**UNIVERSITY OF SOUTHAMPTON**
FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

# Using RDF Graph Provenance to Efficiently Propagate SPARQL Updates

by

Iman Naja

Thesis for the degree of Doctor of Philosophy

December 2019

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
School of Electronics and Computer Science

Doctor of Philosophy

USING RDF GRAPH PROVENANCE TO EFFICIENTLY PROPAGATE SPARQL UPDATES

by Iman Naja

On the Semantic Web, information is published as machine-readable graphs expressed as RDF triples. Information consumers may combine and repackage that information as derived graphs which are based on the originally published source graphs. In addition, the formal semantics of RDF and OWL permit inference, by which reasoners generate entailed graphs: derived graphs containing newly inferred information. The dynamic nature of information presents a challenge when dealing with derived or inferred information; if a source graph changes, any graphs that are derived from it must be updated in order to preserve their integrity. However, such recomputation of derived graphs can be expensive. This is analogous to the view update problem in databases, where changes in source data affect materialised views. Common approaches to this problem use the Delete and Re-Derive (DRed) algorithm to perform incremental view materialisation.

To minimise the resources needed to propagate source graph updates to derived and entailed graphs, we propose to use the provenance of those graphs to guide their recomputation. The provenance of a graph is the documentation of the history of that graph. Provenance is a key requirement in a range of Web applications, and to that end the W3C has endorsed the PROV data model and ontology for the representation of provenance on the Web as RDF graphs. However, provenance may be applied at different granularities, which has significant cost implications; a naïve application of DRed to the graph rederivation problem which individually tracked the provenance of the triples which comprise each graph would generate a provenance graph much larger than the original source graphs.

In this thesis, we present RGPROV, a light-weight extension to the PROV ontology for representing RDF graph creation and updates. RGPROV allows us to understand the dependencies that a derived graph has on its source graphs without the need to document the provenance of individual triples, and facilitates the propagation of graph updates to derived graphs. Additionally, we present a modification to the DRed algorithm that enables the efficient propagation of updates to entailed graphs. By making use of RGPROV, we enable partial updates to be made to the entailed graphs without the need for triple-level provenance, which reduces the need for complete recomputation but results in an identical entailed graph, while using fewer resources. In order to evaluate our approach, we developed a provenance-aware extension to and reimplementation of the EvoGen benchmark for evolving RDF graphs, itself based on the commonly-used LUBM benchmark for RDF storage and SPARQL query engines.

# Contents

# List of Figures

# List of Tables

## Declaration of Authorship

I, Iman Naja , declare that the thesis entitled *Using RDF Graph Provenance to Efficiently Propagate SPARQL Updates* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as:

  Naja I., Gibbins N. (2018) Using Provenance to Efficiently Propagate SPARQL Updates on RDF Source Graphs. In: Belhajjame K., Gehani A., Alper P. (eds) Provenance and Annotation of Data and Processes. IPAW 2018. Lecture Notes in Computer Science, vol 11017. Springer, Cham.

Signed:.......................................................................................................................

Date:..........................................................................................................................

# Acknowledgements

This dissertation would never have seen the light of day were it not for the compassion, endless patience, and unyielding support of my supervisor Dr. Nicholas Gibbins. My PhD journey was not typical, but the constant that I was able to rely on was that he believed in me and for that he has my eternal gratitude.

I am grateful to the School of Electronics and Computer Science at the University of Southampton for fully funding three years of my PhD studies.

I am very grateful for the support of my family. I would not be where I am today had my parents not instilled in me the importance of education and the drive to always improve myself. I will also forever be beholden to them for providing me the financial support that enabled me to finish my PhD studies.

My heartfelt thanks go to Peter Cockersell who has not only supported me and stood by me since 2012, but also encouraged me and coached me how to learn about myself and how to deal with adversities. I also want to express my deep gratitude to Alexandre Boudi Kanjo for standing by me and praising any and all of my achievements with the greatest of enthusiasm regardless of their significance or magnitude. Likewise, I want to thank Yasmin El-Helwe for just being there, for listening, and for being the sister whom I need to brighten my life with positivity and love.

I have received an abundance of encouragement and support from people at the University of Southampton. I am grateful to Dr. Adriane Chapman for her helpful feedback during and after my updgrade, especially on my conference presentation. I would also like to especially thank Eric Cooke, Harry Rose, Mudasser Alam, Maria Priestley, Oleksandr Pryymak, Belfrit Batlajery, Sofia Kitormili, Jarutas Pattanapanchai, Betty Purwandari, and Adriana Wilde. Further thanks go to everyone who was part of WAIS's Room4Writing group for the weekly gatherings, especially Dr. Sue White for founding the group. Lastly, I am also grateful to the staff of the Southampton University Student Union Advice and Information Centre (SUAIC) for all the valuable advise they gave me.

Additionally, I would like to thank Milan Markovic for helping me prepare for the viva. Finally, throughout my PhD I used a number of open-source software and systems which, in addition to being freely available, have made my life a little easier. I would be remiss not to acknowledge the developers of Dropbox, Miktex, TeXnicCenter, SumatraPDF, Eclipse, Protégé, f.lux, draw.io, and Notepad++.

*“The ones who love us never really leave us...”* [1]

***In loving memory of my dear grandfather***
***M.A. A.J. El-Helweh***
*(11 Feb 1922 - 19 Apr 2015)*

[1] J.K. Rowling, Harry Potter and the Prisoner of Azkaban

# Chapter 1

# Introduction

The Semantic Web, an extension to the World Wide Web, promotes the publishing, understanding, discovery, and integration of data (Berners-Lee et al., 2001), with recent years seeing a boost in the publication, inter-linkage, and consumption of large amounts of public datasets owing to government initiatives[1,2], the open access movement[3], and the open-access mandates[4]. It is an environment where masses of data are distributed and where data produced by some are consumed by others who in turn may produce even more data. Information on the Semantic Web is represented in machine-understandable formats, namely RDF and OWL, which provide well-defined meanings and support rules for reasoning on it, i.e. deriving new knowledge from it (Manola et al., 2014; Pascal et al., 2012). This notion of rich inference, is one of the central driving factors for the development of the Semantic Web.

However, this is not without challenge, as knowledge is neither static or complete and the change and evolution of data are inevitable, whether this manifests as new scientific discoveries, new political scenes, or simply in everyday life situations. This change needs to be incorporated and reflected in systems that make use of published data. Whilst reasoning anew on the updated data may be needed, it may be expensive, and sometimes impractical to re-obtain portions of the data that were used and to re-reason with it.

Another challenge arises in the fact that the Semantic Web is an open environment where 'anyone can say anything about anything'[5]. While this serves as a contribution to the significant growth of the availability and consumption of publicly available data over the recent years, it in turn leads to the need to provide a means to trust the data to be consumed, and trust in data is intrinsically linked to knowing its provenance.

[1] https://www.data.gov
[2] http://data.gov.uk
[3] https://doaj.org
[4] http://roarmap.eprints.org
[5] https://www.w3.org/TR/rdf-concepts/

Provenance describes the history of a datum or thing, physical or immaterial, and which activities, entities, and people were involved in how it came to be (Groth and Moreau, 2013). It has proven to be useful in a variety of domains, from e-science to databases to workflow systems, as developers, researchers, and users have been concerned for some time now in establishing trust in, promoting understanding of, providing accountability for, and facilitating reproducibility of outputs of intelligent systems. Moreover, there has been recent community-driven work on achieving an open provenance vision to track provenance beyond the scope of closed database and workflow systems. This has resulted in the PROV data model which has been endorsed as a W3C recommendation (Lebo et al., 2013).

Whilst its effectiveness in addressing the need to promote trust, explanations, understanding, and replication has made its incorporation into systems more widespread and has led to the more prevalent emergence of provenance-aware systems, provenance can also be employed for an additional benefit. It is increasingly being used in efficient recomputation when data change, as demonstrated in the Panda system (Ikeda et al., 2013) and the ReComp framework (Missier et al., 2016), and especially on the Semantic Web as shown in (Flouris et al., 2009) and (Avgoustaki et al., 2016).

## 1.1 Research Focus

On the Semantic Web, simple information is expressed using the graph-based data model RDF, with more complex information expressed in RDF-S and OWL. The building blocks of these graphs are triples of the form subject-predicate-object. Reasoning on these graphs results in the inferrence of new triples. A graph which is created as a result of reasoning on another graph is an *entailed* graph. Graphs may be individually created or they may be formed by combining information from other graphs. In accordance with the PROV terminology, a graph which is created using other graphs is a *derived* graph. All entailed graphs are derived graphs. Further, reasoning may be performed on such graphs which had relied on other source graphs for their creation. Entailed graphs need to be updated when those source graphs change.

Consider, as an illustration, the following case. An organisation, for example, the University of Southampton, publishes through its open data service[6], RDF graphs describing its internal structure. Interested parties consume these graphs by downloading and using them, perhaps along with other relevant RDF graphs from other organisations. When the University of Southampton goes through internal organisational re-structuring[7], its

[6]https://data.southampton.ac.uk/organisation.html?codes

[7]The University of Southampton has undergone re-organisation four times in the last decade. The School of Electronics and Computer Science is currently part of the Faculty of Engineering and Physical Sciences, but has in the past decade been part of the Faculty of Physical Sciences and Engineering, the Faculty of Physical and Applied Sciences, and the Faculty of Engineering, Science, and Mathematics.

published RDF graphs change. It may also be that an error is found and corrected in those graphs. Consumers of those graph who want their data to stay up to date need to reflect those updates.

To keep a derived graph up-to-date along with a changed source graph using a naïve approach, a system would need to recreate it from scratch. Alternatively, it may rely on an incremental maintenance algorithm as well as initially materialising the queries. Then, when changes need to be reflected, they are marked in the graph and the reasoner automatically makes the required modifications and then re-derives - much like RDFox (Motik et al., 2015). This, however, does not make use of provenance nor does it exploit its advantages. Additionally, systems which implement their own reasoners are few and far between. Otherwise, a provenance-aware system may make use of provenance to reflect changes. Most, however, record provenance on the triple level, such as in (Green et al., 2007a; Flouris et al., 2009; Avgoustaki et al., 2016), which we argue is not always feasible or scalable, especially if a system were to use the W3C recommendation PROV. If PROV were used, recording each triple's provenance would result in a graph having the size of its provenance graph substantially larger than it. Even in the case where each triple's provenance consists of only `triple prov:wasDerivedFrom sourceTriple`, a graph's provenance graph would be a little larger than it. This is compounded when even slightly more provenance information is recorded, for example, documenting only who produced a triple using which process, would result in the graph's provenance graph being at minimum triple its size.

Hence, the undertaken work is directed towards addressing the following two research questions:

**Research Question 1:** How can the recording of provenance of a derived RDF graph (on the Semantic Web) enable its re-derivation when one of its source graphs changes?

**Research Question 2:** How can this provenance facilitate scalable partial re-derivation by generating less overhead in update communication and re-entailment?

From our research question, we can identify the following additional subsidiary research questions that elaborate on and support it:

- **SRQ1**: How can we capture more specific provenance of RDF graphs to facilitate its querying when the need arises to refer to this provenance and review the history of its making to pinpoint how it was created?
- **SRQ2**: What set of steps should be taken to partially update and re-reason on an RDF graph without having to create our own reasoner?

## 1.2 Thesis Contributions

Given the outlined thesis question, we provide the following contributions:

1. A vocabulary for Semantic Web graph provenance: While the PROV data model provides a means to capture and express the provenance of entities and how they change, it is generic. Hence, a more specialised vocabulary better served to express the specific provenance of graphs, relating their creation and detailing their changes. Accordingly, we extend PROV and present RGPROV, a vocabulary for RDF graph creation and update. RGPROV allows us to document the dependencies a derived graph, created by applying a set theoretic operation on two source graphs, has on said source graphs. It also facilitates the identification of which parts of insert or delete updates applied to those source graphs need to be applied to said derived graph.

2. An RDF graph partial re-derivation algorithm which implements the Delete and Rederive (DRed) algorithm (Gupta et al., 1993): The DRed algorithm deletes the asserted data and all the data that were derived from it, then re-inserts the subset of the derived data that can be re-inferred using other still present base data. We base our algorithm on it and tailor it to RDF graphs.

3. A model for efficient update propagation which uses RGPROV descriptions to inform the graph re-derivation algorithm above, with a prototype implementation as demonstrator.

4. A provenance-aware evaluation framework: In order to evaluate the model in point 3, we require arbitrary scaling of data, an ontology of moderate size and complexity, and graphs of different sizes ranging from small to very large. These are addressed by the Leigh University Benchmark (LUBM) (Guo et al., 2005). Additionally, we also require dynamic graphs, i.e. we require original graphs and updates applied on them in the forms of insertions and deletions. This is partially addressed by the evolving benchmark EvoGen (Meimaris and Papastefanatos, 2016), which is an extension of LUBM. Finally, we require graphs which contain effective instance links as generated by UOBM (Ma et al., 2006) to address this shortcoming in LUBM. As such, we designed an evaluation framework which extends EvoGen by re-implementing its described approach by including both insertions into and deletions from graphs, produces graphs with effective instance links, and is also provenance-aware.

The first three points were presented in the following peer-reviewed publication:

Naja I., Gibbins N. (2018) Using Provenance to Efficiently Propagate SPARQL Updates on RDF Source Graphs. In: Belhajjame K., Gehani A., Alper P. (eds) Provenance and

Annotation of Data and Processes. IPAW 2018. Lecture Notes in Computer Science, vol 11017. Springer, Cham.

## 1.3 Thesis Structure

The remainder of this thesis is organised as follows:

In Chapter 2, **Background and Related Work**, we provide the background literature relevant to our work. We start by introducing the Semantic Web. We then introduce provenance and list its benefits and usages, and follow that by presenting some previous work that has incorporated provenance in non-Semantic Web applications. Next, we make the case for the need for provenance on the Semantic Web and present some previous work that has incorporated provenance in Semantic Web applications. Finally, we survey the most prominent Semantic Web benchmarks, focusing on those which incorporate change.

In Chapter 3, **RGPROV: A Vocabulary for RDF Graph Provenance**, we examine the necessary steps for tracking the provenance of graphs on the Semantic Web and facilitating the propagation of their modification. We explore a running example where a graph is created using data from two other graphs and outline the implications of the change of one of those source graphs. Then, we present a specialisation of the PROV ontology, RGPROV, which models the classes and properties involved in a graph's creation and update.

In Chapter 4, **Application of RGPROV**, we showcase how the RGPROV vocabulary can be used by applying it to the running example presented in Chapter 3. We also delve into how an update on a source graph is propagated to a graph that uses it.

In Chapter 5, **Design and Implementation**, we present the system we have implemented that makes use of the RGPROV vocabulary. We do so by outlining the design of the system and describing the different components that make it up and expand on each of them.

In Chapter 6, **Evaluation Framework**, we describe the provenance-aware evaluation framework we have developed, which extends EvoGen and UOBM and produces evolving RDF graphs which increase and decrease in size.

In Chapter 7, **Evaluation and Discussion**, we present the datasets that were generated using the evaluation framework described in 6 to test our approach. Then we describe our evaluation criteria. Finally we present our experimental results showing that partial re-derivation based on select parts of the update performed on a source graph produces an identical entailed graph using less resources.

Finally, in Chapter 8, **Conclusions and Future Work**, we summarise this thesis and present our conclusions. Additionally, we present future extensions applicable to our work.

# Chapter 2

# Background and Related Work

In this chapter we present the background literature relevant to our work. We start by introducing the Semantic Web and making the case for it. We then present reason maintenance and materialisation. Next, we introduce provenance, list its benefits and usages, and briefly describe some provenance vocabularies as well as the PROV model. We then make the case for the need for provenance on the Semantic Web and present some previous works that have incorporated it in Semantic Web applications. Afterwards, we present two types of provenance applications on the Semantic Web, the first addressing the capture of provenance of SPARQL queries and the second addressing the use of provenance during recomputation. Finally, we survey a few of the Semantic Web benchmarks.

## 2.1 The Semantic Web

The World Wide Web, where information is merely displayed by machines and only understandable by humans, has been being extended and evolved over the past decade to fulfil the vision of the Semantic Web in (Berners-Lee et al., 2001) where the published content has well-defined and formal meaning. Consequently, this content can be processed, understood, and manipulated by software agents despite originating from miscellaneous, decentralised, and heterogeneous resources. It follows that the Semantic Web is described as "a Web of actionable information", where symbols are semantically interpreted so their meanings are understood, terms are logically connected to establish interoperability, and information is then derived (Shadbolt et al., 2006). This in turn leads to better knowledge sharing and interaction and cooperation between computer programs, in addition to enhanced cooperation between machines and people.

Simplified, the goal of the Semantic Web has been to provide a language for expressing and sharing data with well-defined meanings and for rules for reasoning about them, thus

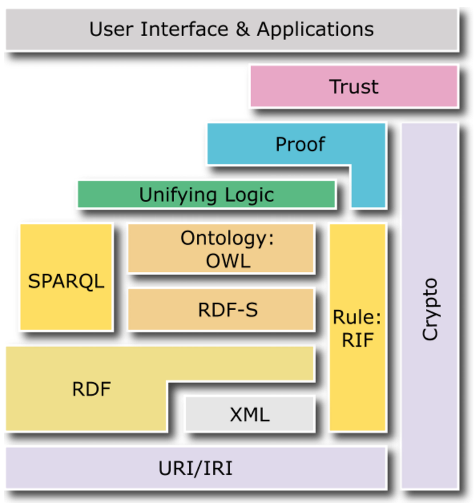


Figure 2.1: The Semantic Web Layer Cake (Bratt, 2007)

enabling their beneficial reuse, so that people and computers can work in cooperation (Berners-Lee et al., 2001).

### 2.1.1 The Case for the Semantic Web

The added value for the Semantic Web is that the documents are expressed in RDF and OWL and as such are machine-understandable. Hence, ontologies improve the functionality of the Web by presenting solutions[1] to some of its problems, such as terminology problems, Web search accuracy, and solving complicated questions whose answers span multiple Web pages (Berners-Lee et al., 2001). Additionally, and as previously mentioned, the goal of the Semantic Web has been to provide a language for expressing and sharing data with well-defined meanings and for expressing rules for reasoning about them, thus enabling their beneficial reuse, so that people and computers can work in cooperation. Thus, the Semantic Web is considered "a linked information space in which data is being enriched and added", and moreover, ontologies and data on the Semantic Web made available by organisations and people are to be discovered by other users and to be substantially reused (Shadbolt et al., 2006).

### 2.1.2 The Semantic Web Architecture

The Semantic Web is a made up of a collection of formats and languages which are standardised by the World Wide Web Consortium[2] (W3C). The architecture of the Semantic Web is illustrated through the Semantic Web Layer Cake, shown in Figure 2.1.

[1] A list of use cases and case studies, although severely outdated, can be found on https://www.w3.org/2001/sw/sweo/public/UseCases/slides/Slides.pdf

[2] https://www.w3.org

#### 2.1.2.1 URIs and IRIs

On the Semantic Web, things, be they concepts or concrete objects, are identified with URIs and IRIs. URIs and IRIs are links that are machine processable, have global scope, and are unambiguous and unique for each resource (Shadbolt et al., 2006; Bratt, 2007). Whereas URIs are limited to ASCII, IRIs extend URIs by allowing the usage of the Universal Character Set.

#### 2.1.2.2 RDF

On the Semantic Web, RDF is the *lingua franca* for formally expressing meaning and representing machine-processable documents (Berners-Lee et al., 2003, 2001). RDF is a graph-based data model that describes things and how they relate to other things. It does so by encoding meaning in sets of triples of the form subject-predicate-object. Each member of the triple is identified by an IRI (Manola et al., 2014; Hayes and Patel-Schneider, 2014). For example, the triple `ComputingMachineryAndIntelligence hasAuthor Turing` provides the information that the resource `ComputingMachineryAnd-Intelligence` has for an author the resource `Turing`. Hence, RDF triples represent information about resources in graph structures. Additionally, RDF allows triples from multiple sources to be combined into one graph. RDF may be serialised, or written, in different syntaxes, such as Turtle, JSON-LD, RDFa, RDF/XML, and Notation-3 (N3). Furthermore, anyone can define new concepts or properties by defining IRIs on the Web for them, and consequently, anyone can link to, refer to, or retrieve representations of them. Therefore, it is mandatory that each concept or property on the Semantic Web has a unique IRI as the IRIs ensure that concepts are not mere words in some document but that they are also tied to a unique definition on the Web, one which everyone can find.

An additional benefit of RDF is that it allows systems to perform reasoning, i.e. to make logical inferences. Inference is the deriving of new data from old ones. When some triples *follow logically* from another set of triples, we say that the latter *entail* the former (Manola et al., 2014). For example, the triple `ComputingMachineryAndIntelligence hasAuthor Turing` entails the triple `hasAuthor rdf:type rdf:Property`. The RDF entailment regime is summarised in Section 2.1.3.2.

#### 2.1.2.3 Ontologies

Often, the need arises for data from different sources to be combined or compared, and in many instances different terms must be recognised as having the same meaning or referring to the same concept or property (Berners-Lee et al., 2001). More importantly, greater expressiveness of objects and relation descriptions than that provided by RDF is required (Shadbolt et al., 2006). On the Semantic Web, this is accomplished using ontologies. An ontology is "an explicit specification of a conceptualization" (Gruber, 1993). In the context of the Semantic Web, an ontology is a document composed of formal, descriptive, and precise statements about concepts within a specific domain and the relations that exist among them (Pascal et al., 2012). Overall, ontologies have the following usages and capabilities: consistency checking, providing completion, interoperability support, validation support and verification testing, encoding test suites, supporting configuration, supporting customised, structured, and comparative search, and finally, exploiting generalisation/specialisation information (McGuinness, 2014).

RDF Schema (RDF-S) extends RDF to support the definition of class and property hierarchies as well as defining domain and range restrictions (Manola et al., 2014). For example, it allows declaring the resource `Article` as a sub-class of the resource `Publication` using the triple `Article rdfs:subClassOf Publication`. RDF-S has been widely adopted by the research community as "a minimal ontology representation language" (Shadbolt et al., 2006).

More complex ontologies are expressed using the Web Ontology Language - OWL 2[‡] (OWL Working Group, 2012). OWL 2 was designed to express formal meanings and to support the development of ontologies and their sharing on the Web. It provides three syntactic sub-languages, referred to as profiles, to address the different needs of applications by presenting trade-offs between reasoning efficiency and expressive powers (Motik et al., 2012a). Since ontologies may be distributed across systems with terms within them referring to other ontologies, OWL 2 is primarily exchanged using the RDF/XML syntax; hence, its ontologies are exchanged as RDF documents.

Ontology tools, such as reasoners, can verify the logical consistency of ontologies and automatically compute consequences. For example, given the two triples `ComputingMachineryAndIntelligence rdf:type Article` and `Article rdfs:subClassOf Publication`, a reasoner would infer the triple `ComputingMachineryAndIntelligence rdf:type Publication`. Therefore, reasoners are used to "query ontologies for implicit knowledge" so as to make it explicit (Pascal et al., 2012). Different entailment regimes are presented in some detail in Section 2.1.3.2.

#### 2.1.2.4 SPARQL Protocol and RDF Query Language

RDF content is stored in repositories called *RDF stores* and is queried and modified by systems and users using the RDF Query Language SPARQL. RDF graphs are stored in Graph Stores and also operated on using SPARQL. Examples of RDF stores include

[‡]OWL 2 extends and reviews OWL 1 thus making it the the newest version of OWL (OWL Working Group, 2012). It is also backward compatible with OWL 1.

Virtuoso[4], Ontotext GraphDB[5], MarkLogic[6], Neo4J[7] Cyclon, and Apache Jena Fuseki[8] TDB. The syntax of SPARQL follows the select-where-from SQL pattern. SPARQL 1.1 allows users and systems to formulate simple and complex queries using aggregation, filters, value expressions, negation, and nested queries.

**SPARQL Query Language** The SPARQL 1.1 Query Language has the following four query forms (Harris and Seaborne, 2013). (*1*) SELECT returns the matching values in a table format. (*2*) CONSTRUCT returns the matching values in an RDF format. (*3*) ASK returns a boolean that indicates if a result was found. Lastly, (*4*) DESCRIBE: returns an RDF graph which describes the results that were found, i.e. it returns the results and any other resources related to those results.

**SPARQL Update Language** The SPARQL 1.1 Update is an update language for RDF graphs (Gearon et al., 2013). It is used for both graph update operations and graph management operations.

Graph update operations do not create or delete graphs; they alter existing ones. There are five fundamental operations as follows.

1. Insert data, this results in one or more triples being added to a graph.
2. Delete data, this results in one or more triples being removed from a graph.
3. Delete/Insert, this is equivalent to a sequence of the above two operations.
4. Load, this results in inserting into a graph all the triples that are present in another graph. It may be treated it as being equivalent to applying an Insert operation for each triple in the other graph.
5. Clear, this results in removing all the triples that are present in a graph. As it does not require the subsequent removal of an empty graph (although some implementations may do so), it may be treated as being equivalent to applying a Delete operation on every triple in the graph.

Graph management operations work on graphs on a whole. There are five fundamental operations as follows. (*1*) CREATE creates a new graph. (*2*) DROP removes an existing graph along with all its content. (*3*) COPY deletes all content from a graph and inserts into it content from another graph. (*4*) MOVE deletes all content from a graph, inserts into it content from another graph, and drops the original graph. Lastly, (*5*) ADD copies all data from one graph and inserts them into another.

[4]http://vos.openlinksw.com/owiki/wiki/VOS/
[5]https://ontotext.com/products/graphdb/
[6]https://www.marklogic.com/
[7]https://neo4j.com/
[8]https://jena.apache.org/documentation/fuseki2/index.html

#### 2.1.2.5 Proof and Trust

In order for users to have confidence in the data that they intend to reuse, trust must be established, and as the content changes, users need to know how, where, when, and by whom the data originated. To verify who created or edited the data, digital signatures are employed. To show how data were produced, proofs are generated in chains of inference steps, with pointers including source and supporting materials. However, in order to address all the above concerns combined, users must know the provenance of the content. Specifically, Berners-Lee et al. (2006) state that provenance information is crucial in order to determine the integrity and value of a resource. We introduce provenance in Section 2.3.1, cover its usages and benefits in Section 2.3.2, and discuss the need for it on the Semantic Web in Section 2.5.1.

### 2.1.3 Graph Operations

We split the types of graph operations that result in the creation of a derived graph in three. The first is set theoretic, the second is entailment, and the third is related to SPARQL. We have already described the SPARQL operations in the previous section, therefore we describe set theoretic and entailment operations next.

Note that there may exist a special type of component in an rdf triple, the blank node. Blank nodes are used to indicate missing or insufficient information. A blank node, or bNode, is a node which is neither an IRI or a literal and which does not have an identifier. In an RDF triple, it may only be a subject or an object. When serialised, a bNode is given a locally scoped identifier, or blank node identifier, which is not portable outside the systems it is defined in (Klyne et al., 2014).

#### 2.1.3.1 Set Theoretic Operations

The set theoretic operations that can be performed on any two graph are: Union, Merging, Intersection, and Difference. Note that Hayes and Patel-Schneider (2014) only describe the first two operations, Union and Merging. The presence of blank nodes adds another layer of complexity when comparing graphs and triples. Without recourse to OWL reasoning using functional properties, inverse functional properties, or keys, blank nodes cannot be verified to be equal, and thus graphs cannot be proven to be isomorphic.

**2.1.3.1.1 Union** The union of two graphs is the set theoretic union of their sets of triples.

If blank nodes are present then their identifiers need to be studied. If any two blank nodes share an identifier, then this identifier needs to be changed, so as not to result in their

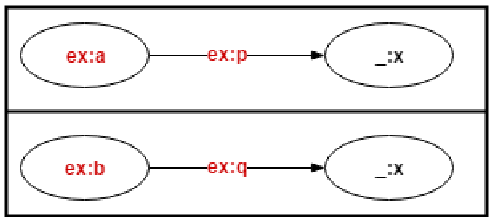


Figure 2.2: Two triples from two graphs (Hayes and Patel-Schneider, 2014).

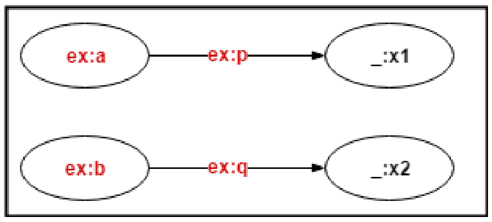


Figure 2.3: Union and Merging Result (Hayes and Patel-Schneider, 2014).

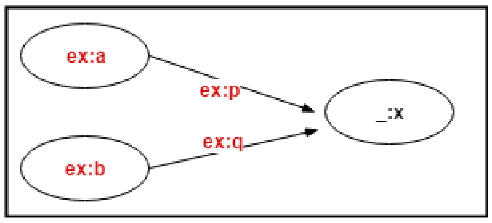


Figure 2.4: Concatenation-Union Result (Hayes and Patel-Schneider, 2014).

being inadvertently fused into a single node. For example, if one graph contains the triple `ex:a ex:p _:x` and another graph contains the triple `ex:b ex:q _:x`, as shown in Figure 2.2, then their union would differentiate the shared blank node and the resulting graph would contain the triples `ex:a ex:p _:x1` and `ex:b ex:q _:x2`, as shown in Figure 2.3. It would not be the graph containing the triples `ex:a ex:p _:x` and `ex:b ex:q _:x`, as shown in Figure 2.3, as that is considered concatenation (this is also the result of merging, discussed next).

**2.1.3.1.2 Merging** This operation is related to the union operation. The result of a merging, a graph called the merge, forces the divide of any shared blank nodes. In the case where two subgraphs of the same graph are merged, the size of the merge may be greater than that of the original graph. Hayes and Patel-Schneider (2014) showcase an example where two identical graphs with three nodes (Figure 2.4), are merged resulting in a merge containing four nodes (Figure 2.3).

**2.1.3.1.3 Intersection** The intersection of two graphs is the set theoretic intersection of their sets of triples.

Because blank nodes cannot be verified to be equal, the triples they are in are discarded.

**2.1.3.1.4 Difference** The difference of two graphs is the set theoretic difference of their sets of triples.

Because blank nodes cannot be verified to be equal, the triples they are in are included.

#### 2.1.3.2 Entailment

An entailment regime specifies - under given semantic conditions - which triples logically follow from the triples present in a graph. Hawke et al. (2013) present different types of entailment regimes. We summarize them next.

**2.1.3.2.1 RDF Entailment** This entailment regime produces some types of inferred triples (Hayes and Patel-Schneider, 2014), we present two of its rules. The first detects that a IRI has an rdf:type rdf:Property. For example, if a triple `ComputingMachineryAndIntelligence hasAuthor Turing` is present in a graph, then the triple `hasAuthor rdf:type rdf:Property` is inferred. The second allocates blank nodes to string literals. For example, if a triple `Turing givenName "Alan Mathison Turing"^^xsd:string` is present in a graph, then the two triples and `Turing givenName _:b` and `_:b rdf:type xsd:string` are inferred.

**2.1.3.2.2 RDFS Entailment** RDFS Entailment extends RDF to support the definition of class and property hierarchies as well as defining domain and range restrictions (Hayes and Patel-Schneider, 2014). We demonstrate four of its rules but list all of them in Table 2.1. The first detects the type of a subject IRI. For example, if the two triples `numPages rdfs:domain Document` and `ComputingMachineryAndIntelligence numPages 43` are present in a graph, then the triple `ComputingMachineryAndIntelligence rdf:type Document` is inferred. The second detects the type of an object IRI. For example, if the two triples `citedBy rdfs:range Document` and `ComputingMachineryAndIntelligence citedBy ArtificalIntelligenceModernApproach` are present in a graph, then the triple `ArtificalIntelligenceModernApproach rdf:type Document` is inferred. The third detects subclass transitivity. For example, if the two triples `Proceedings rdfs:subClassOf Article` and `Article rdfs:subClassOf Publication` are present in a graph, then the triple `Proceedings rdfs:subClassOf Publication` is inferred. Detection of subproperty transitivity takes places in a similar manner. The fourth detects the super-class of a class. For example, if the two triples `Article rdfs:subClassOf Publication` and `ComputingMachineryAndIntelligence rdf:type Article` are present in a graph, then the triple `ComputingMachineryAndIntelligence rdf:type Publication` is inferred. Detection of subproperties of a property takes places in a similar manner.

| | **If a graph contains:** | **then it entails:** |
|---|---|---|
| ***rdfs1*** | any IRI aaa in D | aaa rdf:type rdfs:Datatype . |
| ***rdfs2*** | aaa rdfs:domain xxx .<br>yyy aaa zzz . | zzz rdf:type xxx . |
| ***rdfs3*** | aaa rdfs:range xxx .<br>yyy aaa zzz . | zzz rdf:type xxx . |
| ***rdfs4a*** | xxx aaa yyy . | xxx rdf:type rdfs:Resource . |
| ***rdfs4b*** | xxx aaa yyy. | yyy rdf:type rdfs:Resource . |
| ***rdfs5*** | xxx rdfs:subPropertyOf yyy .<br>yyy rdfs:subPropertyOf zzz . | xxx rdfs:subPropertyOf zzz . |
| ***rdfs6*** | xxx rdf:type rdf:Property . | xxx rdfs:subPropertyOf xxx . |
| ***rdfs7*** | aaa rdfs:subPropertyOf bbb .<br>xxx aaa yyy . | xxx bbb yyy . |
| ***rdfs8*** | xxx rdf:type rdfs:Class . | xxx rdfs:subClassOf rdfs:Resource . |
| ***rdfs9*** | xxx rdfs:subClassOf yyy .<br>zzz rdf:type xxx . | zzz rdf:type yyy . |
| ***rdfs10*** | xxx rdf:type rdfs:Class . | xxx rdfs:subClassOf xxx . |
| ***rdfs11*** | xxx rdfs:subClassOf yyy .<br>yyy rdfs:subClassOf zzz . | xxx rdfs:subClassOf zzz . |
| ***rdfs12*** | xxx rdf:type rdfs:ContainerMembershipProperty . | xxx rdfs:subPropertyOf rdfs:member . |
| ***rdfs13*** | xxx rdf:type rdfs:Datatype . | xxx rdfs:subClassOf rdfs:Literal . |

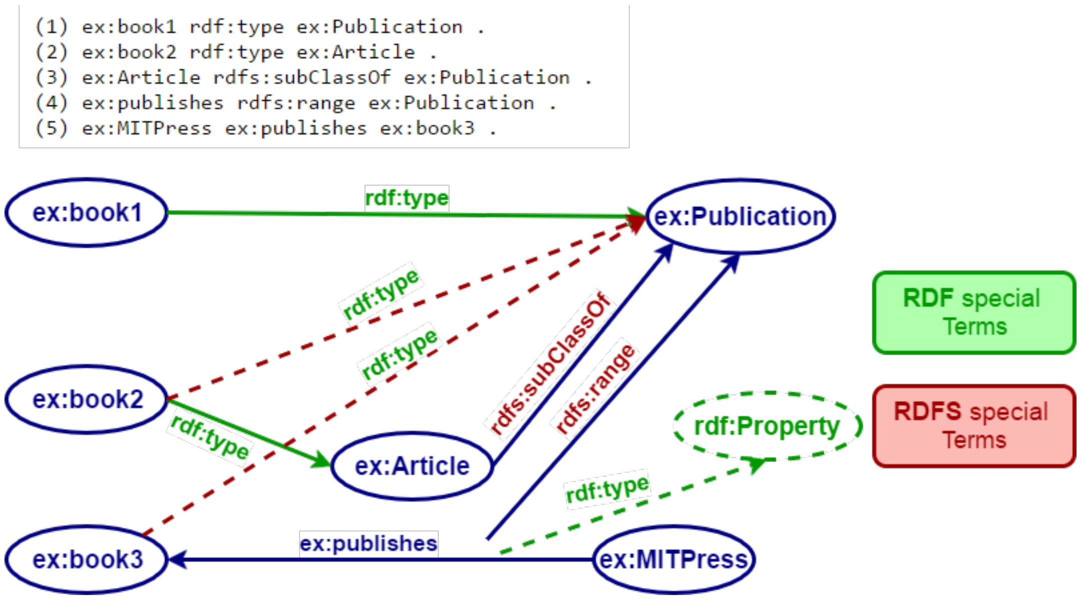
Table 2.1: RDFS Entailment Patterns (Hayes and Patel-Schneider, 2014)

Hawke et al. (2013) present an example showcasing some differences between the effects of RDF and RDFS entailments, presented in Figure 2.5[9].

**2.1.3.2.3 Datatype Entailment (D-entailment)** This entailment regime provides additional support for datatypes (Hayes and Patel-Schneider, 2014). It is considered to be "RDFS with datatype support". We present two of its rules. The first rule is similar to the first RDF entailment rule. It allocates blank nodes to object nodes that are assigned datatypes. For example, if the two triples `hasPublicationYear rdf:type rdfs:Datatype` and `ComputingMachineryAndIntelligence hasPublicationYear "1950"^^xsd:gYear` are present in graph, then the two triples `ComputingMachineryAnd-Intelligence hasPublicationYear _:b` and `_:b rdf:type xsd:gYear` are inferred. The second rule deals with value equality. For example, if the triple `ComputingMachineryAnd-Intelligence numPages "43.0"^^xsd:decimal`, then the triple `ComputingMachinery-AndIntelligence numPages "43"^^xsd:decimal` is inferred.

**2.1.3.2.4 OWL 2 Entailment** Two formal semantics for OWL 2 entailment regimes have been recommended (Hawke et al., 2013), the OWL2 RDF-Based Semantics entailment regime and the OWL 2 Direct Semantics entailment regime (Schneider et al., 2012; Motik et al., 2012b). Despite there being semantic differences between both regimes, as

[9]The figure has been corrected to reflect the reported errata.



```
(1) ex:book1 rdf:type ex:Publication .
(2) ex:book2 rdf:type ex:Article .
(3) ex:Article rdfs:subClassOf ex:Publication .
(4) ex:publishes rdfs:range ex:Publication .
(5) ex:MITPress ex:publishes ex:book3 .
```

Figure 2.5: Example Showcasing the Different Effects of RDF and RDFS Entailments (Hawke et al., 2013).

the former is a semantic extension of RDF, RDFS, and D-Entailment and the latter is related to description logic semantics, the semantics of both are directly specified by the structure and constructs of OWL 2 (Motik et al., 2012c). Moreover, the correspondence theorem states that OWL 2 RDF-Based Semantics can entail all that OWL 2 Direct Semantics can and that any OWL 2 Direct Semantics query can be re-written into a semantically equivalent query that allows the OWL 2 Direct entailment to also be an OWL 2 RDF-Based entailment.

Thus, similar to the RDFS entailment regime, they can both answer queries relating to domains, ranges, subclasses, subproperties, and whether an IRI is a property or a resource. Further to the RDFS entailment regime, they can address additional queries based on the OWL 2 vocabulary. Classes and properties can be found to be equivalent or disjoint, while individuals can be found to be the same or different. Both entailment regimes can also address queries related to class intersection, union, complement, and enumeration. Inferences about property restrictions include values and cardinality. Also, additional entailment queries can be addressed based on properties that are functional, inverse functional, reflexive, symmetric, and transitive. Finally, the entailment regimes can detect data ranges based on whether the ranges consists of intersections, unions, complements, one of, or type restrictions.

**2.1.3.2.5 RIF Core Entailment** RIF Core Entailment deals with two inputs, the RDF graph and the RIF document. The entailment regime checks what is entailed based on the referenced ruleset. There are two different types of rules: declarative rules and production rules. Declarative rules consider facts about the world and infer new knowledge about it. Production rules consider the facts and check which conditions apply so that certain actions are performed and changes are most likely made.

Reasoning in RIF declarative language dialects are thought of as a combination of instantiation, Modus Ponens, and evaluating conjunctions and disjunctions (Morgenstern et al., 2013). Instantiation applies a property that is known about a class to its members. Modus Ponens concludes that the consequent of a rule is true if its antecedent is true. Evaluating a conjunction yields true if each of the conjuncts is true; evaluating a disjunction yields true if one of disjuncts is true.

Reasoning in RIF production language dialects is about checking which rule in the set of if-then-else rules will be fired and hence which action will occur. This happens in five steps. First, if a rule's condition is satisfied then the rule is fired. Second, if several rules are candidates for firing, then they are considered to be in a conflict set. Third, of the rules in the conflict set, one rule is chosen based on the conflict strategy, for example, choosing the rule with the highest priority. Fourth, When a rule is fired, its action is carried out and a change takes place. Finally, the four steps are repeated until no changes occur, thus reaching a fixpoint.

## 2.2 Reason Maintenance and Materialisation

In this section, we set aside the discussion about provenance to briefly present how systems have otherwise tracked the production of their data and how the modification of their source data affects the data arrived at from them. Therefore, we first introduce reason maintenance and show how it is used in local and distributed systems. We then introduce incremental view maintenance with an emphasis on the Delete and Rederive (DRed) algorithm. Finally we briefly cover reason maintenance on the Semantic Web. The common limitation we encounter in these systems is that they do no rely on provenance.

### 2.2.1 Reason Maintenance

Expert systems and knowledge-based intelligent agents use inference procedures and domain knowledge to arrive to conclusions or goals. Their problem solving revolves around storing information and inferring new facts. Thus, their stored information may either be base information or inferred information. A standard system architecture of a problem solver comprising of an *Inference Engine* and a *Reason Maintenance System*[10] (RMS - or *Truth Maintenance System*, TMS) is shown in Figure 2.6. Inferences arrived

[10]We use the term Reason Maintenance System interchangeably with the term Truth Maintenance System; though the former is preferable as it has been described by Doyle (1983) as a less deceptive name.

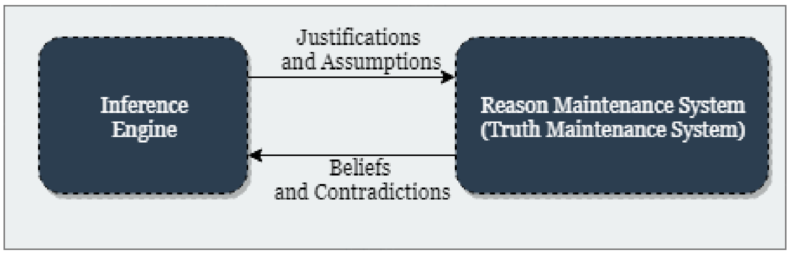


Figure 2.6: The two components of a problem solver

to by the inference engine are communicated to the RMS along with their justifications. A justification of a fact in an RMS comprises the reasons it, the RMS, believes this fact, i.e. the other facts, assumptions, or conclusions on which the validity of this certain fact depends, or the fact's dependencies.

Inevitably, new information will render some inferred facts incorrect or bring about some contradictions, necessitating their retraction; this is referred to as belief revision (Russell and Norvig, 2010). Reason Maintenance Systems are designed to handle complications arising from such situations; they have been originally called Truth Maintenance Systems because of their ability to restore consistency. Consistency is limited to keeping the system free from contradictions and free from beliefs that have no justifications. Martins (1990) identified the following issues that TMSes deal with: (*1*) *Non-monotonicity*[11] studies how belief in one proposition relies on the disbelief on other propositions; (*2*) *Disbelief propagation* studies how a consequence of something that has been disbelieved would in turn be disbelieved; and (*3*) *Revision of beliefs* centers around selecting the 'culprit' when a contradiction has been detected.

A TMS does not delete facts, instead it keeps track of which information is still believed and which is no longer believed. As an example of what an TMS does, assume that a knowledge base contained a sentence $P$ that was proven to be incorrect and must be retracted. If $P$ had been used to infer further sentences, say $Q$ and $R$, then those would also have been needed to be retracted, but only if no sentences other than $P$ had also inferred them. For example, if the knowledge base, KB, contained the sentences $S$ and $S \Rightarrow Q$, then $Q$ would not have been needed to be retracted.

In addition to tracing sources of contradictions or wrong conclusions, problem solvers must also explain and justify their actions like humans do. Explanations are used to clarify reasoning, justify recommendations, and answer possible questions. They also make such systems more intelligible, help in debugging them, produce outcomes for unanticipated situations, and clarify any assumptions made by the systems. There are two types of explanations that are usually provided, 'why' and 'how'. 'Why' explanations typically answer the question 'why was a fact requested?'. 'How' explanations answer

[11]Monotonicity, as the property of logical systems, states that as sentences are added to the knowledge-base the number of entailed can only *increase*; it is also expressed as: if $KB \vDash \alpha$ then $KB \wedge \beta \vDash \alpha$.

the question 'how was a certain conclusion or recommendation reached?'. Thus, reason maintenance systems should generate such explanations. Accordingly, this support for inferred facts, can be considered part of its incomplete provenance.

### 2.2.2 Local Belief Revision

Truth maintenance systems perform local belief revisions based on two[12] main approaches: justification-based and assumption-based. A justification-based TMS records dependencies among beliefs by listing, for each belief, all the beliefs that have immediately originated it. An assumption-based TMS records dependencies by listing, for each belief, all the assumptions that determine its derivation. We quickly review those.

**Doyle's Justification-based Truth Maintenance System** Doyle (1979) presented the justification-based truth maintenance system, or JTMS, to address how changes in beliefs should be handled. Every proposition, or statement, $P$ is annotated with the set of sentences that have inferred it, i.e. its *justification*, and each $P$, may be in one and only one of two states:

1. $P$ is a member of the set of current beliefs because it has a minimum of one valid, i.e. currently acceptable, justification. It is said to be *IN*.

2. $P$ is not a member of the set of current beliefs because it has no valid justifications, i.e. either has none or it has unacceptable ones. It is said to be *OUT*. Thus the JTMS does not delete sentences; this becomes useful when a justification is later restored, the sentence is simply marked back as *IN*.

Note that there is a difference between not believing in $P$ and believing in $\neg P$. Labellings have to be *consistent*, i.e. all the justifications are satisfied, and *well-founded*, i.e. justifications are non-circular.

An *assumption* is a belief that is current and which has a valid reason that depends on another belief that is not current. Assumptions are allowed to have justifications as well. This allows for assumptions to result in reasoned un-grounded beliefs, e.g. believing in an assumption $P$ as a result of a disbelief in $\neg P$.

Retracting assumptions also takes place in a reasoned manner, i.e. no assumption is retracted without a reason for its retraction. In case this reason becomes invalid later, the retraction ceases to be valid and the assumption is reinstated in the list of beliefs. Reasoned retraction of assumptions is achieved by using a *dependency-directed backtracking procedure*. This procedure revises the current set of assumptions that may be inconsistent so as to solve inconsistencies and keep

[12]Bry and Kotowski (2008) survey three more non-monotonic approaches: Logic-based, Hybrid, and Incremental.

the database contradiction-free. It does not operate on the content or the form of beliefs, rather it flags the set of conflicting beliefs and traces backwards to the reasons supporting them. It then retracts one of the assumptions it has reached thus restoring consistency.

**de Kleer's Assumption-based Truth Maintenance System** While a conventional justification-based TMS requires consistency in the set of currently believed data, the demand for consistency presents a challenge for simple qualitative tasks. In addition to being inefficient, JTMSs cannot consider several contradictory assumptions at a time. de Kleer (1986) presented the assumption-based truth maintenance system, or ATMS, that expedites the switch between the hypothetical states. Thus, while a JTMS marks sentences as either *IN* or *OUT* and keeps tracks of their justifications, an ATMS additionally tracks sets of assumptions for each sentence. If all assumptions in a set hold, then the sentence also holds. This waives the requirement for the knowledge-base's consistency.

### 2.2.3 Distributed Belief Revision

Based on Doyle (1979)'s JTMS, Bridgeland and Huhns (1990) presented a distributed Truth Maintenance (DTMS) algorithm that is used to restore inconsistency when justifications for a datum are added or removed. They considered a group of interacting agents, each having their own partially-independent belief system. Each agent's knowledge base contains two types of data, shared data - beliefs which has been shared in the past, and private data - beliefs that have never been shared. When a datum is labeled as IN, it is also labeled as either INTERNAL or EXTERNAL. An INTERNAL datum has a valid justification and is believed to be true. An EXTERNAL datum does not have a valid justification; it is believed to be true because another agent has shared it. The presented algorithm, *label-wrt*, is called when a justification is added or removed so as to identify said justification's consequences, handle the re-labelling, and share the new labels. Although no empirical results were presented, the authors have pointed out the following shortcomings with their algorithm: first, agents with less information may overrule others with more information, that is, if an agent believes in certain datum, it will force another agent to continue to believe in it, even if the other agent has additional information to support the contrary. Also, significant computation overhead arise when the shared data are large, when the data are shared among many agents, or when beliefs frequently change.

Dragoni and Puliti (1994) presented a framework for assumption-based distributed belief revision, with two focuses. First, instead of dealing with just the information, the system deals with the couples <information, informant>, as they asserted that the information's credibility and the source's reliability affect each other. Moreover, their system does not force the different agents to reach mutual agreements about the validity of beliefs, thus adhering to what is termed a ‘Liberal Belief Revision Policy’. This allows agents to uphold their own beliefs based on how they view evidence. Thus local consistency has a higher importance over global consistency, so as to avoid scenarios where some agents may mislead others by presenting compromised information whether deliberately or not. Global consistency, however, may be still be reached via voting as described in their later work in (Dragoni and Giorgini, 2003).

### 2.2.4 Incremental Maintenance of View Materialisations

Often times, a system may precompute and store all the consequences of its inference rules. This is referred to as view materialisation. Materialisation results in queries, performed directly on the stored facts, being faster (Gupta et al., 1993; Motik et al., 2015). This is especially useful when derived facts are inferred from data that are distributed over more than one system. However, when the data change due to insertions and deletions, recomputing the materialisation from scratch is unacceptable because materialisations are inherently expensive. The solution is thus is to compute only the changes to the materialisation responding to the changes in the data. Algorithms which do so are called *incremental view maintenance* algorithms.

Insertions are straightforward and do not raise the same problems as deletions, and thus approaches to enhance incremental maintenance algorithms have focused on deletions.

Consider, for example, a knowledge base with the following rules:

$$Document(x) \leftarrow Publication(x) \tag{2.1a}$$

$$Publication(x) \leftarrow JournalArticle(x) \tag{2.1b}$$

$$Publication(x) \leftarrow ConferenceProceedings(x) \tag{2.1c}$$

$$ConferenceProceedings(x) \leftarrow ConferenceArticle(x) \tag{2.1d}$$

The knowledge base also contains the following asserted fact:

$$ConferenceArticle(ComputingMachineryAndIntelligence) \tag{2.2a}$$

$$Publication(ComputingMachineryAndIntelligence) \tag{2.2b}$$

Therefore, after reasoning the knowledge base would contain the derived facts:

$$ConferenceProceedings(ComputingMachineryAndIntelligence) \tag{2.3a}$$

$$Document(ComputingMachineryAndIntelligence) \tag{2.3b}$$

However, since ‘Computing Machinery and Intelligence’ is a journal article and not a conference article, the knowledge base needs to be corrected to reflect that. Simply deleting the asserted fact (2.2a) is not enough because the derived fact (2.3a) needs to be deleted as well.

Gupta et al. (1993) presented the seminal *Delete and Rederive* (*DRed*) algorithm that starts by deleting all the data that had been derived from the original data that have been deleted. This includes any data that may also have alternative derivations independent of the deleted original data. In the above example, DRed would delete both derived facts (2.3a) and (2.3b).

DRed then re-inserts the subset of the derived data which has other original data that led to their derivation. In the above example, DRed would re-insert the derived fact (2.3b), as it may be arrived to using the asserted fact (2.2b).

Finally, any facts that need to be asserted are added to the knowledge. 'Computing Machinery and Intelligence' can now be correctly asserted as a journal article.

Note that the decision whether to materialise the derivations or to forgo materialisation and recompute each time is an optimisation problem which depends on storage constraints and the costs of updates and queries.

### 2.2.5 Reason Maintenance on the Semantic Web

On the Semantic Web, reason maintenance deals with handling changes affecting data which had previously been inferred.

Broekstra and Kampman (2003) addressed the problem of dealing with 'non-monotonous updates' [*sic*] in an RDF knowledge base, whereby some statements are deleted after inference has taken place. Their truth maintenance algorithm, which is part of Sesame's (Broekstra et al., 2002) architecture, only deals with 'disbelief propagation'. It is only invoked when *explicit* - non-derived- believed facts are retracted. It is based on Doyle (1979)'s JTMS, where dependencies between entailment rules are tracked, so every statement has a list of *justifications*, i.e. all the other statements on which it depends. Statements that are candidates for removal are marked as *suspended* and the algorithm loops over suspended statements and where the statement is explicit it is removed, and where the statement is a justification for a derived one, then that justification is removed. A statement whose justification has been removed is added to the list of suspended statements and re-examined. If upon re-examination no justifications are found then the suspended statement is removed. When a new statement is added, basic backward chaining is performed whereby the inferencer checks whether each statement is a conclusion of an entailment and if that is the case, it identifies which statements form the premise of the entailment. The algorithm's performance is evaluated against a brute-force algorithm which does not store any justifications, instead, when a statement is retracted, it is deleted and all inferred statements are discarded and the reasoner is re-invoked. Four data sets are used for evaluation and although the algorithm's storage and speed performance are significantly better on two datasets, which were medium-sized, it performed worse than the brute-force one on the other two.

Motik et al. (2015) had originally observed in (Volz et al., 2003, 2005) that reason maintenance systems posses an inherent disadvantage in their main feature of never permanently removing beliefs and justifications but merely disabling them. This leads to the collection of beliefs and justifications to progressively grow with their continuous influx thus increasing the cost of updates. Additionally, Motik et al. (2015) observed that Gupta et al. (1993)'s DRed algorithm over-deletes derived data causing inefficiency. They presented their Backward/Forward (B/F) algorithm which improves on DRed by checking whether triples marked for deletion have alternative derivations and if that is the case, then the algorithm do not delete them. Their algorithm outperforms DRed on all their test datasets.

The aforementioned systems track changes on the triple level or require the implementation of a special reasoner so to perform the materialisation and the re-derivation. Additionally, none of these system make use of provenance, which we have previously mentioned its benefits. In the next section we discuss systems that exploit provenance to perform recomputations.

## 2.3 Provenance

In this section, we introduce the concept of provenance. We follow that by enumerating its usages and the benefits it brings about. We also present the different perspectives on provenance and how its intended use affects its collections.

### 2.3.1 What Provenance Is

The Oxford English Dictionary[13] defines the provenance[14] of an item to be its derivation from its particular source to its particular state. Conceptually, the provenance of a piece of data - be it a triple, graph, or dataset - is its history which consists of any data items involved in its inception, i.e. its source and origins, as well as the processes that led to its derivation, i.e. any processing steps involved in how it came to be (Woodruff and Stonebraker, 1997; Buneman et al., 2000a,b; Hartig, 2009). Concretely, the piece of data's provenance refers to the documented records of its derivation and modification.

[13] http://www.oed.com/view/Entry/153408

[14] Bose and Frew (2005) present the following terms that have been used synonymously with provenance: audit trail, data archaeology, data genealogy, derivation history, data set dependence, filiation, lineage, and pedigree.

The source of a piece of data might be a human, a scientific instrument, a database, or a document, and so the origin of that piece of data includes the base data which contributed to its creation, how it was recorded, and - if applicable - the parameters of the recording instrument (Buneman et al., 2000b). The processing steps of the piece of data, or the history-of-how-a-data-item-was-produced side of provenance, include the algorithms applied to produce that piece of data and their respective parameters.

Provenance has become a key requirement in a range of applications. It is a special form of metadata (Buneman et al., 2000b; Berners-Lee et al., 2006); descriptive annotations intended for machine consumption (Goble, 2002). The goal of provenance is to answer the seven $W$ questions: *W*ho, *W*hat, *W*here, *W*hy, *W*hen, *W*hich, and *(W)*how (Goble, 2002). Individually, every one of these contributes a particular type of provenance information that can be used on its own; jointly they provide a complete picture.

### 2.3.2 Provenance Benefits and Usages

Provenance has been thoroughly surveyed in the literature, including by Bose and Frew (2005), Simmhan et al. (2005), Glavic and Alonso (2009), Moreau (2010), and Pérez et al. (2018), and its application to numerous domains has proven to bring about much benefit. We list some of these benefits and usages below:

Attribution, copyright, and credit: Provenance is used to pinpoint creators of data items (Bose and Frew, 2005; Simmhan et al., 2005; Glavic and Alonso, 2009; Moreau, 2010).

Data quality communication: Provenance showing the sources and transformations of data items is used to communicate data quality, to infer their reliability, and prove their accuracy, currency or timeliness, redundancy, and suitability (Buneman et al., 2001; Bose and Frew, 2005; Simmhan et al., 2005).

Explanation and interpretation enhancement: Provenance is used to enhance interpretation of information “in primary, secondary, and personal repositories” (Goble, 2002). Thus, by allowing the examining of the sources of the data items and by gaining insight on how they evolved, provenance facilitates deeper understanding of and promotes learning about the data and how they were generated (Haynes et al., 2009; Ikeda et al., 2013). Only when consumers of information services can understand the information and thus are able to make decisions of when to trust it, is the web’s promise of distributed and interoperable information systems realised (McGuinness and Da Silva, 2003b).

Anomaly investigation and debugging: Provenance can be used to verify the correctness of processed and derived data and to trace and investigate outdated or erroneous processes or source data that may have resulted in any anomalies or errors

(Woodruff and Stonebraker, 1997; Ikeda et al., 2013). It can furthermore identify the formers' impact on the latter. It also protects users from erroneous results arising from mistaken assumptions or misinformation about the setup of systems as well as prevent misuse and misinterpretation of environmental data (Bose and Frew, 2005).

Data understanding by non expert users: Provenance is used to allow the understanding of data by non expert users (Bose and Frew, 2005), and hence, facilitates their acceptance of the former (Haynes et al., 2009).

Historical data resources usage and replication recipes: Provenance is used to enable future users to use historical data resources (Bose and Frew, 2005). Additionally, provenance may be used as a replication recipe when data derivations need to be repeated (Simmhan et al., 2005).

Transparency of systems: Provenance allows for the transparency of systems, thus making them auditable and allowing compliance checks to be performed (Moreau, 2010). Particularly, when provenance is treated as an audit trail, it is used to locate errors in data generation (Simmhan et al., 2005). Hence, it allows developing and increasing trust in the behavior and results of the system (Haynes et al., 2009). Finally, provenance can be used to execute compliance checks (Moreau, 2010).

Recomputation: Provenance is increasingly being used in efficient recomputation when data change. Recently, Missier et al. (2019) reported on the first workshop on incremental recomputation which took place in July 2018. We discuss recomputation in more detail in Sections 2.4 and 2.5.3.

### 2.3.3 The Different Perspectives on Provenance

There have been different approaches to capturing and handling provenance depending on its intended use. Simmhan et al. (2005) distinguished two different perspectives on provenance - with Gil et al. (2013) further introducing a third one - and as a result what provenance would reflect:

- Workflow-oriented or process-centred provenance: this focuses on tracking the steps and actions that result in the production or modification of the data whose provenance information is under consideration. It is usually coarse-grained (Buneman et al., 2008; Ikeda et al., 2013), i.e. each module or component involved is viewed as a 'black-box' and its output depends on all its inputs (Moreau, 2010). Note that Ikeda et al. (2013) label provenance generated at a workflow transformation level as logical provenance.
- Data-oriented or object-centred provenance: this focuses on tracking the other pieces of data that result in the production or modification of the data whose

provenance information is under consideration. It is usually fine-grained (Buneman et al., 2008; Ikeda et al., 2013), i.e. it includes detailed descriptions of how the data came to be. Note that Ikeda et al. (2013) label provenance generated at a data-oriented level as physical provenance.

- Agent-centred provenance: this focuses on tracking the organisations or people that took part in the production or modification of the data whose provenance information is under consideration.

### 2.3.4 Provenance Vocabularies

Provenance data models and ontologies have been widely developed during the past decade. The Dublin Core[15] vocabulary is used to describe resources and provide their basic provenance such as authors' names, contributors' names, and creation and change dates. The Provenir Ontology[16] is an upper-level provenance onology for modeling and managing provenance in eScience (Sahoo et al., 2008). Provenance is represented using three base classes: data, agent, and process, as well as properties which link the three classes to represent the transformation and derivation of data, the participation of processes in the production of data, and the responsibility of agents for the processes. The Provenance Vocabulary's aim is to provide a means for publishing - and consuming - provenance about Linked Data (Hartig and Zhao, 2010). It supports the expression of provenance in RDF and thus allows the providers of the data to publish those data's provenance as Linked Data. The vocabulary is defined as an owl ontology, with general terms including three classes: Actor, Execution, and Artifact, each having further sub-classes. Like Provenir, properties link the three classes to represent the creation and usage of data, the execution of processes, and the responsibility of actors for executions. Other notable vocabularies include PREMIS[17], the Web Of Trust Schema (WOT) [18], the SWAN Ontology[19], the Semantic Web Publishing Vocabulary[20], and the Changeset Vocabulary[21]

The Open Provenance Model[22] (Moreau et al., 2009) describes the provenance of "things", physical or immaterial, in a historic manner. Provenance is represented by a causality Directed Acyclic Graph (DAG) annotated with extra information. The graph is constructed using three different types of nodes linked with five different types of edges representing dependencies between them. The source of an edge is an effect node and the destination is a cause node. The three OPM nodes are Artifact, Process, and Agent,

[15]`http://dublincore.org/`
[16]`http://wiki.knoesis.org/index.php/Provenir_Ontology`
[17]`http://www.loc.gov/standards/premis/`
[18]`http://xmlns.com/wot/0.1/`
[19]`https://www.w3.org/TR/hcls-swan/`
[20]`http://wifo5-03.informatik.uni-mannheim.de/bizer/WIQA/swp/SWP-UserManual.pdf`
[21]`http://vocab.org/changeset/schema.html`
[22]Outcome of the Provenance Challenge Series initiated in May 2006.

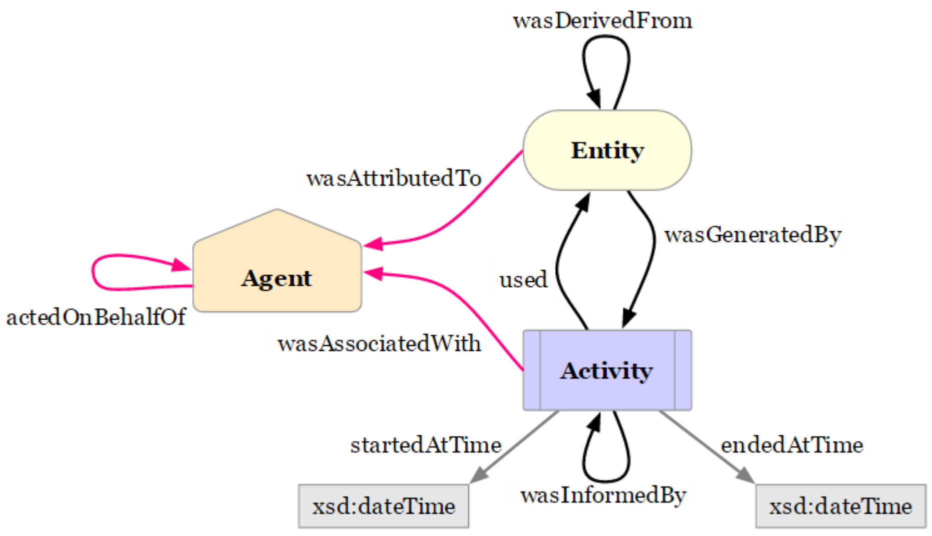


Figure 2.7: PROV Core Structures (Moreau et al., 2013)

while the edges represent the generation, usage, and derivation of artifacts and the triggering and control of processes. OPM laid the foundations for PROV, which we describe next.

### 2.3.5 The PROV Data Model

The PROV specification was produced by the World Wide Web Consortium (W3C) to accommodate the three different perspectives on provenance described in Section 2.3.3 by defining provenance as the description of "the use and production of entities by activities, which may be influenced in various ways by agents" (Groth and Moreau, 2013; Gil et al., 2013). The PROV family of documents contains recommendations and specifications on how to capture and express provenance, as its goal is to support the publication and interchange of provenance on the Web jointly with the data it describes using the popular formats XML and RDF.

Among the PROV family of documents, the conceptual data model PROV-DM is the basis. PROV-DM is domain-agnostic but can be extended to allow the inclusion of domain-specific information. The core structure of PROV contains three types and seven relationships as shown in Figure 2.7.

The *entity* is the thing who provenance is being described. The *activity* is what had taken place over a time period and had produced - *generated* - or had utilised - *used* - one or more entity. Activities would have *communicated* with each other if one activity used an entity that had been generated by another, whereas entities would have *derived* other entities if the the generation of one entities was influenced by other entities. The *agent* is what was responsible for an activity that has taken place, for the existence of an entity, or for another agent's activity." An agent is *associated* who an activity if it was responsible for it; i.e. it had a role in it, and an entity its *attributed* to an agent if it had been ascribed to it. An agent was *delegated* by another agent if the first had acted *on behalf of* the latter.

A few concepts that are worth expanding on are as follows. *(1) Derivation* is not restricted to the generation of a new entity but also includes the transformation or updating of an entity resulting in a separate newer one. *(2) Revision*, a subtype of *derivation*, where the new entity includes considerable content that is the same as the original entity. *(3) Quotation*, another subtype of *derivation*, where an entity is a repetition of another derived from it by a copy action.

The extended structure of PROV provides support for more advanced expressions of provenance. In addition to allowing subtypes, it includes alternates, bundles, and collections. Alternates are expressed in two relationships. The first is where an entity is a *specialisation* of another, whereby while the first has all the elements of the latter, it also contains additional specific ones. The second is *alternate*, whereby two entities are representations of the same thing. Bundles allow the expression of provenance of provenance. Finally, collections allow the expression of provenance of a group of entities, with the *membership* relationship representing the inclusion of an entity in a collection.

## 2.4 Provenance and Partial Recomputation

There are several scenarios which give rise to the need to recompute derived data. Source data may be found to be incorrect or outdated and processes may be discovered to be buggy (Ikeda and Widom, 2010). Additionally, underlying data, algorithms, or dependencies may evolve over time (Missier et al., 2016). In these cases, and while it is imperative to carry out recomputations, full and complete debugging or recomputation may prove to be expensive. The following pieces of work rely on provenance to determine which data would be affected by change and would need to be recomputed.

### The Panda System

Ikeda et al. (2011) presented a prototype system based on their plans in (Ikeda and Widom, 2010) for a comprehensive system that uses provenance for explanations, verifications, and recomputation. Their main goal was to eschew complete recomputations by pinpointing which input elements would have been conducive for generating given output elements, so that a data-oriented workflow would be rerun solely on the affected data that would need to be refreshed. While their Panda system pinpoints which program fragments require re-execution to refresh output data as well as tracking modifications to input data, it does not employ any of the provenance vocabularies, specifically the

PROV data model. We summarize their model that incorporates transformations and provenance next.

Each transformation, $T$, takes an input set, $I$, and produces an output set $O$, which is a set of couples of the form $\langle o, p \rangle$. So, each output $o$ is annotated by a *provenance predicate* $p$. Formally, $T(I) = O = \{\langle o_1, p_1 \rangle, ..., \langle o_n, p_n \rangle\}$. When the input data change, a *single-transformation refresh* procedure is invoked. First, a backward tracing query on the new input is performed using $p$ as its guide in order to find the desired subset of the new input. Second, a forward propagation procedure applies $T$ on that subset to produce the new value of $\langle o', p' \rangle$. They presented an algorithm, *workflow refresh*, which recursively extends *single-transformation refresh* to refresh the output elements of composition of transactions. The provenance predicates were not sufficient to support many-many transformations, so *forward filters* were introduced. A transformation instance thus becomes $T(I_1, I_2, ..., I_m) = O$ where $O = \{\langle o_1, (p_1^1, ..., p_m^1), f_1 \rangle, ..., \langle o_n, (p_n^1, ...p_n^m), f_n \rangle\}$. Despite incurring 30% time overhead and a 56% space overhead to capture provenance, they were able to refresh between 52% and 70% of output data elements before the time cost exceeded that of re-running the workflow.

### The ReComp Framework

Missier et al. (2016) noted that the detailed provenance of derived *knowledge assets* can be analysed to support reasoning when recomputations are required due to the occurrence of some change. They catalogue a change by its effect either on the input data or the algorithms' dependencies. They presented the case for the use of provenance to inform the decisions about which precise knowledge asset would need to be recomputed. They presented an initial model where they introduced the *ReComp* framework whereby data are versioned and prospective functions would detect and quantify changes between any two versions of the data. They aim to enable their model to select processes that require recomputation, to decide between complete or partial recomputation, and - in the case of partial recomputation - to select the starting point, referred to as a *starting component*.

## 2.5 Provenance and the Semantic Web

We have so far provided an overview of both provenance and the Semantic Web. In this section, we highlight the benefits gained from using provenance on the Semantic Web. We then present an overview of the works preceding ours that have applied provenance to Semantic Web applications.

### 2.5.1 The Need for Provenance on the Semantic Web

In open mediums like the World Wide Web, where the reuse of data provides additional value, there is heavy reliance on interconnected information stored in distributed environments, which may at times be questionable, conflicting, or inconsistent. In general, people have learned not to trust the data on the Web blindly and have become capable at examining and assessing data sources (Buneman et al., 2000a). In cases where those sources are unidentified or information about the sources is unattainable, information consumers may become sceptic and require more indications that the sources are reliable and credible publishers of information before choosing to believe the information (da Silva et al., 2003). Moreover, and in the current open medium of the Web, most of the data have been copied from other places and were transformed, edited, corrected, and annotated in the process (Buneman et al., 2000a,b). Correcting and annotating these data provide an added value since, in turn, they become sources for other information repositories. Additionally, it is necessary for certain people to have assurance about the timeliness and accuracy of the data they are working with (Buneman et al., 2000a).

The true potential of the Semantic Web would be attained when programs collect information from varied sources, process it, and exchange it with other programs (Berners-Lee et al., 2001). But, just like people should not trust everything published on the Web, software agents and people should not believe all assertions published on the Semantic Web. Indeed, Berners-Lee et al. (2001) pressed that "agents should be skeptical of assertions that they read on the Semantic Web until they have checked the sources of information". Furthermore, different people can make different statements about a particular resource and these different statements can be kept separate but can also be combined. Thus, there are additional things to be considered by humans or agents who must make informed choices about which data to use from applications and these decisions will depend not only on the source but also on the "the suitability and quality of the reasoning/retrieval engine, and the context of the situation" (McGuinness and da Silva, 2004).

It has been argued that knowing the provenance of a piece of data is as important as knowing its actual value (Buneman et al., 2000b; Berners-Lee et al., 2003), with Berners-Lee et al. (2001) advocating early on for the need of proof generation and exchange. Therefore, it has always been evident that provenance is vital to examine the piece of data's reliability, detect any redundancies, determine its currency and suitability for usage, and allow users to determine whether or not to trust the actual value (Buneman et al., 2000b; Moreau, 2010). Likewise, it has been maintained that users trust RDF graphs depending on the content of the graphs, available information about them, and the tasks the users need to perform (Carroll et al., 2005). Finally, Moreau et al. (2014) have incorporated provenance in the Semantic Web Layer Cake as shown in Figure 2.8.

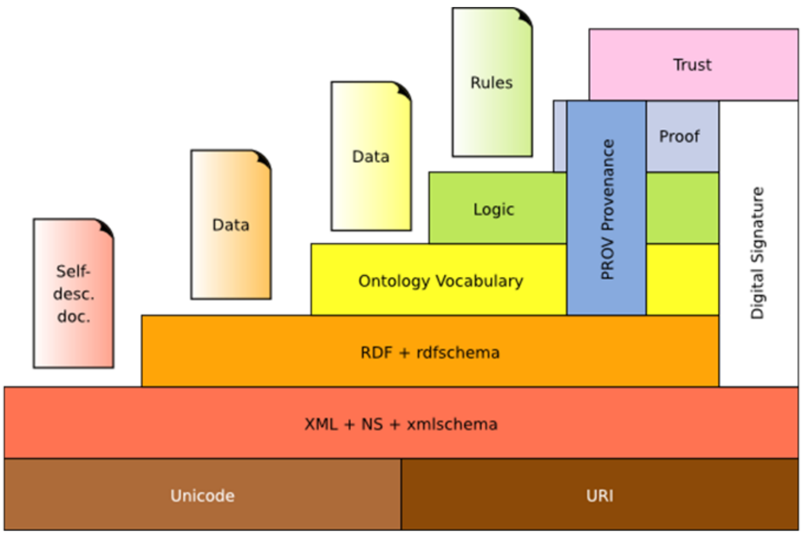


Figure 2.8: Provenance in the Semantic Web Layer Cake (Moreau et al., 2014)

### 2.5.2 Proofs and Explanations for the Semantic Web

In their unpublished paper describing Cwm[23], Berners-Lee et al. (2003) made the argument for Semantic Web agents' responsibility for being aware of provenance and liable for its management. Information is Cwm is expressed in N3, and each triple is stored with a provenance record describing the reason for adding it to the triple store. Provenance records are produced by the reasoner and are used in proof processing. Proofs are generated whenever the knowledge base is modified. This could be due to adding new triples from input sources or to new triples being inferred by the reasoner. Proofs also point to source and supporting materials. Cwm also allows users to sign documents.

The Inference Web and the Proof Markup Language were introduced in (da Silva et al., 2003; McGuinness and Da Silva, 2003a; McGuinness and da Silva, 2004) and further developed in (McGuinness et al., 2007; da Silva et al., 2008) to provide an infrastructure for knowledge provenance. In addition to information about data sources and descriptions of processes contributing to the information produced, knowledge provenance contains proof-like information about how knowledge was arrived to. Thus, the Inference Web and the Proof Markup Language constitute an infrastructure that allows systems to produce portable explanations. The main purpose of the proof language PML, later expanded to PML 2, is to enable various systems to generate, represent, and share proof steps. It also supports the production of other provenance metadata for the purpose of trust and justification, including authorships, credibility of sources, and reasoners' assumptions. The Inference Web's purpose is to allow the display and modification of these proofs. Fox and Huang (2003) expand on the notion of knowledge provenance and describe it as consisting of four levels. On the first level, Static KP focuses on provenance of static and certain information. On the second level, dynamic KP considers how the

[23]Available for download on https://www.w3.org/2000/10/swap/doc/cwm.html

validity of information may change over time. On the third level level, Uncertain KP considers information whose validity is inherently uncertain. Finally, on the fourth level, Judgment-based KP) focuses on social processes necessary to support provenance.

The aforementioned systems enable their reasoners to generate proofs for explanations. Effectively, they provide the provenance of the inferences made. This provide an indicative, albeit not a comprehensive, review of how results were generated.

### 2.5.3 Provenance of SPARQL Updates

While there are works which discussed provenance of SPARQL queries on the Semantic Web, for example by Dividino et al. (2009); Damásio et al. (2012); Wylot et al. (2014); Geerts et al. (2016) with Theoharis et al. (2011) providing a survey on provenance of queries; we focus on the work on provenance of SPARQL updates.

Almost all of the work on provenance of SPARQL queries and updates built on the seminal work presented in Buneman et al. (2001) and Green et al. (2007b), with both formalising the provenance of data in closed database systems. Buneman et al. (2001) defined the data provenance of a tuple as the specification of its origins and of the processes which resulted in its arrival in the database. They distinguished between the where-provenance of a tuple, being the locations in the database where its data came from, and the why-provenance of a tuple which appears in a query, being all the source data which contributed to the former's appearance. They also presented a deterministic model and a query language which they used to represent the provenance of views and queries. Green et al. (2007b) argued that why-provenance and where-provenance were not sufficient and introduced how-provenance, which addresses how the source data contribute to a tuple's appearance in the results. They presented a tuple-based abstract provenance model which they used in their annotation-based approach. Each source tuples is annotated with a unique *provenance token*, represented by the tuple's id; the provenance of an output tuple consists of the *provenance expression* which describes how source tuples are combined to produce it.

Moving on from closed database systems to collaborative data sharing systems, Green et al. (2007a) extended their aforementioned work - as a basis for their ORCHESTRA Collaborative Data Sharing System (Ives et al., 2008), where they focused on update exchange, schema mapping, trust evaluation, and extending Gupta et al. (1993)'s DRed algorithm. Their incremental update exchange algorithm allows the update of both data instances and the provenance associated with each tuple. When a deletion occurs, the algorithm utilises provenance information to flag which tuples are no longer derivable and should be subsequently deleted. Similarly, Flouris et al. (2009) extended Buneman et al. (2008)'s work on the implicit provenance of database queries and updates. They tracked the provenance of all triples - explicit or implicit (inferred), using colours - where a colour attached to a triple represents the ID of its source triple, or the combination of their IDs if it has more than one source triple. Thus, triples are modified to become quadruples of the form subject-predicate-object-colour. However, their work considered inferred quadruples independent of their sources. Before a quadruple is deleted, all the quadruples that can be inferred from it are inserted first. Then, all the quadruples that would infer the quadruple in question are deleted along with the quadruple itself. Further, Avgoustaki et al. (2016) extended both Buneman et al. (2008)'s and Green et al. (2007b)'s work by also using quadruples. Their quadruples' fourth elements are named graphs and quadruples' provenance is maintained in separate tuples with an id element linking to it. They, however, did not consider deletions; their algorithm described how to insert quadruples and record their provenance.

The aforementioned works track provenance on the triple level, which we argue is not always feasible nor is it particularly scalable, especially if they were to use PROV. Given a graph, even if the provenance of each of its triples consists of only one `triple prov:wasDerivedFrom sourceTriple` entry, a graph's provenance graph would be a little larger than it. Adding more provenance information, if restricted to the activity and agent that produced a triple would result in a graph having its provenance graph at minimum triple its size. This is one of the differences between those works and ours, as we rely on provenance on the graph level; the other being that the graphs we consider are produced using the entirety of two other graphs.

Finally, Halpin and Cheney (2014) presents work similar to ours that tracks dynamic provenance of collections using an extension of PROV, the Update Provenance Vocabulary (UPD), which allows them to capture SPARQL updates performed on raw data in a dataset. UPD provides new components that assist in tracking versions of graphs, as well as one subtype of prov:Activity - upd:update - that is to be used with a subtype of prov-type - upd:type - to document update activities. However, their work only considered updates, ignoring other operations - which this report studies - that may affect a graph or its provenance, such as fetching and entailment.

## 2.6 Semantic Web Benchmarks

On the Semantic Web, large applications may have to handle large quantities of data. In order to evaluate such applications' efficiency, scalability, or reasoning capabilities, benchmarks have been created over the past couple of decades. RDF benchmarking has been an established practice for some time now, with the W3C maintaining a list[24] of those benchmarks for the past decade.

[24]The RDF Store Benchmarking list maintained by the W3C is available on https://www.w3.org/wiki/RdfStoreBenchmarking.

| **Mono-tonicity** | **Bench-mark** | **Domain** | **№ of Classes** | **№ of Properties** | **Max Triples** | **№ of Queries** |
|---|---|---|---|---|---|---|
| Static | LUBM | University | 43 | 32 | $6.8 \times 10^6$ | 14 |
| | UOBM | University | 69 | 43 | $2.2 \times 10^6$ | 16 |
| | SP$^2$Bench | Bibliography | 8 | 22 | $2.5 \times 10^7$ | 17 |
| | BSBM | E-Commerce | 8 | 51 | $10^8$ | 12 |
| | WatDiv | E-Commerce | 16 | 13 | $10^7$ | 12500 |
| | DLUBM | University | 43 | 32 | - | - |
| | gMark | *Independent* | *user-defined* | | - | $10^8$ |
| Evolving | EvoGen | University | 53 | 51 | - | - |
| | EGG | *Independent* | *user-defined* | | - | - |
| Streaming | DBPSB | *Independent* | 685* | 2795* | $1.5 \times 10^8$ | 25 |
| | SRBench | *Independent* | 685* | 2795* | $1.7 \times 10^9$ | 17 |

Table 2.2: Summary of Semantic Web Benchmarks

Existing works focus on different factors and performance metrics when presenting their benchmarks, for example, some focus on reasoning capabilities and scalability, while others focus on efficiency of querying and storage. Additionally, almost all benchmarks artificially generate synthetic data; the exceptions being those that use the DBpedia[25] dataset. The former argue that their requirements are not met by real data sources, while the latter argue that synthetically-generated data are not representative of the real world and so their datasets are real world data taken from actual sources.

In this section, we present a short partial survey of existing benchmarks. We start with ones which produce static data then move on to those that produce or make use of dynamic data. The surveyed benchmarks are summarised in Table 2.2.

## 2.6.1 Static Benchmarks

Guo et al. (2005) presented the Lehigh University benchmark - LUBM - based on an OWL Lite ontology of moderate size and complexity for the university domain with 43 classes and 32 properties. They developed a data generator which generates synthetic data based on the university ontology, with random arbitrarily scaled instances of classes and properties as well as some restrictions, such as ratios of the number of instances of some classes relative to the number of instances to other classes. LUBM also comes with 14 SPARQL test queries, with the following factors to be taken into consideration when running the queries: input size, selectivity, complexity, assumed hierarchy information, and assumed logical inference. Additionally, Guo et al. quantitatively analysed four knowledge base systems using five performance metrics: load time, repository size, query response time, query soundness, query completeness, and combined metrics. Five sets of test data were used with increasing sizes, with the largest being 6,800,000 triples.

[25]https://wiki.dbpedia.org

*These are the current numbers of classes and properties of DBpedia.

Arguing that LUBM generated graphs are isolated and instances from different graphs do not link to each other, thus weakening LUBM's ability to adequately measure the scalability of systems and the inferencing capabilities of reasoners, Ma et al. (2006) extended LUBM and presented the University Ontology Benchmark - UOBM. UOBM is based on two ontologies, an OWL Lite ontology containing 51 classes and an OWL DL ontology containing 69 classes, with both containing 43 properties. It also comes with 13 queries for OWL Lite tests and an additional 3 OWL DL tests. Additionally, Ma et al. evaluated three ontology systems using four performance metrics load time, query response time, query completeness and query soundness. Six sets of test data were used with increasing sizes, with the largest containing 2,200,000 statements.

While LUBM and UOBM focus on reasoning and scalability, some succeeding benchmarks turned their attention to evaluating the efficiency of RDF stores' storage, such as Bizer and Schultz (2009)'s Berlin SPARQL Benchmark - BSBM, and the performance of SPARQL engines, such as Schmidt et al. (2009)'s SP$^2$Bench.

Schmidt et al. (2009) argued that LUBM and UOBM are inadequate for comprehensively testing SPARQL implementations and the challenges that SPARQL engines face. They presented SP$^2$Bench which generates synthetic data based on DBLP, but which mirror DBLP's (DBL) real-world characteristics and realistic distributions. SP$^2$Bench also comes with 17 queries. Additionally Schmidt et al. evaluated five systems using the following performance metrics: load time, success rate, per-query performance global performance, and memory consumption. Six sets of test data were used with increasing sizes, the larged containing 25,000,000 triples.

Bizer and Schultz's BSBM does not rely on heavyweight reasoning but aims to help developers of applications to compare RDF stores with SPARQL endpoints accepting concurrent queries. So, it simulates realistic workloads by measuring the performance of a system receiving multiple concurrent executions of SPARQL queries against large amounts of RDF data. The data generator generates synthetic data based on an e-commerce use case. BSBM also comes with 12 queries. Additionally, Bizer and Schultz evaluated four RDF stores with two SPARQL-to-SQL rewriters using three performance metrics: load time, query mixes per hour, and queries per second. Four sets of test data were used with increasing sizes, the largest containing 100,000,000 triples.

Aluç et al. (2014) analysed LUBM, SP$^2$Bench, BSBM, and DBPSB and found that none were properly "suitable for testing systems for diverse queries and varied workloads". So, they presented the Waterloo SPARQL Diversity Test Suite - WatDiv, which focuses on stress testing as well as detecting issues with RDF data management systems' physical designs. They developed a data generator which generates synthetic data based on WatDiv's schema and a query template generator which generates query templates also based on the WatDiv schema and user parameters. The schema contains 16 classes and 13 properties, but they also make use of other schemas such as dc, foaf, gr, and sorg.

Additionally Aluç et al. evaluated five RDF data management systems using two sets of test data, one containing 10,000,000 triples and the other 100,000,000 triples. 12500 queries were generated from 125 query templates.

Motivated by the rapid growth and adoption of Linked Data, as well as the Linked Data Benchmark Council's (LDBC) (Angles et al., 2014) goals to "establish benchmarks, and benchmarking practices for evaluating graph data management systems", Keppmann et al. (2017) presented the Distributed LUBM - DLUBM, a benchmark for creating and deploying distributed and interlinked datasets based on LUBM. Their Linked Data generator generates synthetic data based on LUBM's schema and scaling but has the added functionalities of being interlinked across different graphs. The generated data was used to measure the performance of a Linked Data query engine using the same queries of LUBM. Their evaluation focused on the number of derived triples and different numbers of hosts.

Also motivated by the LDBC's goals, Bagan et al. (2017) presented gMark which focuses on the evaluation of systems based on query workloads instead of individual queries. gMark generates both graphs and query workloads which are both domain and language independent. Queries are classified into three categories. Four sets of test data were used to measure query execution times of four systems with increasing sizes, the largest containing 100,000,000 triples.

### 2.6.2 Dynamic Benchmarks

Despite being included as one of the requirements for future comprehensive benchmarks by Weithöner et al. (2006), modifications to data were not included in benchmarks[26] until nearly a decade later.

To enable the benchmarking of versioning RDF systems, Meimaris and Papastefanatos (2016) presented EvoGen - initially introduced in (Meimaris, 2016) and based on their work in (Meimaris et al., 2014) - an extension of LUBM which generates synthetic datasets that change at both instance and schema levels. They developed a data generator which generates successive versions of data based on a university ontology which extends that of LUBM with 10 new classes and 19 new properties. The change in size of a dataset $D$ from time $t_i$ to time $t_{i+n}$ is set by the user, termed *shift*, and calculated as $h(D)|_{ti}^{t_{i+n}} = \frac{|D_{i+n}|-|D_i|}{|D_i|}$. While the authors claimed in (Meimaris et al., 2014; Meimaris and Papastefanatos, 2016) that their model adds and deletes triples according to a configurable schema evolution parameter by allowing the user to set a negative *shift*, they subsequently confirmed that this had not been the case and that their implemented[27] *Version Management* and *Change Creation* components only generate versions with triples

[26] Ma et al. (2006) did mention in their discussion section intentions to add update tests in UOBM and an incomplete update use case is proposed on Bizer and Schultz (2009)'s BSBM's website.

[27] https://github.com/mmeimaris/EvoGen

added to them and do not generate versions with triples removed from them (Meimaris, 2018). Specifically, the new versions generated by their implementation are formatted in two ways. The first produces RDF graphs containing only the newly-created triples for that version and the second produces a log file containing the inserted triples represented as entities using the Change Ontology described in (Meimaris et al., 2014). Moreover, and despite asserting the possibility of the production of either all fully materialised versions or the initial version along with the series of changes, according to the user's choice, in reality their implementation only does the later. The produced datasets consist of the initial version of each department within a university along with subsequent insertions. Additionally, because the ontology is hard-coded like LUBM and UOBM, the new classes and properties in the extended ontology are also hard-coded, with the new classes being subclasses of LUBM's classes. Finally, as previously mentioned, EvoGen takes as input a degree of change, i.e. the *shift*, then calculates the difference between the size of one version of the dataset and the next by spreading the *shift* over the number of required versions. For each version, this *shift* is combined with hard-coded numbers to calculate the probability of change of an instance type, referred to as *weight*. There is no clear explanation or rationale for why those specific numbers were chosen.

Also looking into producing evolving RDF graphs, Alami et al. (2017) presented a prototype of their work in progress on their Evolving Graph Generator - EGG. As it is in its early stages, the framework is not yet a benchmark. EGG is built to be used along with the previously mentioned gMark (Bagan et al., 2017), and so is also domain-independent and schema-driven. After an initial static version of a graph is produced by gMark, EGG uses evolving configurations set by the user to change property values to simulate change over time, for example, price of a hotel room. So, EGG does not create new instances or delete old ones.

In addition to static and evolving benchmarks, there are a couple of streaming benchmarks which are also the only benchmarks with deal with real data from the real world. Morsey et al. (2011) presented the DBpedia SPARQL Benchmark - DBPSB - aimed at evaluating the performance and scalability of triple stores. DBPSB generates data based on the DBpedia dataset mimicking the latter's property of containing large numbers of classes and properties. It also comes with 25 SPARQL query templates derived from the most common queries posed to DBpedia's SPARQL endpoint. Three portions, ranging from 10% to 50% to 100%, of the dataset were used to test four triple stores, with the largest containing 153,737,776 triples.

Finally, Zhang et al. (2012) presented SRBench to evaluate three streaming engines. It makes use of the real-world datasets DBpedia, LinkedSensorData[28], and GeoNames[29]. Three streaming engines were evaluated using 17 queries with the largest dataset used containing 1,730,284,735 triples.

[28]`http://wiki.knoesis.org/index.php/LinkedSensorData`
[29]`http://www.geonames.org/ontology/`

## 2.7 Summary

In this chapter, we first described the Semantic Web, reason maintenance, and provenance. We detailed the benefits of provenance and described some previous work which incorporated provenance on the Semantic Web.

In the next chapter, we introduce RGPROV, a specialisation of PROV which supports the tracking of the provenance of RDF graphs on the Semantic Web and facilitates the propagation of their modification.

# Chapter 3

# RGPROV: A Vocabulary for RDF Graph Provenance

The Semantic Web promotes the sharing and reuse of information (Berners-Lee et al., 2001). Information on the Semantic Web is presented using RDF and OWL and queried and updated using SPARQL. An RDF graph is reasoned on resulting in the inference of new triples. A graph which is created as a result of reasoning on another graph is an *entailed* graph. Graphs may be manually created or they may be formed by combining information from other graphs, i.e. source graphs. A graph which is created using other graphs is a *derived* graph. Along those lines, a entailed graph is also regarded as a derived graph.

Provenance describes how a thing or a piece of data was produced to promote, among others, replication, explanation, and understanding. PROV-O is a lightweight ontology that is a W3C recommendation to represent provenance information (Lebo et al., 2013).

In order to track the provenance of derived RDF graphs on the Semantic Web and facilitate the propagation of their modification, we propose a specialisation of the PROV-O ontology that models the classes and properties involved in such a graph's creation and update. The proposed vocabulary, RGPROV, allows the specific capture of the provenance of a graph that is both derived from other - source - graphs and has undergone entailment. This vocabulary also allows us to reflect changes made to its source graph without wide scale re-derivation, i.e. recreation and then re-reasoning, and to capture the provenance of the update precisely.

We begin this chapter with a scenario whose aim is to provide a concrete example in which a consumer of published data is affected by changes to that data and needs to keep their cached data up-to-date. Following the scenario, in Section 3.2, we present a summary of the notations used in this chapter and throughout the rest of the report. In Section 3.3, we present a hypothetical running example that will be used for explanation and demonstration throughout this chapter and the rest of the report. In Section 3.4, we demonstrate how, when a graph is fetched from an outside system to another system that uses it, its retrieved provenance graph has to be updated to reflect the act of retrieval. In Section 3.5, we showcase how a graph is created in our running by from different set theoretic operations following by RDFS entailment. In Section 3.6, we present the proposed vocabulary RGPROV. Finally, in Section 3.7, we summarise this chapter.

## 3.1 Scenario

*Charlie, a researcher at Poppleton University, is interested in knowing what books in her research field are available at the University of Maximegalon Library and at the University of Northern New Jersey Library. Both libraries publish their datasets online in the form of RDF graphs, and so Charlie can easily retrieve both lists of books. Being a conscientious researcher who is aware of the value of the provenance of her data, Charlie also retrieves the provenance of both graphs alongside them. After retrieving the two RDF graphs, and assuming that both of them rely on the same bibliography ontology to describe publications, Charlie can combine them to find all the books that could be found in either library. Also, she may compare the graphs to find out which books can be found in both libraries. Moreover, she may be interested in finding out which books can be found in one library but not the other. After the graphs have been combined, compared, or contrasted, Charlie also reasons on the results to infer implicit information from the asserted data and saves those results locally for quick access. She might also publish those graphs for anyone with shared interests for them to access and copy. This allows another researcher at Springfield University, Drew, to retrieve any of Charlie's graphs and use them. We show the production of Charlie's graph in Figure 3.1 and the interactions between the systems in Figure 3.2.*

*Later on, the University of Maximegalon Library adds new books to its collection and changes its published graph to reflect this addition. It could also happen that some books go on loan to another library at a different university and can no longer be found at the University of Maximegalon Library. Any of these updates to the library's graph results in Charlie's graphs becoming stale. Because she wants them to stay up-to-date and error-free, Charlie's graphs now need to incorporate these changes. Thus, Charlie has no choice but to resort to recreating them from scratch by re-retrieving the University of Northern New jersey Library's graph, retrieving the University of Maximegalon Library's new graph, re-comparing or re-contrasting, and finally re-reasoning. Once she has done so, Drew can also retrieve the updated graph if he so wishes.*

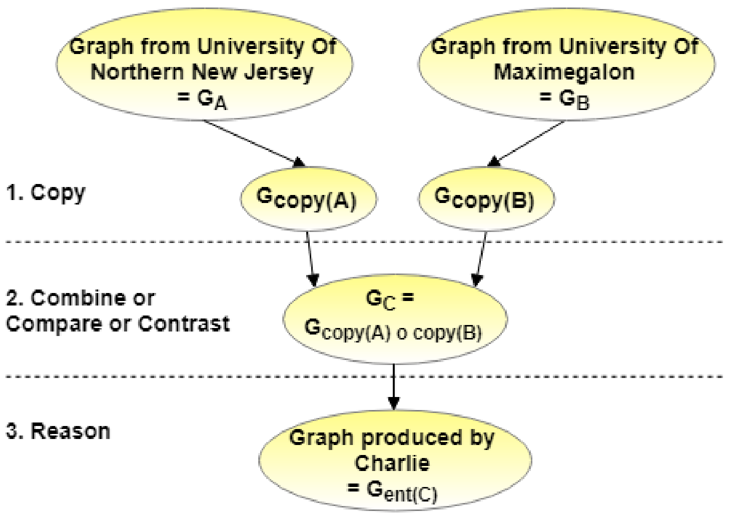

Graph from University Of Northern New Jersey = G_A
Graph from University Of Maximegalon = G_B
1. Copy
Gcopy(A)
Gcopy(B)
2. Combine or Compare or Contrast
G_C = G_copy(A) o copy(B)
3. Reason
Graph produced by Charlie = G_ent(C)

Figure 3.1: Production of Charlie's Graph from the Graphs Published by the Two Universities.

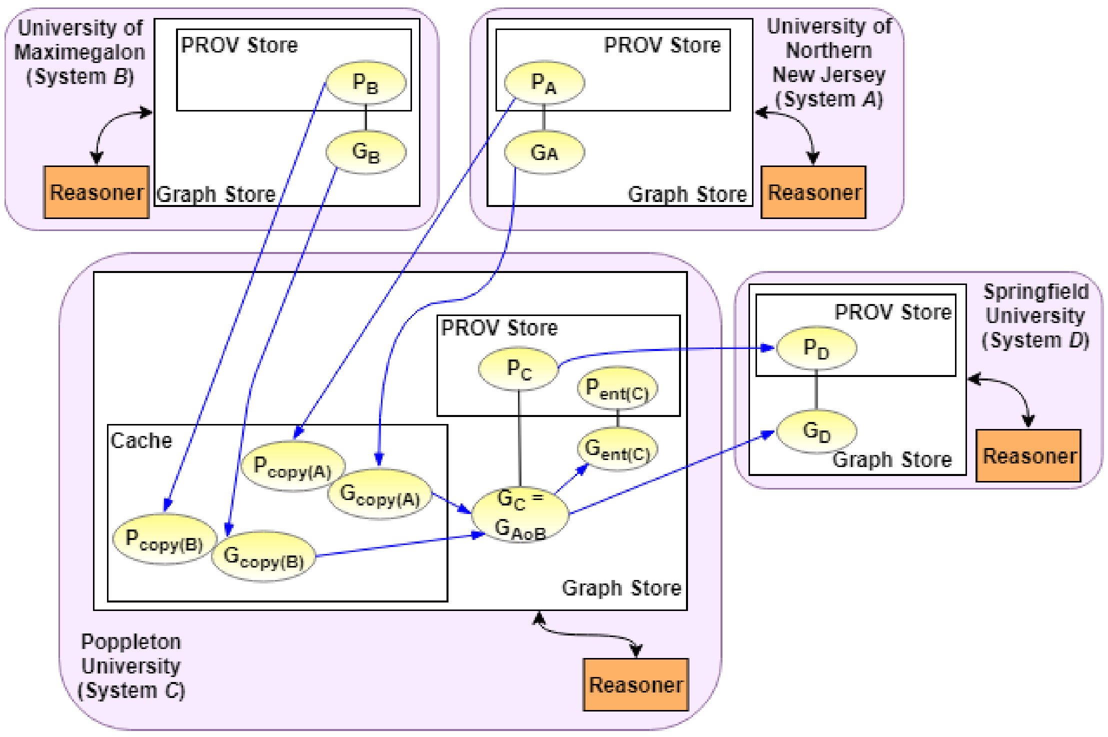


Figure 3.2: Interactions Between Systems in Scenario.

| **Notation** | **Usage** |
|---|---|
| $G_A$ | The graph $A$ |
| $P_A$ | The provenance graph of $G_A$ |
| $G_{copy(A)}$ | A copy of graph $A$ |
| $P_{copy(A)}$ | The provenance graph of $G_{copy(A)}$ $P_{copy(A)}$ is made up of $P_A$ plus additional information describing the copying of $G_A$ which produced $G_{copy(A)}$ |
| $ST_\circ$ | The set theoretic operation consisting of applying the operator $\circ$ (where $\circ$ is one of: $\cup$, $\cap$, $\setminus$) |
| $G_{A\circ B}$ | The graph resulting from applying $ST_\circ$ to graphs $G_A$ and $G_B$ |
| | $P_{A\circ B}$ is made up of $P_A \cup P_B$ plus additional information describing the set theoretic operation which produced $G_{A\circ B}$ |
| $G_{inf(A)}$ | The graph containing the triples inferred by the triples in $G_A$ |
| $G_{ent(A)}$ | The graph entailed by $G_A$, such that: $G_{ent(A)} \equiv G_A \cup G_{inf(A)}$, and $G_A \cap G_{inf(A)} = \phi$ |
| $P_{ent(A)}$ | The provenance graph of $G_{ent(A)}$ $P_{ent(A)}$ is made up of $P_A$ plus additional information describing the entailment operation performed on $G_A$ |
| $Up_A$ | The update operation consisting of inserting into or deleting from graph $A$ triples |
| $G_A^{up}$ | The graph containing the triples to be added to or deleted from $G_A$ |
| $P_A^{up}$ | The provenance graph of $G_A^{up}$ |
| $G_{A'}$ | An updated version of graph $G_A$: $G_{A'} \equiv G_A \cup G_A^{up}$, or $G_{A'} \equiv G_A \setminus G_A^{up}$ |
| $G_{copy(A)}^{up}$ | A copy of graph $G_A^{up}$ |
| $P_{copy(A)}^{up}$ | The provenance graph of $G_{copy(A)}^{up}$ $P_{copy(A)}^{up}$ is made up of $P_A^{up}$ plus additional information describing the copying of $G_A^{up}$ which produced $G_{copy(A)}^{up}$ |
| $G_{sub(copy(A))}^{up}$ | A subset graph of $G_{copy(A)}^{up}$ |

Table 3.1: Summary of Notations

## 3.2 Summary of Notations

To provide a convenient single point of reference for the notations used in this chapter and throughout the rest of the report, we list them in Table 3.1.

## 3.3 Running Example

We assume there are four systems $A$, $B$, $C$, and $D$, corresponding to the University of Northern New Jersey, the University of Maximegalon, Poppleton University, and Springfield University respectively. Each system has ownership of some RDF graphs and maintains their provenance graphs. These provenance graphs are of no direct importance to us, because despite the many benefits of provenance, such as trust, attribution, transparency, etc., we do not make use of the actual metadata they contain. We focus on one graph, $G_{ent(C)}$, produced by system $C$ as follows.

System $C$, as per Charlie's setup, had first retrieved a copy of graph $G_A$ from system $A$ and a copy of graph $G_B$ from system $B$ and stored them internally as $G_{copy(A)}$ and $G_{copy(B)}$ respectively. It had also retrieved their provenance graphs $P_A$ and $P_B$ and updated their copies internally to document how $G_A$ and $G_B$ had been retrieved. It had stored those updated provenance graphs internally as $P_{copy(A)}$ and $P_{copy(B)}$. At this stage, $C$ had started to document the provenance of $G_{ent(C)}$. It does so by creating a new provenance graph $P_{ent(C)}$ by copying the contents of $P_{copy(A)}$ and $P_{copy(B)}$ into it. Note that original contents of the provenance graphs $P_{copy(A)}$ and $P_{copy(B)}$ is irrelevant to $C$ because $C$'s only concern was to document the act of the retrieval of the source graphs. Therefore, because it was only concerned with operations on the graph level, $C$ was not concerned with the provenance of each triple in graphs $G_A$ or $G_B$. Accordingly, even though graphs $P_{copy(A)}$ and $P_{copy(B)}$ had been included as parts of graph $G_{ent(C)}$'s own provenance graph $P_{ent(C)}$, they could then have been discarded or kept for bookkeeping purposes only.

$C$ had then applied $ST_\circ$, one of three possible graph set theoretic operations on them, union, intersection, or difference, on graphs $G_{copy(A)}$ and $G_{copy(B)}$. This resulted in the graph $G_{A \circ B}$, referred to as $G_C$. While there are other possible graph operations that could have been performed on either or both $G_{copy(A)}$ and $G_{copy(B)}$, such as join, construct, and the use of optional values and filter constraints, we restrict the graph operations to the more common binary set theoretic graph operations being union, intersection, or difference. Like at any stage, provenance would have been recorded at the graph level and the specific set operation which produced $G_C$ was indicated in the provenance graph $P_{ent(C)}$ of graph $G_{ent(C)}$.

Finally, $C$ ran graph $G_C$ through a reasoner, producing with graph $G_{ent(C)}$. The provenance graph $P_{ent(C)}$ was updated to indicate that $G_C$ entailed $G_{ent(C)}$. Moreover, we assume that system $D$ has used a copy of graph $G_C$ to produce its graphs $G_D$ and $G_{ent(D)}$. The production of $G_{ent(C)}$ is shown in Figure 3.3 and in more detail in Figure 3.4; note the change is arrow direction to comply with PROV. Note that the grey-shaded entities are the copies of the provenance graphs of the source graphs.

Now, assume that system $B$ performs a SPARQL update operation, namely $Up_B$, on graph $G_B$ by either inserting into it or deleting from it the triples in the graph $G_B^{up}$. This results in graph $G_B$ becoming the new graph $G_{B'}$. We also assume that $B$ additionally creates a new provenance graph $P_B^{up}$ containing the metadata describing $G_B^{up}$. The new provenance graph $P_{B'}$, is $G_{B'}$'s new provenance graph. $P_{B'}$ contains $P_B^{up}$ as well as whatever information contained in $P_B$ that is deemed to still be relevant.

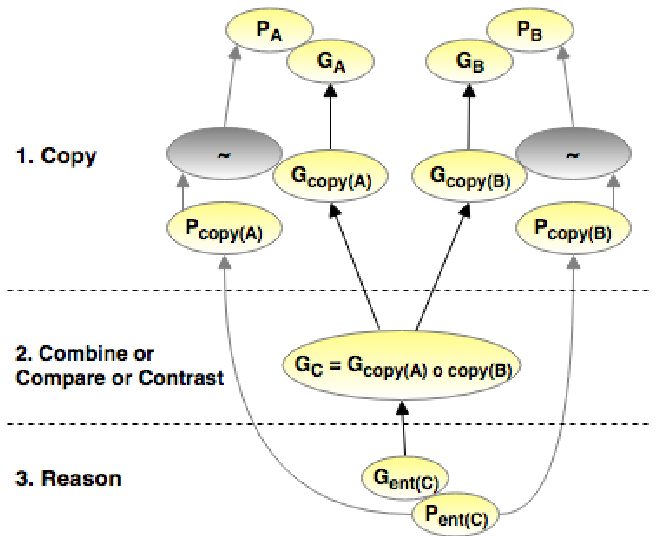


Figure 3.3: Production of $G_{ent(C)}$ from $G_A$ and $G_B$.

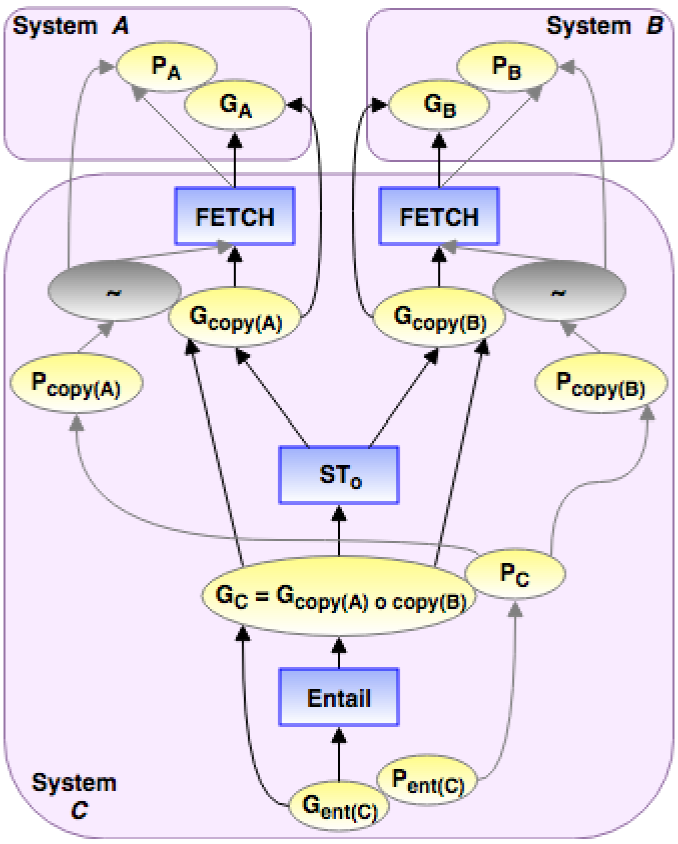


Figure 3.4: Detailed Production of $G_{ent(C)}$ and its Provenance Graph from $G_A$, $G_B$, and their Provenance Graphs.

$C$ should know about this update and subsequently needs to update $G_{ent(C)}$, or whichever parts of it should be affected, thus resulting in the more accurate and up-to-date $G_{ent(C')}$. The standard approach to this problem is to first retrieve a copy of $G_{B'}$, namely $G_{copy(B')}$, as well as another copy of $G_A$ - if $G_{copy(A)}$ had not been kept in storage. Then, to reapply the graph operation $ST_{\circ}$ on $G_{copy(A)}$ and $G_{copy(B')}$ to produce $G_{copy(A) \circ copy(B')}$, i.e. $G_{C'}$, before finally re-entailing to produce $G_{ent(C')}$. This becomes impractical in large systems, especially on the Semantic Web for two reasons. The first is that it is computationally expensive to re-entail the graph from scratch every time there is an update, especially if the graphs are large in size. The second reason is that it requires additional storage, communication overhead, or both since $C$ needs either to store $G_{copy(A)}$, $G_{copy(B)}$, and eventually $G_{copy(B')}$, or re-fetch each of them whenever a change occurs. Thus, we identify the need for a more efficient way to reflect updates and to produce $G_{ent(C')}$.

We propose an approach that considers the set theoretic operation that originally created the graph of interest along with the type of update which has been applied to the source graph to see whether part or all of the source graphs need to be re-retrieved as well as which parts of the graph of interest need to be updated and re-entailed.

## 3.4 Graph Retrieval

Source graphs and their provenance graphs are retrieved from systems that publish them. According to the W3C's best practices recommendation for data on the Web, metadata - the provenance in this case - may be embedded in the published data or provided in a separate resource using standard serialisation formats (Lóscio et al., 2017). Since we are building on the PROV-O ontology, a machine-readable format for presenting provenance, we have opted for the latter approach of separating a graph from its provenance into two different resources. A graph links to its provenance using the link relation `describedby`, as per the W3C Linked Data platform recommendation (Speicher et al., 2015).

We first demonstrate the act of a single graph retrieval and then expand it to our running example.

### 3.4.1 Single Graph Retrieval

When a graph $G$ is retrieved from another system, it is fetched along with its provenance graph $P_G$. $G$ and $P_G$ are two separate entities that may be retrieved in two separate fetch processes or together in a single fetch process. For the sake of brevity, we assume they are retrieved in one fetch process. This fetch process handles the dereferencing of the value of the graph's Linked Data relation `describedby` to its provenance graph and copying the latter. The copied provenance graph is then updated with additional

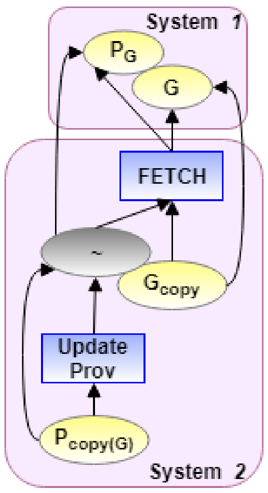


Figure 3.5: Retrieval of a Graph and Its Provenance Graph.

statements added to it to describe which process copied $G$ and when and from where $G$ was copied; thus producing the provenance graph $P_{copy(G)}$.

In accordance with PROV, the copy $G_{copy}$ is considered an entity separate from $G$. Although $P_G$ can be looked as provenance of provenance, i.e. $P_G$ can be looked at as provenance of $P_{copy(G)}$, we will only be considering $P_G$ in the context of it being the provenance of graph $G$.

The retrieval of a single graph and its provenance is shown in Figure 3.5. The grey-shaded entity is the copy of $P_G$ from which $P_{copy(G)}$ is produced.

### 3.4.2 Graph Retrieval in Running Example

There will be two fetch operations performed by $C$ to fetch $G_A$ and its provenance graph $P_A$ from $A$ and $G_B$ and its provenance graph $P_B$ from $B$. These operations will be followed by two other operations which will be responsible for producing $P_{copy(A)}$ and $P_{copy(B)}$ by adding the extra information described above. This is shown in Figure 3.6. Note that the grey-shaded entity is the copy of $P_G$.

Note that we are only interested in keeping track of the provenance of the source graph; we are not interested in the detailed bookkeeping of the provenance of the provenance graph. Therefore, from this point onwards, we will consider retrieving and updating the provenance of a source graph as part of fetching it. This summarised production of $P_{copy(A)}$ and $P_{copy(B)}$ is shown in Figure 3.7.

## 3.5 Graph Operations in Running Example

We split the types of graph operations in system $C$ into three as follows.

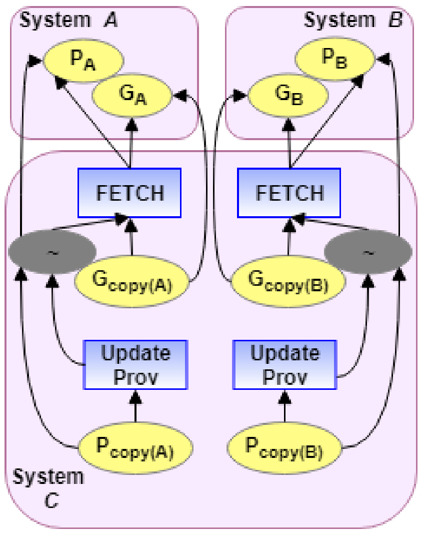


Figure 3.6: Retrieval of a Graphs $G_A$ and $G_B$ and their Provenance Graphs.

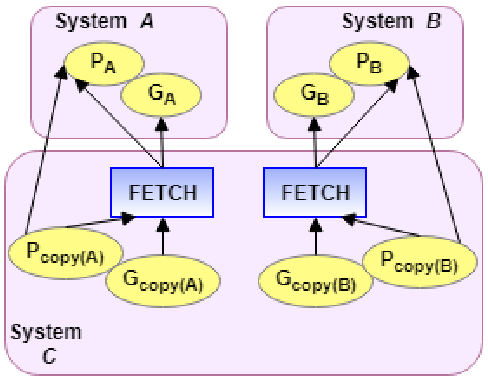


Figure 3.7: Retrieval of a Graphs $G_A$ and $G_B$ and their Provenance Graphs.

The first includes the operations that produce graph $G_C$ from $G_{copy(A)}$ and $G_{copy(A)}$, namely the set theoretic operations previously introduced in Section 2.1.3.1. While there are other possible graph operations that could have been performed on either or both $G_{copy(A)}$ and $G_{copy(B)}$, such as join, construct, and the use of optional values and filter constraints, those are out of scope of this report. Thus, we restrict the graph operations to the more common binary set theoretic graph operations being union, intersection, or difference.

The second includes the operations that produce $G_{ent(C)}$ from $G_C$, namely entailment, previously discussed in Section 2.1.3.2. However, we bring attention to the three different scenarios that may occur when requesting a graph from another system. A system may make available the source graph, the graph entailed from the source graph or both. In the case of the system making available its source graph, its handling by the system which has retrieved it should be straightforward as any entailment will be done by the system operating on the graph. However, in the case where an entailed graph is returned, this may cause inconsistencies if the entailment regime used to produce it was different from the entailment regimes used by the other systems retrieving and operating on it. For the sake of simplicity, we allow all concerned systems to use their preferred entailment regimes internally but assume that the graphs they publish contain only base triples, i.e. those that are asserted and not inferred. In our running example, when $C$ runs graph $G_C$ through a reasoner, we assume the use of RDFS entailment.

The third includes the operations that are applied to $G_C$ to update it, namely SPARQL update queries, previously discussed in Section 2.1.2.4.

Moreover, recall that when serialised, a blank node is given a locally scoped identifier, or blank node identifier, which is not portable outside the systems it is defined in (Klyne et al., 2014). Thus, without recourse to OWL reasoning using functional properties, inverse functional properties, or keys, blank nodes cannot be verified to be equal. Therefore, we only make use of ground graphs. Ground graphs are those that do not contain any blank nodes (Hayes and Patel-Schneider, 2014).

### 3.5.1 Initial Graph Creation

The first two graph operations are presented below:

**Union:** $G_C = G_{copy(A) \cup copy(B)}$.
Note that the size of $G_C$ will be larger than or equal to that of $G_{copy(A)}$ and $G_{copy(B)}$ combined, i.e. $|G_C| \leq |G_{copy(A)}| + |G_{copy(B)}|$

**Merging:** since merging is identical to union in case of the absence of blank nodes, and since we are ignoring blank nodes, then the merge operation will be treated as a union operation and henceforth not considered a separate operation from it.

**Intersection:** $G_C = G_{copy(A) \cap copy(B)}$.
Note that the size of $G_C$ will be smaller than or equal to that of $G_{copy(A)}$ or $G_{copy(B)}$, i.e. $|G_C| \leq min(|G_{copy(A)}|, |G_{copy(B)}|)$.

**Difference:**

1. $G_C = G_{copy(A) \setminus copy(B)}$.
Note that the size of $G_C$ will be smaller than or equal to that of $G_{copy(A)}$, i.e. $|G_C| \leq |G_{copy(A)}|$.

2. $G_C = G_{copy(B) \setminus copy(A)}$.
Note that the size of $G_C$ will be smaller than or equal to that of $G_{copy(B)}$, i.e. $|G_{G_C}| \leq |G_{copy(B)}|$.

**Entailment:** $G_{ent(C)} = G_C \cup G_{inf(C)}$, where $G_{inf(C)}$ contains the inferred triples and $G_C \cap G_{inf(C)} = \phi$.

The production of graph $G_{ent(C)}$ from the above operations is displayed in Figures 3.8 and 3.9. Note that, for the sake of better clarity, the production of the provenance graph $P_{ent(C)}$ is left out of the figure.

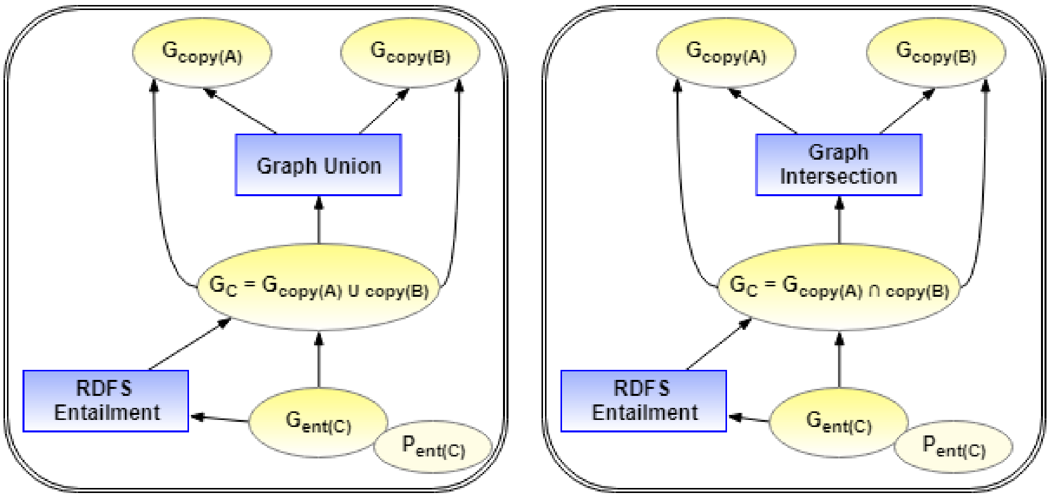


Figure 3.8: Production of $G_{ent(C)}$ from Union and Intersection.

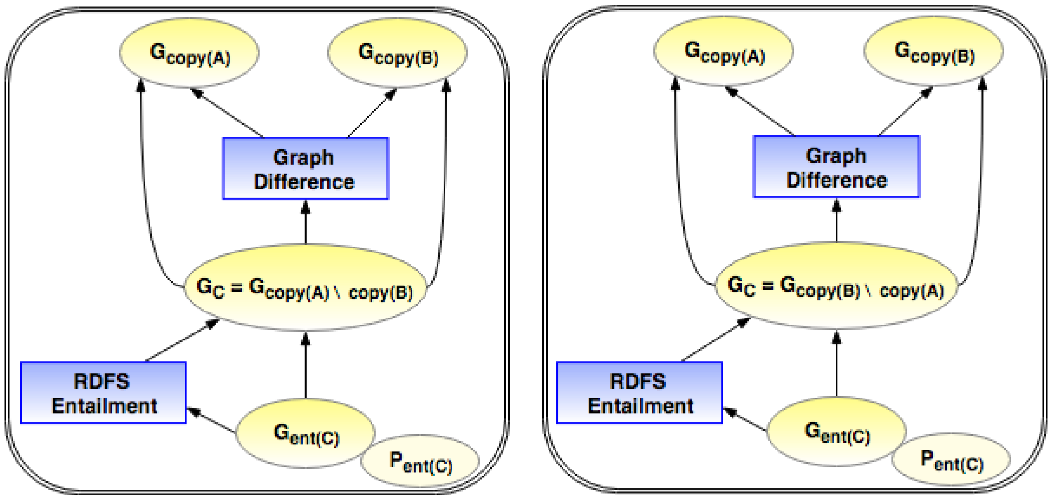


Figure 3.9: Production of $G_{ent(C)}$ from Difference.

### 3.5.2 Graph Updates

Recall that graph update operations are those that alter existing graphs. There are five fundamental operations as follows.

1. Insert data, this results in one or more triples being added to a graph.
2. Delete data, this results in one or more triples being removed from a graph.
3. Delete/Insert, this is equivalent to a sequence of the above two operations.
4. Load, this results in inserting into a graph all the triples that are present in another graph. It may be treated it as being equivalent to applying an Insert operation for each triple in the other graph.
5. Clear, this results in removing all the triples that are present in a graph. As it does not require the subsequent removal of an empty graph (although some

implementations may do so), it may be treated as being equivalent to applying a Delete operation on every triple in the graph.

Thus, we will be focusing only on Insert and Delete as the rest are special cases of the two. We discuss the effects that applying these two update operations on a source graph have on a graph that was created using the sources graph in the next chapter.

## 3.6 The RGPROV Vocabulary

As previously stated, PROV is domain-agnostic, however, it is equipped with extensibility points to provide more expressive capabilities by including domain-specific information. While we chose to extend PROV for such purposes using subtyping, other approaches are viable. Nevertheless, we chose to create subtypes because there is a clear distinction between the subclasses of our graph operations domain, as well as not to have to use the PROV classes excessively and to enable us to construct shorter and clearer queries.

Thus, in this section, we present the vocabulary RGPROV that we propose to represent the provenance of RDF graphs. For it to be as light-weight as possible, its granularity is at the graph level and not at the triple level. Thus, it captures the minimal provenance needed to pinpoint how a derived graph was created from its source graphs and allows for its replication.

RGPROV extends PROV-O and has the namespace prefix rgprov. Where convenient, we use the classes and properties of RDF, RDFS, and PROV in the W3C recommendations. The vocabulary can also be used for graphs encoding OWL but since we are presently utilising RDFS entailment, we are only using it for RDF graphs. We validated RGPROV using HermiT 1.3.8, a reasoner tool provided in Protégé. No inconsistencies or insatiabilities were found.

### 3.6.1 Vocabulary Extensions

In accordance with PROV, we recognize that RDF, OWL, and provenance graphs are entities. In order to differentiate them from other types of entities, we introduce the class rgprov:Graph as a subclass of prov:Entity that has as its members only those entities that are graphs. The actions that retrieve rgprov:Graph, produce them, or operate on them are activities. The initiators of those actions are agents. We extend these concepts along with any necessary properties as follows.

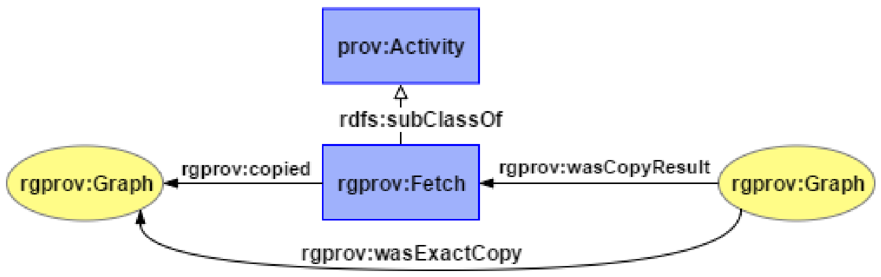


Figure 3.10: RGPROV Components for Graph Retrieval.

#### 3.6.1.1 Vocabulary for Graph Retrieval

PROV provides the properties prov:hadPrimarySource and prov:wasQuotedFrom to express the repetition of an entity or part of it. As we are dealing with copying graphs as-is from their sources, we require stricter terms. We introduce the following:

- rgprov:Fetch, a subclass of prov:Activity that indicates that a fetch operation has taken place.
- rgprov:wasExactCopy, a subproperty of prov:wasQuotedFrom that indicates that a graph was an exact replica of another. Its domain is a rgprov:Graph and its range is also rgprov:Graph.
- rgprov:copied, a subproperty of prov:Used that expresses the action of fetching a copy of a graph. Its domain is a rgprov:Fetch and its range is a rgprov:Graph.
- rgprov:wasCopyResult, a subproperty of prov:wasGeneratedBy that indicates that a graph was the result of a copy (fetch) action. Its domain is a rgprov:Graph and its range is a rgprov:Fetch.

The above terms are shown in Figure 3.10.

We see no need to create additional vocabulary for provenance production and updating.

#### 3.6.1.2 Vocabulary for Graph Operations

We introduce the class rgprov:GraphOperation, a subclass of prov:Activity, that encompasses operations performed on a graph.

**Vocabulary for Set theoretic Operations** Because, as will be discussed in detail in Section 4.2.3, there is a need to keep track of which graph operation produced a graph, we introduce the following:

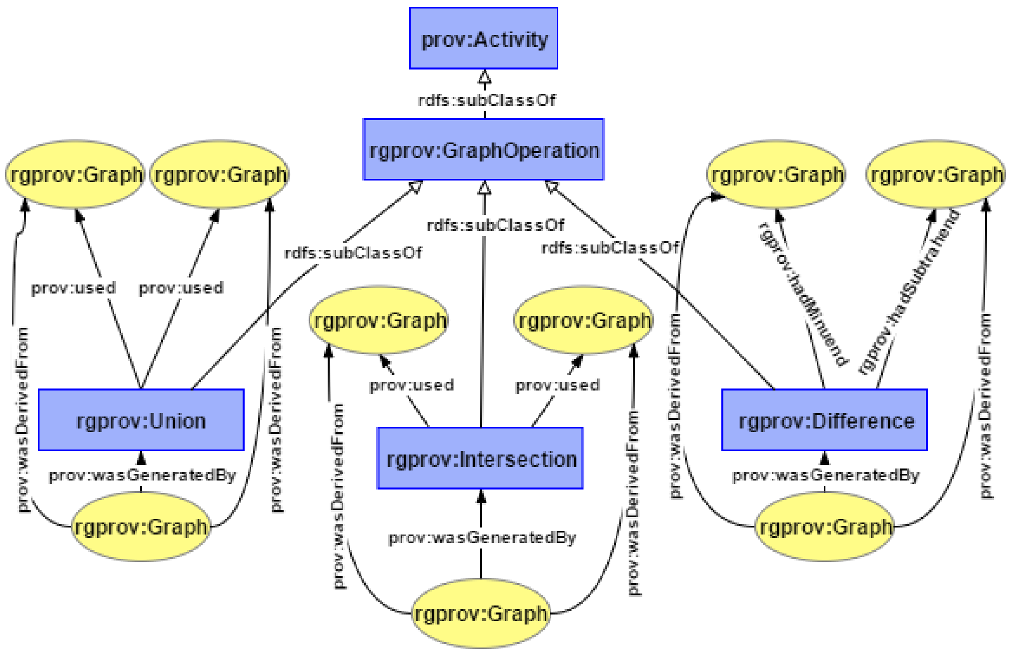


Figure 3.11: RGPROV Components for Set Theoretic Graph Operations.

- rgprov:Union, a subclass of rgprov:GraphOperation that indicates that a union operation has taken place.
- rgprov:Intersection, a subclass of rgprov:GraphOperation that indicates that an intersection operation has taken place.
- rgprov:Difference, a subclass of rgprov:GraphOperation that indicates that a difference operation has taken place.
- rgprov:hadMinuend, a subproperty of prov:used that indicates that a graph was the first component of a graph difference action. Its domain is a rgprov:Difference and its range is an rgprov:Graph.
- rgprov:hadSubtrahend, a subproperty of prov:used that indicates that a graph was the second component of a graph difference action. Its domain is a rgprov:Difference and its range is an rgprov:Graph.

The usage of the above terms is shown in Figure 3.11.

**Vocabulary for Entailment Regimes** Since different systems may implement different entailment regimes - or use different vendor extensions for reasoners, we introduce the following:

- rgprov:Entailment, a subclass of rgprov:GraphOperation that represents an entailment.

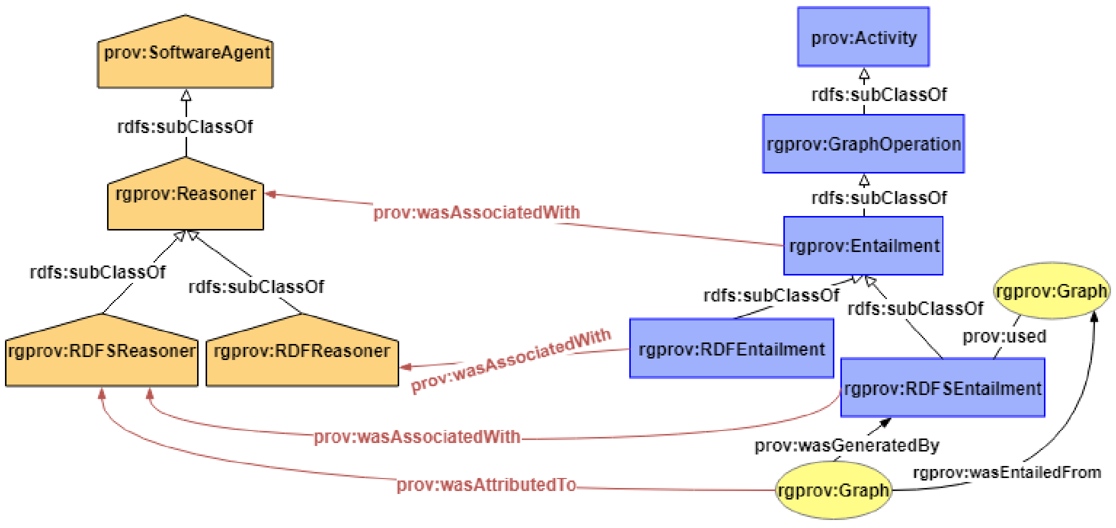


Figure 3.12: Some RGPROV Components of Entailment Regimes.

- rgprov:RDFEntailment, rgprov:RDFSEntailment, rgprov:DEntailment, rgprov:OWLRDFEntailment, rgprov:OWLDirectEntailment, and rgprov:RIFEntailment; these are subclasses of rgprov:Entailment, each indicating the type of entailment regime used.
- rgprov:Reasoner, a subclass of prov:SoftwareAgent that represent a reasoner.
- rgprov:RDFReasoner, rgprov:RDFSReasoner, rgprov:DReasoner, rgprov:OWLRDFReasoner, rgprov:OWLDirectReasoner, and rgprov:RIFReasoner; these are subclasses of rgprov:Reasoner, each indicating the type of reasoner used.
- rgprov:wasEntailedFrom, a subproperty of prov:wasDerivedFrom that represents that a graph was entailed from another. Its domain and range are rgprov:Graph.

A selection of the above terms is shown in Figure 3.12.

**Vocabulary for Updates** PROV provides the terms prov:Revision and prov:wasRevisionOf to describe that an entity has changed or has been updated. Since we differentiate the types of update operations that can be performed on a graph, we introduce the following:

- rgprov:InsertOperation a subclass of rgprov:GraphOperation that represents an insert operation.
- rgprov:DeleteOperation, a subclass of rgprov:GraphOperation that represents a delete operation.
- rgprov:UpdateGraph, a subclass of rgprov:Graph that represents the graphs whose triples are to be inserted or deleted. Although one might argue that an rgprov:UpdateGraph is merely a regular graph, and hence representing it using rgprov:Graph should be sufficient; we argue that a graph that is stored in and being

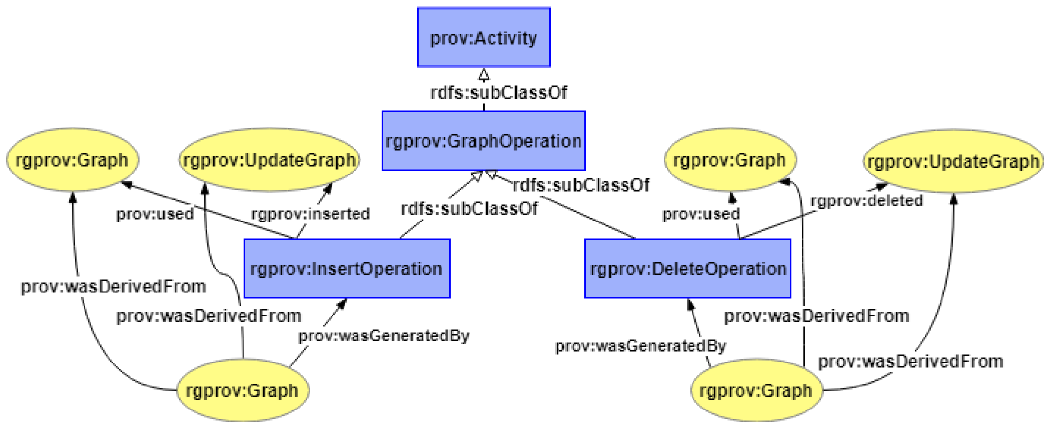


Figure 3.13: RGPROV Components of Update Operations.

used by a system should be differentiated from a graph whose entire purpose is representing triples to be inserted or deleted in the former type of graph.

- rgprov:inserted, a subproperty of prov:used that indicates that the triples of a graph were used by an insert operation performed on another graph. Its domain is rgprov:InsertOperation and its range is rgprov:UpdateGraph.
- rgprov:deleted, a subproperty of prov:used that indicates that the triples of one graph were used by a delete operation performed on another graph another. Its domain is rgprov:DeleteOperation and its range is rgprov:UpdateGraph.

The above terms are shown in Figure 3.13.

### 3.6.2 Vocabulary Usage in Running Example

We now return to the running example presented in Section 3.3 to demonstrate the usage of RGPROV. We present a concise description and postpone showing the production of provenance as we shall be applying the vocabulary in more detail in the next chapter.

First, system $C$ retrieves copies of graphs $G_A$ and $G_B$ and their provenance graphs $P_A$ and $P_B$ by making two fetch requests to $A$ and $B$ respectively. This results in it receiving $G_{copy(A)}$, $G_{copy(B)}$, and their provenance graphs. It then produces the updated provenance graphs $P_{copy(A)}$ and $P_{copy(B)}$ to reflect where the graphs came from. This is shown in Figure 3.14.

Next, $C$ applies $ST_\circ$ on $G_{copy(A)}$ and $G_{copy(B)}$ to produce graph $G_C = G_{copy(A)\circ copy(B)}$. For the sake of brevity and clarity, we will only demonstrate the union operation. This is shown in Figure 3.15.

Finally, $C$ runs $G_C$ through a reasoner to produce the entailed graph $G_{ent(C)}$. This is shown in Figure 3.16.

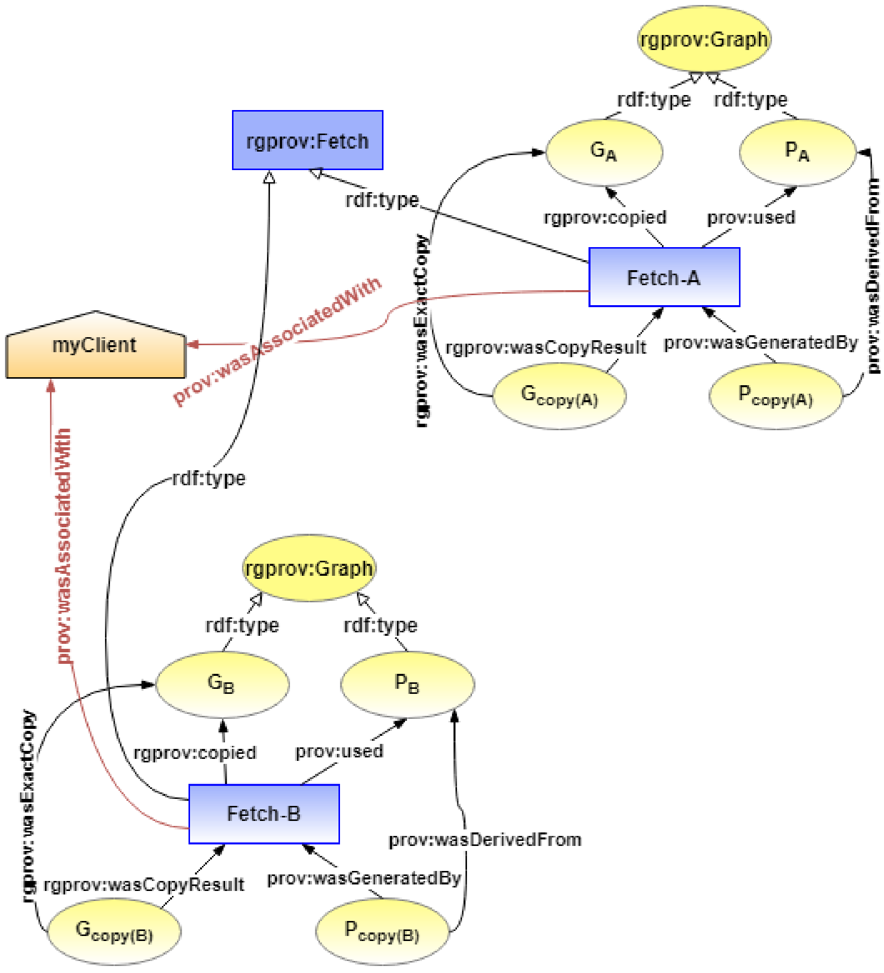


Figure 3.14: Demonstration of RGPROV Components for Graph Retrieval.

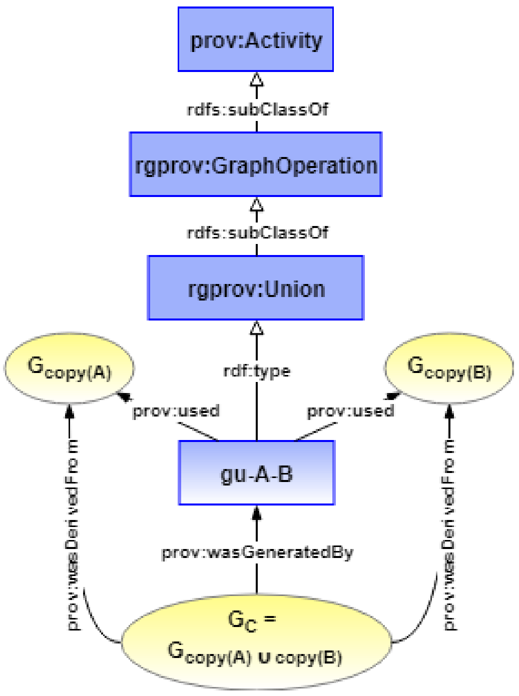


Figure 3.15: Demonstration of RGPROV Components for Union.

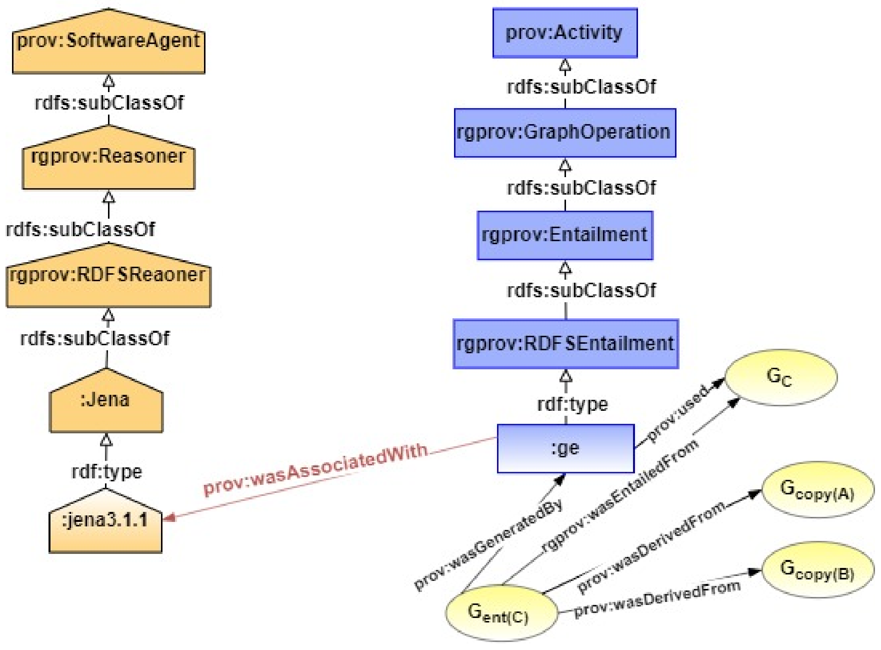


Figure 3.16: Demonstration of RGPROV Components for Entailment.

The full steps of the creation of $G_{ent(C)}$ is shown in Figure 3.17.

## 3.7 Summary

RDF graphs generated by a system can be imported by other systems and used to produce other graphs. Those source graphs are very likely be changed and updated. In order to track the provenance of derived graphs on the Semantic Web and facilitate the propagation of their modification, we presented a specialisation of the PROV-O ontology that models the classes and properties involved in a graph's creation and update. The presented vocabulary captures the provenance of the creation of graphs using other graphs wholly and allows the reflection of changes to those source graphs.

In the next chapter, we first show how RGPROV can be used to capture the provenance of graph retrieval. We then discuss the effects that updates on a source graph have on a derived graph produced using it. Finally, we show how RGPROV can be used to propagate those updates as well as to capture the provenance of graph update.

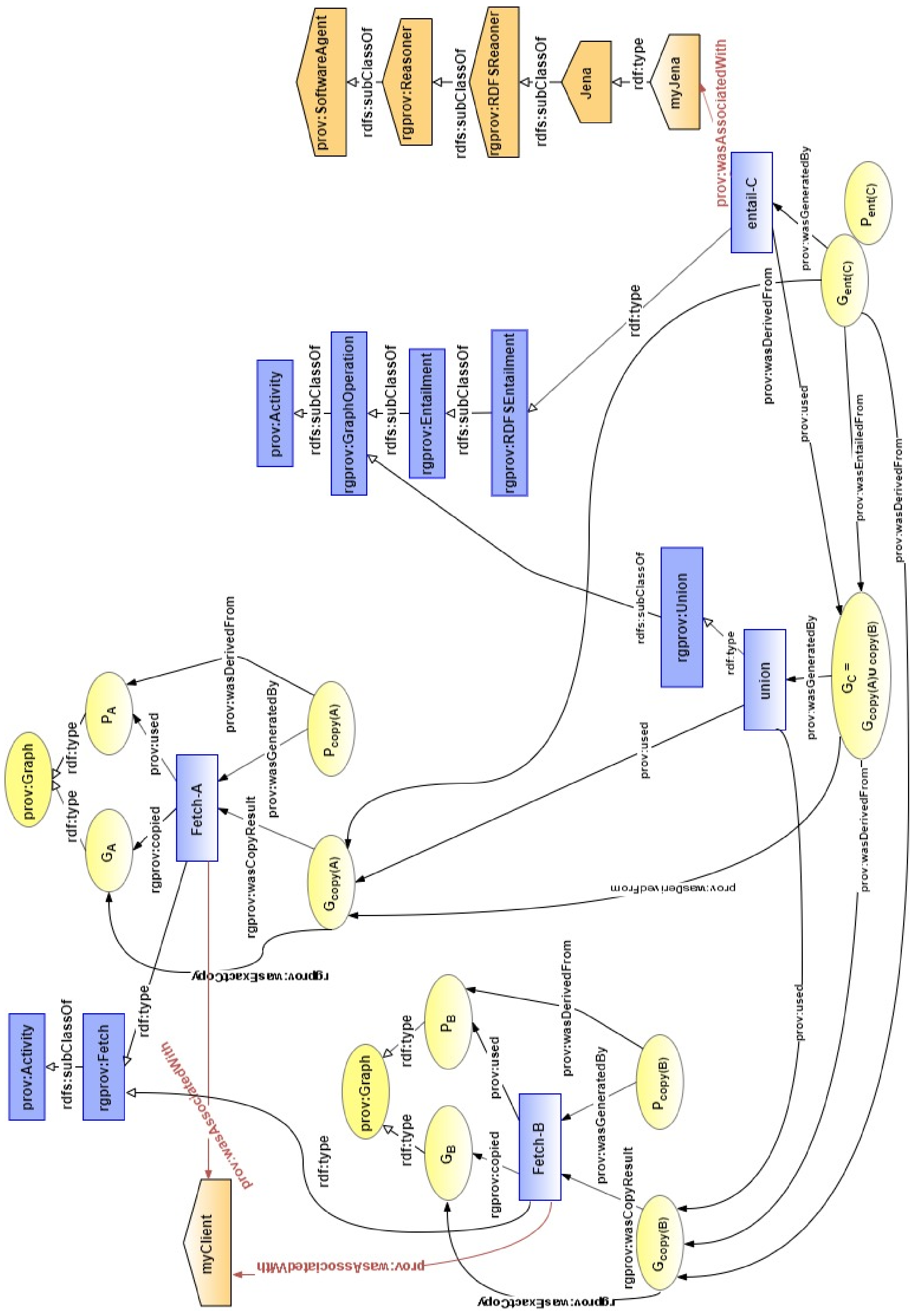


Figure 3.17: Provenance of Graph $G_C$

# Chapter 4

# Application of RGPROV

In the previous chapter, we presented a specialisation of the PROV ontology, RGPROV, that extends the provenance ontology to capture the provenance of RDF graph creation and retrieval and that also aims to facilitate the propagation of update operations on entailed graphs. It is a light-weight vocabulary and its granularity is at the graph level and not at the triple level. Thus, it captures the minimal provenance needed to pinpoint how a derived graph was created from its source graphs to allow for its replication.

In this chapter, we apply RGPROV to the running example presented in the previous chapter by showing what information documenting provenance are created through each step of the process of creating an RDF graph using two other source graphs. We also show the effect of update operations on graphs created using the set theoretic operations presented in the previous chapter and what information representing provenance are created when an update is propagated.

This chapter consists of four parts. In Section 4.1, we show how RGPROV can be used to capture the provenance of graph creation. First, we show the provenance created when the source graphs are retrieved. Second, we show the provenance created when a set theoretic operation is applied. Third, we show the provenance created when a graph is entailed from another. In Section 4.2, we shift our focus to discuss how an update on a source graph is propagated to a graph derived from it by discussing update retrieval and how each update operation affect each set theoretic operation. Then in Section 4.3, we show how RGPROV can be used to capture the provenance of propagated updates. Finally, in Section 4.4, we summarise this chapter.

## 4.1 Vocabulary for Initial Graph Creation

In this section, we return to the running example presented in the previous chapter and we describe the usage of the proposed vocabulary RGPROV during the process of creating the initial graph $G_{ent(C)}$.

### 4.1.1 Graph Retrieval

Retrieving the source graphs $G_A$ and its provenance graph $P_A$ from $A$ and $G_B$ and its provenance graph $P_B$ from $B$ is straightforward. Two fetch requests are made, each to $A$ and $B$, resulting in two replies containing the graphs $G_{copy(A)}$, $G_{copy(B)}$, and their provenance graphs. We have chosen to use RESTful Web services to illustrate the provenance triples that represent graph retrieval. Thus we introduce four new terms, which are not part of RGPROV but are local to our system, to be used throughout, as follows:

- :RESTClient, a subclass of prov:SoftwareAgent, that represents the class of all REST clients.
- :JAXRSJersey, a subclass of :RESTClient, that represents the class of all implementations of the Jersey[1] specifications.
- :jersey2.25, an instance of :JAXRSJersey, that is set up on our system.
- :retrievedFrom, a property that indicates that a process accessed a URI.

After the retrieval is successful, $C$ creates three provenance graphs: $P_{copy(A)}$, $P_{copy(B)}$, and $P_{ent(C)}$. Both $P_{copy(A)}$ are $P_{copy(B)}$ are initially created as copies of $P_A$ and $P_B$ and then more provenance information is added to them to document the process of copying graphs $G_A$ and $G_B$. This information includes the following.

- Which instance of rgprov:Fetch made the fetch call, namely :FETCH-A-YYYYMMDD and :FETCH-B-YYYYMMDD.
- That :FETCH-A-YYYYMMDD and :FETCH-B-YYYYMMDD were ran by :jersey2.25.
- The URIs accessed by :FETCH-A-YYYYMMDD and :FETCH-B-YYYYMMDD to copy the graphs.
- That :FETCH-A-YYYYMMDD and :FETCH-B-YYYYMMDD copied $G_A$ and $G_B$.
- That $G_{copy(A)}$ and $G_{copy(B)}$ were results of copying performed by :FETCH-A-YYYYMMDD and :FETCH-B-YYYYMMDD, and that they were exact copies of $G_A$ and $G_B$.
- The start and end times of :FETCH-A-YYYYMMDD and :FETCH-B-YYYYMMDD.

[1]This is a specific implementation of RESTful Web Services in Java chosen for the example only because the author is familiar with it. It is available on `https://jersey.github.io/`.

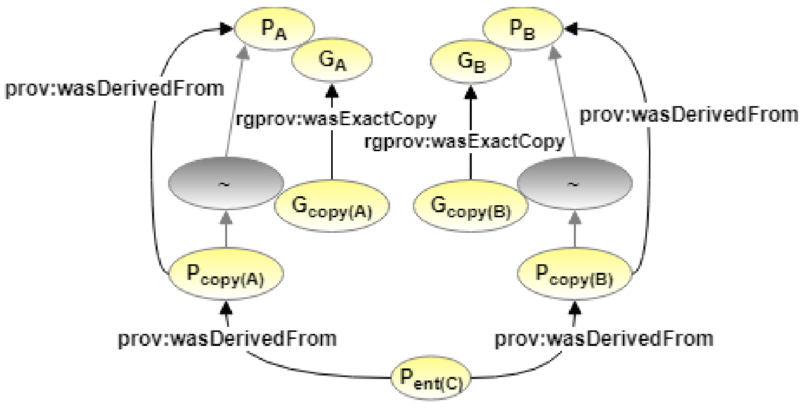


Figure 4.1: Relationships Between Source Graphs and Their Copies.

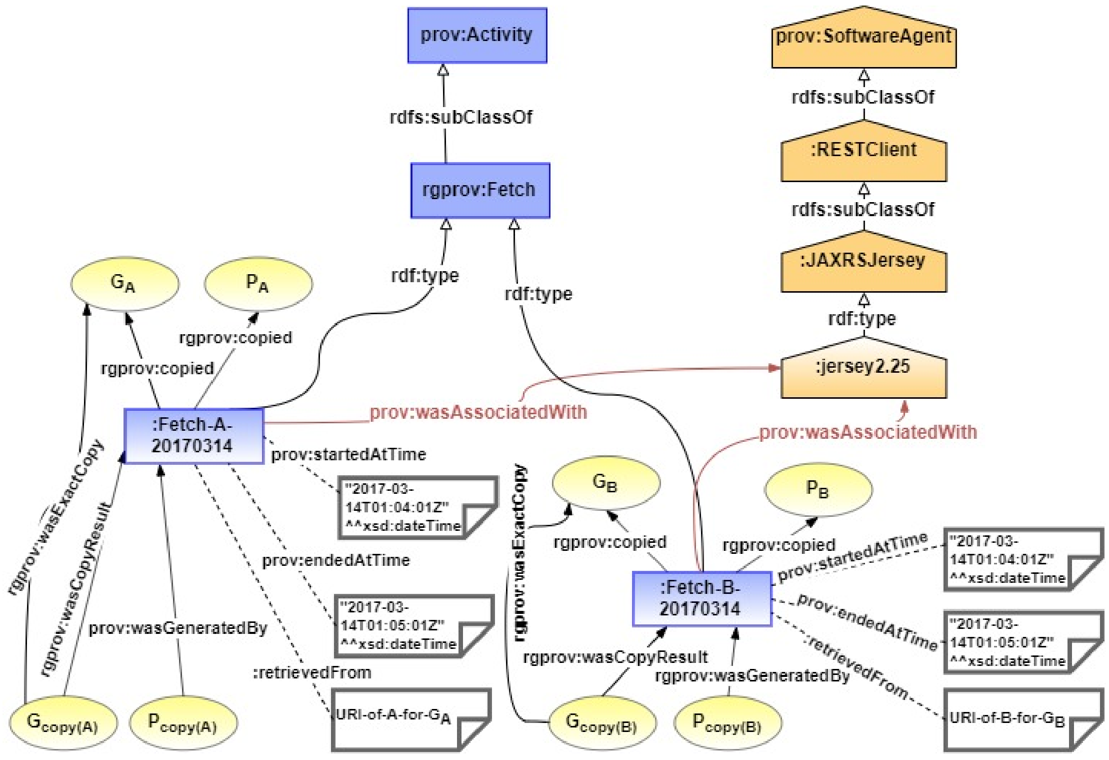


Figure 4.2: First Iteration of $P_{ent(C)}$.

The provenance graph $P_{ent(C)}$ is created by copying $P_{copy(A)}$ and $P_{copy(B)}$ into it and adding that $P_{ent(C)}$ itself was derived from both of them. The relationships between the graphs is shown in Figure 4.1. Note that the grey-shaded entities are the copies of the provenance graphs of the source graphs. This first iteration of $P_{ent(C)}$ is shown in Figure 4.2, with the triple lists detailed in Appendix B.

### 4.1.2 Graph Operations

As detailed in the previous chapter, the production of $G_{ent(C)}$ happens in two steps. The first is by applying a set theoretic operation $ST_\circ$ to the graphs $G_{copy(A)}$ and $G_{copy(B)}$,

resulting in the graph $G_C$. The second is performing an entailment operation on $G_C$ to finally produce $G_{ent(C)}$.
Since we will be using the Jena reasoner for both types of operations, we introduce the following terms, which are not part of RGPROV but are local to our system, to be used throughout:

- :Jena, a subclass of rgprov:RDFSReasoner that represents the class of Jena framework.
- :jena3.1.1, an instance of :Jena that is setup on our system.

#### 4.1.2.1 Set Theoretic Operations

The production of $G_C$ can be the result of any one of union, intersection, or difference. After graph $G_C$ is produced, the following provenance information is added to $P_{ent(C)}$:

- Which instance of rgprov:Union, rgprov:Intersection, or rgprov:Difference was invoked. For brevity, we will use the term :go-A-B-YYYYMMDD when referring to any of the former.
- That :go-A-B-YYYYMMDD was associated with :jena3.1.1.
- That :go-A-B-YYYYMMDD used both $G_{copy(A)}$ and $G_{copy(B)}$ in the case of union or intersection. In case of difference, the first component is indicated as a minuend and the second as a subtrahend.
- That :go-A-B-YYYYMMDD generated the graph $G_C$.
- The start and end times of :go-A-B-YYYYMMDD.
- That $G_{copy(A)}$ and $G_{copy(B)}$ were contributors in the creation of $G_C$.

The second iteration of $P_{ent(C)}$ is shown in Figure 4.3 with the union operation used as an example of set theoretic operations. The full triple list is detailed in Appendix B.

#### 4.1.2.2 Entailment

In the final stage of producing $G_{ent(C)}$ from $G_C$, $C$ invokes the Jena methods for RDFS entailment. After $G_{ent(C)}$ is created, $C$ adds the following information to $P_{ent(C)}$:

- Which instance of rgprov:RDFSEntailment was invoked, namely :ge-C-YYYYMMDD.
- That :ge-C-YYYYMMDD wasAssociatedWith :jena3.1.1.

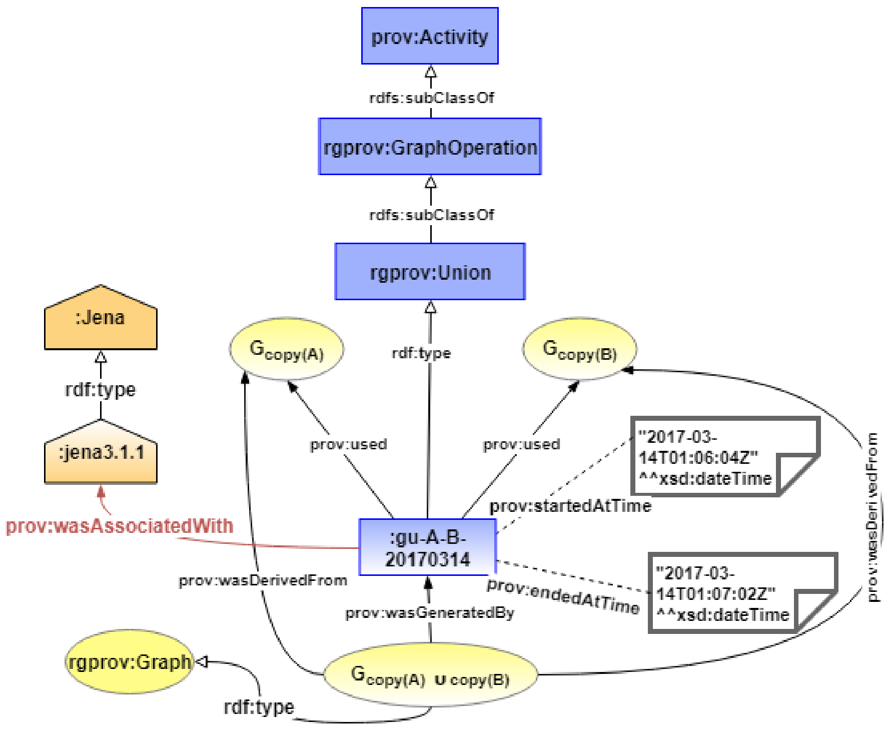


Figure 4.3: Second Iteration of $P_{ent(C)}$.

- That :ge-C-YYYYMMDD used $G_C$.
- That :ge-C-YYYYMMDD produced the graph $G_{ent(C)}$.
- That $G_C$ entailed $G_{ent(C)}$.
- That $G_{copy(A)}$ and $G_{copy(B)}$ both contributed to the creation of $G_{ent(C)}$.
- The start and end times of :ge-C-YYYYMMDD.

The above list constitutes the final additions to $P_{ent(C)}$ and is shown in Figure 4.4. The full triple list is detailed in Appendix B.

## 4.2 Graph Updates

Recall that after the update graph and its provenance graph are retrieved, $C$ queries the provenance graph $P_{ent(C)}$ to retrieve which set theoretic graph operation was used to produce $G_C$. $C$ combines this operation with the type of update that had been applied to the source graph in order to deduce whether all or part of the update graph needs to inserted into or deleted from $G_C$ to produce $G_{C'}$. Finally, $C$ produces $G_{ent(C')}$ from $G_{C'}$ via entailment.

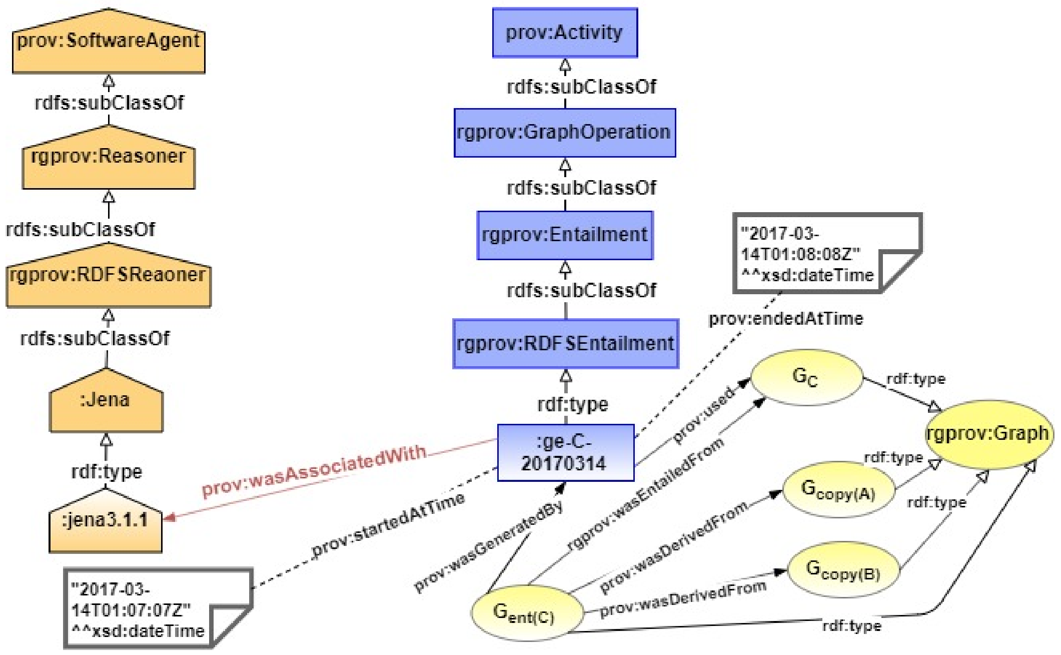


Figure 4.4: Final Iteration of $P_{ent(C)}$.

Thus, in this section, we explain how an update on a source graph is communicated to a system that uses it and how such a system should propagate the update on its graph that was created using that source graph. Those steps are summarised in Table 4.1.

### 4.2.1 Update Retrieval

Updates may reach system $C$ in two ways. First, the system that does the update, $B$ in this case, informs $C$ that an update is present. We recognise that this would force $B$ to keep a log of all the GET requests it gets from all the various systems that have queried it. To address this, $B$ may keep a log of systems that choose to be informed of updates - similar to a list of subscribers to email updates, as some systems may not wish to receive any update notifications. This results in less communication overhead for $B$ as it would need to inform less systems of the update. The second way is for $C$ to check whether a graph it uses has been updated, this may also happen in different ways. First, $C$ would periodically check if $B$ has made any updates, similar to subscribing to an RSS feed. Second, this would occur if a need arises to re-process the data it has in its system. For example, when a graph operation needs to be rerun and the source graphs have not been stored locally because of size constraints, then $C$ can check if $B$ has updated its $G_B$. Finally, if some data are time-sensitive, that is if they are relevant for some period of time and then expire. Thus, when some data are flagged as expired, then $C$ would need to request a fresh copy of the source graph.

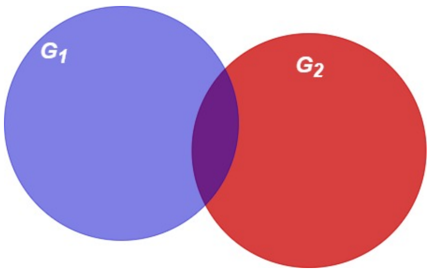


Figure 4.5: Venn Diagram of Relationships Between $G_1$ and $G_2$.

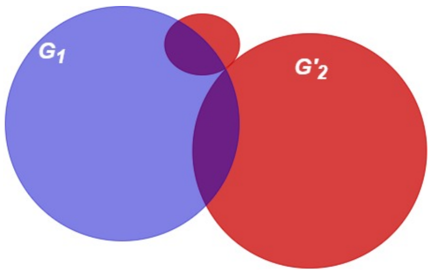


Figure 4.6: Venn Diagram of Relationships Between $G_1$ and $G'_2$, where $G'_2 = G_2 \cup G^{up}$.

### 4.2.2 Update Propagation

We study the effects of the update operations listed in Section 2.1.2.4 on the graph resulting from the set theoretic graph operations listed in Section 2.1.3.1. We assume we have two graphs, $G_1$ and $G_2$. The graph resulting from the graph operation on these two graph is $G_3$. The update operation performed on $G_2$ is $Up_{op}$ using the update graph $G^{up}$ and resulting in $G_2$ becoming the updated graph $G'_2$. A summary of this is shown in Table 4.1.

### 4.2.3 Propagation of Updates According to Set Theoretic Operations

We show the relationships between graphs $G_1$ and $G_2$ in Figure 4.5. The union $G_1 \cup G_2$ corresponds to the whole graph. The intersection $G_1 \cap G_2$ is depicted in purple. The difference $G_1 \setminus G_2$ is depicted in the blue part of $G_1$, i.e. excluding the purple portion which corresponds to the intersection. The difference $G_2 \setminus G_1$ is depicted in the red part of $G_2$, i.e. excluding the purple portion which corresponds to the intersection.

We show the relationships between graphs $G_1$ and $G'_2$ in Figure 4.6, where $G'_2$ was arrived to after performing an insert operation on $G_2$. The union $G_1 \cup G'_2$ corresponds to the whole graph. The intersection $G_1 \cap G'_2$ is depicted in purple. The difference $G_1 \setminus G'_2$ is depicted in the blue part of $G_1$, i.e. excluding the purple portion which corresponds to the intersection. The difference $G_2 \setminus G'_2$ is depicted in the red part of $G'_2$, i.e. excluding the purple portion which corresponds to the intersection.

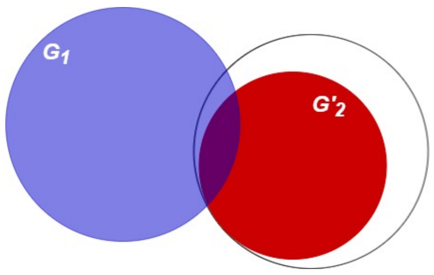


Figure 4.7: Venn Diagram of Relationships Between $G_1$ and $G'_2$, where $G'_2 = G_2 \setminus G^{up}$.

We show the relationships between graphs $G_1$ and $G'_2$ in Figure 4.7, where $G'_2$ was arrived to after performing a delete operation on $G_2$. The portion of $G_2$ which was deleted is shown in the arc between the black border and graph $G'_2$. The union $G_1 \cup G'_2$ corresponds to the whole graph, minus the white portion inside the black arc. The intersection $G_1 \cap G'_2$ is depicted in purple. The difference $G_1 \setminus G'_2$ is depicted in the blue part of $G_1$ excluding the purple portion which corresponds to the intersection. The difference $G'_2 \setminus G_1$ is depicted in the red part of $G'_2$ excluding the purple portion which corresponds to the intersection.

#### 4.2.3.1 Union

**Insert** Algebraically,

$$G'_3 = G_1 \cup G'_2 = G_1 \cup (G_2 \cup G^{up}) = (G_1 \cup G_2) \cup G^{up} = G_3 \cup G^{up}.$$

Thus, adding triples to graph $G_2$ results in adding those triples to $G_3$ if they are not already in it. This is equivalent to adding to $G_3$ the triples in $G'_2 \setminus G_3$. This can be treated as adding the triples in $[\Delta(G'_2, G_2) \setminus G_3] \equiv [G^{up} \setminus G_3]$. Since inserting already existing triples has no effect, this may be reduced to just inserting the triples in $G^{up}$.

Graphically, we see in Figure 4.6 that inserting triples to graph $G_2$ results in adding those triples to $G_3$ if they are not already in it. Thus, the union after insert corresponds to the whole graph, i.e. all the triples in both $G_1$ and $G'_2$.

The only old entity needed to be stored for this update is $G_3$. The only new entities needed for this update are the triples to be added, found in either the update graph $G^{up}$ or in the difference $\Delta$ between $G_2$ and $G'_2$.

**Delete** Algebraically,

$$G'_3 = G_1 \cup G'_2 = G_1 \cup (G_2 \setminus G^{up}) = (G_2 \setminus G^{up}) \cup G_1 = (G_2 \cup G_1) \setminus (G^{up} \setminus G_1) = G_3 \setminus (G^{up} \setminus G_1).$$

Thus, deleting triples from graph $G_2$ results in deleting triples from $G_3$ if they are not shared with $G_1$. This is equivalent to deleting from $G_3$ the triples that are in $[(G_2 \setminus G_2') \setminus G_1] \equiv [\Delta(G_2, G_2') \setminus G_1] \equiv [G^{up} \setminus G_1]$.

Graphically, we see in Figure 4.7 that the union after delete corresponds to the whole graph, i.e. all the triples in both $G_1$ and $G_2'$, minus the white portion inside the black arc, i.e. those that were deleted from $G_2$ and did not intersect those that are in $G_1$.

Unfortunately, this requires that in addition to the needed old entity $G_3$, the old entity $G_1$ is also needed. The new entities needed are the triples to be deleted, found in either the update graph $G^{up}$ or in the difference $\Delta$ between $G_2$ and $G_2'$.

#### 4.2.3.2 Intersection

**Insert** Algebraically,

$$G_3' = G_1 \cap G_2' = G_1 \cap (G_2 \cup G^{up}) = (G_1 \cap G_2) \cup (G_1 \cap G^{up}) = G_3 \cup (G_1 \cap G^{up}).$$

Thus, adding triples to graph $G_2$ results in adding those triples to $G_3$ if they are shared with $G_1$. This is equivalent to adding the triples in $[(G_1 \cap \Delta(G_2', G_2)) \setminus G_3]$, i.e, the triples in $[(G_1 \cap G_2') \setminus G_3] \equiv [(G_1 \cap G_2') \setminus (G_1 \cap G_2)] \equiv [G_2' \cap G_1] \equiv [\Delta(G_2', G_2) \cap G_1] \equiv [G^{up} \cap G_1]$.

Graphically, we see in Figure 4.6 that the intersection after insert adds only the inserted triples that are shared with $G_1$ into the purple portion.

The old entities needed for this update are $G_3$ and $G_1$ to check if the triples are shared. The new entities needed are the triples to be inserted, found in either the update graph $G^{up}$ or in the difference $\Delta$ between $G_2$ and $G_2'$.

**Delete** Algebraically,

$$G_3' = G_1 \cap G_2' = G_1 \cap (G_2 \setminus G^{up}) = (G_2 \setminus G^{up}) \cap G_1 = (G_2 \cap G_1) \setminus G^{up} = G_3 \setminus G^{up}.$$

Thus, deleting triples from graph $G_2$ results in deleting those triples in $G_3$ if they are already in it. This is equivalent to deleting the triples in $G_3 \cap \Delta(G_2, G_2')$, i.e. $[(G_1 \cap G_2) \cap (G_2 \setminus G_2')] \equiv [(G_1 \cap G_2) \cap \Delta(G_2, G_2')] \equiv [G_3 \cap G^{up}]$. Since deleting non-existent triples has no effect, this may be reduced to just deleting the triples in $G^{up}$.

Graphically, we see in Figure 4.7 that the intersection after delete removes from the purple portion all the triples that are not shared with $G_1$.

The only old entity needed to be stored for this update is $G_3$. The only new entities needed for this update are the triples to be deleted, found in either the update graph $G^{up}$ or in the difference $\Delta$ between $G_2$ and $G'_2$.

#### 4.2.3.3 Difference Case 1

This case studies $G_1 \setminus G_2$.

**Insert** Algebraically,

$$G'_3 = G_1 \setminus G'_2 = G_1 \setminus (G_2 \cup G^{up}) = (G_1 \setminus G_2) \cap (G_1 \setminus G^{up}) = G_3 \cap (G_1 \setminus G^{up}).$$

Thus, inserting triples to graph $G_2$ results in deleting those triples from $G_3$ which are now being shared with $G_1$, this may be reduced to just deleting the triples in $G^{up}$.

Graphically, we see in Figure 4.6 that the difference after insert reduces the blue portion of $G_1$ by adding those inserted triples to the intersection.

The only old entity needed to be stored for this update is $G_3$. The only new entities needed for this update are the triples to be deleted, found in either the update graph $G^{up}$ or in the difference $\Delta$ between $G_2$ and $G'_2$.

**Delete** Algebraically,

$$G'_3 = G_1 \setminus G'_2 = G_1 \setminus (G_2 \setminus G^{up}) = (G_1 \cap G^{up}) \cup (G_1 \setminus G_2) = (G_1 \cap G^{up}) \cup G_3 = G_3 \cup (G_1 \cap G^{up}).$$

Thus, deleting triples from $G_2$ results in inserting those triples in $G_3$ if they are shared with $G_1$. This is equivalent to inserting the triples in $G_1 \cap \Delta(G'_2, G_2) \equiv G_1 \cap G^{up}$.

Graphically, we see in Figure 4.7 that the difference after delete increases the blue portion of $G_1$ by including those triples that used to be shared with $G_2$.

Unfortunately, this requires that in addition to the needed old entity $G_3$, the old entity $G_1$ is also needed. The only new entities needed for this update are the triples to be deleted, found in either the update graph $G^{up}$ or in the difference $\Delta$ between $G_2$ and $G'_2$.

#### 4.2.3.4 Difference Case 2

This case studies $G_2 \setminus G_1$.

**Insert** Algebraically,

$$G'_3 = G'_2 \setminus G_1 = (G_2 \cup G^{up}) \setminus G_1.$$

Thus, inserting triples to graph $G_2$ results in inserting into $G_3$ the triples that are not shared with $G_1$. This is equivalent to inserting the triples in $G^{up} \setminus G_1$.

Graphically, we see in Figure 4.6 that the difference after insert increases the red portion of $G'_2$ by including those triples that are not shared with $G_1$.

The old entities needed for this update are $G_3$ and $G_1$ to check if the triples are shared. The new entities needed are the triples to be inserted, found in either the update graph $G^{up}$ or in the difference $\Delta$ between $G_2$ and $G'_2$.

**Delete** Algebraically,

$$G'_3 = G'_2 \setminus G_1 = (G_2 \setminus G^{up}) \setminus G_1 = G_2 \setminus (G_1 \cup G^{up}) = (G_2 \setminus G_1) \cap (G_2 \setminus G^{up}) = G_3 \cap (G_2 \setminus G^{up}).$$

Thus, deleting triples from $G_2$ results in deleting those triples from $G_3$ if they are already in it. This is equivalent to deleting the triples in $G^{up}$.

Graphically, we see in Figure 4.7 that the difference after delete reduces the red portion of $G'_2$ to the triples remaining in $G'_2$ and are not shared with $G_1$.

The only old entity needed to be stored for this update is $G_3$. The new entities needed are the triples to be deleted, found in either the update graph $G^{up}$ or in the difference $\Delta$ between $G'_2$ and $G_2$.

### 4.2.4 Re-Entailment

Re-entailment after update can be done in two ways. The first is to run the newly-produced graph through a reasoner; this is something that we aimed to avoid so as to reduce reasoning overhead caused by re-processing the whole graph. The second way is to use Gupta et al. (1993)'s and Motik et al. (2015)'s[2] approaches of only re-entailing the affected triples. To the best of our knowledge, this approach has not been applied to reasoning on the Semantic Web before, and thus constitutes a novel contribution. The approach is split in two based on whether triples are being inserted or deleted.

#### 4.2.4.1 Re-Entailment After Insert

If the update operation is an insert, then the triples are added to the graph $G_3$, before a SPARQL Describe operation is performed. This results in a new graph that contains

[2]Recall that unlike Motik et al. (2015), we do not implement our own reasoner.

the newly inserted triples as well as all the triples that relate to them, i.e. all the triples which have as a subject any of the IRIs of the described triple's subject, predicate, or object. This is the graph that is used for re-entailment. After running it through the reasoner, the new triples that have been produced are also inserted into the graph, thus resulting in the graph $G'_3$.

#### 4.2.4.2 Re-EntailmentAfter Delete

If the update operation is a delete, then it is not sufficient to only delete the triples in the update operation, as the triples that have been inferred by only those triples also need to be deleted.

Consider as an example a graph containing the following triples:

```
                  Publication rdfs:subClassOf Document        (4.1a)
            JournalArticle rdfs:subClassOf Publication        (4.1b)
     ConferenceProceedings rdfs:subClassOf Publication        (4.1c)
ConferenceArticle rdfs:subClassOf ConferenceProceedings       (4.1d)
```

The graph also contains the below triples:

```
ComputingMachineryAndIntelligence rdf:type ConferenceArticle   (4.2a)
      ComputingMachineryAndIntelligence rdf:type Publication   (4.2b)
```

Therefore, after reasoning the graph contains the following derived triples:

```
ComputingMachineryAndIntelligence rdf:type ConferenceProceedings   (4.3a)
             ComputingMachineryAndIntelligence rdf:type Document   (4.3b)
```

However, since 'Computing Machinery and Intelligence' is not a conference article, the graph needs to be corrected to reflect that. Simply deleting triple (4.2a) is not enough because the derived triple (4.3a) needs to be deleted as well.

**First Step** We perform a SPARQL Describe operation before applying the delete operation, with the triples to be deleted as its parameter. This returns a new graph that contains the triples to be deleted as well as all the other triples that relate to them.

In our example, the returned described graph contains the triples (4.1d), (4.2a), (4.2b), (4.3a), and (4.3b).

**Second Step** We then loop over this graph. For every triple $t$ to be deleted, we check the other triples for those that share the same subject and delete each of those triples, $t_{inf}$, that have been inferred by $t$. The triples are found by examining their predicates. If a triple $t$ has a predicate rdf:type, then the triples $t_{inf}$ to be deleted are those that have the same subject as $t$, a predicate rdf:type, and an object that is an rdfs:subClassOf the object of $t$. If, on the other hand, the triple $t$ has a predicate that is an rdfs:subPropertyOf another predicate, then the triples $t_{inf}$ to be deleted are those that have the same subject as $t$, the predicate that is an rdfs:subPropertyOf $t$'s predicate, and $t$'s predicate.

In our example, we loop over the returned triples and mark the triples (4.3a) and (4.3b) for deletion.

**Third Step** It is then that the delete operation is applied to $G_3$, removing the two sets of triples, the base triples - found in $G^{up}$ - to be deleted and the inferred ones.

In our example, the delete operation is now performed resulting in the removal of the triples (4.2a), (4.3a), and (4.3b)

**Fourth Step** Afterwards, we perform a second describe operation on the now-updated graph giving it as a parameter the subjects and predicates present in base triples that have been deleted. This results in a new graph that is used for re-entailment.

In our example, we request the describe of the IRIs of `ComputingMachineryAndIntelligence`, `ConferenceArticle`, `ConferenceProceedings`, and `Document`. The resulting graph will contain the triple (4.2b), which will need to be sent to the reasoner for entailment.

**Fourth Step** After running it through the reasoner, the new triples that have been produced by the reasoner are inserted; thus resulting in the graph $G'_3$.

In our example, this leads to triple (4.3b) to be re-inserted because of the presence of triple (4.2).

An additional step may take place before this last one, that of inserting new triples. If this happens, then the describe of the inserted triples is combined with the graph from the fourth step and both are sent to the reasoner. The newly inferred triples are then inserted in this fourth and final step.

A summary of the data that would be needed for fetching and applying the updates is presented in Table 4.1

| Graph Operation | Update ($Up_{op}$) | What to do | Old Needed | What to Fetch |
|---|---|---|---|---|
| $G_3 = G_{ent(G_1 \cup G_2)}$ | Insert | (*1*) Insert into $G_3$ the triples in $G^{up}$. (*2*) Re-entail the graph resulting from the describe of the triples. | $G_3$ | $G^{up} \equiv \Delta(G'_2, G_2)$. |
| | Delete | (*1*) Delete from $G_3$ the triples in $G^{up} \setminus G_1$ and those inferred by them. (*2*) Re-entail the graph resulting from the describe of the subjects and predicates. | $G_1$ and $G_3$ | $G^{up} \equiv \Delta(G_2, G'_2)$. And $G_1$ if not cached. |
| $G_3 = G_{ent(G_1 \cap G_2)}$ | Insert | (*1*) Insert into $G_3$ the triples in $G^{up} \cap G_1$. (*2*) Re-entail the graph resulting from the describe of the triples. | $G_1$ and $G_3$ | $G^{up} \equiv \Delta(G'_2, G_2)$. And $G_1$ if not cached. |
| | Delete | (*1*) Delete from $G_3$ the triples in $G^{up}$ and those inferred by them. (*2*) Re-entail the graph resulting from the describe of the subjects and predicates. | $G_3$ | $G^{up} \equiv \Delta(G'_2, G_2)$ |
| $G_3 = G_{ent(G_1 \setminus G_2)}$ | Insert | (*1*) Delete from $G_3$ the triples in $G^{up}$ and those inferred by them. (*2*) Re-entail the graph resulting from the describe of the subjects and predicates. | $G_3$ | $G^{up} \equiv \Delta(G'_2, G_2)$ |
| | Delete | (*1*) Insert into $G_3$ the triples in $G^{up} \cap G_1$. (*2*) Re-entail the graph resulting from the describe of the triples. | $G_1$ and $G_3$. | $G^{up} \equiv \Delta(G'_2, G_2)$. And $G_1$ if not cached. |
| $G_3 = G_{ent(G_2 \setminus G_1)}$ | Insert | (*1*) Insert into $G_3$ the triples in $G^{up} \setminus G_1$. (*2*) Re-entail the graph resulting from the describe of the triples. | $G_1$ and $G_3$ | $G^{up} \equiv \Delta(G'_2, G_2)$. And $G_1$ if not cached. |
| | Delete | (*1*) Delete from $G_3$ the triples in $G^{up}$ and those inferred by them. (*2*) Re-entail the graph resulting from the describe of the subjects and predicates. | $G_3$ | $G^{up} \equiv \Delta(G'_2, G_2)$. |

Table 4.1: Reflecting the Update on $G_3$ based on $ST_\circ$ performed on $G_1$ and $G_2$.

## 4.3 Vocabulary for Update Propagation

We return to the running example to demonstrate the use of RGPROV to describe how the updates get propagated. We map $G_1$ to $G_{copy(A)}$, $G_2$ to $G_{copy(B)}$, $G^{up}$ to $G^{up}_{copy(B)}$, $G'_2$ to $G_{copy(B')}$, and $G_3$ to $G_C$.

### 4.3.1 Update Retrieval

When $C$ has been informed that an update operation using the triples in $G^{up}_B$ has been applied to $G_B$, it sends a get request to retrieve $G^{up}_B$ resulting in the reply containing a copy of the update graph along with a copy of its provenance graph $P^{up}_B$. The reply that $C$ receives back from $B$ consists of a copy of $G^{up}_B$, namely $G^{up}_{copy(B)}$ and a copy of its provenance.

$C$ then creates the provenance graph $P^{up}_{copy(B)}$ similar to how it created $P_{copy(B)}$. So, $P^{up}_{copy(B)}$ contains the information in $P^{up}_B$ as well as the following:

- Which instance of rgprov:Fetch made the fetch call to system $B$, namely :Fetch-BUp-YYYYMMDD.
- That :Fetch-BUp-YYYYMMDD were ran by :jersey2.25.
- The URI accessed by :Fetch-BUp-YYYYMMDD.
- The name of the update graph that Fetch-BUp-YYYYMMDD copied, namely $G^{up}_B$.
- That the copying resulted in the new update graph $G^{up}_{copy(B)}$, and that it was an exact copy of $G^{up}_B$.
- The start and end times of :Fetch-BUp-YYYYMMDD.
- That if the new graph - $G_{B'}$ - was also copied, then $G_{copy(B')}$ is a new version of $G_{copy(B)}$.
- That the provenance graph $P^{up}_{copy(B)}$ was derived from $P^{up}_B$ and is a new version of $P_{copy(B)}$.

Recall that we are only interested in the provenance of the graphs which are pertinent exclusively to system $C$. Therefore, $P_{copy(B)}$ may be discarded at this stage if $P^{up}_{copy(B)}$ contains the whole of the provenance of $G_{copy(B')}$; we discuss the discarding of old provenance graphs and copies of source graphs in our plans for future work, Section 8.2. The relationships between the graphs is shown in Figure 4.8. Note that the grey-shaded entity is the copy of $P^{up}_B$.

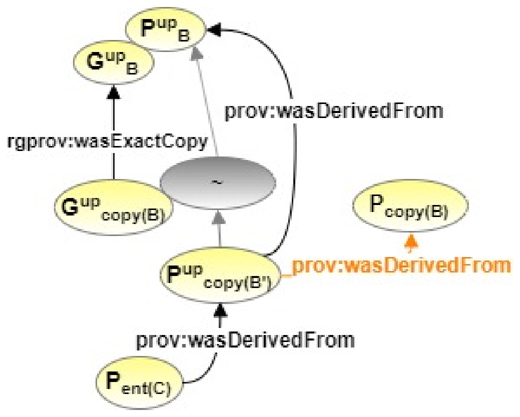


Figure 4.8: Relationships Between Update and Provenance Graphs and Their Copies.

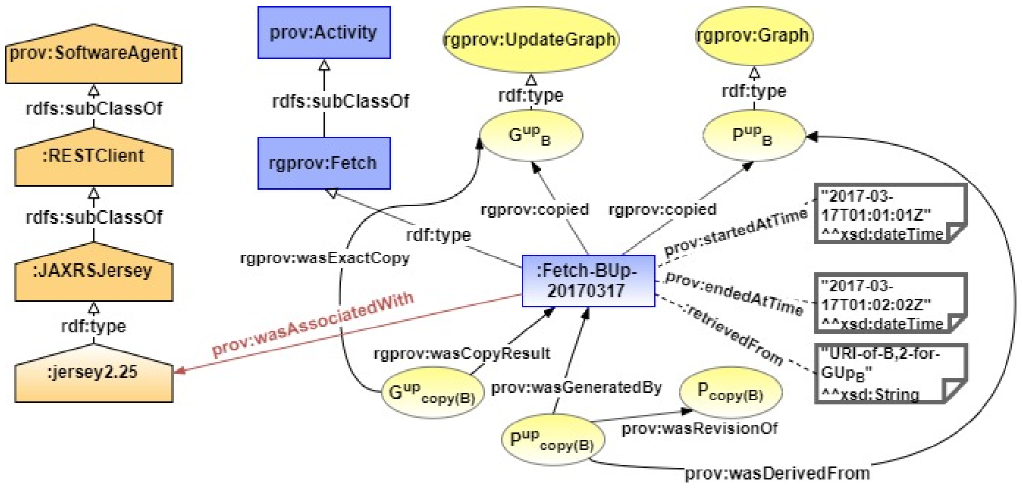


Figure 4.9: First Iteration of $P_{ent(C')}$.

$C$ then creates the provenance graph $P_{ent(C')}$ by including the above information as well as the information stating that $P_{ent(C')}$ is a new version of $P_{ent(C)}$, this creates the triple: $P_{ent(C')}$ prov:wasRevisionOf $P_{ent(C)}$. The first iteration of $P_{ent(C')}$ is shown in Figure 4.9, with the full triple list is detailed in Appendix B. Stating that $G_{ent(C')}$ prov:wasRevisionOf $G_{ent(C)}$ will be added after $G_{ent(C')}$ has been created.

### 4.3.2 Effects of Updates

The next step is that $C$ checks $P_{ent(C)}$ to see which set theoretic operation was performed to produce $G_C$ and which graphs originally contributed to its creation, in this case the graphs $G_{copy(A)}$ and $G_{copy(B)}$. It also checks what sort of propagation needs to be applied and uses Jena to perform the update and produce the graph $G_{C'}$.

#### 4.3.2.1 Insert

Jena performs the Insert operation on graph $G_{ent(C)}$ by inserting into it either the entirety of $G^{up}_{copy(B)}$ or a subgraph of it, namely $G^{up}_{sub(copy(B))}$, to produce graph $G_{C'}$. Afterwards, $C$ adds the following information to $P_{ent(C')}$:

- Which instance of rgprov:InsertOperation was invoked, namely insert-C-YYYYMMDD.
- That :insert-C-YYYYMMDD was associated with :jena3.1.1.
- That :insert-C-YYYYMMDD used $G_C$ in its insert operation.
- If the entirety of $G^{up}_{copy(B)}$ was inserted, then that :insert-C-YYYYMMDD used $G^{up}_{copy(B)}$ in its insert operation. If a subgraph of it was used, then that the update graph $G^{up}_{sub(copy(B))}$ was derived from both $G^{up}_{copy(B)}$ and $G_{copy(A)}$ and that :insert-C-YYYYMMDD used $G^{up}_{sub(copy(B))}$ in its insert operation.
- That :insert-C-YYYYMMDD generated the graph $G_{C'}$.
- That $G^{up}_{copy(B)}$ was a contributor in the creation of $G_{C'}$.
- If the subgraph $G^{up}_{sub(copy(B))}$ was used, then that $G_{copy(A)}$ and $G^{up}_{sub(copy(B))}$ were contributors in the creation of $G_{C'}$.
- That $G_{C'}$ is a new version of $G_C$.
- The start and times of :insert-C-YYYYMMDD.

The above list is shown in Figures 4.10 and 4.11, with the full triple list is detailed in Appendix B.

#### 4.3.2.2 Delete

Jena performs the Delete operation on graph $G_C$ either by deleting the entirety of the update graph $G^{up}_{copy(B)}$ or a subgraph of it, namely $G^{up}_{sub(copy(B))}$, to produce the graph $G_{C'}$. Afterwards, $C$ adds the following information to $P_{ent(C')}$:

- Which instance of rgprov:DeleteOperation was invoked, namely :delete-C-YYYYMMDD.
- That :delete-C-YYYYMMDD was associated with :jena3.1.1
- That :delete-C-YYYYMMDD used $G_C$ in its delete operation.
- If all the triples in $G^{up}_{copy(B)}$ were used in the deletion, then that :delete-C-YYYYMMDD used $G^{up}_{copy(B)}$ in its delete operation. If a subgraph of it was used, then that the update graph $G^{up}_{sub(copy(B))}$ was derived from both $G^{up}_{copy(B)}$ and $G_{copy(A)}$ and that :delete-C-YYYYMMDD used $G^{up}_{sub(copy(B))}$ in its delete operation.

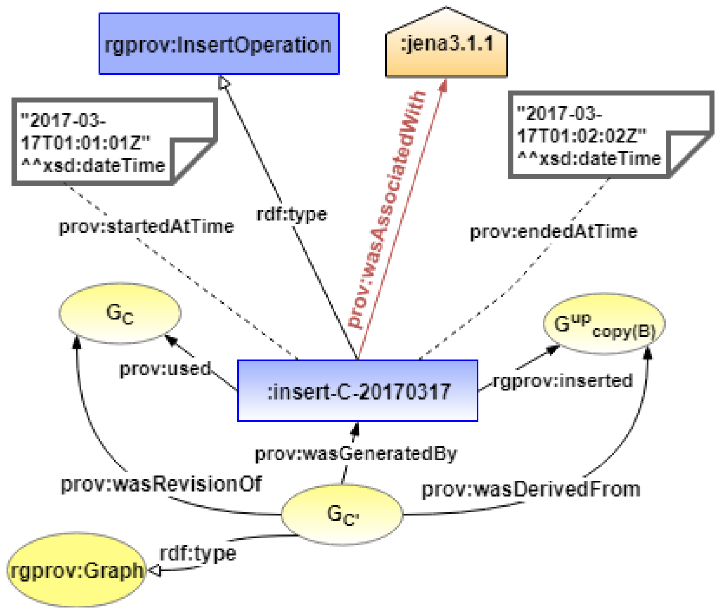


Figure 4.10: Second Iteration of $P_{ent(C')}$ - Case Insert After Union.

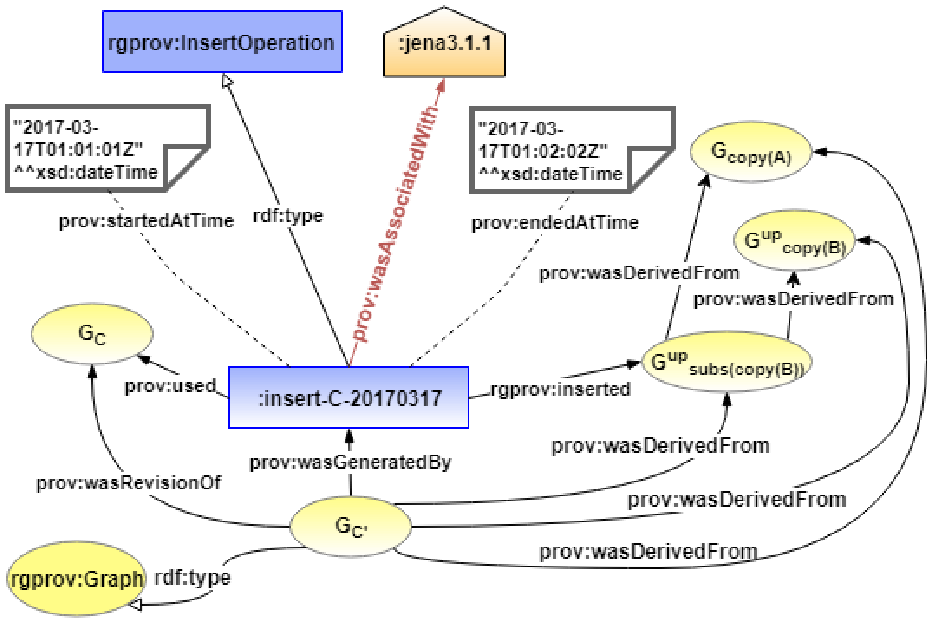


Figure 4.11: Second Iteration of $P_{ent(C')}$ - Cases Insert After Intersection and Difference 2 and Delete After Difference 1.

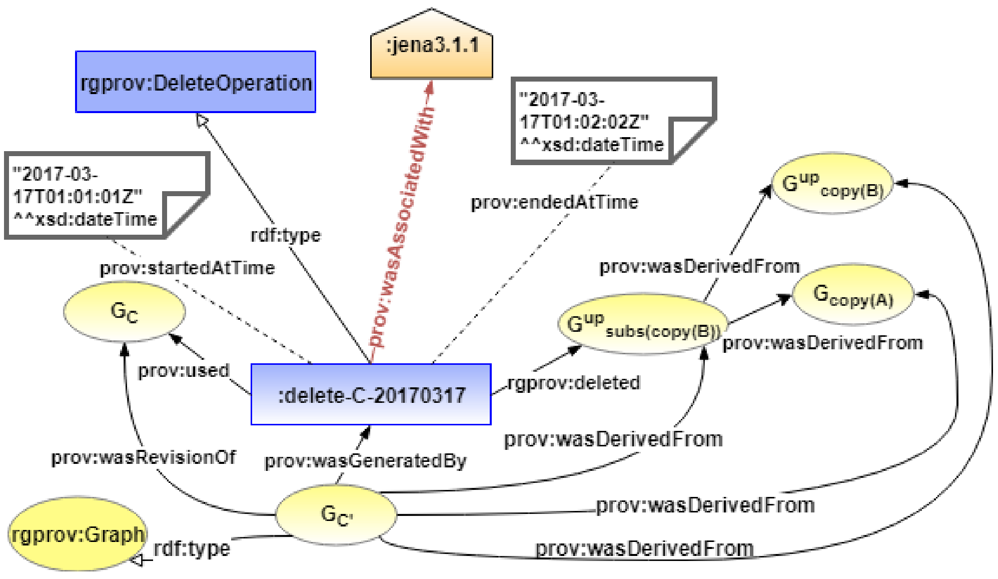


Figure 4.12: Second Iteration of $P_{ent(C')}$ - Case Delete After Union.

- That :delete-C-YYYYMMDD generated $G_{C'}$.
- That $G^{up}_{copy(B)}$ was a contributor in the creation of $G_{C'}$.
- If the subgraph $G^{up}_{sub(copy(B))}$ was used, then that both $G_{copy(A)}$ and $G^{up}_{sub(copy(B))}$ were contributors in the creation of $G_{C'}$.
- That $G_{C'}$ is a new version of $G_C$.
- The start and end times of :delete-C-YYYYMMDD.

The above list is shown in Figures 4.12 and 4.13, with the full triple list is detailed in Appendix B.

### 4.3.3 Re-Entailment

In the final stage of producing $G_{ent(C')}$ from $G_{C'}$, $C$ invokes the Jena methods for RDFS entailment as per Subsection 4.2.4. Despite the fact that only a subgraph of $G_{C'}$ is being re-entailed, we find no need to single that subgraph out when documenting the provenance of re-entailment, as we find it sufficient to state that $G_{ent(C')}$ was the product of an entailment process applied on $G_{C'}$. Hence, after $G_{ent(C')}$ is created, $C$ adds the same information to $P_{ent(C')}$ that is found in Subsection 4.1.2.2, namely:

- Which instance of rgprov:RDFSEntailment was invoked, namely :ge-C-YYYYMMDD.
- That :ge-C-YYYYMMDD was associated with :jena3.1.1.
- That :ge-C-YYYYMMDD used $G_{C'}$.

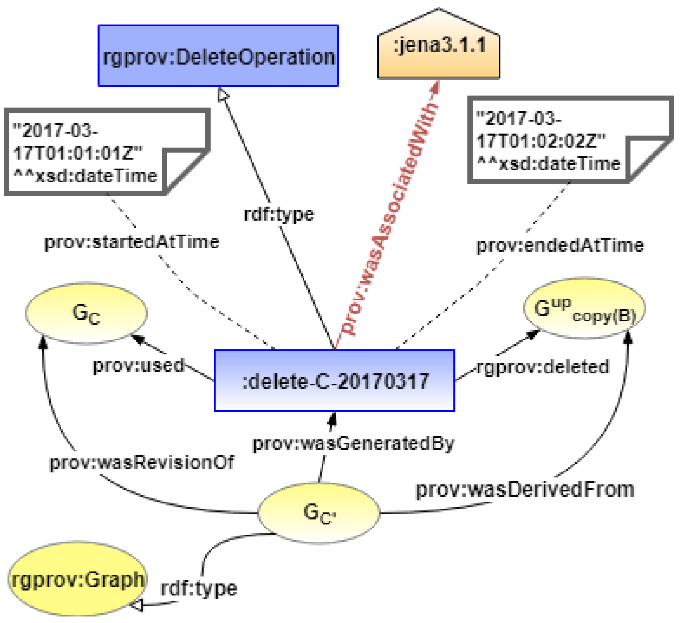


Figure 4.13: Second Iteration of $P_{ent(C')}$ - Cases Delete After Intersection and Difference 2 and Insert After Difference 1.

- That :ge-C-YYYYMMDD produced the graph $G_{ent(C')}$.
- That $G_{C'}$ entailed $G_{ent(C')}$.
- That $G_{ent(C')}$ was derived from $G^{up}_{copy(B)}$. In case $G^{up}_{sub(copy(B))}$ was used to produce $G_{C'}$, then also that $G_{ent(C')}$ was derived from it.
- That $G_{ent(C')}$ is a new version of $G_{ent(C)}$.
- The start and end times of :ge-C-YYYYMMDD's.

The above list constitutes the final additions to $P_{ent(C')}$ and is shown in Figures 4.14 and 4.15. The full triple list is detailed in Appendix B.

## 4.4 Summary

In this chapter, we demonstrated an application of the RGPROV vocabulary on the running example presented in the previous chapter. First, we showed how RGPROV describes the provenance of graphs produced by fetching other graphs from separate systems and by applying set theoretic operations on them, as well as the provenance of graphs produced by entailment. Then we shifted the focus to establish how updates on source graphs are to be propagated in systems whose entailed graphs were derived from them. Finally, we showed how RGPROV can describe the provenance of propagating updates.

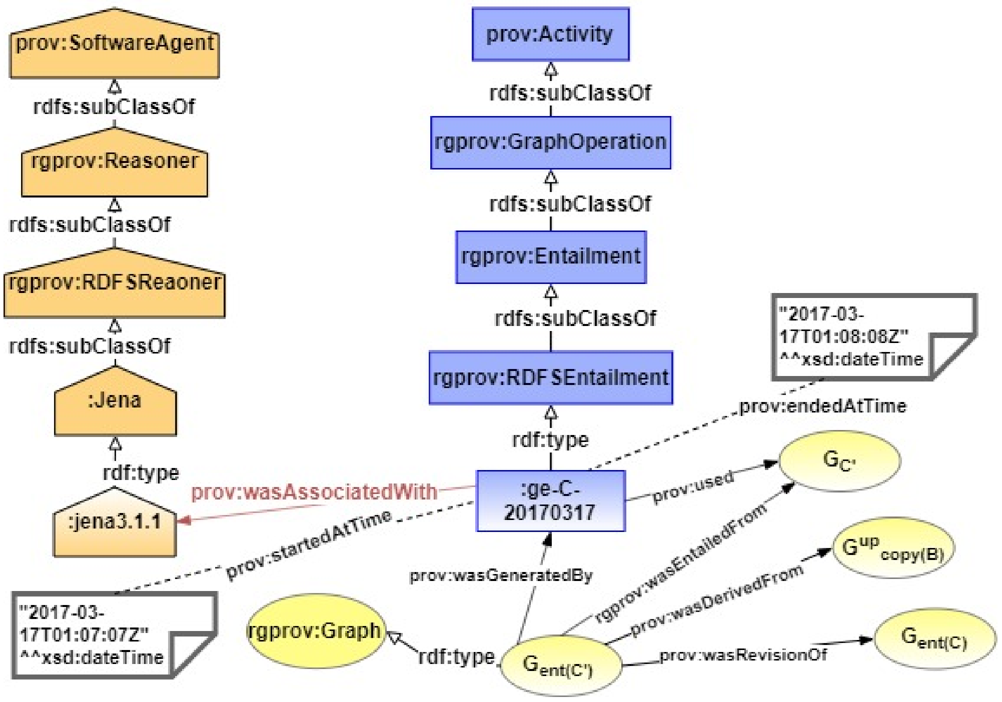


Figure 4.14: Final Iteration of $P_{ent(C')}$ Using all the Update Graph.

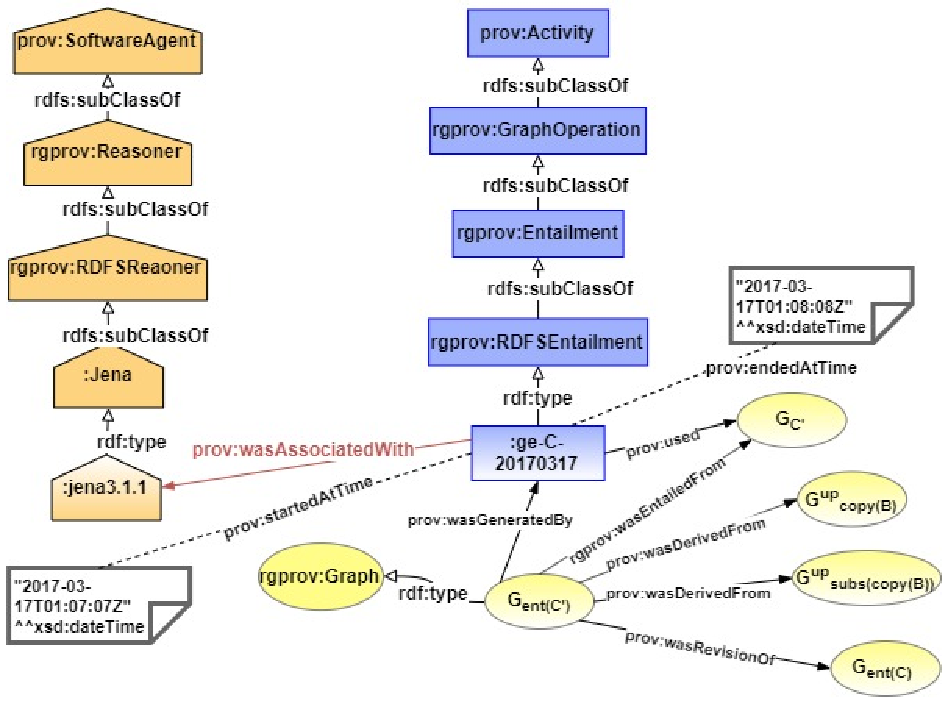


Figure 4.15: Final Iteration of $P_{ent(C')}$ Using a Subgraph of the Update Graph.

In the next chapter, we present the design of the model we have implemented to test our approach, and describe that implementation.

# Chapter 5

# Design and Implementation

In this chapter, we present the system which we have implemented that makes use of the RGPROV vocabulary. We start by presenting its design and then describe the different components that make it up and expand on each of them.

This chapter consists of four parts. In Section 5.1, we outline the design of our system. In Section 5.2, we detail the components of the system which we have implemented. In Section 5.3, we present the third party components which we have used. Finally, in Section 5.4, we summarise this chapter.

## 5.1 System Design

We have designed a system comprising seven components, of which we have implemented four. The system architecture is shown in Figure 5.1. The main component, named the Operator, is responsible for controlling and invoking the operations performed on the graphs in the system. As it is the central component, it's the one which invokes and communicates with all the other components. The second component, named the Provenance Handler, is responsible for creating, querying, and updating the provenance graphs. The third component is the independent SPARQL Server and Graph Store, which we have not implemented but used the third party Jena Fuseki Server. The fourth component is the independent reasoner, which we have also not implemented but used the third party Jena. Jena is responsible for performing the set theoretic and entailment operations on all graphs. The fifth component is the Update Producer, which handles any updates applied on the system's internal graphs that are used as source graphs by outside systems. The sixth component is the Cache. Finally, the remaining component is the REST client, which we have not implemented, as it does not pertain to the demonstrating the application of the RGPROV vocabulary nor does it affect the evaluation of the system.

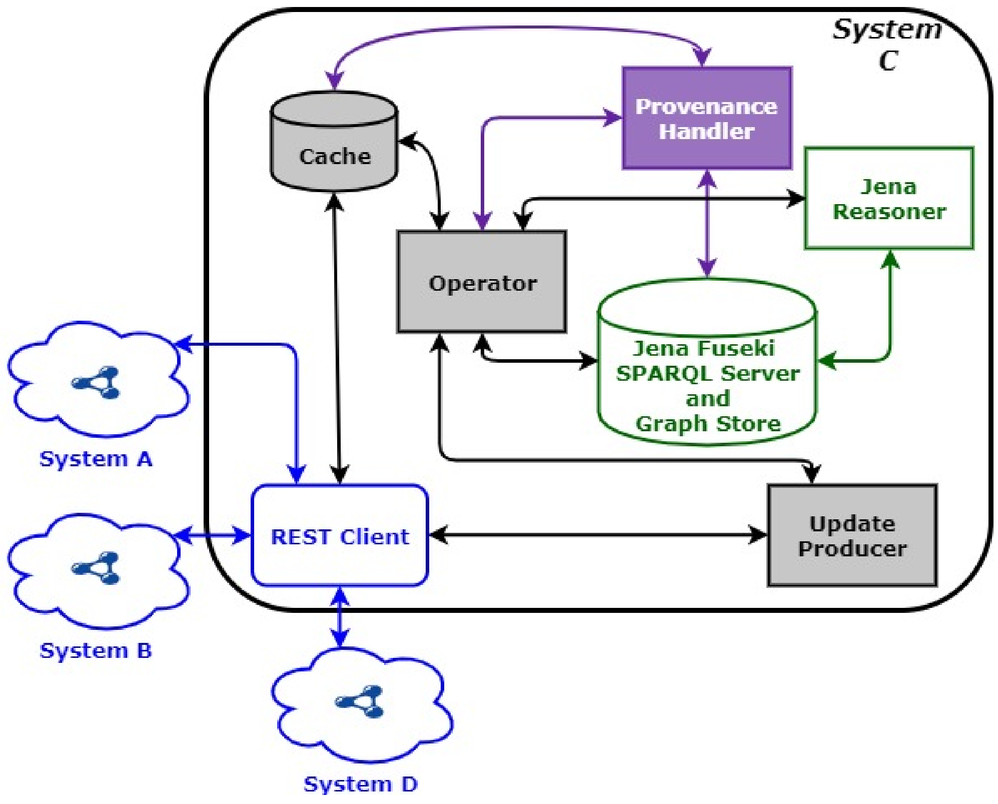


Figure 5.1: System Architecture, with the Shaded Parts Indicating the Implemented Components.

Note that unless they have been marked as inferred triples, all triples in the source graphs are treated as ground triples in the system. Then, after it is produced, graph $G_{ent(C)}$ is stored as two graphs. The first, $G_C$, consists of the ground triples and the second, $G_{inf(C)}$, consists of the inferred triples produced by our system's reasoner. This separation proves beneficial when re-deriving to minimise over-deletions and re-insertions.

All components have been implemented in Java.

## 5.2 Implemented Components

### 5.2.1 Operator

This is the main and central component of our system, and as such it is responsible for invoking the other components and handling most of the communications between them.

#### Initial Graph Creation

Recall that graph $G_{ent(C)}$ is created as shown in Figure 5.2 and in more detail in Figure 5.3 (these figure are copies of Figures 3.3 and 3.4).

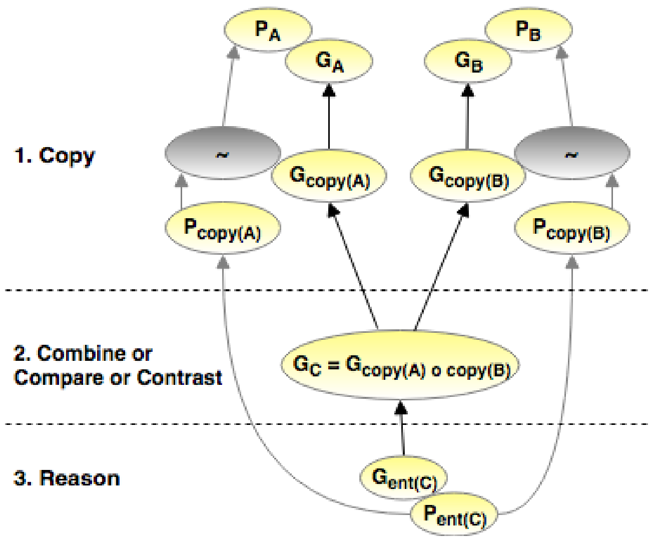


Figure 5.2: Copy of Figure 3.3 - Production of $G_{ent(C)}$ from $G_A$ and $G_B$.

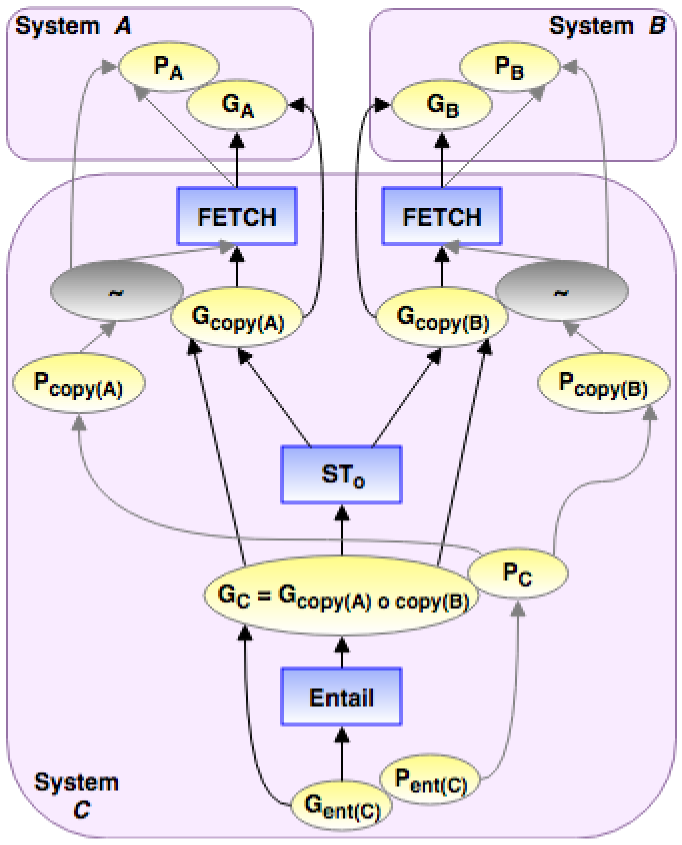


Figure 5.3: Copy of Figure 3.4 - Detailed Production of $G_{ent(C)}$ and its Provenance graph $P_{ent(C)}$ from $G_A$, $G_B$, and their Provenance Graphs.

Before the graph $G_{ent(C)}$ is to be initially created, the Operator loads the copies of graphs $G_A$ and $G_B$, namely $G_{copy(A)}$ and $G_{copy(B)}$ along with their provenance graphs from the Cache. As previously mentioned, we assume, in our implementation, that graphs $G_A$ and $G_B$, and subsequently their copies $G_{copy(A)}$ and $G_{copy(B)}$, contain only ground triples, i.e. there are no inferred triples in them. In addition to allowing different systems to use different entailment regimes internally, this saves the Operator having to go through all the triples to find and mark the inferred triples.

Afterwards, the Operator instructs the Provenance Handler to create the graphs $P_{copy(A)}$ and $P_{copy(B)}$ from the copies of $P_A$ and $P_B$ respectively, as well as the first iteration of $P_{ent(C)}$, as detailed in Section 4.1.1.

The Operator then invokes Jena to create $G_C$ and subsequently instructs the Provenance Handler to produce the second iteration of $P_{ent(C)}$, as detailed in Section 4.1.2.1. Next, it invokes Jena again to create $G_{ent(C)}$ by applying RDFS reasoning to $G_C$. Here, Jena may return to the Operator two graphs, the first, named $G_{inf(C)}$, containing the inferred triples and the second being the entirety $G_{ent(C)}$, containing both the base and the inferred triples. We do need the latter as we are storing it as two graphs. While it is not necessary to split the graph, doing do saves the Operator from having to go through each triple to find the inferred ones whenever there is an update. It also allows the Operator to avoid over-deletions and re-insertions.

Afterwards, the Operator instructs the Provenance Handler to produce the final iteration of $P_{ent(C)}$, as detailed in Section 4.1.2.2. We have elected to keep copies of the original graphs, and so the Operator's final task is uploading $G_{copy(A)}$ and $G_{copy(B)}$ to Fuseki along with $P_{copy(A)}$, $P_{copy(B)}$, $G_C$, $G_{inf(C)}$, and $P_{ent(C)}$ and deleting the graphs $G_{copy(A)}$, $G_{copy(B)}$, and the copies of $P_A$ and $P_B$ from the Cache.

### Graph Update

Recall that graph $G_{ent(C')}$ is entailed from a portion of graph $G_{C'}$, and that the latter may be produced in one of four ways: inserting all the update graph, inserting a subgraph of the update graph, deleting all the update graph, or deleting a subgraph of the update graph. The production of $G_{ent(C')}$ is summed up in Figures 5.4 and 5.5 and expanded upon next.

After an update graph, $G_B^{up}$, has been received into $C$ as $G_{copy(B)}^{up}$, the Operator loads it along with the new provenance from the Cache. It also loads graphs $G_C$ and $G_{inf(C)}$ and the provenance graph $P_{ent(C)}$ from Fuseki into the Cache. It then directs the Provenance Handler to produce the graph $P_{copy(B)}^{up}$ and the first iteration of $P_{ent(C')}$, as described in Section 4.3.1.

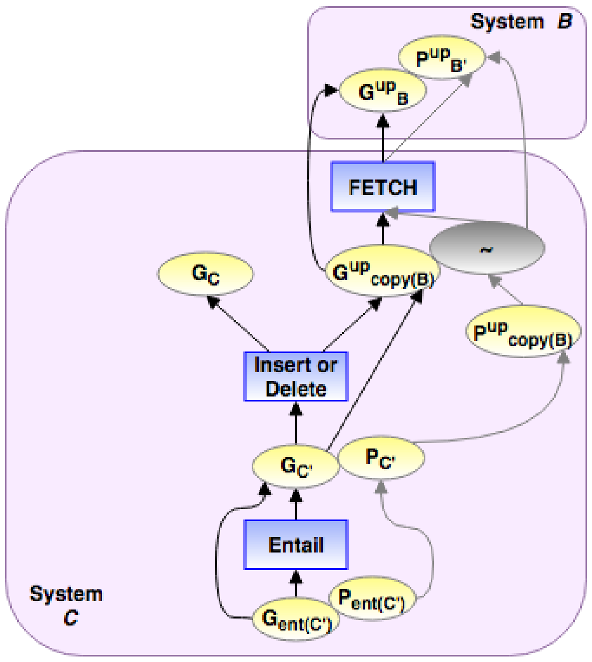


Figure 5.4: Production of $G_{ent(C')}$ (All the Update Graph is Used).

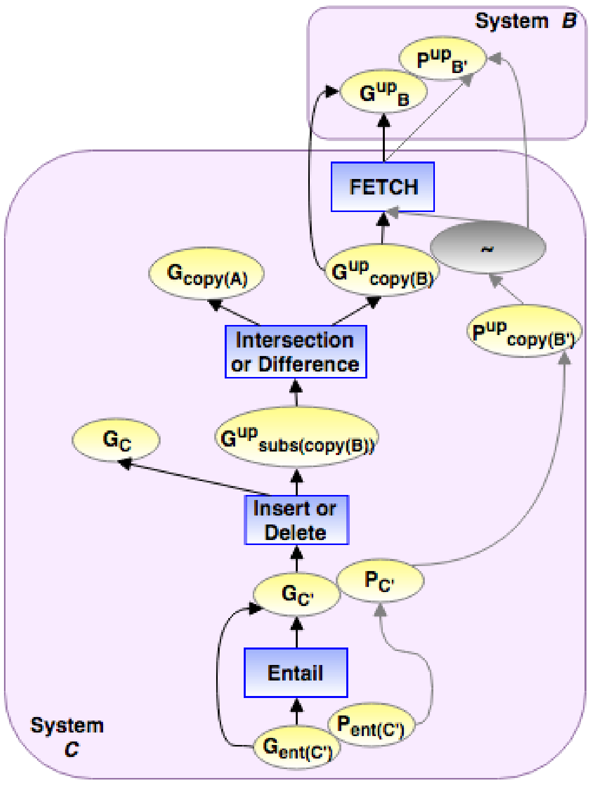


Figure 5.5: Production of $G_{ent(C')}$ (A Subgraph of the Update Graph is Used).

Next, the Operator checks the provenance graph $P_{ent(C)}$, on Fuseki, to see which set theoretic operation was used to create $G_C$ and based on this it determines which update propagation to apply, as detailed in Section 4.2.2 and shown in Steps (*1*) in Table 4.1. The Operator also checks whether all the update $G^{up}_{copy(B)}$ is to be used or invokes Jena to compute the subgraph of $G^{up}_{copy(B)}$, $G^{up}_{sub(copy(B))}$, to be applied. If $G^{up}_{sub(copy(B))}$ is to be computed, then the Operator either requests a new copy of graph $G_A$[1] from system $A$, or, as in our current implementation, loads $G_{copy(A)}$ from Fuseki. The aforementioned is shown in Algorithm 1. Although not required in our implementation, the algorithm can contain an optional extra step at the beginning to check whether any of the triples are inferred by comparing them against $G_{inf(C)}$ and removing them. Algorithm 1 has $O(1)$ complexity, although the complexity of the SELECT queries depend on the implementation of SPARQL.

If the update is an insert, the Operator creates a SPARQL Insert statement and sends it to Fuseki to add those triples to $G_C$. As previously mentioned, those triples may either be all those in the update $G^{up}_{copy(B)}$, or some of them thus comprising $G^{up}_{sub(copy(B))}$. The result of adding those triples is the graph $G_{C'}$. The Operator then requests the graphs resulting from the SPARQL Describe of those triples from Fuseki, and forwards their union to Jena for reasoning. Jena then returns the entailed triples resulting from reasoning on this union, the Operator creates another SPARQL Insert statement that adds those inferred triples into $G_{inf(C)}$, thus evolving it into $G_{inf(C')}$. This completes the steps needed to produce $G_{ent(C'}$. Finally, the Operator directs the Provenance Handler to produce the graph $P_{ent(C')}$, as detailed in Section 4.3.2. This completes the steps needed to produce $G_{ent(C')}$. The aforementioned is shown in Algorithm 2. Algorithm 2 has $O(n)$ complexity - where $n$ is the number of triples to be inserted, although the complexity of the DESCRIBE query depends on the implementation of SPARQL and the complexity of the entailment depends on the implementation of the reasoner.

If the update is a delete, then the Operator first gets, from Fuseki, the graphs resulting from the SPARQL Describe of the triples to be deleted. As previously mentioned, those triples may either be all those in the update $G^{up}_{copy(B)}$, or some of them comprising $G^{up}_{sub(copy(B))}$. The Operator then loops over each triple to be deleted and examines its predicate. If the predicate is an rdf:type or has super-properties (i.e. it is a sub-property of another property), then it adds, to the list of inferred triples to be deleted, the triples with the same subject and any objects that relate it to the predicate. This is in accordance to the RDFS entailment rules described in Section 2.1.3.2.2 in Table 2.1. Next, the Operator sends two SPARQL Delete statements to Fuseki, the first to delete the ground triples from $G_C$, thus resulting in it becoming $G_{C'}$, and the second to delete the inferred triples from $G_{inf(C)}$, thus evolving it into in $G_{inf(C')}$. This completes the steps needed to produce $G_{ent(C')}$.

[1]The graph and provenance retrieval will be identical to the first time $G_A$ had been retrieved.

**Algorithm 1** Generate What Is To Be Applied As Update

```
Function: getGraph : $URL \rightarrow graph$
Function: getStOpFromProv : $graph \rightarrow String$
Function: getSubtrahend : $graph \rightarrow String$

1: procedure generateUpdate($provGraph$, $SPARQLupType$, $updateGraph$)
       ▷ Start by assuming the use of all the update graph
2:     $graphB.updateGraph \leftarrow updateGraph$
       ▷ Get the set theoretic operation
3:     $qStOpType \leftarrow$ getStOpFromProv($provGraph$)
                   Where $getStOpFromProv$ queries $provGraph$ using:
                       SELECT ?stOpType FROM <$provGraph$> WHERE {
               ▷           ?stOp rdf:type ?stOpType .
                           $G_{ent(C)}.Name$ rgprov:wasEntailedFrom ?g .
                           ?g prov:wasGeneratedBy ?stOp . }
4:     if $qStOpType$ = "$Union$" and $SPARQLupType$ = "$Delete$" then
5:         $graphAcopy \leftarrow$ getGraph($G_A$)
6:         $graphB.updateGraph \leftarrow updateGraph \setminus graphAcopy$
7:     else if $qStOpType$ = "$Intersection$" and $SPARQLupType$ = "$Insert$" then
8:         $graphAcopy \leftarrow$ getGraph($G_A$)
9:         $graphB.updateGraph \leftarrow updateGraph \cap graphAcopy$
10:    else if $qStOpType$ = "$Difference$" then
           ▷ Get the subtrahend
11:        $subtrahend \leftarrow$ getSubtrahend($provGraph$)
                   Where $getSubtrahend$ queries $provGraph$ using:
                       SELECT FROM <$provGraph$> ?s WHERE {
                           ?stOp rdf:type rgprov:Difference .
               ▷           ?g prov:wasGeneratedBy ?stOp .
                           $G_{ent(C)}.Name$ rgprov:wasEntailedFrom ?g .
                           ?stOp rgprov:hadSubtrahend ?s . }
12:        if $subtrahend = G_{copy(B)}.Name$ then          ▷ This is Case Difference 1
13:            if $SPARQLupType$ = "$Insert$" then
14:                $SPARQLupType \leftarrow$ "$Delete$"
15:            else
16:                $SPARQLupType \leftarrow$ "$Insert$"
17:                $graphAcopy \leftarrow$ getGraph($G_A$)
18:                $graphB.updateGraph \leftarrow updateGraph \cap graphAcopy$
19:        else                                             ▷ This is Case Difference 2
20:            if $SPARQLupType$ = "$Insert$" then
21:                $graphAcopy \leftarrow$ getGraph($G_A$)
22:                $graphB.updateGraph \leftarrow updateGraph \setminus graphAcopy$
```

It is also possible to apply a delete followed by an insert. In this case, after the triples to be deleted have been removed from $G_C$ and $G_{inf(C)}$, the Operator adds the triples to be inserted to $G_C$. Afterwards, the Operator requests the SPARQL Describe of all the subjects and predicates that were in the triples to be updated and sends the union of the resulting graphs to Jena for reasoning. When Jena returns the entailed triples resulting from reasoning on the union, the Operator sends a SPARQL Insert statement

**Algorithm 2** Apply Insert Update

```
    Function: $\textbf{\textit{describe}}$ : $graph$ x $triple \rightarrow graph$
    Function: $\textbf{\textit{entail}}$ : $graph \rightarrow graph$

1: procedure APPLYINSERTUPDATE($baseGraph, infGraph, triplesTBI$)
2:     $described \leftarrow \phi$
       ▷ Step 1: Insert all $triplesTBI$ into the base graph:
3:     $baseGraph \leftarrow baseGraph \cup triplesTBI$
       ▷ Step 2: Loop over all $triplesTBI$ and get their Describe:
4:     for each $triple$ in $triplesTBI$ do
5:         $described \leftarrow described \cup \textbf{\textit{describe}}(baseGraph \cup infGraph, triple)$
       ▷ Step 3: Re-entail using $described$ only:
6:     $newlyEntailed \leftarrow \textbf{\textit{entail}}(described)$
       ▷ Step 4: Insert the inferred triples into the graph:
7:     $infGraph \leftarrow infGraph \cup newlyEntailed$
```

containing the inferred triples to Fuseki which adds them to $G_{inf(C)}$, thus evolving it into in $G_{inf(C')}$. This completes the steps needed to produce $G_{ent(C')}$. The aforementioned is shown in Algorithm 3. Algorithm 3 has $O(n^3)$ complexity - where $n$ is the number of triples returned by the DESCRIBE query of the triples to be deleted, although the complexity of the DESCRIBE queries depend on the implementation of the SPARQL and the complexity of the entailment depends on the implementation of the reasoner.

Finally, the Operator directs the Provenance Handler to produce the graph $P_{ent(C')}$, as detailed in 4.3.2.

As with the initial creation of graph $G_{ent(C)}$, the last step is to upload the provenance graphs $P^{up}_{copy(B)}$ and $P_{ent(C')}$ to Fuseki and then to empty the Cache. Two additional steps need to be undertaken when an update happens. The first step is to be taken only if copies of the source graphs are kept in the system and that is to either apply $Up_B$ to $G_{copy(B)}$, resulting in $G_{copy(B')}$ which reflects $G_{B'}$, or to upload the update graph - whether $G^{up}_{copy(B)}$ or $G^{up}_{sub(copy(B))}$ on its own - and save it with the other copies of source graphs. The second step is for the Operator to send the list of added or deleted triples to the Update Producer. Finally, since there is now a new graph $G_{ent(C')}$ with its new provenance graph $P_{ent(C')}$, the graphs that form $G_{ent(C)}$, namely $G_C$ and $G_{inf(C)}$ as well as the old provenance $P_{ent(C)}$ may either be deleted or kept for historical bookkeeping purposes. For now, we have chosen to keep the old copies. However, a deeper look needs to be taken into how and why older copies may be kept, and we leave this to our future work.

### 5.2.2 Provenance Handler

This component is responsible for producing and updating provenance graphs. It is invoked by the Operator to perform these tasks at certain steps of producing $G_{ent(C)}$

**Algorithm 3** Delete and Re-entail

```
    Function: describe : $graph$ x $triple \rightarrow graph$
    Function: describe : $graph$ x $iri \rightarrow graph$
    Function: entail : $graph \rightarrow graph$

 1: procedure DeleteReEntail($baseGraph, infGraph, triplesTBD, triplesTBI$)
 2:     $described \leftarrow \phi$, $describedTBI \leftarrow \phi$, $describedTBD \leftarrow \phi$
        ▷ Step 1: Loop over the triples to be deleted and get their describe.
 3:     for each $triple$ in $triplesTBD$ do
 4:         $described \leftarrow described \cup \textbf{describe}(baseGraph \cup infGraph, tripleTBD)$
        ▷ Step 2: Loop over the triples to be deleted again.
 5:     for each $triple$ in $triplesTBD$ do
            ▷ Step 2.1: Get the triples that share this triple's subject.
 6:         $subject = triple.Subj$
 7:         $t \leftarrow \{t \in described | t.Subj = subject\}$
            ▷ Step 2.2: Now loop over these triples to check their predicates.
 8:         for each $tripleWithSameSubject$ in $t$ do
                ▷ Step 2.3: Check the triple's property.
 9:             if $tripleWithSameSubject.Prop = rdf : type$ then
                    ▷ Step 2.3.1: Get all the super classes.
10:                 $superClasses \leftarrow \{tsOAsS.Obj | tsOAsS \in described$
                                    $\wedge tsOAsS.Subj = tripleWithSameSubject.Obj$
                                    $\wedge tsOAsS.Prop = rdfs : subClassOf\}$
                    ▷ Step 2.3.1.1: Loop again to mark inferred triples to be deleted.
11:                 for each $superClass$ in $superClasses$ do
                        ▷ Step 2.3.1.2: Add to the inferred triples to be deleted:
12:                     $infTriplesTBS \leftarrow infTriplesTBS \cup$
                                    $\langle subject, rdf : type, superClass\rangle$
13:             else
                    ▷ Step 2.3.2: Get the super-properties of this property
14:                 $superProps \leftarrow \{tp.Obj | tp \in described$
                                    $\wedge tp.Subj = tripleWithSameSubject.Prop$
                                    $\wedge tp.Prop = rdfs : subPropertyOf\}$
15:                 for each $superProp$ in $superProps$ do
16:                     $infTriplesTBS \leftarrow infTriplesTBS \cup \langle subject,$
                                    $\wedge superProp, tripleWithSameSubject.Obj\rangle$
        ▷ Step 3: Delete from $graph$ all the chosen triples
                ▷ and add the triples to be inserted.
17:     $baseGraph \leftarrow (baseGraph \setminus triplesTBD) \cup triplesTBI$
18:     $infGraph \leftarrow infGraph \setminus infTriplesTBS$
        ▷ Step 4: Re-derive and insert inferred triples.
19:     for each $triple$ in $triplesTBI$ do
20:         $describedTBI \leftarrow describedTBI \cup \textbf{describe}(baseGraph \cup infGraph, triple)$
21:     $subjsAndObjs \leftarrow \{iri | iri \in triplesTBD.Subjects \cup triplesTBD.Objects\}$
22:     for each $iri$ in $subjsAndObjs$ do
23:         $describedTBD \leftarrow describedTBD \cup \textbf{describe}(baseGraph \cup infGraph, iri)$
24:     $infGraph \leftarrow infGraph \cup \textbf{entail}(describedTBI \cup describedTBD)$
```

and updating it to $G_{ent(C')}$.

In the initial step of creating $G_{ent(C)}$, as shown in Section 4.1, copies of graphs $G_A$ and $G_B$ and their provenance graphs are retrieved, then, this component creates the updated provenance graphs $P_{copy(A)}$ and $P_{copy(B)}$ and also creates $P_{ent(C)}$. While we have not implemented it to do so and despite that it doing so counteracts our approach to avoid keeping provenance at the triple level, this component may also keep a list of sources marking them as either credible or untrustworthy, and may use it to check the provenance of the source graphs to decide whether or not to discard any triples that may have been imported from unreliable sources. Then at each step of the creation of graph $G_{ent(C)}$, this component updates $P_{ent(C)}$.

When an update reaches $C$, as shown in Section 4.3, this component creates $P^{up}_{copy(B)}$ and $P_{ent(C')}$. Again, it may check the provenance of the update and decide whether or not to discard any triples that may have been imported from unreliable sources. Thus, it may reject the change $G^{up}_{copy(B)}$, accept part of it, or accept it all. Then, at each step of updating $G_{ent(C)}$ to become $G_{ent(C')}$, this component is responsible for updating $P_{ent(C')}$.

### 5.2.3 Update Producer

This component is responsible for making available the list of triples that have been added or deleted from $G_C$ after every update. As we have previously mentioned, systems publish graphs with only the base triples. Thus, the list either contains $G^{up}_{copy(B)}$ or its subgraph $G^{up}_{sub(copy(B))}$. The Update Producer receives this list from the Operator, and uploads it to Fuseki. It may then perform its job in two ways, as discussed in Section 4.2.1. The first way is by forwarding the list of triples to the REST client to make it available for access to any system which requires it. The second way is by maintaining a list of systems that use graphs produced by $C$.

### 5.2.4 Cache

This component is the straightforward temporary storage. It contains copies of source and provenance graphs retrieved from other systems as well as intermediate graphs produced internally. We have elected to empty the cache after the creation or updates of graphs because whilst storing those graphs may be helpful, the storage overhead may get very high very quickly. Additionally, those intermediate graphs do not serve any purpose and the data they contain can be easily accessed from the graphs saved on Fuseki.

## 5.3 Third Party Components

In this section we present the two third-party components which we have used in our system, the Jena Reasoner and the Jena Fuseki SPARQL Server and Graph Store.

### 5.3.1 Reasoner

The Jena Reasoner is the component we use in our system to perform set theoretic operations on graphs as well as to perform reasoning[2] on them. The version we have in our system is 3.1.1[3]. It is invoked by the Operator. Unfortunately, Jena does not tell us how many triples it processed when performing the set theoretic operation nor how many times each triple was processed. However, after reasoning, Jena can be used to split the entailed graph into two graphs, the first containing the original triples and the second containing the inferred triples. In our configuration, Jena cannot perform any SPARQL operations on the RDF graphs other than select, so the Insert, Delete, and Describe statements that need to be performed are done by Fuseki.

### 5.3.2 SPARQL Server and Graph Store

As previously mentioned, we use the third party Apache Jena Fuseki SPARQL server and graph store. The version in our system is Fuseki2[4].

In its capacity as a graph store, Fuseki allows the storage of named or unnamed graphs in various datasets. Both RDF and provenance graphs are stored on Fuseki. We have created one dataset which contains copies of the original graphs, namely $G_{copy(A)}$ and $G_{copy(B)}$ and their provenance graphs $P_{copy(A)}$ and $P_{copy(B)}$. We have also created another dataset which contains the graph $G_{ent(C)}$, split into $G_C$ and $G_{inf(C)}$, and its provenance $P_{ent(C)}$. A third and last dataset has been created to store previous copies of any graphs that have been updated.

In its capacity as a SPARQL server, Fuseki executes SPARL queries communicated to it by the Operator. This is straightforward and done as per the W3C SPARQL 1.1 Specifications. The implementation of SPARQL Describe in Fuseki allows it to be run on a dataset. In this case, and because both $G_C$ and $G_{inf(C)}$ are stored in the same dataset, the Describe statement of any triple would return a graph describing all the triples related to it, whether they are in the first graph or the second.

[2]At the time of the implementation and subsequent writing of this report, Jena had not yet supported OWL 2 inference.

[3]All Jena binary distributions are available on http://archive.apache.org/dist/jena/binaries/

[4]Fuseki2 is available on https://jena.apache.org/documentation/fuseki2/

## 5.4 Summary

In this chapter, we presented the system which we have implemented that makes use of the RGPROV vocabulary by first illustrating the system's design then describing each component. We also presented algorithms, used by the system, that detail the means to select whether the whole update or a part of it, applied to a source graph, needs to be propagated to an entailed graph derived from that source graph. We also showed how partial re-derivation can be applied to the entailed graph. In the next chapter we describe the evaluation framework we designed and implemented to test our system and present our results.

# Chapter 6

# Evaluation Framework

Benchmarks have been developed by the Semantic Web community to provide default and standard references by which systems can be measured, with the W3C maintaining a list[1] of those benchmarks for the past decade. Existing works focus on different factors and performance metrics when presenting their benchmarks, for example, some focus on reasoning capabilities and scalability, while others focus on efficiency of querying and storage. Additionally, almost all benchmarks artificially generate synthetic data; the exceptions being those that use the DBpedia dataset. The former argue that their requirements are not met by real data sources, while the latter argue that synthetically-generated data are not representative of the real world and so their datasets are real world data taken from actual sources.

In order to test our approach of update propagation, we require a dataset with two features. The first is that it can vary in size. The second is that it would be dynamic, with both the original data and the changes applied to them to be available. Since there are no real data sources that we are aware of which satisfy both our requirements, we chose the synthetically generated data produced by Meimaris and Papastefanatos (2016)'s EvoGen; especially because EvoGen is based on the widely used LUBM (Guo et al., 2005).

However, the description of EvoGen in (Meimaris and Papastefanatos, 2016) is misleading; while the published paper claims that the implementation[2] handles deletions, this is incorrect and has been confirmed as such in (Meimaris, 2018). Specifically, new versions of the generated datasets contain only insertions; with deletions not being part of the changes that graphs undergo. Moreover, its generated datasets consist of the initial version of each department within a university and the subsequent insertions. So, it does not produce fully updated graphs.

[1]The RDF Store Benchmarking list maintained by the W3C is available on https://www.w3.org/wiki/RdfStoreBenchmarking.

[2]https://github.com/mmeimaris/EvoGen

Therefore, in this chapter we describe the evaluation framework that we created by extending EvoGen to address some of its shortcoming, most importantly by applying deletions. At this stage, however, we do not apply strict deletions; we extend EvoGen from only performing insertions to performing insertions followed by having it delete the previously-inserted triples. Additionally, we incorporate UOBM's ontology, as described in (Ma et al., 2006), so that the generated graphs contain effective instance links. Finally, we make use of Moreau et al. (2018)'s PROV-TEMPLATE to produce provenance graphs complying with RGPROV so as to make the evaluation framework provenance-aware. While we leave the evaluation for Chapter 7, we note that there is a need for a provenance-aware Semantic Web benchmark which generates both RDF graphs and provenance graphs, to be used in evaluating systems that require the presence of both the data and their provenance.

This chapter consists of three parts. In Section 6.1, we present the goals of our evaluation benchmark. Next, in Section 6.2, we describe the framework by relating its design and implementation. Finally, in Section 6.3, we summarise this chapter.

## 6.1 The Evaluation Framework's Goals

In order to test our approach of update propagation, we require a dataset with the following two features. The first is that it can vary in size. The second is that it would be dynamic, with both the original data and the changes applied to them to be available. Since there are no real data sources that we are aware of which satisfy both those requirements, we chose to extend EvoGen by addressing its shortcomings and augmenting it to produce instance links similar to those of UOBM's, as well as enriching it with provenance.

Thus, the evaluation framework we present is based on EvoGen, which in turn is based on LUBM. It also incorporates UOBM's ontology and uses PROV-TEMPLATE to become provenance-aware. Figure 6.1 shows these relationships. This combination allows the framework to explicitly satisfy the following goals:

1. The sizes of the generated data to be of a range: Because we want to test our approach on datasets of different sizes, we want the sizes of the generated data to be of a range, including that which could be very large.

2. Ontology of moderate size and complexity: Because our focus is on the data and how our algorithms perform, we made use of an ontology that did not need to be too large.

3. Effective instance links: Because the graphs generated using the operations are shaped depending on existent links between the generated graphs, we require the

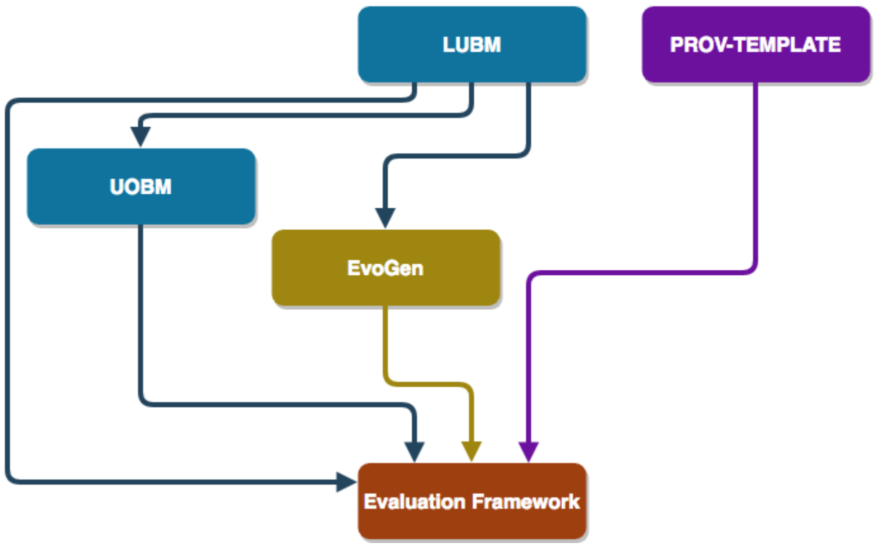


Figure 6.1: The Relationships Among our Evaluation Framework, the Benchmarks, and PROV-TEMPLATE.

generated graphs to contain connected instances, as opposed to LUBM's smaller isolated graphs.

4. The presence of provenance graphs: This stems from our observation that none of the available benchmarks are provenance-aware. While we do not strictly need any of our source graphs to have provenance graphs describing them, because our update propagation approach relies only the provenance graphs created within our system, we have chosen to make our evaluation framework provenance-aware so that it is well-rounded.

## 6.2 Framework Design and Implementation

LUBM, UOBM, EvoGen, and subsequently our evaluation framework, produce graphs containing synthetic information about universities. Each university is made up of a number of departments, along with instances describing people, courses, and publications as well as relations between those instances. The high-level architecture of our framework is shown in Figure 6.2, and its components are described next.

### 6.2.1 Data Generator

LUBM's Univ-Bench Artificial data generator - UBA - generates a dataset containing universities split into departments along with entities describing university faculty members, students, and research publications as per the LUBM ontology schema[3]. The number

[3]http://swat.cse.lehigh.edu/onto/univ-bench.owl

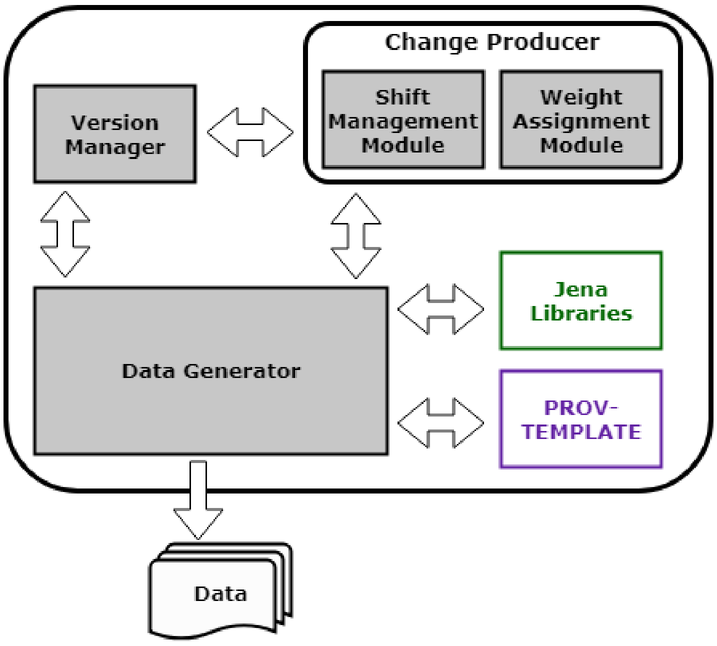


Figure 6.2: Architecture of Evolution Framework, with the Shaded Parts Indicating the Implemented Components.

of generated instances is chosen randomly from a pre-defined range of minimums and maximums according to their type. Instances are linked to each other either directly, for example, an associate professor works for a certain department and is the advisor of certain students, or indirectly, for example, an associate professor gives a course which in turn is taken by certain students. Generated instances also conform to hard-coded ranges, for example, any generated department would contain a minimum of 10 and a maximum of 14 associate professors and each of the associate professors has a minimum of 10 and 18 publications. EvoGen's data generator - EvoGenerator - replicates UBA's exact recipe, but also creates instances types for their 10 new classes linked with their 19 new properties. These new classes and properties are subclasses and subproperties of LUBM's classes and classes, and like LUBM, these new classes and properties are also hard-coded. The default configurations of both LUBM and EvoGen are shown in Table 6.1.

Our Data Generator carries out the generation of the randomised data. As a design decision we kept EvoGen's new classes and properties and produce data according to them. We have also added some of UOBM's classes and properties that create the links between universities. The design of the Data Generator is fundamentally the same as the benchmarks' generators; the most substantial difference is its implementation. First, while both UBA and EvoGenerator produce their data by creating a graph for each department in each university, our Data Generator, like that of UOBM's generator - Instance Generator - creates one graph for each university containing all its departments and respective instances. More importantly, all three benchmark generators save the

| | **LUBM** | **EvoGen** |
|---|---|---|
| Departments per university | $15 \leq r \leq 25$ | |
| **Per Department** | | |
| Full professor | $7 \leq r \leq 10$ | |
| Associate professor | $10 \leq r \leq 14$ | |
| Assistant professor | $8 \leq r \leq 11$ | |
| Lecturer | $5 \leq r \leq 7$ | |
| Visiting professor | N/A | $7 \leq r \leq 10$ |
| (Undergraduate) Course | $r \leq 100$ | |
| Graduate course | $r \leq 100$ | |
| Web course | N/A | 100 |
| Chair | 1 | |
| Research | $r \leq 30$ | |
| Research group | $10 \leq r \leq 20$ | |
| Research project | N/A | $10 \leq r \leq 30$ |
| Event | N/A | $15 \leq r \leq 45$ |
| **Ratios** | | |
| Undergrad students to faculty | $8 \leq r \leq 14$ | |
| Undergrad students to visiting faculty | $2 \leq r \leq 3$ | |
| Graduate students to faculty | $3 \leq r \leq 4$ | |
| Graduate students to TA | $4 \leq r \leq 5$ | |
| Graduate students to RA | $3 \leq r \leq 4$ | |
| Undergrad students to advisor | $r \leq 5$ | |
| Undergrad students to visiting advisor | $r \leq 2$ | |
| Graduate students to advisor | $r \leq 1$ | |
| **Publication Allocation** | | |
| Full professor | $15 \leq r \leq 20$ | |
| Associate professor | $10 \leq r \leq 18$ | |
| Assistant professor | $5 \leq r \leq 10$ | |
| Lecturer | $0 \leq r \leq 5$ | |
| Graduate student | $0 \leq r \leq 5$ | |
| **Course Allocation** | | |
| Course per faculty | $1 \leq r \leq 2$ | |
| Graduate course per faculty | $1 \leq r \leq 2$ | |
| Course per undergrad student | $2 \leq r \leq 4$ | |
| Course per graduate student | $1 \leq r \leq 3$ | |
| Course per visiting student | N/A | $2 \leq r \leq 4$ |

Table 6.1: LUBM's and EvoGen's Default Restrictions for Data Generation

generated instances directly to file. Contrarily, we make use of Jena[4] and its libraries to create a graph object for each university before it is saved to file. This facilitates the process of deleting resources and related triples later.

Similarly to the three benchmarks, the user provides as input to our system the number of universities to be generated. Also, similarly to EvoGen, the user inputs a parameter

[4]All Jena binary distributions are available on http://archive.apache.org/dist/jena/binaries/

indicating the number of versions that each university will have. Our Data Generator creates most of the triples using the same recipe as UBA, Instance Generator, and EvoGenerator, however, we have opted to make the following minor changes:

1. Instances describing people:

   (a) The URI of a person: In the three benchmarks, the URI of the person, whether faculty or student is generated containing the department she was created in. So, an example person URI is `http://www.Department1.University0.edu/AssociateProfessor7`. Contrarily, we have chosen to generate a person's URI independently of her university and department, similar to an ORCiD[5]. So, and example person URI is `http://example.org/Person0000000000000000007`.

   (b) The telephone number of a person: In the three benchmarks, the telephone number of all generated person instances always has the String value "xxx-xxx-xxxx". Contrarily, we have chosen to generate a unique telephone number for each person in the form of a sequence of digits.

   (c) The email address of a person: In the three benchmarks, the email address of a person is generated containing the department she was created in. So, an example email address is `AssociateProfessor7@Department1.University0.edu`. Contrarily, we have chosen to generate a person's email address independently of her department, because not all departments within universities provide their faculty and students with email address, while all universities provide email addresses for them.

2. Instances describing courses:

   (a) The name of a course: In LUBM and EvoGen, while the URI of a course is generated containing its university and department, its name doesn't. For example, Course42, offered by department 11 at university 3 and Course42 offered by department 12 at university 3 both share the name "Course42". Contrarily, we have chosen to include the course's department in its generated name. This reflects more real world scenarios where COMP200, which is offered by the Computer Science department, is different than CHEM200, which is offered by the Chemistry department. Course names are still shared across universities.

   (b) The topic of a web course: In EvoGen, all web courses regardless of which department in which university they are offered, have as their topic a String

[5]https://orcid.org/

comprising the word "topic" and a random number between 1 and 150 appended to it. Contrarily, and in the same vein as our change to a course's name, we have chosen to include the course's department in its generated name.

(c) The URL of a web course: In EvoGen, all web courses regardless of which department in which university they are offered, have as their WWW URL a String comprising "http://example.com/webcourse/" and a random number between 1 and 150 appended to it. Contrarily, and in the same vein as our change to a course's name and a web course's topic, we have chosen to include the course's university and department and the course's name in its generated URL.

3. Instances describing publications:

   (a) The URI of a publication: In LUBM and EvoGen, the URI of a publications is generated containing the URI of its first author. So, an example publication URI is `http://www.Department1.University0.edu/AssociateProfesso-r7/Publication3`. Contrarily, we have chosen to generate a publication's URI independently of its first author, whilst still containing its department's URI. So, an example publication URI is `http://www.Department1.University0.edu/Publication0001000000802`. This is similar to UOBM's generation of publications.

   (b) The name of a publication: In LUBM and EvoGen, while the URI of a publication is generated containing its university and department, its name doesn't. For example, Publication42 published by AssociateProfessor7 in department 11 at university 3 and Publication42 published by FullProfessor12 in department 12 at university 3 both share the name. Contrarily, we have chosen to generate a unique name for each publication.

   (c) The ISBN of a publication: In EvoGen, the ISBN of a publication is generated using Java's `System.nanotime` method. While this ensures uniqueness, we have opted to use the generated unique 13 digit number used for its name.

   (d) The date of a publication: In EvoGen, the date of a publication is generated by getting a random number between 1 and 28 for days, a random number between 1 and 12 for months, and a random number between 2000 and 2016 for years. We have chosen to generate a random date after 2007 so it conforms better to the ISBN.

### 6.2.2 Change Producer

EvoGen's Change Creation component is responsible for computing the number of new instances to be created or deleted for each class type. The Change Producer component in our evaluation framework is an exact re-implementation of EvoGen's Change Creation component with one difference. Our Change Producer does not implement its Change Materialisation and Change Creation modules. These two modules are responsible for creating the log files according to the Characteristic Set encoded as per the Change Ontology described in (Meimaris et al., 2014). Since we are not utilising the Change Set or the Change ontology, we have forgone implementing these modules. The remaining two modules in the Change Producer in our evaluation framework have been implemented to reflect EvoGen's Change Creation component without change. This is because our aim was not to change the basis of EvoGen's proposed approach; rather, our aim was to extend EvoGen to do deletions as well as to make it provenance-aware.

We first list the notations and parameters used before describing the modules of the Change Producer.

#### 6.2.2.1 Notations and Parameters

In order to facilitate the description of the Change Producer, we start of by defining the notations used throughout the description.

- $D$: a dataset.
- $D_i$: the $i$th version of the dataset $D$.
- $|D_i|$: the size of $D_i$ which is the number of triples in it.
- $h$: the *shift*, defined as the percentage of change in size between any two versions of $D$, measured using the difference of number of triples.
- $h(D)|_i^j$: the shift between versions $D_i$ and $D_j$. It is defined as $h(D)|_i^j = \frac{|D_j|-|D_i|}{|D_i|}$, where $j > i$.

The change in EvoGen, and subsequently our evaluation framework, is tunable using the following parameters set by the user:

1. The number of total versions for each university: The user can also decide on producing one version per university.
2. The *shift*, $h$: As previously mentioned, this is "the percentage of change in size (measured as triples) between" any two versions of $D$. A positive shift indicates an increase in size between the two version, while a negative shift indicates a decrease

in size between them. The greater the shift, the greater the difference in sizes between versions. When the user provides it as an input, this shift is used to calculate the difference between the size of one version of the dataset and the next by spreading it over the number of total versions. We have chosen to rely on this concept despite its current limitation of evenly distributing the change across all versions.

3. Parameters defined by EvoGen's authors but ignored in its implementation:

   (a) The *monotonicity*: This indicates whether a change incorporates either both insertions and deletions or one of insertions and deletions. While the authors have defined this, they were clear that this parameter is ignored as they do not incorporate both types of updates in their system. We incorporate it as we have extended EvoGen to perform deletions after insertions.

   (b) The option to save either all fully materialised versions or the initial version along with the series of changes: This has not been implemented by the authors; only the inserted triples are saved. Contrarily, we have elected to always produce all the fully materialised versions as well as the series of changes. So, we produce an RDF graph for every version of the university, along with all the changes applied to the previous version to produce it.

   (c) The *schema variation*: We do not incorporate this parameter because it relates to characteristic sets, which we have not implemented.

#### 6.2.2.2 Weight Assignment and Shift Management

As previously mentioned, EvoGen's Change Creation module is responsible for going through each instance type per ontology class within each university and calculating the probability of its change to dictate the number of instances to be created or deleted. This is referred to as its *weight*, and is handled by the Weight Assignment and Shift Management modules. It is based on some parameters set by the user as well as some internal hard-coded configurations. We note the following:

1. The change in the size of the dataset between two consecutive versions $D_i$ and $D_{i+1}$, $change(D_i, D_{i+1})$, is calculated by EvoGen as follows:

   (a) Between the first version of the dataset, $D_0$, and its second version, $D_1$: $change(D_0, D_1)$ is set to the value of the *shift* as input by the user.

   (b) After the first version is created, the change is calculated as: $change(D_i, D_{i+1}) = change(D_{i-1}, D_i) + [(total\ versions) \times (change(D_0, D_1))^2]$

| **Instance Type** | **Weight** |
|---|---|
| Department | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 0.2$ |
| Faculty | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor$ |
| Full professor | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 16$ |
| Associate professor | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 22$ |
| Assistant professor | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 18$ |
| Lecturer | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 11$ |
| Visiting professor | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 18$ |
| Undergraduate student | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 35$ |
| Graduate student | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 24$ |
| Visiting student | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 14$ |
| Undergraduate course | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor$ |
| Graduate course | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor$ |
| Web course | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor$ |
| Research group | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 28$ |
| Project | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 14$ |
| Publication | $\lfloor N^{\underline{o}}$ of existing instances $\times$ $change(D_i, D_{i+1})\rfloor \times 61$ |

Table 6.2: EvoGen's Assignment of Change Probability

2. The probability of change to be applied to each instance type when generating each new version - after the initial version has been produced - is its *weight*. It is calculated as per Table 6.2. There is no clear explanation as to why those specific numbers were chosen.

As it is part of the Change Producer module, and much like the rest of that module, we have opted not to change anything in those modules so we use them as-is, because our aim was not to change the basis of EvoGen's proposed approach; rather, our aim was to extend EvoGen to do the deletions as well as to make it provenance-aware. We have done so by re-implementing it whilst maintaining some continuity.

### 6.2.3 Version Manager

Similar to the Change Producer component, the Version Manager in our evaluation framework is an exact re-implementation of EvoGen's Version Management component with two differences. The first is, in addition to generating the changes applied to evolve the graphs, it generates all the fully materialised versions of the universities as well. The second is that, in case deletions are needed along with insertions, it deletes the triples inserted to the previous version. This is temporary and to realise its full potential, the evaluation framework needs to be extended to delete randomly-chosen data according to their weights. We propose this in our Future Work in Section 8.2.3. The Version Manager would then not be responsible for deleting the previously inserted triples.

### 6.2.4 Provenance

Despite the adoption of provenance becoming more widespread and the emergence of provenance-aware systems becoming more prevalent, to the best of our knowledge, there is still no provenance-aware Semantic Web benchmark which generates both RDF graphs and provenance graphs to be used in evaluating systems that require the presence of both the data and their provenance. Our solution to addressing this shortcoming is to make use of PROV-TEMPLATE (Moreau et al., 2018), as a stand-alone component appended to our evaluation framework to produce provenance graphs.

PROV-TEMPLATE was developed to remove the responsibility of the generation of provenance graphs from systems' other components which performs the systems' functions. Thus, using PROV-TEMPLATE allows the components of a system to be free from any code used solely to generate PROV triples. Instead, a system generates a different sort of data, a set of bindings to be provided to PROV-TEMPLATE to use when generating provenance graphs. To produce provenance graphs which are compliant with PROV, PROV-TEMPLATE is provided with a document, or a template, which contains placeholders, or variables. It is also provided with the aforementioned set of bindings produced by the system. Then, when invoked, PROV-TEMPLATE declaratively binds the template's variables by replacing them with real values.

While the optimal way to make use of PROV-TEMPLATE is to produce the bindings using a system's code, at this current stage, we make use of our pre-prepared templates and binding documents and feed them to PROV-TEMPLATE to generate the provenance graphs.

## 6.3 Summary

In this chapter, we presented the evaluation framework which we have implemented to generate synthetic data based on LUBM, UOBM, and EvoGen and which makes use of PROV-TEMPLATE to generate provenance graphs. We first presented our goals for the evaluation framework. Then we illustrated the evaluation framework's design. Finally, we described each component focusing on how it differs from the three benchmarks it is based on. The evaluation framework fills the gap of synthetic evolving data generation accompanied with provenance graphs.

In the next chapter we test our approach of update propagation using data generated by the evaluation framework and present and discuss the results.

# Chapter 7

# Evaluation and Discussion

In this chapter, we turn our attention to evaluating our update propagation approach implemented in the system described in Chapter 5 by using the data produced using the evaluation framework described in the previous chapter. We do so as follows. First, we describe the evaluation criteria we will be using to assess our system. We also list and describe the dimensions we view as needed for a thorough evaluation. While we do not use all those dimensions, due to space and time restrictions, we argue that their usage enable a full evaluation of approaches similar to ours. Next, we describe the datasets generated using the evaluation framework described in the previous chapter and that we used to test our approach. Finally, we present our results and discuss them.

This chapter consists of four parts. In Section 7.1, we describe our evaluation criteria and list and describe the dimensions for evaluating update approaches. In Section 7.2, we describe the datasets used to evaluate our approach. In Section 7.3, we present and discuss the results. Finally, in Section 7.4, we summarise this chapter.

## 7.1 Evaluation Criteria and Dimensions

In this section, we present the evaluation criteria we will be using to assess our system. We also list and describe the dimensions we view as needed for a thorough evaluation. While we do not use all those dimensions, due to space and time restrictions, we argue that their usage enable a full evaluation of approaches similar to ours.

### 7.1.1 Evaluation Criteria

The evaluation criteria we chose aim to address our second research question, 'How can the recorded provenance of a derived RDF graph facilitate scalable partial re-derivation by generating less overhead in update communication and re-entailment?'. There are three criteria as follows. The first criterion is concerned with communication overhead. It studies where our approach saves on the number of triples to be communicated to our system from the systems managing the source graphs. The second criterion presents a trade-off with the first criterion. It studies where our approach saves on the number of triples, which are copies of those of the source graphs', to be stored in our system. Finally, the third criterion is concerned with re-derivation. It studies where our approach saves on the number of triples involved in the set theoretic operations and in re-entailment.

The criteria are summarised in the list below:

1. EC1 - Communication: retrieving the update is less overhead than retrieving both source graphs.
2. EC2 - Storage: the possibility of avoiding storing source graphs.
3. EC3 - Execution: propagating the update results in less triples processed during:
    (a) EC3a - The set theoretic operation.
    (b) EC3b - Re-derivation: this is related to the number of triples sent to the reasoner after an update.

### 7.1.2 Evaluation Dimensions

The dimensions involved in evaluating update approaches similar to ours are as follows:

1. Sizes of the initial graphs: The size of the initial graph plays a role in assessing the performance of update approaches. A full evaluation would look at how the performance varies among small, medium, and large graphs. While the exact mappings of these sizes to number of triples may vary depending on individual preferences, we consider a graph to be small if it contains less than 50,000 triples, medium if it contains less than 250,000 triples, and large if it contains 1 million triples or more. Others may choose different mappings of sizes to number of triples.
2. Sizes of the update graphs: Similar to the size of the initial graphs, the size of the update also plays a role in assessing the performance of approaches, and a full evaluation would look at how the performance varies among small, medium, and large update graphs. While the exact mappings of these sizes to number of triples in the update graph may also vary depending on individual preferences, we consider an update graph to be small if it contains less than a quarter of the number of triples contained in the initial graph, medium if it contains less than half the number of triples contained in the initial graph, and large if it contains almost as much triples in the initial graph or more. Others may chose different mappings of

sizes to number of triples. The nine combinations of both initial graph and update graph sizes also provide a more comprehensive evaluation of the performance of an update approach.

3. Sizes of the provenance graphs: The size of the provenance graphs plays a role in assessing the storage requirements of the approaches. A full evaluation would look at how storage needs vary among the absence of provenance graphs, the size of a provenance graph containing triples describing the graphs in the dataset, i.e. graph-based provenance, and the size of a provenance graph containing triples describing the triples in each graph in the dataset, i.e. triple-based provenance. While our evaluation framework is provenance aware on the graph-level, we do not take this dimension into consideration when evaluating our approach.

4. The connections among instances: Similar to the sizes of graphs, the connections among instances contained in the graphs also plays a role in assessing the performance of update approaches, and a full evaluation would look at how the performance varies among minimal connections, moderate connections, and high connections among instances. At the moment, we do not take this dimension into account due to two reasons. First, the connections are produced according to the scaling of UOBM, which in addition to being somewhere between minimal and moderate, is constant. Second, the types of instances contained in updates depend on EvoGen's assignment of weights. As shown in Table 6.2, the instance types which ends up being more prevalent in updates is the publications and its subclasses, which do not contribute to connections among the different graphs.

5. Ontology complexity: Similar to the connections among instances, the complexity of the algorithm also plays a role in assessing the performance of update approaches, and a full evaluation would look at how the performance varies among minimal complexity, moderate complexity, and high complexity of the ontology. At the moment, we do not take this dimension into account because our evaluation framework makes use of EvoGen's ontology, which is moderate in size and complexity.

6. The type of set theoretic operation: The type of set theoretic operation used to produce the initial graph influences the number of triples to be processed when an update is propagated. Coupled with how the instances are connected among the graphs, the operation type also affects the number of processed triples. As we are not taking the degrees of connections between instances into consideration, we are only concerned with which of the four previously identified set theoretic operations has been performed.

7. The type of update applied on the graphs: Whether an update is an insert or a delete also influences the number of triples to be processed.

We remind the reader that we do not take all the dimensions into consideration when evaluating our approach, this is because the total number of experiments needed to be run to evaluate our approach would be 1296[*] and another 1296 to test the naïve graph re-creation approach and be able to compare the results, totalling 2592 runs. Instead we perform one test per set theoretic operation per initial graph size with one update graph size, resulting in 24[†] runs along with another 24 runs to compare with, totalling 48 tests.

## 7.2 Experimental Data

For the purpose of testing the system we presented Chapter 5, we have used our evaluation framework to create two universities with the second having triples added to it and then removed from it. We have combined the generated data about the two universities with a modified university schema, originating from the LUBM schema[3] but containing only RDF descriptions, i.e. any object or data properties are converted to RDF properties and OWL restrictions and equivalent class definitions have been removed. Additionally, descriptions of the 10 new classes and 19 new properties introduced in EvoGen have been added to the ontology, bringing the total to 53 classes and 51 properties.

We grouped the tests into three bundles depending on the sizes of the graphs. Each group contains the results of inserting then deleting taking place after performing the four set theoretic operations. Recall that we are deleting the same triples we had inserted into $G_{C'}$.

The small graph $G_A$ contains ≈50K triples, while the small graph $G_B$ contains ≈40K triples. There are ≈37K triples to be inserted into $G_B$, resulting in graph $G_{B'}$ having ≈77K triples. The sizes of the produced graphs $G_C$, $G_{ent(C)}$, $G_{C'}$ and $G_{ent(C')}$ are shown in Tables 7.1. Since the same triples that had been inserted are then deleted, the sizes of the final base graph and final entailed graph revert to the original sizes after creation, i.e. those of $G_C$ and $G_{ent(C)}$ respectively.

The medium graph $G_A$ contains ≈300K triples, while the medium graph $G_B$ contains ≈242K triples. There are ≈223K triples to be inserted into $G_B$, resulting in graph $G_{B'}$ having ≈465K triples. The sizes of the produced graphs $G_C$, $G_{ent(C)}$, $G_{C'}$ and $G_{ent(C')}$ are shown in Table 7.2. Since the same triples that had been inserted are then deleted, the sizes of the final base graph and final entailed graph revert to the original sizes after creation, i.e. those of $G_C$ and $G_{ent(C)}$ respectively.

[*]Calculated as *Initial Graph Size* (3) × *Update Graph Size* (3) × *Provenance Graph Size* (2) × *Connectivity Level* (3) × *Ontology Complexity* (3) × *Set theoretic Operation Type* (4) × *Update Type* (2) = 1296.

[†]Calculated as *Initial Graph Size* (3) × *Update Graph Size* (1) × *Set theoretic Operation Type* (4) × *Update Type* (2) = 24.

[3]http://swat.cse.lehigh.edu/onto/univ-bench.owl

Table 7.1: Size of Small Graphs Initially and After Insert.

| | Sizes after Creation | | Sizes After Insert | |
|---|---|---|---|---|
| ST | $\lvert G_C \rvert$ | $\lvert G_{ent(C)} \rvert$ | $\lvert G_{C'} \rvert$ | $\lvert G_{ent(C')} \rvert$ |
| Union | ≈87K | ≈130K | ≈120K | ≈180K |
| Intersection | ≈3K | ≈4K | ≈5K | ≈7K |
| Difference 1 | ≈48K | ≈70K | ≈45K | ≈66K |
| Difference 2 | ≈37K | ≈56K | ≈73K | ≈110K |

Table 7.2: Size of Medium Graphs Initially and After Insert.

| | Sizes after Creation | | Sizes After Insert | |
|---|---|---|---|---|
| ST \Sizes | $\lvert G_C \rvert$ | $\lvert G_{ent(C)} \rvert$ | $\lvert G_{C'} \rvert$ | $\lvert G_{ent(C')} \rvert$ |
| Union | ≈540K | ≈800K | ≈760K | ≈1.2M |
| Intersection | ≈6K | ≈8K | ≈10K | ≈13K |
| Difference 1 | ≈300K | ≈440K | ≈300K | ≈440K |
| Difference 2 | ≈240K | ≈360K | ≈460K | ≈700K |

Table 7.3: Size of Large Graphs Initially and After Insert.

| | Sizes After Creation | | Sizes After Insert | |
|---|---|---|---|---|
| ST \Sizes | $\lvert G_C \rvert$ | $\lvert G_{ent(C)} \rvert$ | $\lvert G_{C'} \rvert$ | $\lvert G_{ent(C')} \rvert$ |
| Union | ≈2.3M | ≈3.5M | ≈3.25M | ≈5M |
| Intersection | ≈25K | ≈33.6K | ≈41K | ≈55K |
| Difference 1 | ≈1.27M | ≈1.86M | ≈1.27M | ≈1.86M |
| Difference 2 | ≈1.03M | ≈1.5M | ≈2M | ≈3M |

The large graph $G_A$ contains ≈1.27M triples, while the large graph $G_B$ contains ≈1.03M triples. There are ≈950K triples to be inserted into $G_B$, resulting in graph $G_{B'}$ having ≈2M triples. The sizes of the produced graphs $G_C$, $G_{ent(C)}$, $G_{C'}$ and $G_{ent(C')}$ are shown in Table 7.3. Since the same triples that had been inserted are then deleted, the sizes of the final base graph and final entailed graph revert to the original sizes after creation, i.e. those of $G_C$ and $G_{ent(C)}$ respectively.

## 7.3 Results and Discussion

We now present and discuss our results and show that there is indeed less overhead in applying our approach as detailed below.

### 7.3.1 Evaluation Criteria 1: Communication

When the update is an Insert, the size of update will always be less than the size of the whole graph. Hence, there is less communication overhead.

The number of triples to be fetched per our experiment is shown in Table 7.4

Table 7.4: Comparison of Number of Fetched Triples.

| | Size of $G_{B'}$ | Size of $G_{B'}^{up}$ | % of Difference |
|---|---|---|---|
| Small Graphs | ≈77K | ≈37K | ≈48% |
| Medium Graphs | ≈465K | ≈223K | |
| Large Graphs | ≈2M | ≈950K | |

However, when the update is a Delete, the overhead of communicating the update is acceptable unless more than half of the triples in the graph are to be deleted. In this case, the size of the update becomes greater than the size of the new source graph, and it may be more preferable for the system to retrieve $G_{B'}$ than to retrieve the update graph $G_B^{up}$. As we saw in the analysis of the update propagation in Section 4.2.2, in the case of intersection and the second case of difference, there is no need to store $G_A$, thus retrieving $G_{B'}$ will force the re-retrieval of $G_A$ - if it is not stored in the system - and the generation of $G_{C'}$ from scratch. Together, this would cause more overhead depending on the availability and on the comparative size of $G_A$. Hence, retrieving $G_{B'}$ would be more preferable. In the case of union and the first difference case where we may need to re-retrieve $G_A$ - if it is not stored in the system, it may be more beneficial to retrieve $G_{B'}$ instead of update graph $G_B^{up}$. This boils down to a case-by-case bases and can be alleviated by requesting the size of the update from system $B$ and depends on whether the other source graph needs to be retrieved as well.

Since we delete the same triples that were inserted, the overhead saved is the same as that of the insert, shown in Table 7.4.

### 7.3.2 Evaluation Criteria 2: Storage

The storage of source graphs is not needed in half of the cases. In the other half of the cases, when a source graph is needed in the system, it may be retrieved. This leads to a trade-off between storage and retrieval. It is further complicated when a chain of updates is applied, we discuss this further in Appendix D.

### 7.3.3 Evaluation Criteria 3: Execution

#### 7.3.3.1 Evaluation Criteria 3a: Set theoretic operations

We were not able to use Jena to count the number of triples processed during set theoretic operations. However, from our analysis in Section 4.2.2, we see that there are less triples to be checked because we are at most using the update and one source graph and not the entirety of both source graphs.

#### 7.3.3.2 Evaluation Criteria 3b: Re-derivation

We note that there is a consistency in the reduction of the number of triples being sent to the reasoner. This is because, as we mentioned before, the scaling of instance generation and the connections among instances are constant.

**Union:** Inserting or deleting part of the update and then re-deriving by only taking into account the affected triples and those related to them results in sending to the reasoner 38% and 26% of the total number of triples that would have been sent to the reasoner respectively. This is shown in Figures 7.1 and 7.2.

**Intersection:** Inserting or deleting part of the update and then re-deriving by only taking into account the affected triples and those related to them results in sending to the reasoner 64% and 81% of the total number of triples that would have been sent to the reasoner respectively. This is shown in Figures 7.3 and 7.4.

**Difference 1:** Inserting or deleting part of the update and then re-deriving by only taking into account the affected triples and those related to them results in sending to the reasoner 34% and 30% of the total number of triples that would have been sent to the reasoner respectively. This is shown in Figures 7.5 and 7.6.

**Difference 2:** Inserting or deleting part of the update and then re-deriving by only taking into account the affected triples and those related to them results in sending to the reasoner 62% and 55% of the total number of triples that would have been sent to the reasoner respectively. This is shown in Figures 7.7 and 7.8.

### 7.3.4 Additional Observations

#### 7.3.4.1 Datasets' Shortcomings

We note that the results obtained in the previous section are influenced by the datasets produced using our evaluation framework because of the following two reasons.

First, the university ontology, while moderate in size and complexity, has at most four degrees of class subtypes and two degrees of property subtypes. Recall that entailment is done according to the RDFS entailment regime, which was described in Section 2.1.3.2, and that the Delete and Re-entail Algorithm is based on the regime's patterns. This affects the re-derivation after deletion results as the algorithm would produce better results with a simpler ontology and worse results with a more complex ontology.

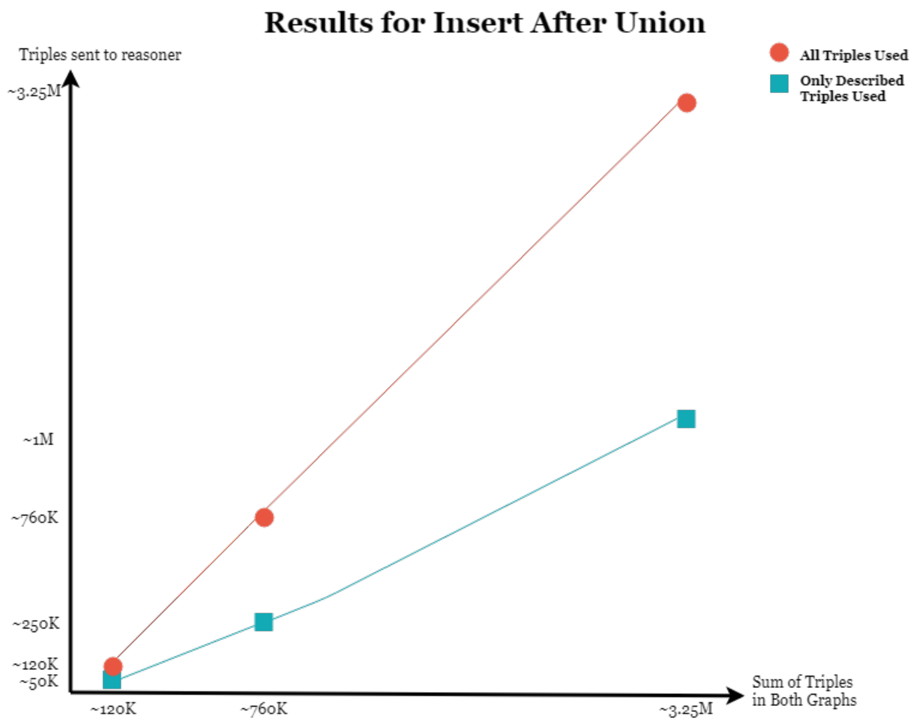


Figure 7.1: Comparison of Triples Sent to Reasoner - Insert After Union.

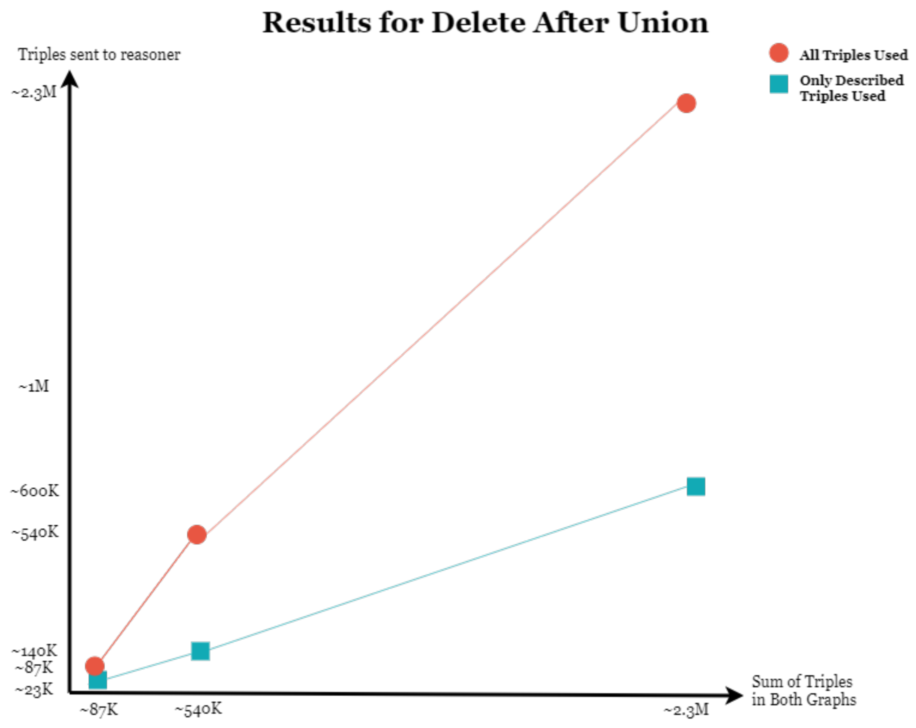


Figure 7.2: Comparison of Triples Sent to Reasoner - Delete After Union.

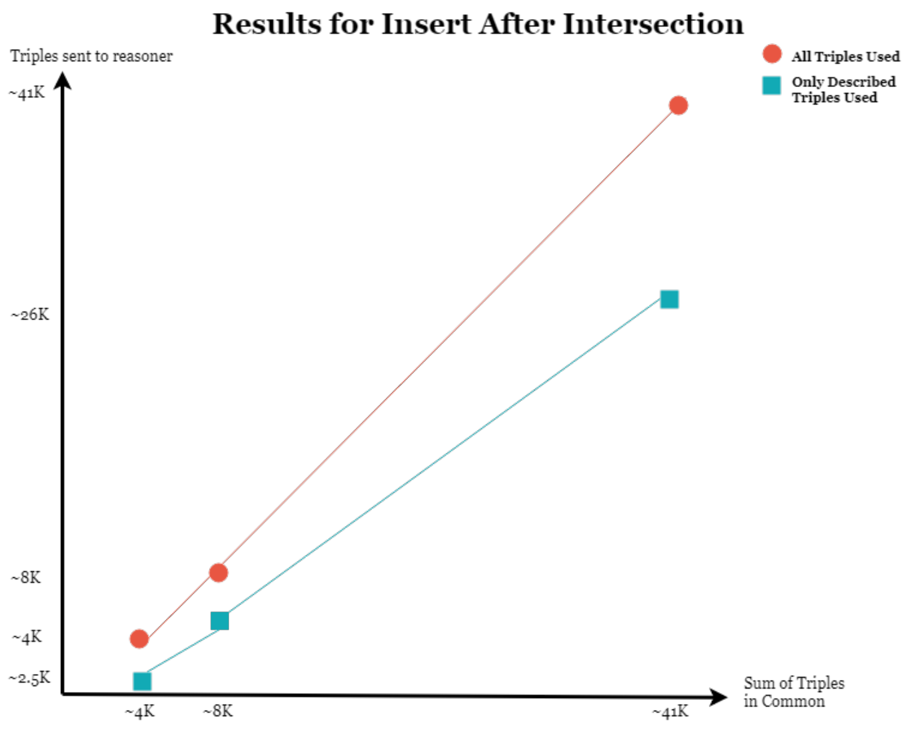


Figure 7.3: Comparison of Triples Sent to Reasoner - Insert After Intersection.

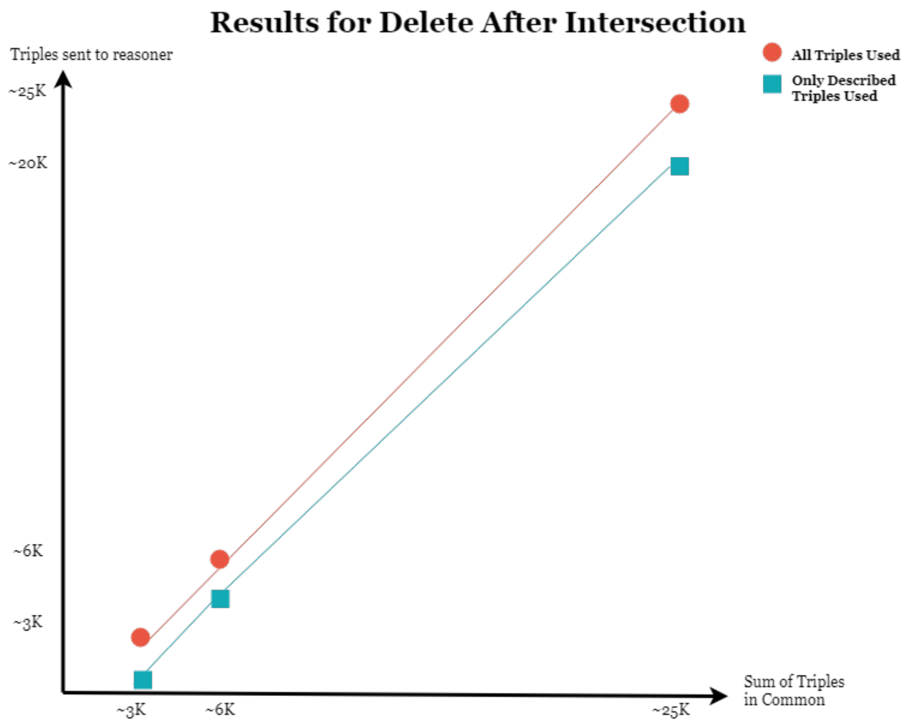


Figure 7.4: Comparison of Triples Sent to Reasoner - Delete After Intersection.

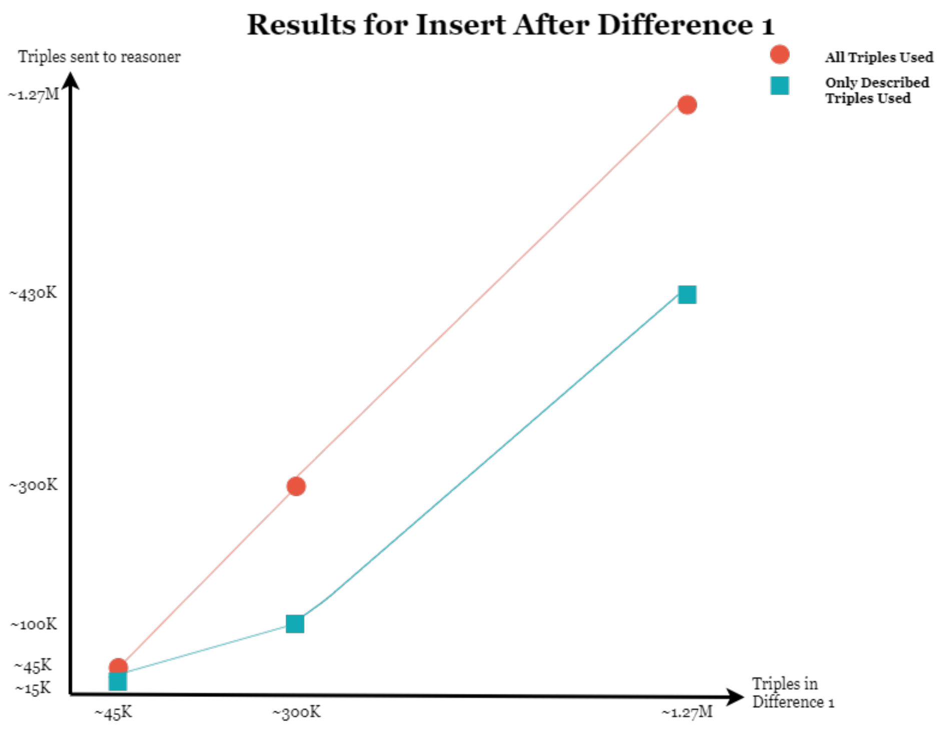


Figure 7.5: Comparison of Triples Sent to Reasoner - Case Insert After Difference 1.

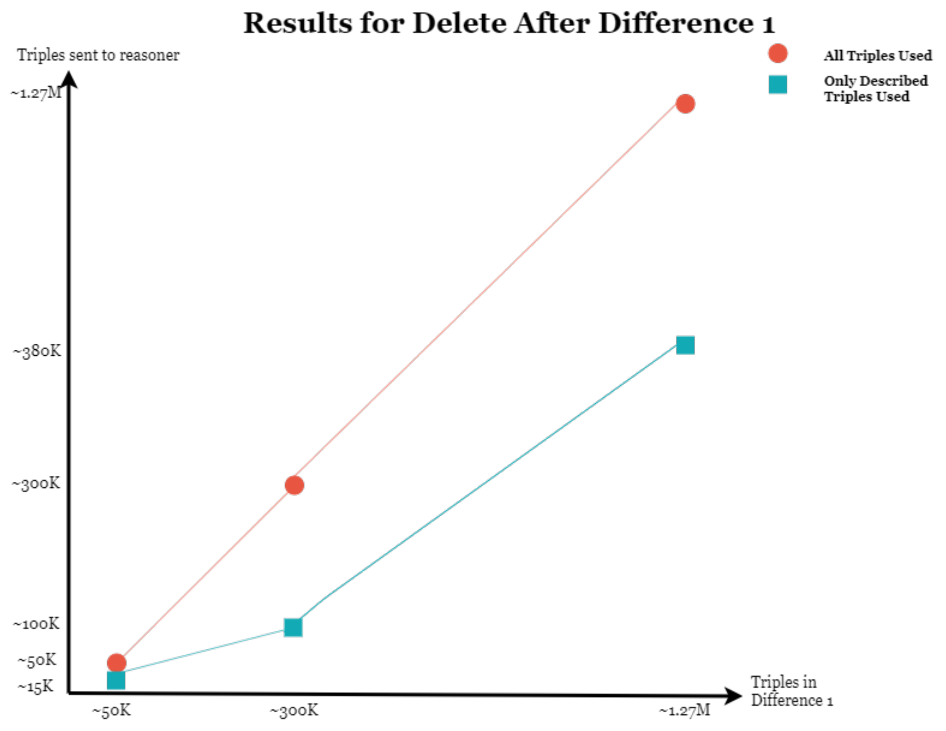


Figure 7.6: Comparison of Triples Sent to Reasoner - Case Delete After Difference 1.

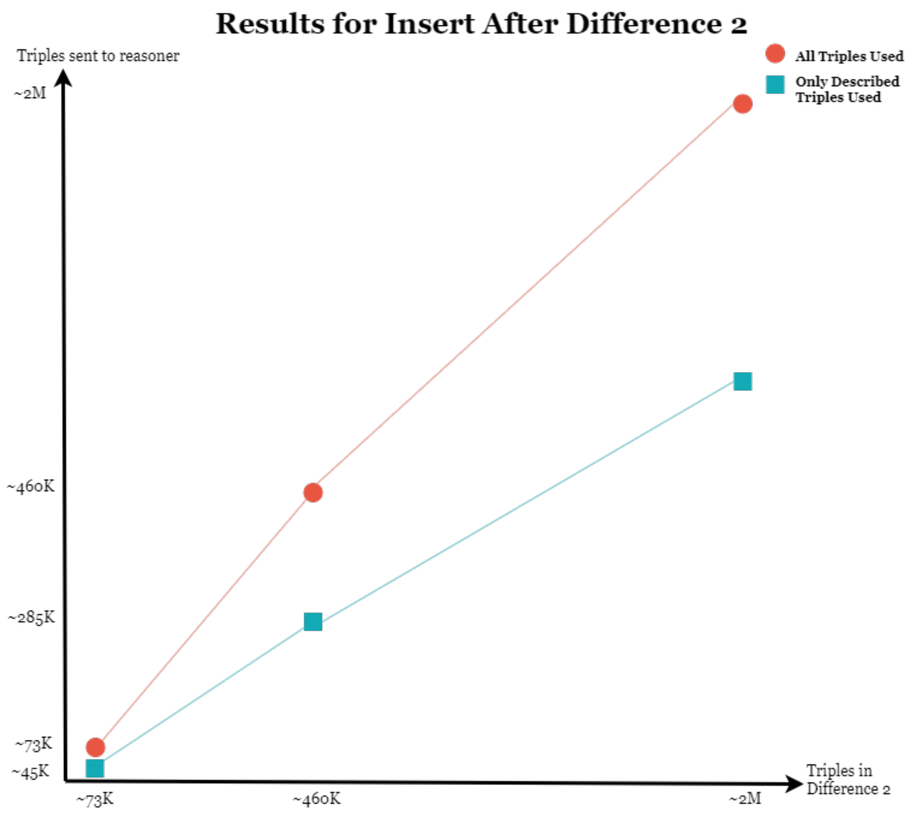


Figure 7.7: Comparison of Triples Sent to Reasoner - Case Insert After Difference 2.

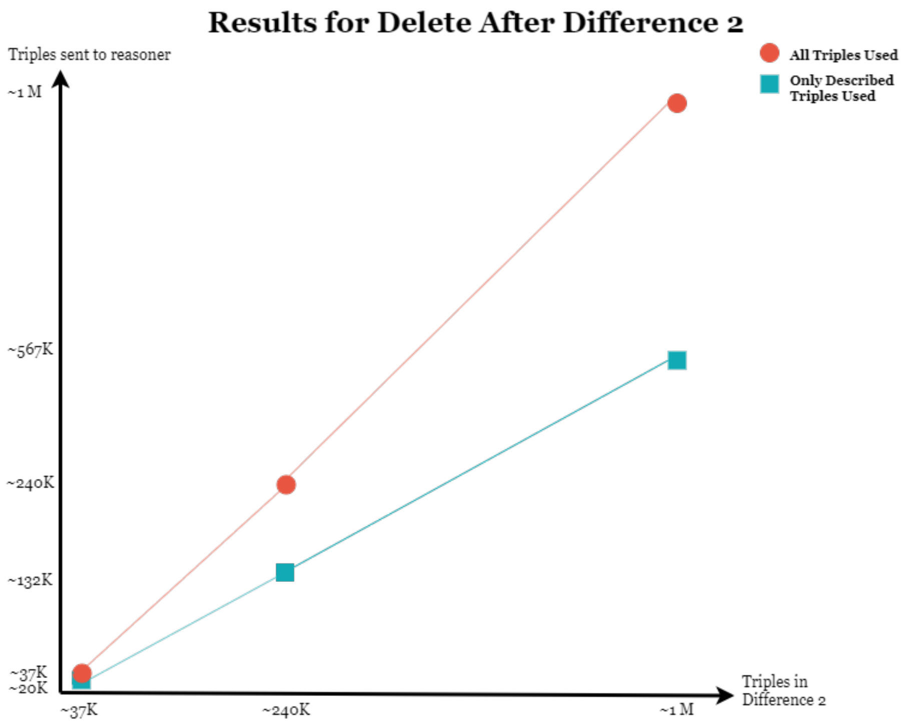


Figure 7.8: Comparison of Triples Sent to Reasoner - Case Delete After Difference 2.

Second, the datasets produced suffer shortcomings arising from the evaluation framework's default configuration and restrictions; specifically, the instances to be inserted or deleted depend on the following. First, both the maximum number of instances to be generated per class type and the ratios of some instances to others are hard-coded - as described in Section 6.2.1 and summarised in Table 6.1. Second, the probability that a change is applied to an instance type is also calculated based on hard-coded values - as described in Section 6.2.2 and summarised in Table 6.2. This results in update graphs where instances of certain types are disproportionally higher than others, especially instances of type 'publication'. This is most obvious in the case of re-derivation after delete on graph $G_C$ produced from intersection. Because the triples sent to the reasoner are all those that were already in the graph $G_C$ minus those in the update graph $G^{up}$, and since $G^{up}$ contains a high proportion of triples not shared, a large proportion of $G_C$ ends up being used for re-derivation. The worst case scenario would be when none of the triples in $G^{up}$ to be deleted are present in $G_C$, resulting in all of $G_C$ being used for re-entailment. Similar effects can be seen in the difference cases but to a lesser extent.

#### 7.3.4.2 Provenance Cost

Because we developed RGPROV with the intent of keeping it as light-weight as possible, the cost associated with managing such graph-based provenance produced inside the system is minimal. We list below the number of triples created at each stage of creating $G_{ent(C)}$ and updating it to $G_{ent(C')}$:

1. Creating $G_{ent(C)}$, a total of 42 triples are created as follows:
    (a) After fetching the source graphs: 22 triples are created - independent of the triples inside the provenance graphs of $G_A$ and $G_B$.
    (b) After the set theoretic operation: 10 triples are created.
    (c) After entailment: 10 triples are created.
2. Propagating an update, a minimum of 31 triples are created, with the maximum being 37 triples, as follows:
    (a) After fetching the update graph: 12 triples are created - independent of the triples inside the provenance graph of $G_B^{up}$. If a copy of the new graph $G_{B'}$ is also fetched then 1 additional triple is created.
    (b) After the update is reflected: either 9 triples or 14 are created, depending on whether the whole update graph was used or a subset of it.
    (c) After re-entailment: 10 triples are created.

Therefore, the total size of producing graph $G_{ent(C')}$ ranges between 73 and 79 triples - in addition to the triples copied from the provenance graphs of the source graphs.

We therefore consider this cost negligible when considering the two other possible ways a graph may be re-derived. In the first case, triple-based provenance is used, i.e. provenance is tracked on the triple level. While this results in minimal re-derivation costs, we have argued in Section 2.5.3 that it comes at the hight cost of managing provenance, as the size of the provenance graph would be at least as large as the graph. In the second case, where provenance is not used, i.e. the provenance cost is 0, the graph needs to be created from scratch and entailments need to be redone. As shown in the previous section, compared to this approach, our solution saves on both communication and re-derivation overhead and only requires 79 additional triples.

#### 7.3.4.3 Implementation-Specific Issues

We realised that the implementation of the update propagation algorithm needs to be further improved as its run-time is currently hindered in two ways when the update graph contains more than 1000 triples.

The first way is related to inserting or deleting from graphs on the Fuseki server. When there is a large number of insert or delete statements to be executed, our first approach was to create one insert or delete statement containing all the affected triples. However, this caused a Java Stack Over Flow Error. To avoid this, we split the update statements and group them into a collection of inserts or deletes having a manageable size. However, those additional update requests sent to the server end up taking some additional time due to the communication between the system and the Fuseki server[4].

Second, the algorithm currently loops over the triples to be inserted or those to be deleted and requests their describe statements one by one from the Fuseki server. Sending all those describe requests to the server one after the other causes Fuseki to run out of ports to listen on. In Jena's current implementation, closing the connection does not actually send a close request to the server's port. To avoid the resulting Java Bind Exception, we forced the system to go to sleep for a few second after every 1000 describe requests sent to the Fuseki server. This workaround is obviously not optimal, but has brought our attention that even if there were enough ports to accept all created connections, there will always be additional time spent on communicating all the describe statements individually to the server.

[4]This communication between the components inside our system is separate from the communication between our system and the systems where the source graphs are published, hence why we did not include this observation and the next within Evaluation Criteria 1.

## 7.4 Summary

In this chapter, we first described our evaluation criteria. Then we presented the datasets that were generated using the evaluation framework described in Chapter 6 to test our update propagation approach. Next, we presented our experimental results showing that, under the current default configurations, partial re-derivation based on select parts of the update performed on a source graph consumes less resources whilst requiring minimal provenance management costs. Specifically, there is an average of 48% less triples being communicated when an update happens. Moreover, when the update is an Insert, there are between 36% and 66% less triples being processed by the reasoner. Lastly, when the update is a delete, there are between 19% and 73% less triples being processed by the reasoner. Finally, we discussed the datasets' shortcomings and how different factors would affect the aforementioned results.

# Chapter 8

# Conclusions and Future Work

This thesis looked into how the light-weight and high-level provenance of a derived RDF graph on the Semantic Web can enable its scalable partial re-derivation when one of its source graphs changes compared to the naïve approach of recreating it. In this chapter, we summarise the thesis and then discuss how our work can be extended in the future.

## 8.1 Conclusions

In Chapter 3, we started by examining what documentation is needed to track the provenance of derived RDF graphs which use the entirety of other source graphs, from their initial creation and through their modification. We did so as follows. We presented a scenario and explored a running example where a derived RDF graph is created using data from two other graphs and impressed upon the need to propagate changes applied to those source graphs. In order to track the provenance of derived graphs on the Semantic Web and facilitate the propagation of their modification, we presented the RGPROV vocabulary, a specialisation of the PROV-O ontology that models the classes and properties involved in a graph's creation and update. We split those into four categories according to the activity that produces a graph. The first describes how the provenance of a graph fetched from an outside system is represented. The second describes how the provenance of a graph produced using a set theoretic operation is represented. The third describes how the provenance of a graph produced using entailment is represented. Finally, the fourth describes how the provenance of a graph produced using an update operation is represented. RGPROV addresses our first subsidiary research question (SRQ1) and constitutes our first contribution.

Then, in Chapter 4, we looked into how an update on a source graph needs to be propagated in an entailed graph derived from it. We studied which triples, inserted into or deleted from a source graph, needs to be propagated through to the derived graph.

This is based on the combination of the set theoretic operation applied initially to the source graphs and whether the update applied to the source graph is an insert or a delete. To manage these stages efficiently, we made use of the proposed vocabulary, RGPROV, and detailed how it can be applied in our running example. This chapter addresses our first research question and our second subsidiary research question (SRQ2).

Next, in Chapter 5, we described the model we had implemented to test our approach of efficient update propagation. The described system contains three algorithms. The first makes use of RGPROV to determines whether or not the whole update needs to be propagated to the entailed graph. If only a part of it is to be propagated, then it determines which triples are to be used. The second and third algorithms are RDF graph partial re-derivation algorithms based on the Delete and Rederive (DRed) algorithm and tailored to RDF graphs. To the best of our knowledge, this is the first time DRed has been used in a Semantic Web context, and thus the algorithms constitute our second contribution. The model we presented constitutes our third contribution.

Afterwards, in Chapter 6, we moved on to presenting the goals of a framework that would adequately evaluate our system. The goals were arbitrary scaling of data, an ontology of moderate size and complexity, graphs of different sizes ranging from small to very large, effective instance links, and dynamic graphs, as well as for the framework to be provenance-aware and produce provenance graphs of different sizes. We then described the design of the evaluation framework we implemented to achieve these goals, which is based on LUBM, UOBM, and EvoGen and which makes use of PROV-TEMPLATE. The evaluation framework we presented constitutes our forth and final contribution.

Finally, in Chapter 7, we discussed our evaluation criteria, described the datasets that we used in our experiments, presented our results, and offered our observations on the short-comings of the datasets and implementation issues. We showed that, under the current default configurations, partial re-derivation based on select parts of the update performed on a source graph consumes less resources whilst requiring minimal provenance management costs. Specifically, there is an average of 48% less triples being communicated when an update happens. Moreover, when the update is an Insert, there are between 36% and 66% less triples being processed by the reasoner. Lastly, when the update is a delete, there are between 19% and 73% less triples being processed by the reasoner. The results presented address our second research question.

## 8.2 Future Work

Based on the work we presented, there are a few directions worth exploring that extend our work.

### 8.2.1 Graph Operations

The operations that produce a derived graph in our system are limited and we acknowledge that there are additional operations that we have not taken into consideration.

#### 8.2.1.1 Beyond Set Theoretic

We have restricted the operations performed on the copies of the two source graphs to the set theoretic union, intersection, and difference, and we have also limited the usage of those copies to their entirety. A graph, however, may be produced in more than those ways. Our system may be extended to use a portion of the copy of a source graph by filtering out undesired triples. This would need to be reflected after an update on a source graph when choosing triples to be propagated to the source graph. Moreover, after the two copies of the source graphs - or parts of them - have been combined, compared, or contrasted, a user may wish to further filter out some triples, or insert some of their triples into the produced graph. This would also need to be reflected after an update on a source graph when choosing triples to be propagated to the source graph. Finally, we have also limited the number of source graphs to only two. However, a user may wish to work with three or more graphs. Again, this would also need to be reflected after an update on a source graph when choosing triples to be propagated to the source graph.

#### 8.2.1.2 Different Entailment Regimes

Our system may be extended to support more than just the RDFS entailment rules by also expanding the deletion and re-derivation algorithm to take into account the different OWL entailment regimes. Additionally, our system currently only uses source graphs containing base triples, i.e. non-inferred triples. It may be extended to allow it to use source graphs that may contain inferred triples. This, however, brings about two complexities that need to be taken into consideration. The first is that the delete and re-derive algorithm would need to look into each triple to see whether it is inferred or not. While this is straightforward, it adds time expenses. The second is that this may give rise to a situation where one source graph was produced using one entailment regime while the other source graph was produced using a different entailment regime and the derived graph from the two is produced using yet a third entailment regime. To address this problem of heterogeneous entailments, the extended system needs to implement a method to reconcile what is to be entailed.

### 8.2.2 Time-Sensitivity and Streaming

Our system may be extended to handle time-sensitive information; i.e. data which at a certain pre-known point in time in the future become automatically invalid. This could occur if a certain property has a time limit, for example, the fixed term membership of parliament. Extending the system to cater to such a scenario would require, in addition to a new component, introducing a new vocabulary that can describe the current and future validity of information as described by Della Valle et al. (2009). Current systems which propose such vocabularies cater more towards sensor systems. It would also require handling of more fine-grained provenance than our system does. While the entirety of the provenance graphs may not necessarily need to be as fine-grained as provenance information used to maintain ontological entailments with data streams, it would still add storage expenses.

### 8.2.3 Benchmarking the Evaluation Framework

The evaluation framework we developed is limited in its current state. First, it only deletes previously inserted data. We would like to extend it to delete random data chosen by its Change Producer module according to their weights, similar to how insertions are done. This would lead to the removal of its second limitation; in its current state the evaluation framework affords the user one of two options for evolving datasets: monotonic insertions or insertions followed by deletions. By deleting randomly-chosen data, the framework would provide the user with an additional third option of monotonic deletions.

Moreover, the evaluation framework may be extended to afford the user the option to produce modified information about instances and relations. Thus far, we have only discussed inserting into or deleting from a dataset. A user may wish for more changes than those offered by the current insertions and deletions. We have compiled a list of suggested modifications in Appendix C. By allowing for modifications, the framework would provide the user with an additional fourth option for evolving datasets, that of modifications.

Additionally, the evaluation framework, in its current state, makes use of pre-prepared binding templates and provides those to PROV-TEMPLATE to generate PROV compliant provenance graphs. It may be expanded to produce those bindings dynamically within the framework's code while the datasets are being generated.

Finally, the evaluation framework we developed can be extended to become a benchmark for testing reasoning and querying performance as well as storage scalability. To bring it to the such a standard, queries need to be introduced. Such queries would include those of LUBM and UOBM as well as queries for both the evolving data and the provenance. We have compiled a list of suggested categories of queries in Appendix C.

### 8.2.4 Storage of Old Graphs and Provenance Elision

We have previously alluded to the issue of having to deal with older copies of changing graphs. This is twofold and pertains to both RDF graphs and the provenance graphs that describe them.

The old RDF graphs in the system may be old copies of source graphs or old copies of graphs produced by the system. We study which graphs are required in case a system needs to deal with chains of different updates. This is shown in Appendix D.

Keeping all previous copies of a graph and its previous provenance would result in maximal storage expenses that would keep on growing. However, and while deleting all old graphs would result in minimal storage expenses, one needs to also consider what would happen to the most recent provenance graph that describes the current version of an RDF graph. In our current approach, when creating the most recent provenance graph, we make a copy of the older provenance graph and insert into it the provenance that describes how the most recent RDF graph was created. Thus, such a provenance graph could end up describing parts of an RDF graph that initially contributed to a previous version of it despite those parts not influencing the most recent version of that graph. Moreover, this leads to a provenance graph that continues to increase in size. Along those lines, Pimentel et al. (2018)'s reference sharing and checkpoints may be adapted to address this limitation.

Finally, one may need more information describing the versioning of the provenance graphs. In the current version of RGPROV, versioning is not documented. Building on Halpin and Cheney (2014), RGPROV may be extended to include relations similar to UPD's version and prevVersion. Thus, we propose the addition of the relations currentVersion, previousVersion, and firstVersion.

# Appendix A

# RGPROV Ontology

```
@base <http://www.ecs.soton.ac.uk/rgprov> .
@prefix : <http://www.ecs.soton.ac.uk/rgprov#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix prov: <http://www.w3.org/ns/prov#> .

<http://www.ecs.soton.ac.uk/rgprov#> a owl:Ontology ;
owl:imports <http://www.w3.org/ns/prov-o> ;
rdfs:comment "A specialization of PROV-O"@en .

######################################################################
#    Classes
######################################################################

###  http://www.ecs.soton.ac.uk/rgprov#Graph
:Graph a owl:Class ;
rdfs:subClassOf prov:Entity ;
prov:definition "An RDF or OWL graph" .

###  http://www.ecs.soton.ac.uk/rgprov#UpdateGraph
:UpdateGraph a owl:Class ;
rdfs:subClassOf :Graph ;
prov:definition "A subclass of rgprov:Graph that represents the graphs
whose triples are to be inserted or deleted." .

###  http://www.ecs.soton.ac.uk/rgprov#Fetch
```

```
:Fetch a owl:Class ;
rdfs:subClassOf prov:Activity ;
owl:disjointWith :GraphOperation ;
prov:definition "An activity that indicates that a fetch (copy) operation
has taken place." .

### http://www.ecs.soton.ac.uk/rgprov#GraphOperation
:GraphOperation a owl:Class ;
rdfs:subClassOf prov:Activity ;
prov:definition "An activity that was performed on at least one
rgprov:Graph to produce another graph." .

### http://www.ecs.soton.ac.uk/rgprov#Intersection
:Intersection a owl:Class ;
 rdfs:subClassOf :GraphOperation ;
 owl:disjointWith :Union ;
 prov:definition "An activity that was performed on at least two
 rgprov:Graphs to produce their intersection graph." .

### http://www.ecs.soton.ac.uk/rgprov#Union
:Union a owl:Class ;
rdfs:subClassOf :GraphOperation ;
prov:definition "An activity that was performed on at least two
rgprov:Graphs to produce their union graph." .

### http://www.ecs.soton.ac.uk/rgprov#Difference
:Difference a owl:Class ;
rdfs:subClassOf :GraphOperation ;
owl:disjointWith :Intersection , :Union ;
prov:definition "An activity that was performed on exactly two
rgprov:Graphs to produce their difference." .

### http://www.ecs.soton.ac.uk/rgprov#InsertOperation
:InsertOperation a owl:Class ;
rdfs:subClassOf :GraphOperation ;
prov:definition "An activity that was performed on exactly one
rgprov:Graph, consisting of inserting triples found in exactly
one other rgprov:Graph, to produce a third rgprov:Graph." .

### http://www.ecs.soton.ac.uk/rgprov#DeleteOperation
:DeleteOperation a owl:Class ;
```

```
rdfs:subClassOf :GraphOperation ;
owl:disjointWith :InsertOperation ;
prov:definition "An activity that was performed on exactly one
rgprov:Graph, consisting of deleting triples found in exactly
one other rgprov:Graph, to produce a third rgprov:Graph." .

###  http://www.ecs.soton.ac.uk/rgprov#Entailment
:Entailment a owl:Class ;
rdfs:subClassOf :GraphOperation ;
prov:definition "An activity that was performed on exactly one
rgprov:Graph to produce another rgprov:Graph that contains
inferred triples." .

###  http://www.ecs.soton.ac.uk/rgprov#DEntailment
:DEntailment a owl:Class ;
rdfs:subClassOf :Entailment ;
prov:definition "An activity that was performed on exactly one
rgprov:Graph to produce another rgprov:Graph that contains
inferred triples, based on the Datatype Entailment Regime." .

###  http://www.ecs.soton.ac.uk/rgprov#OWLDirectEntailment
:OWLDirectEntailment a owl:Class ;
rdfs:subClassOf :Entailment ;
prov:definition "An activity that was performed on exactly one
rgprov:Graph to produce another rgprov:Graph that contains
inferred triples, based on the OWL Direct Entailment Regime." .

###  http://www.ecs.soton.ac.uk/rgprov#OWLRDFEntailment
:OWLRDFEntailment a owl:Class ;
rdfs:subClassOf :Entailment ;
prov:definition "An activity that was performed on exactly one
rgprov:Graph to produce another rgprov:Graph that contains
inferred triples, based on the OWL RDF-based Entailment Regime." .

###  http://www.ecs.soton.ac.uk/rgprov#RDFSEntailment
:RDFSEntailment a owl:Class ;
   rdfs:subClassOf :Entailment ;
   prov:definition "An activity that was performed on exactly one
 rgprov:Graph to produce another rgprov:Graph that contains
 inferred triples, based on the RDFS Entailment Regime." .
```

```
###  http://www.ecs.soton.ac.uk/rgprov#RIFEntailment
:RIFEntailment a owl:Class ;
  rdfs:subClassOf :Entailment ;
  prov:definition "An activity that was performed on exactly one
rgprov:Graph to produce another rgprov:Graph that contains
inferred triples, based on the RIF Core Entailment Regime." .

###  http://www.ecs.soton.ac.uk/rgprov#RDFEntailment
:RDFEntailment a owl:Class ;
rdfs:subClassOf :Entailment ;
prov:definition "An activity that was performed on exactly one
rgprov:Graph to produce another rgprov:Graph that contains
inferred triples, based on the RDF Entailment Regime." .

###  http://www.ecs.soton.ac.uk/rgprov#Reasoner
:Reasoner a owl:Class ;
rdfs:subClassOf prov:SoftwareAgent ;
prov:definition "The software that bears the responsibility of reasoning
on an rgprov:Graph." .

###  http://www.ecs.soton.ac.uk/rgprov#OWLRDFReasoner
:OWLRDFReasoner a owl:Class ;
rdfs:subClassOf :Reasoner ;
prov:definition "The software that bears the responsibility of reasoning
on an rgprov:Graph using the OWL RDF-Based Entailment Regime." .

###  http://www.ecs.soton.ac.uk/rgprov#DReasoner
:DReasoner a owl:Class ;
rdfs:subClassOf :Reasoner ;
prov:definition "The software that bears the responsibility of reasoning
on an rgprov:Graph using the Datatype Entailment Regime." .

###  http://www.ecs.soton.ac.uk/rgprov#OWLDirectReasoner
:OWLDirectReasoner a owl:Class ;
rdfs:subClassOf :Reasoner ;
prov:definition "The software that bears the responsibility of reasoning
on an rgprov:Graph using the OWL Direct Entailment Regime." .

###  http://www.ecs.soton.ac.uk/rgprov#RDFReasoner
:RDFReasoner a owl:Class ;
rdfs:subClassOf :Reasoner ;
```

```
prov:definition "The software that bears the responsibility of reasoning
on an rgprov:Graph using the RDF Entailment Regime." .

###  http://www.ecs.soton.ac.uk/rgprov#RDFSReasoner
:RDFSReasoner a owl:Class ;
 rdfs:subClassOf :Reasoner ;
 prov:definition "The software that bears the responsibility of reasoning
 on an rgprov:Graph using the RDFS Entailment Regime." .

###  http://www.ecs.soton.ac.uk/rgprov#RIFReasoner
:RIFReasoner a owl:Class ;
rdfs:subClassOf :Reasoner ;
prov:definition "The software that bears the responsibility of reasoning
on an rgprov:Graph using the RIF Core Entailment Regime." .

######################################################################
#    Object Properties
######################################################################

###  http://www.ecs.soton.ac.uk/rgprov#copied
:copied a owl:ObjectProperty ;
rdfs:subPropertyOf prov:used;
rdfs:domain :Fetch ;
rdfs:range :Graph ;
rdfs:comment "A copy of rgprov:Graph was fetched."@en .

###  http://www.ecs.soton.ac.uk/rgprov#wasExactCopy
:wasExactCopy a owl:ObjectProperty ;
rdfs:subPropertyOf prov:wasQuotedFrom ;
rdfs:domain :Graph ;
rdfs:range :Graph ;
rdfs:comment "A graph was an exact replica of another."@en .

###  http://www.ecs.soton.ac.uk/rgprov#wasCopyResult
:wasCopyResult a owl:ObjectProperty ;
rdfs:subPropertyOf prov:wasGeneratedBy ;
rdfs:domain :Graph ;
rdfs:range :Fetch ;
rdfs:comment "An rgprov:Graph was the result of copy (fetch)
activity."@en .
```

```
###  http://www.ecs.soton.ac.uk/rgprov#inserted
:inserted a owl:ObjectProperty ;
rdfs:subPropertyOf prov:used ;
rdfs:domain :InsertOperation ;
rdfs:range :UpdateGraph ;
rdfs:comment "The rgprov:UpdateGraph used by the rgprov:InsertOperation
to add triples into an rgprov:Graph."@en .

###  http://www.ecs.soton.ac.uk/rgprov#deleted
:deleted a owl:ObjectProperty ;
rdfs:subPropertyOf prov:used ;
rdfs:domain :DeleteOperation ;
rdfs:range :UpdateGraph ;
rdfs:comment "The rgprov:UpdateGraph used by the rgprov:InsertOperation
to delete triples into an rgprov:Graph."@en .

###  http://www.ecs.soton.ac.uk/rgprov#hadMinuend
:hadMinuend a owl:ObjectProperty ;
rdfs:subPropertyOf prov:used ;
rdfs:domain :Difference ;
rdfs:range :Graph ;
rdfs:comment "An rgprov:Graph was used as the first component of a graph
difference activity."@en .

###  http://www.ecs.soton.ac.uk/rgprov#hadSubtrahend
:hadSubtrahend a owl:ObjectProperty ;
rdfs:subPropertyOf prov:used ;
rdfs:domain :Difference ;
rdfs:range :Graph ;
rdfs:comment "An rgprov:Graph was used as the second component of a graph
difference activity."@en .

###  http://www.ecs.soton.ac.uk/rgprov#wasEntailedFrom
:wasEntailedFrom a owl:ObjectProperty ;
rdfs:subPropertyOf prov:wasDerivedFrom ;
rdfs:domain :Graph ;
rdfs:range :Graph ;
rdfs:comment "An entailment is the construction of a new rgprov:Graph
based on a pre-existing rgprov:Graph."@en .
```

# Appendix B

# Application of RGPROV - Extended

## B.1 Vocabulary for Initial Graph Creation

In this section we describe the usage of the proposed vocabulary RGPROV during the process of creating the initial graph $G_{ent(C)}$.

### B.1.1 Graph Retrieval

Graph $P_{copy(A)}$ contains, in addition to the information contained from $P_A$, the following information:

- :FETCH-A-YYYYMMDD, an instance of rgprov:Fetch, that represents a fetch call that was made to system $A$ at that time. This creates the triple `:FETCH-A-YYYYMMDD rdf:type rgprov:Fetch .`
- :jersey2.25 ran a GET method. This creates the triple `:FETCH-A-YYYYMMDD prov:wasAssociatedWith :jersey2.25 .`
- information describing the copying process. This creates the five triples:
  - `:FETCH-A-YYYYMMDD :retrievedFrom URI-of-(A)-for-`$\mathtt{G_A}$ `.`
  - `:FETCH-A-YYYYMMDD rgprov:copied :`$\mathtt{G_A}$ `.`
  - `:`$\mathtt{G_{copy(A)}}$ `rdf:type rgprov:Graph .`
  - `:`$\mathtt{G_{copy(A)}}$ `rgprov:wasCopyResult :FETCH-A-YYYYMMDD .`
  - `:`$\mathtt{G_{copy(A)}}$ `rgprov:wasExactCopy :`$\mathtt{G_A}$ `.`
- :FETCH-A-YYYYMMDD's start and end time. This creates the two triples:

  - `:FETCH-A-YYYYMMDD prov:startedAtTime "YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`
  - `:FETCH-A-YYYYMMDD prov:endedAtTime "YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`

Similarly, graph $P_{copy(B)}$ contains, in addition to the contained in $P_B$, the following information:

- :FETCH-B-YYYYMMDD, an instance of rgprov:Fetch, that represents a fetch call that was made to system $B$ at that time. This creates the triple
  `:FETCH-A-YYYYMMDD rdf:type rgprov:Fetch .`
- :jersey2.25 ran a GET method. This creates the triple
  `:FETCH-B-YYYYMMDD prov:wasAssociatedWith :jersey2.25 .`
- information describing the copying process. This creates the five triples:
  - `:FETCH-B-YYYYMMDD :retrievedFrom URI-of-(B)-for-`$\mathtt{G_B}$ .
  - `:FETCH-B-YYYYMMDD rgprov:copied :`$\mathtt{G_B}$ .
  - :$\mathtt{G_{copy(B)}}$ `rdf:type rgprov:Graph .`
  - :$\mathtt{G_{copy(B)}}$ `rgprov:wasCopyResult :FETCH-B-YYYYMMDD .`
  - :$\mathtt{G_{copy(B)}}$ `rgprov:wasExactCopy :`$\mathtt{G_B}$ .
- :FETCH-B-YYYYMMDD's start and end time. This creates the two triples:
  - `:FETCH-B-YYYYMMDD prov:startedAtTime "YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`
  - `:FETCH-B-YYYYMMDD prov:endedAtTime "YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`

The provenance graph $P_{ent(C)}$ contains the information listed in the preceding two lists. It is shown in Figures B.1. Additionally, to show that $P_{ent(C)}$ contains the information in the other provenance graphs, we add the two triples

- :$\mathtt{P_{ent(C)}}$ `prov:wasDerivedFrom :`$\mathtt{P_{copy(A)}}$ .
- :$\mathtt{P_{ent(C)}}$ `prov:wasDerivedFrom :`$\mathtt{P_{copy(B)}}$ .

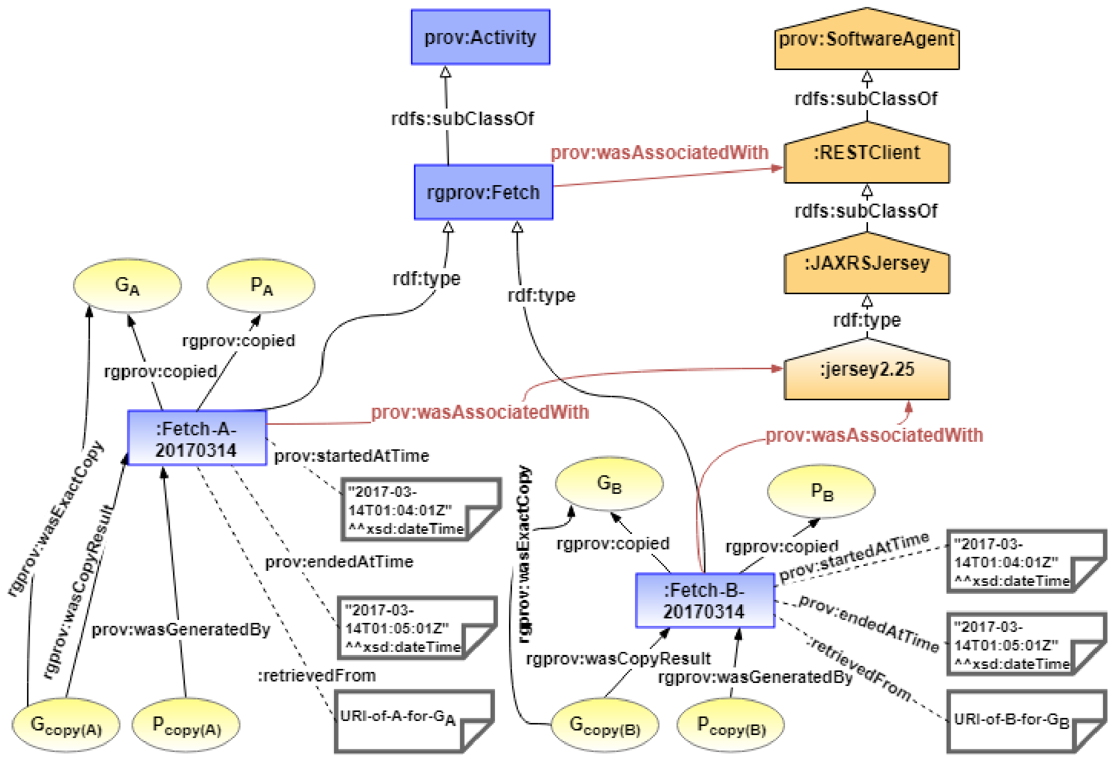


Figure B.1: First Iteration of $P_{ent(C)}$.

### B.1.2 Graph Operations

#### B.1.2.1 Set Theoretic Operations

The production of $G_{ent(C)}$ can be the result of any one of the following set theoretic operations:

**Union,** $G_{ent(C)} = G_{copy(A) \cup copy(B)}$. After Jena creates $G_{ent(C)}$, $C$ adds the following information to $P_{ent(C)}$:

- :gu-A-B-YYYYMMDD, an instance of rgprov:Union, that represents a union operation on $G_{copy(A)}$ and $G_{copy(B)}$. This creates the triple `:gu-A-B-YYYYMMDD rdf:type rgprov:Union .`
- :jena3.1.1 ran the union operation :gu-A-B-YYYYMMDD. This creates the triple `:gu-A-B-YYYYMMDD prov:wasAssociatedWith :jena3.1.1 .`
- :gu-A-B-YYYYMMDD used both $G_{copy(A)}$ and $G_{copy(B)}$ in its union operation. This creates the two triples
  - `:gu-A-B-YYYYMMDD prov:used G`$_{copy(A)}$ `.`
  - `:gu-A-B-YYYYMMDD prov:used G`$_{copy(B)}$ `.`

- $G_C$ is a graph which was produced by :gu-A-B-YYYYMMDD. This creates the two triples:
  - `G`$_{\texttt{C}}$ `rdf:type rgprov:Graph .`
  - `G`$_{\texttt{C}}$ `prov:wasGeneratedBy :gu-A-B-YYYYMMDD .`
- :gu-A-B-YYYYMMDD's start and end time. This creates the two triples:
  - `:gu-A-B-YYYYMMDD prov:startedAtTime "YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`
  - `:gu-A-B-YYYYMMDD prov:endedAtTime "YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`

**Intersection,** $G_C = G_{copy(A) \cap copy(B)}$. After Jena creates $G_C$, $C$ adds the following information to $P_{ent(C)}$:

- :gi-A-B-YYYYMMDD, an instance of rgprov:Intersection, that represents an intersection operation performed on $G_{copy(A)}$ and $G_{copy(B)}$. This creates the triple `:gi-A-B-YYYYMMDD rdf:type rgprov:Intersection .`
- :jena3.1.1 ran the intersection operation :gi-A-B-YYYYMMDD. This creates the triple `:gi-A-B-YYYYMMDD prov:wasAssociatedWith :jena3.1.1 .`
- :gi-A-B-YYYYMMDD used both $G_{copy(A)}$ and $G_{copy(B)}$ in its intersection operation. This creates the two triples
  - `:gi-A-B-YYYYMMDD prov:used G`$_{\texttt{copy(A)}}$ `.`
  - `:gi-A-B-YYYYMMDD prov:used G`$_{\texttt{copy(B)}}$ `.`
- $G_C$ was produced by :gi-A-B-YYYYMMDD. This creates the two triples:
  - `G`$_{\texttt{C}}$ `rdf:type rgprov:Graph .`
  - `G`$_{\texttt{C}}$ `prov:wasGeneratedBy :gi-A-B-YYYYMMDD .`
- :gi-A-B-YYYYMMDD's start and end time. This creates the two triples:
  - `:gi-A-B-YYYYMMDD prov:startedAtTime "YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`
  - `:gi-A-B-YYYYMMDD prov:endedAtTime "YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`

**Difference Case 1,** $G_C = G_{copy(A) \setminus copy(B)}$. After Jena creates $G_C$, $C$ adds the following information to $P_{ent(C)}$:

- :gd-A-B-YYYYMMDD, an instance of rgprov:Difference, that represents a difference operation performed on $G_{copy(A)}$ and $G_{copy(B)}$. This creates the triple `:gd-A-B-YYYYMMDD rdf:type rgprov:Difference .`
- :jena3.1.1 ran the difference operation :gd-A-B-YYYYMMDD. This creates the triple `:gd-A-B-YYYYMMDD prov:wasAssociatedWith :jena3.1.1 .`
- :gd-A-B-YYYYMMDD used $G_{copy(A)}$ as the first component in its difference operation. This creates the triple
  `:gd-A-B-YYYYMMDD rgprov:hadMinuend` $\mathtt{G_{copy(A)}}$ `.`
- :gd-A-B-YYYYMMDD used $G_{copy(B)}$ as the second component in its difference operation. This creates the triple
  `:gd-A-B-YYYYMMDD rgprov:hadSubtrahend` $\mathtt{G_{copy(B)}}$ `.`
- $G_C$ was produced by :gd-A-B-YYYYMMDD. This creates the two triples:
  - $\mathtt{G_C}$ `rdf:type rgprov:Graph .`
  - $\mathtt{G_C}$ `prov:wasGeneratedBy :gd-A-B-YYYYMMDD .`
- :gd-A-B-YYYYMMDD's start and end time. This creates the two triples:
  - `:gd-A-B-YYYYMMDD prov:startedAtTime`
    `"YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`
  - `:gd-A-B-YYYYMMDD prov:endedAtTime`
    `"YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`

**Difference: Case 2,** $G_C = G_{copy(B) \setminus copy(A)}$, After Jena creates $G_C$, $C$ adds the following information to $P_{ent(C)}$:

- :gd-A-B-YYYYMMDD, an instance of rgprov:Difference, that represents a difference operation performed on $G_{copy(A)}$ and $G_{copy(B)}$. This creates the triple `:gd-A-B-YYYYMMDD rdf:type rgprov:Difference .`
- :jena3.1.1 ran the difference operation :gd-A-B-YYYYMMDD. This creates the triple `:gd-A-B-YYYYMMDD prov:wasAssociatedWith :jena3.1.1 .`
- :gd-A-B-YYYYMMDD used $G_{copy(B)}$ as the first component in its difference operation. This creates the triple
  `:gd-A-B-YYYYMMDD rgprov:hadMinuend` $\mathtt{G_{copy(B)}}$ `.`

- :gd-A-B-YYYYMMDD used $G_{copy(A)}$ as the second component in its difference operation. This creates the triple
  `:gd-A-B-YYYYMMDD rgprov:hadSubtrahend` $\mathtt{G_{copy(A)}}$ `.`
- $G_C$ was produced by :gd-A-B-YYYYMMDD. This creates the two triples:
  - $\mathtt{G_C}$ `rdf:type rgprov:Graph .`
  - $\mathtt{G_C}$ `prov:wasGeneratedBy :gd-A-B-YYYYMMDD .`
- :gd-A-B-YYYYMMDD's start and end time. This creates the two triples:
  - `:gd-A-B-YYYYMMDD prov:startedAtTime`
    `"YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`
  - `:gd-A-B-YYYYMMDD prov:endedAtTime`
    `"YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`

Two triples are shared among all the above set theoretic operations and they indicate that:

- $G_{copy(A)}$ was a contributor in the creation of $G_C$. This creates the triple
  $\mathtt{G_C}$ `prov:wasDerivedFrom` $\mathtt{G_{copy(A)}}$ `.`
- $G_{copy(B)}$ was a contributor in the creation of $G_C$. This creates the triple
  $\mathtt{G_C}$ `prov:wasDerivedFrom` $\mathtt{G_{copy(B)}}$ `.`

The second iteration of $P_{ent(C)}$ is shown in Figure B.2 with the union operation used as an example of set theoretic operations.

#### B.1.2.2 Entailment

After $C$ invokes Jena to entail $G_{ent(C)}$ from $G_C$, the following information is added to $P_{ent(C)}$:

- :ge-C3-YYYYMMDD, an instance of rgprov:RDFSEntailment, that represents an entailment operation performed on $G_C$. This creates the triple
  `:ge-C3-YYYYMMDD rdf:type rgprov:RDFSEntailment .`
- :jena3.1.1 ran the entailment operation :ge-C3-YYYYMMDD. This creates the triple `:ge-C3-YYYYMMDD prov:wasAssociatedWith :jena3.1.1 .`
- :ge-C3-YYYYMMDD used $G_C$ to produce $G_{ent(C)}$. This creates the triple
  `:ge-C3-YYYYMMDD prov:used` $\mathtt{G_C}$ `.`

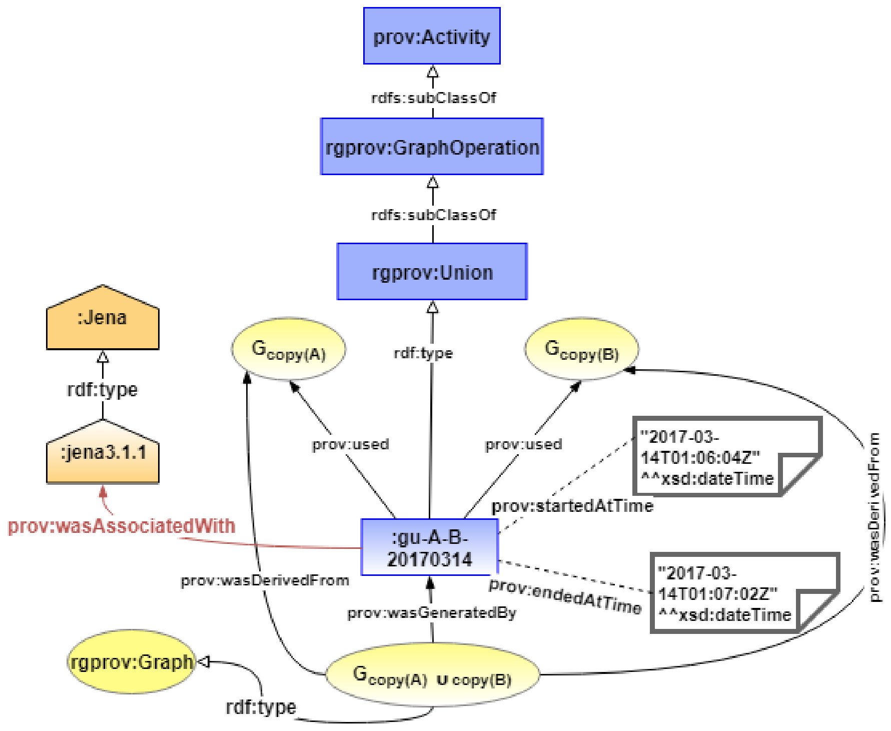


Figure B.2: Second Iteration of $P_{ent(C)}$.

- $G_{ent(C)}$ was produced by :ge-C3-YYYYMMDD. This creates the two triples
  - `G`$_{\texttt{ent(C)}}$ `rdf:type rgprov:Graph .`
  - `G`$_{\texttt{ent(C)}}$ `prov:wasGeneratedBy :ge-C3-YYYYMMDD .`
- $G_C$ was a contributor in the creation of $G_{ent(C)}$, more specifically $G_C$ entailed $G_{ent(C)}$. This creates the triple `G`$_{\texttt{ent(C)}}$ `rgprov:wasEntailedFrom G`$_{\texttt{C}}$ `.`
- Since $G_{ent(C)}$ was itself derived from $G_{copy(A)}$ and $G_{copy(B)}$, the two additional triples are also created:
  - `G`$_{\texttt{ent(C)}}$ `prov:wasDerivedFrom G`$_{\texttt{copy(A)}}$ `.`
  - `G`$_{\texttt{ent(C)}}$ `prov:wasDerivedFrom G`$_{\texttt{copy(B)}}$ `.`
- :ge-C3-YYYYMMDD's start and end time. This creates the two triples:
  - `:ge-C3-YYYYMMDD prov:startedAtTime "YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`
  - `:ge-C3-YYYYMMDD prov:endedAtTime "YYYY-MM-DDYThh:mm:ssZ"^^xsd:dateTime .`

The above list constitutes the final additions to $P_{ent(C)}$ and is shown in Figure B.3.

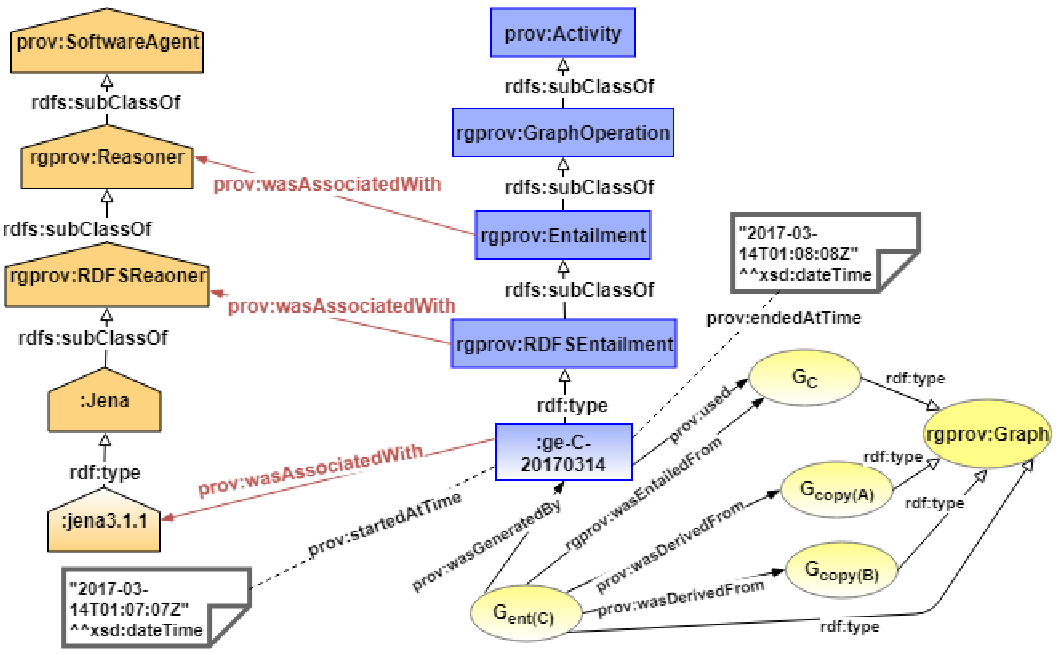


Figure B.3: Final Iteration of $P_{ent(C)}$.

## B.2 Vocabulary for Update Propagation

### B.2.1 Update Retrieval

$C$ creates the provenance graph $P^{up}_{copy(B)}$ similar to how it created $P_{copy(B)}$, where the information is copied from $P^{up}_{B}$ into $P^{up}_{copy(B)}$ and the following is added to it:

- :Fetch-BUp-YYYYMMDD, an instance of rgprov:Fetch, that indicates that a fetch call that made to system $B$ at that time. This creates the triple
  `:FETCH-A-YYYYMMDD rdf:type rgprov:Fetch .`
- :jersey2.25 ran a GET method. This creates the triple
  `:Fetch-BUp-YYYYMMDD prov:wasAssociatedWith :jersey2.25 .`
- information describing the copying process. This creates the five triples
  - :Fetch-BUp-YYYYMMDD :retrievedFrom URI-of-(B)-for-$\mathtt{G}^{\mathtt{up}}_{\mathtt{B}}$ .
  - :Fetch-BUp-YYYYMMDD rgprov:copied $\mathtt{G}^{\mathtt{up}}_{\mathtt{B}}$ .
  - $\mathtt{G}^{\mathtt{up}}_{\mathtt{copy(B)}}$ rgprov:wasCopyResult :Fetch-BUp-YYYYMMDD .
  - $\mathtt{G}^{\mathtt{up}}_{\mathtt{copy(B)}}$ rdf:type rgprov:UpdateGraph .
  - $\mathtt{G}^{\mathtt{up}}_{\mathtt{copy(B)}}$ rgprov:wasExactCopy $\mathtt{G}^{\mathtt{up}}_{\mathtt{B}}$ .
- :Fetch-BUp-YYYYMMDD's start and end time. This creates the two triples:

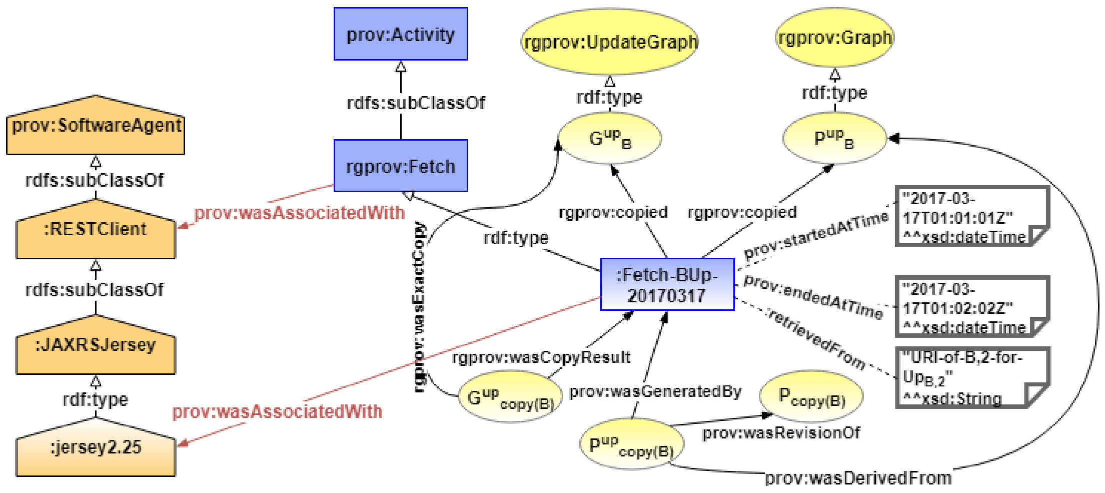


Figure B.4: First Iteration of $P_{ent(C')}$.

- :Fetch-BUp-YYYYMMDD prov:startedAtTime "YYYY-MM-DDYThh:mm:ssZ"∧∧xsd:dateTime .
- :Fetch-BUp-YYYYMMDD prov:endedAtTime "YYYY-MM-DDYThh:mm:ssZ"∧∧xsd:dateTime .

- If the graphs are being copied, then $G_{copy(B')}$ is a new version of $G_{copy(B)}$. This creates the triple $\mathtt{G_{copy(B')}}$ prov:wasRevisionOf $\mathtt{G_{copy(B)}}$ .
- That the provenance graph $P^{up}_{copy(B)}$ was derived from $P^{up}_B$ and is a new version of $P_{copy(B)}$. $P^{up}_{copy(B)}$ is a new version of $P_{copy(B)}$, this creates the two triples:
  - $\mathtt{P^{up}_{copy(B)}}$ prov:wasDerivedFrom $\mathtt{P^{up}_B}$ .
  - $\mathtt{P^{up}_{copy(B)}}$ prov:wasRevisionOf $\mathtt{P_{copy(B)}}$ .

The above list is shown in Figure B.4.

$C$ then creates the provenance graph $P_{ent(C')}$ by including the above information as well as the information stating that $P_{ent(C')}$ is a new version of $P_{ent(C)}$, this creates the triple: $\mathtt{P_{ent(C')}}$ prov:wasRevisionOf $\mathtt{P_{ent(C)}}$ . Stating that $G_{ent(C')}$ is a new version of $G_{ent(C)}$ will be added after $G_{ent(C')}$ has been created.

### B.2.2 Effects of Updates on Set Theoretic Operations

#### B.2.2.1 Union

**Insert** $C$ invokes Jena to insert the triples in $G^{up}_{copy(B)}$ into $G_C$, resulting in graph $G_{C'}$. The following information is then added to $P_{ent(C')}$:

- :insert-C-YYYYMMDD, an instance of rgprov:InsertOperation, that represents an insert operation performed on $G_C$. This creates the triple
  `:insert-C-YYYYMMDD rdf:type rgprov:InsertOperation .`

- :jena3.1.1 ran the insert operation :insert-C-YYYYMMDD. This creates the triple
  `:insert-C-YYYYMMDD prov:wasAssociatedWith :jena3.1.1 .`

- :insert-C-YYYYMMDD used $G^{up}_{copy(B)}$ in its insert operation. This creates the triple `:insert-C-YYYYMMDD rgprov:inserted` $\texttt{G}^{\texttt{up}}_{\texttt{copy(B)}}$ `.`

- :insert-C-YYYYMMDD used $G_C$ in its insert operation. This creates the triple
  `:insert-C-YYYYMMDD prov:used` $\texttt{G}_{\texttt{C}}$ `.`

- $G_{C'}$ was produced by :insert-C-YYYYMMDD. This creates the triple
  $\texttt{G}_{\texttt{C'}}$ `prov:wasGeneratedBy :insert-C-YYYYMMDD .`

- $G^{up}_{copy(B)}$ was a contributor in the creation of $G_{C'}$. This creates the triple
  $\texttt{G}_{\texttt{C'}}$ `prov:wasDerivedFrom` $\texttt{G}^{\texttt{up}}_{\texttt{copy(B)}}$ `.`

- $G_C$ was a contributor in the creation of $G_{C'}$. This creates the triple
  $\texttt{G}_{\texttt{C'}}$ `prov:wasDerivedFrom` $\texttt{G}_{\texttt{ent(C)}}$`.`

- :insert-C-YYYYMMDD's start and end time. This creates the two triples:

  – `:insert-C-YYYYMMDD prov:startedAtTime "YYYY-MM-DDThh:mm:ssZ"^^xsd:dateTime .`

  – `:insert-C-YYYYMMDD prov:endedAtTime "YYYY-MM-DDThh:mm:ssZ"^^xsd:dateTime .`

The above list is shown in Figure B.5.

**Delete** $C$ invokes Jena to produce $G^{up}_{sub(copy(B))} \equiv G^{up}_{copy(B)} \setminus G_{copy(A)}$ and then to delete that from $G_C$ resulting in $G_{C'}$. The following information is then added to $P_{ent(C')}$:

- :delete-C-YYYYMMDD, an instance of rgprov:DeleteOperation, that represents a delete operation performed on $G_C$. This creates the triple
  `:delete-C-YYYYMMDD rdf:type rgprov:DeleteOperation .`

- :jena3.1.1 ran the delete operation :delete-C-YYYYMMDD. This creates the triple
  `:delete-C-YYYYMMDD prov:wasAssociatedWith :jena3.1.1 .`

- $G^{up}_{sub(copy(B))}$, an instance of rgprov:UpdateGraph, that represents the subgraph of the $G^{up}_{copy(B)}$ to be deleted. This creates the triple
  $\texttt{G}^{\texttt{up}}_{\texttt{sub(copy(B))}}$ `rdf:type rgprov:UpdateGraph .`

- $G^{up}_{sub(copy(B))}$ was derived from $G^{up}_{copy(B)}$. This creates the triple
`G`$^{\texttt{up}}_{\texttt{sub(copy(B))}}$ `prov:wasDerivedFrom G`$^{\texttt{up}}_{\texttt{copy(B)}}$ `.`
- $G_{copy(A)}$ was a contributor in the creation of $G^{up}_{sub(copy(B))}$. This creates the triple
`G`$^{\texttt{up}}_{\texttt{sub(copy(B))}}$ `prov:wasDerivedFrom :G`$_{\texttt{copy(A)}}$ `.`
- :delete-C-YYYYMMDD used $G^{up}_{sub(copy(B))}$ in its delete operation. This creates the triple `:delete-C-YYYYMMDD rgprov:deleted G`$^{\texttt{up}}_{\texttt{sub(copy(B))}}$ `.`
- :delete-C-YYYYMMDD used $G_C$ in its delete operation. This creates the triple
`:delete-C-YYYYMMDD prov:used G`$_{\texttt{C}}$ `.`
- $G_{C'}$ was produced by :delete-C-YYYYMMDD. This creates the triple
`G`$_{\texttt{C'}}$ `prov:wasGeneratedBy :delete-C-YYYYMMDD .`
- $G_{copy(A)}$ was a contributor in the creation of $G_{C'}$. This creates the triple
`G`$_{\texttt{C'}}$ `prov:wasDerivedFrom :G`$_{\texttt{copy(A)}}$ `.`
- $G^{up}_{copy(B)}$ was a contributor in the creation of $G_{C'}$. This creates the triple
`G`$_{\texttt{C'}}$ `prov:wasDerivedFrom G`$^{\texttt{up}}_{\texttt{copy(B)}}$ `.`
- $G^{up}_{sub(copy(B))}$ was a contributor in the creation of $G_{C'}$. This creates the triple
`G`$_{\texttt{C'}}$ `prov:wasDerivedFrom G`$^{\texttt{up}}_{\texttt{sub(copy(B))}}$ `.`
- $G_{C'}$ was a new version of $G_C$. This creates the triple
`G`$_{\texttt{C}}$ `prov:wasRevisionOf G`$_{\texttt{C'}}$ `.`
- :delete-C-YYYYMMDD's start and end time. This creates the two triples:
  - `:delete-C-YYYYMMDD prov:startedAtTime`
    `"YYYY-MM-DDYThh:mm:ssZ"∧∧xsd:dateTime .`
  - `:delete-C-YYYYMMDD prov:endedAtTime`
    `"YYYY-MM-DDYThh:mm:ssZ"∧∧xsd:dateTime .`

The above list is shown in Figure B.6.

#### B.2.2.2 Intersection

**Insert** $C$ invokes Jena to produce $G^{up}_{sub(copy(B))} \equiv G_{copy(A)} \cap G^{up}_{copy(B)}$ and then to insert that into graph $G_C$ resulting in $G_{C'}$. The following information is then added to $P_{ent(C')}$:

- :insert-C-YYYYMMDD, an instance of rgprov:InsertOperation, that represents an insert operation performed on $G_C$. This creates the triple
`:insert-C-YYYYMMDD rdf:type rgprov:InsertOperation .`

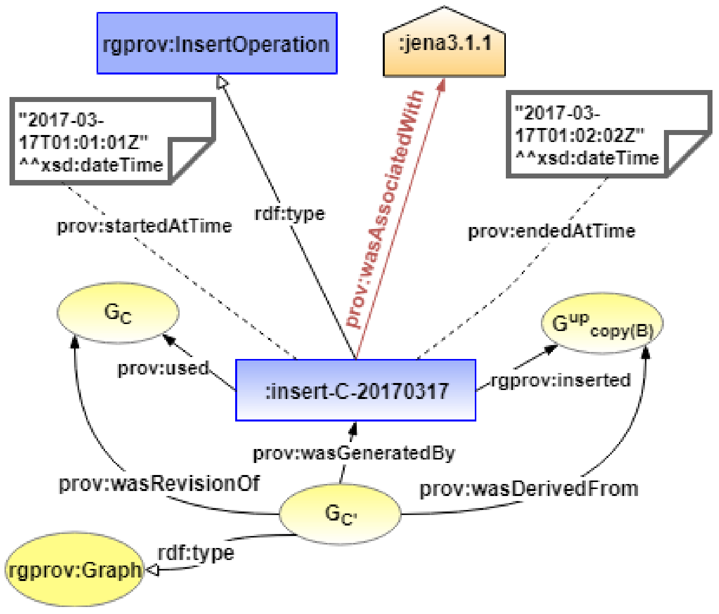


Figure B.5: Provenance of Insert Propagation on Graph from Union.

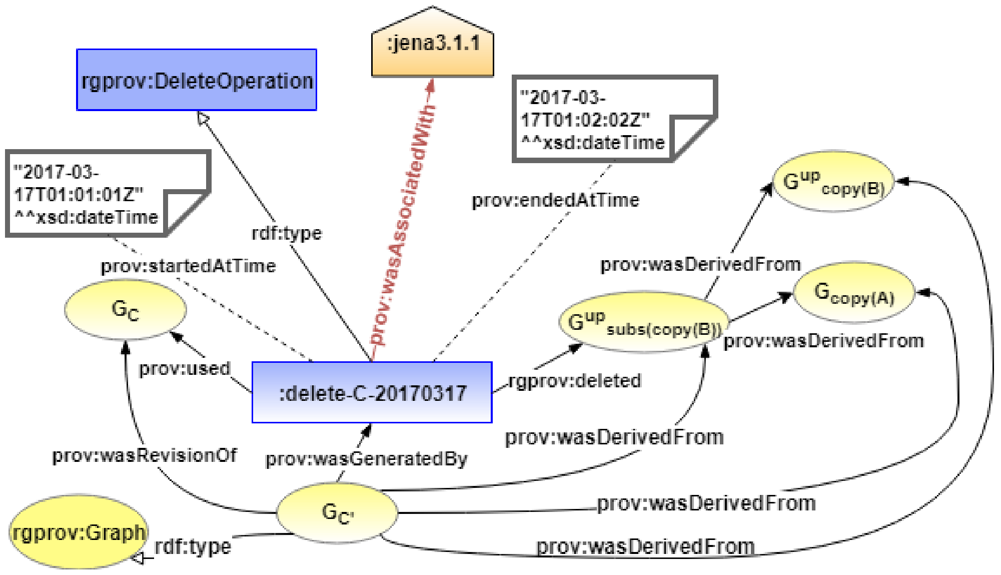


Figure B.6: Provenance of Delete Propagation on Graph from Union.

- :jena3.1.1 ran the insert operation :insert-C-YYYYMMDD. This creates the triple
  `:insert-C-YYYYMMDD prov:wasAssociatedWith :jena3.1.1 .`

- $G^{up}_{sub(copy(B))}$, an instance of rgprov:UpdateGraph, that represents the subgraph of $G^{up}_{copy(B)}$ to be inserted. This creates the triple
  $\texttt{G}^{\texttt{up}}_{\texttt{sub(copy(B))}}$ `rdf:type rgprov:UpdateGraph .`

- $G_{copy(A)}$ was a contributor in the creation of $G^{up}_{sub(copy(B))}$. This creates the triple
  $\texttt{G}^{\texttt{up}}_{\texttt{sub(copy(B))}}$ `prov:wasDerivedFrom` $\texttt{:G}_{\texttt{copy(A)}}$ `.`

- $G^{up}_{sub(copy(B))}$ was derived from $G^{up}_{copy(B)}$. This creates the triple
  $\texttt{G}^{\texttt{up}}_{\texttt{sub(copy(B))}}$ `prov:wasDerivedFrom` $\texttt{G}^{\texttt{up}}_{\texttt{copy(B)}}$ `.`

- :insert-C-YYYYMMDD used $G^{up}_{sub(copy(B))}$ in its insert operation. This creates the triple `:insert-C-YYYYMMDD rgprov:inserted` $\mathtt{G^{up}_{sub(copy(B))}}$ `.`
- :insert-C-YYYYMMDD used $G_C$ in its insert operation. This creates the triple `:insert-C-YYYYMMDD prov:used` $\mathtt{G_C}$`.`
- $G_{C'}$ was produced by :insert-C-YYYYMMDD. This creates the triple $\mathtt{G_{C'}}$ `prov:wasGeneratedBy :insert-C-YYYYMMDD .`
- $G^{up}_{copy(B)}$ was a contributor in the creation of $G_{C'}$. This creates the triple $\mathtt{G_{C'}}$ `prov:wasDerivedFrom` $\mathtt{G^{up}_{copy(B)}}$ `.`
- $G^{up}_{sub(copy(B))}$ was a contributor in the creation of $G_{C'}$. This creates the triple $\mathtt{G_{C'}}$ `prov:wasDerivedFrom` $\mathtt{G^{up}_{sub(copy(B))}}$ `.`
- $G_{copy(A)}$ was a contributor in the creation of $G_{C'}$. This creates the triple $\mathtt{G_{C'}}$ `prov:wasDerivedFrom :`$\mathtt{G_{copy(A)}}$ `.`
- $G_{C'}$ was a new version of $G_C$. This creates the triple $\mathtt{G_C}$ `prov:wasRevisionOf` $\mathtt{G_{C'}}$ `.`
- :insert-C-YYYYMMDD's start and end time. This creates the two triples:
  - `:insert-C-YYYYMMDD prov:startedAtTime "YYYY-MM-DDThh:mm:ssZ"^^xsd:dateTime .`
  - `:insert-C-YYYYMMDD prov:endedAtTime "YYYY-MM-DDThh:mm:ssZ"^^xsd:dateTime .`

The above list is shown in Figure B.7.

**Delete** $C$ invokes Jena to delete from $G_C$ the triples in $G^{up}_{copy(B)}$ resulting in graph $G_{C'}$. The following information is then added to $P_{ent(C')}$:

- :delete-C-YYYYMMDD, an instance of rgprov:InsertOperation, that represents a delete operation performed on $G_C$. This creates the triple `:delete-C-YYYYMMDD rdf:type rgprov:DeleteOperation .`
- :jena3.1.1 ran the delete operation :delete-C-YYYYMMDD. This creates the triple `:delete-C-YYYYMMDD prov:wasAssociatedWith :jena3.1.1 .`
- :delete-C-YYYYMMDD used $G^{up}_{copy(B)}$ in its delete operation. This creates the triple `:delete-C-YYYYMMDD rgprov:deleted` $\mathtt{G^{up}_{copy(B)}}$ `.`
- :delete-C-YYYYMMDD used $G_C$ in its delete operation. This creates the triple `:delete-C-YYYYMMDD prov:used` $\mathtt{G_C}$ `.`

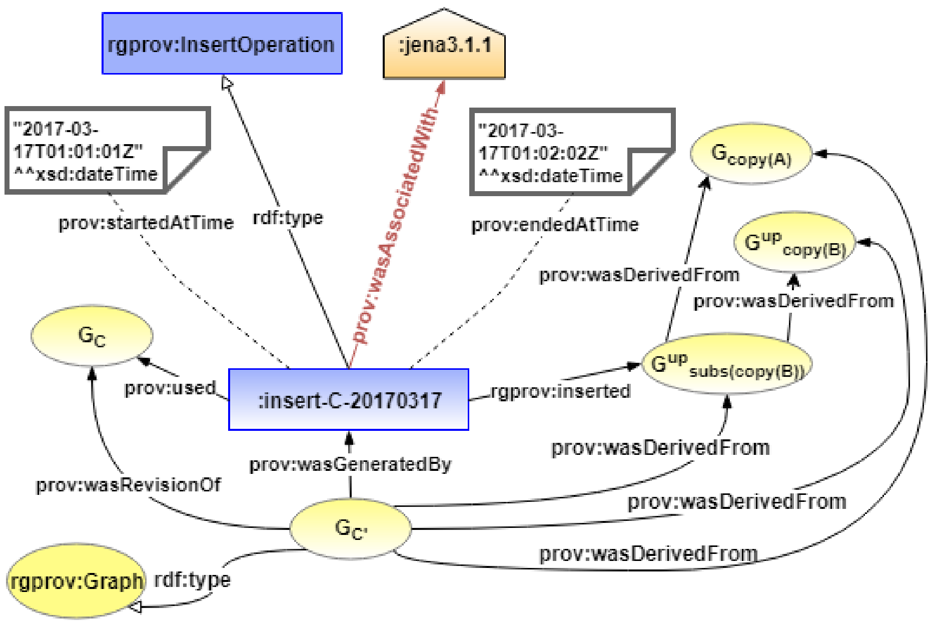


Figure B.7: Provenance of Insert Propagation on Graph from Intersection and Difference Case 2 and of Delete Propagation on Graph from Difference Case 1.

- $G_{C'}$ was produced by :delete-C-YYYYMMDD. This creates the triple
  `G`$_{\mathtt{C'}}$ `prov:wasGeneratedBy :delete-C-YYYYMMDD .`
- $G^{up}_{copy(B)}$ was a contributor in the creation of $G_{C'}$. This creates the triple
  `G`$_{\mathtt{C'}}$ `prov:wasDerivedFrom G`$^{\mathtt{up}}_{\mathtt{copy(B)}}$ `.`
- $G_{C'}$ was a new version of $G_C$. This creates the triple
  `G`$_{\mathtt{C}}$ `prov:wasRevisionOf G`$_{\mathtt{C'}}$ `.`
- :delete-C-YYYYMMDD's start and end time. This creates the two triples:
  - `:delete-C-YYYYMMDD prov:startedAtTime`
    `"YYYY-MM-DDThh:mm:ssZ"^^xsd:dateTime .`
  - `:delete-C-YYYYMMDD prov:endedAtTime`
    `"YYYY-MM-DDThh:mm:ssZ"^^xsd:dateTime .`

The above list is shown in Figure B.8.

### B.2.2.3 Difference Case 1

This case studies $G_{copy(A)} \setminus G_{copy(B)}$.

**Insert** This operation results in deleting from $G_C$ the triples in $G^{up}_{copy(B)}$. As its results are equivalent to deleting from a graph produced from intersection, we refer the reader to Section B.2.2.2.

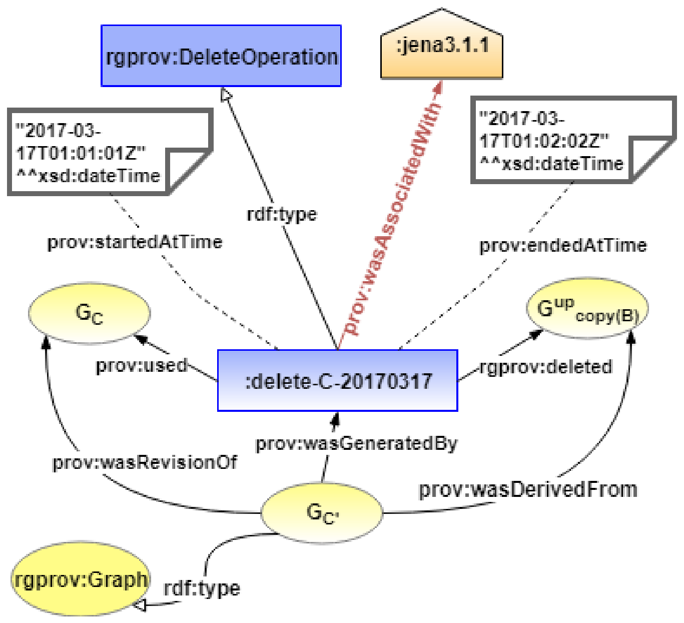


Figure B.8: Provenance of Delete Propagation on Graph from Intersection and Difference Case 2 and of Insert Propagation on Graph from Difference Case 1.

**Delete** This operation results in inserting into $G_C$ the triples in $G^{up}_{copy(B)} \cap G_{copy(A)}$. As its results are equivalent to inserting into a graph produced from intersection, we refer the reader to Section B.2.2.2.

#### B.2.2.4 Difference Case 2

This case studies $G_{copy(B)} \setminus G_{copy(A)}$.

**Insert** This operation results in inserting into $G_C$ the triples in $G^{up}_{copy(B)} \setminus G_{copy(A)}$. As its results are equivalent to inserting into a graph produced from intersection, with the only difference being how $G^{up}_{sub(copy(B))}$ is produced, we refer the reader to Section B.2.2.2.

**Delete** Deleting triples from $G_{copy(B)}$ results in deleting from $G_C$ the triples in $G^{up}_{copy(B)}$ As its results are equivalent to deleting from a graph produced from intersection, we refer the reader to Section B.2.2.2.

### B.2.3 Re-Entailment

After $C$ invokes Jena to entail $G_{ent(C')}$, the following information is added to $P_{ent(C')}$:

- :ge-C3-YYYYMMDD, an instance of rgprov:RDFSEntailment, that represents an entailment operation performed on - a portion of - $G_{C'}$. This creates the triple `:ge-C3-YYYYMMDD rdf:type rgprov:RDFSEntailment .`

- :jena3.1.1 ran the entailment operation :ge-C3-YYYYMMDD. This creates the triple `:ge-C3-YYYYMMDD prov:wasAssociatedWith :jena3.1.1 .`

- :ge-C3-YYYYMMDD used $G_{C'}$ to produce $G_{ent(C')}$. This creates the triple `:ge-C3-YYYYMMDD prov:used` $\mathtt{G_{C'}}$ `.`

- $G_{ent(C')}$ was produced by :ge-C3-YYYYMMDD. This creates the triple $\mathtt{G_{ent(C')}}$ `prov:wasGeneratedBy :ge-C3-YYYYMMDD .`

- $G_{C'}$ was a contributor in the creation of $G_{ent(C')}$, more specifically $G_{C'}$ entailed $G_{ent(C')}$. This creates the triple $\mathtt{G_{ent(C')}}$ `rgprov:wasEntailedFrom` $\mathtt{G_{C'}}$ `.`

- $G_{ent(C')}$ was derived from $G^{up}_{copy(B)}$, and if $G^{up}_{sub(copy(B))}$ was used then also that $G_{ent(C')}$ was derived from $G^{up}_{sub(copy(B))}$. This creates one or both of the following two triples:

  - $\mathtt{G_{ent(C')}}$ `prov:wasDerivedFrom` $\mathtt{G^{up}_{copy(B)}}$ `.`

  - $\mathtt{G_{ent(C')}}$ `prov:wasDerivedFrom` $\mathtt{G^{up}_{sub(copy(B))}}$ `.`

- $G_{ent(C')}$ is a new version of $G_{ent(C)}$. This creates the triple $\mathtt{G_{ent(C)}}$ `prov:wasRevisionOf` $\mathtt{G_{ent(C')}}$ `.`

- :ge-C3-YYYYMMDD's start and end time. This creates the two triples:

  - `:ge-C3-YYYYMMDD prov:startedAtTime "YYYY-MM-DDThh:mm:ssZ"^^xsd:dateTime .`

  - `:ge-C3-YYYYMMDD prov:endedAtTime "YYYY-MM-DDThh:mm:ssZ"^^xsd:dateTime .`

The above list constitutes the final additions to $P_{ent(C')}$ and is shown in Figures B.9 and B.10.

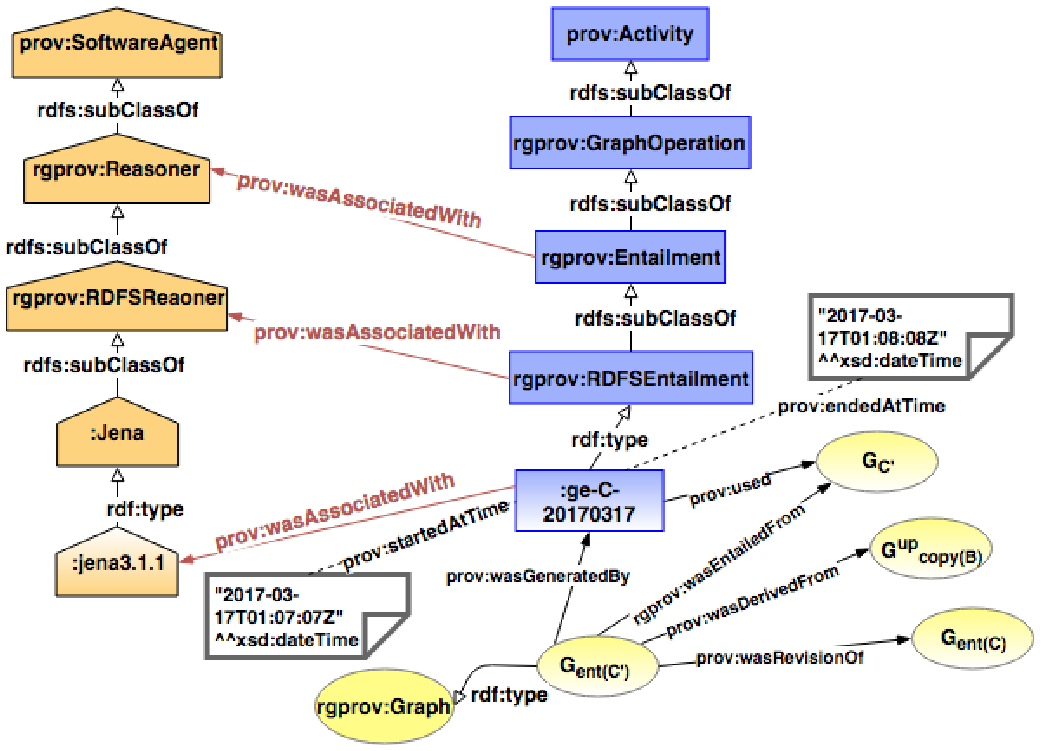


Figure B.9: Final Iteration of $P_{ent(C')}$ Using all the Update Graph.

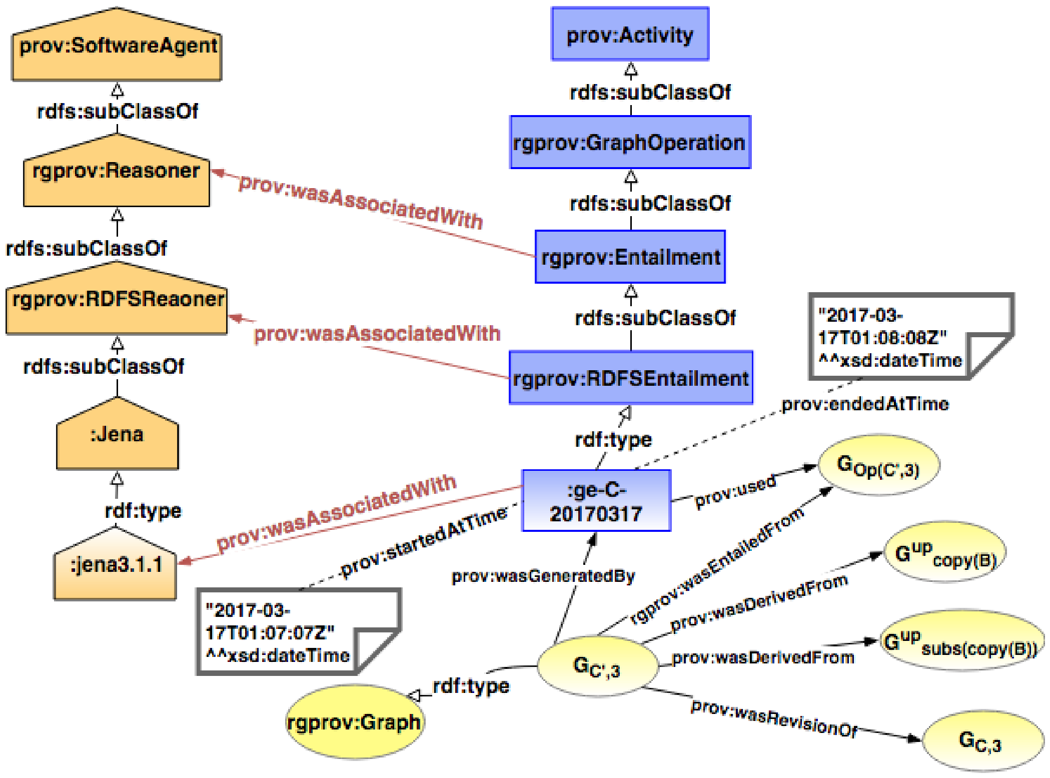


Figure B.10: Final Iteration of $P_{ent(C')}$ Using a Subgraph of the Update Graph.

# Appendix C

# Extending the Evaluation Framework

## C.1 Suggested Modifications

In this section, we describe the proposed changes that can be applied to instances and relations in our evaluation framework. These are listed below per instance type.

1. University Instances: a new university may be created and an existing one may be deleted. This would require the user to input, instead of the number of universities to be generated, a minimum number and a maximum number of universities to be generated. A university may also undergo a name change.
2. Department Instances: In addition to creating new departments and deleting existing ones, the following changes may be applied:
    (a) Name change.
    (b) Merge two departments together.
    (c) Split an existing department into two departments.
3. People instances: the suggested modifications apply to all types of faculty and student instances.
    (a) Name change.
    (b) Telephone change.
    (c) Email change.
    (d) Move to a different department in the same university.

   (e) Move to a different department in a different university.

4. Faculty instances: the suggested modifications apply to all types of faculty instances.

   (a) Give one or more additional courses.

   (b) Stop giving one or more courses.

5. Full Professor: In addition to creating new full professors and deleting existing ones, the following changes may be applied:

   (a) If they are chair of a department, then they may be removed as chair of said department.

   (b) Make chair of department.

6. Associate Professor: In addition to creating new associate professors and deleting existing ones, an associate professor may be promoted to full professor.

7. Assistant Professor: In addition to creating new assistant professors and deleting existing ones, an assistant professor may be promoted to associate professor.

8. Lecturer: In addition to creating new lecturers and deleting existing ones, a lecturer may be promoted to assistant professor.

9. Student instances: the suggested modifications apply to all types of student instances.

   (a) Take one or more additional courses.

   (b) Stop taking one or more courses.

10. Graduate Student: In addition to creating new graduate students and deleting existing ones, a graduate student may become a lecturer, at the same university or at a different university. This requires the deletion of any courses they take as well as having a faculty member as an adviser. It also requires providing them with courses to give and students to advise.

11. Undergraduate Student: In addition to creating new undergraduate students and deleting existing ones, an undergraduate student may become a graduate student, at the same university or at a different university.

12. Project: In addition to creating new projects and deleting existing ones, the following changes may be applied:

    (a) A new budget.

   (b) A new end date.

   (c) Mark as complete if no longer active.

13. Research Group: In addition to creating new research groups and deleting existing ones, the following changes may be applied:

    (a) Name change.

    (b) Merging two research groups together.

    (c) Split a research group into two others.

Note that the following need to be taken into consideration:

1. When deleting university and department instances, a decision needs to be made whether all its faculty, students, courses, etc. need to be deleted as well or moved to a different university or department respectively.
2. When deleting faculty instances or when moving them to new departments:

   (a) Any courses they give, provided they themselves are not deleted as well, need to be assigned new lecturers.

   (b) The students who have the deleted faculty as advisers, provided they themselves are not deleted as well, need to provided a new adviser. They may not necessarily be the case in the real-world, and may not be implemented.
3. When deleting a course, the relation linking it to the faculty that gives it is to be deleted as well.

## C.2 Suggested Categories of Queries

In this section, we describe the proposed categories of queries that the Evaluation Framework may be extended to produce in addition those of LUBM and UOBM.

### C.2.1 Querying Evolving Datasets

The suggested queries below are related to evolving datasets, they have been adapted from EvoGen's queries.

**Retrieval of a specific version:** The user must be able to retrieve a specific version of a dataset.

**Queries on changes:** The user must be able to query the changes a dataset has gone through.

### C.2.2 Querying Provenance Graphs

The suggested queries below are related to the generated provenance graphs. Note that most of the below have been adapted from Chebotko et al. (2012)'s test queries.

1. Dependencies:
   (a) Find all derivation dependencies of all entities.
   (b) Find all derivation dependencies of all processes.
   (c) Find all usage dependencies of all entities.
   (d) Find all generation dependencies of all entities.
   (e) Find all attribution dependencies.
2. Entities: Find all entities which a particular graph depends on.
3. Activities: Find all activities which resulted in a particular graph.
4. Agents:
   (a) What are the graphs that an agent contributed to?
   (b) Chain of Custody: Find all agents whose contribution resulted in a particular graph. This has been suggested by Allen et al. (2015).
5. Time span: Find the time span between the oldest and the newest graph. This has been suggested by Allen et al. (2015)

Appendix D

# Chain of Events

In our main work, we only considered reflecting one update on the Graph $G_C$. We now look at how a chain of updates needs to be propagated, taking into consideration the old entities needed and the entities that need to be fetched or to have been stored.

For readability, we revert to using $G_1$ for $G_{copy(A)}$, $G_2$ for $G_{copy(B)}$, and $G_3$ for $G_C$. The update operation performed on $G_2$ is $Up_{op}$ using the update graph $G^{up}$ and resulting in $G_2$ becoming the updated graph $G_2'$. The second update graph to be applied to any of $G_1$ or $G_2'$ graph is $G^{up'}$ and the third is $G^{up''}$.

For each of the four set operations, we consider 32 cases where a sequence of updates may occur. The first 16 cases are shown in Table D.1 and reflect changes after initially performing an Insert on Graph $G_2$. The second 16 cases are shown in Table D.2 and reflect changes after initially performing a Delete on Graph $G_2$.

## D.1 Union

Inserting into a graph created by performing a union is straightforward regardless of the previous sequence of updates. Only the update needs to be fetched and applied on $G_3$.

Deleting, however, requires additional inspection. In addition to requiring the other source graph, we must check if there has been any updates on it. If no updates were previously performed, then this is straightforward as explained in Table 4.1. If, however, an update had been performed on the other source graph, then there is a need for either that updated graph or the old one and any updates that were applied on it.

The above is shown in Tables D.3, D.4, D.5, D.6, D.7, D.8, D.9, and D.10.

| Event 1 | Event 2 | Event 3 |
|---|---|---|
| Insert $G^{up}$ | Insert $G^{up'}$ into $G_2'$ resulting in $G_2''$. | Insert $G^{up''}$ into $G_2''$. |
| into $G_2$ | | Insert $G^{up''}$ into $G_1$. |
| resulting in $G_2'$. | Propagate update resulting in $G_3'$ | Delete $G^{up''}$ from $G_2''$. |
| | becoming $G_3''$. | Delete $G^{up''}$ from $G_1$. |
| Propagate update | Insert $G^{up'}$ into $G_1$ resulting in $G_1'$. | Insert $G^{up''}$ into $G_2'$ |
| resulting in $G_3$ | | Insert $G^{up''}$ into $G_1'$. |
| becoming $G_3'$. | Propagate update resulting in $G_3'$ | Delete $G^{up''}$ from $G_2'$. |
| | becoming $G_3''$. | Delete $G^{up''}$ from $G_1'$. |
| | Delete $G^{up'}$ from $G_2'$ resulting in $G_2''$. | Insert $G^{up''}$ into $G_2''$. |
| | | Insert $G^{up''}$ into $G_1$. |
| | Propagate update resulting in $G_3'$ | Delete $G^{up''}$ from $G_2''$. |
| | becoming $G_3''$. | Delete $G^{up''}$ from $G_1$. |
| | Delete $G^{up'}$ from $G_1$ resulting in $G_1'$. | Insert $G^{up''}$ into $G_2'$. |
| | | Insert $G^{up''}$ into $G_1'$. |
| | Propagate update resulting in $G_3'$ | Delete $G^{up''}$ from $G_2'$. |
| | becoming $G_3''$. | Delete $G^{up''}$ from $G_1'$. |

Table D.1: Chain of Updates After Insert

| Event 1 | Event 2 | Event 3 |
|---|---|---|
| Delete $G^{up}$ | Insert $G^{up'}$ into $G_2'$ resulting in $G_2''$. | Insert $G^{up''}$ into $G_2''$. |
| from $G_2$ | | Insert $G^{up''}$ into $G_1$. |
| resulting in $G_2'$. | Propagate update resulting in $G_3'$ | Delete $G^{up''}$ from $G_2''$. |
| | becoming $G_3''$. | Delete $G^{up''}$ from $G_1$. |
| Propagate update | Insert $G^{up'}$ into $G_1$ resulting in $G_1'$. | Insert $G^{up''}$ into $G_2'$ |
| resulting in $G_3$ | | Insert $G^{up''}$ into $G_1'$. |
| becoming $G_3'$. | Propagate update resulting in $G_3'$ | Delete $G^{up''}$ from $G_2'$. |
| | becoming $G_3''$. | Delete $G^{up''}$ from $G_1'$. |
| | Delete $G^{up'}$ from $G_2'$ resulting in $G_2''$. | Insert $G^{up''}$ into $G_2''$. |
| | | Insert $G^{up''}$ into $G_1$. |
| | Propagate update resulting in $G_3'$ | Delete $G^{up''}$ from $G_2''$. |
| | becoming $G_3''$. | Delete $G^{up''}$ from $G_1$. |
| | Delete $G^{up'}$ from $G_1$ resulting in $G_1'$. | Insert $G^{up''}$ into $G_2'$. |
| | | Insert $G^{up''}$ into $G_1'$. |
| | Propagate update resulting in $G_3'$ | Delete $G^{up''}$ from $G_2'$. |
| | becoming $G_3''$. | Delete $G^{up''}$ from $G_1'$. |

Table D.2: Chain of Updates After Delete

| $G_3=$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cup G_2$ | Insert into $G_2$, resulting in $G'_2$.<br><br>$(G'_2 = G_2 \cup G^{up})$ | Insert $G^{up}$ into $G_3$, resulting in $G'_3$. Then re-entail.<br><br>$(G'_3 = G_1 \cup G'_2 = G_1 \cup G_2 \cup G^{up})$ | $G_3$ | $G^{up}$ | Insert into $G'_2$, resulting in $G''_2$.<br><br>$(G''_2 = G'_2 \cup G^{up\prime} = G_2 \cup G^{up} \cup G^{up\prime})$ | Insert $G^{up\prime}$ into $G'_3$, resulting in $G''_3$. Then re-entail.<br><br>$(G''_3 = G_1 \cup G''_2 = G_1 \cup G'_2 \cup G^{up\prime} = G_1 \cup G_2 \cup G^{up} \cup G^{up\prime})$ | $G'_3$ | $G^{up\prime}$ | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up\prime\prime}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Insert $G^{up\prime\prime}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_3$ the triples in $G^{up\prime\prime} \setminus G_1$. Then re-entail. | $G''_3$ and $G_1$ | $G^{up\prime\prime}$ and $G_1$ if not cached. |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_3$ the triples in $G^{up\prime\prime} \setminus G''_2$. Then re-entail. This is equivalent to deleting the triples in $[G^{up\prime\prime} \setminus (G_2 \cup G^{up} \cup G^{up\prime})]$ $= G^{up\prime\prime} \setminus (G'_2 \cup G^{up\prime})$ $= (G^{up\prime\prime} \setminus G^{up\prime}) \setminus G'_2$ $= (G^{up\prime\prime} \setminus G'_2) \setminus G^{up\prime}$ $= [(G^{up\prime\prime} \setminus (G_2 \cup G^{up})] \setminus G^{up\prime}$ $= [(G^{up\prime\prime} \setminus G_2) \setminus G^{up}] \setminus G^{up\prime}$ $\equiv [(G^{up\prime\prime} \setminus G^{up}) \setminus G_2] \setminus G^{up\prime}$. | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up\prime}$. Or $G''_3$, $G_2$, $G^{up}$, and $G^{up\prime}$ | $G^{up\prime\prime}$ and $G''_2$ if not cached. Or $G^{up\prime\prime}$, $G'_2$, and $G^{up\prime}$. Or $G^{up\prime\prime}$, $G_2$, $G^{up}$ and $G^{up\prime}$. |

Table D.3: Chain of Updates 1 for Graph Union.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cup G_2$ | Insert into $G_2$, resulting in $G'_2$.<br><br>($G'_2 = G_2 \cup G^{up}$) | Insert $G^{up}$ into $G_3$, resulting in $G'_3$. Then re-entail.<br><br>($G'_3 = G_1 \cup G'_2 = G_1 \cup G_2 \cup G^{up}$) | $G_3$ | $G^{up}$ | Insert into $G_1$, resulting in $G'_1$.<br><br>($G'_1 = G_1 \cup G^{up'}$) | Insert $G^{up'}$ into $G'_3$, resulting in $G''_3$. Then re-entail.<br><br>($G''_3 = G'_1 \cup G'_2 = G_1 \cup G^{up} \cup G_2 \cup G^{up'}$) | $G'_3$ | $G^{up'}$ | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_3$, the triples in $G^{up''} \setminus G'_1$. Then re-entail. This is equivalent to deleting the triples in $G^{up''} \setminus (G_1 \cup G^{up'})$ $= (G^{up''} \setminus G_1) \setminus G^{up'}$ $= (G^{up''} \setminus G^{up'}) \setminus G_1$. | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up'}$ | $G^{up''}$ and $G'_1$ if not cached. Or $G^{up''}$, $G_1$, and $G^{up'}$. |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_3$, the triples in $G^{up''} \setminus G'_2$. Then re-entail. This is equivalent to deleting the triples in $G^{up''} \setminus G_2 \cup G^{up}$ $= (G^{up''} \setminus G_2) \setminus G^{up}$ $= (G^{up''} \setminus G^{up}) \setminus G_2$. | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up'}$ | $G^{up''}$ and $G'_2$ if not cached. Or $G^{up''}$, $G_2$, and $G^{up}$. |

Table D.4: Chain of Updates 2 for Graph Union.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cup G_2$ | Insert into $G_2$, resulting in $G'_2$. <br><br> $(G'_2 = G_2 \cup G^{up})$ | Insert $G^{up}$ into $G_3$, resulting in $G'_3$. Then re-entail. <br><br> $(G'_3 = G_1 \cup G'_2 = G_1 \cup G_2 \cup G^{up})$ | $G_3$ | $G^{up}$ | Delete from $G'_2$, resulting in $G''_2$. <br><br> $(G''_2 = G'_2 \setminus G^{up'}$ $G'_2 \setminus G^{up'} = (G_2 \cup G^{up}) \setminus G^{up'})$. | Delete from $G'_3$ the triples in $G^{up'} \setminus G_1$, resulting in $G''_3$. Then re-entail. <br><br> $(G''_3 = G_1 \cup (G'_2 \setminus G^{up'}) = G_1 \cup [(G_2 \cup G^{up}) \setminus G^{up'}] = G_1 \cup [(G_2 \setminus G^{up'}) \cup (G^{up} \setminus G^{up'})]$. | $G'_3$ and $G_1$. | $G^{up'}$ and $G_1$ if not cached | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_3$ the triples in $G^{up''} \setminus G_1$. Then re-entail. | $G''_3$ and $G_1$ | $G^{up''}$ and $G_1$ if not cached. |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_3$ the triples in $G^{up''} \setminus G''_2$. Then re-entail. This is equivalent to deleting the triples in $G^{up''} \setminus (G'_2 \setminus G^{up'})$ $\equiv G^{up''} \setminus [(G_2 \cup G^{up}) \setminus G^{up'}]$ $\equiv [G^{up''} \setminus (G^{up'} \setminus G'_2)]$ | $G''_3$ and $G''_2$. Or $G''_3$, $G_2$, $G^{up}$, and $G^{up'}$ | $G^{up''}$ and $G''_2$ if not cached. Or $G^{up''}$, $G_2$, $G^{up}$, and $G^{up'}$. |

Table D.5: Chain of Updates 3 for Graph Union.

| $G_3=$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cup G_2$ | Insert into $G_2$, resulting in $G'_2$. <br><br> $(G'_2 = G_2 \cup G^{up})$ | Insert $G^{up}$ into $G_3$, resulting in $G'_3$. Then re-entail. <br><br> $(G'_3 = G_1 \cup G'_2 = G_1 \cup G_2 \cup G^{up})$ | $G_3$ | $G^{up}$ | Delete from $G_1$, resulting in $G'_1$. <br><br> $(G'_1 = G_1 \setminus G^{up})$. | Delete from $G'_3$ the triples in $G^{up'} \setminus G'_2$ resulting in $G''_3$. Then re-entail. <br><br> $G''_3 = G'_1 \cup G'_2 = [(G_2 \cup G^{up}) \cup (G_1 \setminus G^{up'})]$. | $G'_3$ and $G'_2$. Or $G'_3$, $G_2$, and $G^{up}$. | $G^{up'}$ | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_3$, the triples in $G^{up''} \setminus G'_1$. Then re-entail. This is equivalent deleting the triples in $G^{up''} \setminus (G_1 \setminus G^{up'})$. | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up'}$ | $G^{up''}$ and $G'_1$ if not cached. Or $G^{up''}$, $G_1$, and $G^{up'}$. |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_3$, the triples in $G^{up''} \setminus G'_2$. Then re-entail. This is equivalent to deleting the triples in $[G^{up''} \setminus (G_2 \cup G^{up})]$ $= [G^{up''} \setminus (G_2 \setminus G^{up})]$ $= [G^{up''} \setminus (G^{up} \setminus G_2)]$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$ | $G^{up''}$ and $G''_2$ if not cached. Or $G^{up''}$, $G_2$, $G^{up'}$, and $G^{up}$. |

Table D.6: Chain of Updates 4 for Graph Union.

| $G_3=$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cup G_2$ | Delete from $G_2$, resulting in $G'_2$. $(G'_2 = G_2 \setminus G^{up})$ | Delete from $G_3$ the triples in $G^{up} \mid G_1$, resulting in $G'_3$. Then re-entail. $(G'_3 = G_1 \cup G'_2 = G_1 \cup (G_2 \setminus G^{up})$. | $G_3$ and $G_1$. | $G^{up}$ and $G_1$ if not cached. | Insert into $G'_2$, resulting in $G''_2$. $(G''_2 = G'_2 \cup G^{up'} = (G_2 \setminus G^{up}) \cup G^{up'}$. | Insert $G^{up'}$ into $G'_3$, resulting in $G''_3$. Then re-entail. $G''_3$ is equivalent to $G_1 \cup G''_2$. It is also equivalent to $G_1 \cup (G'_2 \cup G^{up'}) \equiv G_1 \cup [(G_2 \setminus G^{up}) \cup G^{up}]$. | $G'_3$ | $G^{up'}$ | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_3$ the triples in $G^{up''} \setminus G_1$. Then re-entail. | $G''_3$ and $G_1$. | $G^{up''}$ and $G_1$ if not cached. |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_3$ the triples in $G^{up''} \setminus G''_2$. Then re-entail. This is equivalent to deleting the triples in $G^{up''} \setminus (G'_2 \cup G^{up'}) = (G^{up''} \setminus G'_2) \setminus G^{up} = G^{up''} \setminus [(G_2 \setminus G^{up}) \cup G^{up}] = [G^{up''} \setminus (G_2 \setminus G^{up})] \setminus G^{up'}$ | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up'}$. Or $G''_3$, $G_2$, $G^{up'}$, and $G^{up}$. | $G^{up''}$ and $G''_2$ if not cached. Or $G^{up''}$, $G'_2$, and $G^{up'}$. Or $G^{up''}$, $G_2$, $G^{up'}$ and $G^{up}$. |

Table D.7: Chain of Updates 5 for Graph Union.

| $G_3$= | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cup G_2$ | Delete from $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \setminus G^{up}$) | Delete from $G_3$ the triples in $G^{up} \setminus G_1$, resulting in $G'_3$. Then re-entail. ($G'_3 = G_1 \cup G'_2 = G_1 \cup (G_2 \setminus G^{up})$. | $G_3$ and $G_1$. | $G^{up}$ and $G_1$ if not cached. | Insert into $G_1$, resulting in $G'_1$. ($G'_1 = G_1 \cup G^{up'}$) | Insert $G^{up'}$ into $G'_3$, resulting in $G''_3$. Then re-entail. ($G''_3 = G'_1 \cup G'_2 = [(G_1 \cup G^{up'}) \cup (G'_2 \setminus G^{up})]$ | $G'_3$ | $G^{up'}$ | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_3$ the triples in $G^{up''} \setminus G'_1$. Then re-entail. This is equivalent to deleting the triples in $[G^{up''} \setminus (G_1 \cup G^{up'})] = (G^{up''} \setminus G_1) \setminus G^{up'} = (G^{up''} \setminus G^{up'}) \setminus G_1$ | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up'}$. | $G^{up''}$ and $G'_1$ if not cached. Or $G^{up''}$, $G_1$, and $G^{up'}$. |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_3$ the triples in $G^{up''} \setminus G'_2$. Then re-entail. This is equivalent to deleting the triples in $[G^{up''} \setminus (G_2 \setminus G^{up})]$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$ | $G^{up''}$ and $G'_2$ if not cached. Or $G^{up''}$, $G_2$, and $G^{up}$. |

Table D.8: Chain of Updates 6 for Graph Union.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cup G_2$ | Delete from $G_2$, resulting in $G'_2$. <br><br> $(G'_2 = G_2 \setminus G^{up})$ | Delete from $G_3$ the triples in $G^{up} \setminus G_1$, resulting in $G'_3$. Then re-entail. <br><br> $(G'_3 = G_1 \cup G'_2 = G_1 \cup (G_2 \setminus G^{up})$. | $G_3$ and $G_1$. | $G^{up}$ and $G_1$ if not cached. | Delete from $G'_2$, resulting in $G''_2$. <br><br> $(G'_2 = G'_2 \setminus G^{up'} = (G_2 \setminus G^{up}) \setminus G^{up'})$ | Delete from $G'_3$ the triples in $G^{up'} \setminus G_1$ resulting in $G''_3$. Then re-entail. <br><br> $(G''_3 = G_1 \cup G''_2 = G_1 \cup (G'_2 \setminus G^{up'}) = G_1 \cup [(G_2 \setminus G^{up}) \setminus G^{up'}]$. | $G_3$ and $G_1$. | $G^{up'}$ and $G_1$ if not cached. | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_3$ the triples in $G^{up''} \setminus G_1$. Then re-entail. | $G''_3$ and $G_1$. | $G^{up''}$ and $G_1$ if not cached. |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_3$ the triples in $G^{up''} \setminus G''_2$. Then re-entail. This is equivalent to deleting the triples in $[G^{up''} \setminus (G'_2 \setminus G^{up'}) \equiv [G^{up''} \setminus ((G_2 \setminus G^{up}) \setminus G^{up'})$. | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up'}$. Or $G''_3$, $G_2$, $G^{up'}$, and $G^{up}$. | $G^{up''}$ and $G''_2$ if not cached. Or $G^{up''}$, $G'_2$, and $G^{up'}$. Or $G^{up''}$, $G_2$, $G^{up'}$ and $G^{up}$. |

Table D.9: Chain of Updates 7 for Graph Union.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cup G_2$ | Delete from $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \setminus G^{up}$) | Delete from $G_3$ the triples in $G^{up} \setminus G_1$, resulting in $G'_3$. Then re-entail. ($G'_3 = G_1 \cup G'_2 = G_1 \cup (G_2 \setminus G^{up})$). | $G_3$ and $G_1$. | $G^{up}$ and $G_1$ if not cached. | Delete from $G_1$, resulting in $G'_1$. ($G'_1 = G_1 \setminus G^{up'}$) | Delete from $G'_3$ the triples in $G^{up'} \setminus G'_2$ resulting in $G''_3$. Then re-entail. ($G''_3 = G'_1 \cup G'_2 = [G_1 \setminus G^{up'}) \cup (G_2 \setminus G^{up})]$). | $G_3$ and $G'_2$. Or $G_3$, $G_2$, and $G^{up}$. | $G^{up'}$ and $G'_2$ if not cached. Or $G^{up'}$, $G_2$, and $G^{up}$. | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up''}$ into $G''_3$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_3$ the triples in $G^{up''} \setminus G'_1$. Then re-entail. This is equivalent to deleting the triples in $[G^{up''} \setminus (G_1 \setminus G^{up'})]$ | $G''_3$ and $G'_1$ if not cached. Or $G''_3$, $G_1$, and $G^{up'}$. | $G^{up''}$ and $G'_1$ if not cached. Or $G^{up''}$, $G_1$, and $G^{up'}$. |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_3$ the triples in $G^{up''} \setminus G'_2$. Then re-entail. This is equivalent to deleting the triples in $[G^{up''} \setminus (G_2 \setminus G^{up})]$ | $G''_3$ and $G'_2$ if not cached. Or $G''_3$, $G_2$, and $G^{up}$. | $G^{up''}$ and $G'_2$ if not cached. Or $G^{up''}$, $G_2$, and $G^{up}$. |

Table D.10: Chain of Updates 8 for Graph Union.

## D.2 Intersection

Inserting into a graph created by performing an intersection requires additional inspection. In addition to requiring the other source graph, we must check if there has been any updates on it. If no updates were previously performed, then this is straightforward as explained in Table 4.1. If, however, an update had been performed on the other source graph, then there is a need for either that updated graph or the old one and any updates that were applied on it.

Deleting is straightforward regardless of the previous sequence of updates. Only the update needs to be fetched and applied on $G_3$.

The above is shown in Tables D.11, D.12, D.13, D.14, D.15, D.16, D.17, and D.18.

## D.3 Difference Case 1

Similar to intersection, inserting into a graph created by performing an intersection requires additional inspection. In addition to requiring the other source graph, we must check if there has been any updates on it. If no updates were previously performed, then this is straightforward as explained in Table 4.1. If, however, an update had been performed on the other source graph, then there is a need for either that updated graph or the old one and any updates that were applied on it.

Deleting is straightforward regardless of the previous sequence of updates. Only the update needs to be fetched and applied on $G_3$.

The above is shown in Tables D.19, D.20, D.21, D.22, D.23, D.24, D.25, and D.26.

## D.4 Difference Case 2

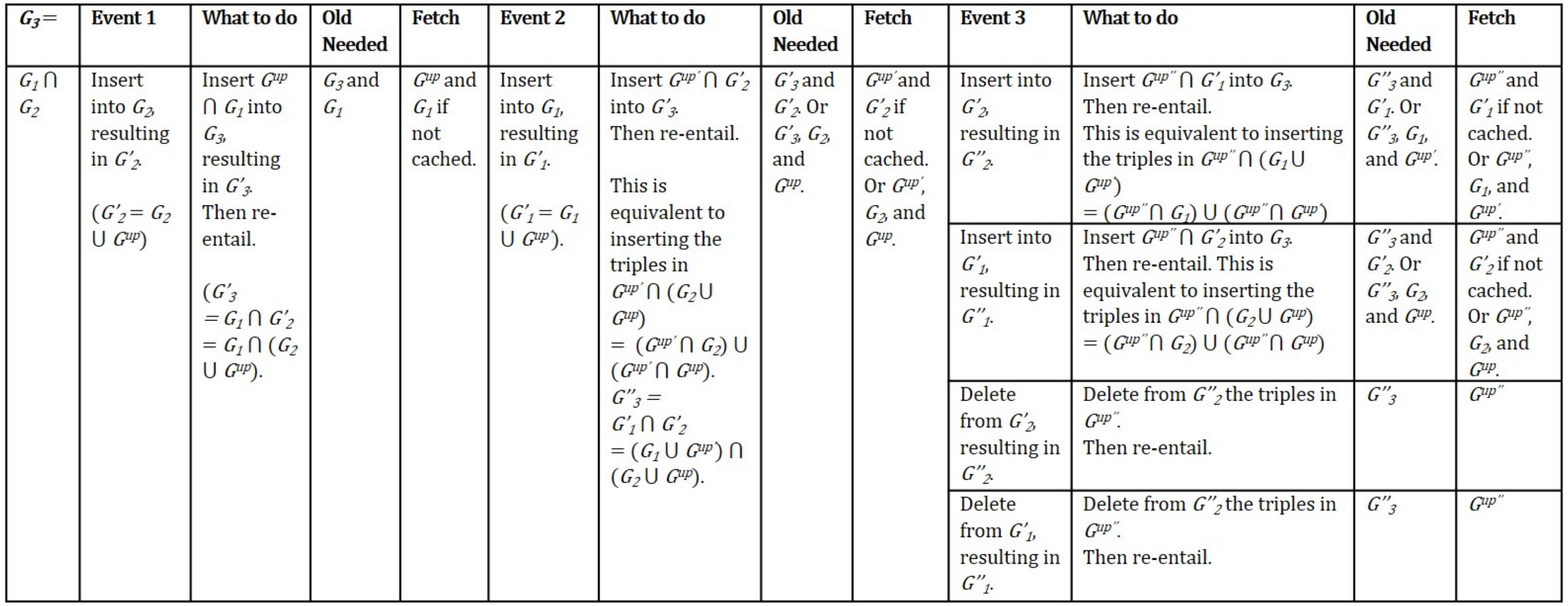
Similar to intersection and difference case 1, inserting into a graph created by performing an intersection requires additional inspection. In addition to requiring the other source graph, we must check if there has been any updates on it. If no updates were previously performed, then this is straightforward as explained in Table 4.1. If, however, an update had been performed on the other source graph, then there is a need for either that updated graph or the old one and any updates that were applied on it.

Deleting is straightforward regardless of the previous sequence of updates. Only the update needs to be fetched and applied on $G_3$.

The above is shown in Tables D.27, D.28, D.29, D.30, D.31, D.32, D.33, and D.30.

| $G_3=$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cap G_2$ | Insert into $G_2$, resulting in $G'_2$.<br><br>$(G'_2 = G_2 \cup G^{up})$ | Insert $G^{up} \cap G_1$ into $G_3$, resulting in $G'_3$. Then re-entail.<br><br>$(G'_3 = G_1 \cap G'_2 = G_1 \cap (G_2 \cup G^{up})$. | $G_3$ and $G_1$ | $G^{up}$ and $G_1$ if not cached. | Insert into $G'_2$, resulting in $G''_2$.<br><br>$(G''_2 = G'_2 \cup G^{up'} = G_2 \cup G^{up} \cup G^{up'})$. | Insert $G^{up'} \cap G_1$ into $G'_3$. Then re-entail.<br><br>$(G''_3 = G_1 \cap (G'_2 \cup G^{up'}) = G_1 \cap (G_2 \cup G^{up} \cup G^{up'})$. | $G'_3$ and $G_1$ | $G^{up'}$ and $G_1$ if not cached. | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up''} \cap G_1$ into $G_3$. Then re-entail. | $G''_3$ and $G_1$ | $G^{up''}$ and $G_1$ if not cached. |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Insert $G^{up''} \cap G''_2$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G'_2 \cup G^{up'})$<br>$= (G^{up''} \cap G'_2) \cup (G^{up''} \cap G^{up'})$<br>$= G^{up''} \cap (G_2 \cup G^{up} \cup G^{up'})$ | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up'}$. Or $G''_3$, $G_2$, $G^{up}$, and $G^{up'}$ | $G^{up''}$ and $G''_2$ if not cached. Or $G^{up''}$, $G'_2$, and $G^{up'}$. Or $G^{up''}$, $G_2$, $G^{up}$ and $G^{up'}$. |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |

Table D.11: Chain of Updates 1 for Graph Intersection.



| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cap G_2$ | Insert into $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \cup G^{up}$) | Insert $G^{up} \cap G_1$ into $G_3$, resulting in $G'_3$. Then re-entail. ($G'_3 = G_1 \cap G'_2 = G_1 \cap (G_2 \cup G^{up})$). | $G_3$ and $G_1$ | $G^{up}$ and $G_1$ if not cached. | Insert into $G_1$, resulting in $G'_1$. ($G'_1 = G_1 \cup G^{up'}$). | Insert $G^{up'} \cap G'_2$ into $G'_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up'} \cap (G_2 \cup G^{up})$ $= (G^{up'} \cap G_2) \cup (G^{up'} \cap G^{up})$. $G''_3 = G'_1 \cap G'_2 = (G_1 \cup G^{up'}) \cap (G_2 \cup G^{up})$. | $G'_3$ and $G'_2$. Or $G'_3$, $G_2$, and $G^{up}$. | $G^{up'}$ and $G'_2$ if not cached. Or $G^{up'}$, $G_2$, and $G^{up}$. | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up''} \cap G'_1$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G_1 \cup G^{up'})$ $= (G^{up''} \cap G_1) \cup (G^{up''} \cap G^{up'})$ | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up'}$. | $G^{up''}$ and $G'_1$ if not cached. Or $G^{up''}$, $G_1$, and $G^{up'}$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up''} \cap G'_2$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G_2 \cup G^{up})$ $= (G^{up''} \cap G_2) \cup (G^{up''} \cap G^{up})$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$. | $G^{up''}$ and $G'_2$ if not cached. Or $G^{up''}$, $G_2$, and $G^{up}$. |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |

Table D.12: Chain of Updates 2 for Graph Intersection.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cap G_2$ | Insert into $G_2$, resulting in $G'_2$. <br><br> $(G'_2 = G_2 \cup G^{up})$ | Insert $G^{up} \cap G_1$ into $G_3$, resulting in $G'_3$. Then re-entail. <br><br> $(G'_3 = G_1 \cap G'_2 = G_1 \cap (G_2 \cup G^{up})$. | $G_3$ and $G_1$ | $G^{up}$ and $G_1$ if not cached. | Delete from $G'_2$, resulting in $G''_2$. <br><br> $(G''_2 = G'_2 \setminus G^{up\prime} = (G_2 \cup G^{up}) \setminus G^{up\prime}$. <br><br> $(G^{up\prime} = G_2 \setminus G'_2 = [G_2 \setminus (G_2 \cup G^{up})])$. | Delete from $G'_2$ the triples in $G^{up\prime}$. Then re-entail. <br><br> $(G''_3 = G_1 \cap G''_2 = G_1 \cap (G'_2 \setminus G^{up\prime}) = G_1 \cap [(G_2 \cup G^{up}) \setminus G^{up\prime}]$ | $G''_3$ | $G^{up\prime}$ | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up''} \cap G_1$ into $G_3$. Then re-entail. | $G''_3$ and $G_1$ | $G^{up''}$ and $G_1$ if not cached. |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Insert $G^{up''} \cap G''_2$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G'_2 \setminus G^{up\prime}) = (G^{up''} \cap G'_2) \setminus (G^{up''} \cap G^{up\prime}) = G^{up''} \cap [(G_2 \cup G^{up}) \setminus G^{up\prime}] = [G^{up''} \cap (G_2 \cup G^{up})] \setminus (G^{up''} \cap G^{up\prime})$ | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up\prime}$. Or $G''_3$, $G_2$, $G^{up}$, and $G^{up\prime}$ | $G^{up''}$ and $G''_2$ if not cached. Or $G^{up''}$, $G'_2$, and $G^{up\prime}$. Or $G^{up''}$, $G_2$, $G^{up}$ and $G^{up\prime}$. |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |

Table D.13: Chain of Updates 3 for Graph Intersection.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cap G_2$ | Insert into $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \cup G^{up}$) | Insert $G^{up} \cap G_1$ into $G_3$, resulting in $G'_3$. Then re-entail. ($G'_3 = G_1 \cap G'_2 = G_1 \cap (G_2 \cup G^{up})$. | $G_3$ and $G_1$ | $G^{up}$ and $G_1$ if not cached. | Delete from $G_1$, resulting in $G'_1$. ($G'_1 = G_1 \setminus G^{up}$) ($G^{up} = G_1 \setminus G'_1$). | Delete from $G'_2$ the triples in $G^{up\prime}$. Then re-entail. ($G''_3 = G'_1 \cap G'_2 = [(G_1 \setminus G^{up}) \cap (G_2 \cup G^{up})]$) | $G''_3$ | $G^{up\prime}$ | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up\prime\prime} \cap G'_1$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G_1 \cup G^{up\prime}) \equiv (G^{up\prime\prime} \cap G_1) \cup (G^{up\prime\prime} \cap G^{up\prime})$ | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up\prime}$. | $G^{up\prime\prime}$ and $G'_1$ if not cached. Or $G^{up\prime\prime}$, $G_1$, and $G^{up\prime}$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up\prime\prime} \cap G'_2$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G_2 \cup G^{up}) \equiv (G^{up\prime\prime} \cap G_2) \cup (G^{up\prime\prime} \cap G^{up})$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$. | $G^{up\prime\prime}$ and $G'_2$ if not cached. Or $G^{up\prime\prime}$, $G_2$, and $G^{up}$. |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_2$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_2$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |

Table D.14: Chain of Updates 4 for Graph Intersection.

| $G_3=$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cap G_2$ | Delete from $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \setminus G^{up}$). | Delete from $G_3$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail. $G'_3 = G_1 \cap G'_2 = G_1 \cap (G_2 \setminus G^{up})$. | $G_3$ | $G^{up}$ | Insert into $G'_2$, resulting in $G''_2$. ($G''_2 = G'_2 \cup G^{up'} = [(G_2 \setminus G^{up}) \cup G^{up}]$ | Insert $G^{up'} \cap G_1$ into $G'_3$. Then re-entail. ($G''_3 = G_1 \cap G''_2$. $= G_1 \cap (G_2 \setminus G^{up})$. | $G'_3$ and $G'_2$. Or $G'_3$, $G_2$, and $G^{up}$. | $G^{up'}$ and $G'_2$ if not cached. Or $G^{up'}$, $G_2$, and $G^{up}$. | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up''} \cap G_1$ into $G''_3$. Then re-entail. | $G''_3$ and $G_1$ | $G^{up''}$ and $G_1$ if not cached. |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Insert $G^{up''} \cap G''_2$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G'_2 \setminus G^{up'})$ $= (G^{up''} \cap G'_2) \setminus (G^{up''} \cap G^{up'})$ $= G^{up''} \cap [(G_2 \setminus G^{up}) \setminus G^{up}]$ | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up'}$. Or $G''_3$, $G_2$, $G^{up}$, and $G^{up'}$ | $G^{up''}$ and $G''_2$ if not cached. Or $G^{up''}$, $G'_2$, and $G^{up'}$. Or $G^{up''}$, $G_2$, $G^{up}$ and $G^{up'}$. |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |

Table D.15: Chain of Updates 5 for Graph Intersection.

| **$G_3$=** | **Event 1** | **What to do** | **Old Needed** | **Fetch** | **Event 2** | **What to do** | **Old Needed** | **Fetch** | **Event 3** | **What to do** | **Old Needed** | **Fetch** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cap G_2$ | Delete from $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \setminus G^{up}$). | Delete from $G_3$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail. ($G'_3 = G_1 \cap G'_2 = [G_1 \cap (G_2 \setminus G^{up})]$). | $G_3$ | $G^{up}$ | Insert into $G_1$, resulting in $G'_1$. ($G'_1 = G_1 \cup G^{up'}$). | Insert $G^{up'} \cap G'_2$ into $G'_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up'} \cap (G_2 \setminus G^{up}) = (G^{up'} \cap G_2) \setminus (G^{up'} \cap G^{up})$. ($G''_3 = G'_1 \cap G'_2 = [(G_1 \cup G^{up'}) \cap (G_2 \setminus G^{up})]$). | $G'_3$ and $G'_2$. Or $G'_3$, $G_2$ and $G^{up}$. | $G^{up'}$ and $G'_2$ if not cached. Or $G^{up'}$, $G_2$, and $G^{up}$. | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up''} \cap G'_1$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G_1 \cup G^{up'}) \equiv (G^{up''} \cap G_1) \cup (G^{up''} \cap G^{up'})$ | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up'}$. | $G^{up''}$ and $G'_1$ if not cached. Or $G^{up''}$, $G_1$, and $G^{up'}$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up''} \cap G'_2$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G_2 \setminus G^{up}) \equiv (G^{up''} \cap G_2) \setminus (G^{up''} \cap G^{up})$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$. | $G^{up''}$ and $G'_2$ if not cached. Or $G^{up''}$, $G_2$, and $G^{up}$. |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |

Table D.16: Chain of Updates 6 for Graph Intersection.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cap G_2$ | Delete from $G_2$, resulting in $G'_2$.<br><br>$(G'_2 = G_2 \setminus G^{up})$. | Delete from $G_3$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail.<br><br>$(G'_3 = G_1 \cap G'_2 = [G_1 \cap (G_2 \setminus G^{up})])$. | $G_3$ | $G^{up}$ | Delete from $G'_2$, resulting in $G''_2$.<br><br>$(G'_2 = G'_2 \setminus G^{up'} = [(G_2 \setminus G^{up}) \setminus G^{up'}])$ | Delete from $G'_2$ the triples in $G^{up'}$. Then re-entail.<br><br>$(G''_3 = G_1 \cap G''_2 = G_1 \cap (G'_2 \setminus G^{up'}) = G_1 \cap [(G_2 \setminus G^{up}) \setminus G^{up'}])$ | $G''_3$ | $G^{up'}$ | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up''} \cap G_1$ into $G''_3$. Then re-entail. | $G''_3$ and $G_1$ | $G^{up''}$ and $G_1$ if not cached. |
| | | | | | | | | | Insert into $G_1$, resulting in $G_1$. | Insert $G^{up''} \cap G''_2$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G'_2 \setminus G^{up'})$ $= (G^{up''} \cap G'_2) \setminus (G^{up''} \cap G^{up'})$ $= G^{up''} \cap [(G_2 \setminus G^{up}) \setminus G^{up'}]$ | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up'}$. Or $G''_3$, $G_2$, $G^{up}$, and $G^{up'}$ | $G^{up''}$ and $G''_2$ if not cached. Or $G^{up''}$, $G'_2$, and $G^{up'}$. Or $G^{up''}$, $G_2$, $G^{up}$ and $G^{up'}$. |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |

Table D.17: Chain of Updates 7 for Graph Intersection.

| **$G_3 =$** | **Event 1** | **What to do** | **Old Needed** | **Fetch** | **Event 2** | **What to do** | **Old Needed** | **Fetch** | **Event 3** | **What to do** | **Old Needed** | **Fetch** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \cap G_2$ | Delete from $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \setminus G^{up}$). | Delete from $G_3$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail. ($G'_3 = G_1 \cap G'_2 = [G_1 \cap (G_2 \setminus G^{up})]$) | $G_3$ | $G^{up}$ | Delete from $G_1$, resulting in $G'_1$. ($G'_1 = G_1 \setminus G^{up'}$) | Delete from $G'_2$ the triples in $G^{up'}$. Then re-entail. ($G''_3 = G'_1 \cap G'_2 = [(G_1 \setminus G^{up'}) \cap (G_2 \setminus G^{up})]$) | $G''_3$ | $G^{up'}$ | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up''} \cap G'_1$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G_1 \setminus G^{up'})$ $= (G^{up''} \cap G_1) \setminus (G^{up''} \cap G^{up'})$ | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up'}$. | $G^{up''}$ and $G'_1$ if not cached. Or $G^{up''}$, $G_1$, and $G^{up'}$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up''} \cap G'_2$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G_2 \setminus G^{up})$ $= (G^{up''} \cap G_2) \setminus (G^{up''} \cap G^{up})$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$. | $G^{up''}$ and $G'_2$ if not cached. Or $G^{up''}$, $G_2$, and $G^{up}$. |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |

Table D.18: Chain of Updates 8 for Graph Intersection.

| $G_3=$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event $_3$ | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \setminus G_2$ | Insert into $G_2$, resulting in $G'_2$. <br><br> $(G'_2 = G_2 \cup G^{up})$ | Delete from $G_3$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail. <br><br> $[G'_3 = G_1 \setminus G'_2 = G_1 \setminus (G_2 \cup G^{up}) = (G_1 \setminus G_2) \setminus G^{up} = (G_1 \setminus G^{up}) \setminus G_2.]$ | $G_3$ | $G^{up}$ | Insert into $G'_2$, resultin g in $G''_2$. <br><br> $(G''_2 = G'_2 \cup G^{up'} = G_2 \cup G^{up} \cup G^{up'})$. | Delete from $G'_3$ the triples in $G^{up'}$, resulting in $G''_3$. Then re-entail. <br><br> $(G''_3 = G_1 \setminus (G'_2 \cup G^{up'}) = [G_1 \setminus (G_2 \cup G^{up} \cup G^{up'})])$. | $G'_3$ | $G^{up'}$ | Insert into $G''_2$, resulting in $G'''_2$. | Delete from $G''_3$ the triples in $G^{up''}$. <br> Then re-entail. | $G''_3$ | $G^{up''}$. |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Insert $G^{up''} \setminus G''_2$ into $G''_3$. <br> Then re-entail. <br> This is equivalent to inserting the triples in $G^{up''} \setminus (G'_2 \cup G^{up'})$ <br> $= (G^{up''} \setminus G'_2) \setminus G^{up'}$ <br> $= [G^{up''} \setminus (G_2 \cup G^{up})] \setminus G^{up'}$ <br> $= [(G^{up''} \setminus G_2) \setminus G^{up})] \setminus G^{up'}$ | $G''_3$ and $G''_2$. <br> Or $G''_3$, $G'_2$, and $G^{up'}$. <br> Or $G''_3$, $G_2$, $G^{up}$, and $G^{up'}$ | $G^{up''}$ and $G''_2$ if not cached. <br> Or $G^{up''}$, $G'_2$, and $G^{up'}$. <br> Or $G^{up''}$, $G_2$, $G^{up}$ and $G^{up'}$. |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Insert $G^{up''} \cap G_1$ into $G''_3$ <br> Then re-entail. | $G''_3$ and $G_1$. | $G^{up''}$ and $G_1$ if not cached. |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_2$ the triples in $G^{up''}$. <br> Then re-entail. | $G''_3$ | $G^{up''}$ |

Table D.19: Chain of Updates 1 for Graph Difference 1.

| $G_3$= | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event $_3$ | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \setminus G_2$ | Insert into $G_2$, resulting in $G'_2$.<br><br>($G'_2 = G_2 \cup G^{up}$) | Delete from $G_3$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail.<br><br>[$G'_3 = G_1 \setminus G'_2 = G_1 \setminus (G_2 \cup G^{up}) = (G_1 \setminus G_2) \setminus G^{up} = (G_1 \setminus G^{up}) \setminus G_2$.] | $G_3$ | $G^{up}$ | Insert into $G_1$, resulting in $G'_1$.<br><br>($G'_1 = G_1 \setminus G^{up\prime}$) | Insert $G^{up\prime} \setminus G'_2$ into $G'_3$, resulting in $G''_3$. Then re-entail.<br><br>This is equivalent to inserting the triples in $G^{up\prime} \setminus (G_2 \cup G^{up}) = (G^{up\prime} \setminus G_2) \setminus G^{up}$.<br><br>($G''_3 = G'_1 \setminus G'_2 = [(G_1 \cup G^{up\prime}) \setminus (G_2 \cup G^{up})]$). | $G'_3$ and $G'_2$.<br>Or $G'_3$, $G_2$, and $G^{up}$. | $G^{up\prime}$ and $G'_2$ if not cached.<br>Or $G^{up\prime}$, $G_2$, and $G^{up}$. | Insert into $G'_2$, resulting in $G''_2$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$.<br>Then re-entail. | $G''_3$ | $G^{up\prime\prime}$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up\prime\prime} \setminus G'_2$ into $G''_3$.<br>Then re-entail.<br>This is equivalent to inserting the triples in $G^{up\prime\prime} \setminus (G_2 \cup G^{up}) = (G^{up\prime} \setminus G_2) \setminus G^{up}$ | $G''_3$ and $G'_2$.<br>Or $G''_3$, $G_2$, and $G^{up}$. | $G^{up\prime\prime}$ and $G'_2$ if not cached.<br>Or $G^{up\prime\prime}$, $G_2$, and $G^{up}$. |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Insert $G^{up\prime\prime} \cap G'_1$ into $G''_3$.<br>Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G_1 \cup G^{up\prime}) = (G^{up\prime\prime} \cap G_1) \cup (G^{up\prime\prime} \cap G^{up\prime})$ | $G''_3$ and $G'_1$.<br>Or $G''_3$, $G_1$, and $G^{up\prime}$. | $G^{up\prime\prime}$ and $G'_1$ if not cached.<br>Or $G^{up\prime\prime}$, $G_1$, and $G^{up\prime}$. |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |

Table D.20: Chain of Updates 2 for Graph Difference 1.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event $_3$ | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \setminus G_2$ | Insert into $G_2$, resulting in $G'_2$. <br><br> $(G'_2 = G_2 \cup G^{up})$ | Delete from $G_3$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail. <br><br> $[G'_3$ <br> $= G_1 \setminus G'_2$ <br> $= G_1 \setminus (G_2 \cup G^{up})$ <br> $= (G_1 \setminus G_2) \setminus G^{up}$ <br> $= (G_1 \setminus G^{up}) \setminus G_2]$ | $G_3$ | $G^{up}$ | Delete from $G'_2$, resulting in $G''_2$. <br><br> $(G''_2 = G'_2 \setminus G^{up\prime} = (G_2 \cup G^{up}) \setminus G^{up\prime}$ <br><br> $(G^{up\prime}$ <br> $= G_2 \setminus G'_2$ <br> $= [G_2 \setminus (G_2 \cup G^{up})])$. | Insert $G^{up\prime} \cap G_1$ into $G'_3$, resulting in $G''_3$. Then re-entail. | $G'_3$ and $G_1$. | $G^{up\prime}$ and $G_1$ if not cached. | Insert into $G''_2$, resulting in $G'''_2$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. <br> Then re-entail. | $G''_3$ | $G^{up\prime\prime}$. |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Insert $G^{up\prime\prime} \setminus G''_2$ into $G_3$. <br> Then re-entail. <br> This is equivalent to inserting the triples in $G^{up\prime\prime} \setminus (G'_2 \setminus G^{up\prime})$ <br> $= G^{up\prime\prime} \setminus [(G_2 \cup G^{up}) \setminus G^{up\prime}]$ | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up\prime}$. Or $G''_3$, $G_2$, $G^{up}$, and $G^{up\prime}$ | $G^{up\prime\prime}$ and $G''_2$ if not cached. Or $G^{up\prime\prime}$, $G'_2$, and $G^{up\prime}$. Or $G^{up\prime\prime}$, $G_2$, $G^{up}$ and $G^{up\prime}$. |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Insert $G^{up\prime\prime} \cap G_1$ into $G''_3$. <br> Then re-entail. | $G''_3$ and $G_1$. | $G^{up\prime\prime}$ and $G_1$ if not cached. |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_2$ the triples in $G^{up\prime\prime}$. <br> Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |

Table D.21: Chain of Updates 3 for Graph Difference 1.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \setminus G_2$ | Insert into $G_2$, resulting in $G'_2$. <br><br> $(G'_2 = G_2 \cup G^{up})$ | Delete from $G_3$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail. <br><br> $[G'_3 = G_1 \setminus G'_2 = G_1 \setminus (G_2 \cup G^{up}) = (G_1 \setminus G_2) \setminus G^{up} = (G_1 \setminus G^{up}) \setminus G_2]$ | $G_3$ | $G^{up}$ | Delete from $G_1$, resulting in $G'_1$. <br><br> $(G'_1 = G_1 \setminus G^{up})$ <br><br> $(G^{up} = G_1 \setminus G'_1)$. | Delete from $G'_3$ the triples in $G^{up\prime}$, resulting in $G''_3$. Then re-entail. <br><br> $(G''_3 = [(G_1 \setminus G^{up\prime}) \setminus G'_2] = [(G_1 \setminus G^{up\prime}) \setminus (G_2 \cup G^{up})])$. | $G'_3$ | $G^{up\prime}$ | Insert into $G'_2$, resulting in $G''_2$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. <br> Then re-entail. | $G''_3$ | $G^{up\prime\prime}$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up\prime\prime} \setminus G'_2$ into $G''_3$. <br> Then re-entail. <br> This is equivalent to inserting the triples in <br> $G^{up\prime\prime} \setminus (G_2 \cup G^{up})$ <br> $= (G^{up\prime} \setminus G_2) \setminus G^{up}$ | $G''_3$ and $G'_2$. <br> Or $G''_3$, $G_2$, and $G^{up}$. | $G^{up\prime\prime}$ and $G'_2$ if not cached. <br> Or $G^{up\prime\prime}$, $G_2$, and $G^{up}$. |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Insert $G^{up\prime\prime} \cap G'_1$ into $G''_3$. <br> Then re-entail. This is equivalent to inserting the triples in <br> $G^{up\prime\prime} \cap (G_1 \setminus G^{up\prime})$ <br> $= (G^{up\prime\prime} \cap G_1) \setminus (G^{up\prime\prime} \cap G^{up\prime})$ | $G''_3$ and $G'_1$. <br> Or $G''_3$, $G_1$, and $G^{up\prime}$. | $G^{up\prime\prime}$ and $G'_1$ if not cached. <br> Or $G^{up\prime\prime}$, $G_1$, and $G^{up\prime}$. |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_2$ the triples in $G^{up\prime\prime}$. <br> Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |

Table D.22: Chain of Updates 4 for Graph Difference 1.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \setminus G_2$ | Delete from $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \setminus G^{up}$) | Insert $G^{up} \cap G_1$ into $G_3$, resulting in $G'_3$. Then re-entail. [$G'_3 = G_1 \setminus G'_2 = G_1 \setminus (G_2 \setminus G^{up})$]. | $G_3$ and $G_1$. | $G^{up}$ and $G_1$ if not cached. | Insert into $G'_2$, resulting in $G''_2$. ($G''_2 = G'_2 \cup G^{up'} = [(G_2 \setminus G^{up}) \cup G^{up'}$)] | Delete from $G'_3$ the triples in $G^{up'}$, resulting in $G''_3$. Then re-entail. ($G''_3 = G_1 \setminus (G'_2 \cup G^{up'}) = G_1 \setminus [(G_2 \setminus G^{up}) \cup G^{up'}]$). | $G'_3$ | $G^{up'}$ | Insert into $G''_2$, resulting in $G'''_2$. | Delete from $G''_3$ the triples in $G^{up''}$, resulting in $G'''_3$. Then re-entail. | $G''_3$ | $G^{up''}$. |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Insert $G^{up''} \setminus G''_2$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \setminus (G'_2 \cup G^{up'})$ $= (G^{up''} \setminus G'_2) \setminus G^{up'}$ $= G^{up''} \setminus [(G_2 \setminus G^{up}) \cup G^{up'}]$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$. | $G^{up''}$ and $G'_2$ if not cached. Or $G^{up''}$, $G_2$, and $G^{up}$. |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Insert $G^{up''} \cap G_1$ into $G''_3$. Then re-entail. | $G''_3$ and $G_1$. | $G^{up''}$ and $G_1$ if not cached. |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |

Table D.23: Chain of Updates 5 for Graph Difference 1.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \setminus G_2$ | Delete from $G_2$, resulting in $G'_2$. <br><br> ($G'_2 = G_2 \setminus G^{up}$). | Delete from $G_3$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail. <br><br> ($G'_3 = G_1 \cap G'_2 = [G_1 \cap (G_2 \setminus G^{up})]$). | $G_3$ | $G^{up}$ | Insert into $G_1$, resulting in $G'_1$. <br><br> ($G'_1 = G_1 \cup G^{up\prime}$). | Insert $G^{up\prime} \cap G'_2$ into $G'_3$. Then re-entail. <br><br> This is equivalent to inserting the triples in $G^{up\prime} \cap (G_2 \setminus G^{up}) = (G^{up\prime} \cap G_2) \setminus (G^{up\prime} \cap G^{up})$. <br><br> ($G''_3 = G'_1 \cap G'_2 = [(G_1 \cup G^{up\prime}) \cap (G_2 \setminus G^{up})]$). | $G'_3$ and $G'_2$. Or $G'_3$, $G_2$, and $G^{up}$. | $G^{up\prime}$ and $G'_2$ if not cached. Or $G^{up\prime}$, $G_2$, and $G^{up}$. | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up\prime\prime} \cap G'_1$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G_1 \cup G^{up\prime}) \equiv (G^{up\prime\prime} \cap G_1) \cup (G^{up\prime\prime} \cap G^{up\prime})$ | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up\prime}$. | $G^{up\prime\prime}$ and $G'_1$ if not cached. Or $G^{up\prime\prime}$, $G_1$, and $G^{up\prime}$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up\prime\prime} \cap G'_2$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G_2 \setminus G^{up}) \equiv (G^{up\prime\prime} \cap G_2) \setminus (G^{up\prime\prime} \cap G^{up})$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$. | $G^{up\prime\prime}$ and $G'_2$ if not cached. Or $G^{up\prime\prime}$, $G_2$, and $G^{up}$. |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_2$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_2$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |

Table D.24: Chain of Updates 6 for Graph Difference 1.

| $G_3=$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \setminus G_2$ | Delete from $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \setminus G^{up}$). | Delete from $G_3$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail. ($G'_3 = G_1 \cap G'_2 = [G_1 \cap (G_2 \setminus G^{up})]$). | $G_3$ | $G^{up}$ | Delete from $G'_2$, resulting in $G''_2$. ($G'_2 = G'_2 \setminus G^{up'} = [(G_2 \setminus G^{up}) \setminus G^{up}]$) | Delete from $G'_2$ the triples in $G^{up'}$. Then re-entail. ($G''_3 = G_1 \cap G''_2 = G_1 \cap (G'_2 \setminus G^{up'}) = G_1 \cap [(G_2 \setminus G^{up}) \setminus G^{up'}]$) | $G''_3$ | $G^{up'}$ | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up''} \cap G_1$ into $G''_3$. Then re-entail. | $G''_3$ and $G_1$ | $G^{up''}$ and $G_1$ if not cached. |
| | | | | | | | | | Insert into $G_1$, resulting in $G_1$. | Insert $G^{up''} \cap G''_2$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G'_2 \setminus G^{up'})$ $= (G^{up''} \cap G'_2) \setminus (G^{up''} \cap G^{up'})$ $= G^{up''} \cap [(G_2 \setminus G^{up}) \setminus G^{up}]$ | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up'}$. Or $G''_3$, $G_2$, $G^{up}$, and $G^{up'}$ | $G^{up''}$ and $G''_2$ if not cached. Or $G^{up''}$, $G'_2$, and $G^{up'}$. Or $G^{up''}$, $G_2$, $G^{up}$ and $G^{up'}$. |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |

Table D.25: Chain of Updates 7 for Graph Difference 1.

| $G_3=$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_1 \setminus G_2$ | Delete from $G_2$, resulting in $G'_2$.<br><br>$(G'_2 = G_2 \setminus G^{up})$. | Delete from $G_3$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail.<br><br>$(G'_3 = G_1 \cap G'_2 = [G_1 \cap (G_2 \setminus G^{up})])$ | $G_3$ | $G^{up}$ | Delete from $G_1$, resulting in $G'_1$.<br><br>$(G'_1 = G_1 \setminus G^{up'})$ | Delete from $G'_2$ the triples in $G^{up'}$. Then re-entail.<br><br>$(G''_3 = G'_1 \cap G'_2 = [(G_1 \setminus G^{up'}) \cap (G_2 \setminus G^{up})])$ | $G''_3$ | $G^{up'}$ | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up''} \cap G'_1$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G_1 \setminus G^{up'})$ $= (G^{up''} \cap G_1) \setminus (G^{up''} \cap G^{up'})$ | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up'}$. | $G^{up''}$ and $G'_1$ if not cached. Or $G^{up''}$, $G_1$, and $G^{up'}$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Insert $G^{up''} \cap G'_2$ into $G_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up''} \cap (G_2 \setminus G^{up})$ $= (G^{up''} \cap G_2) \setminus (G^{up''} \cap G^{up})$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$. | $G^{up''}$ and $G'_2$ if not cached. Or $G^{up''}$, $G_2$, and $G^{up}$. |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Delete from $G''_2$ the triples in $G^{up''}$. Then re-entail. | $G''_3$ | $G^{up''}$ |

Table D.26: Chain of Updates 8 for Graph Difference 1.

| $G_3=$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_2 \setminus G_1$ | Insert into $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \cup G^{up}$) | Insert $G^{up} \setminus G_1$ into $G_3$, resulting in $G'_3$. Then re-entail. [$G'_3 = G'_2 \setminus G_1 = (G_2 \cup G^{up}) \setminus G_1$]. | $G_3$ and $G_1$. | $G^{up}$ and $G_1$ if not cached. | Insert into $G'_2$, resulting in $G''_2$. ($G''_2 = G'_2 \cup G^{up\prime} = G_2 \cup G^{up} \cup G^{up\prime}$). | Insert $G^{up\prime} \setminus G_1$ into $G'_3$, resulting in $G'_3$. Then re-entail. ($G''_3 = G''_2 \setminus G_1 = (G'_2 \cup G^{up\prime}) \setminus G_1 = (G_2 \cup G^{up} \cup G^{up\prime}) \setminus G_1$. | $G'_3$ and $G_1$. | $G^{up\prime}$ and $G_1$ if not cached. | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up\prime\prime} \setminus G_1$ into $G''_3$. Then re-entail. | $G''_3$ and $G_1$. | $G^{up\prime\prime}$ and $G_1$ if not cached. |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Insert $G^{up\prime\prime} \cap G''_2$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G'_2 \cup G^{up\prime}) = (G^{up\prime\prime} \cap G'_2) \cup (G^{up\prime\prime} \cap G^{up\prime}) = G^{up\prime\prime} \cap (G_2 \cup G^{up} \cup G^{up\prime})$ | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up\prime}$. Or $G''_3$, $G_2$, $G^{up}$, and $G^{up\prime}$ | $G^{up\prime\prime}$ and $G''_2$ if not cached. Or $G^{up\prime\prime}$, $G'_2$, and $G^{up\prime}$. Or $G^{up\prime\prime}$, $G_2$, $G^{up}$ and $G^{up\prime}$. |

Table D.27: Chain of Updates 1 for Graph Difference 2.

| $G_3$= | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_2 \setminus G_1$ | Insert into $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \cup G^{up}$) | Insert $G^{up} \setminus G_1$ into $G_3$, resulting in $G'_3$. Then re-entail. [$G'_3 = G'_2 \setminus G_1 = (G_2 \cup G^{up}) \setminus G_1$]. | $G_3$ and $G_1$. | $G^{up}$ and $G_1$ if not cached. | Insert into $G_1$, resulting in $G'_1$. ($G'_1 = G_1 \setminus G^{up\prime}$). | Delete from $G'_3$ the triples in $G^{up\prime}$, resulting in $G'_3$. Then re-entail. ($G''_3 = G'_2 \setminus G'_1 = (G_2 \cup G^{up}) \setminus (G_1 \cup G^{up})$. | $G'_3$ | $G^{up\prime}$ | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up\prime\prime} \setminus G'_1$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \setminus (G_1 \cup G^{up\prime})$ $= (G^{up\prime\prime} \setminus G_1) \setminus G^{up\prime}$ | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up\prime}$. | $G^{up\prime\prime}$ and $G'_1$ if not cached. Or $G^{up\prime\prime}$, $G_1$, and $G^{up\prime}$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Insert $G^{up\prime\prime} \cap G'_2$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G_2 \cup G^{up})$ $= (G^{up\prime\prime} \cap G_2) \cup (G^{up\prime\prime} \cap G^{up})$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$ | $G^{up\prime\prime}$ and $G''_2$ if not cached. Or $G^{up\prime\prime}$, $G_2$, and $G^{up}$. |

Table D.28: Chain of Updates 2 for Graph Difference 2.

| $G_3$= | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_2 \setminus G_1$ | Insert into $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \cup G^{up}$) | Insert $G^{up} \setminus G_1$ into $G_3$, resulting in $G'_3$. Then re-entail. [$G'_3 = G'_2 \setminus G_1 = (G_2 \cup G^{up}) \setminus G_1$]. | $G_3$ and $G_1$. | $G^{up}$ and $G_1$ if not cached. | Delete from $G'_2$, resulting in $G''_2$. ($G''_2 = G'_2 \setminus G^{up\prime} = (G_2 \cup G^{up}) \setminus G^{up\prime}$) ($G^{up\prime} = G_2 \setminus G'_2 = [G_2 \setminus (G_2 \cup G^{up})]$). | Delete from $G'_2$ the triples in $G^{up\prime}$, resulting in $G'_3$. Then re-entail. $G''_3 = G''_2 \setminus G_1 = (G'_2 \setminus G^{up\prime}) \setminus G_1 = [(G_2 \cup G^{up}) \setminus G^{up\prime}) \setminus G_1$ | $G''_3$ | $G^{up\prime}$ | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up\prime\prime} \setminus G_1$ into $G''_3$. Then re-entail. | $G''_3$ and $G_1$. | $G^{up\prime\prime}$ and $G_1$ if not cached. |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Insert $G^{up\prime\prime} \cap G''_2$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G'_2 \setminus G^{up\prime})$ $= (G^{up\prime\prime} \cap G'_2) \setminus (G^{up\prime\prime} \cap G^{up\prime})$ $= G^{up\prime\prime} \cap [(G_2 \cup G^{up}) \setminus G^{up\prime}]$ | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up\prime}$. Or $G''_3$, $G_2$, $G^{up}$, and $G^{up\prime}$ | $G^{up\prime\prime}$ and $G''_2$ if not cached. Or $G^{up\prime\prime}$, $G'_2$, and $G^{up\prime}$. Or $G^{up\prime\prime}$, $G_2$, $G^{up}$ and $G^{up\prime}$. |

Table D.29: Chain of Updates 3 for Graph Difference 2.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_2 \setminus G_1$ | Insert into $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \cup G^{up}$) | Insert $G^{up} \setminus G_1$ into $G_3$, resulting in $G'_3$. Then re-entail. [$G'_3 = G'_2 \setminus G_1 = (G_2 \cup G^{up}) \setminus G_1$]. | $G_3$ and $G_1$. | $G^{up}$ and $G_1$ if not cached. | Delete from $G_1$, resulting in $G'_1$. ($G'_1 = G_1 \setminus G^{up}$) ($G^{up} = G_1 \setminus G'_1$) | Insert $G^{up}{}'' \cap G'_2$ into $G'_3$, resulting in $G'_3$. Then re-entail. ($G''_3 = G'_2 \setminus G'_1 = (G_2 \cup G^{up}) \setminus (G_1 \setminus G^{up})$. | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$ | $G^{up}{}''$ and $G''_2$ if not cached. Or $G^{up}{}''$, $G_2$, and $G^{up}$. | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up}{}'' \setminus G'_1$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up}{}'' \setminus (G_1 \setminus G^{up}{}')$ | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up}{}'$. | $G^{up}{}''$ and $G'_1$ if not cached. Or $G^{up}{}''$, $G_1$, and $G^{up}{}'$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Delete from $G''_3$ the triples in $G^{up}{}''$. Then re-entail. | $G''_3$ | $G^{up}{}''$ |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_3$ the triples in $G^{up}{}''$. Then re-entail. | $G''_3$ | $G^{up}{}''$ |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Insert $G^{up}{}'' \cap G'_2$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up}{}'' \cap (G_2 \cup G^{up}) = (G^{up}{}'' \cap G_2) \cup (G^{up}{}'' \cap G^{up})$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$ | $G^{up}{}''$ and $G''_2$ if not cached. Or $G^{up}{}''$, $G_2$, and $G^{up}$. |

Table D.30: Chain of Updates 4 for Graph Difference 2.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_2 \setminus G_1$ | Delete from $G_2$, resulting in $G'_2$. <br><br> ($G'_2 = G_2 \setminus G^{up}$) | Delete from $G_2$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail. <br><br> [$G'_3 = G'_2 \setminus G_1 = (G_2 \setminus G^{up}) \setminus G_1$]. | $G_3$. | $G^{up}$ | Insert into $G'_2$, resulting in $G''_2$. <br><br> ($G''_2 = G'_2 \cup G^{up\prime} = [(G_2 \setminus G^{up}) \cup G^{up\prime}]$). | Insert $G^{up\prime} \setminus G_1$ into $G'_3$, resulting in $G'_3$. Then re-entail. <br><br> ($G''_3 = G''_2 \setminus G_1 = (G'_2 \cup G^{up\prime}) \setminus G_1 = [(G_2 \setminus G^{up}) \cup G^{up\prime}] \setminus G_1$) | $G'_3$ and $G_1$. | $G^{up\prime}$ and $G_1$ if not cached. | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up\prime\prime} \setminus G_1$ into $G''_3$. Then re-entail. | $G''_3$ and $G_1$. | $G^{up\prime\prime}$ and $G_1$ if not cached. |
| | | | | | | | | | Insert into $G_1$, resulting in $G'_1$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Insert $G^{up\prime\prime} \cap G''_2$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G'_2 \setminus G^{up\prime}) = (G^{up\prime\prime} \cap G'_2) \setminus (G^{up\prime\prime} \cap G^{up\prime}) = G^{up\prime\prime} \cap [(G_2 \setminus G^{up}) \setminus G^{up\prime}]$ | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up\prime}$. Or $G''_3$, $G_2$, $G^{up}$, and $G^{up\prime}$ | $G^{up\prime\prime}$ and $G''_2$ if not cached. Or $G^{up\prime\prime}$, $G'_2$, and $G^{up\prime}$. Or $G^{up\prime\prime}$, $G_2$, $G^{up}$ and $G^{up\prime}$. |

Table D.31: Chain of Updates 5 for Graph Difference 2.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_2 \setminus G_1$ | Delete from $G_2$, resulting in $G'_2$. <br><br> $(G'_2 = G_2 \setminus G^{up})$ | Delete from $G_2$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail. <br><br> $[G'_3 = G'_2 \setminus G_1 = (G_2 \setminus G^{up}) \setminus G_1]$. | $G_3$. | $G^{up}$ | Insert into $G_1$, resulting in $G'_1$. <br><br> $G'_1 = G_1 \cup G^{up\prime}$. | Delete from $G'_3$ the triples in $G^{up\prime}$, resulting in $G'_3$. Then re-entail. <br><br> $(G''_3 = G'_2 \setminus G'_1 = [(G_2 \setminus G^{up}) \setminus (G_1 \cup G^{up})])$. | $G'_3$ | $G^{up\prime}$ | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up\prime\prime} \setminus G'_1$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \setminus (G_1 \cup G^{up\prime}) = (G^{up\prime\prime} \setminus G_1) \setminus G^{up\prime}$ | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up\prime}$. | $G^{up\prime\prime}$ and $G'_1$ if not cached. Or $G^{up\prime\prime}$, $G_1$, and $G^{up\prime}$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Insert $G^{up\prime\prime} \cap G'_2$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G_2 \setminus G^{up}) = (G^{up\prime\prime} \cap G_2) \setminus (G^{up\prime\prime} \cap G^{up})$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$ | $G^{up\prime\prime}$ and $G''_2$ if not cached. Or $G^{up\prime\prime}$, $G_2$, and $G^{up}$. |

Table D.32: Chain of Updates 6 for Graph Difference 2.

| $G_3 =$ | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_2 \setminus G_1$ | Delete from $G_2$, resulting in $G'_2$.<br><br>$(G'_2 = G_2 \setminus G^{up})$ | Delete from $G_2$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail.<br><br>$[G'_3 = G'_2 \setminus G_1 = (G_2 \setminus G^{up}) \setminus G_1]$. | $G_3$. | $G^{up}$ | Delete from $G'_2$, resulting in $G''_2$.<br><br>$(G'_2 = G'_2 \setminus G^{up\prime} = [(G_2 \setminus G^{up}) \setminus G^{up\prime}])$ | Delete from $G'_2$ the triples in $G^{up\prime}$, resulting in $G'_3$.<br>Then re-entail.<br><br>$(G''_3 = G''_2 \setminus G_1 = (G'_2 \setminus G^{up\prime}) \setminus G_1 = [(G_2 \setminus G^{up}) \setminus G^{up\prime}) \setminus G_1)$ | $G''_3$ | $G^{up\prime}$ | Insert into $G''_2$, resulting in $G'''_2$. | Insert $G^{up\prime\prime} \setminus G_1$ into $G''_3$. Then re-entail. | $G''_3$ and $G_1$. | $G^{up\prime\prime}$ and $G_1$ if not cached. |
| | | | | | | | | | Insert into $G_1$, resulting in $G_1$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$.<br>Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G''_2$, resulting in $G'''_2$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$.<br>Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G_1$, resulting in $G'_1$. | Insert $G^{up\prime\prime} \cap G''_2$ into $G''_3$.<br>Then re-entail.<br>This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G'_2 \setminus G^{up\prime})$ $= (G^{up\prime\prime} \cap G'_2) \setminus (G^{up\prime\prime} \cap G^{up\prime})$ $= G^{up\prime\prime} \cap [(G_2 \setminus G^{up\prime}) \setminus G^{up}]$ | $G''_3$ and $G''_2$. Or $G''_3$, $G'_2$, and $G^{up\prime}$.<br>Or $G''_3$, $G_2$, $G^{up}$, and $G^{up\prime}$ | $G^{up\prime\prime}$ and $G''_2$ if not cached.<br>Or $G^{up\prime\prime}$, $G'_2$, and $G^{up\prime}$.<br>Or $G^{up\prime\prime}$, $G_2$, $G^{up}$ and $G^{up\prime}$. |

Table D.33: Chain of Updates 7 for Graph Difference 2.

| $G_3$= | Event 1 | What to do | Old Needed | Fetch | Event 2 | What to do | Old Needed | Fetch | Event 3 | What to do | Old Needed | Fetch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $G_2 \setminus G_1$ | Delete from $G_2$, resulting in $G'_2$. ($G'_2 = G_2 \setminus G^{up}$) | Delete from $G_2$ the triples in $G^{up}$, resulting in $G'_3$. Then re-entail. [$G'_3 = G'_2 \setminus G_1 = (G_2 \setminus G^{up}) \setminus G_1$]. | $G_3$. | $G^{up}$ | Delete from $G_1$, resulting in $G'_1$. $G'_1 = G_1 \setminus G^{up\prime}$. | Insert $G^{up\prime\prime} \cap G'_2$ into $G'_3$, resulting in $G'_3$. Then re-entail. ($G''_3 = G'_2 \setminus G'_1 = (G_2 \setminus G^{up}) \setminus (G_1 \setminus G^{up})$. | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$ | $G^{up\prime\prime}$ and $G''_2$ if not cached. Or $G^{up\prime\prime}$, $G_2$, and $G^{up}$. | Insert into $G'_2$, resulting in $G''_2$. | Insert $G^{up\prime\prime} \setminus G'_1$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \setminus (G_1 \setminus G^{up\prime})$ | $G''_3$ and $G'_1$. Or $G''_3$, $G_1$, and $G^{up\prime}$. | $G^{up\prime\prime}$ and $G'_1$ if not cached. Or $G^{up\prime\prime}$, $G_1$, and $G^{up\prime}$. |
| | | | | | | | | | Insert into $G'_1$, resulting in $G''_1$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G'_2$, resulting in $G''_2$. | Delete from $G''_3$ the triples in $G^{up\prime\prime}$. Then re-entail. | $G''_3$ | $G^{up\prime\prime}$ |
| | | | | | | | | | Delete from $G'_1$, resulting in $G''_1$. | Insert $G^{up\prime\prime} \cap G'_2$ into $G''_3$. Then re-entail. This is equivalent to inserting the triples in $G^{up\prime\prime} \cap (G_2 \setminus G^{up}) = (G^{up\prime\prime} \cap G_2) \setminus (G^{up\prime\prime} \cap G^{up})$ | $G''_3$ and $G'_2$. Or $G''_3$, $G_2$, and $G^{up}$ | $G^{up\prime\prime}$ and $G''_2$ if not cached. Or $G^{up\prime\prime}$, $G_2$, and $G^{up}$. |

Table D.34: Chain of Updates 8 for Graph Difference 2.

# References

Computer Science Bibliography. DBLP. URL: http://dblp.uni-trier.de.

Karim Alami, Radu Ciucanu, and Engelbert Mephu Nguifo. EGG: A framework for generating evolving RDF graphs. In *ISWC Posters & Demonstrations*, 2017.

M. David Allen, Adriane P Chapman, and Barbara Blaustein. Engineering Choices for Open World Provenance. In Bertram Ludäscher and Beth Plale, editors, *International Provenance and Annotation Workshop*, pages 242–253, Allen2014, 2015. URL: https://link.springer.com/chapter/10.1007/978-3-319-16462-5_25.

Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *The Semantic Web - ISWC 2014*, pages 197–212. Springer International Publishing, 2014. URL: https://link.springer.com/chapter/10.1007/978-3-319-11964-9_13.

Renzo Angles, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. The Linked Data Benchmark Council: a Graph and RDF industry benchmarking effort. *ACM SIGMOD Record*, 43(1):27–31, 2014.

Argyro Avgoustaki, Giorgos Flouris, Irini Fundulaki, and Dimitris Plexousakis. Provenance Management for Evolving RDF Datasets. In *International Semantic Web Conference*, pages 575–592. Springer, Cham, 2016. URL: http://link.springer.com/10.1007/978-3-319-34129-3.

Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurelien Lemay, and Nicky Advokaat. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):856–869, April 2017. URL: http://ieeexplore.ieee.org/document/7762945/.

Tim Berners-Lee, Wendy Hall, James Hendler, Kieron O'Hara, Nigel Shadbolt, and Daniel J. Weitzner. A Framework for Web Science. *Foundations and Trends in Web Science*, 1(1):1–130, January 2006.

Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, pages 34–43, May 2001.

Tim Berners-Lee, Vladimir Kolovski, Dan Connolly, James Hendler, and Yoseph Scharf. A Reasoner for the Web, Unpublished, 2003. URL: https://www.w3.org/2000/10/swap/doc/paper/index.pdf.

Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems*, 5(2):1–24, 2009. URL: https://www.igi-global.com/article/berlin-sparql-benchmark/4112.

Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys (CSUR)*, 37(1):1–28, 2005.

Steve Bratt. Semantic Web, and Other Technologies to Watch, W3C Talk, 2007. URL: https://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb.

David Murray Bridgeland and Michael N Huhns. Distributed Truth Maintenance. In *Proceedings of AAAI-90*, pages 72–77, 1990.

Jeen Broekstra and Arjohn Kampman. Inferencing and Truth Maintenance in RDF Schema: Exploring a Naive Practical Approach. In *Workshop on Practical and Scalable Semantic Systems (PSSS)*, 2003.

Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Ian Harrock and James Hendler, editors, *The Semantic Web - ISWC 2002*, pages 54–68. Springer, Berlin, Heidelberg, 2002. URL: http://link.springer.com/10.1007/3-540-48005-6_7.

François Bry and Jakub Kotowski. Reason Maintenance - State of the Art. Technical report, 2008. URL: http://epub.ub.uni-muenchen.de/14900/.

Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems*, 33(4):1–47, November 2008. URL: http://portal.acm.org/citation.cfm?doid=1412331.1412340.

Peter Buneman, Sanjeev Khanna, and Wang-chiew Tan. Data Provenance : Some Basic Issues. In *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, pages 87–93, 2000a.

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *Proceedings of 8th International Conference on Database Theory (ICDT'01)*, pages 316–330, 2001.

Peter Buneman, David Maier, and Jennifer Widom. Where was your data yesterday , and where will it go tomorrow? Data Annotation and Provenance for Scientific Applications. In *Position paper for NSF Workshop on Information and Data Management (IDM '00)*, pages 10–12, Chicago IL, 2000b.

Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *Proceedings of the 14th international conference on World Wide Web - WWW '05*, pages 613–622, New York, New York, USA, 2005. ACM Press.

Artem Chebotko, Eugenio De Hoyos, Carlos Gomez, Andrey Kashlev, Xiang Lian, and Christine Reilly. UTPB: A Benchmark for Scientific Workflow Provenance Storage and Querying Systems. In *2012 IEEE 8th World Congress on Services*, pages 17–24, 2012. URL: http://ieeexplore.ieee.org/document/6274027/.

Paulo Pinheiro da Silva, Deborah L. Mcguinness, Nicholas Del Rio, and Li Ding. Inference Web in Action: Lightweight Use of the Proof Markup Language A Use Case for Provenance. In Amit Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy Finin, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2008*, pages 847–860. Springer, Berlin, Heidelberg, 2008.

Paulo Pinheiro da Silva, Deborah L. McGuinness, and Rob McCool. Knowledge Provenance Infrastructure. *IEEE Data Engineering Bulletin*, 26(4):26–32, 2003.

Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. Provenance for SPARQL queries. In *The Semantic Web - ISWC 2012*. Springer, Berlin, Heidelberg, 2012.

Johan de Kleer. An Assumption-Based TMS. *Artificial Intelligence*, 28(2):127–162, March 1986.

Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel. It's a Streaming World! Reasoning upon Rapidly Changing Information. *IEEE Intelligent Systems*, 24(6):83–89, November 2009. URL: http://ieeexplore.ieee.org/document/5372206/.

Renata Dividino, Simon Schenk, Sergej Sizov, and Steffen Staab. Provenance, trust, explanations - and all that other meta knowledge. *KI*, 23(2):24–30, February 2009.

Jon Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12(3):231–272, November 1979.

Jon Doyle. The Ins and Outs of Reason Maintenance. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 349–351, 1983.

Aldo Franco Dragoni and Paolo Giorgini. Distributed Belief Revision. *Autonomous Agents and Multi-Agent Systems*, 6(2):115–143, 2003.

Aldo Franco Dragoni and Puliti Puliti. Distributed Belief Revision versus Distributed Truth Maintenance. In *Proceedings. Sixth International Conference on Tools with Artificial Intelligence*, pages 499–505, November 1994.

Giorgos Flouris, Irini Fundulaki, Panagiotis Pediaditis, Yannis Theoharis, and Vassilis Christophides. Coloring RDF triples to capture provenance. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009*, pages 196–212. Springer, Berlin, Heidelberg, 2009.

Mark S Fox and Jingwei Huang. Knowledge Provenance: An Approach to Modeling and Maintaining The Evolution and Validity of Knowledge. *EIL Technical Report*, 2003. URL: http://www.eil.toronto.edu/wp-content/uploads/km/papers/fox-kp1.pdf.

Paula Gearon, Alexandre Passant, and Axel Polleres. SPARQL 1.1 Update, W3C Recommendation 21 March, 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-update-20130321/.

Floris Geerts, Grigoris Karvounarakis, Vassilis Christophides, and Irini Fundulaki. Algebraic structures for capturing the provenance of SPARQL queries. *Journal of the ACM*, 63(1), 2016. URL: http://dl.acm.org/citation.cfm?doid=2448496.2448516.

Yolanda Gil, Simon Miles, Khalid Belhajjame, Helena Deus, Daniel Garijo, Graham Klyne, Paolo Missier, Stian Soiland-Reyes, and Stephan Zednik. PROV Model Primer, W3C Working Group Note 30 April, 2013. URL: https://www.w3.org/TR/2013/NOTE-prov-primer-20130430/.

Boris Glavic and Gustavo Alonso. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. *2009 IEEE 25th International Conference on Data Engineering*, pages 174–185, 2009.

Carole Goble. Position Statement: Musings on Provenance, Workflows, and (Semantic Web) Annotations for Bioinformatics. In *Workshop on Data Derivation and Provenance*, Chicago, 2002.

Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update Exchange with Mappings and Provenance. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 675–686, Vienna, Austria, 2007a. URL: http://dl.acm.org/citation.cfm?id=1325851.1325929.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance Semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 31–40, Beijing, China, 2007b. ACM. URL: http://doi.acm.org/10.1145/1265530.1265535.

Paul Groth and Luc Moreau. PROV-Overview: An Overview of the PROV Family of Documents, World Wide Web Consortium Note, 2013. URL: http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/.

Tom R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics*, 3(2-3):158–182, 2005.

Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining Views Incrementally. *ACM SIGMOD Record*, 22(2):157–166, 1993.

Harry Halpin and James Cheney. Dynamic provenance for SPARQL updates. *International Semantic Web Conference*, pages 425–440, 2014.

Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language, W3C Recommendation 21 March, 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

Olaf Hartig. Provenance Information in the Web of Data. In *Proceedings of the Linked Data on the Web LDOW Workshop at WWW*, volume 39, pages 1–9. CEUR-WS, 2009.

Olaf Hartig and Jun Zhao. Publishing and Consuming Provenance Metadata on the Web of Linked Data. In Deborah L McGuinness, James R Michaelis, and LucEditors Moreau, editors, *Provenance and Annotation of Data and Processes*, pages 78–90. Springer, Berlin, Heidelberg, 2010.

Sandro Hawke, Ivan Herman, Bijan Parsia, Axel Polleres, and Andy Seaborne. SPARQL 1.1 Entailment Regimes, W3C Recommendation 21 March, 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/.

Patrick J. Hayes and Peter F. Patel-Schneider. RDF 1.1 Semantics, W3C Recommendation 25 February, 2014. URL: https://www.w3.org/TR/2014/REC-rdf11-mt-20140225/.

Steven R. Haynes, Mark A. Cohen, and Frank E. Ritter. Designs for explaining intelligent agents. *International Journal of Human-Computer Studies*, 67(1):90–110, January 2009.

Robert Ikeda, Semih Salihoglu, and Jennifer Widom. Provenance-based refresh in data-oriented workflows. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, pages 1659–1668, 2011.

Robert Ikeda, Akash Das Sarma, and Jennifer Widom. Logical Provenance in Data-Oriented Workflows? In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 877–888, 2013. URL: http://dx.doi.org/10.1109/ICDE.2013.6544882.

Robert Ikeda and Jennifer Widom. Panda: A System for Provenance and Data. In *Proceedings of the 2nd USENIX Workshop on the Theory and Practice of Provenance TaPP'10*, volume 33, pages 1–8, 2010.

Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. The ORCHESTRA Collaborative Data Sharing System. *ACM SIGMOD Record*, 37(3):26–32, 2008. URL: http://portal.acm.org/citation.cfm?doid=1462571.1462577.

Felix Leif Keppmann, Maria Maleshkova, and Andreas Harth. DLUBM: A Benchmark for Distributed Linked Data Knowledge Base Systems. In Hervé Panetto, Christophe Debruyne, Walid Gaaloul, Mike Papazoglou, Adrian Paschke, Claudio Agostino Ardagna, and Robert Meersman, editors, *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*, pages 427–444, 2017. URL: http://link.springer.com/10.1007/978-3-319-73805-5.

Graham Klyne, Jeremy J. Carroll, and Brian McBride. RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February, 2014. URL: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

Timothy Lebo, Satya Sahoo, Deborah L. McGuinness, Khalid Belhajjame, James Cheney, David Corsar, Daniel Garijo, Stian Soiland-Reyes, Stephan Zednik, and Jun Zhao. PROV-O: The PROV Ontology, W3C Recommendation 30 April, 2013. URL: http://www.w3.org/TR/2013/REC-prov-o-20130430/.

Bernadette Farias Lóscio, C Burle, and Newton Calegari. Data on the Web Best Practices, W3C Recommendation 31 January, 2017. URL: https://www.w3.org/TR/dwbp.

Li Ma, Yang Yang, Zhaoming Qiu, Guotong Xie, Yue Pan, and Shengping Liu. Towards a Complete OWL Ontology Benchmark. In York Sure and John Domingue, editors, *European Semantic Web Conference*, pages 125–139. Springer, Berlin, Heidelberg, 2006. URL: https://link.springer.com/chapter/10.1007%2F11762256_12.

Frank Manola, Eric Miller, and Brian McBride. RDF 1.1 Primer, W3C Working Group Note 24 June 2014, June 2014. URL: http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/.

João P Martins. The Truth, the Whole Truth and Nothing But the Truth: An Indexed Bibliography to the Literature of Truth Maintenance Systems. *AI Magazine*, 11(5): 7–25, 1990.

Deborah L. McGuinness. Ontologies Come of Age. In Dieter Fensel, Jim Hendler, Henry Lieberman, and Wolfgang Wahlster, editors, *Spinning the Sematic Web: Brining the World Wide Web to Its Full Potential*, pages 1–14. MIT Press, 2014.

Deborah L. McGuinness and Paulo Pinheiro Da Silva. Infrastructure for Web Explanations. In *Proceedings of 2nd International Semantic Web Conference - ISWC2003*, pages 113–129, 2003a.

Deborah L. McGuinness and Paulo Pinheiro Da Silva. Registry-Based Support for Information Integration. In *Proceedings of IJCAI-2003 Workshop on Information Integration on the Web (IIWeb-03)*, pages 117–122, 2003b.

Deborah L. McGuinness and Paulo Pinheiro da Silva. Explaining answers from the Semantic Web: the Inference Web approach. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):397–413, October 2004. URL: http://www.sciencedirect.com/science/article/pii/S1570826804000083.

Deborah L. McGuinness, Li Ding, Paulo Pinheiro da Silva, and Cynthia Chang. PML 2: A Modular Explanation Interlingua. In *Proceedings of AAAI*, volume 7, 2007.

Marios Meimaris. EvoGen: A generator for synthetic versioned RDF. In *EDBT/ICDT Workshops*, 2016.

Marios Meimaris. Personal Communication (Email), September 2018.

Marios Meimaris and George Papastefanatos. The EvoGen Benchmark Suite for Evolving RDF Data. In *MEPDaW/LDQ@ ESWC*, pages 20–35, 2016.

Marios Meimaris, George Papastefanatos, and Christos Pateritsas. Towards a Framework for Managing Evolving Information Resources on the Data Web. In *PROFILES@ ESWC*, 2014. URL: http://ceur-ws.org/Vol-1151/paper6.pdf.

Paolo Missier, Jacek Cała, and Eldarina Wijaya. The data, they are a-changin'. In *8th USENIX Workshop on the Theory and Practice of Provenance (TaPP 16)*, Washington, D.C., 2016. USENIX Association. URL: https://www.usenix.org/conference/tapp16/workshop-program/presentation/missier.

Paolo Missier, Tanu Malik, and Jacek Cala. Report on the first international workshop on incremental re-computation: Provenance and beyond. *SIGMOD Rec.*, 47(4):35–38, May 2019. ISSN 0163-5808. URL: http://doi.acm.org/10.1145/3335409.3335418.

Luc Moreau. The Foundations for Provenance on the Web. *Foundations and Trends in Web Science*, 2(2-3):99–241, February 2010. URL: https://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=8187020.

Luc Moreau, Belfrit Victor Batlajery, Trung Dong Huynh, Danius Michaelides, and Heather Packer. A Templating System to Generate Provenance. *IEEE Transactions on Software Engineering*, 44(2):103–121, 2018.

Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh L Simmhan, Eric Stephan, and Jan Van Den Bussche. The Open Provenance Model Core Specification (v1.1). *Future Generation Computer Systems*, pages 1–30, 2009. URL: http://eprints.ecs.soton.ac.uk/18332/.

Luc Moreau, Paul Groth, and Trung Dong Huynh. Provenance: An Introduction to PROV, 2014. URL: http://www.provbook.org/tutorial/provenanceweek2014/prov-tutorial.pptx.

Luc Moreau, Paolo Missier, Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. PROV-DM: The PROV Data Model, W3C Recommendation 30 April, 2013. URL: http://www.w3.org/TR/2013/REC-prov-dm-20130430/.

Leora Morgenstern, Chris Welty, Harold Boley, and Gary Hallmark. RIF Primer (Second Edition), W3C Working Group Note 5 February, 2013. URL: http://www.w3.org/TR/2013/NOTE-rif-primer-20130205/.

Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. DBpedia SPARQL Benchmark – Performance. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Noy, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2011*, pages 454–469. Springer, Berlin, Heidelberg, 2011.

Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 Web Ontology Language Profiles, W3C Recommendation 11 December, 2012a. URL: https://www.w3.org/TR/2012/REC-owl2-profiles-20121211/.

Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Incremental Update of Datalog Materialisation: The Backward / Forward Algorithm. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 1560–1568. AAAI Press, 2015. URL: http://dl.acm.org/citation.cfm?id=2886521.2886537.

Boris Motik, Peter F. Patel-Schneider, Bernardo Cuenca Grau, Ian Horrocks, Bijan Parsia, and Uli Sattler. OWL 2 Web Ontology Language Direct Semantics (Second Edition), W3C Recommendation 11 December, 2012b. URL: https://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/.

Boris Motik, Peter F. Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, and Michael Smith. OWL 2 Web Ontology Language - Structural Specification and Functional-Style Syntax (Second Edition), W3C Recommendation 11 December, 2012c. URL: https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview (Second Edition), W3C Recommendation 11 December, 2012. URL: https://www.w3.org/TR/2012/REC-owl2-overview-20121211/.

Hitzler Pascal, Markus Krotzsch, Pasia Bijan, Patel-Schneider Peter, and Rudolph Sebastian. OWL 2 Web Ontology Language Primer (Second Edition), W3C Recommendation 11 December, 2012. URL: https://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

Beatriz Pérez, Julio Rubio, and Carlos Sáenz-Adán. A systematic review of provenance systems. *Knowledge and Information Systems*, 57(3):495–543, December 2018. URL: https://doi.org/10.1007/s10115-018-1164-3.

João Felipe N. Pimentel, Paolo Missier, Leonardo Murta, and Vanessa Braganholo. Versioned-PROV: A PROV Extension to Support Mutable Data Entities. In Khalid Belhajjame, Ashish Gehani, and Pinar Alper, editors, *Provenance and Annotation of Data and Processes*, pages 87–100. Springer International Publishing, 2018.

Stuart Russell and Peter Norvig. Knowledge Represenation. In *Artifiicial Intelligence A Modern Approach*, chapter 12. Prentice Hall, third edition, 2010.

Satya Sahoo, Roger S. Barga, Jonathan Goldstein, and Amit P Sheth. Provenance Algebra and Materialized View-based Provenance Management. Technical report, Microsoft Research, 2008. URL: http://research.microsoft.com/pubs/76523/tr-2008-170.pdf.

Michael Schmidt, Thomas Hornung, Michael Meier, Christoph Pinkel, and Georg Lausen. SP2bench: A SPARQL performance benchmark. In *ICDE'09. IEEE 25th International Conference on Data Engineering*, pages 222–233, 2009.

Michael Schneider, Jeremy J. Carroll, Ivan Herman, and Peter F. Patel-Schneider. OWL 2 Web Ontology Language RDF-Based Semantics ( Second Edition ), W3C Recommendation 11 December, 2012. URL: https://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/.

Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The Semantic Web Revisited. *IEEE Intelligent Systems*, 21(3):96–101, May 2006. URL: http://ieeexplore.ieee.org/document/1637364/.

Yogesh L Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, September 2005. URL: http://doi.acm.org/10.1145/1084805.1084812.

Steve Speicher, John Arwe, and Ashok Malhotra. Linked Data Platform 1.0, W3C Recommendation 30 April, 2015. URL: https://www.w3.org/TR/ldp/.

Yannis Theoharis, Irini Fundulaki, Grigoris Karvounarakis, and Vassilis Christophides. On provenance of Queries on Semantic Web Data. *IEEE Internet Computing*, 15(1): 31–39, January 2011. URL: http://dx.doi.org/10.1109/MIC.2010.127.

Raphael Volz, Steffen Staab, and Boris Motik. Incremental Maintenance Of Materialized Ontologies. In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pages 707–724. Springer, Berlin, Heidelberg, 2003.

Raphael Volz, Steffen Staab, and Boris Motik. Incrementally maintaining materializations of ontologies stored in logic databases. *Journal on Data Semantics II*, pages 1–34, 2005.

Timo Weithöner, Thorsten Liebig, Marko Luther, and Sebastian Böhm. What's Wrong with OWL Benchmarks? In *Proc. of the Second Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006)*, pages 101–114, 2006. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.139.6934&rep=rep1&type=pdf.

Allison Woodruff and Michael Stonebraker. Supporting Fine-Grained Data Lineage in a Database Visualization Environment. In *Proceedings of the Thirteenth International Conference on Data Engineering*, number January in ICDE '97, pages 91–102, Birmingham, England, 1997. IEEE Computer Society Washington, DC, USA. URL: http://dl.acm.org/citation.cfm?id=645482.653450.

Marcin Wylot, Philippe Cudré-Mauroux, and Paul Groth. TripleProv: Efficient Processing of Lineage Queries in a Native RDF Store. In *Proceedings of the 23rd international conference on World Wide Web - WWW '14*, pages 455–466. ACM, 2014. URL: http://doi.acm.org/10.1145/2566486.2568014.

Ying Zhang, Minh-Duc Pham, Oscar Corcho, and Jean-paul Calbimonte. SRBench : A Streaming RDF / SPARQL Benchmark. In *Iswc 2012*, pages 641–657, 2012.