UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

VERSIONED DATASETS AND QUERIES IN RDF/SPARQL

TESIS PARA OPTAR AL GRADO DE
MAGISTER EN CIENCIAS, MENCION COMPUTACION

MEMORIA PARA OPTAR AL TITULO DE
INGENIERO CIVIL EN COMPUTACION

IGNACIO IVAN CUEVAS ROJAS

PROFESOR GUÍA:
AIDAN HOGAN

MIEMBROS DE LA COMISIÓN:
BENJAMIN BUSTOS CARDENAS
BARBARA POBLETE LABRA
CARLOS BUIL ARANDA

Este trabajo ha sido parcialmente financiado por el Instituto Milenio Fundamentos de los Datos

SANTIAGO DE CHILE
2020

Resumen

Hoy en día, una gran cantidad de *datasets* están basados en RDF, dada su denominación como estándar de la Web. Wikidata [19] es uno los más importantes y masivos *datasets* RDF, conteniendo más de 84 millones de *items*[1], y tiene por función estructurar y estandarizar el contenido de datos de Wikipedia.

Wikidata – al igual que muchos otros *datasets* RDF publicados como *Linked Data* [12] – está en constante cambio, debido a que corresponde a un *dataset* abierto y colaborativo que puede ser editado por sus usuarios. Además, datos adicionales son importados desde fuente externas constantemente, lo cual convierte a Wikidata en un *dataset* muy dinámico en ciertos aspectos. Dadas estas circunstancias, se puede decir que Wikidata tiene distintas "versiones" en el tiempo, donde los datos disponibles son distintos dependiendo del periodo de tiempo donde se consulten. Este hecho hace posible analizar una componente temporal de Wikidata y, por lo tanto, sería relevante estudiar la posibilidad de desarrollar un sistema de versionamiento y una metodología para construir consultas usando sólamente SPARQL base junto a un modelo RDF que sea capaz de representar datos versionados, manteniéndose abierto a recibir consultar sin utilizar herramientas o *software* especializado.

En este trabajo proponemos y analizamos un sistema que permite, de manera eficiente, almacenar y realizar consultas sobre grafos RDF que mantienen un historial de cambios en el tiempo. Algunos sistemas que llevan a cabo objetivos similares ya existen. Sin embargo, dichos sistemas utilizan extensiones, índices especializados y herramientas fuera del estándar. Teniendo esto en mente, se añade la meta de construir un sistema que logre nuestro objetivo usando sólamente RDF y SPARQL base, con la intención de dar al problema una solución estándar y lista para usar en cualquier ambiente.

Se hacen pruebas con varias alternativas para representar el *dataset* versionado. Cada una requiere su propio método para convertir consultas SPARQL, permitiendo la compatibilidad con los *datasets* construidos. Se mantiene una versión sin compresión para comparar los resultados obtenidos. Junto a ésta, se construyen representaciones basadas de deltas e intervalos. Se definen dos tipos de deltas: *absolutos*, donde cada versión del grafo se compara con la versión base (puede ser la más antigua o la más reciente), y *secuenciales*, donde cada versión se compara con la versión anterior a sí misma, manteniendo así una representación "paso a paso" de los cambios a través del tiempo.

En general, la representación basada en intervalos tiene el mejor desempeño, excepto por el tiempo que toma su construcción. Sin embargo, dichos tiempos de construcción se pueden mejorar con pequeñas optimizaciones. Además, la representación de intervalos es la única que ofrece soporte parcial para *property paths*, mientras que el resto de las representaciones son completamente incompatibles con éstos dado que no posible aplicar un *property path* sobre múltiples *named graphs*.

[1] https://www.wikidata.org/wiki/Wikidata:Statistics

Abstract

Currently, a large number of existing datasets are based on RDF due to its status as a Web standard, with Wikidata being one of the most notable. Wikidata [19] is a massive RDF dataset, with over 84 million items[2], which is meant to serve as a way to structure and standarize the data content of Wikipedia.

Wikidata – like many other RDF datasets published as Linked Data [12] – is in constant change, due to it being an open dataset that can be edited by users. Furthermore, data is being imported from external sources constantly, which makes Wikidata very dynamic in some areas. Given these circumstances, it can be said that Wikidata has many "versions" over time, where different data is available at different time intervals, making it possible to analyze its time component. As such, it would be relevant to study the possibility of developing a versioning system and methodology for queries using only base SPARQL with an RDF model capable of representing versioned data and open to be queried without needing specialized tools or software.

In this work we propose and evaluate a system that allows for efficiently storing and querying versioned RDF graphs that track changes over time. A few systems that achieve similar goals currently exist. These systems, however, employ extensions, specialized indices and other custom tools in their implementations. Considering this, the goal of building a system that achieves our objetive using only base RDF and SPARQL is added, aiming to serve as an "off-the-shelf" solution to the problem presented.

Several alternatives to represent the versioned data are tested, each requiring a different method of conversion for SPARQL queries in order to ensure compatibility with the versioned datasets created. A version without compression is kept to compare the results obtained. Alongside it, versioned representations based on intervals are deltas are built. Two types of deltas are defined: *absolute* deltas that take each version of the graph and compare it to the base version (either oldest or most recent), and *sequential* deltas, where each version is compared against the previous one to keep a "step by step" representation of the changes in the graph through time.

Overall, the interval-based representation has the most competitive results, with the exception of build times. These times could be easily be improved upon with small optimizations to the construction method. Moreover, the interval-based representation is the only one that partially supports property paths, while the rest of the representations are fully incompatible with them as property paths cannot be evaluated across different named graphs.

[2] https://www.wikidata.org/wiki/Wikidata:Statistics

A mi familia, a mis amigos
y a mi Mikkita.

Muchas gracias

Agradecimientos

Quisiera en primer lugar darle las gracias a mi profesor guía Aidan Hogan por la infinita paciencia que tuvo conmigo durante este trabajo de tesis, además de siempre tener la disposición a contestar dudas y ayudarme con las partes más difíciles del trabajo.

A mis padres, Miriam e Iván, les agradezco también por asegurarse que nunca me faltara nada y por todo el cariño que me entregan día a día. Y a mi hermano Sebastián y mi sobrina Millaray les doy las gracias porque sé que a la distancia siempre me envian ánimos y buenos deseos.

A mis amigos les agradezco por la compañía, las risas y todo el cariño que me entregan con sólo su presencia. Son muchos como para nombrarlos a todos, pero sé que van a entender que me refiero a ellos y ellas.

Finalmente, a mi Mikkita le doy las gracias porque sin ella probablemente no hubiese completado esta tesis. Ella me dio el ánimo para seguir adelante en los momentos donde ya casi no me quedaba energía.

Muchas gracias a todos por ser parte de este viaje junto a mí.

Contents

Index of Tables

Index of Figures

Chapter 1

Introduction

1.1. Motivation

The Semantic Web[1] is an initiative of the World Wide Web Consortium, which envisions a "Web of Data"; that is, a web of *Linked Data* where information has a standarized structure, making it easier for machines to comunicate. In order to make such standarization possible, it is necessary to incorporate meta-data that describes the content, meaning and relationships of data across websites and online databases.

The Resource Description Framework [16] (RDF for short) is a standard model for data interchange on the Web. Its design is based on "triples", which represent connections in a graph, where every node is a "resource" with "predicates" as the edges connecting them. Both resources and predicates are formally defined, and a collection of rules allow for inferences about the relationships in the data. Alongside RDF, the ontological languages RDFS [3] and OWL [11] provide vocabulary for expressing basic relation and type definitions, and SPARQL [9] serves as RDF's query language (similar to SQL over relational databases).

Currently, a large number of existing datasets are based on RDF due to its status as a Web standard, with Wikidata being one of the most notable. Wikidata [19] is a massive RDF dataset, with over 84 million items[2], which is meant to serve as a way to structure and standarize the data content of Wikipedia. This structure and standarization is very useful for abstracting data and making it independent of a language. Currently, Wikipedia articles can have conflicting information for a topic depending on the language selected. Wikidata allows for collecting and curating data independently of a language, as well as reutilizing pieces of information associated to different entities. Another possible use for Wikidata would be in query systems that can parse natural language queries into structured ones. A simple example would be querying for winners of a certain award that have a verified Twitter account. Such information can easily be obtained from Wikidata with a short SPARQL query. The dataset is also available through weekly "dumps", which represent a snapshot of the database at a given moment. These dumps can also help populate other dedicated databases that may be built for specific purposes.

Wikidata – like many other RDF datasets published as Linked Data [12] – is in constant change, due to it being an open dataset that can be edited by users. Furthermore, data is being imported from external sources constantly, which makes Wikidata very dynamic in

[1] https://www.w3.org/2001/sw/wiki/Main_Page
[2] https://www.wikidata.org/wiki/Wikidata:Statistics

some areas. Given these circumstances, it can be said that Wikidata has many “versions” over time, where different data is available at different time intervals, making it possible to analyze its time component. An important note, however, is that Wikidata does not keep any of its older versions, so the only data accesible at any given time is the latest. Moreover, there is no support for queries about recent changes or which entities or relationships have just been added or removed. Keeping such information would require building a dedicated system and/or database.

Dataset versioning in RDF/SPARQL is a research topic yet to be fully explored. While a number of implementations have been proposed very recently [18, 5], most of them rely on specialized indices or SPARQL extensions. As such, it would be relevant to study the possibility of developing a versioning system and methodology for queries using only base SPARQL with an RDF model capable of representing versioned data and open to be queried without needing specialized tools or software. This way, an off-the-shelf approach is constructed, contrasting it with the currently proposed solutions.

Such a system could be used as version control for the dataset; administrators would be able to detect erroneous or malicious editions on data that does not normally change. On top of that, it would be possible to analyze the evolution of data, possibly predicting future changes. The system could also be used on datasets other than Wikidata, where data history may be poorly (or not at all) preserved.

1.2. Main Challenge

Due to the existence of large RDF datasets in constant change (such as Wikidata), there exists time-based information linked to the data that is completely lost when the dataset is queried on its current “final” version. Moreover, it is impossible to know since when a certain query result has been valid in the dataset, and, unless historic data is kept, there is also no way to determine which results would have been valid in a previous time period. Because of these reasons, it becomes more difficult to analyze the evolution of the dataset or of the solutions to queries over it.

In this work, we assume discrete versions of a dataset, which will typically reflect "snapshots" of it at a certain point in time. Based on these facts, two key questions relevant for research are proposed:

- How to build an RDF database that keeps track of information from both current and past versions of the data?
- How to query over said database?

Keeping time-based or versioned information has several applications that can be of interest, such as archiving dynamic RDF datasets [17], predicting future changes [6], detecting malicious or erroneous editions [10] and drawing conclusions about data evolution [4]. A number of technical challenges arise from this, some of them being:

- Compression of historical data to create a versioned dataset.
- Rewriting queries and their results to add versioned information.
- Achieving the previous points using the existing technology deployed in practice.

1.3. Proposed Solution

The construction of the proposed solution can be separated into two parts:

1. Defining a method to combine several versions of a dataset, creating a versioned dataset.
2. Defining a method for rewriting SPARQL queries so that they can be used over the versioned dataset, without modifying the query engine.

Additionally, this presents an optimization problem regarding these two parts, both in storage space and execution time for the dataset construction and query responses. A first approach would be to keep every version as-is, using the most storage possible. However, such an approach can be easily improved upon, especially in terms of storage size.

Five representation alternatives are considered:

(a) Calculate deltas between consecutive versions using the oldest version as a base

(b) Calculate deltas between consecutive versions using the latest version as a base

(c) Calculate deltas between each version and the oldest, which will be used as a base

(d) Calculate deltas between each version and the latest, which will be used as a base

(e) Build graphs representing different intervals of versions and include each triple in the longest interval(s) for which they are valid, avoiding repetition where possible

An additional alternative without any optimization is added as a comparison point. Every version is available in its entirety, allowing queries over any historical (or current) data. In every representation, named graphs are used to separate between each version, interval or delta within the versioned database.

1.4. Hypothesis

The main hypothesis of this work can be summarized as follows: *a versioned database can be implemented using unmodified versions of RDF and SPARQL and achieve competitive performance for storage space, database building time and query execution times when compared to queries over the current version alone.* Given the context presented, some relevant questions arise. We will discuss each briefly.

1. What representations of a versioned dataset allow for better compression, more efficient indexing, and/or more efficient updates?
2. How should queries be rewritten to be compatible with a versioned dataset?
3. How large is the performance cost of implementing a versioned dataset and query system?
4. Which of the representation alternatives will have the most well-rounded performance?

The rest of this section consists of initial predictions for these questions with respect to weekly versions of Wikidata (as our key use-case). Experiments will be run to determine whether the predictions hold true in practice or not.

1.4.1. What representations of a versioned dataset allow for better compression, more efficient indexing, and/or more efficient updates?

For the compression rate, a simple comparison between the number of total triples in all versions and the one resulting from the compression should be enough. Both can then be compared against the number of triples in the base version to observe the percentual increase.

Due to the fact that monthly editions never exceed 50% of the total size of Wikidata[3], the delta graphs should never exceed 50% of the base version's size when using consecutive deltas. And, given that this project uses weekly versions, the delta size is expected to be considerably smaller than said upper bound. However, when deltas are being calculated against a single base version, it becomes possible that the size of each delta grows. However, given that most of the data does not change, the differences between versions should not be large, achieving a good compression rate regardless. In the case of time intervals, most of the data will likely not appear in more than one graph, since triples should only appear in more than one graph if they were deleted and then added back in a different version, which is very rare in Wikidata. Given those facts, the best compression should be easier to achieve.

The cost of adding a new version to an already built database varies greatly between representations. In the non-optimized version, it requires indexing the new graph without any additional operations. Alternative (a) needs to add the delta between the version being added and its previous one, which requires re-assembling the latter using all the deltas leading up to it. Alternative (b) has a relatively low cost, requiring the same new delta as alternative (a), but since the version being added is preceded by the base, the delta can be calculated directly and the new version can easily replace the current base. Alternative (c) has a similar low cost to alternative (b), requiring only calculating the delta between the version being added and the base, then adding said delta to the available ones. Adding a new version to alternative (d) has the highest cost compared to the other representations, as it requires re-assembling every previous version and re-calculating every delta up to the version being added. As such, it has a similar cost to building the database in its entirety, so it may not be suitable for versioned datasets that frequently add new versions. Alternative (e) needs to recalculate all intervals that ended on the now second-to-last version, separating triples that are valid for the new version from the ones that were removed.

1.4.2. How should queries be rewritten to be compatible with a versioned dataset?

It is expected that every (SPARQL 1.1) query can be rewritten as an extended query (in the same SPARQL 1.1 language) with additional parameters to retrieve versioned information. Query conversion is one-to-one, meaning no extra queries will be needed to represent a previously valid query.

An important note to make, however, is that not all queries will be rewritten in the same way. Depending on the nature of each query, it is possible that rewriting one query in particular will be more difficult than others. Specifically, when it comes to monotone

[3] http://bit.ly/2FwZHXj

queries[4] [1, 2], rewriting them should be easier when the dataset changes are only additions compared to non-monotone queries or when changes are more than just additions.

1.4.3. How large is the performance cost of implementing a versioned dataset and query system?

No matter the representation, querying over a versioned database should, in general, be slower than querying over a non-versioned one (i.e., running the same query over a database indexing just the current version). However, queries concerning the latest version should be answered with competitive performance by the alternatives that use it as a base and in the non-optimized version.

1.4.4. Which of the representation alternatives will have the most well-rounded performance?

A representation having a "well-rounded performace" can be understood as having several strong points while minimizing any drawbacks. Each representation will likely have strong and weak points and, as such, none of them will be the best at everything.

Alternatives (a) and (b) should have the lowest storage size, but may struggle with queries for versions far from their base. Alternative (b) benefits from using the latest version as base for two reasons: queries about recent versions are expected to be more common than queries about older versions, leading to lower query times overall, and adding a new version only involves calculating the delta between the current base version and the one being added. Alternatives (c) and (d) have a better average query time, since accessing any non-base version takes the same amount of time, unlike (a) and (b). Alternative (c) has a low cost for adding a new version, since it only needs to calculate and add one new delta. However, alternative (d) must be rebuilt in its entirety when adding a new version. Alternative (e) has an storage size almost identical to (a) and (b), avoiding repetition of triples as much as possible. Query times for it are consistent and expected to be competitive with alternatives (c) and (d). Its cost of adding a new version is expected to be in the middle of the rest of the alternatives.

1.5. Objectives

1.5.1. General Objectives

The main objective of this work is to develop storage representations for versioned RDF datasets, and methods to rewrite queries over said representations in order to be able to obtain versioned data using only base SPARQL. The different versioning representations will be tested, evaluating advantages and disadvantages in an attempt to determine which one is the best for any given situation.

[4] Let Q be a SPARQL query. Let G, G' be RDF graphs. Q is said to be *monotone* if and only if $Q(G) \subseteq Q(G')$ when $G \subseteq G'$.

1.5.2. Specific Objectives

1. Work simultaneously with multiple weekly Wikidata versions.
2. Define a methodology to calculate differences between two graphs (any problems created by blank nodes will be ignored for this work).
3. Build a versioned dataset containing all versions with no optimization as a baseline for comparing results.
4. Build additional datasets with different representations for versioning using named graphs.
5. Create compression statistics for the versioned databases.
6. Build a framework for the process of rewriting queries and explore possible optimizations.
7. Create experiments to compare query times for different representations of the versioned dataset and the different query rewriting strategies.

Chapter 2

Related Work

In this chapter we'll discuss systems and concepts relevant to this work, as well as contexualize the state-of-the-art and alternative approaches to solve the problem presented in the previous chapter. In particular, both RDF and SPARQL play a fundamental role, being, respectively, the database and query systems that this work is built upon. The concepts of Semantic Web and Linked Data are also discussed, since the work done for this thesis belongs to these research topics.

2.1. Semantic Web

The Semantic Web[1] is an extension of the World Wide Web based on standards set by the World Wide Web Consortium (W3C). Its main goal is to structure and standarize data available on the Internet, making it easier for machines to process and interact with it. This would allow for better comunication between systems, as well as for more automation of complex tasks. To achieve this goal, available data must be coupled with suitable meta-data that describes the nature of its associated data and the relationships it has with other data. Three of the key standards proposed by the W3C are RDF, OWL and SPARQL. RDF is a model that allows representing linked data and connections between it. OWL (short for Web Ontology Language) provides vocabulary for expressing basic relantionships as well as type definitions to label data. SPARQL is the standard query engine for RDF, allowing to evaluate complex queries over RDF databases. Since both RDF and SPARQL are strongly linked to the work of this thesis, they will discussed more in-depth in the following section and chapters. Linked Data is essential to the existence and usability of a Semantic Web and, as such, will be further discussed in a following section.

2.1.1. RDF

The Resource Description Framework [16] (RDF for short) is the standard model for representing data on the Web. Recommended by the W3C, RDF has become widespread for publishing data on the Web, being used on millions of websites, helping the vision of a Semantic Web to become reality[2]. RDF was designed as a system to represent data and its based on *triples*, which are tuples of three elements describing relationships between two

[1] https://www.w3.org/2001/sw/wiki/Main_Page
[2] https://bit.ly/3eNOLGq

resources linked by a *property* in a subject-predicate-object structure. RDF databases can be represented by a graph where resources are the nodes and properties are the edges. An example of a small collection of RDF triples can be seen in Code 2.1 and Figure 2.1.

```
@prefix ex: <http://example.org/>
ex:SuperMetroid ex:developer ex:Nintendo .
ex:SuperMetroid ex:platform ex:SNES .
ex:SuperMetroid ex:genre ex:ActionAdventure .
ex:SuperMetroid ex:genre ex:Platformer .
ex:MegaManX ex:developer ex:Capcom .
ex:MegaManX ex:platform ex:SNES .
ex:MegaManX ex:genre ex:ActionAdventure .
ex:MegaManX ex:genre ex:Platformer .
ex:SNES ex:manufacturer ex:Nintendo .
```

Code 2.1: A sample RDF database

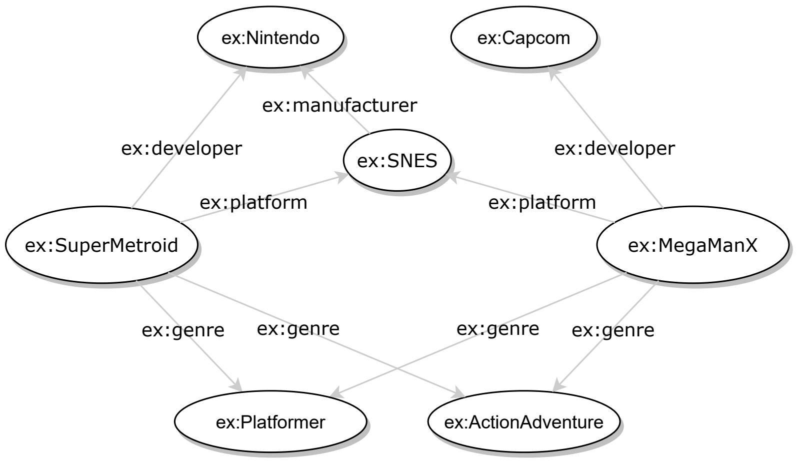


Figure 2.1: Graph representing the sample RDF database in Code 2.1

2.1.2. SPARQL

SPARQL [9] is the standard query engine for RDF (similar to SQL over relational databases). Queries are built using triple matching, unions, insersections and other operations which help constrain the number of matches, navigating and filtering links in the RDF graph. An example of a query and its results over the sample RDF database can be seen in Code 2.2 and Table 2.1. A more in-depth view on SPARQL queries will be covered in Chapter 3.

```
PREFIX ex: <http://example.org/>
SELECT ?game ?developer
WHERE {
   ?game ex:platform ex:SNES .
   ?game ex:developer ?developer .
}
```

Code 2.2: A sample SPARQL query

?game	?developer
ex:SuperMetroid	ex:Nintendo
ex:MegaManX	ex:Capcom

Table 2.1: Query results for Code 2.2

2.1.3. Linked Data

Structuring data is not the only goal of the Semantic Web. Building connections or *links* between data so that it can be easily associated to related data and to allow navigation of the Web is another one of its main objectives. Considering that fact, the concept of Linked Data[3] comes to light. Linked Data is structured data linked to other data. These links between related data form chains useful for semantic querying. Linked Data constitutes the building block for the Semantic Web, aiming to create a global database out of all the data available on the Internet, thus making W3C's vision a reality.

Linked Data (and by extension, the Semantic Web) aims to follow four main principles or *rules*:

1. Use URIs (Universal Resource Identifiers) as names for objects, concepts, etc.

2. Use HTTP URIs so that the names can be looked up.

3. When the URI is looked up, provide useful information using the standards (RDF, SPARQL).

4. Include links to other URIs, making navigation and discovery easy.

The first rule is crucial to make sure every item has a unique identifier so it can be referenced as needed. The second rule allows for mostly standarized URIs that can be easily looked up and linked, given their nature as URLs. The third rule makes it so that humans can access the URLs and get information related to the item in question, making URIs useful for both humans and machines alike. Coupled with the previous rules, the fourth rule allows for easy navigation of related items in a manner that is intuitive for both humans and machines. If the data is published under an open license, it is known as Linked *Open* Data, which can be rated according to criteria proposed by the same author of the four principles. These rating criteria are as follows:

1. The data is available on the web with an open license. Regardless of format, this makes it *open data*.

[3] https://www.w3.org/DesignIssues/LinkedData.html

2. The data is available as machine-readable structured data.
3. Same as the above, but using a non-propietary format.
4. Use open standards from the W3C to identify items.
5. The data is linked to other datasets, thus making it linked data.

As one of the cornerstones of the Semantic Web, Linked Data is used in a large number of databases and applications on the Web[4]. The Linked Open Data cloud[5] collects information about open datasets published as Linked Data and the links between them, representing them visually in a graph diagram including 1200 datasets and over 16000 links, shown in Figure 2.2. One the most prominent use cases for Linked Data is DBpedia [13], an RDF database that makes the content of Wikipedia available as Linked Data, adding links to other relevant databases, such as Geonames[6]. Another example of such a use case is Wikidata, a collaborative RDF database that aims to store information about objects, concepts and their relationships as structured data available for use in Wikimedia projects. Wikidata will be especially important for this work, since it will be used for most experiments. However, the principles and methods discussed in this work can be used any other similar database with a SPARQL endpoint.

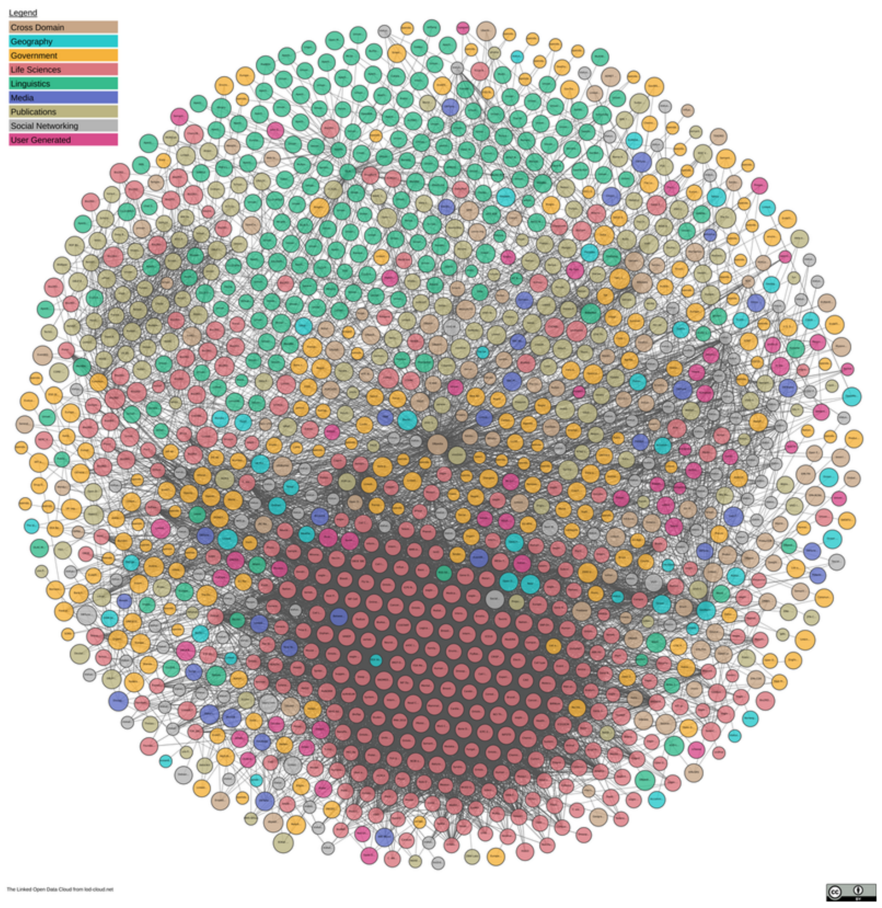


Figure 2.2: Diagram of the Linked Open Data cloud

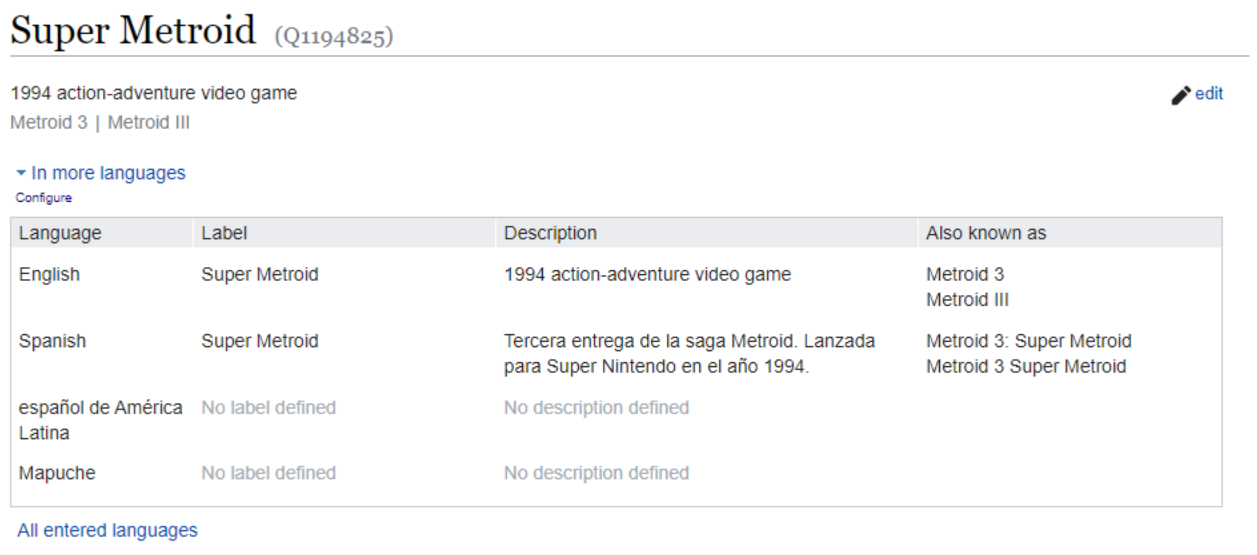
[4] https://www.w3.org/2001/sw/sweo/public/UseCases/
[5] https://lod-cloud.net/
[6] http://www.geonames.org/

2.2. Wikidata

Wikidata [19] is an RDF document-oriented database. Its goal is to store structured data that can be accessed by both humans and machines alike and contain information about people, objects, topics, etc. in a generalized format that can work disregarding constraints such as language. Wikidata is a open dataset, allowing manual and automated edits by users and systems and can be used to feed into other databases such as Wikipedia. Each item in Wikidata has a label and a description that identify them, as well as a unique ID known as *QID*, since each ID starts with the letter Q followed by a numeric value.

Wikidata was originally conceived as a “Wikipedia for data”, managing the structured, factual information that Wikipedia had accumulated so far. As Wikipedia grew in size and scope, it collected a large amount of structured data, such as numeric amounts, dates, coordinates, etc. Given the value of said data, a central repository is needed to store it, while also making sure the data is lifted from restrictions such as language. Wikipedia does not have a way to access its massive amount of data, lacking any kind of query service endpoint or data exports. In light of these facts, Wikidata arose as a solution for these problems, while still keeping the strengths that Wikipedia originally had due to its community-managed approach. The following points characterize Wikidata’s design:

- Open editing: Wikidata allows users to both add and edit information in the database, in a similar way that Wikipedia allows.
- Community control: Both the data and its scheme are controlled by the community of contributors, granting them the power to decide not only which data is stored, but in what way as well.
- Plurality: Data is never treated as the only possible truth, allowing uncertainty or disputed facts to coexist and providing mechanisms to organize conflicting data.
- Secondary data: Factual data is stored along with its source, making it so that data doesn’t exist by itself.
- Multilingual data: Wikidata is multilingual by design. Labels are translated into as many languages as possible, while numeric or similar information with universal meaning is available for any language, unlike Wikipedia.
- Easy access: The data gathered in Wikidata is meant to be able to be used by Wikipedia and external applications. Data is exported in several formats and legally allows reuse.
- Continuous evolution: Similar to Wikipedia, Wikidata is in constant growth, so the platform itself adapts and grows as needed, adding features incrementally.


Super Metroid (Q1194825)

1994 action-adventure video game

Metroid 3 | Metroid III

Language	Label	Description	Also known as
English	Super Metroid	1994 action-adventure video game	Metroid 3 Metroid III
Spanish	Super Metroid	Tercera entrega de la saga Metroid. Lanzada para Super Nintendo en el año 1994.	Metroid 3: Super Metroid Metroid 3 Super Metroid
español de América Latina	No label defined	No description defined	
Mapuche	No label defined	No description defined	

Figure 2.3: Wikidata article for the game "Super Metroid" taken from https://www.wikidata.org/wiki/Q1194825

Wikidata is a massive dataset, including over 84 million *items* and with an active userbase of over 26 thousand as of May 2020[7]. Moreover, it is by no means a static database: over 600 thousand daily additions were made on average for the month of March 2020[8] with the number keeping a mostly increasing rate through Wikidata's history. Wikidata items can be accessed and edited individually using their URIs or downloaded in bulk through weekly "dumps". It can also be queried and edited using its own SPARQL endpoint through HTTP requests or accessing the user interface provided by Wikidata itself[9].

Currently, a number of applications use Wikidata as their source for data, Wikipedia being one of them. Voice assistants, such as Amazon's "Alexa" and Apple's "Siri" get answers to simple queries using data extracted from Wikipedia and Wikidata[10]. Another one of these applications is WikiGenomes [15], an open web application that allows to add, edit and view gene annotation data in Wikidata.

2.3. Versioning of RDF

RDF is a general format for data representation and thus temporal meta-data can be directly encoded without any modifications within an RDF graph and queried using SPARQL. In Figure 2.4, we show an example where the original release and re-releases of the game "Super Metroid" are presented in an RDF graph utilizing blank nodes. However, some authors propose to extend RDF with special annotations to represent various meta-data, including temporal information. In Figure 2.5, the previous graph is simplified utilizing annotations such as the ones proposed. In order to query annotated data, specialized languages and implementations must be developed.

[7] https://www.wikidata.org/wiki/Special:Statistics
[8] https://stats.wikimedia.org/v2/#/wikidata.org
[9] https://www.wikidata.org/wiki/Wikidata:Data_access
[10] https://internethealthreport.org/2019/wikidata-gives-wings-to-open-knowledge/

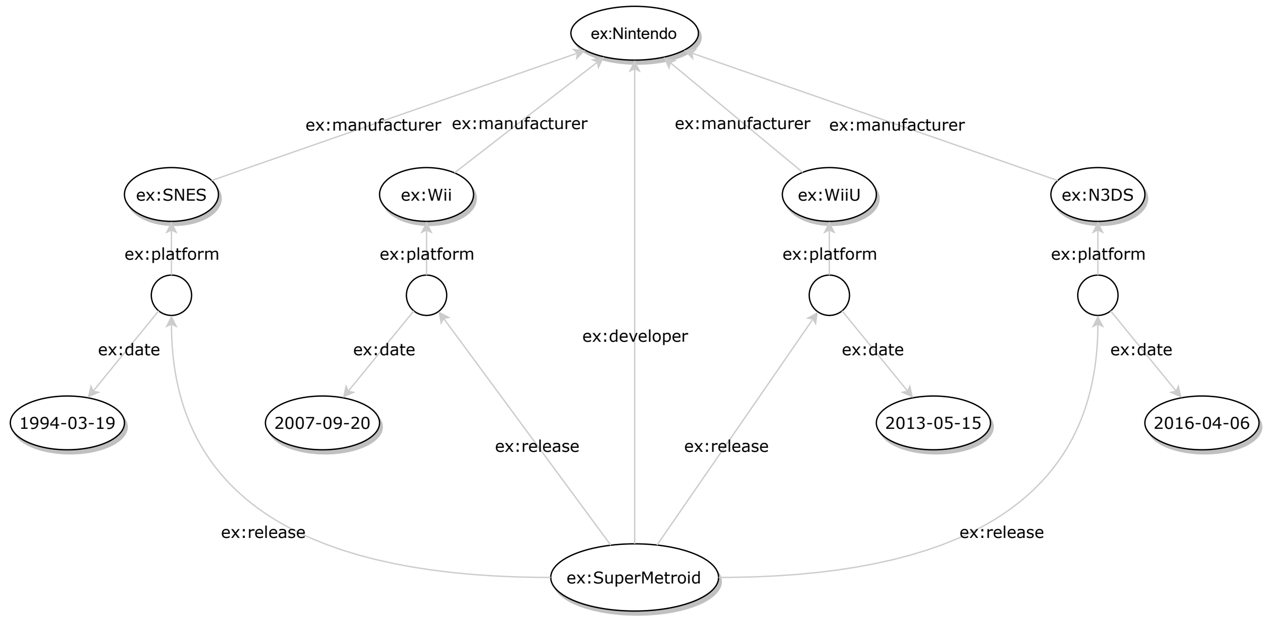


Figure 2.4: Graph representing the various releases of the game "Super Metroid"

Gutierrez et al. [8] describe a framework that allows for representing time in RDF using annotations. An important distinction to make is that their work is focused on **labeling** over **versioning**; **labeling** means adding meta-data to query result validity, while **versioning** implies simultaneously keeping several versions of the same dataset. This work also formally defines labels and time intervals, but does not cover implementation details for either. In order to add time to RDF graphs, this work uses annotations over triples to mark their validity period, which allows the timeline to coexist in a single graph and querying to be simpler. It requires, however, adapting the query language to account for and filter through these annotations and return the query results in a format that accomodates the extra temporal information associated with each triple that has it. The work also formally proves that the addition of the time dimension does not increase complexity in query answering.

Zimmerman et al. [20] extend RDF, allowing the use of generalized annotations, as well as a query language compatible with the annotations. The annotations can be used to represent time intervals associated to each triple to represent their validity, as well as several other applications. The data is not limited to only one type of annotation and use cases where several can be combined are presented as well. The concept of an *annotation domain* is defined, representing annotations that are compatible to combine in some way, such as time. SPARQL is also extended in a similar way, producing a query language they call *AnQL*. Instead of using Basic Graph Patterns (BGPs), AnQL utilizes *Basic Annotated Patterns* that work in a similar way to BGPs but allow annotations in the query. As per the previously mentioned work, the objectives sought by Zimmerman et al. do not completely align with the present work, due to the fact that an RDF extension and custom query language are used.

Grandi [7] proposes an extension for SPARQL, including a time component in queries. It also introduces the concept of a *multi-temporal RDF database* where triples are defined as $(s, p, o|T)$ where T represents the union of all time intervals where (s, p, o) is valid. A datatype called `xs:period` defined as a tuple of dates is used to represent each data interval.

While the concepts are properly introduced and defined, no practical implementation details are provided.

The prior three works share their focus on formal definitions over implementations. They also do not, for the most part, consider efficiency or scalability analysis and employ SPARQL extensions and/or specialized indices. In light of these facts, said works have a slightly different focus to ours, where we aim to implement versioning over off-the-shelf SPARQL implementations.

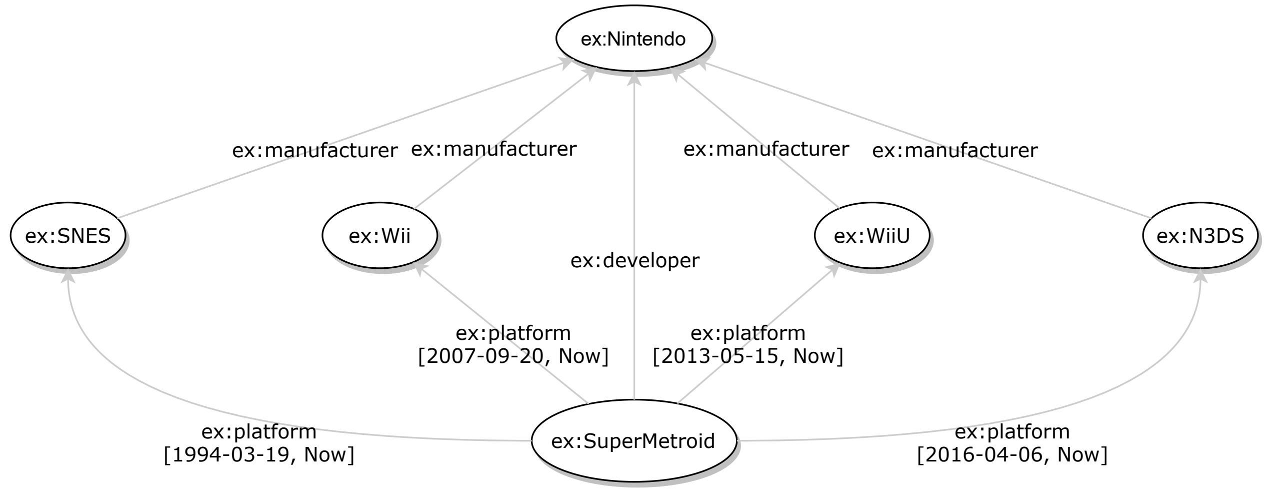


Figure 2.5: Simplification of the graph in Figure 2.4 using annotations

Tappolet and Bernstein [18] add a time component to RDF's syntax, proposing an efficient method to make SPARQL queries over such data. The dataset is annotated with time intervals corresponding to each triple's validity, and a specialized index complements the query engine. An extended version of SPARQL called τ-SPARQL is presented as an alternative to query over these specialized dataset. τ-SPARQL can be mapped to standard SPARQL without a loss of expressivity, making it mostly used for convenience. Among the annotations presented, the concept of *snapshot* queries is introduced. Queries can be filtered for an specific time point using the custom `FROM SNAPSHOT` statement or temporal wildcard variables can be use to extract the validity period of each result.

Lastly, Fernández et al. [5] formally define the concepts of an *RDF Archive* and an *RDF Version* with *Deltas* being the difference between two Versions. They also propose several metrics to evaluate and characterize the dynamicity of an RDF dataset. The work also describes an implementation for storing such archives, a SPARQL extension for making queries on them, and a testing suite for RDF archives.

These two works share a similar focus to the one presented in our work. However, they also employ extensions to either RDF or SPARQL (or both) and will be mostly considered as reference points, aiming to achieve similar performance without the use of extensions or specialized indices.

Chapter 3

Preliminaries

In this chapter we present several concepts relevant to the thesis, as well as their formal definitions.

3.1. RDF

RDF [16] is a framework designed for data representation and storage. In Figure 2.1, we provided an example of a small RDF graph. Since RDF is the data format used in this work, the concepts associated with its design will be explained further, including notation that will be used throughout this work.

3.1.1. RDF terms

Some of the basic concepts associated with RDF are its smallest components, called *RDF terms*. They are as follows.

- An *Internationalized Resource Identifier* (IRI for short) is an expansion of Universal Resource Indentifiers (URIs) with added Unicode support. Their main function is, as their name implies, providing a unique identifier for the *resources* in an RDF graph. Following the Sematic Web principles, IRIs usually use HTTP link format. Prefixes common among several IRIs can be defined separately for the sake of brevity, displaying them as `prefix:resource`. The set containing all IRIs will be referred to as **I**.

- A *Literal* is an instance of a datatype, such as an integer, date, string, etc. Literals are used to store numeric or otherwise structured data that can be associated to resources. The set containing all Literals will be referred to as **L**.

- A *Blank Node* is an element of an RDF graph that represents a resource without a URI or literal. Blank Nodes are typically used as connectors for complex relationships that cannot be expressed in a single triple. In an RDF database, Blank Nodes are typically prefixed with _: followed by string that gives them a local label. The set containing all Blank Nodes will be referred to as **B**.

3.1.2. RDF triple

An RDF triple is a tuple of three elements describing a relationship between two *resources* linked by a *property* in a subject-predicate-object structure. Properties denote the nature of the relationship between the two resources, and are represented by an IRI. Resources can be one of three alternatives: IRIs, Blank Nodes or Literals. Literals, however, are not typically used as subjects, and blank nodes are never used as predicates. In general, a triple can be defined as:

$$(s, p, o) \in (\mathbf{I} \cup \mathbf{B}) \times (\mathbf{I}) \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$$

Triples can also be represented by a connection in a graph where each node is an IRI, Blank Node or Literal and the edge is labeled with a Property. A collection of RDF triples is known as an *RDF graph* and typically denoted by the letter G. We provided an example of an RDF graph in Figure 2.1.

3.1.3. RDF graph and dataset

An RDF graph is a set of RDF triples that encode information (see Figure 2.1 for an example). An RDF dataset consists of one or more RDF graphs, where one RDF graph is called the default graph, and other graphs are named with an IRI or blank node. A brief formalization of this concept is as follows: An RDF dataset D is the set $D := \{G_0, (x_1, G_1), \dots, (x_n, G_n)\}$ where $G_0, \dots, G_n$ are RDF graphs and $\{x_1, \dots, x_n\} \subseteq (\mathbf{I} \cup \mathbf{B})$. Each pair (x_i, G_i) $(1 \leq i \leq n)$ is called a *named graph* while G_0 is called the *default graph*. The identifiers x_i $(1 \leq i \leq n)$ are all different from each other in order to uniquely identify each graph in the set. These identifiers may freely appear in any triples, however. This allows to add data related to the graph linked to the identifier, among other uses.

The graphs contained in an RDF dataset can be navigated and queried using SPARQL, allowing the retrieval of the information stored. For this work, we will assume SPARQL queries over different versions of a single RDF graph (a default graph) whose content changes over time. These versions and changes will be encoded and represented using named graphs in a few different ways, as will be presented next chapter. We will discuss SPARQL in more detail in the following.

3.2. SPARQL

SPARQL [9] is the standard query language and engine for RDF. It's mainly based on *graph pattern matching* with variables and operators that allow for searching patterns in the RDF graph and returning the matches as solutions for the query. In Code 2.2 we provided an example of a SPARQL query, whose results are shown in Table 2.1. We now introduce SPARQL in more detail, along with notation that will be used later. This notation largely follows the conventions first introduced by Perez et al.[14] for SPARQL.

3.2.1. Basic Concepts

SPARQL queries use RDF terms for pattern matching, as well as elements unique to SPARQL. Some SPARQL concepts are explained below:

- A *Variable* is an element of SPARQL queries that acts as a wildcard for values in its pattern matching, allowing any value to match that position. Variables are identified with strings that start with the character `?`. An example of a Variable would be `?var`. The set containing all Variables will be referred to as $\mathbf{V}$, which is considered disjoint from IRIs, Blank Nodes and Literals.

- A *Filter Condition* is an expression that uses elements from $(\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \mathbf{V})$ and constants, alongside arithmetic comparisons and built-in operations that result in a boolean value. Filter Conditions are used to constrain the results of a SPARQL query.

3.2.2. Basic Graph Pattern

A *Triple Pattern* is the most simple pattern in SPARQL queries, used to match triples. It has a very similar syntax to an RDF triple, with the exception that each part (subject, predicate or object) can be replaced with a variable. As such, a Triple Pattern can be defined as:

$$t \in (\mathbf{I} \cup \mathbf{B} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \mathbf{V})$$

Whenever a triple matches a Triple Pattern, the value taken by the variables is recorded, and can later be displayed in the results of the query. A *Basic Graph Pattern* (BGP for short) is a set of Triple Patterns, and is the base of SPARQL queries.

3.2.3. SPARQL query

A SPARQL query is a set of Basic Graph Patterns and operators which can find matches in the RDF graph. `UNION` operators can be used to match multiple alternatives in the query, while `OPTIONAL` statements allow to include data only if it exists, without adding constraints to the results already obtained. Finally, a *Projection* can be applied using `SELECT`, choosing which variables are displayed in the results table. A few other options exist for the final step of the query, such as `ASK` queries which return a boolean value if a binding exists; or `CONSTRUCT` queries, which return an RDF graph according to the parameters given. Our work, however, will be focused on projections or `SELECT` queries. Following these parameters, a SPARQL *query pattern* can be defined formally as such:

- If t is a triple pattern then t is a query pattern.
- If Q_1 and Q_2 are query patterns, the following expressions are also query patterns:
  - $Q_1 \, \texttt{UNION} \, Q_2$.
  - $Q_1 \, . \, Q_2$ (conjunction). Also represented as $Q_1 \, \texttt{AND} \, Q_2$.
  - $Q_1 \, \texttt{OPTIONAL} \, Q_2$.
- If Q is a query pattern and F is a filter condition, then $Q \, \texttt{FILTER} \, F$ is a query pattern. This pattern can also be expressed as $\texttt{FILTER}_F(Q)$
- Finally, if Q is a query pattern and V is a set of variables, then $\texttt{SELECT} \, V \, \texttt{WHERE} \, Q$ is also a query pattern, called a *projection*. For simplicity, this pattern is expressed as $\texttt{SELECT}_V(Q)$.

– The keyword `DISTINCT` can be added to a `SELECT` statement to remove duplicated results that are otherwise included by default. This is typically expressed as $\texttt{SELECT}^{\Delta}_{V}(Q)$ where Δ takes a boolean value: **true** when `DISTINCT` is used, and **false** when it isn't.

Graph operators

SPARQL allows for queries over several graphs and, as such, datasets may use this feature to store data of different origins alongside each other. A subset of operators are dedicated to choose and determine the origin of the triples matched. The data matched can be restricted to only a subset of all graphs available, or the graph IRIs themselves can be matched and filtered using a variable and pattern matching or filter conditions. As mentioned earlier in Section 3.1.3, RDF datasets have one Default Graph and several Named Graphs. If no graph is specified in a query, the Default Graph is used as origin for the data. However, a new default graph may be specified for the query using the `FROM` statement. In more formal terms, preceding a query Q with the expression `FROM` G declares that the *evaluation* of Q will be done over G. In other words, `FROM` G $\{Q\}$ is equivalent to $Q(G)$. We discuss the concept of evaluation in Section 3.2.4. Alternatively, a graph may be included in the query using the `FROM NAMED` statement. The graph can be later accessed using the `GRAPH` statement. When a named graph is specified in the `GRAPH` statement, the effect is equivalent to changing the default graph for that section of the query. That is to say, a query of the form `FROM NAMED` G $\{$`GRAPH` G $\{Q\}\}$ is equivalent to $Q(G)$. The behaviour of a `GRAPH` statement using a variable (such as `GRAPH` $?g$) is slightly more complex, but can be expressed as: a query of the form `GRAPH` $?g$ $\{Q\}$ is equivalent to $\bigcup_{G \in \mathcal{G}} Q(G)$ with $\mathcal{G}$ being the set containing all named graphs declared for the query; furthermore, in the solutions $Q(G_i)$ for each named $G_i \in \mathcal{G}$, the variable $?g$ will be mapped to n_i, the name of G_i.

Other operators

We remark that SPARQL contains further operators such as `LIMIT`, `ORDER BY`, etc., that, for brevity, we do not consider explicitly in these definitions. These operators can be supported by applying them over the results of our framework, without modification to their standard definitions [9].

3.2.4. Solution mapping

Applying a query Q over an RDF graph G is denoted as $Q(G)$ and called an *evaluation*. The evaluation produces a set of partial mappings for the projected variables represented as the partial function μ such that $\mu : \mathbf{V} \to (\mathbf{I} \cup \mathbf{L})$. The domain of μ specifies which variables (a subset of $\mathbf{V}$) are considered for the mapping, and is denoted by $dom(\mu)$. μ is only defined within its domain. Two mappings μ_1, μ_2 are said to be *compatible* when for all $?v \in dom(\mu_1) \cap dom(\mu_2)$ then $\mu_1(?v) = \mu_2(?v)$. In essence, for two mappings to be compatible, they must assign the same values to their shared variables. As such, two disjoint mappings are always compatible, and the empty mapping $\mu_\emptyset$ is compatible with any other mapping.

Since evaluation is defined in terms of mappings, it becomes useful to define SPARQL operators in the same manner. But first, some operations between mappings must be defined.

Let Ω_1 and Ω_2 be sets of mappings, then:

- $\Omega_1 \bowtie \Omega_2$ is the join operation, defined as:
  $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ and μ_1, μ_2 are compatible mappings$\}$.
- $\Omega_1 \cup \Omega_2$ is the union operation, defined as:
  $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1$ or $\mu \in \Omega_2\}$.
- $\Omega_1 \setminus \Omega_2$ is the difference operation, defined as:
  $\Omega_1 \cup \Omega_2 = \{\mu \in \Omega_1 \mid$ for all $\mu' \in \Omega_2$, μ and μ' are not compatible$\}$.
- With the previous definitions, we can also define the left outer-join ($\Omega_1 ⟕ \Omega_2$) as:
  $\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

Additionally, in order to properly describe the use of filter conditions in a query, we must introduce the concept of a mapping *satisfying* a filter condition. For a mapping μ and a filter condition F, this is denoted $\mu \models F$. Let c be a constant, $?X, ?Y$ variables, and F', F_1, F_2 filter conditions. We define $\mu \models F$ such that:

- If $F = \texttt{bound}(?X)$, then $\mu \models F \iff ?X \in \mathit{dom}(\mu)$.
- If $F = \mathit{op}(?X)$ with *op* being a built-in unary boolean operator other than `bound`, then $\mu \models F \iff [?X \in \mathit{dom}(\mu) \implies \mathit{op}(\mu(?X))]$. Unlike `bound`, other unary operators don't require $?X \in \mathit{dom}(\mu)$. In the case that $?X \notin \mathit{dom}(\mu)$ then the value of $\mathit{op}(\mu(?X))$ is given as an error. Solutions that return an error for a filter condition will be removed from the results.
- If F is an arithmetic comparison (such as $=, >, <$, etc.) between $?X$ and c, then $\mu \models F$ if and only if $?X \in \mathit{dom}(\mu)$ and the same comparison holds true for $\mu(?X)$ and c.
- If F is an arithmetic comparison (such as $=, >, <$, etc.) between $?X$ and $?Y$, then $\mu \models F$ if and only if $?X \in \mathit{dom}(\mu)$, $?Y \in \mathit{dom}(\mu)$ and the same comparison holds true for $\mu(?X)$ and $\mu(?Y)$.
- If $F = \neg F'$, then $\mu \models F$ if and only if μ does not satisfy F'.
- If $F = F_1 \vee F_2$, then $\mu \models F$ if and only if $\mu \models F_1$ or $\mu \models F_2$.
- If $F = F_1 \wedge F_2$, then $\mu \models F$ if and only if $\mu \models F_1$ and $\mu \models F_2$.

Now that we have defined operations over mappings, we can then define query pattern evaluations over a graph G in the same manner. $\mu(t)$ represents applying the mapping μ to the variables in a triple t, which will be in turn denoted by $V(t)$. Let t be an SPARQL triple pattern, Q', Q_1, Q_2 query patterns, and F a filter condition. The following definitions characterize the evaluation $Q(G)$.

- If $Q = t$, a single triple pattern, then $Q(G) = \{\mu \mid \mathit{dom}(\mu) = V(t)$ and $\mu(t) \in G\}$.
- If $Q = Q_1 \, \texttt{AND} \, Q_2$, then $Q(G) = Q_1(G) \bowtie Q_2(G)$.
- If $Q = Q_1 \, \texttt{UNION} \, Q_2$, then $Q(G) = Q_1(G) \cup Q_2(G)$.
- If $Q = Q_1 \, \texttt{OPTIONAL} \, Q_2$, then $Q(G) = Q_1(G) ⟕ Q_2(G)$.
- If $Q = Q' \, \texttt{FILTER} \, F$, then $Q(G) = \{\mu \mid \mu \in Q'(G)$ and $\mu \models F\}$.

3.2.5. SPARQL 1.1

So far we've discussed formal definitions for operations defined by SPARQL 1.0. SPARQL 1.1 is an extension of SPARQL 1.0 adding support for nested projections, aggregations, negations and other features that further complex queries. For the most part, this work implicitly offers support for SPARQL 1.1 features, which can be applied over the results generated from the SPARQL 1.0 operators previously discussed, with one main exception: *property paths*. We now discuss property paths, as well as the `MINUS` feature of SPARQL 1.1 that will become important later.

Property paths

Property paths are an extension of triple patterns that allow matching paths of arbitrary length, replacing the predicate in a triple pattern with a *path element*. Path elements are defined recursively as follows:

- A predicate IRI p is a path element.
- If e, e_1, e_2 are path elements, then the following are also path elements:
  - $\hat{}e$: an inverse path (object to subject).
  - (e): a group path e with brackets to control precedence.
  - e_1/e_2: a sequence path of e_1 followed by e_2
  - $e_1\hat{}e_2$: shorthand for $e_1/\hat{}e_2$
  - $e_1|e_2$: an alternative path where both e_1 and e_2 are tried.
  - e^*: a path of zero of more occurrences of e.
  - $e+$: a path of one or more occurrrences of e.
  - $e?$: a path of zero or one occurrences of e.

Property paths can be divided into two different types: *simple* and *complex*. Paths are considered "simple" if they only use sequence (/), inverse (^), zero-or-one (?) and alternative (|). Simple paths can be converted into other operators and, therefore, do not require any extension or special evaluation. Paths are considered "complex" if they contain operators other than the ones allowed in simple paths ($*$ and $+$). Such paths require an algebraic extension. Due to the nature of the implementation of versioning (discussed in Chapter 4), this work only offers support for simple paths, which are converted to their equivalent BGPs. Property paths in SPARQL 1.1 can only be applied within a single graph at a time, which an be either the default graph or a named graph. It is not possible to apply a property path over multiple named graphs loaded with a `FROM NAMED` statement, for example. While this issue can be avoided by combining multiple graphs in the default graph, there would be no way to distinguish the graphs contained within it.

Negation and Filtering

SPARQL 1.1 introduces several new ways to filter queries and substract results. For this work, the operation `MINUS` is the most important of these, since it will be integral to the query design that will be explained in the next chapter. Following the previous definitions in terms of solution mappings, we add the formal definition of the `MINUS` operation between two graph patterns. Let Q_1, Q_2 be query patterns and G an RDF graph. If $Q = Q_1 \, \texttt{MINUS} \, Q_2$ then $Q(G) = Q_1(G) \setminus Q_2(G)$.

Other Features

SPARQL 1.1 introduces several features besides the ones already presented, such as `SERVICE` ,which allows interfacing with other systems or services to provide data in SPARQL queries; and `VALUES`, which allows the assignment of fixed values to a variable. While they're not discussed or directly used in this work, they can be supported as per their standard definitions.

Chapter 4

Native RDF Versioning

In this chapter we define the construction and inner workings of our versioning representation and query conversion techniques. Since multiple versioning representation alternatives are used, a brief analysis of their expected advantages and disadvantages is included with each explanation.

4.1. Preliminary Concepts

We first begin by introducing some preliminary concepts relating to versioned (RDF) graphs, graph deltas, and temporal queries.

4.1.1. Versioned Graph

We call an RDF graph that changes through time a *Versioned Graph*. Snapshots of the graph are taken at different points in time, creating different *versions* of the graph. Time increments are considered discrete instead of continuous for simplicity. While each version can be identified with its timestamp, version numbers are usually preferred for brevity. In more formal terms, a versioned graph is the set $\mathcal{G} = (G_1, \ldots, G_n)$ where each G_i $(1 \leq i \leq n)$ represents a version of the graph, with $1, \ldots, n$ being the version numbers. In the following sections we discuss several methods to represent and store the versioned data corresponding to a versioned graph using only RDF and SPARQL. We'll call these datasets containing temporal information a *Temporal* or *Versioned* Dataset. Each version of the RDF graph must be processed in order to be added to a Temporal Dataset. We restrict both the processing and storage of the temporal data to RDF and SPARQL, although some pre-processing is allowed. Moreover, whenever a new version is added to an existing Temporal Dataset, it must first be proccessed in the similar way, using both the already existing versions in the dataset and the new version being added. This can be achieved by SPARQL, keeping our restrictions possible.

4.1.2. Graph Deltas

Let G_0, G_1 be two RDF graphs representing two versions of the same dataset. A *Delta* between versions is defined as the pair of graphs:

$$\Delta(G_0, G_1) = (\Delta^+(G_0, G_1), \Delta^-(G_0, G_1))$$

$$\text{where } \Delta^+(G_0, G_1) = G_1 \setminus G_0 \text{ and } \Delta^-(G_0, G_1) = G_0 \setminus G_1$$

Here $G_0 \setminus G_1$ represents set subtraction. Given this definition, it is easy to observe that for any two graphs G_0 and G_1, $\Delta^+(G_0, G_1) = \Delta^-(G_1, G_0)$ and $\Delta^-(G_0, G_1) = \Delta^+(G_1, G_0)$.

For the purposes of this work, two types of Graph Deltas are defined: *Absolute* and *Sequential* Deltas. These will be discussed in Section 4.5 and Section 4.6, respectively.

4.1.3. Temporal Query

A *Temporal Query* will be defined for this work as a special type of SPARQL query that extracts data from a versioned dataset. Two types of Temporal Queries are used in this work for the most part: *Single Version* and *Version Delta* queries.

4.1.3.1. Single Version query

A Single Version query is a Temporal Query Q_i that returns solutions that were available in a specific version of the dataset G_i. In other words, $Q_i(\mathcal{G})$ is equivalent to $Q(G_i)$ where Q is the base query and i represents the version ID. In order to do this, the default graph of the query dataset is constructed from the Versioned Dataset such that the solutions over the default graph equate to the solutions over the requested version. In most representations, however, the default graph is insufficient to achieve this result, and named graphs must be used as well. For simplicity, evaluation of Single Version queries will be denoted as $Q_i(G_i)$.

4.1.3.2. Version Delta query

A Version Delta query Q_i^* extracts only the new results for a specific version of temporal dataset. Formally, $Q_i^*(\mathcal{G}) = \{\mu \in Q_i(G_i) \mid \mu \notin Q_i(G_{i-1})\}$ for $i > 1$ or $Q_i(G_i)$ otherwise. In order to achieve this, two queries are built to obtain results from the target version and the version preceding it in a similar manner to a Single Version query. Each query is projected independently and their results are subtracted, creating a *Solution Delta* containing only the results added in the target version. As with Single Version queries, the evaluation of a Version Delta query will be denoted as $Q_i^*(G_i)$ for simplicity.

4.2. Running Example

In order to better illustrate both the construction of the Versioned Dataset alternatives and their associated query conversion processes, a running example is presented. This running example will be based on the RDF graph in Figure 2.4 separated in 4 versions and giving blank nodes labels in order to avoid issues related to them, as can be seen in Figure 4.1. A mistake on one of the dates was included in the second version in order to further complex the versioned dataset. A simple query presented in Code 4.1 will be used as reference for

```
PREFIX ex: <http://example.org/>
SELECT ?platform ?date
WHERE {
   ex:SuperMetroid ex:release ?r .
   ?r ex:date ?date .
   ?r ex:platform ?platform .
}
```

Code 4.1: Query for the running example

the query conversions. In order to properly show the different conversion methods, the third version of the graph will be used as the target version of the queries.

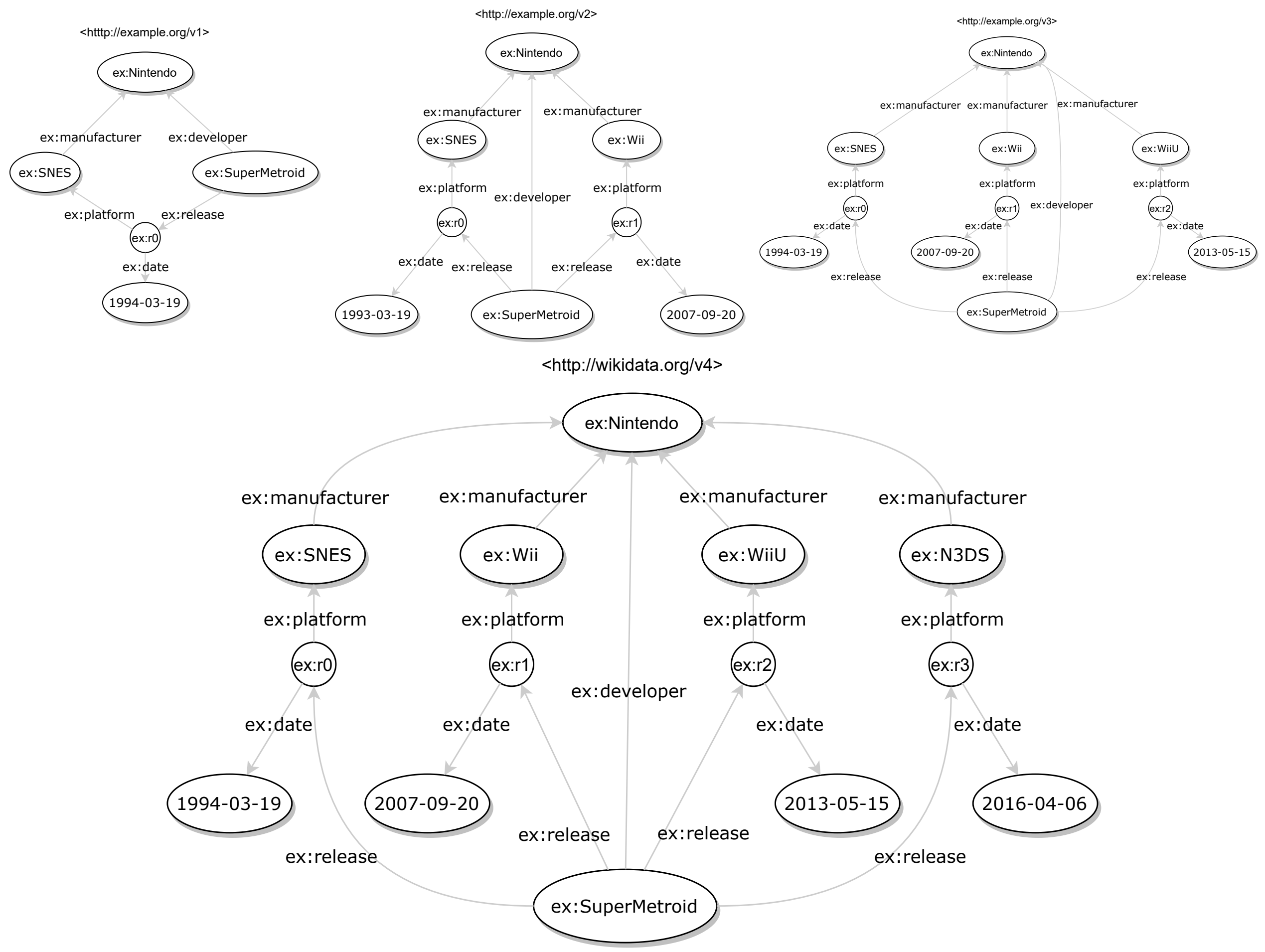


Figure 4.1: Graphs representing the four versions of the running example graph

4.3. Complete Versions

Keeping each version in its entirety in separate named graphs serves as a way to compare statistics with the rest of the representations. As such, this would-be “naive” approach will be referred to as the *Baseline* from now on.

4.3.1. Construction details

4.3.1.1. Dataset construction

In order to build the dataset, all versions included are labeled accordingly as named graphs. This makes sure the versions can be independently queried. For the running example, the four versions are labeled `ex : v1`, `ex : v2`, `ex : v3`, and `ex : v4`. None of the graphs are modified, since they are included in full.

4.3.1.2. Query conversion

In Single Version queries, the body of the query is intact, since the only neccesary changes are made to the data source of the query. The named graphs are filtered using a `FROM` statement in SPARQL to extract results only from the appropiate graph, as can be seen in Code 4.2. In the case of Version Delta queries, the target version is assigned to the default graph while the preceding version is accessed using `FROM NAMED` and `GRAPH` statements. An example of a Version Delta query using the baseline dataset can be seen in Code 4.3.

4.3.2. Expected advantages

Since all versions are completely available, any query, particularly Single Version queries, can be answered in competitive time, disregarding any particularities they may have. The only required step for rewriting a Single Version query is adding the corresponding graph in a `FROM` statement. Version Delta queries are easy to rewrite as well, only requiring an extra named graph and the same query wrapped in a graph statement in order to compute the difference of results for the two different versions. Adding versions to this representation is another one of its strong points, since it only requires adding it to the available versions as a new named graph.

4.3.3. Expected disadvantages

As mentioned before, since this version doesn't compact the dataset in any way, the storage size will be the biggest possible with abundant repetition of triples. Even triples that never change will be present in each named graph, and, as the number of versions grows, the dataset might become too massive to handle easily.

```
PREFIX ex: <http://example.org/>
SELECT ?platform ?date
FROM <http://example.org/v3>
WHERE {
   ex:SuperMetroid ex:release ?r .
   ?r ex:date ?date .
   ?r ex:platform ?platform .
}
```

Code 4.2: Single Version query (Q_3) using the baseline dataset

```
PREFIX ex: <http://example.org/>
SELECT ?platform ?date
FROM <http://example.org/v3>
FROM NAMED <http://example.org/v2>
WHERE {
   {  SELECT ?platform ?date
      WHERE {
         ex:SuperMetroid ex:release ?r .
         ?r ex:date ?date .
         ?r ex:platform ?platform .
      }
   }
   MINUS
   {  GRAPH <http://example.org/v2>
      {  SELECT ?platform ?date
         WHERE {
            ex:SuperMetroid ex:release ?r .
            ?r ex:date ?date .
            ?r ex:platform ?platform .
         }
      }
   }
}
```

Code 4.3: Version Delta query (Q_3^*) using the baseline dataset

4.4. Interval-based graphs

In this representation, an approach based on time intervals was selected; each graph of the dataset represents a time interval, containing all the triples that are valid in said period. Since the time is determined by the versions used, it is measured discretely and determined by the timestamp of each given version.

For graphs representing time intervals, the following definition applies: A triple t belongs to the interval $I(i, j)$ if and only if t is present for all versions between i and j, both inclusive, and $I(i, j)$ is maximal in this sense. By maximal we mean that either i is the first version or t is not in $i - 1$, and that either j is the last version or t is not in $j + 1$. According to this definition, a triple may belong to several interval graphs if and only if said intervals are discontinuous and the previous conditions apply.

4.4.1. Construction Details

4.4.1.1. Dataset construction

In order to construct the dataset, interval graphs are built for every possible interval and labeled accordingly. First, each individual version of the graph has its triples sorted lexicographically. This is done in order to help with the construction process. In the next step, the versions are read in parallel, determining the validity of each triple and in which intervals it belongs, populating the new graphs. The comparison of graphs only reads each version once, and, along with the sorting, ends up with a complexity of $O(mn\log(n) + mn)$ where m is the amount of versions and n is the size of the largest version. For the running example, several graphs are empty due to the fact that not many triples are removed between versions. The dataset converted to interval graphs can be seen in Figure 4.2

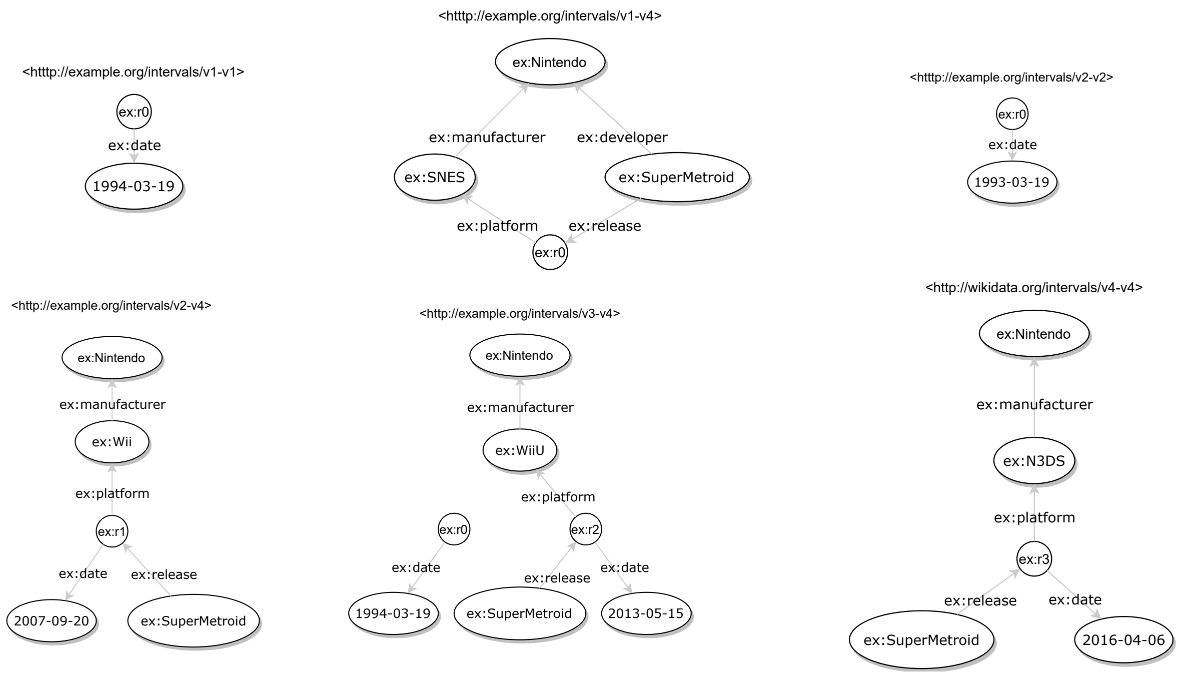


Figure 4.2: Interval graphs representing the running example graph and its versions

```
PREFIX ex: <http://example.org/>
SELECT ?platform ?date
FROM <http://example.org/intervals/v1-v3>
FROM <http://example.org/intervals/v1-v4>
FROM <http://example.org/intervals/v2-v3>
FROM <http://example.org/intervals/v2-v4>
FROM <http://example.org/intervals/v3-v3>
FROM <http://example.org/intervals/v3-v4>
WHERE {
   ex:SuperMetroid ex:release ?r .
   ?r ex:date ?date .
   ?r ex:platform ?platform .
}
```

Code 4.4: Single Version query (Q_3) using interval graphs

4.4.1.2. Query conversion

The relevant interval graphs are merged into the default graph at the runtime of a query using the `FROM` statement. Only intervals containing the target version are used, such that only triples existing in the target version are considered for the query. An example of a Single Version query using this construction method can be seen in Code 4.4, which asks for results that were valid in version 3 of the dataset. Since the intervals are combined as the default graph, this query method does not requires the addition of `MINUS` statements, and, as such, still provides full support for queries containing complex property paths in Single Version queries. This does not hold true, however, for Version Delta queries.

In order to convert a query into a Version Delta query using interval graphs, the target version can be written in the same way as a Single Version query, while the preceding version must be separated into its triple patterns, wrapping each of them in `GRAPH` statement and then filtering them to only keep relevant versions. An example of a Version Delta query using interval graphs can be seen in Code 4.5.

4.4.2. Expected advantages

Querying over an inteval-based approach only requires filtering the graphs which are relevant to the target version of the query, making the conversion rather simple. As such, querying times should not be greatly impacted. Compaction should be near-ideal since triples should not be repeated in different graphs unless they are intermittently added and removed across different versions, which we claim to be rare in practice.

4.4.3. Expected disadvantages

Adding a new version has a moderate impact on re-indexing, affecting all intervals containing the version previous to the one being added. A filter needs to be applied to determine whether a triple continues to be valid towards the new latest version or not. Assuming in general that for a new version G_{n+1} that $G_n \setminus G_{n+1}$ will be smaller than $G_{n+1} \cap G_n$, an optimization to reduce the impact of adding a new version would be using open intervals without

```
PREFIX ex: <http://example.org/>
SELECT ?platform ?date
FROM <http://example.org/intervals/v1-v3>
FROM <http://example.org/intervals/v1-v4>
FROM <http://example.org/intervals/v2-v3>
FROM <http://example.org/intervals/v2-v4>
FROM <http://example.org/intervals/v3-v3>
FROM <http://example.org/intervals/v3-v4>
FROM NAMED <http://example.org/intervals/v1-v2>
FROM NAMED <http://example.org/intervals/v1-v3>
FROM NAMED <http://example.org/intervals/v1-v4>
FROM NAMED <http://example.org/intervals/v2-v2>
FROM NAMED <http://example.org/intervals/v2-v3>
FROM NAMED <http://example.org/intervals/v2-v4>
WHERE {
    {  SELECT ?platform ?date
       WHERE {
          ex:SuperMetroid ex:release ?r  .
          ?r  ex:date ?date  .
          ?r  ex:platform ?platform  .
       }
    }
    MINUS
    {  SELECT ?platform ?date
       WHERE {
          GRAPH ?g0 { ex:SuperMetroid ex:release ?r . }
          GRAPH ?g1 {?r ex:date ?date . }
          GRAPH ?g2 {?r ex:platform ?platform . }
       }
    }
}
```

Code 4.5: Version Delta query (Q_3^*) using interval graphs an end version, interpreted as valid up the current version. This would reduce the amount of triples in need of updating from $(G_n \cap G_{n+1}) \cup (G_{n+1} \setminus G_n)$ to $(G_n \setminus G_{n+1}) \cup (G_{n+1} \setminus G_n)$. Since this optimization is not implemented in this work, it is possible than a larger number of operations than necessary will take place when adding a new version.

4.5. Absolute deltas

In this representation, a version is used as a base and kept in its own graph. Deltas are calculated for each remaining version compared against the base and kept in separate graphs. While in theory any version could serve as a base, only the first or latest are selected in this work. When the first version is selected as a base, the scheme is similar to *differential backups* that are sometimes used in database systems. When the latest version is selected, the scheme is similar to *reverse differential backups* sometimes used in archiving systems.

4.5.1. Construction details

4.5.1.1. Dataset construction

As previously stated, the base version (either the first or latest) is kept in its entirety in its own graph. Positive and negative deltas are then calculated for each remaining version against the base and kept in separate graphs. In order to keep the graph names unambiguous and short, deltas are named in a way that makes them all negative. For any positive delta needed, the order of versions is inverted, since $\Delta^+(G_0, G_1) = \Delta^-(G_1, G_0)$ for any pair of graphs G_0, G_1. For the running example, the first version was used as a base. Since no triples are removed, there are no negative deltas from the base version. The versioned dataset for the running example using absolute deltas can be seen in Figure 4.3.

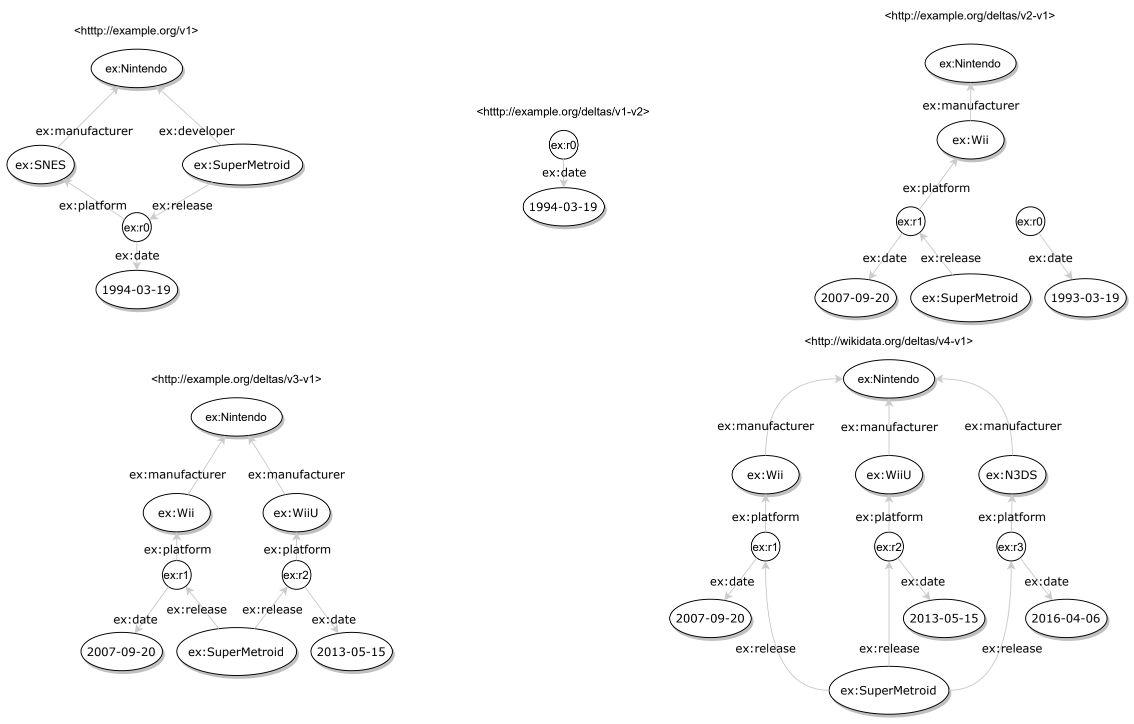


Figure 4.3: Absolte delta graphs representing the running example graph and its versions

4.5.1.2. Query conversion

In Single Version queries the base version and the positive delta for the target version are combined into the default graph, while the negative delta is loaded as a named graph. The query is separated into its BGPs, which are `MINUS` operated with each triple inside a `GRAPH` statement, aiming to remove triples contained in the negative delta, since those represent data that is no longer available in the target version. This follows the notion that $G_{target} = (G_{base} \cup \Delta^{+}(G_{base}, G_{target})) \setminus \Delta^{-}(G_{base}, G_{target})$. An example of a Single Version query using absolute deltas can be seen in Code 4.6.

In Version Delta queries, the target version portion of the query is constructed in the same manner as a single version query, while the preceding version must be built using `GRAPH`, `UNION` and `MINUS` statements for each triple (this is because the default graph, which can store the union of multiple graphs, has already been used for the current version). An example of a Version Delta query using absolute deltas can be seen in Code 4.7.

4.5.2. Expected advantages

Absolute deltas have simple query conversions, needing only the base versions and two other graphs with the positive and negative deltas. Accessing any non-base version has a similar cost across all of them, keeping query times consistent when varying the target version. If the first version is used as a base, adding a version to an already built dataset has a relative low cost, requiring only one new pair of deltas to be calculated for the version being added and indexed as new named graphs.

```
PREFIX ex: <http://example.org/>
SELECT ?platform ?date
FROM <http://example.org/v1>
FROM <http://example.org/deltas/v3-v1>
FROM NAMED <http://example.org/deltas/v1-v3>
WHERE {
   SELECT ?platform ?date
   WHERE {
      {  {  ex:SuperMetroid ex:release ?r  .
            ?r  ex:date ?date  .
            ?r  ex:platform ?platform  .
            MINUS
            {  GRAPH <http://example.org/deltas/v1-v3>
               {  ex:SuperMetroid ex:release ?r }
            }
         }
         MINUS
         {  GRAPH <http://example.org/deltas/v1-v3>
            { ?r  ex:date ?date }
         }
      }
      MINUS
      {  GRAPH <http://example.org/deltas/v1-v3>
         { ?r  ex:platform ?platform }
      }
   }
}
```

Code 4.6: Single Version query (Q_3) using absolute deltas

```
PREFIX ex: <http://example.org/>
SELECT ?platform ?date
FROM <http://example.org/v1>
FROM <http://example.org/deltas/v3-v1>
FROM NAMED <http://example.org/deltas/v1-v3>
FROM NAMED <http://example.org/v1>
FROM NAMED <http://example.org/deltas/v2-v1>
FROM NAMED <http://example.org/deltas/v1-v2>
WHERE {
  {  SELECT ?platform ?date
     WHERE {
        {  {  ex:SuperMetroid ex:release ?r  .
              ?r  ex:date ?date  .
              ?r  ex:platform ?platform  .
              MINUS
              {  GRAPH <http://example.org/deltas/v1-v3>
                 {  ex:SuperMetroid ex:release ?r  }  }  }
           MINUS
           {  GRAPH <http://example.org/deltas/v1-v3>
              { ?r  ex:date ?date } }  }
        MINUS
        {  GRAPH <http://example.org/deltas/v1-v3>
           { ?r  ex:platform ?platform }  }  }  }
  MINUS
  {  SELECT ?platform ?date
     WHERE {
        {  { GRAPH <http://example.org/v1>
              { ex:SuperMetroid ex:release ?r }   }
           UNION
           { GRAPH <http://example.org/deltas/v2-v1>
              { ex:SuperMetroid ex:release ?r }   }
           MINUS
           { GRAPH <http://example.org/deltas/v1-v2>
              { ex:SuperMetroid ex:release ?r }   }  }
        {  { GRAPH <http://example.org/v1>
              { ?r  ex:date ?date } }
           UNION
           { GRAPH <http://example.org/deltas/v2-v1>
              { ?r  ex:date ?date } }
           MINUS
           { GRAPH <http://example.org/deltas/v1-v2>
              { ?r  ex:date ?date } }  }
        {  { GRAPH <http://example.org/v1>
              { ?r  ex:platform ?platform }  }
           UNION
           { GRAPH <http://example.org/deltas/v2-v1>
              { ?r  ex:platform ?platform }  }
           MINUS
           { GRAPH <http://example.org/deltas/v1-v2>
              { ?r  ex:platform ?platform }  }  }  }  }  }
```

Code 4.7: Version Delta query (Q_3^*) using absolute deltas

4.5.3. Expected disadvantages

The size of the deltas is expected to grow with the number of versions, since each of them has all the accumulated changes since the beginning. This may lead to a larger dataset overall when compared to other representation alternatives. If the last version is used as a base, adding a version to an already built dataset has the highest possible cost, requiring all deltas to be calculated again, essentially rebuilding the dataset from scratch.

4.6. Sequential deltas

In this representation, a version is used as a base and kept in its own graph. Deltas are calculated between each pair of consecutive versions. As per the absolute deltas, the first or latest version can be used as the base. This scheme is similar to *incremental backups* and *reverse incremental backups* sometimes used in database and archiving systems.

4.6.1. Construction details

4.6.1.1. Dataset construction

Similarly to absolute deltas, the base version is kept whole in its own graph. Deltas are calculated bewtween consecutive versions and kept in separate graphs. Given the large number of deltas that need to be accessed for versions distant to the base, a different query rewriting methodology was used. As such, it was neccesary to add metadata detailing the version and direction of the delta graphs, as can be seen in Code 4.8. Direction is measured regarding the latest version depending on whether the delta is positive ("forwards") or negative ("backwards"). Another requirement for the dataset is that the version labels must follow an ascending lexicographical order. For the running example, the first version will be used for the base. The versioned dataset for the running example using sequential deltas can be seen in Figure 4.4.

```
@prefix ex: <http://example.org//>
@prefix delta: <http://example.org/deltas/>
ex:v1 delta:direction "forwards" .
ex:v1 delta:version "v1" .
delta:v2-v1 delta:direction "forwards" .
delta:v2-v1 delta:version "v2" .
delta:v1-v2 delta:direcion "backwards" .
delta:v1-v2 delta:version "v2" .
delta:v3-v1 delta:direction "forwards" .
delta:v3-v1 delta:version "v3" .
delta:v1-v3 delta:direcion "backwards" .
delta:v1-v3 delta:version "v3" .
delta:v4-v1 delta:direction "forwards" .
delta:v4-v1 delta:version "v4" .
delta:v1-v4 delta:direcion "backwards" .
delta:v1-v4 delta:version "v4" .
```

Code 4.8: RDF triples that add sequential delta graph metadata

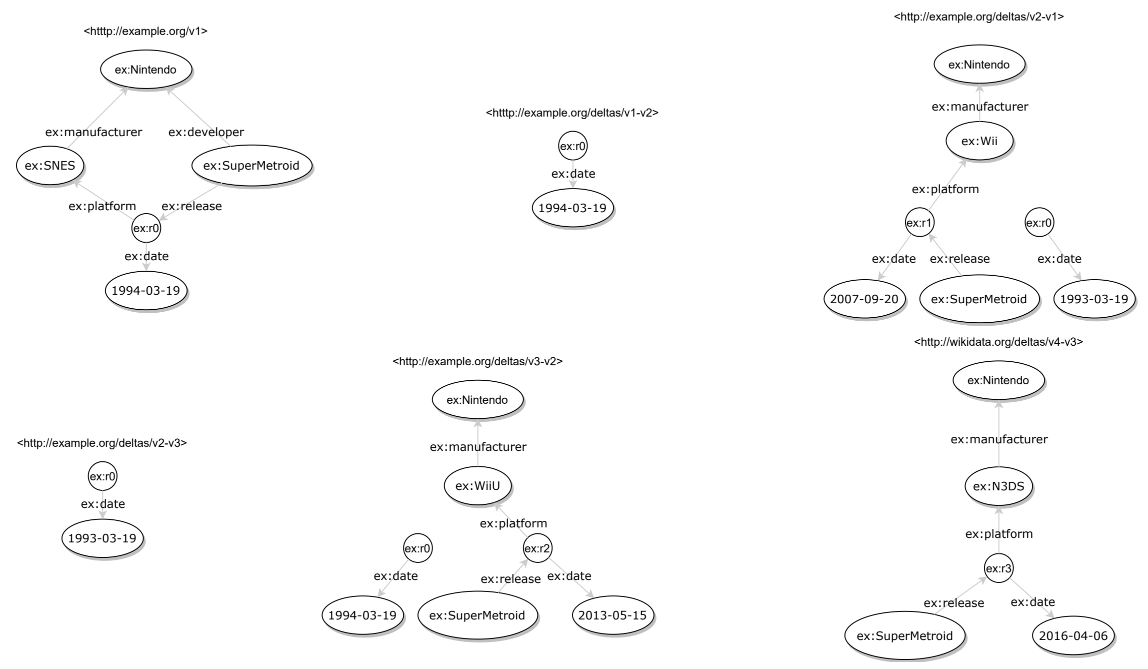


Figure 4.4: Sequential delta graphs representing the running example graph and its versions

4.6.1.2. Query conversion

Graphs are implicitly searched using a `GRAPH` statement with a variable and then filtered according to the direction and version necessary. In more detail, the maximum positive (or "forwards") version containing the triple is searched. Then, the maximum negative (or "backwards") version containing the triple is searched. If the negative version is lexicographically larger (a latter version), the triple is discarded. Otherwise, it is included in the results. An example of a Single Version query can be seen in Code 4.9. In the case of Version Delta queries, both the target version and preceding version queries are built individually and then their results compared and subtracted. This corresponds to a limitation due to the way queries were constructed. Since queries are so lengthy in characters, some query engines fail to accept the query at runtime.

```
PREFIX ex: <http://example.org/>
PREFIX delta: <http://example.org/deltas/>
SELECT  ?platform ?date
WHERE
  { { { SELECT  ?r (str(MAX(?v0_)) AS ?max_v0)
        WHERE
          { { { GRAPH ?g0  { ex:SuperMetroid ex:release  ?r }
                ?g0  delta:direction  "forwards" .
                ?g0  delta:version  ?v0 }
              BIND(str(?v0) AS ?v0_) }
            FILTER ( str(?v0) <= "v3" ) }
        GROUP BY ?r }
      OPTIONAL
        { { SELECT  ?r (str(MAX(?v1_)) AS ?max_v1)
            WHERE
              { { { GRAPH ?g1 { ex:SuperMetroid ex:release  ?r }
                    ?g1  delta:direction  "backwards" .
                    ?g1  delta:version  ?v1 }
                  BIND(str(?v1) AS ?v1_) }
                FILTER ( str(?v1) <= "v3" ) }
            GROUP BY ?r }
          FILTER ( ( ! bound(?max_v1) ) || ( str(?max_v0) > str(?max_v1) ) ) } }
    { { SELECT  ?r ?date (str(MAX(?v2_)) AS ?max_v2)
        WHERE
          { { { GRAPH ?g2  { ?r  ex:date  ?date }
                ?g2  delta:direction  "forwards" .
                ?g2  delta:version  ?v2 }
              BIND(str(?v2) AS ?v2_) }
            FILTER ( str(?v2) <= "v3" ) }
        GROUP BY ?r ?date }
      OPTIONAL
        { { SELECT  ?r ?date (str(MAX(?v3_)) AS ?max_v3)
            WHERE
              { { { GRAPH ?g3 { ?r  ex:date  ?date }
                    ?g3  delta:direction  "backwards" .
                    ?g3  delta:version  ?v3 }
                  BIND(str(?v3) AS ?v3_) }
                FILTER ( str(?v3) <= "v3" ) }
            GROUP BY ?r ?date }
          FILTER ( ( ! bound(?max_v3) ) || ( str(?max_v2) > str(?max_v3) ) ) } }
    { { SELECT  ?r ?platform (str(MAX(?v4_)) AS ?max_v4)
        WHERE
          { { { GRAPH ?g4  { ?r  ex:platform  ?platform }
                ?g4  delta:direction  "forwards" .
                ?g4  delta:version  ?v4 }
              BIND(str(?v4) AS ?v4_) }
            FILTER ( str(?v4) <= "v3" ) }
        GROUP BY ?r ?platform }
      OPTIONAL
        { { SELECT  ?r ?platform (str(MAX(?v5_)) AS ?max_v5)
            WHERE
              { { { GRAPH ?g5 { ?r  ex:platform  ?platform }
                    ?g5  delta:direction  "backwards" .
                    ?g5  delta:version  ?v5 }
                  BIND(str(?v5) AS ?v5_) }
                FILTER ( str(?v5) <= "v3" ) }
            GROUP BY ?r ?platform }
          FILTER ( ( ! bound(?max_v5) ) || ( str(?max_v4) > str(?max_v5) ) ) } } }
```

Code 4.9: Single Version query (Q_3) using sequential deltas

4.6.2. Expected advantages

Sequential deltas, unlike their absolute counterpart, will typically mantain a consistent size throughout, assuming a consistent number of changes between versions. This consistent size helps the representation keep an optimal storage size, expected to be smaller than when absolute deltas are used. Adding a new version to an already built dataset has a relative low cost assuming the deltas are built in the forwards direction, requiring only one new pair of deltas to be calculated for the new version added, even if it replaces the current base.

4.6.3. Expected disadvantages

Queries using sequential deltas are, in general, more complex, needing to access several deltas per target version. When querying over the version farthest from the base, every named graph in the dataset must be accessed, impacting query times negatively. Since queries about recent data are expected to be more common, using the first version as the base risks facing this issue more often.

4.7. Complex queries

So far we have only discussed query rewritting for Basic Graph Patterns for the sake of both simplicity and brevity when it comes to the examples given. It should be noted, however, that most other graph patterns and operations between them – such as `OPTIONAL`, `UNION`, `MINUS`, etc. – can be supported by simply rewriting the BGPs that constitute them and joining each part of the rewritten query with the same operations that joined them in the original query. As discussed in the previous chapter, the main exception to this corresponds to property paths.

Simple property paths (/, ^, ?, and |) can be converted into Basic Graph Patterns, which is used in order to offer support for them. Arbitrary length (or complex) paths (such as $*$

and +) lack this property, however. Since they cannot be applied over several graphs, complex property paths present an issue for representations that use one or more named graphs alongside the default graph. Some of the possible paths may be split across several graphs while building the Versioned Dataset and cannot be restored at query runtime. Versioning alternatives that do not suffer this limitation are as follows: the Baseline representation for both Single Version queries and Version Delta queries, since each version is completely contained in a single graph and paths of arbitrary length are preserved; while the interval-based representation supports property paths in Single Version queries, since all the relevant graphs are merged into the default graph at execution, and arbitrary length paths are restored. Any other representation cannot offer support for complex property paths, including interval-based graphs when it comes to Version Delta queries.

4.8. Implementation

The system used to process and rewrite queries was built using the Jena ARQ[1] API for Java. The `Transformer` framework, along with implementations of the `Transform` interface were used in the query conversion process for each versioned dataset representation. All of the relevant code can be found on GitHub[2].

[1] https://jena.apache.org/documentation/query/
[2] https://github.com/HunterNacho/sparql-query-versioning

Chapter 5

Evaluation

In this chapter we discuss the construction and execution of the experiments considered for this work, as well as the results obtained.

5.1. Experimental goals

The experiments and measurements presented in this chapter have the goal of answering the research questions proposed by the hypothesis in Section 1.4. A contrast between the expected results and the ones obtained experimentally is also included. The questions relevant to the experimental section of this work are as follows:

1. What representations of a versioned dataset allow for better compression, more efficient indexing, and/or more efficient updates?

2. How large is the performance cost of implementing a versioned dataset and query system?

3. Which of the representation alternatives will have the most well-rounded performance?

How these questions will be addressed by the experiments is discussed in the following sections. It is expected that using the same dataset and query set in each representation alternative will give us enough insight to answer them.

5.2. Experimental settings

5.2.1. System specifications

Experiments are run in a Linux virtual machine with the following specifications:

- 4TB hard drive space

- 120GB of R.A.M.

A Virtuoso[1] Server was deployed in said virtual machine, making a different instance for each versioning representation and QID threshold. The relevant RDF graphs were uploaded

[1] http://vos.openlinksw.com/owiki/wiki/VOS

to each instance using Virtuoso's bulk loader, making sure to register how much time the upload and indexing took.

5.2.2. Dataset and representations

The data used for the experiments consists of weekly Wikidata "dumps" from August and September of 2017. In total, 8 dumps are used, each containing over 1500 million triples. These dumps are used as versions of the same graph for the construction of each version dataset alternative. The amount of triples contained in each version, along with the amount added and removed can be seen in Table 5.1. In order to assess how the size of the graph affects query runtimes, additional datasets are constructed by limiting the amount of QIDs allowed (see Section 2.2). Since the largest QIDs within the dumps used are around 40 million, the following thresholds are built: 1 million, 2 million, 4 million, 8 million, 16 million, 32 million, and full dataset. These will be referred to as 1M, 2M, 4M, 8M, 16M, 32M, and full for brevity. All representations discussed in the previous chapter are used, making a total of six Versioned Datasets:

- Complete versions as the "baseline"
- Interval-based graphs (or simply "intervals")
- Absolute deltas using the first version as the base ("absolute-0")
- Absolute deltas using the latest version as the base ("absolute-n")
- Sequential deltas using the first version as the base ("sequential-0")
- Sequential deltas using the latest version as the base ("sequential-n")

Version	Total triples	Triples added	Triples removed
2017-08-09	1,505,807,223	–	–
2017-08-16	1,541,942,580	42,630,489	6,494,985
2017-08-23	1,592,997,542	56,681,292	5,626,215
2017-08-30	1,613,490,986	25,306,043	4,811,224
2017-09-07	1,684,476,127	75,166,341	4,180,870
2017-09-13	1,771,301,517	93,039,159	6,213,763
2017-09-20	1,842,785,338	77,087,119	5,603,102
2017-09-27	1,924,663,703	88,263,688	6,384,825

Table 5.1: Number of total triples, triples added, and triples removed in each version

5.2.3. Experiments

The experiments run are based on the Wikidata example queries: a set consisting of over 300 SPARQL queries contributed by various volunteers with many different levels of complexity. Several of these queries, however, had to be removed due to incompatibilities related to the use of external services on the Wikidata query engine, as well as the use of property paths not supported by all representation alternatives. The resulting test set is comprised of 146 SPARQL queries.

Each query is converted into both Single Version and Version Delta queries. In the case of Single Version queries, target versions 1, 5 and 8 are used. For Version Delta queries, target versions 2, 5 and 8 are used, since target version 1 would be equivalent to a Single Version query. The different target versions were chosen to present the best case, worst case and average case of delta-based representations. The baseline and interval-based representations should not be greatly affected by the target version, however. Every converted query is run three times in order to reduce any variance in their execution times.

Others factors considered for each representation are their respective construction times, storage size and the impact adding a new version has. Moreover, using the QID thresholds, it becomes possible to observe how the size of a versioned graph affects each individual versioned dataset representation.

5.3. Experimental results

5.3.1. Dataset size

In order to contextualize some of the results in the following sections, we present the impact of the QID thresholds when it comes to dataset size. As we can see in Figure 5.1, the growth in dataset size (expressed in total number of triples) is consistently linear for most representation alternatives. The only exception to this comes from both absolute delta representations, since they lose compression efficiency for larger graphs.

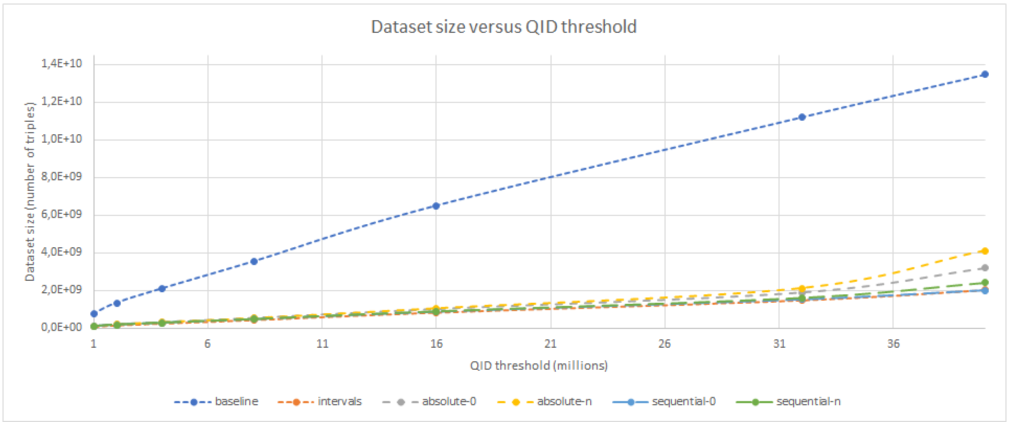


Figure 5.1: Graph showing dataset size (in triples) for each QID threshold

5.3.2. Storage size

Storage size corresponds to one of the easiest parameters to analyze, but that does not diminish its importance. In Figure 5.2 we can see a graph that visualizes the size on disk of each dataset. As expected, the baseline dataset is the largest by far, doubling the size of other representations at most scales. Sequential deltas are better at compression than absolute deltas, with intervals being between them in terms of size. The results observed for this parameter line up with the preliminary expectations. An interesting note to add is that deltas based on the first version have a better compression overall than deltas based on the latest version. This may be due to the growth of subsequent versions, making deltas slightly larger.

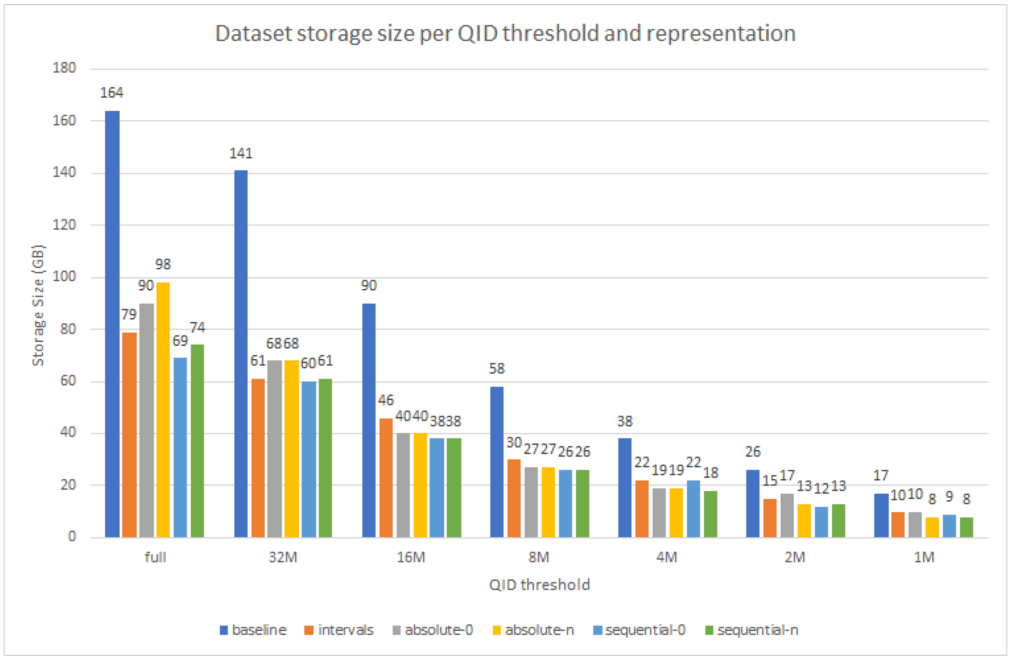


Figure 5.2: Graph showing the storage size for all representation alternatives

It is natural to assume that storage size is directly linked to dataset size, and in Figure 5.3 we can see that the relationship between these two parameters is mostly linear.

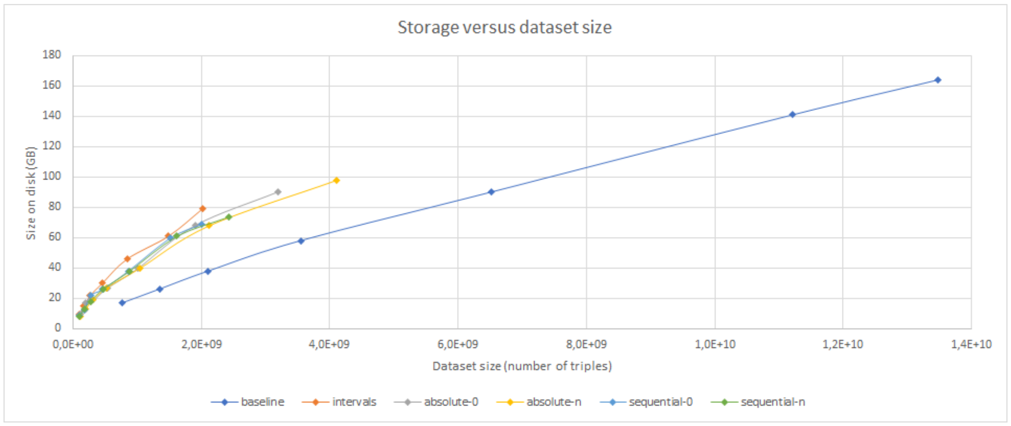


Figure 5.3: Graph showing the relationship between dataset size and size on disk for each representation

5.3.3. Construction times

Another relevant parameter to analyze corresponds to the time each dataset takes to be uploaded and indexed when it is being built for the first time. This measurement assumes starting with all versions of the graph as dumps, so the time necessary to calculate intervals and deltas is included in the representations that need them. Since the constrution of intervals and deltas took place before the QID filtering, the time spent building these additional graphs for the full version was taken as a reference and kept fixed. Using this data, some estimations can be made about the time it would take to add a new version in each representation in the case of an already built dataset. Figure 5.4 shows the amount of time each represetation takes to build and index, while Figure 5.5 shows an estimation of the time each representation would take to add a new version. In the case of absolute deltas with the latest version as a base, each time a new version is added, the dataset must be built nearly from scratch. For the interval-based representation, the estimated time assumes the worst case where the most amount of graphs must be altered and indexed again.

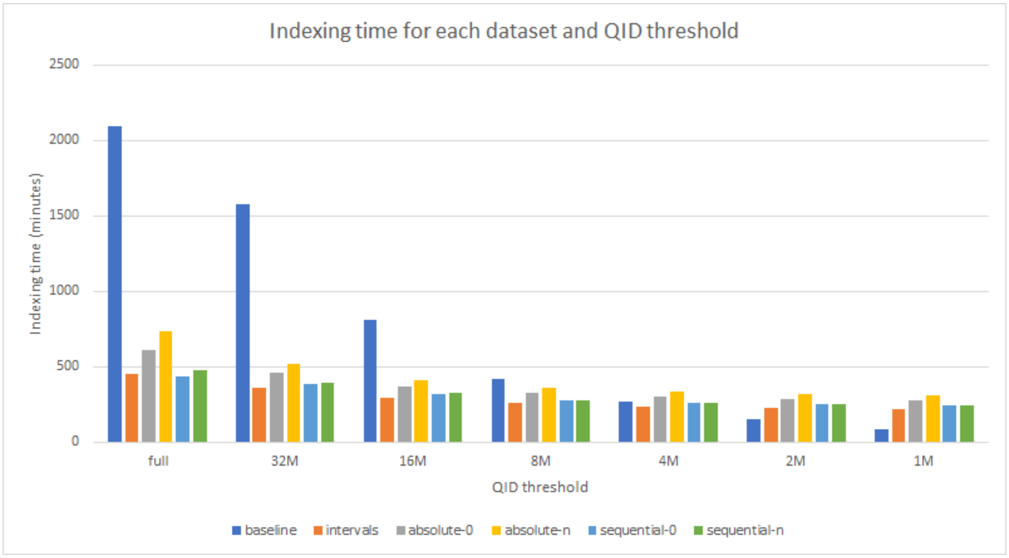


Figure 5.4: Graph showing the time taken by each representation to build and index initially for each QID threshold

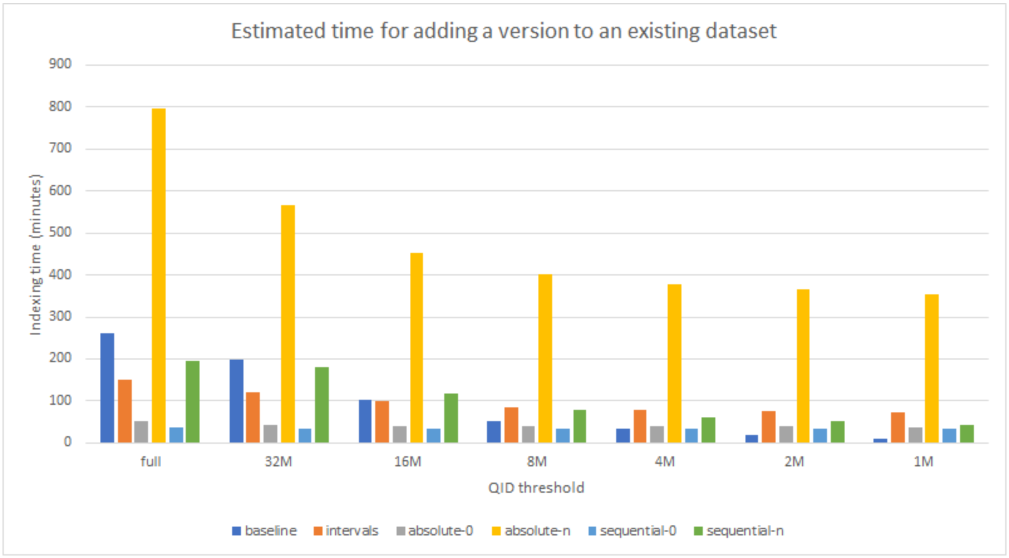


Figure 5.5: Graph showing the estimated time taken by each representation to add a new version for each QID threshold

As can be seen, the baseline has the largest times overall due to having to index a larger amount of data. The rest of the representations take an amount of time proportional to their storage size, with absolute-n being the highest besides the baseline. For the lower QID thresholds, the fixed time of building intervals and deltas is larger than the upload and indexing times, making them very similar.

When it comes to the cost of adding a new version, absolute-n has the highest. This occurs due to the fact that adding a new version replaces the base being used, which invalidates the existing deltas and requires the entire dataset to be rebuilt. A similar issue arises in sequential-n, but the impact is much lower since the deltas already calculated are still relevant. The sequential-0 representation times may be higher in practice than what is shown, due to the extra operations required to generate the new delta tuple $\Delta(G_n, G_{n+1})$. Since G_n is not available as a single graph, it must be simulated using G_0 and the delta tuple $\Delta(G_0, G_n)$. As mentioned earlier, the times for the intervals representation could be improved upon with the implementation of open-ended intervals.

5.3.4. Query times

The largest amount of experiments comes from the queries run in each representation. Since each query is converted into both Single Version and Version Delta queries, with each of them built for three different target versions, six different versions of the same query are run in each representation alternative. A total of 146 queries are kept from the initial set, since a large amount of them presented issues for one or several reasons. The timeout for execution was set at five minutes (300 seconds). If the execution time of a query exceeds this number, it is halted and considered to have taken said amount of time. This is done in order to prevent excessively long query executions, as well as an attempt to simulate some amount of responsivity in the system. Since the execution times vary greatly with query complexity, the graphs display times in a logarithmic scale. For graphs showing all query executions for the same type of query (Single Version or Version Delta) for all representations, times are not grouped by query.In these graphs, the executions of each representation are sorted individually and then grouped by their overall percentile rank. This allows to better compare the representations at the cost of losing direct query-to-query comparison. Additional graphs are built to show how the performance of a representation varies with the target version of queries.

5.3.4.1. Single Version queries

Since Single Version queries are generally simple in their conversions, the difference between query times is not expected to be drastic for the most part. The representations using sequential deltas, however, may have some variance in execution time depending on the target version of the query. In Figure 5.6 we can see the query times for Single Version queries using target version 1. Figure 5.7 has the same results for target version 5, while Figure 5.8 has the results for target version 8. Additional graphs showing the performace of each representation are included. Table 5.2 shows the amount of time-outs for Single Version queries. Finally, as a means to compare each alternative more directly, Figure 5.9 has the execution times for a specific query q016.

Representation	Target Version 1	Target Version 5	Target Version 8
baseline	1	1	1
intervals	1	2	1
absolute-0	1	1	1
absolute-n	1	2	1
sequential-0	17	18	18
sequential-n	17	18	12

Table 5.2: Number of timeouts per representation and target version for Single Version queries

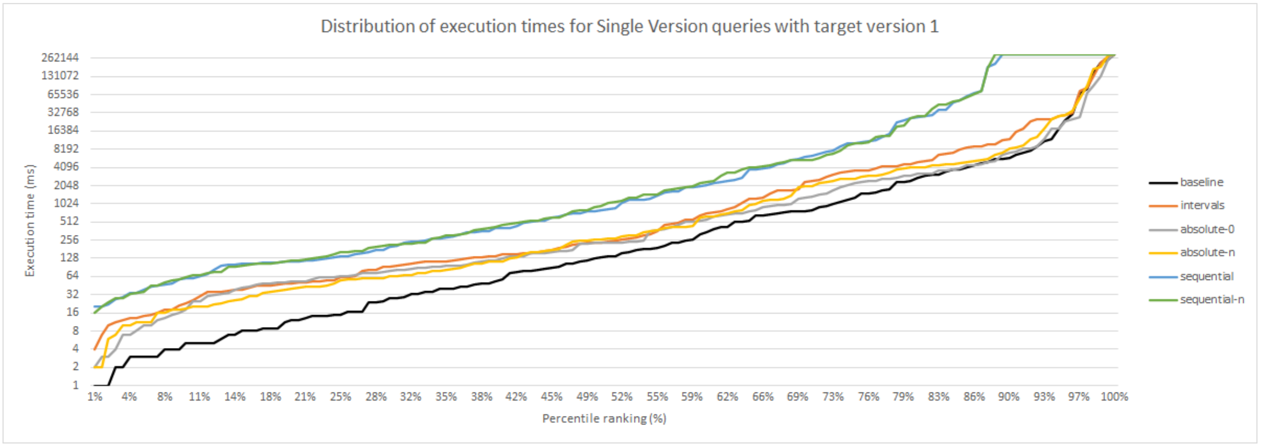


Figure 5.6: Graph showing the distribution of execution times for all Single Version queries with target version 1

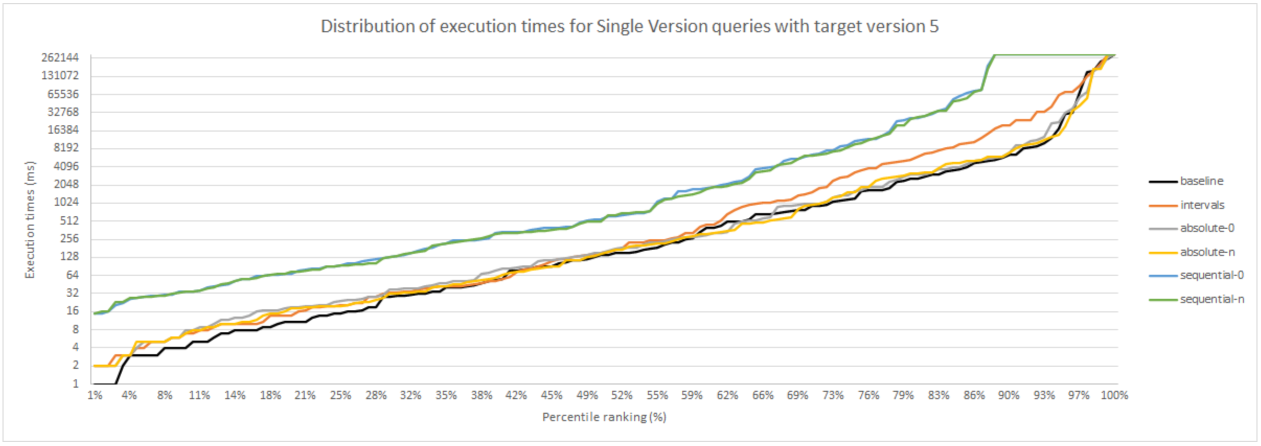


Figure 5.7: Graph showing the distribution of execution times for all Single Version queries with target version 5

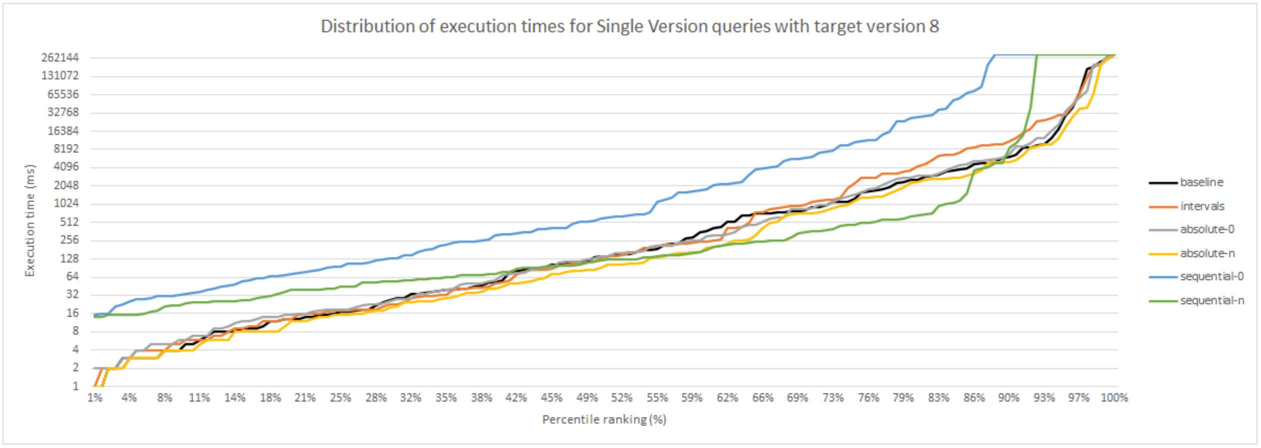


Figure 5.8: Graph showing the distrution of execution times for all Single Version queries with target version 8

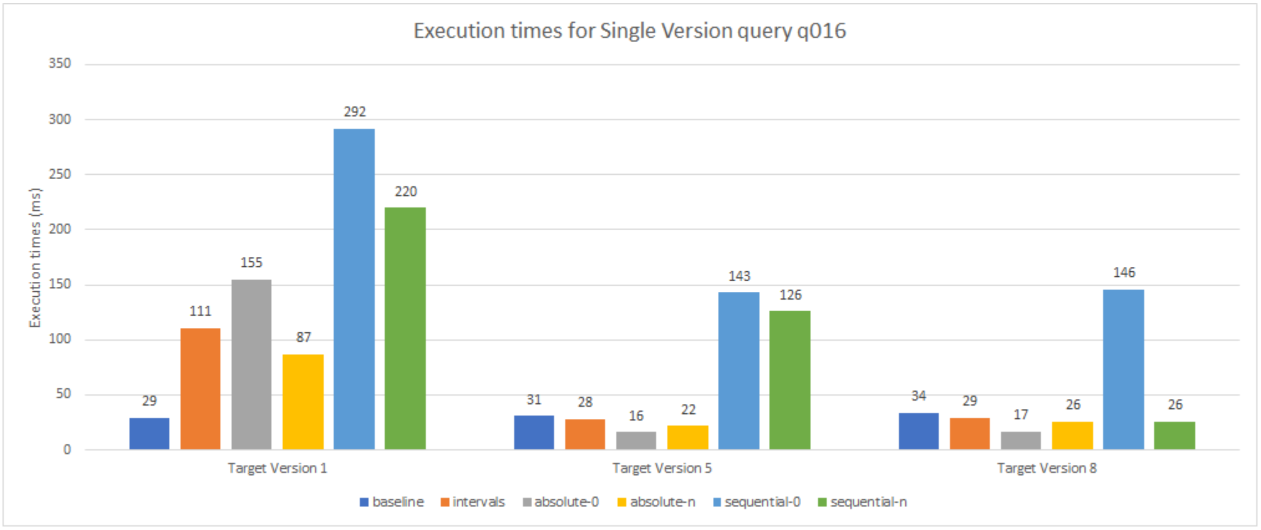


Figure 5.9: Graph showing execution times for query q016

As expected, the baseline representation has the lowest query times overall, save for some rare exceptions. Sequential delta representations take the largest execution times and amount of timeouts. However, it seems that the target version has a smaller impact than expected; only sequential-n performs consistently better when querying over the latest version, the remaining representations do not show as much variance. Absolute deltas and interval-based graphs have similar performances, with some exceptions where the interval-based representation times out.

5.3.4.2. Version Delta queries

Version Delta queries are generally more complex than their Single Version counterparts, making it easier to contrast between each representation alternative. As a consequence of their complexity, these queries may be harder for the query engine to optimize, causing more timeouts overall. In Figure 5.10 we can see the query times for Version Delta queries using target version 2. As with Single Version queries, the graph displays execution times in a logarithmic scale and sorts them by percentile ranking within each representation. Figure 5.11 has the same results for target version 5, while Figure 5.12 has the results for target version 8. Graphs showing the performance of each representation are also included. An important note to make is that the absolute delta representations failed to return results for most queries due to errors caused by the complexity of the queries generated (see Code 4.7 for an example). Also, even though Version Delta queries were separated into two queries for the sequential delta representations, their combined maximum query execution time is kept at 300 seconds and any time larger than that threshold is truncated. Table 5.3 shows the amount of timeouts for Version Delta queries. Same as for Single Version queries, Figure 5.13 shows the results for a specific query q016 in order to compare the results obtained in a more direct manner (note that this is one of the few queries that can be executed succesfully for absolute deltas).

Representation	Target Version 1	Target Version 5	Target Version 8
baseline	17	17	17
intervals	21	21	20
absolute-0	132	132	132
absolute-n	132	132	132
sequential-0	26	26	26
sequential-n	26	26	25

Table 5.3: Number of timeouts per representation and target version for Version Delta queries

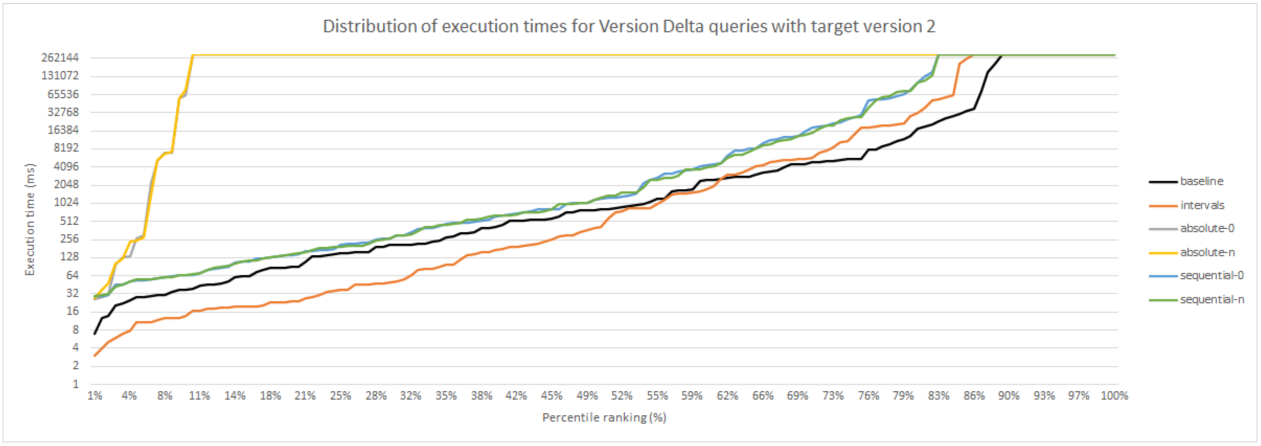


Figure 5.10: Graph showing weighted execution times for all Version Delta queries with target version 2

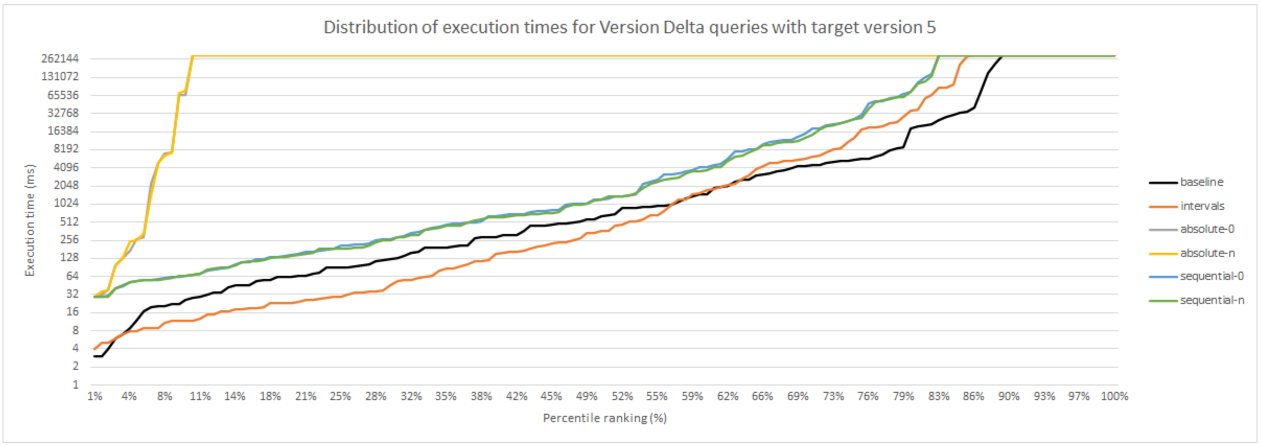


Figure 5.11: Graph showing weighted execution times for all Version Delta queries with target version 2

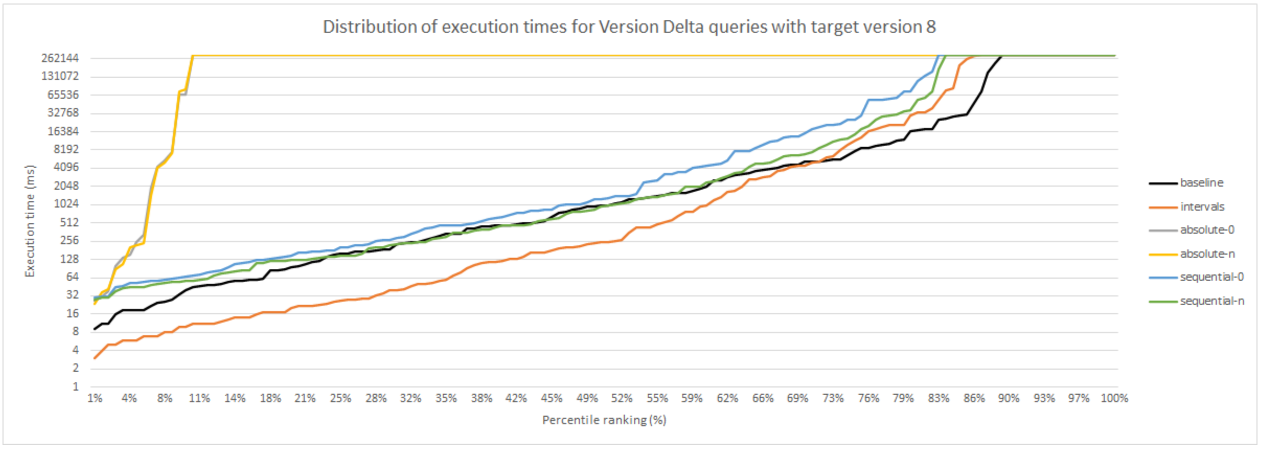


Figure 5.12: Graph showing weighted execution times for all Version Delta queries with target version 2

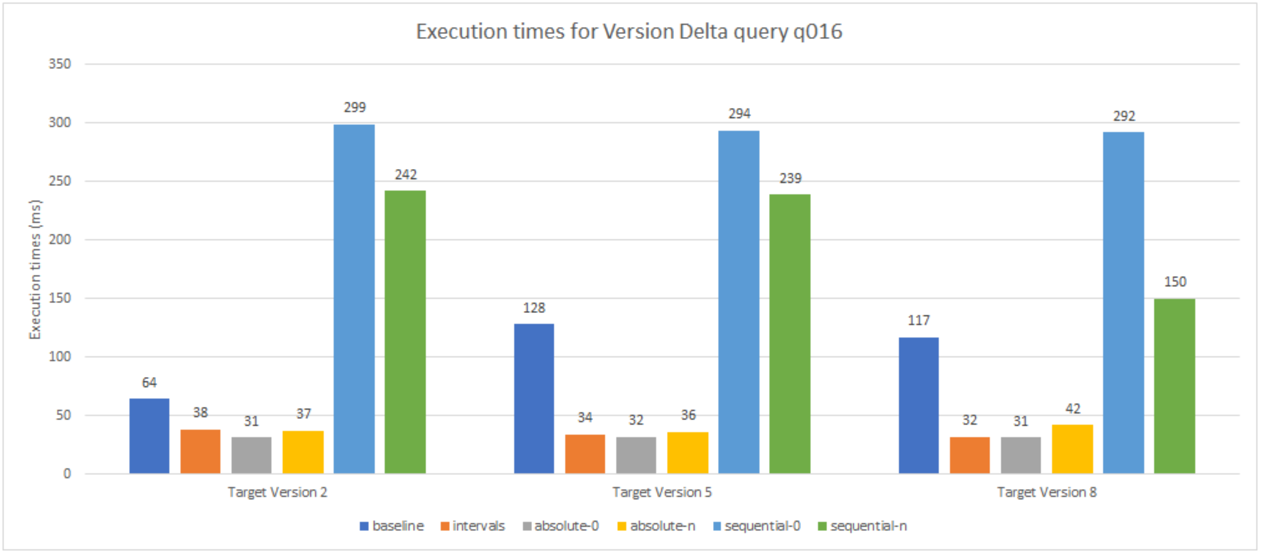


Figure 5.13: Graph showing execution times for query q016

Disregarding absolute deltas since they do not work for most of the queries, sequential deltas have the longest execution times overall. Intervals have a somewhat inconsistent performance, but are competitive in most cases with the baseline query times, managing to out-perform it in queries with lower complexity, but timing out more often in complex queries. As with Single Version queries, target version does not impact execution times significantly. Only sequential-n presents slightly lower execution times for target version 8.

5.4. Synthesis

All representations achieve a compression of around 50% when compared against the baseline, with sequential deltas being the best and absolute deltas the worst. Uploading and indexing times directly follow this pattern, since they are directly proportional to the storage size of each dataset. Times for adding a new version are the worst when it comes to deltas using the last version as a base, but significantly better using the first version. The interval-based representation is between them, only failing to hold up for smaller QID thresholds due to the high cost of buiding all the intervals from the Wikidata dumps.

When it comes to query execution times, sequential deltas perform the worst overall. Absolute deltas have issues with Version Delta queries but perform quite well with Single Version queries. The interval-based representation shows query times competitive with the baseline, while also being the most consistent overall. Target version for the queries does not impact queries as much as expected. Only the sequential-n representation show some amount of variance in execution time, favoring queries with target version 8, its base version.

Interval-based graphs is chosen as the best overall representation presented due to the following facts:

- Its indexing times are among the lowest of all representations and could be further improved with open intervals (see Figure 5.4).
- It offers partial support for property paths, which could be extended to full support using separate queries for Version Delta queries.
- It has the simplest query rewriting overall, since it does not modify the body of the query as much as the other representations (compare, for example, Code 4.5 with Code 4.9).

Chapter 6

Conclusions

6.1. Summary

In this work we have proposed and evaluated a system that allows for efficiently storing and querying versioned RDF graphs that track changes over time. This was done in order to help preserve temporal data in datasets that otherwise lose it whenever an update occurs. A few systems that achieve similar goals currently exist. These systems, however, employ extensions, specialized indices and other custom tools in their implementations. Considering this, the goal of building a system that achieves our objetive using only base RDF and SPARQL was added. This would serve as an "off-the-shelf" solution to the problem presented.

Several alternatives to represent the versioned data were tested, each requiring a different method of conversion for SPARQL queries in order to ensure compatibility with the versioned datasets created. A version without compression was kept to compare the results obtained. Alongside it, versioned representations based on intervals and deltas were built. Two types of deltas were defined: *absolute* deltas that take each version of the graph and compare it to the base version (either oldest or most recent), and *sequential* deltas, where each version is compared against the previous one to keep a "step by step" representation of the changes in the graph through time.

Overall, the interval-based representation had the most competitive results, with the exception of build times. These times could be easily be improved upon with small optimizations to the construction method. Moreover, the interval-based representation is the only one that partially supports property paths; the rest of the representations are fully incompatible with them as property paths cannot be evaluated across different named graphs.

6.2. Scope and limitations

While most of the techniques and methods developed can be used beyond what was shown in this work, acknowledging the limitations associated to its scope is important to analyze how this work could be expanded upon in the future.

6.2.1. Query engine

While Virtuoso served as an easy-to-deploy RDF server and SPARQL engine, it is not without its shortcomings and particularities. Virtuoso has issues supporting some valid queries or implements some operations in a non-standard way. Building a query conversion system with Virtuoso in mind may have imposed some limititations that are not easy to detect. Exploring other SPARQL engines, such as Apache Jena[1] would help remove any biases associated to working exclusively in Virtuoso, as well as prove that the results generalize to multiple settings.

6.2.2. Query set

The Wikidata example queries were the natural choice given the dataset used for this work. These queries however, are originally designed to be used on Wikidata's own query endpoint and contain some `SERVICE` statements that only work in that context. Since the queries had to be adjusted to work around that restriction, a few patterns are repeated through most of them. The `OPTIONAL` and `COALESCE` statements that extract labels for query results give queries a similar structure with a few exceptions. The few Version Delta queries that returned results for absolute delta representations are the ones that do no contain this query pattern. As such, we can state that the Wikidata example queries have an accidental bias, impacting absolute deltas in a way that causes them to fail. Using a different query set, such as the *BEAR* test suite presented in [5] may help avoid any related to the Wikidata sample queries and detect possible flaws in the query conversion process.

6.2.3. Dataset

As with the engine and query set, it would be useful to use the versioning representations with a dataset other than Wikidata. Testing the versioning representations in a different context would allow us to determine whether or not they are suitable for any dataset.

6.2.4. Versioning representations

The versioning representations presented cover many possible alternatives, but several others are possible. A different implementation of intervals where triples refer to their valid intervals could maximize compression while still allowing queries to be answered.

6.2.5. Property paths

The biggest shortcoming of this work is the lack of support for property paths. It would be useful to extend the versioning representations and query conversion system to allow any type of query, though it is not immediately clear how this might be done without either storing full versions in each graph, or extending the query language with additional features.

6.3. Future Work

Some of the points to be addressed if this work is to be continued are as follows:

- As it was previously mentioned, queries including complex property paths are not supported by this method of versioning. Therefore, a future version supporting all

[1] https://jena.apache.org/

property path queries would be a possible next step. An important reminder, however, is that such system would have to take a very different approach for this to be possible. Since this is one the most prominent shortcomings of this work, it may be a good starting point.

- A small optimization for interval-based graph comes in the form of open-ended intervals. As dicussed in Section 4.4.3, having intervals defined as “valid until the last version” may help with the time it takes to add a new version to an already built dataset.

- An additional type of query that was not implemented in this work is a *Version Interval* query. This type of Temporal Query would return results valid within a range of versions, instead of a single one. While both the baseline and interval-based graphs representations could support these queries with minor modifications, delta-based representations would need a moderate amount of work and changes to be compatible with them.

- The experiments in this work assume the dataset is built for the first time. A proper method for adding versions to already built datasets was not implemented, and the time it would take to add a version to each representation was estimated from the data already collected. Implementing such a method may be very relevant if this work is to be expanded upon.

Bibliography

[1] Marcelo Arenas and Jorge Pérez. Querying semantic web data with sparql. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, pages 305–316, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0660-7. doi: 10.1145/1989284.1989312. URL http://doi.acm.org.uchile.idm.oclc.org/10.1145/1989284.1989312.

[2] Marcelo Arenas and Martín Ugarte. Designing a query language for rdf: Marrying open and closed worlds. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, pages 225–236, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4191-2. doi: 10.1145/2902251.2902298. URL http://doi.acm.org.uchile.idm.oclc.org/10.1145/2902251.2902298.

[3] Dan Brickley, R.V. Guha, and Brian McBride. RDF Schema 1.1. W3C Recommendation, February 2014. http://www.w3.org/TR/rdf-schema/.

[4] Renata Queiroz Dividino, Thomas Gottron, Ansgar Scherp, and Gerd Gröner. From Changes to Dynamics: Dynamics Analysis of Linked Open Data Sources. In *Workshop on Dataset PROFIling & fEderated Search for Linked Data*, volume 1151 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014.

[5] Javier D Fernández, Jürgen Umbrich, Axel Polleres, and Magnus Knuth. Evaluating query and storage strategies for rdf archives. *Semantic Web*, 10(2):247–291, 2019.

[6] Larry González and Aidan Hogan. Modelling dynamics in semantic web knowledge graphs with formal concept analysis. In *World Wide Web Conference*, pages 1175–1184. ACM, 2018.

[7] Fabio Grandi. T-SPARQL: A tsql2-like temporal query language for RDF. In *Local Proceedings of the Fourteenth East-European Conference on Advances in Databases and Information Systems*, pages 21–30. CEUR-WS.org, 2010. URL http://ceur-ws.org/Vol-639/021-grandi.pdf.

[8] Claudio Gutierrez, Carlos A. Hurtado, and Alejandro A. Vaisman. Introducing time into RDF. *IEEE Trans. Knowl. Data Eng.*, 19(2):207–218, 2007. doi: 10.1109/TKDE.2007.34. URL https://doi.org/10.1109/TKDE.2007.34.

[9] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation, March 2013. http://www.w3.org/TR/sparql11-query/.

[10] Stefan Heindorf, Martin Potthast, Benno Stein, and Gregor Engels. Vandalism Detection in Wikidata. In *Conference on Information and Knowledge Management*, pages 327–336. ACM, 2016.

[11] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer. W3C Recommendation, October 2009. http://www.w3.org/TR/owl2-primer/.

[12] Tobias Käfer, Ahmed Abdelrahman, Jürgen Umbrich, Patrick O'Byrne, and Aidan Hogan. Observing Linked Data Dynamics. In *European Semantic Web Conference (ESWC)*, pages 213–227. Springer, 2013.

[13] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015. doi: 10.3233/SW-140134. URL https://doi.org/10.3233/SW-140134.

[14] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45, 2009.

[15] Tim E. Putman, Sebastien Lelong, Sebastian Burgstaller-Muehlbacher, Andra Waagmeester, Colin Diesh, Nathan Dunn, Monica Munoz-Torres, Gregory S. Stupp, Chunlei Wu, Andrew I. Su, and Benjamin M. Good. WikiGenomes: an open web application for community consumption and curation of gene annotation data in Wikidata. *Database*, 2017, 03 2017. ISSN 1758-0463. doi: 10.1093/database/bax025. URL https://doi.org/10.1093/database/bax025. bax025.

[16] Guus Schreiber and Yves Raimond. RDF 1.1 Primer. W3C Working Group Note, June 2014. http://www.w3.org/TR/rdf11-primer/.

[17] Kostas Stefanidis, Ioannis Chrysakis, and Giorgos Flouris. On Designing Archiving Policies for Evolving RDF Datasets on the Web. In *Conceptual Modeling (ER)*, volume 8824 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 2014.

[18] Jonas Tappolet and Abraham Bernstein. Applied temporal RDF: efficient temporal querying of RDF data with SPARQL. In *Extended Semantic Web Conference (ESWC)*, pages 308–322, 2009. doi: 10.1007/978-3-642-02121-3_25. URL https://doi.org/10.1007/978-3-642-02121-3_25.

[19] Denny Vrandecic and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014. doi: 10.1145/2629489. URL http://doi.acm.org/10.1145/2629489.

[20] Antoine Zimmermann, Nuno Lopes, Axel Polleres, and Umberto Straccia. A general framework for representing, reasoning and querying with annotated semantic web data. *J. Web Sem.*, 11:72–95, 2012. doi: 10.1016/j.websem.2011.08.006. URL https://doi.org/10.1016/j.websem.2011.08.006.