query time than querying the updates between two states directly. Again, regarding performance it not necessary if updates are invertible. Both for retrieving updates between two states as well as for retrieval all updates from the initial revision it is needed to go over all updates in the revision-store. This process is probably more the bottleneck than the number of updates. Moreover, for retrieving updates between two states we must rewind some updates, and we must fast-forward others. Therefore, in our implementation we query the endpoint of revision-store multiple times, which might be less efficient than a single time. (iii) The implicit and the combined reference strategy have around the same constant DM query performance, whereas the performance of the explicit reference strategy is worse and declines. Te implicit and combined strategies are not affected by the number of versions, properly, because for each version we check the same amount of updates whether they meet the valid and transaction time conditions. The explicit strategy obtains the updates via SPARQL property paths, which takes more time when this path is longer. (iv) Similar to the VM query performance, a larger number of updates, modified updates, and branches in the revision-store also lead to a higher query look up time for DM queries.

Version Query (VQ) From the query results presented in section 8.4.3 we conclude the following points. (i) The VQ performance is slow in general. We go from one tag to another tag, and we construct every state separately. Although we build one state from its previous state, this process is still time consuming. It has to compare many updates whether they satisfy the time constrains. However, it is complex to query all updates at once and to subsequently construct every version. A version depends on two time lines simultaneously. Two consecutive tags might have different valid times, and therefore, we do not have a single valid time to query the corresponding updates. (ii) The explicit reference strategy has a much higher VQ query time than the implicit and combined reference strategy. Which are almost the same. For the explicit reference strategy constructing a

state from another state is much more time consuming than for the implicit and the combined reference strategies. Which we can also conclude from the DM results. The SPARQL queries to retrieve the explicit referenced updates might be less efficient. (iii) For all three reference strategies it is faster to first sort the updates and then construct every version separately than to aggregate the updates and construct a version from another version. Similar to the VM and DM performance, it means that we can better sort the updates than aggregate them, and that thus the "invertibility check" is not needed. (iv) In conclusion, again the number of updates, modified updates, and branches in the revision-store lead to lower query performance.

In summary, the query performance (immensely) declines when we query all updates instead of only the updates that match the users' SPARQL queries; and when we want to determine the updates between two states, such as for the VM queries with an intermediate materialised version of a state (snapshot) or for the DM and VQ queries. For the implicit and the combined reference strategy the query performance is around the same or even lower when we first obtain all updates from the initial revision, and sort them instead of aggregating updates directly. It means that the "invertibility check" of the modifications in the update is not needed in order to obtain a better query performance. In addition, the number of updates, modified updates, and branches in the revision-store slightly affect the query performance of all three strategies. Finally, over all query types and various input parameters, the implicit reference strategy is the best performing strategy, and is quickly followed by the combined reference strategy. The explicit reference strategy performs well for VM queries, but worse for the other query atoms.

9.1.3 Limitations of Experiments

In this section we discuss some limitations of our experiments in more detail. One of the limitations of our experiments is that we only have evaluated Bi-VAKs on triples. We decided to not include named graphs to the BEAR-B-Daily dataset to limit the number of experiments. These named graphs could have been added to the dataset by giving some triples randomly a fourth element (an IRI) that pretends to be a name of a named graph. A second limitation is that we did not evaluate our versioning system, Bi-VAKs, on the other BEAR Benchmark datasets. We only used BEAR-B-Daily. This dataset is the smallest dataset. Therefore, it is suitable for the time and the computer memory we had, but it is absolutely not representative to fully assess a (large-scale) version control system. It is quite possible that Bi-VAKs works well for BEAR-B-Daily, but not for BEAR-A. Namely, whereas all change-based versions in BEAR-B have together a storage size of 44MB, these versions in BEAR-A have a storage size of 138GB.

Another limitation is that we did not compare our work with existing RDF versioning systems. It would have been possible to put all changes in a single *Update* and to give them all the same valid time or no valid time at all. Instead of evaluation a bi-temporal versioning system, we would have evaluated a uni-temporal system. Therefore, we could have compared our results with the results of other systems. However, our system is a bi-temporal versioning approach and not a uni-temporal system. If we had considered these uni-temporal updates in the revision-store, we would have lost an import aspect of our system in the experiments. Another comparison we have not made is the annotation of triples by using named graphs instead of RDF-star. The literature on RDF-star suggested that RDF-star is better annotation approach than named graphs, but it would have been valuable to really assess this statement.

Where our performance analysis also falls short is that we do not take into account the number of retrieved updates per query. In our experiments we computed the median over all query time for each update, and we did not look at whether some queries needed more updates to determine a query result than others. It is also likely that it leads to performance differences. The last limitation we discuss is our usage of Jena Fuseki in our experiments. Jena Fuseki might perform poorly compared to other triple/quad-stores, such as MarkLogic[1], or Stardog[2]. However, some of these stores might not be open source and therefore, too costly for a master student. In addition, some other open source triple/quad-stores do not support RDF-star, such as Virtuoso[3].

9.2 Limitations of Bi-VAKs

In addition to the limitation of our experiments, in this section we discuss some limitations of our prototypical Linked Data Version Control System, Bi-VAKs. One of main limitations of Bi-VAKs is that needs a lot of memory to store the revisions in a single revision-store and plenty of time to retrieve the revisions from this revision-store. To describe these revision we rely on the pure RDF data model. It would be interesting to compress these revisions in order to reduce the revision-store its storage size, and to develop an indexing technique in order to query the revisions efficiently. However, Bi-VAKs would require a compression approach that supports both RDF-star and bi-temporal time information. A second performance limitation of Bi-VAKs is that its algorithms are not optimised for constructing a single version or the difference between versions as fast as possible. They are mainly based on SPARQL queries. Thus, their performance primarily depends on how well we have set up these SPARQL queries, and on how fast the triple/quad-store can process these queries, and can return a result back to our system. However, it was not our primary goal to fully optimise this construction process, but just to show that our system can retrieve versions. Finally, a third performance limitation is the use of a materialised version of a state of the RDF dataset (snapshot). Since this snapshot has both a valid and a transaction time, it is inefficient to query the updates between two states. However, existing research already has proven the performance increase for snapshots. Therefore, a possible improvement may be to make Bi-VAKs its snapshots valid time independent by annotating each triple with its valid time.

In addition to performance limitations, Bi-VAKs also has some RDF application limitations. First of all, Bi-VAKs only support INSERT DATA and DELETE DATA update operations. It does not provide for the other graph update and management operations, such as the creation and deletion of entire graphs, and it cannot handle data dumps of a complete in RDF presented data file. Moreover, the insertion or deletion of an empty graph requires a new representation in an *Update*, because we must insert or delete a graph name instead of a triple. Secondly, Bi-VAKs does not support complex SPARQL queries. It only supports basic triple pattern queries. It means that these complex SPARQL queries can only be computed from the results of each basic triple pattern in the complex query separately. Nonetheless, a solution might be to retrieve all updates matching one of the basic triple patterns stated in the complex SPARQL query and merge them. In addition to complex queries, Bi-VAKs also does not support blank nodes, although blank nodes are an import concept within RDF, and certainly some RDF dataset will contain blank nodes. However, because of their local scope one cannot assume that blank nodes with the same identifier but from different graphs are the same. In a situation where an unique name is

[1] https://www.marklogic.com
[2] https://www.stardog.com
[3] https://virtuoso.openlinksw.com

required, like ours, we could replace all blank nodes in an RDF graph with IRIs, also called Skolemisation.

The last group of limitations are some implementation limitations of Bi-VAKs. In Bi-VAKs it is not possible yet to do a revert or merge request. A merge request also means that we must develop a merge strategy that solves how two contradicting updates can be merged. Furthermore, it is also not possible to exchange updates and thus to synchronise a versioned RDF dataset with each other, which also calls for a merge strategy. Another limitation is that thus far Bi-VAKs does not have an endpoint to request for provenance data such that users can obtain all updates created by themselves, or all collaborators that work on a diverged state of the dataset. In conclusion, Bi-VAKs is, nonetheless, a first step to a bi-temporal versioning approach. It still lacks support for many other versioning concepts, such as authentication, concurrency issues, or a remote repository.

Chapter 10

Conclusion and Future Work

This thesis presents a Bi-temporal Versioning Approach for Knowledge graphs (Bi-VAKs): a bi-temporal collaborative change-based Linked Data Version Control System. Bi-VAKs is a prototypical approach that records both the transaction time and the valid time of a collection of modified quads in a collaborative setting. Thus, Bi-VAKs can provide for coupled historical and retrospective SPARQL queries. To conclude this work, we explicitly answer the research questions as presented in the introduction. In addition to the fact that Bi-VAKs concentrates on bi-temporal modifications to a RDF dataset, and bi-temporal SPARQL (Update) queries, it also focuses on collaboration between its Linked Data users by keeping track of provenance data, supporting diverged states, and providing a standard data access interface. Therefore, Bi-VAKs divides a change, also called a revision, into a transaction revision and (multiple) valid revision(s), which respectively denote the valid time and the transaction time. This leads us to answer the first research question.

RQ1: How can we design a Linked Data change-based collaborative version control system that can manage both the valid time and transaction time of a collection of changes made to a RDF dataset. In Chapter 5 we have introduced a conceptual design that depicts the structure of the revisions in the revision-store. Each revision is divided into (multiple) valid revision(s) and a transaction revision. More specifically, in Section 5.1 we describe that we make this distinction, primarily, in order to denote the transaction time and the valid time. We also discuss in this section the three reference strategies that connects a transaction revision with its corresponding valid revision(s): the explicit, the implicit, and the combined reference strategy. These strategies let a transaction revision either refer explicitly to its corresponding valid revision(s), or implicitly by the same revision number and branch index. In Section 5.2 we give an general explanation about these valid and transaction revisions, but we mainly address the different revision types. Namely, Bi-VAKs also supports other versioning operations besides updating, such as branching, tagging, and reverting. These other versioning operations are generally aimed to improve the collaboration between users, and thus these revisions have a different type and representation.

Although the conceptual design of the revision-store represents how the revisions are structured, and how they are related to one and other, it does not explain how these revisions can be stored in an actual RDF database. Therefore, to describe and to retrieve an update in a RDF syntax Bi-VAKs uses RDF-star and its corresponding query language SPARQL-star. Due to its comprehensibility it is a good candidate for describing triples within a RDF structured update, by which we answer the second research question.

RQ2: How can we represent updates by using RDF-star, and enable efficient bi-temporal version materialisation, delta materialisation, and version basic triple pattern queries by using SPARQL-star? Chapter 6 discusses the interaction between the

Linked Data users and the revision-store to enable bi-temporal SPARQL (Update) queries. More specifically, in Section 6.1 we explain how Bi-VAKs extracts the modified triples or quads from a SPARQL Update query, and how it represents these modifications in an update. In Section 6.2 we explain how these updates can be retrieved, and how a version or state of the RDF dataset can be constructed. We construct a version by querying the updates starting from the initial revision or from a materialised version of the dataset (snapshot). We could query all updates or only the updates that match the basic triple pattern in the SPARQL query. And, we could aggregate the updates, or sort them to construct the version.

In order to evaluate these representation, and version construction decisions, we must assess Bi-VAKs' ingestion and version materialisation (VM), delta materialisation (DM), and version (VQ) basic triple pattern query performance. However, existing benchmark only assess uni-temporal versioning systems. Therefore, we first expand the BEAR benchmark to a bi-temporal benchmark (Bi-BEAR), and we subsequently use it to evaluate Bi-VAKs on the different design decisions. This leads us to answer the third and final research question.

RQ3: How can we expand the uni-temporal BEAR Benchmark into a bi-temporal benchmark, and what is the ingestion and query performance of our prototypical change-based version control system? In Chapter 8 we first explain how we expand the BEAR Benchmark into a bi-temporal benchmark (Bi-BEAR). We split up a BEAR version into multiple updates with a randomly assigned valid time, and we denote a BEAR version by a tag. Then, in Section 8.2 we describe on which various input parameters we evaluate Bi-VAKs to assess our design decisions. Finally, in the last two sections we give the results of the ingestion, and VM, DM, and VQ basic triple pattern query performance of our implementation of Bi-VAKs, as described in Chapter 7. In Section 8.3 we show the storage size and the ingestion time to create each revision-store in Bi-VAKs. We observe that the ingestion time immensely increases when we check for each update whether we can invert it, and that all three reference strategies have about the same storage size. From Section 8.4 we notice that the usage of a snapshot, and retrieval of all updates worsen the query performance. The query time considerably decreases when only the matching updates are queried. And modified updates, branches, and more updates in the revision-store slightly affects the query performance. In addition, for the implicit and combined reference strategy the query time is rather the same, and sometimes even better if we sort the updates instead of aggregating them directly.

10.1 Future Work

In the discussion, we already suggested some future research directions for Bi-VAKs. In this section we elaborate on several of those directions.

Optimise Version Retrieval Although we have already made some suggestions to improve the version construction, this procedure could be further optimised. Instead of using a SPARQL query to retrieve the requested updates we could also let an optimised algorithm determine whether an update encloses the requested transaction and valid time. In addition, we could also use a better performing triple/quad-store to store and query the revision-store instead of using the in-memory store of Apache Jena-Fuseki. Or we could optimise the SPARQL queries itself to receive a faster response from the triple-store. Moreover, we could try to make a snapshot valid time independent, but we still require to fast-forward or rewind all modified updates in order to avoid repetition.

Compression of Revisions Currently, Bi-VAKs might not be very storage and query efficient. While it offers many opportunities for collaboration, other RDF versioning approaches may have a better storage and query performance. Namely, Bi-VAKs does not use efficient compression and indexing methods to store and retrieve the updates (Fernández, Polleres, and Umbrich, 2015). It stores the revision-store in a simple triple/quad-store, which is not optimised for storing and querying revisions efficiently. Although the compression of revisions could result in a lower storage space, it requires a complex indexing technique to query these revision based on their valid time, transaction time, and basic triple patterns. Furthermore, compressing may make the revisions less easily exchangeable, which will worsen cooperation.

Expand on Versioning Concepts Our present work already keeps track of provenance data; provides for diverging states; and has a standard data access interface to support collaboration between Linked Data users. However, it still lacks the support for many other versioning concepts that also enhance cooperation. In Bi-VAKs users cannot conflate their diverged states. They cannot exchange the revisions between systems in order to synchronise their states with the state of others. They cannot reverse their faulty revisions, and they cannot query the revisions based on their metadata. Furthermore, it does not support authentication of the users. Its implementation still concentrates on a single user. And, it does not provide for a remote repository where users can publish their RDF dataset, and from where other collaborators can copy (clone) the whole or parts of the dataset. In conclusion, Bi-VAKs is still a very simple prototypical versioning approach that can be expanded on many different versioning concepts.

Support for more complex RDF Concepts Our work currently only provide for INSERT DATA and DELETE DATA update operations and basic triple pattern SELECT SPARQL queries. In addition, Bi-VAKs does not support blank nodes. For a versioning approach to work well on existing Linked Data systems it must also support these more complex concepts. For example, we could replace all inserted blank nodes with skolem-IRIs. We could combine all the results of each triple pattern in order to obtain a result for the complex query. In addition, we could come up with an approach to identify the modified triples/quads in the INSERT/DELETE update operation, and we could propose a representation to insert and delete empty graphs in the update.

Bibliography

Ahn, Ilsoo and Richard T. Snodgrass (1986). "Performance Evaluation of a Temporal Database Management System". In: *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*. Ed. by Carlo Zaniolo. ACM Press, pp. 96–107.

Anderson, James and Arto Bendiken (2016). "Transaction-Time Queries in Dydra". In: *Joint Proceedings of the 2nd Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW 2016) and the 3rd Workshop on Linked Data Quality (LDQ 2016) co-located with 13th European Semantic Web Conference (ESWC 2016), Heraklion, Crete, Greece, May 30th, 2016*. Ed. by Jeremy Debattista et al. Vol. 1585. CEUR Workshop Proceedings. CEUR-WS.org, pp. 11–19.

Arndt, Natanael (2020). "Distributed Collaboration on Versioned Decentralized RDF Knowledge Bases". PhD thesis. HTWK Leipzig, Germany.

Arndt, Natanael et al. (2019). "Decentralized Collaborative Knowledge Management Using Git". In: *J. Web Semant.* 54, pp. 29–47.

Auer, Sören and Heinrich Herre (2006). "A Versioning and Evolution Framework for RDF Knowledge Bases". In: *Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27-30, 2006. Revised Papers*. Ed. by Irina B. Virbitskaite and Andrei Voronkov. Vol. 4378. Lecture Notes in Computer Science. Springer, pp. 55–69.

Berners-Lee, Tim (July 2006). *Linked Data*. Tech. rep. W3C. URL: https://www.w3.org/DesignIssues/LinkedData.html.

Berners-Lee, Tim and Dan Connolly (2004). "Delta: an ontology for the distribution of differences between RDF graphs". In: *World Wide Web, http://www. w3. org/DesignIssues/Diff* 4.3, pp. 4–3.

Bertails, Alexandre, Pierre-Antoine Champin, and Andrei Sambra (July 2015). *Linked Data Patch Format*. W3C Working Group Note. W3C. URL: https://www.w3.org/TR/ldpatch/.

Bhardwaj, Anant P. et al. (2015). "DataHub: Collaborative Data Science & Dataset Version Management at Scale". In: *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org.

Bhattacherjee, Souvik et al. (2015). "Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff". In: *Proc. VLDB Endow.* 8.12, pp. 1346–1357.

Brickley, Dan and R.V. Guha (Feb. 2014). *RDF Schema 1.1*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/rdf-schema/.

Brisaboa, Nieves R. et al. (2015). "A Compact RDF Store Using Suffix Arrays". In: *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*. Ed. by Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz. Vol. 9309. Lecture Notes in Computer Science. Springer, pp. 103–115.

Carothers, Gavin (Feb. 2014). *RDF 1.1 N-Quads*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/rdf-sparql-query/.

Carothers, Gavin and Andy Seaborne (Feb. 2014). *RDF 1.1 N-Triples*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/n-triples/.

Carroll, Jeremy J. et al. (2005). "Named graphs, provenance and trust". In: *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*. Ed. by Allan Ellis and Tatsuya Hagino. ACM, pp. 613–622.

Cassidy, Steve and James Ballantine (2007). "Version Control for RDF Triple Stores". In: *ICSOFT 2007, Proceedings of the Second International Conference on Software and Data Technologies, Volume ISDM/EHST/DC, Barcelona, Spain, July 22-25, 2007*. Ed. by Joaquim Filipe, Boris Shishkov, and Markus Helfert. INSTICC Press, pp. 5–12.

Cerdeira-Pena, Ana et al. (2016). "Self-Indexing RDF Archives". In: *2016 Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30 - April 1, 2016*. Ed. by Ali Bilgin et al. IEEE, pp. 526–535.

Cyganiak, Richard, David Wood, and Markus Lanthaler (Feb. 2014). *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/rdf11-concepts/.

Fernández, Javier D., Axel Polleres, and Jürgen Umbrich (2015). "Towards Efficient Archiving of Dynamic Linked Open Data". In: *Proceedings of the First DIACHRON Workshop on Managing the Evolution and Preservation of the Data Web co-located with 12th European Semantic Web Conference (ESWC 2015), Portorož, Slovenia, May 31, 2015*. Ed. by Jeremy Debattista, Mathieu d'Aquin, and Christoph Lange. Vol. 1377. CEUR Workshop Proceedings. CEUR-WS.org, pp. 34–49.

Fernández, Javier D. et al. (2019). "Evaluating query and storage strategies for RDF archives". In: *Semantic Web* 10.2, pp. 247–291.

Frommhold, Marvin et al. (2016). "Towards Versioning of Arbitrary RDF Data". In: *Proceedings of the 12th International Conference on Semantic Systems, SEMANTICS 2016, Leipzig, Germany, September 12-15, 2016*. Ed. by Anna Fensel et al. ACM, pp. 33–40.

Gao, Shi, Jiaqi Gu, and Carlo Zaniolo (2016). "RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases". In: *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*. Ed. by Evaggelia Pitoura et al. OpenProceedings.org, pp. 269–280.

Gearon, Paula, Alexandre Passant, and Axel Polleres (Mar. 2013). *SPARQL 1.1 Update*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/sparql11-update/.

Grandi, Fabio (2011). "Light-weight ontology versioning with multi-temporal RDF schema". In: *SEMAPRO 2011, The Fifth International Conference on Advances in Semantic Processing*, pp. 42–48.

Graube, Markus, Stephan Hensel, and Leon Urbas (2014). "R43ples: Revisions for Triples - An Approach for Version Control in the Semantic Web". In: *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems, LDQ@SEMANTiCS 2014, Leipzig, Germany, September 2nd, 2014*. Ed. by Magnus Knuth, Dimitris Kontokostas, and Harald Sack. Vol. 1215. CEUR Workshop Proceedings. CEUR-WS.org.

Guo, Yuanbo, Zhengxiang Pan, and Jeff Heflin (2005). "LUBM: A benchmark for OWL knowledge base systems". In: *J. Web Semant.* 3.2-3, pp. 158–182.

Gutiérrez, Claudio, Carlos A. Hurtado, and Alejandro A. Vaisman (2007). "Introducing Time into RDF". In: *IEEE Trans. Knowl. Data Eng.* 19.2, pp. 207–218.

Haase, Peter and Ljiljana Stojanovic (2005). "Consistent Evolution of OWL Ontologies". In: *The Semantic Web: Research and Applications, Second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece, May 29 - June 1, 2005, Proceedings*. Ed. by Asunción Gómez-Pérez and Jérôme Euzenat. Vol. 3532. Lecture Notes in Computer Science. Springer, pp. 182–197.

Hartig, Olaf (2017). "RDF* and SPARQL*: An Alternative Approach to Annotate Statements in RDF". In: *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 23rd - to - 25th, 2017*. Ed. by Nadeschda Nikitina et al. Vol. 1963. CEUR Workshop Proceedings. CEUR-WS.org.

Hartig, Olaf and Bryan Thompson (2014). "Foundations of an Alternative Approach to Reification in RDF". In: *CoRR* abs/1406.3399.

Hartig, Olaf et al. (July 2021). *RDF-star and SPARQL-star*. Draft Community Group Report. W3C. URL: https://w3c.github.io/rdf-star/cg-spec/2021-07-01.html.

Hartung, Michael, Anika Groß, and Erhard Rahm (2013). "COnto-Diff: generation of complex evolution mappings for life science ontologies". In: *J. Biomed. Informatics* 46.1, pp. 15–32.

Hauptmann, Claudius, Michele Brocco, and Wolfgang Wörndl (2015). "Scalable Semantic Version Control for Linked Data Management". In: *Proceedings of the 2nd Workshop on Linked Data Quality co-located with 12th Extended Semantic Web Conference (ESWC 2015), Portorož, Slovenia, June 1, 2015*. Ed. by Anisa Rula et al. Vol. 1376. CEUR Workshop Proceedings. CEUR-WS.org.

Hayes, Patrick J. and Peter F. Patel-Schneider (Feb. 2014). *RDF 1.1 Semantics*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/rdf11-mt/.

Huang, Silu et al. (2017). "OrpheusDB: Bolt-on Versioning for Relational Databases". In: *Proc. VLDB Endow.* 10.10, pp. 1130–1141.

Im, Dong-Hyuk, Sang-Won Lee, and Hyoung-Joo Kim (2012). "A Version Management Framework for RDF Triple Stores". In: *Int. J. Softw. Eng. Knowl. Eng.* 22.1, pp. 85–106.

Jensen, Christian S and Richard T Snodgrass (1999). "Temporal data management". In: *IEEE Transactions on knowledge and data engineering* 11.1, pp. 36–44.

Käfer, Tobias et al. (2013). "Exploring the Dynamics of Linked Data". In: *The Semantic Web: ESWC 2013 Satellite Events - ESWC 2013 Satellite Events, Montpellier, France, May 26-30, 2013, Revised Selected Papers*. Ed. by Philipp Cimiano et al. Vol. 7955. Lecture Notes in Computer Science. Springer, pp. 302–303.

Khurana, Udayan and Amol Deshpande (2013). "Efficient snapshot retrieval over historical graph data". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. Ed. by Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou. IEEE Computer Society, pp. 997–1008.

Klein, Michel C. A. et al. (2002). "Ontology Versioning and Change Detection on the Web". In: *Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, 13th International Conference, EKAW 2002, Siguenza, Spain, October 1-4, 2002, Proceedings*. Ed. by Asunción Gómez-Pérez and V. Richard Benjamins. Vol. 2473. Lecture Notes in Computer Science. Springer, pp. 197–212.

Knuth, Magnus, Johannes Hercher, and Harald Sack (2012). "Collaboratively Patching Linked Data". In: *CoRR* abs/1204.2715.

Kotsev, Venelin et al. (2016). "Benchmarking RDF Query Engines: The LDBC Semantic Publishing Benchmark". In: *Proceedings of the Workshop on Benchmarking Linked Data (BLINK 2016) co-located with the 15th International Semantic Web Conference (ISWC), Kobe, Japan, October 18, 2016*. Ed. by Irini Fundulaki et al. Vol. 1700. CEUR Workshop Proceedings. CEUR-WS.org.

Kovacevic, Filip et al. (2022). "StarVers - Versioning and Timestamping RDF data by means of RDF* - An Approach based on Annotated Triples". In:

Lebo, Timothy et al. (Apr. 2013). *PROV-O: The PROV Ontology*. W3C Working Draft. W3C. URL: https://www.w3.org/TR/prov-o/.

Longley, Dave (Apr. 2021). *RDF Dataset Canonicalization, A Standard RDF Dataset Canonicalization Algorithm*. Draft Community Group Report. W3C. URL: https://www.w3.org/TR/prov-o/.

Maddox, Michael et al. (2016). "Decibel: The Relational Dataset Branching System". In: *Proc. VLDB Endow.* 9.9, pp. 624–635.

McGuinness, Deborah L, Frank Van Harmelen, et al. (2004). "OWL web ontology language overview". In: *W3C recommendation* 10.10, p. 2004.

Meimaris, Marios and George Papastefanatos (2016). "The EvoGen Benchmark Suite for Evolving RDF Data". In: *Joint Proceedings of the 2nd Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW 2016) and the 3rd Workshop on Linked Data Quality (LDQ 2016) co-located with 13th European Semantic Web Conference (ESWC 2016), Heraklion, Crete, Greece, May 30th, 2016*. Ed. by Jeremy Debattista et al. Vol. 1585. CEUR Workshop Proceedings. CEUR-WS.org, pp. 20–35.

Meinhardt, Paul, Magnus Knuth, and Harald Sack (2015). "TailR: a platform for preserving history on the web of data". In: *Proceedings of the 11th International Conference on Semantic Systems, SEMANTICS 2015, Vienna, Austria, September 15-17, 2015*. Ed. by Axel Polleres et al. ACM, pp. 57–64.

Miles, Alistair and Sean Bechhofer (Aug. 2009). *SKOS Simple Knowledge Organization System*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/2009/REC-skos-reference-20090818/.

Morsey, Mohamed et al. (2012). "DBpedia and the live extraction of structured data from Wikipedia". In: *Program* 46.2, pp. 157–181.

Neumann, Thomas and Gerhard Weikum (2010). "x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases". In: *Proc. VLDB Endow.* 3.1, pp. 256–263.

Nguyen, Vinh, Olivier Bodenreider, and Amit P. Sheth (2014). "Don't like RDF reification?: making statements about statements using singleton property". In: *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*. Ed. by Chin-Wan Chung et al. ACM, pp. 759–770.

Noy, Natalya Fridman and Mark A. Musen (2002). "PROMPTDIFF: A Fixed-Point Algorithm for Comparing Ontology Versions". In: *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*. Ed. by Rina Dechter, Michael J. Kearns, and Richard S. Sutton. AAAI Press / The MIT Press, pp. 744–750.

Opijnen, Marc van (2018). "Legal (Ly) Linked Data. Over De Noodzaak Tot Betere Standaardisatie Van Juridisch Bronmateriaal (Legal (Ly) Linked Data About the Need for Better Standardisation of Legal Sources)". In: *Over De Noodzaak Tot Betere Standaardisatie Van Juridisch Bronmateriaal (Legal (Ly) Linked Data About the Need for Better Standardisation of Legal Sources)(January 15, 2018). Computerrecht* 51.

Otterdijk, Matthijs van, Gavin Mendel-Gleason, and Kevin Feeney (2020). "Succinct Data Structures and Delta Encoding for Modern Databases". In:

Papakonstantinou, Vassilis et al. (2017). "SPBv: Benchmarking Linked Data Archiving Systems". In: *Joint Proceedings of BLINK2017: 2nd International Workshop on Benchmarking Linked Data and NLIWoD3: Natural Language Interfaces for the Web of Data co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 21st - to - 22nd, 2017*. Ed. by Ricardo Usbeck et al. Vol. 1932. CEUR Workshop Proceedings. CEUR-WS.org.

Papavassiliou, Vicky et al. (2009). "On Detecting High-Level Changes in RDF/S KBs". In: *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*. Ed. by Abraham Bernstein et al. Vol. 5823. Lecture Notes in Computer Science. Springer, pp. 473–488.

Pelgrin, Olivier, Luis Galárraga, and Katja Hose (2021). "Towards fully-fledged archiving for RDF datasets". In: *Semantic Web* 12.6, pp. 903–925.

Plessers, Peter and Olga De Troyer (2005). "Ontology Change Detection Using a Version Log". In: *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005, Proceedings*. Ed. by Yolanda Gil et al. Vol. 3729. Lecture Notes in Computer Science. Springer, pp. 578–592.

Prud'hommeaux, Eric and Andy Seaborne (Jan. 2008). *SPARQL Query Language for RDF*. W3C Recommendation. W3C. URL: https://www.w3.org/TR/rdf-sparql-query/.

Roussakis, Yannis et al. (2015). "A Flexible Framework for Understanding the Dynamics of Evolving RDF Datasets". In: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*. Ed. by Marcelo Arenas et al. Vol. 9366. Lecture Notes in Computer Science. Springer, pp. 495–512.

Salzberg, Betty and Vassilis J. Tsotras (1999). "Comparison of Access Methods for Time-Evolving Data". In: *ACM Comput. Surv.* 31.2, pp. 158–221.

Sande, Miel Vander et al. (2013). "R&Wbase: git for triples". In: *Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013*. Ed. by Christian Bizer et al. Vol. 996. CEUR Workshop Proceedings. CEUR-WS.org.

Seaborne, Andy (Jan. 2010). *SPARQL 1.1 Property Paths*. W3C Working Draft. W3C. URL: https://www.w3.org/TR/sparql11-property-paths/.

Seering, Adam et al. (2012). "Efficient Versioning for Scientific Array Databases". In: *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*. Ed. by Anastasios Kementsietsidis and Marcos Antonio Vaz Salles. IEEE Computer Society, pp. 1013–1024.

Singh, Anuj, Rob Brennan, and Declan O'Sullivan (2018). "DELTA-LD: A change detection approach for linked datasets". In: *4th Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW)*.

Sompel, Herbert Van de et al. (2009). "Memento: Time Travel for the Web". In: *CoRR* abs/0911.1112.

Soroush, Emad and Magdalena Balazinska (2013). "Time travel in a scientific array database". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. Ed. by Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou. IEEE Computer Society, pp. 98–109.

Taelman, Ruben et al. (2019). "Triple storage for random-access versioned querying of RDF archives". In: *J. Web Semant.* 54, pp. 4–28.

Tansel, Abdullah Uz et al., eds. (1993). *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings. ISBN: 0-8053-2413-5.

Tichy, Walter F. (1982). "Design, Implementation, and Evaluation of a Revision Control System". In: *Proceedings, 6th International Conference on Software Engineering, Tokyo, Japan, September 13-16, 1982*. Ed. by Yutaka Ohno et al. IEEE Computer Society, pp. 58–67.

Umbrich, Jürgen, Sebastian Neumaier, and Axel Polleres (2015). "Quality Assessment and Evolution of Open Data Portals". In: *3rd International Conference on Future Internet of Things and Cloud, FiCloud 2015, Rome, Italy, August 24-26, 2015*. Ed. by Irfan Awan, Muhammad Younas, and Massimo Mecella. IEEE Computer Society, pp. 404–411.

Umbrich, Jürgen et al. (2010). "Towards Dataset Dynamics: Change Frequency of Linked Open Data Sources". In: *Proceedings of the WWW2010 Workshop on Linked Data on the Web, LDOW 2010, Raleigh, USA, April 27, 2010*. Ed. by Christian Bizer et al. Vol. 628. CEUR Workshop Proceedings. CEUR-WS.org.

Völkel, Max and Tudor Groza (2006). “SemVersion: An RDF-based ontology versioning system”. In: *Proceedings of the IADIS international conference WWW/Internet*. Vol. 2006. Citeseer, p. 44.

Watkins, E. Rowland and Denis A. Nicole (2006). “Named Graphs as a Mechanism for Reasoning About Provenance”. In: *Frontiers of WWW Research and Development - APWeb 2006, 8th Asia-Pacific Web Conference, Harbin, China, January 16-18, 2006, Proceedings*. Ed. by Xiaofang Zhou et al. Vol. 3841. Lecture Notes in Computer Science. Springer, pp. 943–948.

Yang, Dan and Li Yan (2018). “Transforming XML to RDF(S) with Temporal Information”. In: *J. Comput. Inf. Technol.* 26.2, pp. 115–129.

Zhang, Fu et al. (2019). “Temporal Data Representation and Querying Based on RDF”. In: *IEEE Access* 7, pp. 85000–85023.

Zhang, Fu et al. (2021). “RDF for temporal data management–a survey”. In: *Earth Science Informatics* 14.2, pp. 563–599.