Towards a Read/Write Web of Linked Data

Luis-Daniel Ibáñez

l'université nantes angers le mans
COMMUNAUTÉ D'UNIVERSITÉS ET ÉTABLISSEMENTS

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Luis Daniel IBÁÑEZ

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 25 février 2015

Towards a Read/Write Web of Linked Data

JURY

Président :	**M. Marc GELGON**, Président, Jury
Rapporteurs :	**M. Johan MONTAGNAT**, Directeur de Recherche, CNRS **M^me^ Esther PACITTI**, Professeur, Université Montpellier 2
Examinateurs :	**M. Marc GELGON**, Professeur, Polytech Nantes **M. Philippe LAMARRE**, Professeur, INSA Lyon **M^me^ Maria-Esther VIDAL**, Professeur, Universidad Simón Bolívar - Venezuela
Directeur de thèse :	**M. Pascal MOLLI**, Professeur, Université de Nantes
Co-encadrante : Co-encadrant :	**M^me^ Hala SKAF-MOLLI**, Maître de Conferences, Université de Nantes **M. Olivier CORBY**, Chargé de Recherche, INRIA Sophia-Antipolis Mediterranée

Acknowledgements

First of all, I would like to thank the *rapporteurs*, Dr. Johan Montagnat and Prof. Dr. Esther Pacitti for accepting the duty of reading my manuscript and write up a report on it. Prof. Dr. Marc Gelgon and Prof. Dr. Philippe Lamarre also accepted to be part of the committee as examiners. They also were part of the *Comité de Suivi de Thèse*, so a big thanks for them for keeping up with me year after year.

On this long road called Ph.D. I had the luck of meeting many wonderful people that made this 3 years trip much more enjoyable. I owe infinite gratitude to my principal advisors at Nantes, Prof. Dr. Pascal Molli and Dr. Hala Skaf-Molli for giving me the opportunity of embarking in this scientific journey. Their guidance and counseling through passionate (and sometimes heated :) discussions were invaluable for the happy ending of this thesis. Members of the GDD team provided a nice working environment and tolerated my bad jokes during team meetings. I thank my co-advisor Dr. Olivier Corby and everyone in the Wimmics team at INRIA Sophia-Antipolis for three fruitful weeks of staying. The need for partial replication support and the link with provenance that led to chapter 9 were conjectured during this stay. Prof. Dr. Maria-Esther Vidal sparkled my interest in Science and the academic world at undergraduate level and believed enough in my abilities to convince me to apply to a Ph.D. and endorse my application to this position. I am also honored to have her as part of the committee, as none of this would have happened without her.

The LINA lab and all its people proved to be very useful and kind along the way. I am very thankful of all fellow Ph.D. students, it was a pleasure to work in such a multi-cultural and friendly environment that made my adaptation to a new country and a new language much easier. Risking incompleteness, I would like to list, in order of appearance, the ones that helped me the most and that also shared off-work moments (and drinks!): Khaled Aslan, Nagham Alhadad, Olivier Finot, Amir Hazem, James Scicluna, Diego Torres, Raziel Carvajal-Gómez, Nicolò Rivetti, Georges Nassopoulos, Alejandro Reyes-Amaro, Marko Budinich and Jorge Calvo.

I make a special note to Gabriela Montoya, whom which I shared dozens of galettes and several blank-nights in the quest for the perfect paper. She also had the patience to listen to me when I was down and to join me in many strange spectacles when no one else dared to go. Contributions in chapter 5 would not have been possible without her participation.

Old time friends scattered around the world and back home were always there despite the distance, and received me arms wide open when we were lucky enough to be geographically in the same place during these 3,5 years. Risking incompleteness again, I would like to mention Carlos Castro, Andrea Alejo, Alejandro Gutiérrez, Flavia Buonanno, Celso Gorrín, Carlos Goncalves, Lessly Jaramillo, Alberto Quirós, Alfredo Garboza, Jorge Garboza and the LetrHado association.

Finalmente, mando un gran abrazo a toda mi familia, en especial a mis padres y hermanos. El cumplimiento de esta meta es en gran parte gracias a ustedes.

1

Introduction

The "Linked Data" initiative [17, 55] has led to the publication and interlinking of billions of pieces of data, transforming the traditional *Web of Documents* into the *Web of Linked Data*. In the Web of Linked Data ideal, data consumers, *i.e.*, users, developers and their applications, make use of links between pieces of data to discover related data stored in remote servers, augmenting their added-value and enriching user experience.

The principles of the Linked Data initiative are the following [55]:

1. Use URIs as names for things. This can be seen as extending the scope of the Web from online resources to encompass any object or concept in the world.
2. Use HTTP URIs, so they can be looked up. This principle enables the dereferencing of these URIs over the HTTP protocol into a description of the identified object or concept.
3. When someone looks up a URI, provide useful information, using the RDF and SPARQL standards. The purpose of advocating a standardized format is to ease the interoperability and contribute to the scalability, as with HTML in the Web of Documents.
4. Include links to other URIs, so that users can discover more data. This principle follows from the idea of hyperlinking documents in the traditional Web. The main difference is that in Linked Data, links are *typed*, *e.g*, two persons can be linked with a hyperlink of type *friend* or

relative.

This thesis focuses on solving the following two problems:

Problem 1. *How to integrate format-heterogeneous data sources to the Web of Linked Data? How to query semantic-heterogeneous data sources in the Web of Linked Data?*

Problem 1 means to take data represented in different formats and query it in RDF and SPARQL and to be able to perform queries on data already in RDF but expressed using different vocabularies or ontologies in an effective way.

Subsequently, we focus on the problem of the *writability* of the Web of Linked Data:

Problem 2. *How to allow Linked Data consumers and publishers to write each other's data and turn the Linked Data into Read/Write?. Which consistency criteria are suitable for a Read/Write Linked Data? How to maintain them respecting the autonomy of the participants and without compromising their availability and scaling in large number of consumers and publishers and in large quantity of data?*

Read/Write support would allow participants to enhance the general quality of the Web of Linked Data in a collaborative way.

1.1 Contributions

We have four major contributions, two for Problem 1 and two for Problem 2. Concerning Problem 1, we define it as a Local-as-View data integration problem. LAV mediators rely on views to define semantic mappings between a uniform interface defined at the mediator level, and local schemas or views that describe the integrated data source. LAV mediators use a query rewriter to translate a query posed to the mediator to a union of queries against the local views. However, the query rewriting problem in LAV mediators has been shown to be NP-Complete [79]: millions of rewritings could be generated when using SPARQL conjunctive queries, consequently, the execution of millions of rewritings may not produce results in a timely manner.

Our contributions to ease the *rewriting explosion* issue are (i) 1. The formulation of the Result-Maximal k-Execution problem (Re-MakE) as the maximization of the query results obtained from the execution of only k rewritings 2. The proposal of a novel rewriting execution strategy called

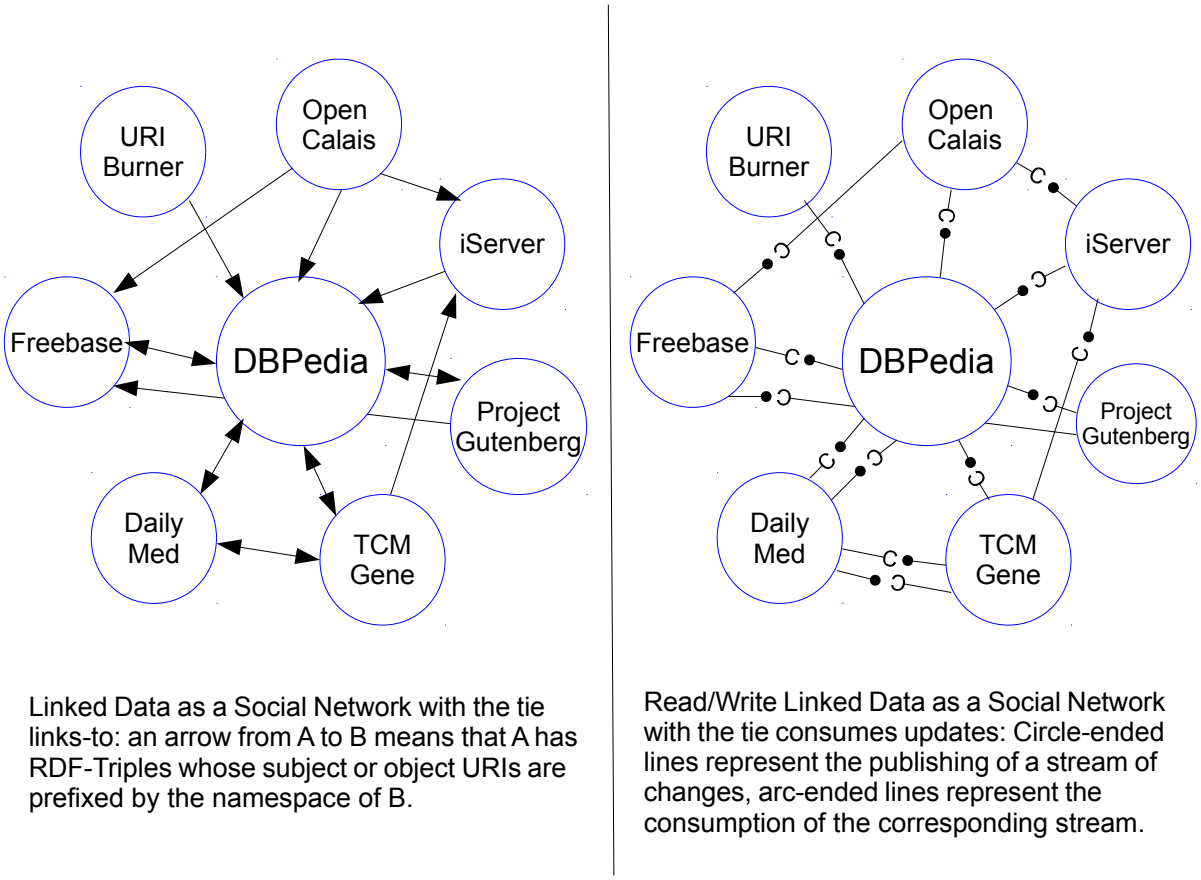


Figure 1.1: Linking Open Data Cloud and Read/Write Linked Data as social networks

Graph-Union (GUN) to solve Re-MakE [84]. Our experimental evaluation demonstrates that GUN outperforms traditional techniques in terms of efficiency and effectiveness. (i)

Concerning Problem 2, we model the Read/Write Linked Data as a social network of update exchange that overlays the current Linking Data Cloud, as illustrated in Figure 1.1. On the left side of the Figure, a fragment of Linking Data Cloud, where relationships among datasets are defined by the presence of links to the URIs prefixed by the domain name of others. On the right side of the Figure, our vision of the Read/Write Linked Data: the relationship is defined by what actors in the social network copy from others. Copied data can be updated, and such updates may, directly or indirectly, be consumed back by the original data producer.

Assuming the model described above, we make two contributions. First, the use of Strong Eventual Consistency (SEC) [110] as consistency criterion. SEC means that, assuming that all updates made at all participants are eventually delivered to all participants, then, when participants stop updating and all updates have been delivered, all participants have an equivalent state. A recent

formalism called Conflict-Free Replicated Data Type (CRDT) [110] provides the means to achieve SEC while respecting the autonomy of participants and scaling to a large number of participants. We designed SU-Set [61], a CRDT for the RDF-Graph type operated with SPARQL 1.1 Update and showed it has a low overhead in time, space and communication.

Second, we develop Fragment Consistency (FC), a criterion stronger than SEC that allows the assertion of guarantees on fragments of data instead of on full states. Each participant copies fragments, *i.e.*, subsets of data of their interest defined as SPARQL CONSTRUCT queries, from other participants, receiving the updates that concern such fragments from the sources, and pushing the updates they have made on their copies to the participants that have copied fragments from them.

FC means that when the system is idle, every fragment copied at a participant T from a participant S equals to the evaluation of the fragment at S after applying the updates locally executed by T Contrary to SEC, with FC there is no assumption that all updates reach all participants, allowing participants full flexibility when establishing their connections to copy data.

We propose a coordination-free protocol to reach FC and analyze its worst case complexity, showing that it is similar to the one of SU-Set except in the case of a very connected network [62]. We argue that the act of copying fragments is driven by social rules, and conducted an empirical experimental study showing that performance is better for synthetically generated social networks than for random ones.

1.2 Outline

This dissertation is comprised of two parts. Part I treats Problem 1 and part II treats Problem 2. Chapter 2 details the RDF data model and the SPARQL 1.1 query and update languages, central concepts common to both parts.

1.2.1 Part 1

- Chapter 3 serves as introduction and gives the outline of Part 1.
- Chapter 4 details the state of the art in Data Integration and Query Execution on the Web of Linked Data.
- Chapter 5 describes the ReMaKe problem and its solution: Graph Union.

1.2.2 Part 2

- Chapter 6 serves as introduction and details the outline of Part 2.
- Chapter 7 details the State of the Art in consistency criteria and algorithms to maintain it in different research communities.
- Chapter 8 presents SU-Set a Conflict-Free Replicated Data type for the Web of Linked Data.
- Chapter 9 presents the Fragment Consistency criterion and Col-Graph, a protocol to maintain it based on annotated RDF-Graphs and updates.

1.2.3 Part 3

Chapter 10 summarizes overall conclusions and outlines the perspectives.

1.3 Publications list

This work led to the following publications, listed in chronological order:

1. Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Olivier Corby. Synchronizing semantic stores with commutative replicated data types. In Alain Mille, Fabien L. Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab, editors, *WWW (Companion Volume)*, pages 1091–1096. ACM, 2012
2. Gabriela Montoya, Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. Gun: An efficient execution strategy for querying the web of data. In *24th International Conference on Database and Expert Systems Applications*, DEXA, pages 180–194, 2013
3. Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Olivier Corby. Live linked data: Synchronizing semantic stores with commutative replicated data types. *International Journal of Metadata, Semantics and Ontologies*, 8(2):119–133, 2013
4. Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Olivier Corby. Col-graph: Towards writable and scalable linked open data. In *13th International Semantic Web Conference*, ISWC, 2014

I also collaborated in two other papers that are not detailed in this dissertation:

– Gabriela Montoya, Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. Semlav: Local-as-view mediation for sparql queries. *T. Large-Scale Data-and Knowledge-Centered Systems*, 13:33–58, 2014. This is a follow-up of the work started in [84].

– Maria-Esther Vidal, Jean Carlo Rivera, Luis-Daniel Ibáñez, Louiqa Raschid, Guillermo Palma, Héctor Rodríguez-Drummond, and Edna Ruckhaus. An authority-flow based ranking approach to discover potential novel associations between linked data. *Semantic Web*, 5(1), 2014. This paper is the culmination of a several years project coordinated by Prof. Dr. María-Esther Vidal at Universidad Simón Bolívar in which I actively participated as an undergraduate student and kept collaborating in parallel with my other works.

2

Background

The Web of Linked Data makes available a large amount of data distributed across many participants. Participants make available their data for others to query. A user that asks a query on the Web of Linked Data is asking a query on a federation of organizations that publish their data following the Linked Data principles. In this chapter we give background on the W3C recommendations to store, query and update Linked Data: the Resource Description Framework (RDF) data model and the SPARQL 1.1 query and update languages. We will build upon these recommendations to solve the problems described in Chapter 1: to integrate *format-heterogeneous data sources* in the Web of Linked Data (Problem 1). For the writability problem (Problem 2), we will enable the use of SPARQL Update expressions on data that is not originally stored by the participant issuing the update, therefore, enabling the collaborative enhancing of the general quality of the Linked Data.

The Resource Description Framework (RDF) [127] is the W3C recommendation to represent information in the Web. The core structure of the data model is the RDF triple, comprised of a *subject*, a *predicate*, and an *object*. An RDF triple represents some relationship, indicated by the predicate, holding between the resources denoted by a subject and an object. An alternative way to represent it is as a directed graph, where the subject and object nodes are connected by a labeled directed edge (the predicate).

SPARQL 1.1 [125] is the query language for RDF recommended by the W3C. We recall some definitions of the semantics of SPARQL and RDF from [96]:

Definition 2.0.1. *The Sets I (IRI Identifiers), B (Blank Nodes), L (Literals) and Υ (Variables) are four infinite and pairwise disjoint sets. We also define $T = I \cup B \cup L$.*

Definition 2.0.2. *An* RDF-Triple *is 3-tuple $(s, p, o) \in (I \cup B) \times I \times T$.*

Definition 2.0.3. *An* RDF-Graph *is a set of RDF-Triples.*

Definition 2.0.4. *A* mapping *μ from Υ to T is a partial function $\mu : \Upsilon \to T$. The domain of μ, $dom(\mu)$, is the subset of Υ where μ is defined.*

Definition 2.0.5. *A* triple pattern *is a tuple $t \in (I \cup \Upsilon \cup L) \times (I \cup \Upsilon) \times (I \cup \Upsilon \cup L)$. A* Basic Graph Pattern (BGP) *is a finite set of triple patterns. Given a triple pattern t, $var(t)$ is the set of variables occurring in t, analogously, given a basic graph pattern B, $var(B) = \cup_{t \in B} var(t)$. Given two basic graph patterns B_1 and B_2, the expression B_1 AND B_2 is a graph pattern.*

Definition 2.0.6. *Given a triple pattern t and a mapping μ such that, $var(t) \subseteq dom(\mu)$, $\mu(t)$ is the triple obtained by replacing the variables in t according to μ. Given a basic graph pattern B and a mapping μ such that $var(B) \subseteq dom(\mu)$, then $\mu(B) = \cup_{t \in B} \mu(t)$.*

Definition 2.0.7. *Two mappings μ_1, μ_2 are compatible (we denote $\mu_1 \shortparallel \mu_2$) iff for all variable $?X \in (dom(\mu_1) \cap dom(\mu_2))$, then $\mu_1(?X) = \mu_2(?X)$. This is equivalent to say that $\mu_1 \cup \mu_2$ is also a mapping.*

Definition 2.0.8. *Let Ω_1, Ω_2 two sets of mappings.*

- *The join between Ω_1 and Ω_2 is defined as: $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \,|\, \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \shortparallel \mu_2\}$*
- *The union between Ω_1 and Ω_2 is defined as: $\Omega_1 \cup \Omega_2 = \{\mu \,|\, \mu \in \Omega_1 \vee \mu \in \Omega_2\}$*
- *The difference between Ω_1 and Ω_2 is defined as: $\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \,|\, (\forall \mu' \in \Omega_2 : \neg \mu \shortparallel \mu')\}$*
- *The left outer join between Ω_1 and Ω_2 is defined as: $\Omega_1 \dashv \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$*

Definition 2.0.9. *Given an RDF-Graph G, the evaluation of a graph pattern P over G is defined recursively as follows:*

1. *if P is a triple pattern t, then $[[t]]_G = \{\mu \,|\, dom(\mu) = var(t) \wedge \mu(t) \in G\}$.*
2. *if P is (P_1 AND P_2), then $[[P]]_G = [[P_1]]_G \bowtie [[P_2]]_G$.*

3. *if* P *is* $(P_1\ OPT\ P_2)$, *then* $[[P]]_G = [[P_1]]_G \dashv [[P_2]]_G$.
4. *if* P *is* $(P_1\ UNION\ P_2)$, *then* $[[P]]_G = [[P_1]]_G \cup [[P_2]]_G$.
5. *given a boolean condition* R, *the filter expression* $(P\ FILTER\ R)$ *is evaluated as* $[[P\ FILTER\ R]]_G = \{\mu \in [[P]]_G \,|\, \mu\ satisfies\ R\}$

In short, each mapping yielded by the evaluation of a BGP gives one way in which selected variables can be bound so that the pattern matches the data in the RDF-Graph. BGPs represent the basic building blocks of SPARQL queries, from them, the following extra features are provided:

1. Optional values: queries that allow information to be added to the solution where the information is available, but do not reject the solution because some part of the query pattern does not match. Optional matching provides this facility: if the optional part does not match, it creates no bindings but does not eliminate the solution.
2. Filtering: a restriction on solutions according on a Boolean condition. For example, restrict the value of a variable to be less than 10.
3. Union: the mappings that match one pattern or the other.

SPARQL queries can have four forms:

1. SELECT: returns variables and their bindings. One can select variables to project out.
2. CONSTRUCT: returns a single RDF graph specified by a graph template. The result is an RDF graph formed by taking each query solution in the solution sequence, substituting for the variables in the graph template, and combining the triples into a single RDF graph by set union.
3. ASK: tests whether or not a query pattern has a solution. No information is returned about the possible query solutions, just whether or not a solution exists.
4. DESCRIBE: returns a single result RDF graph containing RDF data about resources

The RDF data model represents information as graphs consisting of triples with subject, predicate and object. Many RDF data stores hold multiple RDF graphs and record information about each graph, allowing an application to make queries that involve information from more than one graph. This notion is captured in the RDF Dataset definition.

Definition 2.0.10 (Named Graph)**.** *A named graph is a pair* $(name, G)$ *where* $name \in I$ *and* G *is an RDF-Graph.*

Definition 2.0.11 (RDF Dataset). *An RDF Dataset is a set of named graphs which contains at least a pair* $('', G)$, i.e., *a graph associated to an empty name, called the* Default Graph.

The GRAPH keyword can be used to evaluate a query or a portion of it in one or several named graphs. We overload the notation of definition 2.0.9 and write $[[Q]]_G$ to denote the evaluation of any form of SPARQL query on an RDF-Graph or in an RDF-Dataset.

Data providers can make datasets available in two ways: one, in one of the many serialization formats available so consumers can download it, load it and query it; two, providing a server that implements the SPARQL 1.1 Protocol [124] so clients can make http requests on it. Such servers are known as *SPARQL endpoints*, or simply, endpoints.

To allow federated query processing, *i.e.*, to invoke a portion of a SPARQL query against a remote endpoint, SPARQL provides an extension through the SERVICE keyword [123]. For example, the following query finds the names of the persons stored in the dataset at `http://people.example.org` that John knows, as stored in the local RDF-Graph.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT ?name
FROM <http://example.org/myfoaf.rdf>
WHERE
{
  <http://example.org/myfoaf/John> foaf:knows ?person .
  SERVICE <http://people.example.org/sparql> {
    ?person foaf:name ?name . }
}
```

SPARQL 1.1 Update [126] is the W3C recommendation to update RDF-Graphs. It also introduces the concept of *RDF-Graph Store*. An RDF-Graph Store is a mutable RDF Dataset, *i.e.*, one where one can add and delete named graphs. SPARQL 1.1 Update defines two types of operations: Graph Management operations to create and delete RDF-Graphs in a Graph Store; and Graph Update operations to update RDF-Graphs. To illustrate its usage, we use the examples of the Graph Update operations from the recommendation.

– Insert(T): Performs the set union between an RDF-Graph and a set of triples T defined inline by the user. Example:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA {
<http://expl/book1> dc:title 'A new book';
                    dc:creator 'A.N.Other'.
}
```

will insert the triples

```
<http://expl/book1> dc:title 'A new book' .
<http://expl/book1> dc:creator 'A.N.Other' .
```

into the RDF-Graph.

– Delete(T): Performs the set difference between an RDF-Graph and a set of triples T defined inline by the user. Considering an RDF-Graph with the following triples:

```
<http://expl/book2> ns:price 42 .
<http://expl/book2> dc:title 'Copperfield'.
<http://expl/book2> dc:creator 'Edmund Wells'.
```

the operation

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
DELETE DATA
{
<http://example/book2> dc:title 'Copperfield';
        dc:creator 'Edmund Wells' .
}
```

will leave the RDF-Graph with:

```
<http://expl/book2> ns:price 42 .
```

– Delete-Insert(delTemplate,insTemplate,whrPat): This operation takes N steps, executed atomically:

1. Compute the mappings (solutions) corresponding to the evaluation of the $whrPat$ in the current RDF-Graph as a select SPARQL query.
2. For each mapping, delete from the current RDF-Graph the triples of the $delTemplate$, where variables are replaced by their value in the mapping.
3. For each mapping, insert in the current RDF-Graph the triples of the $insTemplate$ where variables are replaced by their value in the mapping.

If delTemplate is null, step 2 will not be executed. If insTemplate is null, step 3 will not be executed. At least one of them must not be null. For example, if to an RDF-Graph containing:

```
<http://expl/pres25> foaf:givenName 'Bill'.
<http://expl/pres25> foaf:familyName 'McKinley'.
<http://expl/pres27> foaf:givenName 'Bill' .
<http://expl/pres27> foaf:familyName 'Taft'.
<http://expl/pres42> foaf:givenName 'Bill'.
<http://expl/pres42> foaf:familyName 'Clinton'.
```

we apply the operation:

```
DELETE { ?person foaf:givenName 'Bill' }
INSERT { ?person foaf:givenName 'William' }
WHERE
  { ?person foaf:givenName 'Bill'
  }
```

the final result is:

```
<http://expl/pres25> foaf:givenName 'William'.
<http://expl/pres25> foaf:familyName 'McKinley'.
<http://expl/pres27> foaf:givenName 'William'.
<http://expl/pres27> foaf:familyName 'Taft'.
<http://expl/pres42> foaf:givenName 'William'.
<http://expl/pres42> foaf:familyName 'Clinton'.
```

– Load(IRI): Loads into the current RDF-Graph all the triples available in the document. Load is similar to an insertion of triples given inline.

– Clear(): Deletes all the triples in the current RDF-Graph, can be considered as the following delete-insert operation:

```
DELETE { ?s ?p ?o }
WHERE { ?s ?p ?o }
```

Note that all SPARQL-Update 1.1 operations can be seen as set operations. If S is the RDF-Graph, we have:

– Insert(T) equals $S \cup T$

– Delete(T) equals $S \setminus T$

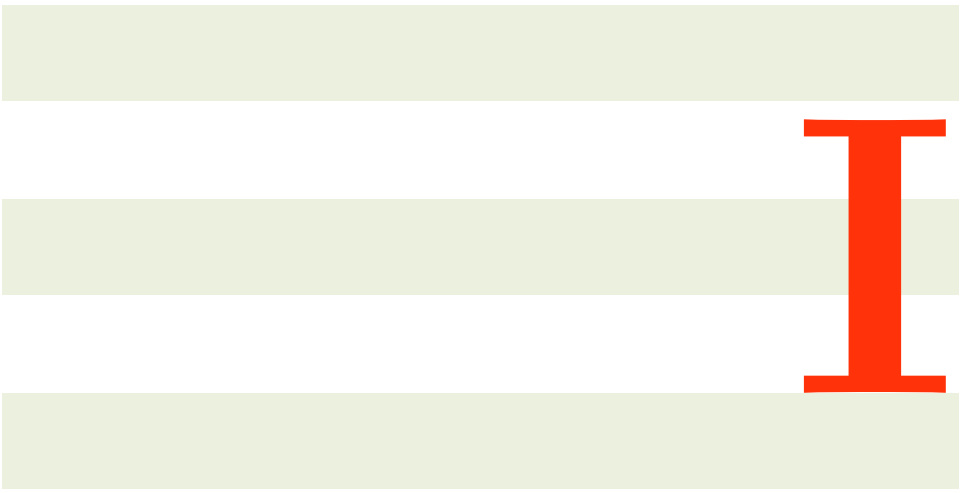
– Delete-Insert(delTemplate,insTemplate,whrPat), if D is the set of triples to delete and I the set of triples to insert, both calculated as explained above, then is equal to $(S \setminus D) \cup I$.

There are two basic Graph Management: *Create* that takes an IRI id and adds a pair $(id, \emptyset)$ to a Graph Store, and *Drop* that takes an IRI id an deletes the pair with name equal to id from the Graph Store. Three keywords provide shortcuts for special combinations of graph management and graph update operations:

– COPY: inserts all data from an input graph into a destination graph. Data from the input graph is not affected, but data from the destination graph, if any, is removed before insertion.

– MOVE: moves all data from an input graph into a destination graph. The input graph is removed after insertion and data from the destination graph, if any, is removed before insertion.

– ADD: inserts all data from an input graph into a destination graph. Data from the input graph is not affected, and initial data from the destination graph, if any, is kept intact.



Efficiently Querying the Web of Linked Data

3

Introduction

In this part of the dissertation we focus on Problem 1: how to integrate several format-heterogeneous and semantic-heterogeneous data sources in the Web of Linked Data so they could be queried with SPARQL.

The advent of the World Wide Web led to the publication of an enormous amount of data. The Linked Data principles state that data should be published following the standards and recommendations (RDF), such that its querying could be simplified. Unfortunately, a large amount of currently published data is not in RDF. Figure 3.1 compares the number of datasets published in RDF with respect to other popular formats on the sum of the datasets available in four of the main Open Data portals: the Data Hub [1], and the official open data portals of the United States [2], United Kingdom [3] and European Union [4]. Only a little more than 3% of the datasets is in RDF. Moreover, there is a great variety in the publication formats. Note that the ZIP format could include any other format inside. Moreover, data available in RDF is not always available through a SPARQL endpoint, making impossible the execution of federated queries.

Therefore, we need a way to *integrate* all these datasets in different formats to query them with

1. http://datahub.io
2. http://data.gov
3. http://data.gov.uk
4. http://open-data.europa.eu

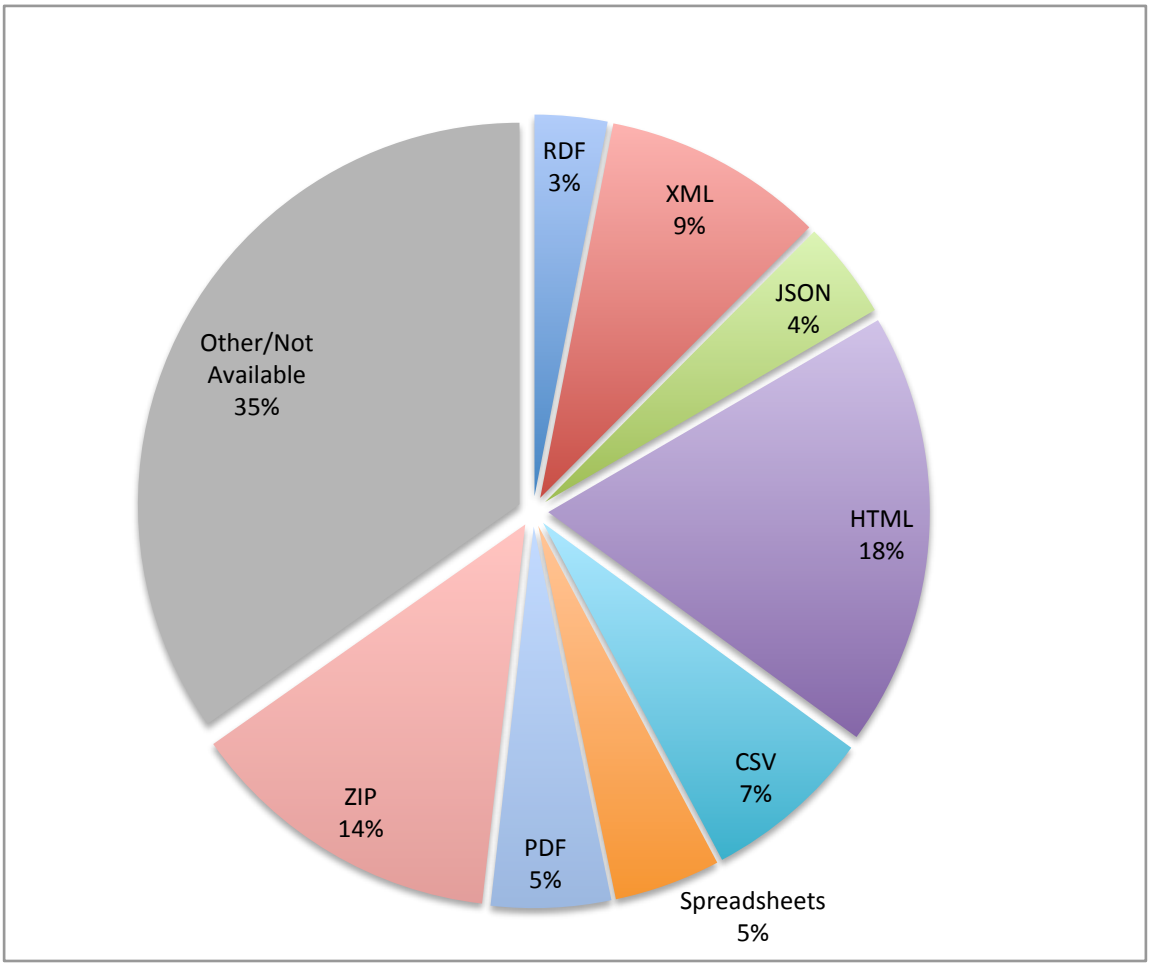


Figure 3.1: Distribution formats of datasets published on the main Open Data portals. Retrieved 25/08/2014.

SPARQL. This integration needs to take in account the dynamics of the Web of Linked Data, *i.e.*, the possible appearance and disappearance of data sources.

We formulate the problem as a Local-as-View (LAV) mediation problem. LAV is a well-known and flexible approach to perform data integration over heterogeneous and autonomous data sources. A LAV mediator relies on views to define semantic mappings between a uniform interface defined at the mediator level, and local schemas or views that describe the integrated data sources. A LAV mediator relies on a query rewriter to translate a mediator query into a union of queris against the local views. LAV is suitable for environments where data sources frequently change, and entities of different types are defined in a single source. Furthermore, LAV can naturally integrate sources from the Web of data [2]. However, LAV mediation has well known severe bottlenecks [50]:

1. The query rewriting problem is NP-Complete for conjunctive queries.

2. The number of rewritings may be exponential.

SPARQL queries exacerbate LAV limitations, even in the case of conjunctions of triple patterns. For example, in relational database systems, a LAV mediator with 140 conjunctive views can generate 10,000 rewritings for a conjunctive query with 8 goals [72]. In contrast, the number of rewritings for a SPARQL query can be in the order of millions. To explain, SPARQL queries are commonly comprised of a large number of triple patterns and some may be bound to *general predicates* of the RDFS or OWL vocabularies, e.g., *rdf:type*, *owl:sameAs* or *rdfs:label*, which are usually published by the majority of the data sources. Additionally, these triple patterns can be grouped into chained connected, star-shaped sub-queries [122]. Finally, a large number of variables can be projected out. All these properties impact on the complexity of the query rewriting problem and lead to the explosion of the number of query rewritings.

If the number of rewritings is very high, its execution may take a significant amount of time, undermining the ability of the system to produce timely and complete results. Our contribution is *Graph Union* (GUN), an efficient rewriting execution strategy that outperforms traditional techniques. GUN profits from the relatively low cost of the RDF-Graph Union operation to construct an aggregation of the data retrieved from the views and execute the original mediator query, thus, obtaining more results at the cost of a higher memory consumption.

3.1 Outline of this part

In chapter 4 we detail the State of the Art on querying the Web of Data and in Data Integration, justifying our selection of the Local-as-View (LAV) paradigm as the most appropriate for the dynamics of the Web of Linked Data.

In chapter 5 we present Graph-Union (GUN), an execution strategy for LAV mediation systems that increases the chance of obtaining results faster from a set of query rewritings in exchange of a higher memory consumption. Our experiments suggest that GUN outperforms traditional execution strategies. Results reported in this part were published in [84]

4

State of the Art

In this chapter, we detail the State of the Art in querying on the Web of Data and in Data Integration.

4.1 Querying the Web of Data

In recent years, several approaches have been proposed for querying the Web of Data. *Link Traversal* [54] conceives the Web of Data as a set of initially unknown data sources that are discovered by following data links at query execution time. The main idea is that, under the assumption that every source includes a set of RDF triples that *describe* a given entity, the local dataset where the query is executed is progressively augmented with the descriptions collected from intermediate results.

SIH-Join [74] is a non-blocking, pushed-based, stream-based, join operator that is able to process both remote and local linked data. SIH-Join is specially tailored to *Exploration Based* approaches where sources are incomplete like Link Traversal.

ANAPSID [4] is an adaptive query processing engine for SPARQL endpoints. ANAPSID stores information about the available endpoints and the ontologies used to describe the data, to decompose queries into sub-queries that can be executed by the selected endpoints. FedX [107] aims the same

goal on federations of endpoints.

All of these approaches assume that queries are expressed in terms of RDF vocabularies used to describe the data in the RDF sources; their main challenge is to effectively select the sources, and efficiently execute the queries on the data retrieved from the selected sources. Data integration is not considered, as all sources are assumed to be available through SPARQL endpoints and following a global ontology.

4.2 Data Integration

Two main paradigms have been proposed to define the data sources in integration systems. The LAV approach is commonly used because it permits the scalability of the system as new data sources become available [118]. Under LAV, the appearance of a new source only causes the addition of a new mapping describing the source in terms of the concepts in the RDF global vocabulary. On the other hand, in the Global-As-View (GAV) approach, entities in the RDF global vocabulary are semantically described using views in terms of the data sources. Thus, the extension or modification of the global vocabulary is an easy task in GAV as it only involves the addition or local modification of few descriptions [118]. Therefore, the LAV approach is best suited for applications with a stable RDF global vocabulary but with changing data sources whereas the GAV approach is best suited for applications with stable data sources and a changing vocabulary.

Given the nature of the Semantic Web, we rely on the LAV approach to describe the data sources in terms of a global and unified RDF vocabulary, and assume that the global vocabulary of concepts used by the mediator is stable while data sources may constantly pop up or disappear from the Web.

Figure 4.1 illustrates a LAV mediation scenario. At the top left of the Figure there is a *global* ontology describes recipes, ingredients and their associations. Using *wrappers*, we can define *LAV-Mappings* or *views* on the heterogeneous data sources. Wrappers take the data expressed in the local ontology or format of the source and translate it to the global ontology. For example, the source *Taaable* is a semantic wiki where data is mostly in text and in relational format and expressed in an ontology defined by *Taaable*'s administrators. The mediator uses a *wrapper* on *Taaable* that translates parts of such data in terms of the global ontology. In the case of Figure 4.1, the wrapper implements a view that returns recipes with *banana* as ingredient.

A client wanting to query the data sources simply formulates a conjunctive query in terms of

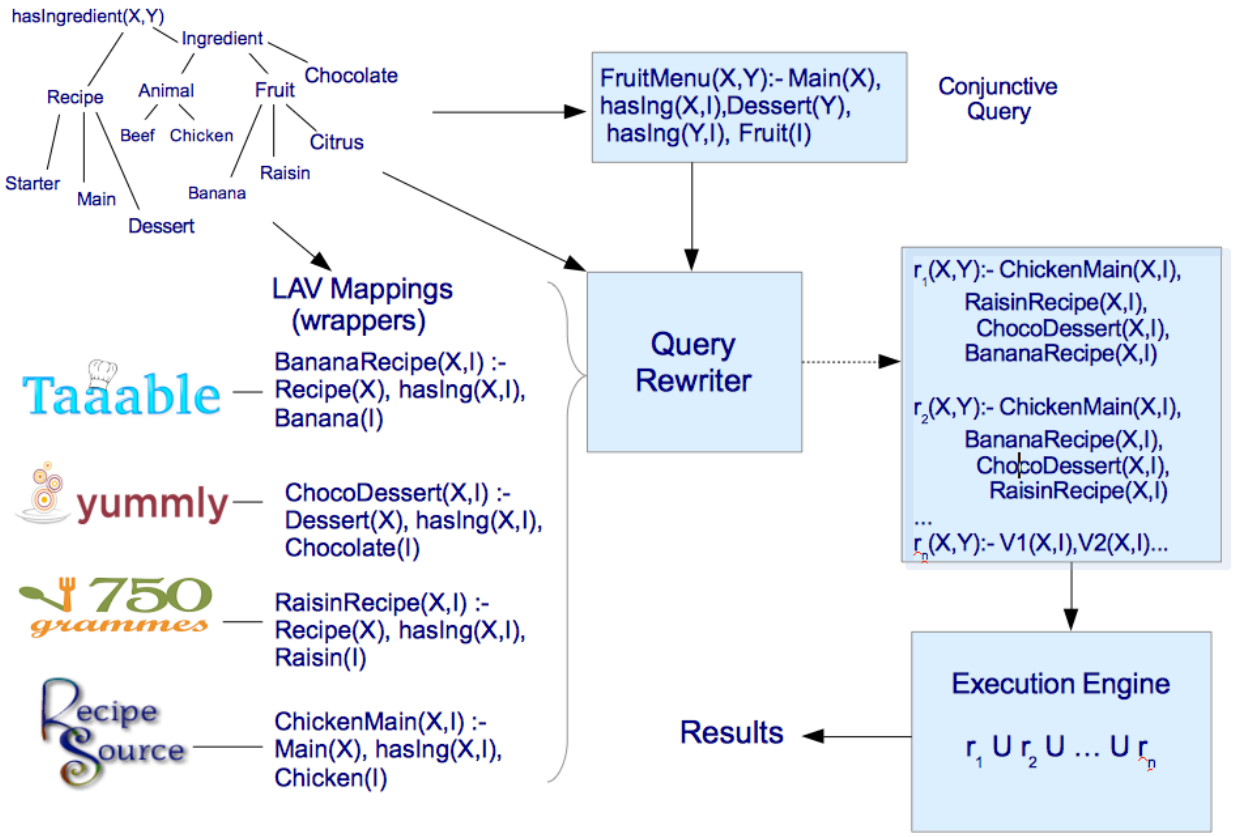


Figure 4.1: Example of Local-As-View Mediation

the global ontology. In the Figure, the client asks for a main dish and a dessert containing a fruit as ingredient. This query is received by a component of the mediator called *Query Rewriter* that generates a set of queries on the LAV mappings *contained* in the user's query called *rewritings*. In Figure 4.1, the rewriter identifies that a recipe in the *RaisinRecipe* view also in the *ChickenMain* view is a main dish containing fruit, and a recipe in *ChocoDessert* also in *RaisinRecipe* is a dessert containing a fruit, therefore, it produces a rewriting equal to the combination of these two joins.

An *Execution Engine* (bottom right of the Figure) receives the rewritings and executes them to collect the results. The union of all these results is the answer returned to the user.

However, LAV-Mediation has a fundamental bottleneck: the query rewriting problem was shown to be NP-complete, and the number of rewritings can be exponential even if mediated queries and local views are conjunctive queries [1, 7]. For example, a LAV mediator with 140 conjunctive views can generate 10,000 rewritings for a conjunctive query with 8 goals [72]. Complexity can be exacerbated by the usage of mediator queries and local views defined as SPARQL conjunctive queries. SPARQL queries are commonly comprised of a large number of triple patterns and many of them are defined on general predicates that can be answered by the majority of the data sources, *i.e.*,

`rdf:type` or `rdfs:seeAlso`. Additionally, these triple patterns can be grouped into chained connected star-shaped sub-queries [122]. Finally, a large number of variables can be projected out. The conjunction of all these properties impacts on the complexity of the query rewriting problem and leads to the explosion of the number of query rewritings.

4.2.1 Query Rewriting

The problem of rewriting a query expressed in terms of a global schema into queries on data sources under local schemas is a relevant problem in integration systems [78], and several approaches have been defined to efficiently enumerate the query rewritings and to scale when a large number of views exists.

The bucket algorithm [78] was the first rewriting algorithm, it is based in three very simple steps: first, from the set of views, *select* the ones that are *relevant* for the query to rewrite; second, to generate candidate rewritings by combining the relevant views and third, check if the rewritings are valid, *i.e.*, if they are contained in the original query. MiniCon [98] optimizes the bucket algorithm by using a more complex algorithm to select relevant views such that all generated candidate rewritings are valid.

MCDSAT [7] casts the query rewriting problem into a propositional theory such that the models of the theory represent the solutions that can be solved using off-the-shelf AI techniques. They show that the casting is more efficient than MiniCon. Finally, GQR [72] uses a graph representation of queries and views to identify common subexpressions, allowing faster discard of invalid rewritings than with previous algorithms.

Concerning the use of rewritings in Semantic Web, [75] propose a solution to identify and combine GAV SPARQL views that rewrite SPARQL queries against a global vocabulary, and Izquierdo et al [65] extends the MCDSAT with preferences to identify the combination of semantic services that rewrite a user request.

A great effort has been made to provide solutions able to produce query rewritings in the least time possible, however, to the best of our knowledge, the problem we tackle, executing the query rewritings against the selected sources, still remains open. Tackling rewriting execution is important, as in many cases the number of generated rewritings is very high due to the exponential complexity on the number of subgoals. For example, when the following query with 7 subgoals on the ontology

of the Berlin Benchmark [19]

```
SELECT ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X8
WHERE {
?X1 rdfs:label ?X2 .
?X1 rdfs:comment ?X3 .
?X1 bsbm:productPropertyTextual1 ?X4 .
?X1 bsbm:productPropertyTextual2 ?X5 .
?X1 bsbm:productPropertyTextual3 ?X6 .
?X1 bsbm:productPropertyNumeric1 ?X7 .
?X1 bsbm:productPropertyNumeric2 ?X8 .
}
```

is input to MCDSAT with 224 sources, $1,127e+10$ rewritings are produced. Too many to be sequentially executed.

4.2.2 Transformation to RDF

In the Semantic Web community, the semi-automatic transformation of format heterogeneous data sources to RDF in order to publish them as Linked Data has been already studied. The D2RQ system [1] allows the read-write access to relational databases as a virtual RDF-Graph. The Karma [2] system [71, 53] eases the transformation of structured data into an RDF Graph or a database following an input ontology. Both systems can be used to implement the wrappers of the LAV mediation.

R2RML [3] isa language for expressing customized mappings from relational databases to RDF datasets. Such mappings provide the ability to view existing relational data in the RDF data model, expressed in a structure and target vocabulary of the mapping author's choice. Some systems have been developed to semi-automatically and interactively create and edit R2RML mappings, for example [91, 108]. Recent research [34] has focused on extending R2RML to make it source-agnostic, in order to use it as part of a Global-as-View integration approach.

4.3 Summary

Our study of the State of the Art can be summarized as follows: Even with the clear benefits that LAV can bring to the Semantic Web, this approach has not been fully adopted, mainly because *it is not realistic to generate or execute the huge number of rewritings produced by query rewriters.* In

1. http://d2rq.org
2. http://www.isi.edu/integration/karma/
3. http://www.w3.org/TR/r2rml/

Chapter 5, we aim at providing an efficient solution to this issue by maximizing the results obtained from k rewritings, where k corresponds to the first k rewritings produced by a LAV query rewriter.

5

Graph-Union (GUN)

In this chapter we describe an efficient strategy for querying the Web of Data under the LAV data integration paradigm. We solve the issue of the evaluation of the large number of query rewritings produced by query rewriters by maximizing the results obtained from the first k.

We devise the Result-Maximal k-Execution Problem (ReMakE) as an extension of the Query-Rewriting-Problem (QRP) as follows: given a subset R_k of size k of a solution R of a QRP for a query Q, the ReMakE problem is to evaluate a set of rewritings R' containing R_k and contained in Q such that R' is result-maximal. Furthermore, we propose the Graph-Union execution strategy (GUN) as a solution to the ReMakE problem.

Unlike traditional techniques, GUN relies on wrappers to populate an RDF graph that is locally managed by the execution engine. This approach takes advantage of the relatively low cost of the RDF-Graph union operation to construct an aggregation of the data retrieved from the views. This approach attempts to execute the original mediator query directly on the graph union and consequently, it may find results hidden to the k first rewritings. For a given set of rewritings, GUN always gathers at least all the answers collected by a traditional engine by executing the rewritings independently. If all relevant views identified by the rewriter are in R_k, GUN guarantees to return the complete answer without further processing of rewritings. Thus, the execution time of GUN

depends on the number of the relevant views that comprise the rewritings in R_k, which is usually considerably lower than the total number of rewritings.

We compare GUN against traditional strategies in an experiment on synthetic data generated with the Berlin SPARQL benchmark tool [19] and views proposed by Castillo-Espinola [24]. We measure execution time and answer completeness for a benchmark of queries. In the experiments, we can observe that GUN retrieves much more results in less time than existing engines. The amount of main memory required to maintain a GUN graph is in general higher than the one required to execute traditional approaches; however, improvements in execution time and results are substantial enough to consider it a good trade-off.

5.1 Preliminaries

Formally, we define an RDF LAV system as follows:

Definition 5.1.1. *An RDF LAV integration system is a tuple* $RLAV = (MS, S, V)$ *where* MS *is a mediator schema or vocabulary,* S *is a set of data sources and* $V = \{v_1, \cdots, v_n\}$ *is a set of views on the sources in* S*. Views are defined as SPARQL queries over* MS*.* D *is a virtual RDF dataset on the mediator schema* MS*.*

A conjunctive query Q over a database or mediator schema D has the form

$$Q(\bar{X}) \text{ :- } P_1(\bar{X}_1), \ldots, P_n(\bar{X}_n)$$

where Q, P_1, ..., P_n are predicates name of some finite arity, and $\bar{X}$, $\bar{X}_1$, ..., $\bar{X}_n$ are tuples of variables. These predicates constitute the global schema. We define the body of the query as

$$body(Q) = \{P_1(\bar{X}_1), \ldots, P_n(\bar{X}_n)\}$$

Any non-empty subset of $body(Q)$ is called a subgoal of Q, singleton subgoals are called atomic subgoals. Predicates in the body stand for relations of D, while the head Q represents the answer relation of the query over D. We consider queries that are *safe*, i.e., $\bar{X} \subseteq \bigcup_{i=1}^{n} \bar{X}_i$, and call $Q(D)$ the result of executing Q over D.

We take three definitions from the work in [72]: (i) a view v as a safe query over D. (ii) We

establish the difference between the available data through the implementation of the view, or the *extension* of v, denoted $ext(v)$, and its evaluation over the database D, $v(D)$. (iii) We assume that $ext(v) \subseteq v(D)$ in order to state two important hypothesis: (i) there may be data belonging to the database that is not available to the extensions. (ii) The extensions never hold data that is not in the database D.

A rewriting of a query Q over a database D with a set of views V is a conjunctive query

$$r(\bar{x}) \text{ :- } v_1(\bar{x_1}), \ldots, v_m(\bar{x_m}).v_i \in V$$

A query rewriting is *contained* in Q, if for all database D and set of views V over D, the result of executing r in V is contained in the result of executing Q on D, *i.e.*, $r(V) \subseteq Q(D)$. The scientific problem is to find a set of rewritings of Q on V such that their evaluation is the closest to the evaluation of Q on D, classical database literature identifies this problem as the *Query Rewriting Problem* or QRP:

Maximally Contained Query Rewriting Problem (QRP). *Given a conjunctive query Q and a set of views $V = \{ v_1, \ldots, v_n \}$ over a dataset D, QRP is to find a set of rewritings R, called the solution of the QRP, such that:*

- *For all extensions of the views in the bodies of all rewritings in R, the union of the results of executing each query rewriting in the views V is contained in the result of executing Q in D, i.e., $\bigcup_{r \in R} r(ext(v_1), \ldots, ext(v_n)) \subseteq Q(D)$*
- *R is maximal, i.e., there is no other set R', such that:*

$$\bigcup_{r \in R} r(ext(v_1), \ldots, ext(v_n)) \subset \bigcup_{r' \in R'} r'(ext(v_1), \ldots, ext(v_n)) \subseteq Q(D)$$

For a set R of rewritings, we define the set of relevant views $\Lambda(R) = \{v \mid v \in body(r) \wedge r \in R\}$ as the set of views in the rewritings in R, and its execution $R(D) = \bigcup_{r \in R} r(D)$. We also call $ext(\Lambda(R))$, the extension of the elements in $\Lambda(R)$.

5.2 Problem Statement: Result-Maximal k-Execution (ReMakE)

The main drawback of existing query rewriting problem solutions for LAV [79, 72, 7, 50] is that the size of the set or rewritings R can be exponential in the number of query subgoals [2, 50]. Therefore, instead of generating and executing a very large number of rewritings, we consider more realistic to generate and execute only k rewritings but maximizing the results we can obtain from them.

Result-Maximal k-Execution Problem (ReMakE) 1. *Given a subset R_k of size k of a solution R of a QRP comprised of a query Q and a set of views V over a database D, ReMakE is to find a set of rewritings R' over the set of relevant views $\Lambda(R_k)$, such that:*

$$\bigcup_{r_k \in R_k} r_k(ext(\Lambda(R_k))) \subseteq \bigcup_{r' \in R'} r'(ext(\Lambda(R_k))) \subseteq Q(D)$$

and that is result-maximal, i.e., that there is no another set R'' such that:

$$\bigcup_{r' \in R'} r'(ext(\Lambda(R_k))) \subset \bigcup_{r'' \in R''} r''(ext(\Lambda(R_k))) \subseteq Q(D)$$

We define this problem over the extensions of the views, as they are the real datasets where the query will be evaluated. It is important to note that the ReMakE problem only uses the query rewritings as an input and they could be obtained using any query rewriter, therefore, it is independent of the approach used to solve QRP. We also highlight that ReMakE is independent of the format of the data inside the extensions of the views, the wrappers would transform any format to the mediator schema.

To illustrate ReMakE, consider the generic set of rewritings in Figure 5.1. Suppose the execution engine has only enough time to execute the first five query rewritings. It is possible that the execution engine misses a rewriting comprised of some combination of views that were gathered for evaluating these five rewritings but that is not in the first five rewritings. ReMakE aims to consider all the rewritings that could be obtained from the already materialized views, hence in Figure 5.1 answers for rewriting r_n would also be obtained.

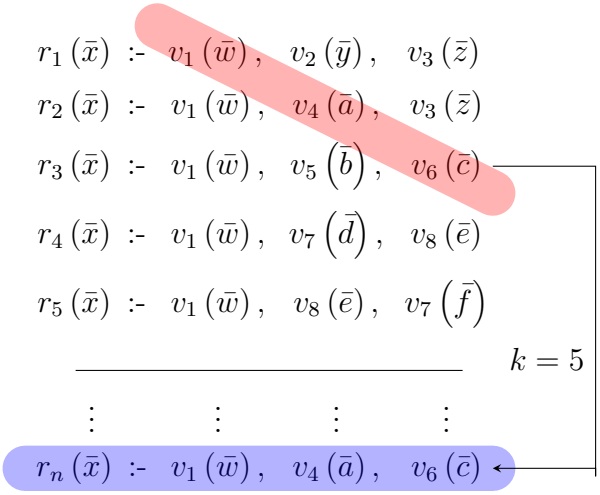


Figure 5.1: Illustration of the Result-Maximal k-Execution problem. Some combinations of views materialized during the execution of a subset of rewritings (R_k) could correspond to valid rewritings that do not belong to R_k.

5.3 Graph-Union (GUN), a solution to the ReMakE problem

The main idea behind our solution is to take advantage of the relative low cost of the RDF-Graph union operation to store the content of all the views in the bodies of the processed rewritings in one RDF-Graph, that we call *Graph-Union* (GUN). We translate data coming from the LAV Mappings to RDF by the means of the wrappers, translate the user query to SPARQL and execute it on the GUN.

A conjunctive query over a general database is analogous to the following SPARQL query:

SELECT ?x
WHERE {
$F(p_1(\bar{x}_1))$.
...
$F(p_n(\bar{x}_n))$.
}

where F is a translation function from predicates to triple patterns. For example, in [10], a natural translation is proposed as following:

1. A unary predicate *Pred* is assigned an URI and represented as *http://assigned-uri.org/Pred rdf:type rdf:Class*.

2. Variables in unary predicates are mapped to variables in SPARQL, creating Basic Graph Patterns, for example, $Pred(X)$ is represented as the BGP *?X rdf:type http://assigned-uri.org/Pred*

3. A binary predicate *BinPred* is assigned an URI and represented as *http://assigned-uri.org/BinPred rdf:type rdf:Property.*

4. Variables in binary predicates are mapped to variables in SPARQL, creating Basic Graph Patterns, for example, $BinPred(X, Y)$ is represented as the BGP *?X http://assigned-uri.org/BinPred ?Y.*

5. N-ary predicates are decomposed in binary predicates before transforming them to RDF.

The definitions of variables, head and body are the same. As the definitions of views and rewritings are based on the definition of query, they remain equivalent, together with the definitions of QRP and ReMakE. We define the evaluation of a rewriting $[[r(x)]]_G$ as:

$$[[r(x)]]_G = [[v_1(\bar{x}_1), \ldots, v_m(\bar{x}_m)]]_G = ([[p_a(\bar{x}_a)]]_{ext(v_1)} \bowtie \ldots \bowtie [[p_z(\bar{x}_z)]]_{ext(v_1)}) \\ \bowtie \ldots \bowtie ([[p_\alpha(\bar{x}_\alpha)]]_{ext(v_m)} \bowtie \ldots \bowtie [[p_\beta(\bar{x}_\beta)]]_{ext(v_m)})$$

where $p_a \ldots p_z \in body(v_1)$ and $p_\alpha \ldots p_\beta \in body(v_m)$. Note that this definition captures the practical implementation of the execution engine, where we materialize each call to a view (or more precisely, to its extension) and then, perform the joins between the sub-results. Traditionally, plans like Left Linear, Right Linear or Bushy Trees [25] are used to evaluate the rewritings over the extension of the views present in each rewriting; but to solve the ReMakE problem, we should ensure that any relevant combinations of obtained views are not missed, even if these combinations are not part of the rewritings in R_k.

Consider the following example, inspired by the well-known Berlin SPARQL Benchmark [19]: assume the mediator has a namespace prefixed *mediator* and the ontology described in table 5.1.

Predicate	Meaning	BGP translation
type(prod,t)	Product *prod* has type *t*	*?prod rdf:type ?t*
product(offer,prod)	Offer *offer* applies for product *prod*	*?offer mediator:product ?prod*
vendor(offer,vend)	Offer *offer* is offered by vendor *vend*	*?offer mediator:Vendor ?vend*
validity(offer,dat)	Offer *offer* has date of validity *dat*	*?offer mediator:valid ?dat*
country(vend,c)	Vendor *vend* is based in country *c*	*?vend mediator:country ?c*
review(rev,prod)	Review *rev* is about product *prod*	*?rev mediator:review ?prod*
rating(rev,rat)	Review *rev* has rating *rat*	*?rev mediator:rating ?rat*
author(rev,reviewer)	Review *rev* was made by *reviewer*	*?rev mediator:reviewer ?reviewer*
price(offer,p)	Offer *offer* has price *p*	*?prod mediator:price ?p*
feature(prod,feat)	Product *prod* has feature *feat*	*?prod mediator:feature ?feat*

Table 5.1: Example ontology and its translation to RDF triple patterns

Wrappers have been implemented on top of heterogeneous data sources, implementing the following views:

- *origin(prod,t,produc,feat) :- type(prod,t), producer(prod,produc), feature(prod,feat).* Type, producer, and features of products.
- *market(prod,offer,p,vend) :- product(offer,prod), vendor(offer,vend), price(offer,p).* Products' offers with their vendors and prices.
- *offers(offer,vend,c,dat) :- vendor(offer,vend), country(vend,c), validity(offer,dat).* Products' offers with their prices and the base countries of the vendors.
- *opinions(prod,rev,rat,reviewer) :- review(rev,prod), rating(rev,rat), author(rev,reviewer).*. Products' reviews with their authors and ratings.

Suppose a user wants to retrieve the products of type t that are sold by vendors based in Germany and their ratings, *i.e.*:

Q(Product, Vendor, Rating) :- type(Product, t), product(Offer, Product), vendor(Offer, Vendor), country(Vendor, de), review(Review, Product), rating(Review, Rating).

Translated to SPARQL following table 5.1:

```
SELECT ?product ?vendor ?rating
WHERE {
    ?product rdf:type ?t .
    ?offer mediator:product ?product .
    ?offer mediator:vendor ?vendor .
    ?vendor mediator:country 'DE' .
    ?review mediator:review ?product .
    ?review mediator:rating ?rating .
}
```

A rewriting of Q in the set of views is:

r(Product,Vendor,Rating) :- origin(Product,t,_0,_1), opinions(Review,Product,_5,Rating), market(Offer, Product,_2,_3), offers(Offer,Vendor,DE,_4).

Assuming for simplicity that each view is reachable through an alias of its name, the SPARQL translation of the rewriting is:

```
SELECT ?product ?vendor ?rating
WHERE {
    SERVICE <origin>{
        ?product rdf:type ?t .
    }
    SERVICE <opinions>{
```

```
        ?review mediator:review ?product .
        ?review mediator:rating ?rating .
    }
    SERVICE <market>{
        ?offer mediator:product ?product .
    }
    SERVICE <offers>{
        ?offer mediator:vendor ?vendor .
        ?vendor mediator:country 'DE' .
    }
}
```

Figure 5.2 shows the execution of this rewriting following a left linear execution plan using the RDF Graphs retrieved from the sources through the wrappers. The execution is comprised of the following steps:

1. The join by products of type *t* is done between the *origin* and *opinions* views. The join yields an intermediate result with *prod1*, *rev1*, and *rat1*.
2. The intermediate result is joined by product with the *market* view. This yields another intermediate result, the same as before plus data concerning offer *off1*.
3. The second intermediate result is joined with offers in the view *offers* whose vendor is based on Germany. Unfortunately, this join does not succeed, as the offer *off1* is not present in the *offers* view, therefore, the final result of the execution is empty.

Note that the rewriting:

r(Product,Vendor,Rating) :- origin(Product,t,_0,_1), opinions(Review,Product,_5,Rating), market(Offer, Product,_2,_3), offers(_5,Vendor,DE,_4).

i.e., the same excepting that the join with the *offers* view is made only by Vendor will produce results. However, recall that we do not have means to rank the rewritings before executing them, as in highly dynamic sources it is very hard to estimate which joins will lead to more results. Rewritings that produce results may be executed only if the execution engine is lucky enough to have them at the top of the list it receives from the rewriter.

Executing rewritings that do not produce any results represents a practical problem: time runs and the user could decide that too much time has passed without results and abort the query. Notice that in the case depicted in Figure 5.2, the mediator had enough data to produce an answer: the *offers* view has the information that *ven1* is based in Germany that could have been joined with information about *off1* in the *market* view.

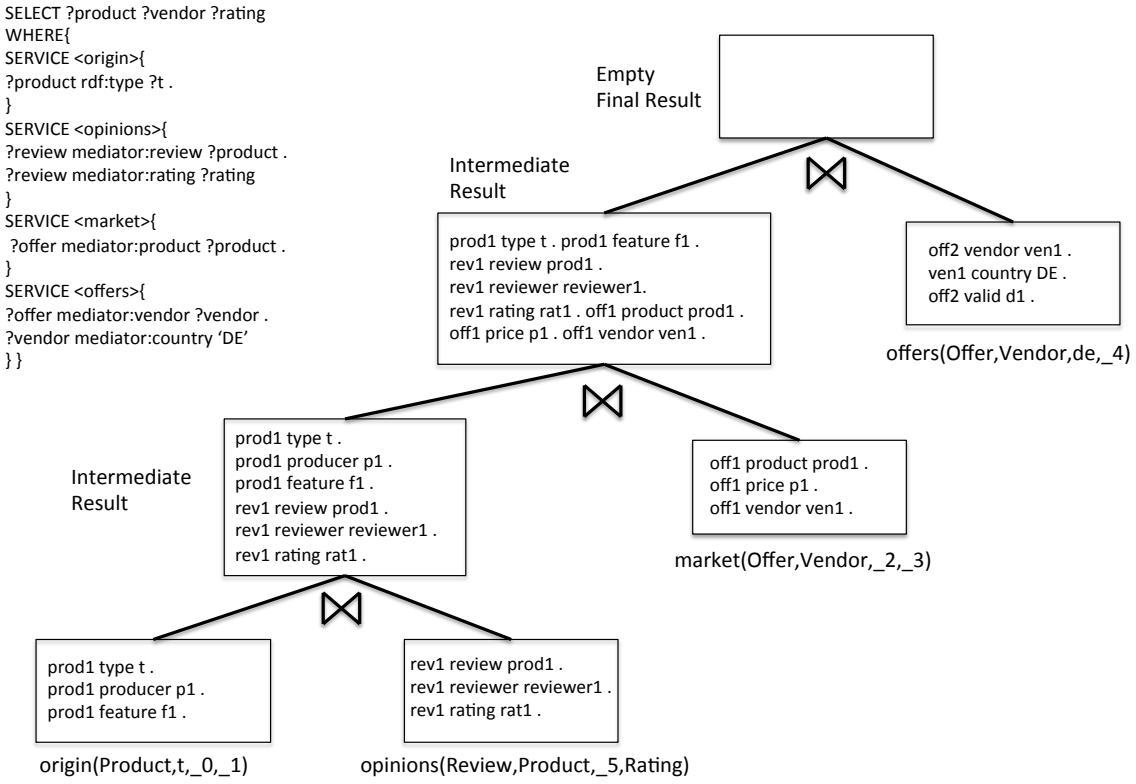


Figure 5.2: Left Linear execution of the rewriting r of query Q. Views *origin*, *opinions*, *market* and *offers* are loaded, but it is not possible to produce any results since the join for *Offer* is empty. Prefixes are omitted to improve legibility.

Our solution to the ReMakE problem takes advantage of these retrieved data to increase the chances of producing results.

Graph Union (GUN) 1. *Given R_k a subset of a set of rewritings R of a query Q over a set of views V, apply Q to the union of the extensions of the views in the bodies of the elements of R_k:*

$$GUN(R_k) = [[Q]]_{\bigcup ext(\Lambda(R_k))}$$

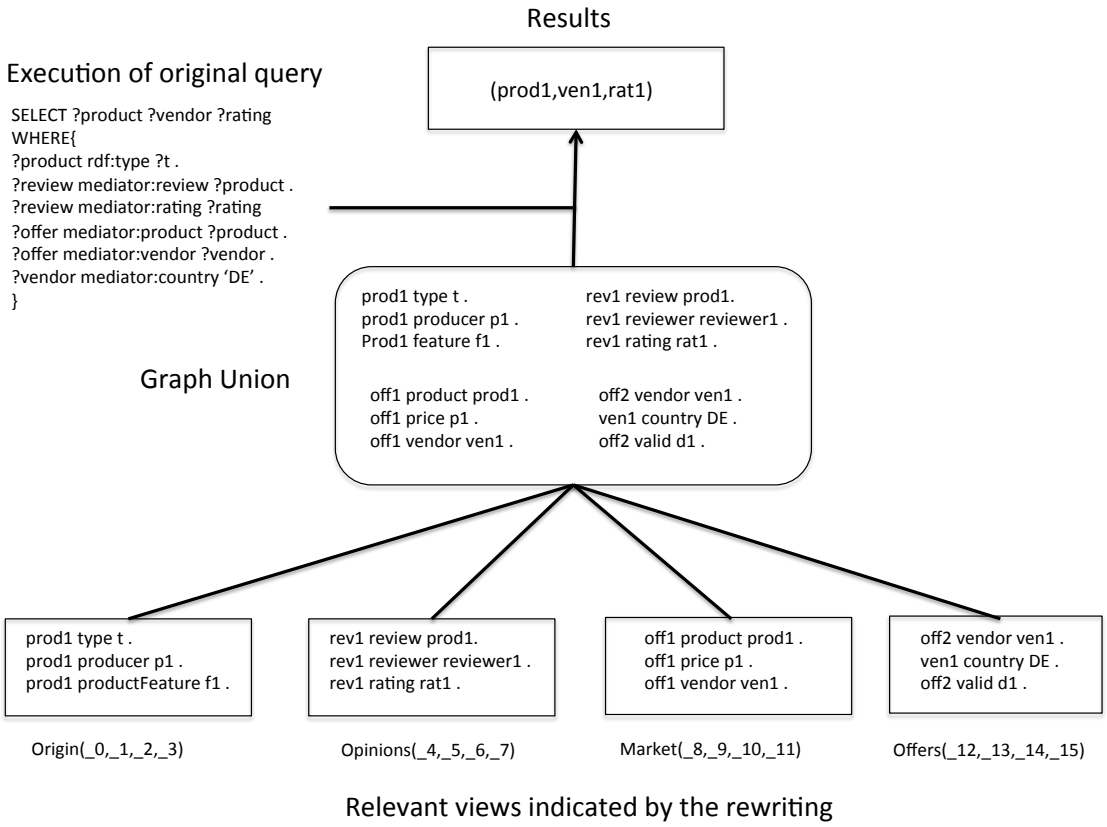


Figure 5.3: GUN execution of the rewriting r of query Q. Results are produced, at the cost of building and querying over an aggregated graph. Prefixes are omitted to improve legibility.

Figure 5.3 shows the execution with GUN of the rewriting of query Q analyzed in Figure 5.2. The execution can be described in two steps:

1. For each view in the rewriting, we retrieve all its answers and store them into an aggregate RDF-Graph.
2. As we are using the Local-As-View approach, data in the views are expressed in terms of the global schema, therefore, we execute the original query Q.

GUN takes advantage of all retrieved data to produce an answer for the user. It can execute joins that are not considered in the rewritings, like the one between *market* and *offers* that allows it to return *off1* as result. GUN is affordable in the context of the Semantic Web thanks to the simplicity of the RDF data model. Implementing the same idea in relational databases would require to create the universal relation, which may have a prohibitive cost.

Theorem 5.3.1. *Graph Union is a solution to the ReMakE problem, i.e.,*

$$\bigcup_{r \in R_k} [[r]]_{ext(\Lambda(R_k))} \subseteq [[Q]]_{\bigcup ext(\Lambda(R_k))} \tag{5.1}$$

$$[[Q]]_{\bigcup ext(\Lambda(R_k))} \subseteq [[Q]]_G \tag{5.2}$$

And is result-maximal.

Proof. As by construction of the views and their extensions $\bigcup_{v \in V} [[v]]_G \subseteq G$, then $\bigcup ext(\Lambda(R_k)) \subseteq G$, making straightforward to see that (5.2) holds. For (5.1) note that the set $\Lambda(R_k)$ can be considered as a set of views over the graph $\bigcup \Lambda(R_k)$, then by the containment property, each member of R_k is contained in Q. As we are applying the original query Q, it is clear that there is no rewriting that can return more results than Q, meaning that GUN is maximal. □

5.3.1 Graph-Union's properties

In this section we formalize some properties of the GUN execution strategy. For the following, assume a query Q and a set of views V on a database D and let V_R be the set of relevant views, R the set of rewritings of Q over V, and R_k a subset of R.

Theorem 5.3.2 (Answer Completeness)**.** *If GUN is executed on an R_k such that all relevant views are present, then, such execution will produce the complete query answer,* i.e.*:*

$$\Lambda(R_k) = \Lambda(R) \Rightarrow GUN(R_k) = Q(D)$$

Proof. By definition of QRP, only the relevant views contribute to the answer, therefore, if GUN's aggregation graph contains all the relevant views, it follows that executing Q on it yields the complete answer. □

Theorem 5.3.3 (Execution time independence of k)**.** *GUN's execution time does not depend on the number of rewritings executed (k).*

Proof. Follows from GUN's definition. □

Theorem 5.3.4 (Memory needed)**.** *GUN solves the ReMaKe problem if all data in the relevant views in R_k fit in memory.*

Proof. Follows from GUN's definition. □

Theorem 5.3.4 is not an equivalence because we can construct a case where we let out a triple of one of the views that does not contribute to the result, *e.g.*, in Figure 5.2 the triple *off1 price p1* in the *market* view is not used. However, in the general case results will be missed if data from relevant views is excluded from the aggregation graph. Formally:

Theorem 5.3.5 (Separation inequality)**.** *Suppose $G = \bigcup ext(\Lambda(R_k))$ does not fit in the available memory, and let H and I be two disjoint proper subsets of G such that $H \cup I = G$ and each of them fits in memory, then:*

$$[[Q]]_H \cup [[Q]]_I \subseteq [[Q]]_G$$

Proof. Clearly, each of $[[Q]]_H$ and $[[Q]]_I$ are subset of $[[Q]]_G$. To see why is not an equality, consider Figure 5.3 if H is equal to the views $Origin$ and $Market$ and I is equal to the views $Opinions$ and $Offers$, both $[[Q]]_H$ and $[[Q]]_I$ are empty. □

Nevertheless, although the separation breaks the maximality property making GUN a partial solution to ReMaKe in the case of not having enough memory, it grants GUN the property of *Non-Blocking*, *i.e.*, GUN can continue working and having a chance of producing results even if memory is not enough for all the relevant views, or if one of the views has a slow response time.

5.4 Experimental Study

The goal of the experimental study is to compare GUN against sequential rewriting execution using the left linear plan used by default by the Jena SPARQL engine, answering the following questions:

1. Does GUN produce the complete answer faster?

Query	Answer Size	# rewritings	# of RV
Q1	3.33E+07	1.61E+09	260
Q2	2.99E+05	6.37E+21	260
Q3	2.03E+05	3.52E+24	280
Q4	1.42E+02	6.02E+03	240
Q5	2.82E+05	1.30E+07	240
Q6	9.84E+04	1.22E+05	100
Q7	1.12E+05	1.15E+12	180
Q8	2.82E+05	4.08E+04	100
Q9	1.41E+04	2.00E+01	20
Q10	1.49E+06	9.76E+05	260
Q11	1.49E+06	3.24E+03	80
Q12	2.99E+05	2.37E+08	260
Q13	2.99E+05	2.41E+04	260
Q14	2.82E+05	8.08E+05	180
Q15	1.41E+05	4.64E+09	280
Q16	1.41E+05	8.36E+04	100
Q17	9.84E+04	2.02E+03	100
Q18	2.82E+05	3.12E+08	240

(a) Query information

Views	Size
V1-V20	147,327
V21-V40	133,992
V41-V60	41,463
V61-V80	22,410
V81-V100	4,515
V101-V120	53,131
V121-V140	32,511
V141-V160	90,873
V161-V180	21,138
V181-V200	9,836
V201-V220	4,515
V221-V240	4,515
V241-V260	67,364
V261-V280	81,313
V281-V300	840,470
Total	1,555,373

(b) Views size

Table 5.2: Queries and their answer size, number of rewritings, number of relevant views (RV) and views size.

2. Does GUN produce more answers?

3. Is GUN's execution faster? How much?

4. Is GUN's memory consumption higher? How much?

5.4.1 Experimental Setup

We used the Berlin SPARQL Benchmark tool (BSBM) [19] to generate a dataset of 5,000,251 triples, using a scale factor of 14,091 products. We used the 18 queries and the 10 views proposed in [24], detailed in Appendix A. These queries are very challenging for a query rewriter since their triple patterns can be grouped into chained connected star-shaped sub-queries, that have between 1 and 13 subgoals, with only distinguished variables.

We defined 5 additional views to cover all the predicates in the queries. From these 15 views, we produced 300 views by horizontally partitioning each original view into 20 parts, such that each part produces 1/20 of the answers given by the original view. Table 5.2a details the following information about queries: (i) The size, in number of triples, of the complete answer. This was computed by loading all the views into the persistent RDF-Store Jena-TDB and executing the queries on it. (ii) The number of rewritings of the query against the views. This was obtained through the model counting feature of the SSDSAT rewriter. (iii) The number of relevant views for the query, also obtained as part of the SSDSAT output. Table 5.2b details the size, in number of triples, of the views.

Notice that the number of rewritings may be very large, making unfeasible their full execution. Furthermore, the time to generate the rewritings is not negligible: in some cases (Q2, Q3 and Q7) SSDSAT could not generate any after 72 hours, so we do not report results about them. We chose to compute 500 rewritings, as this was the best compromise we could find between number of rewritings and generation time, *i.e.*, 500 is the larger number of rewritings (multiple of 50) that could be produced for all queries (but Q2, Q3 and Q7) in less than 15 hours. Additionally, we do not have any statistics about the sources to select the best rewritings or to shrink the set of relevant views. Q1 was also left out of the evaluation because its execution to collect the complete answer did not finish after 48 hours.

Some general predicates like *rdfs:label* are present in most of the views; therefore, the queries that have a triple pattern with these predicates will have a large number of relevant views, but not all of these views will contribute to the answer. The size of a view corresponds to the number of triples that can be accessed through that view. The definition of queries and views can be found in Appendix A.

As the experimental datasets are already in RDF and aligned with the ontology of the mediator, wrappers are implemented as simple file readers. For executing rewritings with a traditional strategy, we loaded each view in a named graph, and let Jena evaluate the rewriting on them. Jena uses Left Linear Plans for this task. For example, for the rewriting described in Figure 5.2, we would create a dataset with four named graphs, one for each of the views: *Origin*, *Opinion*, *Market* and *Offers* and execute the following SPARQL query:

```
SELECT ?product ?vendor ?rating
WHERE {
    GRAPH <Origin> {?product rdf:type t}
    GRAPH <Opinions> {
        ?review mediator:review ?product .
        ?review mediator:rating ?rating .
    }
    GRAPH <Market> {?offer mediator:vendor ?vendor}
    GRAPH <Offers> {
        ?offer mediator:vendor ?vendor .
        ?vendor mediator:country DE .
    }
```

		Q4	Q5	Q6	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
CA	GUN	281	45	>500	381	20	21	29	36	21	>500	20	21	21	56
	Jena	281	>500	>500	383	20	141	119	>500	320	>500	>500	>500	40	>500
GUN's	k=80	0	1	0.0016	0		1	1	1	1	0.0476	1	1	0	1
Effectiveness	k=160	0	1	0.0002	0		0	0	1	1	0.0451	1	1	0	1
	k=320	0	1	0.0018	0		0	0	1	0	0.0406	1	1	0	1
	k=500	0	1	0.0024	0		0	0	1	0	0.0382	1	1	0	1

Table 5.3: Values of k for obtaining the Complete Answers (CA) for queries Q4-Q6, Q8-Q18; using GUN and Jena. GUN's Effectiveness (equation 5.3) for different values of k. Effectiveness for Q9 is not reported here since it only has 20 rewritings.

5.4.2 Experimental Results

To study which strategy produces quicker the complete answer, we executed the GUN and Jena strategies over R_k rewritings and counted the number of rewritings needed to obtain the complete answer or until all k rewritings have been processed. Table 5.3 shows the results, from which we can make the following observations:

1. GUN is able to produce the complete answer for 12 queries whereas Jena is able to do so only for 7 queries.
2. 3 of the 7 queries where Jena found the complete answer (Q10,Q11,Q13) required more than four times more rewritings than GUN, and one (Q17) required almost two times more rewritings.
3. For queries Q9, Q11, Q13 and Q17, GUN produced complete answers because at the reported k, $\Lambda(R_k) = \Lambda(R)$. For the rest of the queries where GUN produced the complete answer, the non aggregated relevant views did not contribute to produce more results.

Details about the ratio of relevant views for each R_k are provided in table 5.4. Note that, except for Q8 and Q13, the ratio grow after $k = 160$ is marginal, or even, inexistent. This means that the rewritings analyzed for these queries are mostly variations in the join variables of the same relevant views.

To compare the number of answers produced by both strategies, we define GUN's *effectiveness* as follows:

$$GUNEffect(R_k) = \frac{|GUN(R_k)| - |\bigcup_{r_k \in R_k} r_k(ext(\Lambda(R_k)))|}{|Q(D)| - |\bigcup_{r_k \in R_k} r_k(ext(\Lambda(R_k)))|} \tag{5.3}$$

Intuitively, GUN is more effective than the regular sequential execution of rewritings if it can find more answers. If $|Q(D)| - |\bigcup_{r_k \in R_k} r_k(ext(\Lambda(R_k)))| = 0$, *i.e.*, the execution of R_k finds all the answers, then, GUN effectiveness is defined to be 0. GUN's effectiveness equals to 1 when it founds all possible answers and the traditional executional finds nothing.

Query	Ratio $\|\Lambda(R_k)\|/\|\Lambda(R)\|$					
	$k = 20$	$k = 40$	$k = 80$	$k = 160$	$k = 320$	$k = 500$
Q4	0.0875	0.0917	0.0958	0.1042	0.1292	0.1583
Q5	0.0833	0.1667	0.3333	0.4167	0.4167	0.4167
Q6	0.2100	0.4100	0.6200	0.6300	0.6600	0.6900
Q8	0.2100	0.2200	0.2400	0.2800	0.3600	0.4000
Q9	1					
Q10	0.0731	0.1500	0.3038	0.3077	0.3077	0.3077
Q11	0.2375	0.4875	0.9875	1	1	1
Q12	0.0769	0.1538	0.3077	0.3077	0.3077	0.3077
Q13	0.0731	0.1346	0.2346	0.4731	0.9231	1
Q14	0.1167	0.2278	0.4500	0.5611	0.5611	0.5611
Q15	0.0714	0.1429	0.2143	0.2143	0.2143	0.2143
Q16	0.1900	0.2000	0.4000	0.4000	0.4000	0.4000
Q17	0.1900	0.2000	0.4100	1	1	1
Q18	0.0833	0.1667	0.3333	0.4167	0.4167	0.4167

Table 5.4: Ratio of the number of relevant views in R_k over the number of relevant views in R

We executed the GUN and Jena strategies on R_k with $k \in \{80, 160, 320, 500\}$ and counted the number of answers and computed the effectiveness. Table 5.3 shows that GUN has effectiveness 1 for $k = 80$ for half of the queries, moreover, in 5 of these 7 queries, the maximum effectiveness remains even after Jena executes 500 rewritings. In 4 cases, GUN is not effective because Jena already found the complete answer for this value of k. Finally, in Q6 and Q14, GUN found more results than Jena. Effectiveness values are not monotonic, since they can increase when considering a rewriting that contains a view that contributes to produce results in GUN and not in Jena. However, they can decrease after executing a rewriting that does not add new views to GUN, but produces results for Jena.

Regarding the execution time, we want to: *1*) compare GUN's execution time against Jena's, and *2*) verify experimentally property 5.3.3. We measured total execution time as the sum of the following partial times:

- For both strategies, the time to generate R_k rewritings.
- For both strategies, the time taken by the wrappers to return the requested data.
- For GUN, the time to create the aggregation graph.
- For Jena, the time to load the named graphs to execute the query. (check this, I forgot it)
- For GUN, the time to execute the query on the aggregation graph.
- For Jena, the time to execute all rewritings in R_k.

Query		Execution Time (s)			
		K=80	K=160	K=320	K=500
Q4	GUN	39	39	63	73
	Jena	167	293	48,943	49,721
Q5	GUN	377	400	400	400
	Jena	1,155	2,302	3,848	5,935
Q6	GUN	336	337	338	339
	Jena	398	798	1,610	2,516
Q8	GUN	41	47	58	64
	Jena	190	377	751	1,278
Q10	GUN	132	132	132	132
	Jena	2,214	5,941	119,137	251,641
Q11	GUN	121	121	121	121
	Jena	1,906	3,707	9,985	16,939
Q12	GUN	28	28	28	28
	Jena	79	146	288	475
Q13	GUN	71	203	478	522
	Jena	146	352	734	2,034
Q14	GUN	328	395	395	395
	Jena	439	842	1,657	2,485
Q15	GUN	358	358	358	358
	Jena	1,207	3,000	5,812	9,160
Q16	GUN	35	35	35	35
	Jena	119	283	596	972
Q17	GUN	69	345	345	345
	Jena	168	965	2,450	4,029
Q18	GUN	324	414	415	415
	Jena	1,149	2,413	4,355	6,808

(a) Execution Time for GUN and Jena

Query		ET and # of RV			
		k=80	k=160	k=320	k=500
Q4	GUN	39	39	63	73
	# RV	23	25	31	38
Q5	GUN	377	400	400	400
	# RV	80	100	100	100
Q6	GUN	336	337	338	339
	# RV	62	63	66	69
Q8	GUN	41	47	58	64
	# RV	24	28	36	40
Q10	GUN	132	132	132	132
	# RV	79	80	80	80
Q11	GUN	121	121	121	121
	# RV	79	80	80	80
Q12	GUN	28	28	28	28
	# RV	80	80	80	80
Q13	GUN	71	203	478	522
	# RV	61	123	240	260
Q14	GUN	328	395	395	395
	# RV	81	101	101	101
Q15	GUN	358	358	358	358
	# RV	60	60	60	60
Q16	GUN	35	35	35	35
	# RV	40	40	40	40
Q17	GUN	69	345	345	345
	# RV	41	100	100	100
Q18	GUN	324	414	415	415
	# RV	80	100	100	100

(b) Execution Time and Number of Relevant Views for GUN

Table 5.5: Execution Time (ET) for GUN and Jena. Impact of Number of Relevant Views (RV) over Execution Time in GUN.

Table 5.5a shows the total execution time in seconds for GUN and Jena. Partial execution times are detailed in Tables 5.6 and 5.7. For all queries, GUN has better execution time, and for all but Q6 with $k = 80$, is more than twice faster. When $k = 500$, the difference is dramatic, varying from almost 4 times faster (Q13) to 680 times faster (Q4).

Table 5.5b shows total execution time and number of loaded views for GUN. Execution time grows linearly in $|\Lambda(R_k)|$, this is particularly visible in Q4 and Q13.

If we compare the times detailed in Section 5.3.1, we notice that the wrapper execution time dominates. GUN loads views into the aggregated graph only once, whereas Jena reloads them for each executed rewriting. Note that if we try to cache the views in Jena to avoid reloading, it would consume more memory and could consume even more memory than GUN if the views have overlapped information, as it is the case in our setup.

Finally, we compare the memory usage of both strategies. For GUN, we count the number of triples of the aggregated graph. For Jena, we report an upper bound, that is, the maximum number of triples loaded for executing a rewriting in R_k. Table 5.8 summarizes the results. Neither GUN nor Jena consumes all the available memory (8GB). GUN needs to load more triples than Jena, varying

Query	# Relevant Views	Execution Time						# Answers		Maximal Graph Size	
		WT		GCT		PET					
		GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena
Q4	21	30	44	7	0	0	9	0	0	1,169,684	148,739
Q5	20	86	110	18	1	2	181	197,274	13,471	1,567,231	907,905
Q6	21	90	86	18	0	0	8	4,784	4,784	1,427,018	850,337
Q8	21	28	41	8	0	0	8	14,091	14,091	1,437,413	148,717
Q9	20	3	3	0	0	0	0	14,091	14,091	84,835	4,517
Q10	19	45	87	7	0	6	777	1,487,995	1,415,733	382,145	294,678
Q11	19	41	93	7	0	5	638	1,491,990	1,420,411	382,944	294,701
Q12	20	13	25	3	0	1	9	253,935	14,937	290,858	83,260
Q13	19	9	25	2	0	1	13	283,810	14,938	294,312	44,854
Q14	21	80	131	16	0	0	2	13,291	717	1,491,658	912,378
Q15	20	13	34	3	0	2	5	140,910	7,046	443,882	97,453
Q16	19	14	32	3	1	3	3	133,865	7,045	401,595	42,273
Q17	19	6	10	1	0	0	2	93,433	4,918	186,866	19,672
Q18	20	62	97	14	0	3	134	270,401	14,091	1,564,496	907,873

(a) k=20

Query	# Relevant Views	Execution Time						# Answers		Maximal Graph Size	
		WT		GCT		PET					
		GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena
Q4	22	31	88	8	0	0	18	0	0	1,197,866	148,739
Q5	40	164	213	30	1	2	371	267,729	13,901	1,686,212	907,905
Q6	41	180	173	31	0	0	17	4,909	4,909	1,521,434	850,376
Q8	22	29	80	9	0	0	16	28,182	28,182	1,451,504	148,725
Q10	39	85	182	13	0	6	1,570	1,493,735	1,421,473	383,293	294,678
Q11	39	78	190	13	0	5	1,318	1,493,735	1,422,152	383,293	294,701
Q12	40	17	40	3	0	1	11	298,747	14,937	373,726	83,260
Q13	35	33	67	9	1	1	20	298,747	14,938	1457,918	862,831
Q14	41	154	246	28	0	0	3	14,037	717	1,690,296	912,395
Q15	40	178	235	29	0	2	426	140,910	7,046	1,559,682	905,423
Q16	20	14	57	3	1	3	5	140,910	7,045	422,730	42,273
Q17	20	6	14	1	0	0	3	98,350	98,350	196,700	19,672
Q18	40	161	261	30	1	3	438	280,465	14,091	1,784,756	907,908

(b) k=40

Query	# Relevant Views	Execution Time						# Answers		Maximal Graph Size	
		WT		GCT		PET					
		GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena
Q4	23	31	147	8	0	0	20	0	0	1,201,671	148,739
Q5	80	319	414	56	1	2	740	281,820	14,051	1,993,617	907,905
Q6	62	289	363	47	0	0	35	9,836	9,691	1,578,294	850,376
Q8	24	32	159	9	0	0	31	56,364	56,364	1,479,686	148,725
Q10	79	109	316	17	1	6	1,897	1,493,735	1,421,473	422,269	294,678
Q11	79	99	309	17	0	5	1,597	1,493,735	1,422,152	422,268	294,701
Q12	80	23	67	4	0	1	12	298,747	14,937	439,946	83,260
Q13	61	57	112	13	1	1	33	298,747	14,938	1,713,056	862,917
Q14	81	278	432	50	1	0	6	14,086	717	2,095,418	912,422
Q15	60	309	426	47	0	2	781	140,910	7,046	1,568,458	905,450
Q16	40	26	107	6	1	3	11	140,910	7,045	584,792	53,678
Q17	41	55	76	14	0	0	92	98,350	98,350	1,496,262	850,331
Q18	80	271	438	50	1	3	710	281,820	14,091	2,175,448	907,916

(c) k=80

Table 5.6: Execution time, answer size and maximal graph size, for execution of queries Q4-Q6 and Q8-Q18, with GUN and Jena. For k in {20, 40, 80}. The execution time is discriminated as Wrapper Execution Time (WT), Graph Creation Time (GCT) and Plan Execution Time (PET). The number of answers corresponds to the number of mappings obtained. Maximal Graph Size corresponds to the maximal number of triples required to store at any time.

Query	# Relevant Views	Execution Time						# Answers		Maximal Graph Size	
		WT		GCT		PET					
		GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena
Q4	25	31	271	8	0	0	22	0	0	1,208,714	148,739
Q5	100	338	816	60	2	2	1,484	281,820	28,101	2,275,437	907,923
Q6	63	289	728	48	0	0	70	14,754	14,741	1,583,212	850,376
Q8	28	36	315	10	0	1	62	112,728	112,728	1,536,050	148,745
Q10	80	109	704	17	1	6	5,236	1,493,735	1,493,735	422,269	294,678
Q11	80	99	602	17	0	5	3,105	1,493,735	1,493,735	422,269	294,701
Q12	80	23	124	4	0	1	22	298,747	29,875	439,946	83,260
Q13	123	169	292	33	1	1	59	298,747	14,938	2,277,638	862,962
Q14	101	334	829	61	1	0	12	14,086	1,438	2,279,248	926,356
Q15	60	309	985	47	0	2	2,015	140,910	7,046	1,568,458	905,529
Q16	40	26	260	6	1	3	22	140,910	7,045	584,792	63,411
Q17	100	295	366	50	0	0	599	98,350	98,350	1,807,718	850,376
Q18	100	347	887	64	1	3	1,525	281,820	28,182	2,275,437	921,840

(a) k=160

Query	# Relevant Views	Execution Time						# Answers		Maximal Graph Size	
		WT		GCT		PET					
		GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena
Q4	31	50	880	13	0	0	48,063	142	142	1,753,969	907,775
Q5	100	338	1,432	60	2	2	2,414	281,820	55,634	2,275,437	907,923
Q6	66	290	1,470	48	0	0	140	29,506	29,379	1,597,964	850,376
Q8	36	44	628	12	0	2	123	225,456	225,456	1,648,778	148,745
Q10	80	109	1,690	17	1	6	117,446	1,493,735	1,493,735	422,269	442,052
Q11	80	99	1,377	17	0	5	8,608	1,493,735	1,493,735	422,269	294,748
Q12	80	23	245	4	0	1	43	298,747	59,750	439,946	83,260
Q13	240	400	623	77	1	1	110	298,747	298,747	2,923,233	862,962
Q14	101	334	1,622	61	1	0	34	14,086	2,751	2,279,248	935,825
Q15	60	309	1,890	47	1	2	3,921	140,910	7,046	1,568,458	905,529
Q16	40	26	557	6	1	3	38	140,910	7,045	584,792	63,411
Q17	100	295	913	50	0	0	1,537	98,350	98,350	1,807,718	850,376
Q18	100	347	1,623	64	1	4	2,731	281,820	56,364	2,275,437	921,840

(b) k=320

Query	# Relevant Views	Execution Time						# Answers		Maximal Graph Size	
		WT		GCT		PET					
		GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena
Q4	38	58	1,268	15	0	0	48,453	142	142	1,878,666	907,775
Q5	100	338	2,210	60	2	2	3,723	281,820	70,268	2,275,437	907,923
Q6	69	291	2,298	48	0	0	218	44,258	44,128	1,612,716	850,376
Q8	40	48	1,078	13	0	3	200	281,820	281,820	1,705,142	230,045
Q10	80	109	2,757	17	1	6	249,212	1,493,735	1,493,735	422,269	442,052
Q11	80	99	2,222	17	0	5	14,717	1,493,735	1,493,735	422,269	294,748
Q12	80	23	400	4	1	1	74	298,747	104,562	439,946	83,276
Q13	260	438	935	83	1	1	1,098	298,747	298,747	2,923,233	862,962
Q14	101	334	2,438	61	1	0	46	14,086	3,438	2,279,248	935,825
Q15	60	309	2,948	47	1	2	6,211	140,910	7,046	1,568,458	905,529
Q16	40	26	912	6	1	3	59	140,910	7,045	584,792	74,802
Q17	100	295	1,491	50	0	0	2,538	98,350	98,350	1,807,718	850,376
Q18	100	347	2,515	64	1	4	4,292	281,820	70,455	2,275,437	921,859

(c) k=500

Table 5.7: Execution time, answer size and maximal graph size, for execution of queries Q4-Q6 and Q8-Q18, with GUN and Jena. For k in {160, 320, 500}. The execution time is discriminated as Wrapper Execution Time (WT), Graph Creation Time (GCT) and Plan Execution Time (PET). The number of answers corresponds to the number of mappings obtained. Maximal Graph Size corresponds to the maximal number of triples required to store at any time.

Query	Maximal Graph Size k=80		Maximal Graph Size k=160		Maximal Graph Size k=320		Maximal Graph Size k=500	
	GUN	Jena	GUN	Jena	GUN	Jena	GUN	Jena
Q4	1,201,671	148,739	1,208,714	148,739	1,753,969	907,775	1,878,666	907,775
Q5	1,993,617	907,905	2,275,437	907,923	2,275,437	907,923	2,275,437	907,923
Q6	1,578,294	850,376	1,583,212	850,376	1,597,964	850,376	1,612,716	850,376
Q8	1,479,686	148,725	1,536,050	148,745	1,648,778	148,745	1,705,142	230,045
Q10	422,269	294,678	422,269	294,678	422,269	442,052	422,269	442,052
Q11	422,268	294,701	422,269	294,701	422,269	294,748	422,269	294,748
Q12	439,946	83,260	439,946	83,260	439,946	83,260	439,946	83,276
Q13	1,713,056	862,917	2,277,638	862,962	2,923,233	862,962	2,923,233	862,962
Q14	2,095,418	912,422	2,279,248	926,356	2,279,248	935,825	2,279,248	935,825
Q15	1,568,458	905,450	1,568,458	905,529	1,568,458	905,529	1,568,458	905,529
Q16	584,792	53,678	584,792	63,411	584,792	63,411	584,792	74,802
Q17	1,496,262	850,331	1,807,718	850,376	1,807,718	850,376	1,807,718	850,376
Q18	2,175,448	907,916	2,275,437	921,840	2,275,437	921,840	2,275,437	921,859

Table 5.8: Maximum number of triples loaded by a rewriting in R_k in Jena. The number of triples of the aggregated graph of GUN.

from less than twice to 12 times more, in all cases except for Q10 with $k \geq 320$. GUN's aggregation is in general larger than the sum of the named graphs of the most memory-consuming rewriting in R_k.

In summary, GUN finds the same or more answers than traditional executions, with better execution time at the cost of higher memory consumption. In our experimentation GUN never exhausts the available memory in spite of the challenging setup. This makes it a very appealing solution for the ReMakE problem.

5.5 Conclusion

The execution of complex queries on multiple data sources in the Web of Data raises the issue of format and semantic heterogeneity. Local-As-View mediation is one of the main approaches to solve this data integration problem. LAV is particularly well suited for the dynamics of the Web of Data, as it allows easy add and remove of data sources. However, a high number of query rewritings may need to be executed to obtain results, representing a severe bottleneck. In this chapter, we defined the ReMakE problem as the maximisation of the results obtained by considering only a subset of rewritings of size k and proposed GUN as a solution.

GUN uses the RDF data model and takes advantage of the low cost of the graph union operation to construct an aggregate graph with the data from the relevant views used in the rewritings and execute the user's query on it. Compared to state-of-the-art approaches, GUN provides an alternative way to improve performance at the execution engine level rather than at the rewriter level. This makes

GUN usable with any LAV rewriter guaranteeing to achieve greater or equal answer completeness for the same R_k.

Our experiments demonstrate that GUN gain is real, *i.e.*, it returns all the answers when traditional execution returns none for 57% of the queries for values of k of at most 80, and for 38% of the queries for values of k of at most 500.

Furthermore, GUN consumes considerably less execution time than Jena in all the cases; the difference in execution time is tremendous, ranging from 2,5 to 681 times. However, this improvement in effectiveness and execution time is at the cost of an additional memory consumption of up to 12 times.

5.6 Perspectives

This work opens new perspectives for the practical implementation of LAV mediation in the context of the Web of Data. The main line of work is to avoid the rewriting phase and directly load a subset of relevant views such that maximise the chance of obtaining more results faster. Another interesting side-effect of not using rewriters is that a broader class of queries can be supported. This idea is already in development by fellow Ph.D. students at the GDD team, the interested reader is referred to [85, 42].

Other future works include:

- Study the effectiveness decrease in low-memory scenarios and the development of strategies to mitigate it, *e.g.*, to select the subset of data to load such that maximise the chances to obtain results.
- Study the execution time when queries require inference tasks and the development of strategies specially tailored to this case.

II

The Read/Write Web of Linked Data

6

Introduction

In this part we focus on solving Problem 2:

*H*ow to allow Linked Data participants to write each other's data and turn the Linked Data into Read/Write?. Which consistency criteria are suitable for a Read/Write Linked Data? How to maintain them respecting the autonomy of the participants and without compromising their availability and scaling in large number of participants and large quantity of data?

In this chapter we formalize the vision of the Read/Write Linked Data as a social network of update exchange. In chapter 7 we review the state of the art of different research communities concerning consistency criteria. In chapter 8 we propose to use Strong Eventual Consistency (SEC), a criterion from Optimistic Replication, for the Read/Write Linked Data, and detail SU-Set, a Conflict-Free Replicated Data Type to achieve it. In chapter 9, we develop *Fragment Consistency* a criterion stronger than SEC that takes in account the existence of fragments of data copied between participants, and detail Col-Graph, a protocol to achieve it.

In part 1 we tackled the problem of querying heterogeneous data sources in the Web of Linked Data. Once data can be queried, data consumers can start taking action on its *quality*, for example, the elimination of duplicate data, the addition of more links and/or the correction of links. However, the current Web of Linked Data is Read-Only, meaning that when consumers detect inconsistencies,

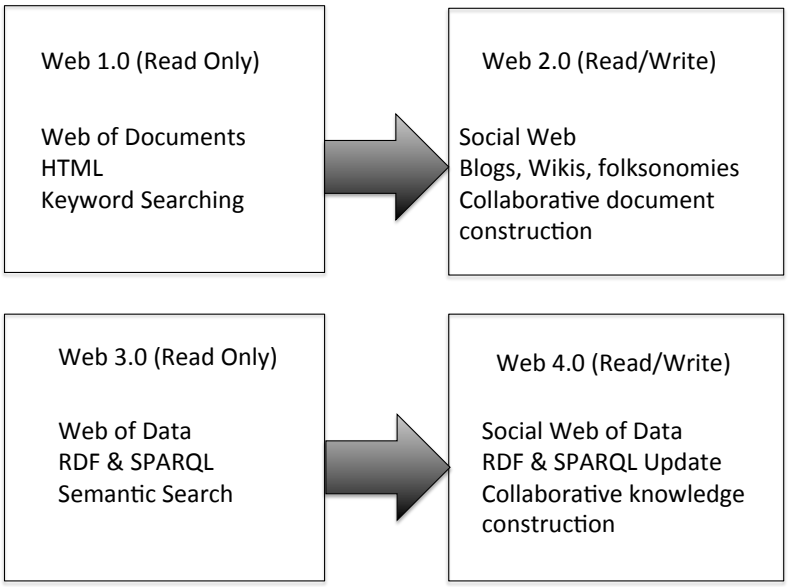


Figure 6.1: From Read-only to Read/Write in the web of documents and Linked Data.

they cannot fix them in place. To be able to do so, the Web of Linked Data needs to evolve from Read-Only to *Read/Write* [13]. If participants of the Web of Linked Data could write, data could be cleaned and evolve with the collaborative intervention of human and machine agents. The knowledge stored in different communities or even by different individuals or applications could *co-evolve* [37].

The paradigm shift from Read-Only to Read/Write would have a similar impact on the Web of Linked Data than the one of the advent of the Social Web had to the Web of Documents, as illustrated in Figure 6.1. The shift from the Web 1.0 to 2.0 allowed the collaborative edition of documents. The Web of Data could draw the same benefits to allow the collaborative construction of knowledge.

As a side effect of the co-evolution process, the availability of public Linked Data, which is currently an issue [22], could be improved. Data consumers copy data from different sources in order to perform intensive querying, keeping themselves up-to-date through live update feeds or notification protocols. While querying, mistakes can be identified and repaired. These updates can be integrated by the sources or exchanged between data consumers through copying or through *pull requests*, in the spirit of Distributed Version Control Systems (DVCS). Federated Query Engines

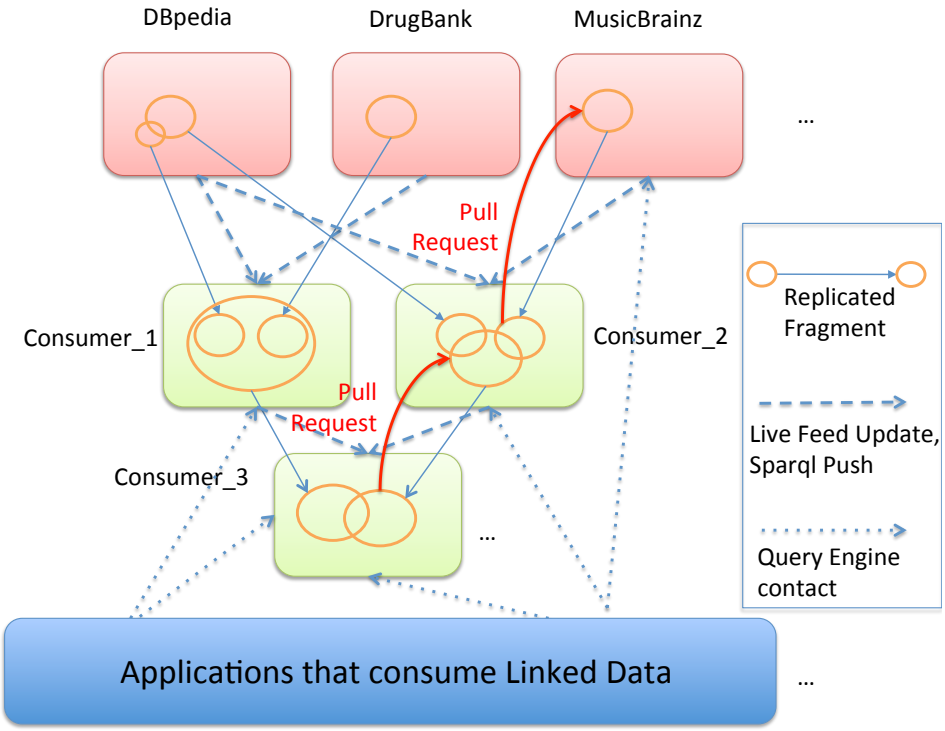


Figure 6.2: Federation of Read/Write Linked Data

could opportunistically exploit the generated replication scheme to balance the query load [104, 62].

Figure 6.2 [62] illustrates this vision. The top three boxes represent three major Linked Data publishers, DBpedia [1], DrugBank [2] and MusicBrainz [3]. *Consumer_1* copies fragments from DBPedia and DrugBank, *Consumer_2* copies fragments from DBPedia and MusicBrainz and *Consumer_3* copies fragments from *Consumer_1* and *Consumer_2*. Applications that consume Linked Data can request data from consumers besides that from the original data publishers.

However, if this update exchange is not properly managed, data may diverge uncontrollably, or in a non-deterministic way, making impossible the assertion of guarantees on the results of the queries and updates and severely undermining the interaction between participants [119]. Therefore, to realize this vision of a Read/Write Web of Linked Data, a suitable *consistency criterion* and an efficient algorithm to maintain it is needed.

The Web of Linked Data has the following characteristics that need to be taken in account when designing consistency criteria and algorithms to maintain them:

1. http://dbpedia.org
2. www.drugbank.ca
3. http://musicbrainz.org

1. A steadily growing and potentially very large number of participants. The advent of the "Web of Things" [47], where every-day use devices also start to publish data is expected to accelerate the growing.
2. A very large and steadily growing volume of data distributed among its participants, in the order of billions of pieces of data.
3. Published data are *format-heterogeneous*. Not all the data comply with the Linked Data principles, and instead of using RDF, one can find XML, JSON, CSV, spreadsheets or PDFs among many others.
4. Published data may use different vocabularies or ontologies to refer to the same facts about the same entities, *i.e.* they are *semantic-heterogeneous*.
5. Participants are *autonomous*, *i.e.*, there is no central control nor a priori coordination between them, further than the permission to read.

The Web of Linked Data is about individuals and organizations publishing and interlinking data. Each of these individuals and organizations may have different points of view on the same data [16], giving a social dimension to the Web of Linked Data. On the other hand, this organizations or individuals may share their data with each other: an individual may give access to its information, stored in a personal device, to several applications that she/he uses to connect with friends or professional contacts; organizations may share their data to collaborate in cleaning it or to discover new links. These use cases lead to the idea that the Web of Linked Data is strongly driven by *social interactions*. As such, we model it as a *social network* [128].

Results in this part were published in two research articles: those presented in chapter 8 in [61, 60] and those in chapter 9 in [62].

6.1 Motivating example

Suppose a medical laboratory wants to study the chemical structure of the drugs used for the treatment with drugs for T-Cell Lymphoma. For that purpose, they use the LinkedCT dataset, which has information about interventions on diseases and links to the DrugBank dataset on chemical structure of the drugs via the *rdfs:seeAlso* property. As they plan to do intensive querying, they need to have a local copy of the data. They copy the data they are interested into, for example, in the

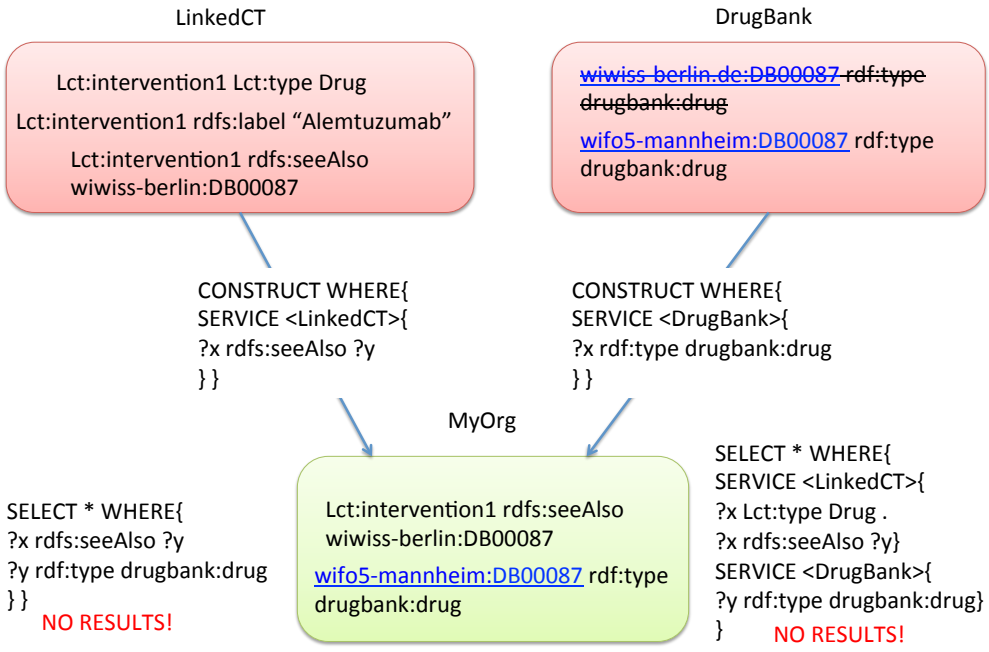


Figure 6.3: Data Quality issues prevent queries on Linked Data to return results.

Figure, the interventions of type drug linked to the T-Cell Lymphoma condition and the *rdfs:seeAlso* links to DrugBank from LinkedCT, and the labels and the chemical structure of the drugs from DrugBank.

However, when querying, they realize that DrugBank have recently changed of maintainer; all IRI's have been changed to reflect the new namespace, but the *rdf:seeAlso* links in LinkedCT link to triples with the old one, therefore, queries using these links do not return any result. Note that the same query evaluated in a federated way against the LinkedCT and DrugBank endpoints will suffer the same lack of results. Figure 6.3 illustrates the situation.

As LinkedCT updates happen each three months and they cannot wait, they decide to fix this other issues locally updating the *rdf:seeAlso* links. To help other researchers and to collect data on queries performed on their data, they publish their new derived dataset through a SPARQL endpoint. This has two positive side effects illustrated in Figure 6.4: first, local queries executed on the new endpoint work; second, external entities performing other queries targeting LinkedCT and DrugBank on the federation can profit of the fix to obtain results by evaluating the *rdfs:seeAlso* links on the new dataset, the updates made by the medical laboratory do not fix the query only for itself, but for the whole Linked Data Federation.

We believe that a system to support a Read/Write Linked Data needs to comply with the following

requirements:

1. Copying data and exchanging updates on it follows social rules. Updates will be exchanged and accepted only if they come from trusted participants, therefore, the system must support *social organisation*.
2. Sources may modify data, making copies outdated. Queries made on outdated copies do not return the same result in the sources. Therefore, the system requires a way to *synchronize* copies with sources while giving consistency guarantees. When copies are updated with curation purposes, sources may want to integrate them, therefore, synchronization of sources with copied is also required.
3. Synchronization can be performed in a naive way by re-executing the query that defines the copied data. However popular sources, *i.e.*, the ones where many copies have been taken, may become flooded with constant querying, therefore, *Self-Maintenance* of copies is needed, *i.e.*, synchronization must be achieved only by the means of the access of an update log or the sole reception of updates.
4. We assume that participants in the Linked Data network are autonomous and we project their number to be in the order of the several thousands. As such, a central coordinator cannot be used to synchronize, *i.e.*, we require a *coordination-free* protocol.
5. For the same autonomy of participants, synchronization must be *non-blocking*, *i.e.*, do not impede the querying and update of sources and copies by any means.do not impede the querying and update of sources and copies by any means.

6.2 Definitions

In this section, we model the Read/Write Linked Data as a social network, first we give the definition of a social network as in [128].

Definition 6.2.1. Actors *are discrete individual, corporate or collective social units. We will also refer to actors as* Participants.

Definition 6.2.2. *A* tie *is a link between a pair of actors, e.g. social interaction, like "friendship", or behavioural interaction, like "message sending". Formally, a* tie *is an ordered pair* $(a, b)_{Tie-Name}$ *meaning "actor a has a* $Tie - Name$ *tie with actor b.*

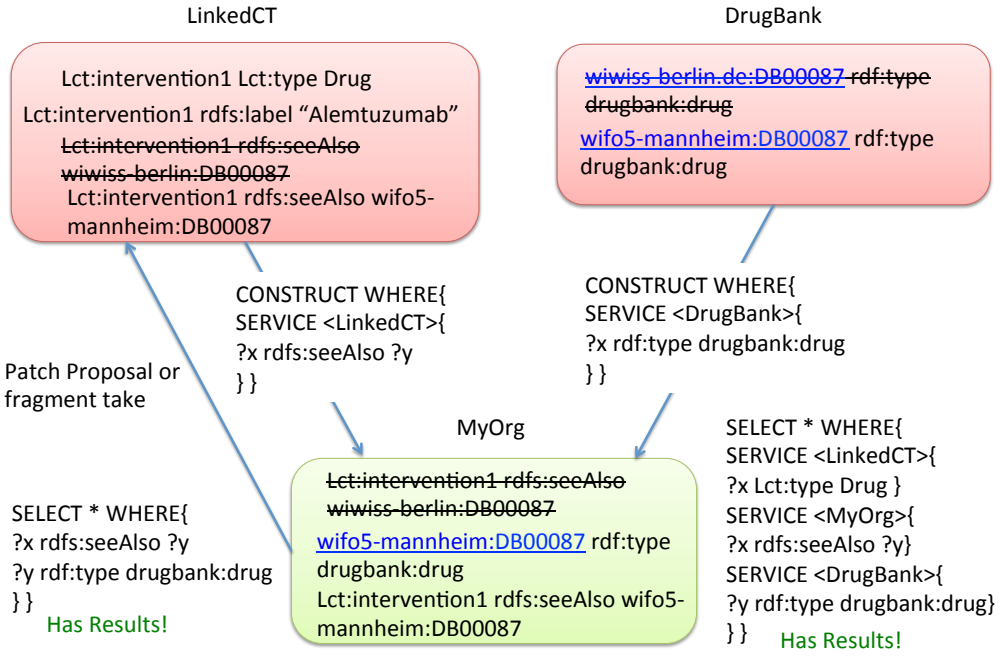


Figure 6.4: When a data consumer resolves ambiguities for its own use and publishes the updated dataset, other consumers can take advantage to correct queries and data publishers can take advantage to correct their own data.

Definition 6.2.3. *A* group *is a finite set of actors on which ties are to be measured.*

Definition 6.2.4. *A* relation *is a set of ties of a specific kind among members of a group.*

Definition 6.2.5. *A* Social Network *is a directed graph* (N, L) *where* N *is a group and* L *is a relation on the members of* N*.*

In the Web of Linked Data, actors can be individuals that store their information in a personal device (*e.g.* a rented server or plug), organizations that publish data (Government Open Data, DBpedia, etc.), or even real-world things connected to the web (sensors, infrastructure).

To align with the standards, we consider that of all these entities store their data in RDF-GraphStores. Therefore, we consider the group of the Web of Linked Data is a finite set of RDF-GraphStores.

The tie that defines the current Read-Only Web of Linked Data is *links to*, *i.e.*, the existence of triples in one actor that link to triples prefixed by the namespace of other. For a Read/Write Linked Data, we define the tie *Consume Updates*

Definition 6.2.6. *Let A an RDF-GraphStore that executes SPARQL 1.1 Update operations, we say*

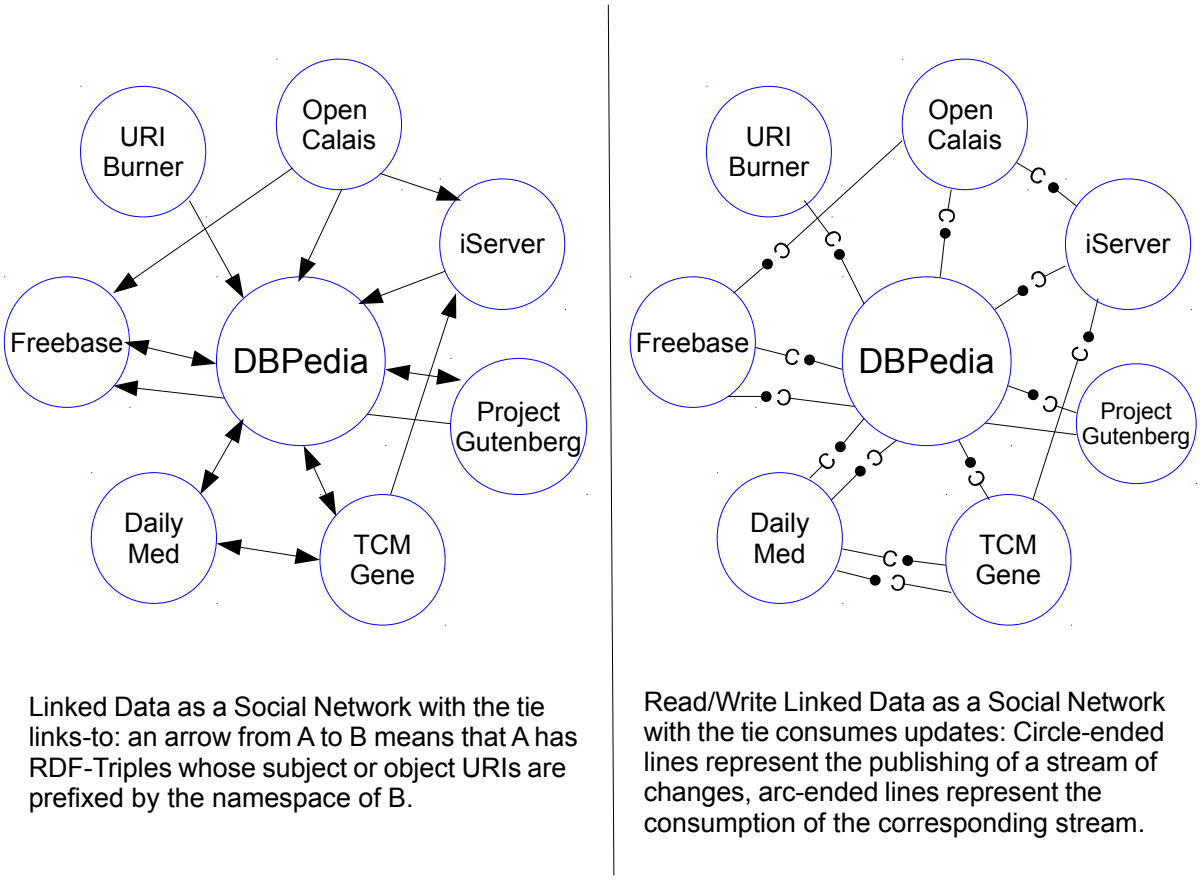


Figure 6.5: Comparison between the "links-to" and "consumes updates" ties. Actors are extracted from the Linking Open Data Cloud diagram [30].

that an RDF-GraphStore B Consumes Updates *of A if a subset of such updates arrive to B and B executes them.*

Definition 6.2.7. *Read/Write Linked Data The Read/Write Linked Data is a Social Network with* (A, T) *where A is a set of RDF-GraphStores and T is a set of ties* Consume Updates.

Figure 6.5 compares the networks generated by the links-to and consumes updates ties. When actors consume updates from each other, consistency issues arise. The presence of cycles, as depicted in Figure 6.6, where an update executed at the iServer node follows the path of consumption highlighted with the dashed line all the way back to itself may produce duplicate data or an infinite propagation of the operations in the network.

Concurrency issues are illustrated in figure 6.7. Imagine two nodes starting from the same state, one of them deletes a certain triple T and immediately inserts it again, at the same time a second site deletes the same triple but without inserting it back. After the mutual exchange and re-execution of

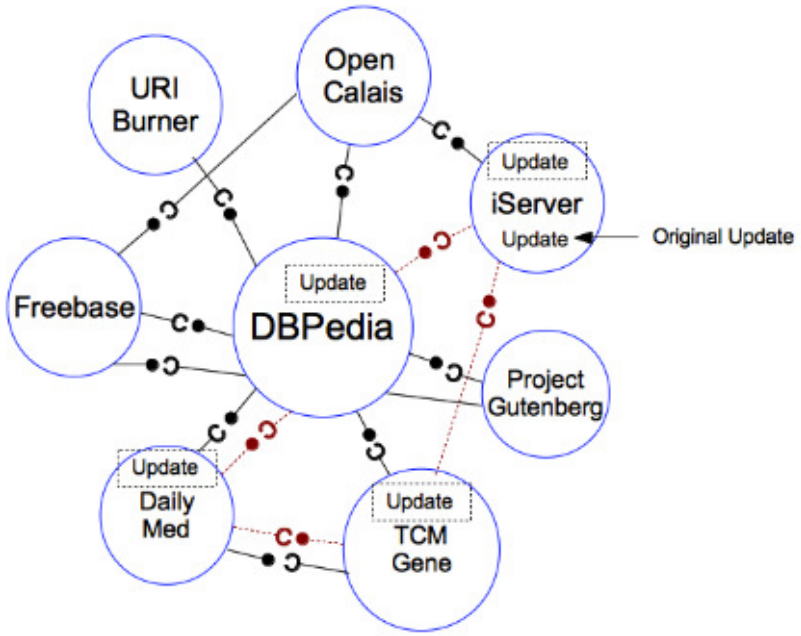


Figure 6.6: Cycles in a Read/Write Linked Data network can introduce consistency problems like receiving your own updates.

updates, the Graph Stores diverge, meaning that queries will not return the same results.

Therefore, the Read/Write Linked Data requires the update exchange to be driven by algorithms ensuring a *consistency* criterion, in order to give guarantees to the participants that even when they integrate concurrent updates; the final state will be the same independently of the order on which such updates are integrated. Furthermore, such algorithms need to scale in the characteristics of Linked Data enumerated in chapter 1, *i.e.*, a large number of autonomous participants that hold together a high volume of data.

We assume that participants may crash or go off-line for an undetermined period of time, and when they recover or go back on-line they do it with their memory intact.

Before going further, we define what an update means in the Read/Write Linked Data. The representation format for updates in the Semantic Web was first openly discussed in [12], where an ontology for semantic diff patches was discussed. [92] proposed to consider changes at the structural level, like in version control systems. In their model, the RDF-Triple is the minimal change unit, RDF-Triples cannot be changed, only added or removed, and the two basic update operations are the add or removal of an RDF-Triple. The SPARQL 1.1 Update specification (presented in chapter 2) follows this model. [9] extends this model to incorporate the concept of a logical connection between atomic changes and to support ontology evolution.

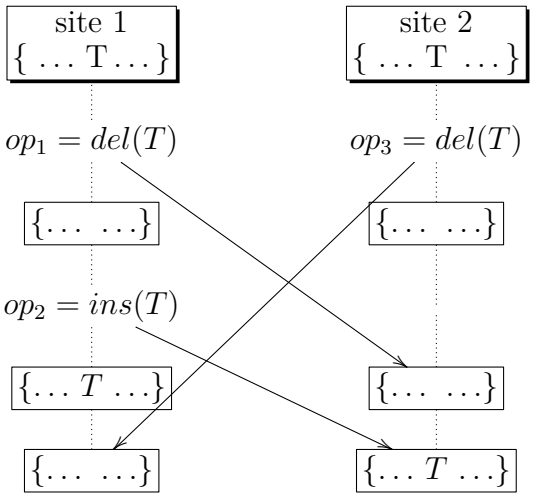


Figure 6.7: Example of inconsistency when exchanging updates on the same data

Recent efforts revive the idea of a patch as a piece of knowledge, implementing it following Distributed Version Control Patterns: [23] uses the Darcs' theory of patches, while [105] express them as a Git commit. Both approaches share the notion of representing a logically connected set of basic operations as an add-set and a del-set together with some provenance related to the change. Following their spirit, we define an Update as follows:

Definition 6.2.8. *An* update *is a pair* (Q, GS) *where* Q *is a SPARQL 1.1 Update query and* GS *an RDF-GraphStore comprised of three named graphs:*

- *The del-graph, comprised of all triples deleted by the effect of* Q*.*
- *The add-graph, comprised of all triples inserted by the effect of* Q*.*
- *The metadata graph, comprised of all other data about the update,* (e.g., *provenance).*

Updates are uniquely identifiable, e.g. *with an IRI minted from the id of the actor that created it. This information is stored in the metadata graph.*

This model allows each actor to decompose its updates at discretion: each insertion or deletion of a triple separately, or logically connected sequences of operations in the same update. At the implementation level, it also allows to query updates with SPARQL. Figure 6.8 shows an example of an update.

We abstract from the particular implementation of communication channel used to propagate updates (Publish-Subscribe, pull from a log, etc.), we will precise and cost-analyze required conditions on the communication channel , *i.e.*, if an order needs to be imposed or if additional *metadata* is needed, when studying each of our proposed consistency criterion.

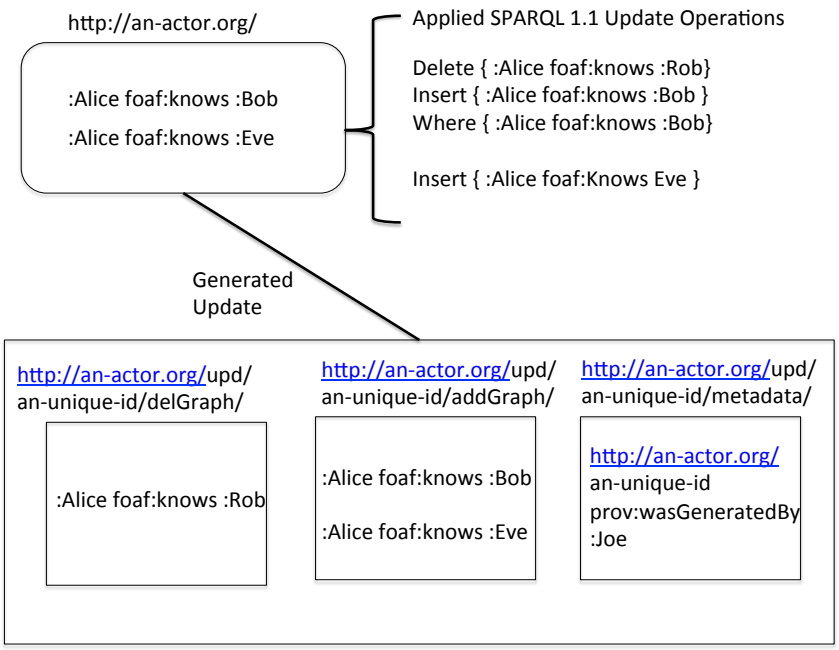


Figure 6.8: An Update represented as an RDF-GraphStore with named graphs for the deleted triples, the inserted triples and the metadata.

We consider Blank nodes to be *skolemized*, *i.e.*, uniquely identified by an IRI. This feature is detailed as optional for systems needing it in the RDF specification [127]. Non-skolemized blank nodes have local scope and pose problems when exchanging updates concerning them. For more information on the subject, the reader is referred to [81].

6.3 Problem Statement

Given the Read/Write Linked Data (definition 6.2.7), we look for a consistency criterion C and an algorithm A to execute update exchange that ensures C such that:

1. A is at most linear in space, time and communication (message size and number of message exchange).
2. A does not require coordination.
3. A is not blocking, *i.e.*, it allows participants to keep receiving queries from users and not depend on consensus.

7

State of the Art

In this chapter we describe the efforts made in several research communities in data consistency. In Computer-Supported Cooperative Work (CSCW), data consistency is studied in the context of cooperative edition, while in Distributed Systems, Databases, and Semantic Web, data consistency often appears when studying the general *replication* problem: A set of *replicas* on which operations are issued by clients (human or agents) and exchanged through messages [130]. When all messages are exchanged, the system must comply with a set of assertions: the consistency criterion. In the following sections, we will detail the efforts of each of them.

7.1 Consistency in Computer-Supported Cooperative Work

In CSCW, consistency appears in the context of cooperative edition: A certain number of users need to collaboratively work in a document, *e.g.*, source code, plain text or a complex CAD file. If users update simultaneously, consistency problems consistency appear, as the intended effects of the executed updates may conflict.

The most simple and intuitive way of dealing with this problem is to *lock* the document while someone is working on it and release it when he/she is done. Clearly, this has the problem of blocking the other users of doing something useful and the need of deciding an order on which users will edit

the document. If there are many users, or if users need a medium to high amount of time to update the document, locking is not the good strategy.

To avoid the problems of locking, Dourish proposed the *Copy-Modify-Merge* model [35]. Each user has his/her own *working copy* of the document, on which he/she can update at will. At some point, users merge their copies, resolve the conflicts that updates could have generated and continue updating. The most successful implementations of *Copy-Modify-Merge* are Version Control Systems (VCS). VCS can be (i) *Centralized*, *i.e.*, a central server holds the last version of the document. Users copy from it and merge with it. The server has the ability to detect conflicts between its current version and the version of an user that has not merged in a while, that is, other users have merged with the server during this time. Practical examples include Subversion and CVS. (ii) *Distributed*, *i.e.*, no central server, users merge with other users at will, each working copy has the ability to detect conflicts. Practical examples include Git and Darcs.

In the context of Real-Time editing systems, Sun et al. proposed the *Convergence-Causality-Intention* (CCI) criterion [113]:

Definition 7.1.1. *CCI*

- *Convergence: When the same set of operations have been executed at all sites, all copies of the shared document are identical.*
- *Causality preservation: For any pair of operations O_a and O_b, if O_a happens before O_b, then O_a is executed before O_b at all sites.*
- *Intention: The intention of an operation O is the execution effect which can be achieved by applying O on the document state from which O was generated.*
- *Intention preservation: For any operation O, the effects of executing O at all sites are the same as the intention of O, and the effect of executing O does not change the effects of independent operations*

Essentially, convergence states that all copies have the same state, causality that all operations were executed in the same order they were executed and transmitted, and intention that post-conditions of independent operations do not interfere with each other. In practice, intention preservation ensures that convergence is not guaranteed in a naive way, *e.g.*, if each participant ignores every arriving operation, all sites will be equal to the empty state, giving a correct, but practically useless guarantee of convergence.

VCS guarantee convergence and causality preservation. Intention preservation is difficult to formalise and prove. To the best of our knowledge, the most recent effort is [14].

The CCI model has been applied to Semantic Wikis [101, 111, 129], notably through the Operational Transformation (OT) approach [36]. The basic idea of OT is the transformation of updates operation according to the effects of previously executed concurrent updates so that the transformed update achieves the correct effect and maintain document consistency. The main drawback of OT is its limited scalability.

7.2 Replication and Consistency in Distributed Systems

Replication algorithms and consistency criteria in Distributed Systems can be divided in two main families [103, 70]. The first one is the *Pessimistic* category, the goal is to attain *Strong* consistency, *i.e.*, clients will have the illusion that there is only one replica, fully protected from the effects of failures and concurrent updates. However, the fundamental CAP Theorem [21] states than in the presence of partitions, whether it be by communication disconnection or by *off-line operations*, is it not possible to have strong consistency without sacrificing high availability. Indeed, the protocols to guarantee strong consistency need to block the system, therefore, the family of *Optimistic* replication algorithms [103] was developed.

Optimistic protocols focus on *weak consistency* criteria, where replicas are allowed to diverge during some time (the time of the partition) but remain available, causing the output of some reads (queries) to be different at different replicas during this time window. The main criterion studied in this family is *Eventual Consistency* [103]

Definition 7.2.1. *Eventual Consistency*

- Eventual Delivery: *An update delivered at some replica is eventually delivered to all replicas.*
- Termination: *All methods execution terminates.*
- Convergence: *Replicas that have delivered the same updates eventually reach equivalent state.*

Note that Version Control Systems described in section 7.1 comply with Eventual Consistency. Other systems that implement eventual consistency are Bayou [97] and IceCube [99]. Both consider the concept of *tentative writes* that are susceptible of being rolled back in case of conflict. However, roll-back procedures depend on the application semantics and are in general complex to develop.

This procedures require, either a central coordinator that decides an operation schedule to apply, or a consensus between actors. Both solutions undermine the autonomy of actors and the scalability of the system.

An alternative is the use of Conflict-Free Replicated Data Types (CRDTs) [110, 109]. A CRDT is a data type whose operations, when concurrent, yield the same result regardless the execution order. Two operations are concurrent if they occur at different nodes and we cannot determine which one happened before the other. Common data types are in general not CRDTs, for example, the set data type is not a CRDT, because concurrent insertion and deletion does not commute.

CRDTs satisfy a stronger version of Eventual Consistency called *Strong Eventual Consistency* (SEC). SEC differs from Eventual Consistency in the replacement of the Convergence property by *Strong Convergence*:

Definition 7.2.2. *Strong Convergence([110]) Replicas that have delivered the same updates have equivalent state.*

A CRDT has a *payload*, the internal structure that holds the state of the object, a *lookup* function that queries the payload and returns data element and an *update* function for handling operations such as *add* and *remove*. These operations could have some pre and post conditions.

An update operation is prepared at the generator site and sent to all nodes, including the generator one. (i) *prepare*, the arguments for sending the operation to the other nodes are prepared, if the preconditions of the operation does not hold, the operation is ignored; and (ii) *effect*, the operation, with the previously prepared arguments, is sent to all nodes including the generating one. When the operation is received, it will be executed if its preconditions are evaluated to true, else it will be delayed until they do so.

Many CRDTs have been developed to mimic basic data types (for example, sequences for cooperative text editing [89]). We will focus on the ones developed for the Set type as they are closer to RDF-Graphs.

B-Set [132] relies on the storage of "tombstones", i.e., the removed elements are just hidden from the user via the lookup operation. The use of tombstones is not appropriate for big sets like the ones we can find in the Web of Data, as their space complexity is high. C-Set [8] uses a counter associated to each element to keep track of how many times it has been added and deleted. An element is considered member of the C-Set if its counter is greater than zero.

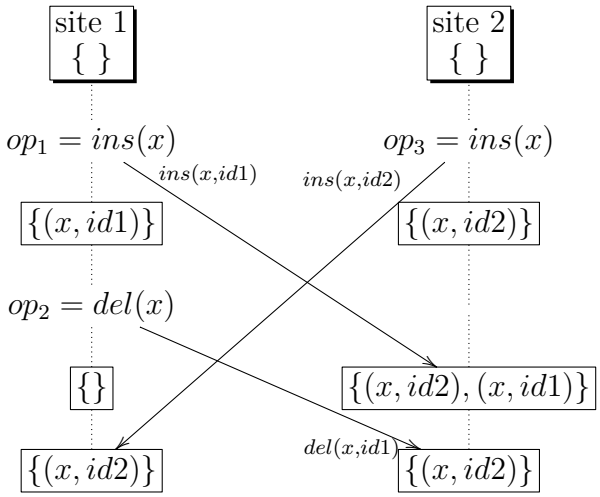


Figure 7.1: Execution of concurrent insertion and deletion of the same element in OR-Set

In the Observed-Removed Set (OR-Set) [110], detailed in Specification 7.1, each time an element is inserted (lines 5-11), is tagged with a globally unique id. These unique ids can be generated by means of a UUID [76] implementation, or by a combination of an unique site id and a monotonic counter. An element is considered member of the set if there is at least one pair of the form (element,id) in the payload (lines 3-4). When deleting an element, one can only remove the pairs (element,id) that are present in the set at the instant of the deletion (lines 10-17). Note that for applying an incoming delete, a *causal delivery* precondition (line 15-16) is required, *i.e.*, the operation that inserted the specific pair (element,id) being deleted must have been received and applied before applying the deletion.

or by a mechanism for ordering events in a distributed system such as vector clocks [82]. Figure 7.1 shows an execution of OR-Set in the case of a concurrent insertion.

Further improvements of OR-Set were developed in [88] and [15] to optimize its complexity, and in [33] to add sharding in order to optimize scalability.

7.3 Consistency and Update Exchange in Databases

The Thomas Write Rule [66] is a classical rule to manage duplicated databases, it allows multiple database copies to be synchronized over an unreliable communication channel connecting a fixed number of processes. From the point of view of the CCI consistency model, it guarantees only convergence, but it requires extra storage. In addition to the element identifier, it needs to store the

```
1  payload set S
2      initial ∅
3  query lookup (element e) : boolean b
4      let b = (∃u | : (e, u) ∈ S)
5  update add (element e)
6      prepare(e)
7          let α = unique()
8      effect(e, α)
9          S := S ∪ {(e, α)}
10 update remove (element e)
11     prepare(e)
12         pre lookup(e)
13             let R = {(e, u) | (∃u : (e, u) ∈ S)}
14     effect(R)
15         // Causal Reception precondition
16         pre ∀(e, u) ∈ R : add(e, u) delivered
17         S := S \ R
```

Specification 7.1: OR-Set

timestamp of creation, the timestamp of modification and a flag of deletion called “tombstone". All of this implies that an element cannot be really deleted unless its timestamp is older than the oldest timestamp of the whole system, which requires a protocol not suitable for the autonomy constraints, or when we consider that they are old enough, which is unsafe. Also, at the semantic web scale, the maintenance of tombstones is impractical.

In Distributed Databases [93], the Linked Data scenario falls into the *distributed* (or *multi-master*) family, as the execution of updates is allowed at every participant, and into the *Lazy propagated* category, as local updates are immediately committed and only eventually propagated and committed at the followers.

In this area, two types of consistency are studied: *Transaction Consistency*, that refers to the maintenance of integrity constraints when concurrent updates or transactions occur, and *Mutual Consistency*, that refers to the data items having identical values at all replicas. *Mutual Consistency* can be *strong*, *i.e.*, all instances of a data item must have the same value at the end of an update, or *weak*, *i.e.*, when the update its locally executed but the values at the replicas are allowed to temporally diverge. *Weak Mutual Consistency* is the same as the *Eventual Consistency* notion presented in section 7.2.

Distributed Lazy propagated systems have been developed in the context of clustered databases [29, 94], however, they rely on the fast and highly reliable network that connects cluster systems to avoid

blocking, an assumption we do not have in our context.

Recently, [6] introduced Webdamlog, a datalog-style language for managing distributed data and knowledge. Webdamlog extends Datalog adding *delegation*, *i.e.*, the ability to exchange rules besides facts among peers, and the possibility to use variables to denote peers and relations. However, Webdamlog is aimed towards *sharing* data(*e.g.*, Facebook pictures, selected tweets) instead of collaborating. As such, data received from one peer cannot be modified and sent back to its origin.

7.3.1 Materialized Views: Maintenance and Update

Another related research area is *Materialized Views* [28]. The main context is as follows: a snapshot of the result of a query targeting a set of databases is stored in a separate location in order to have faster access and/or decrease the load of the source databases. This raises two issues:

1. When data is updated in the source databases, changes need to be reflected in the snapshot in an efficient way, *i.e.* avoiding when possible the re-evaluation of the query. This problem is known as *View Maintenance*.

2. If data is updated on the snapshot, how to reflect such changes in the source databases?. This problem is known as *View Update*.

The consistency criterion for the system comprised by the snapshot and the sources is that the snapshot must be equal to the evaluation of the query that defines the view on the sources.

Many solutions for the *View Maintenance* problem for the relational context were developed during the late 90s and early 2000s [135, 80, 86, 133, 26, 27], and more recently, for Very Large Scale Distributed (VLSD) data storage systems [5], Social Networking applications [83] and Social Semantic Web [100]. In the RDF and Graph context, foundations were laid in [134, 3]. Further development includes [59, 40].

Concerning View Update, foundations were first described in [11] and [32]. The recent book by Date [31] thoroughly covers the relational case. A formalism based on the mathematical theory of Bi-directional transformations or *lenses* that enhances the classical treatment was developed in the series [20, 43, 57].

7.3.2 Peer Data Management

In Peer Data Management (PDM), participants act like peer-to-peer (P2P) networks : they make their own data accessible to others, but retaining full control on it and using their own schema. Participants might act as data providers offering query answering services or as mediators providing an integrated view on the data of multiple other nodes. Participants are interconnected by "semantic" links representing schema correspondences (called mappings), which are used for query rewriting and routing.

The consistency criteria used in PDM is that all rules defined by the schema mappings must hold, which is in general expensive to maintain. Each time an update occurs in one peer and is propagated forward, the mappings need to be checked and repaired if needed. The main procedure for this task is called *chase*, and was formalized in [39]. However, the chase is in general expensive and to be decidable requires some restrictions in the topology induced by the mappings.

Surveys about PDM systems can be found in [102, 58], we focus here on the systems that provide *Update Exchange* capabilities as they are closer to our vision of a Read/Write Linked Data. In Update Exchange, each participant controls a local database instance, including his data and data imported from others. Declarative schema mappings specify relationships between participants. Each peer updates their data locally occasionally *exchanging updates* with others, *i.e.*, propagating updates to make its database consistent with the others according to the schema mappings and local trust policies. The update exchange process is bidirectional and involves publishing any updates made locally by the participant, then importing new updates from other participants. Consistency criteria for Update Exchange able systems are weaker than other PDM systems.

Youtopia [73] challenges the idea that all schema mapping violations must be repaired *ipso facto* when updates occur. The main idea is to make the chase stop if it finds an inconsistency to ask for user input. The consistency criterion is derived from the Distributed Systems notion of *Final-State Serializability*. However, current Youtopia is tailored to a single database comprised of multiple relational tables that is updated by several users concurrently.

Orchestra [114, 63, 69] implements the vision of *Collaborative Data Sharing Systems (CDSS)* [64], based on Update Exchange. As Youtopia, CDSS rejects the idea that when peers with databases with heterogeneous schemas collaborate, they need to all align to a global schema and converge to the same final state, in other words, to tolerate disagreement between peers. They propose to establish

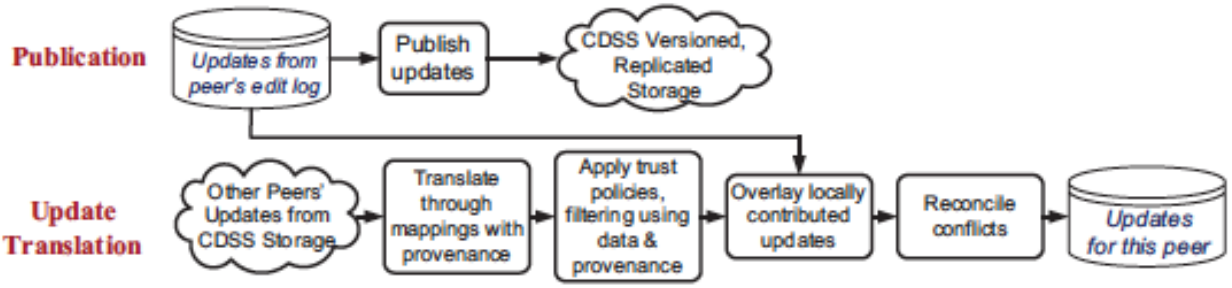


Figure 7.2: Flow of data in the update translation and exchange of Orchestra, taken from [69]

connections between peers via GLAV mappings, a generalisation of the GAV and LAV mappings detailed in chapter 4. Data and updates are annotated with elements of *Provenance Semirings* [68, 45] that allow the identification of which peer made it, and the introduction of trust expressions to solve conflicts. The consistency criterion is based on the chase procedure. The specification of a criterion independent of the chase is an open research topic [63].

Update exchange in Orchestra is illustrated in Figure 7.2, each peer updates its instance locally. Updates are recorded in a local edit log. Periodically, peers request that the CDSS performs an update exchange operation. The peer's local edit log becomes globally available and the CDSS translates the effect of updates published by other peers to the schema of the peer that requested the update exchange. The CDSS has an epoch number that advances after each batch of updates published by a peer, when an update exchange is triggered, the peer receives all state published up to that epoch and nothing more until its next update exchange. The need of this epoch number makes the system depending on a central coordinator and makes also impossible two simultaneous update exchange requests.

In summary, PDM systems are tailored towards supporting the most expressive mappings possible between peers and strong consistency guarantees at the expense of scalability and in some cases, availability. None of the systems described in this section scale above low hundreds of peers.

7.4 Consistency and Update Exchange in Semantic Web

The work of [12] identified the relationship between synchronization and updating of semantic data stores. They proposed an ontology to handle the synchronization by exchanging diff files, but without explaining how to achieve consistency.

[23] and [105] adapt to RDF the Distributed Version Control Systems DARCS and Git respectively. Both guarantee Eventual Consistency. The main goal of both works is to support version control and historical queries, as such, they store an important amount of meta-data that may be difficult to handle in highly dynamic environments.

Edutella [90] is intended to be a standardized query exchange mechanism for RDF metadata stored in distributed RDF repositories and is meant to serve as both query interface for individual RDF repositories located at single Edutella peers as well as query interface for distributed queries spanning multiple RDF repositories. Edutella is an hybrid P2P network based on the concept of super peers that allow faster query routing. As such, the network is self-organized, not social, and there is not a formal consistency criterion.

Other P2P based idea is *ViP2P* [67, 136]. Peers materialize views of XML data they are interested in the network. View definitions are indexed in a DHT so that each time a query is posed on a peer, it is answered by rewriting it on the views published in it.

Piazza [52, 51] was the first Peer Data Management System tailored to the Semantic Web. It proposes a language to guarantee schema mapping between XML and RDF sources and an algorithm to query on such a system. As the rest of PDM systems described in section 7.3.2, Piazza's main drawback is its limited scalability.

RDFSync [116] is an algorithm to synchronize two RDF graphs inspired by the file synchronization tool rsync. It is based on the decomposition in Minimal Self-Contained graphs (MSGs), that are hashed and stored in an ordered list. By calculating the difference between this hashes' list, and transmitting only the relevant MSGs, RDFSync greatly reduces the communication complexity compared to a pure rsync executed over a string representation of the graphs. However, it does not consider concurrent synchronizations between many participants.

RDFGrowth [117] is an algorithm for semantic P2P applications, designed for the particular scenario where peers want to increment their knowledge of certain topics with the knowledge of the other peers. It only considers the grow of the knowledge base (*i.e.* add operations), based on the monotonicity of the RDF Semantics. RDFGrowth considers that non-monotonic operations should not affect the shared knowledge and therefore, should be locally kept. RDFGrowth requires a bounded complexity of queries to scale over an unreliable network. A collaborative platform based on these ideas is presented in [115].

Schandl [106] proposes a partial replication of RDF graph for mobile devices using the same principles of SVN with a limited lifetime of local replica checkout-commit cycle. Therefore, it is not possible to ensure synchronization of partial copies with the source since a data consumer has to checkout a new partial graph after committing changing to the data provider.

Linked Data Fragments (LDF) proposes data fragmentation and replication as an alternative to pure SPARQL endpoints to improve availability. Instead of answering complex queries itself, the server publishes a set of fragments that corresponds to specific triple patterns in the query, offloading to the clients the task of deriving results from them and enabling the use of http caches. However, the problem of writability is not considered[120].

7.4.1 Update Propagation in the Semantic Web

Some works have considered the problem of how to propagate updates in the context of semantic web. SparqlPush [95] implements the well-known Publish-Subscribe model through the Pubsubhubbub protocol in SPARQL. A server performs continuous querying on the data publishers of the queries demanded by subscribers, and sends them the changes it detects.

Halaschek-Wiener and Kolovski [49] formalize an OWL-based syndication framework that uses description logic reasoning to match published information with subscription requests. They use the concept of incremental view maintenance in response to changes in the published data. The connection between consumers and publishers is handled by special actors called *syndication brokers* that, as the SparqlPush server, are in charge of delivering the changes.

DBpedia Live [87] is a framework to extract and transform in real time the text and info-boxes data in Wikipedia into DBpedia [18]. One of the components of the system is a synchronization tool to keep DBpedia mirrors up-to-date. Updates are published as pairs of plain text files, one with the insertions and the other with deletions that happen in a timeframe of a few seconds. The names of the files include a timestamp that totally orders them, allowing the mirrors to be consistent with the master.

The three systems described above have the limitation that consumers are passive, *i.e.*, they do not make updates on the data they consume.

Data-Fu [112] is a declarative rule-based execution language with a state transition system as formal grounding that enables the development of datadriven applications that facilitate the RESTful

manipulation of read/write Linked Data resources. No consistency criteria is considered.

7.5 Summary

In conclusion, the Distributed Systems, Database, Semantic Web and CSCW research communities agree on the result of the CAP theorem: there is a trade-off between the consistency level and the overhead impacting the availability and scalability of the system, Strong consistency is not attainable in a scalable way. For applications where availability and scalability are a must, only *Weak Consistency* guarantees can be assured, usually, *Eventual Consistency* and *Strong Eventual Consistency* are the criteria of choice.

As the Read/Write Linked Data requires availability and scalability, we will focus on criteria and protocols for weak consistency. In chapter 8 we show how to achieve Strong Eventual Consistency in the Read/Write Linked Data using SU-Set, a CRDT for RDF-Graphs and SPARQL 1.1 Update operations.

SU-Set requires that all updates are delivered to all participants, *i.e.*, forcing the Read/Write Linked Data to be connected and that participants copy all data from sources. In chapter 9, we present Fragment Maintenance, a stronger criterion than SEC, and Col-Graph, a protocol to achieve it in the Read/Write Linked Data, that eliminates SU-Set restrictions.

8

SU-Set: Strong Eventual Consistency

In this chapter, we present SU-Set, a CRDT for the RDF-Graph type with SPARQL 1.1 Update operations that solves problem 6.3 using Strong Eventual Consistency as consistency criterion.

In the CRDT model detailed in section 7.2, operations are assumed to be transmitted through a fully connected communication graph without loss, granting the *Eventual Delivery* condition of SEC. Therefore, we introduce on the Read/Write Web of Linked Data the assumption that the update exchange network is strongly connected, all updates eventually arrive to their destination, and the following definition of the consume-updates tie:

Definition 8.0.1 (Consumes Updates)**.** *Let A be an RDF-GraphStore that executes SPARQL 1.1 Update operations in the Read-Write Linked Data (definition 6.2.7), we say that an RDF-GraphStore B* Consumes Updates *of A if all such updates arrive to B and B executes them.*

8.1 SU-Set

As pointed out in chapter 2, all SPARQL 1.1 Update graph update operations can be expressed as set union and difference on RDF-Graphs. Therefore, one can adapt the existing CRDTs for the set type. Our proposal, dubbed SU-Set, extends OR-Set's (Specification 7.1) single-element insertion

```
1  payload set S
2      initial ∅
3  query lookup (element e) : boolean b
4      let b = (∃u : (t, u) ∈ S)
5  update insert (set<element> T)
6      prepare(T)
7          let R = ∅
8          foreach t in T:
9              let α = unique()
10             R := R ∪ {(t, α)}
11     effect(R)
12         S := S ∪ R
```

Specification 8.1: Union extension to OR-Set

and deletion to union and difference. Specification 8.1, shows how to do it for the insert. Figure 8.1 shows a SU-Set execution, SPARQL 1.1 Update operations executed at Graph Stores are rewritten to SU-Set operations over pairs (triple,id) in a transparent way for the user, and sent downstream, where they are re-executed upon reception.

SU-Set inherits from OR-Set a precondition concerning the delivery of updates: It must be granted that deletions of unique pairs are always delivered after the insertions that generated them. We will use the same strategy used in OR-Set to make the pre-condition hold: a causal delivery of updates, implemented with *Vector Clocks* [82]. Each Graph Store holds a monotonically increasing counter (the clock) that ticks each time an update is made, and an array whose keys are the identifiers of all other Graph Stores and whose values are the last received clock values from the corresponding Graph Store. Updates are piggybacked with the full vector at execution time. By comparing the vectors, one can determine the partial order of update executions in all the network to ensure that a deletion always happens after its corresponding insertion.

This first SU-Set version has two important overheads to consider: first, in the delete-insert operations, computing the triples affected locally and sending them downstream instead of sending the patterns directly greatly increases the traffic; second, if each element needs to be sent with its globally unique id, the size of the packets sent will grow.

To test this in a real case, we analyzed the publishing method of DBpedia Live. The core of the system is a set of extractors that computes the triples affected each time there is an edition in a Wikipedia page or in the mappings that define the relation between info boxes and triples. After updating the store, the system writes two files, one with the added RDF-triples, and another with

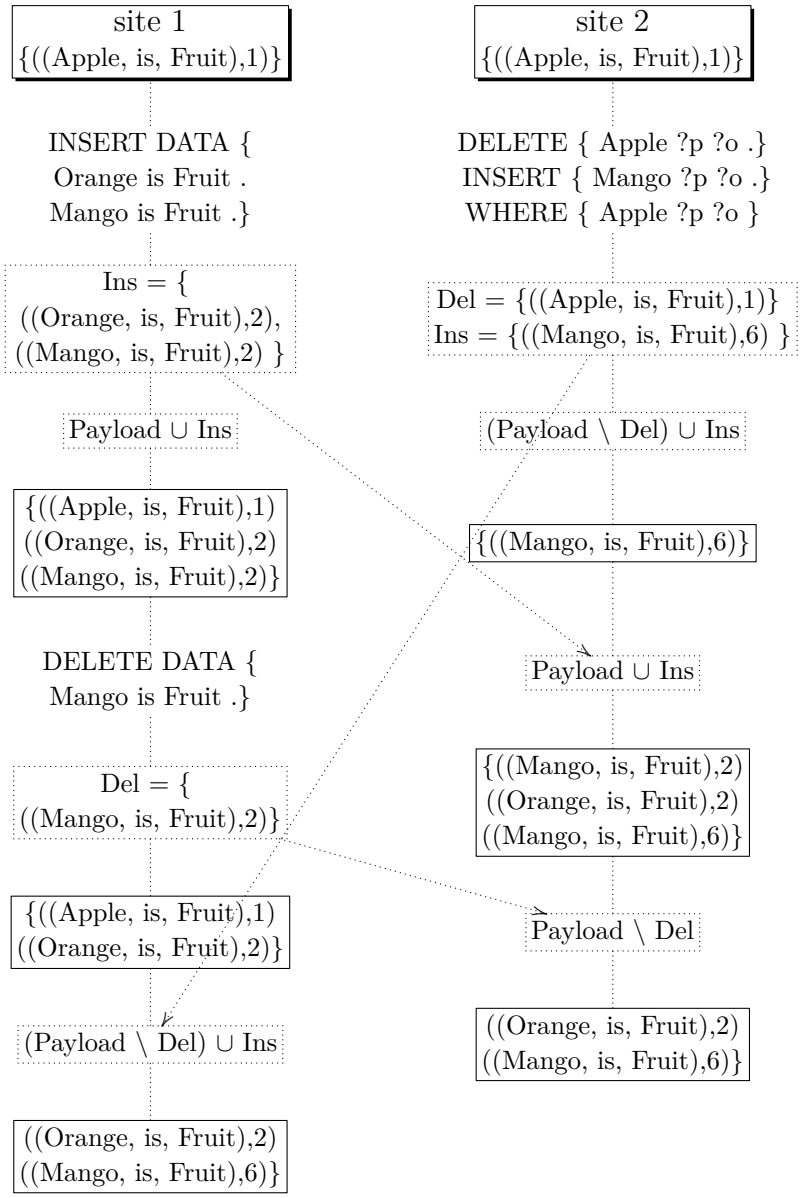


Figure 8.1: SU-Set execution

the deleted ones. These files do not have a fixed size, as this depends on the number and nature of the editions at a given time. DBpedia Live publishing can be considered as SPARQL Update Insert Data and Delete Data operations with the triples defined by the change set files. This means that for the DBpedia Live case, the overhead of computing and sending the affected triples for each operation is already considered and SU-Set do not adds any further cost.

To evaluate the impact of using globally unique ids, we consider its implementation with two UUIDs [76] of 16 bytes each, one to identify the generator site, and another to hold a big enough monotonic counter that increases with each insert. We downloaded the N3 files published by DBpedia Live from march 10th to march 16th 2012, totalizing 3,403 Megabytes, and appended to each triple the base64 representation of the two UUIDs. Finally, we measured the new file size with the UNIX command wc -c. The difference between the version with ids and the version without ids was 2,04 GB, and the percentage of increase, 54,47%.

However, we can greatly reduce this overhead if the receivers can afford to spend some time in constructing the IDs from a resume. The strategy varies depending on the strategy chosen to achieve the delivery condition. When vector clocks are used, we showed in [60] that the same id can be shared by the triples inserted in the same operation, as the uniqueness of the element comprised by the triple and its id is maintained. Therefore, one can send only one id per insert operation and let the receiver reconstruct the pairs.

In DBpedia Live, this would mean that only one id is needed for each file containing added triples. We recomputed the overhead in our case of study using this strategy and we obtained a negligible 2,5 Megabytes for the insertions and a 4,68% file size increase overall. Note that, as the average triple size is greater than the id size, deletion with the first version of SU-Set is less expensive than without using any ids.Table 8.1 compares the differences in communication overhead of insertions and deletions between the two versions of SU-Set and the current publication method without ids. Note that in both solutions, it is possible to further optimize the deletion by sending only the id (as it is unique), however, in the case that we would like to analyze the deletes, an extra computation effort needs to be done to search the triples. As such, we report the overhead of sending only the ids and, in parentheses, the overhead of sending full pairs.

Note only that sending ids for deletion is less expensive than sending the serialized n3 triples. Further optimisations like compression are out of the scope of our work, the interested reader is

referred to [41].

Table 8.1: Comparison of communication overhead between SU-Set, its optimized version and the use of no ids (nothing). The data used are the triples published by DBpedia Live from the 10th to the 16th march 2012.

		Size (MB)		
Operation	# Triples	Nothing	SU-Set	Optim.
Insert	21762190	3294,08	5334,29	3296,6
Delete	1755888	265,78	164,61 (430,4)	164,61 (430,4)
Total	23518078	3559,86	5498,9 (5794.69)	3461,21 (3727)
Overhead			54,47% (64,77%)	-2,77% (4,68%)

8.2 Proof of correctness

To prove that SU-Set is a CRDT, *i.e.*, the execution of concurrent downstream operations a, b on a SU-Set S commutes: $S \circ a \circ b = S \circ b \circ a$. We proceed with a case-based analysis for each combination of insert, delete and delete/insert for the optimized version. The proof for the non-optimized version is the same. Let T_1, T_2 sets of RDF-Triples and S the payload of an arbitrary SU-Set.

- **Two inserts:** We need to show that $S \circ insert(T_1, \alpha_1) \circ insert(T_2, \alpha_2) = S \circ insert(T_2, \alpha_2) \circ insert(T_1, \alpha_1)$. Applying the definition of the downstream insert from specification 8.2, let $R_1 = \{(t, \alpha_1) : t \in T_1\}$ and $R_2 = \{(t, \alpha_2) : t \in T_2\}$, therefore, we need to verify that $(S \cup R_1) \cup R_2 = (S \cup R_2) \cup R_1$, which is straightforward by the associativity and commutativity of the set union.
- **Two deletes:** We need to show that $S \circ delete(R_1) \circ delete(R_2) = S \circ delete(R_2) \circ delete(R_1)$. Applying the definition of the downstream insert from specification 8.2, we need to verify that $(S \setminus R_1) \setminus R_2 = (S \setminus R_2) \setminus R_1$. We start from the left hand of the equation and we arrive to the right hand by basic set theory theorems [46], particularly the set difference definition as the intersection with the complement. In the following, we will use the proof style of Gries and Schneider [46], each step of the proof is followed by the axiom or theorem that justifies the logical symbol (equality, equivalence, implication) that leads to the next step.

  $(S \setminus R_1) \setminus R_2$
  $= <A \setminus B = A \cap B^{\complement}>$
  $(S \setminus R_1) \cap R_2^{\complement}$

```
1  payload set S
2      initial ∅
3  query lookup (triple t) : boolean b
4      let b = (∃u : (t, u) ∈ S)
5  update insert (set<triple> T)
6      prepare(T)
7          let α = unique()
8      effect(T, α)
9          let R = {(t, α) : t ∈ T}
10         S := S ∪ R
11 update delete (set<triple> T)
12     prepare(T)
13         let R = ∅
14         foreach t in T:
15             let Q = {(t, u) | (∃u : | (t, u) ∈ S)}
16             R := R ∪ Q
17     effect(R)
18         // Causal Reception
19         pre All add(t,u) delivered
20         S := S \ R
21 update delete − insert(whrPat, delPat, insPat)
22     // match(m, pattern): triples matching
23     //     pattern within mapping m.
24     prepare(whrPat, delPat, insPat)
25         let S′ = {t | (∃u | : (t, u) ∈ S)}
26         // M is a Multiset of mappings
27         let M = eval(Select *
28                          from S′ where whrPat)
29         let D′ = ∅
30         foreach m in M:
31             D′ = D′ ∪ match(m, delPat)
32         let D = {(t, u) : t ∈ D′ ∧ (t, u) ∈ S}
33         let I′ = ∅
34         foreach m in M:
35             let I′ = I′ ∪ match(m, insPat)
36         let α = unique()
37     effect(D, I′, α)
38         // Causal Reception
39         pre All add(f,u) ∈ D delivered
40         let I = {(i, α) : i ∈ I′}
41         S := (S \ D) ∪ I
```

Specification 8.2: Optimized SU-Set

$= <A \setminus B = A \cap B^{\complement}>$

$(S \cap R_1^{\complement}) \cap R_2^{\complement}$

$= <\text{Associativity of } \cap>$

$S \cap (R_1^{\complement} \cap R_2^{\complement})$

$= <\text{Commutativity of } \cap>$

$S \cap (R_2^{\complement} \cap R_1^{\complement})$

$= <\text{Associativity of } \cap>$

$(S \cap R_2^{\complement}) \cap R_1^{\complement}$

$= <A \setminus B = A \cap B^{\complement}>$

$(S \setminus R_2) \cap R_1^{\complement}$

$= <A \setminus B = A \cap B^{\complement}>$

$(S \setminus R_2) \setminus R_1$

We will make use of this result in subsequent demonstrations under the name of *Lemma 1*: for sets A, B, C, it holds that $(A \setminus B) \setminus C = (A \setminus C) \setminus B$.

– **One insert and one delete:** We need to show that $S \circ insert(T_1, \alpha_1) \circ delete(R_2) = S \circ delete(R_2) \circ insert(T_1, \alpha_1)$. Applying the definition of the downstream insert and delete operations, let $R_1 = \{(t, \alpha_1) : t \in T_1\}$, we need to verify that $(S \cup R_1) \setminus R_2 = (S \setminus R_2) \cup R_1$. We consider first the case where $R_1 \cap R_2 = \emptyset$:

$(S \cup R_1) \setminus R_2$

$= <A \setminus B = A \cap B^{\complement}>$

$(S \cup R_1) \cap R_2^{\complement}$

$= <\text{Distributivity of } \cap \text{ over } \cup>$

$(S \cap R_2^{\complement}) \cup (R_1 \cap R_2^{\complement})$

$= <A \cup \emptyset = A >$

$(S \cap R_2^{\complement}) \cup ((R_1 \cap R_2^{\complement}) \cup \emptyset)$

$= <\text{Assumption } R_1 \cap R_2 = \emptyset>$

$(S \cap R_2^{\complement}) \cup ((R_1 \cap R_2^{\complement}) \cup (R_1 \cap R_2))$

$= <\text{Distributivity of } \cap \text{ over } \cup>$

$(S \cap R_2^{\complement}) \cup (R_1 \cap (R_2 \cup R_2^{\complement}))$

$= <R_2 \cup R_2^{\complement} = U>$

$(S \cap R_2^{\complement}) \cup (R_1 \cap U)$

$= <\text{Identity element of } \cap>$

$(S \cap R_2^{\complement}) \cup R_1$

$= <A \setminus B = A \cap B^{\complement}>$

$(S \setminus R_2) \cup R_1$

We will make use of this result in subsequent demonstrations under the name of *Lemma 2*: for sets A, B, C, it holds that $B \cap C = \emptyset \Rightarrow (A \cup B) \setminus C = (A \setminus C) \cup B$.

Note that if $R_1 \cap R_2 \neq \emptyset$, we know that there is a causal relation between these two operations, as a consequence of the uniqueness of the pairs $(triple, id)$. Therefore, the causality condition of the network will guarantee that the insert will be delivered and applied before the delete at every other SU-Set.

– **One insert and one delete-insert:** We need to show that $S \circ insert(T_1, \alpha_1) \circ delete - insert(D, I', \alpha_2) = S \circ delete - insert(D, I', \alpha_2) \circ insert(T_1, \alpha_1)$. Applying the specification of downstream insert and delete operation, let $R_1 = \{(t, \alpha_1) : t \in T_1\}$, and $I = \{(t, \alpha_1) : t \in I'\}$, thus, we need to verify that $((S \cup R_1) \setminus D) \cup I = ((S \setminus D) \cup I) \cup R_1$. We start with the case where $R_1 \cap D = \emptyset$:

$((S \cup R_1) \setminus D) \cup I$

$= <R_1 \cap D = \emptyset \wedge \text{ Lemma 2}>$

$((S \setminus D) \cup R_1) \cup I$

$= <\text{Associativity of } \cup>$

$(S \setminus D) \cup (R_1 \cup I)$

$= <\text{Commutativity of } \cup>$

$(S \setminus D) \cup (I \cup R_1)$

$= <\text{Associativity of } \cup>$

$((S \setminus D) \cup I) \cup R_1)$

As in the case of one insert and one delete, if $R_1 \cap D \neq \emptyset$, there is a causal relation between both operations and the network will guarantee that they are delivered in the right order.

– **One delete and one delete-insert:** We need to show that $S \circ delete(R_1) \circ delete - insert(D, I', \alpha_1) = S \circ delete - insert(D, I', \alpha_1) \circ delete(R_1)$. Applying the definition of the downstream delete and delete-insert operations, let $I = \{(t, \alpha_1) : t \in I'\}$, thus, we need

to verify that $((S \setminus R_1) \setminus D) \cup I = ((S \setminus D) \cup I) \setminus R_1$. We start with the case where $R_1 \cap I = \emptyset$:

$((S \setminus D) \cup I) \setminus R_1$

$= <R_1 \cap I = \emptyset \wedge$ Lemma 2$>$

$((S \setminus D) \setminus R_1) \cup I$

$= <$Lemma 1$>$

$((S \setminus R_1) \setminus D) \cup I$

As in the previous two cases, if $R_1 \cap I \neq \emptyset$, there is a causal relation between both operations (the delete-insert occurred before the delete) and the network will guarantee that they are delivered in the right order.

– **Two delete-insert:** We need to show that $S \circ delete{-}insert(D_1, I'_1, \alpha_1) \circ delete{-}insert(D_2, I'_2, \alpha_2) = S \circ delete - insert(D_2, I'_2, \alpha_2) \circ delete - insert(D_1, I'_1, \alpha_1)$. Applying the specification of the downstream delete-insert operation, let $I_1 = \{(t, \alpha_1) : t \in I'_1\}$ and $I_2 = \{(t, \alpha_2) : t \in I'_2\}$ We need to verify that $(((S \setminus D_1) \cup I_1) \setminus D_2) \cup I_2 = (((S \setminus D_2) \cup I_2) \setminus D_1) \cup I_1$. We start with the case where $I_1 \cap D_2 = \emptyset \wedge I_2 \cap D_1 = \emptyset$:

$(((S \setminus D_1) \cup I_1) \setminus D_2) \cup I_2$

$= <I_1 \cap D_2 = \emptyset \wedge$ Lemma 2$>$

$(((S \setminus D_1) \setminus D_2) \cup I_1) \cup I_2$

$= <$Lemma 1$>$

$(((S \setminus D_2) \setminus D_1) \cup I_1) \cup I_2$

$= <$Associativity of $\cup>$

$((S \setminus D_2) \setminus D_1) \cup (I_1 \cup I_2)$

$= <$Commutativity of $\cup>$

$((S \setminus D_2) \setminus D_1) \cup (I_2 \cup I_1)$

$= <$Associativity of $\cup>$

$(((S \setminus D_2) \setminus D_1) \cup I_2) \cup I_1$

$= <I_2 \cap D_1 = \emptyset \wedge$ Lemma 2$>$

$(((S \setminus D_2) \cup I_2) \setminus D_1) \cup I_1$

As with the previous cases, we observe that if $I_1 \cap D_2 \neq \emptyset$ or $I_2 \cap D_1 \neq \emptyset$, there is a causal relation between both operations, and the network will guarantee their delivery in the right order.

8.3 Complexity Analysis

In this section we analyze the complexity of the optimized SU-Set, in order to evaluate the overhead of its implementation on a semantic store. We use as reference case the DBpedia Live system. So far, we have seen that CRDTs' properties allow convergence without the need for reconciling algorithms, which is very efficient in terms of time and messages sent. However, the overhead introduced in space and communication can be high.

We will use the following notation:

- $RWLD$ for the Read/Write Linked Data.
- N, for the number of online participants in $RWLD$.
- N_{max}, for the number of online and offline participants in $RWLD$
- T_p, for the number of triples stored by participant p.
- T_N, for the number of triples in the whole network.
- T_U, for the number of affected triples (inserted and deleted) of an operation U.
- $sizeOf(X)$, for the size in memory of an object X.

8.3.1 Complexity in number of rounds

We measured this dimension as the number of *rounds* of messages needed to exchange between participants to achieve consistency. One round of communication is a sequence of production-sending-reception-execution of operations between the participants. For DBpedia Live the production of the operation is the writing of the change-set file, sending and receiving is the consumption of the file from the site of interest; when re-executed, we arrive to the final state. With SU-Set or its optimized version, there is no difference in the *prepare* phase of the operation except for the size of the change-set file, which will be bigger. The final state is achieved after only one round of communication, which is optimal, and means that independently of the update rate in the system, the time to achieve consistency is only affected by the speed of the network connecting the Semantic Stores.

Notice that in the case of OR-Set, the number of rounds would be equal to the number of triples being operated, as we need to send one message for each one. Duplicate triples do not affect round complexity, they are treated as normal insertions.

Vector Clocks do not introduce any extra number of rounds. Table 8.3.2 summarizes best and worst case complexities.

	Vector Clocks
Best Case	$O(1)$
Worst Case	$O(1)$

Table 8.2: SU-Set's best and worst case round complexity.

8.3.2 Space Complexity

This dimension refers to the disk space used by each strategy. We measured it in terms of extra Gigabytes with respect to a plain RDF-GraphStore.

We assume ids are comprised of two 16 bytes parts, a UUID for unambiguously identify the generating participant and the other to uniquely identify the triple[1], whether it be with another UUID in the case of using vector clocks, or with a monotonically increasing number in the case of interval vectors. If we add 32 extra bytes to each of the 1 billion triples in DBpedia, we will have 29.8 GigaBytes more, which is an acceptable overhead considering that servers currently hosting DBpedia have 3 disks of one Terabyte each.

When many participants insert the same triple at the same "time", *i.e.*, *concurrently*, a different id will be generated for each insertion, yielding many pairs with the same RDF-triple as first coordinate. We define this as "duplicated" RDF-triples.

Definition 8.3.1. *Given a SU-Set with payload S and an RDF-triple t such that $S.lookup(t)$ is true, the* Duplicates *of t in S, noted $Dups(t, S)$, are:*

$$Dups(t, S) = \{(t, i) \mid (t, i) \in S\}$$

Ideally, each RDF-triple stored in the SU-Set will have only one "duplicate". Given the size of an id, we can quantify the overhead with respect to this ideal with the following simple equation.

$$(|S| - |\{t \mid S.lookup(t)\}|) * sizeId$$

Fortunately, our approach allows an easy and safe way to minimize this overhead, as described in Figure 8.2: If site 2 considers that it has too many "duplicates" it deletes the duplicated pairs and inserts a new one, as if the user had executed a Delete-Insert operation deleting and re-inserting the "troublesome" triple. The question of how many duplicates is too many can be answered by each

1. To be more compliant with the Linked Data principles, URIs could be minted using those UUID

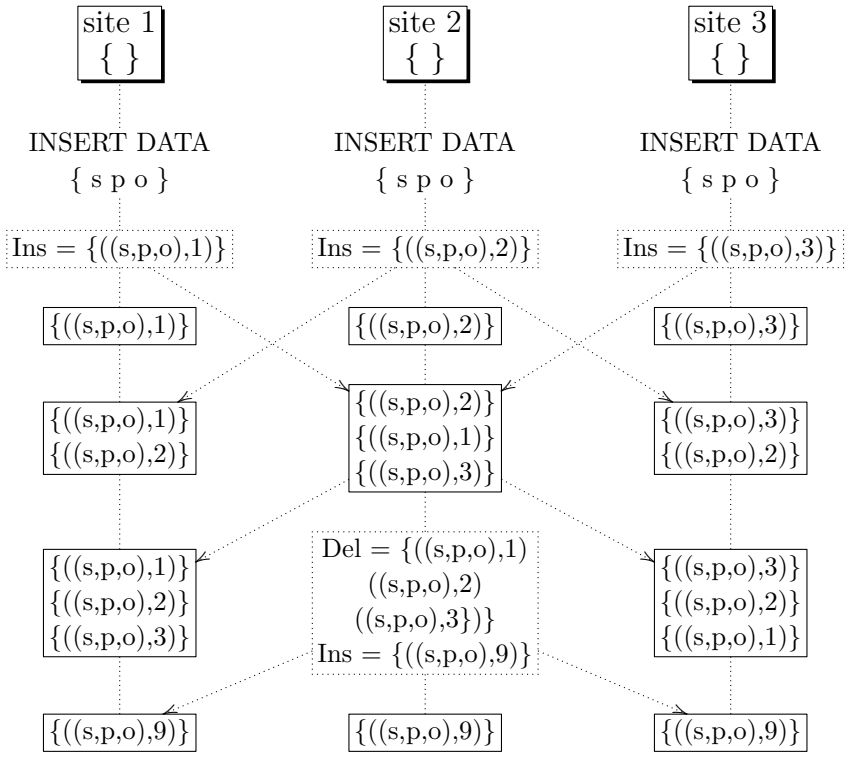


Figure 8.2: Generation and garbage collection of pairs referring to the same triple

participant. A simple implementation when the update rate is not too high is to compute $Dups(t, S)$ each time an RDF-triple is inserted. An alternative when the update rate is too high is to set a disk usage threshold and compute the duplicate sets larger than a fixed value whenever the disk usage threshold is trespassed.

In the worst case, where all participants always add concurrently the same triples, the overhead is $O(N * T_N)$. Based on the probability and frequency of this kind of additions, the frequency of the garbage collection needs to be adjusted. Note also that garbage collecting too often augments the traffic on the network. The use of Vector Clocks introduces an overhead of $O(N_{max})$, each participant needs to store a vector clock, and as the network is strongly connected, every participant will have a vector with one entry for each other participant.

Best Case	$O(T_p) + O(N_{max})$
Worst Case	$O(N * T_p) + O(N_{max})$

Table 8.3: SU-Set's best and worst case space complexity.

8.3.3 Time Complexity

This dimension refers to the overhead in execution time introduced by SU-Set. It has two aspects to consider: first, the extra cost of the CRDT usage, which is $O(T_u)$. In SU-Set, this is done by the participant generating the insertion, while in the optimized version, each receiving node is in charge of constructing the pairs from the received id and the triples. Considering that the cost of an insertion of a set of triples is also linear, we are not adding too much overhead.

The second aspect is the possibility of many pairs having the same triple as the first coordinate to add an overhead to query and update operations. For the querying, there is no extra cost, because it is defined as the existence of a pair with the looked-up triple; for the selection, there is the cost of filtering the extra pairs from the final answer; for the deletion, we could need to delete many pairs instead of only one triple. All these overheads are linear in the number of extra pairs, but as explained in section 8.3.2, we have a way to keep their number under control.

For each processed update, the complexity of checking the delivery condition is linear on the size of the vector, *i.e.*, $O(N_{max})$, then, linear on the number of triples updated. Table 8.4 shows the best and worst casse time complexities for SU-Set.

Best Case	$O(N_{max}) + O(T_u)$
Worst Case	$O(N_{max}) + O(T_u)$

Table 8.4: SU-Set's best and worst case time complexity

8.3.4 Communication Complexity

This dimension refers to the size of the messages exchanged. In the Delete and Insert/Delete operations, the number of pairs $(triple, id)$ sent is $n + d$ where n is the number of triples affected by the operation and d is the number of "extra" pairs for a triple. The number of extra pairs for a given triple t is $|\{(t, id) : (t, id) \in Payload\}| - 1$. Figure 8.3 describes this situation with a Delete-Insert operation.

As shown in subsection 8.3.2, there is a safe way to keep the number of extra pairs for a given triple controlled, by deleting them all and adding a new one with a fresh id, as if a user had deleted it and immediately inserted it again. This will generate $|\{(t, id) : (t, id) \in Payload\}| + 1$ pairs for sending. The maximum number of extra pairs for a triple is the same as the number of participants in the network.

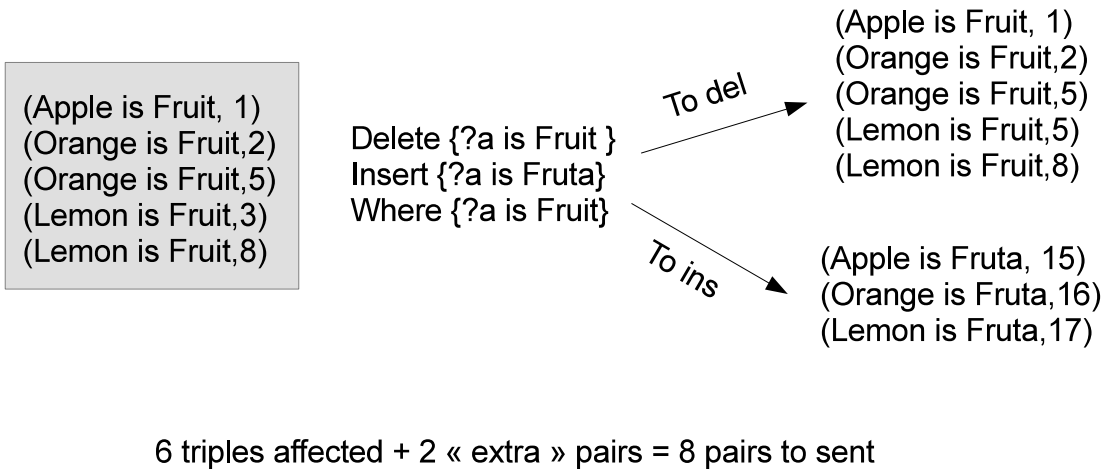


Figure 8.3: Overhead of highly selective pattern operations

As explained in section 8.1, the overhead of sending the ids in the change sets can be greatly reduced by sending only one id per message and letting the receiver construct the pairs. With causal delivery, the same id can be shared by the triples inserted in the same operation [60], as the uniqueness of the element comprised by the triple and its id is maintained.

This means that the saving also depends on how many triples per operation are streamed, in DBpedia Live case, different end being logged in the same change file, or, in SPARQL Update terms, updated with the same DELETE/INSERT operation. For a given time period, the non optimized SU-Set adds an overhead of:

$$(\#triplesInserted + \#pairsDeleted) * sizeOf(id)$$

while the optimized version adds:

$$(\#insertOperations + \#pairsDeleted) * sizeOf(id)$$

Notice that in the deletions we talk about the number of pairs instead of the number of RDF triples, to take into account duplicate triples (cf. Section 8.3.2). Also, if the number of inserts is the same as the number of triples being inserted (one triple inserted at a time), both overheads are equal. Therefore, the “chunk size", or how many affected triples are packed in the same message to send, is relevant to our approach. However, if we pack all triples affected in one day in one big message, message size will be optimized but freshness will be lost.

Figure 8.4 compares the overhead between SU-Set and its optimized version in the DBpedia Live context. The update rate of DBpedia is very high, giving us a good insight of what we can expect under stressed conditions. Considering that for every change file of addition there is one of deletion, we computed the average number of files for each day of the week analyzed in table 8.1, then, we computed the average size of such files per day, for the following cases: before adding the ids (the baseline), after adding the ids to each triple (SU-Set) and after adding only one id per addition file (Optimized SU-Set).

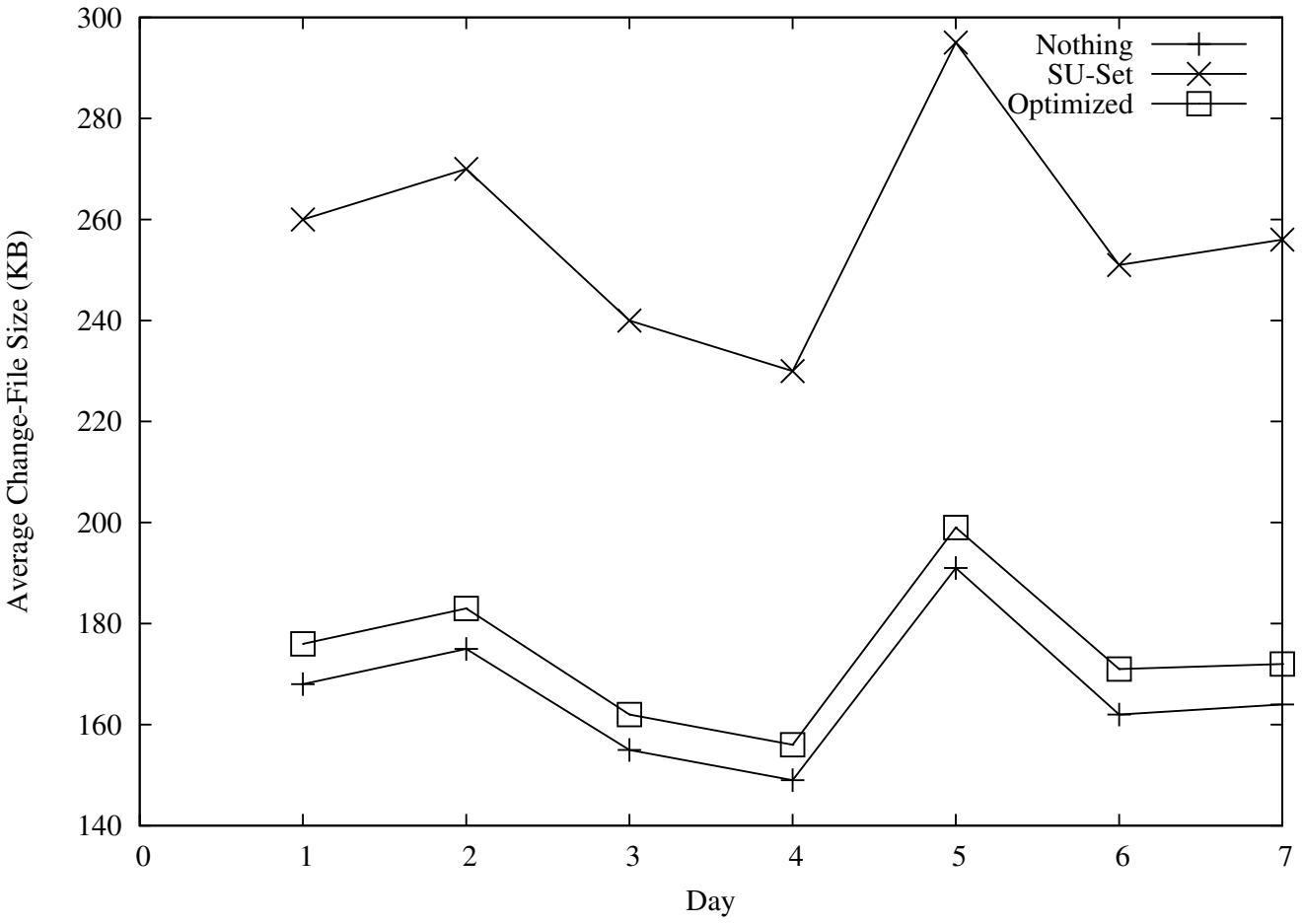


Figure 8.4: Performance in communication for SU-Set and its optimized version in DBpedia Live case.

This difference is due to the fact that in DBpedia Live, the number of triples inserted is more than 12 times the deleted ones, so our optimization is really saving a lot of ids. In other datasets, we could expect many more editions than insertions, and as an edition is a deletion followed by an insertion, the number of triples added and deleted would be roughly the same. In this case, the overhead will depend on the difference between the average triple size and id size, and the chunk size.

Notice that in Figure 8.4 we did not considered the presence of “extra" pairs, which impact the communication complexity of deletions. For the DBpedia Live case, assuming an average of three “extra" pairs per triple would mean that the overhead of optimized SU-Set would go from 5% to 15%. Further studies are needed to confirm if there are enough concurrent insertions (the source of extra pairs) to have a negative impact on performance, or if we would need to set our garbage collection strategy more often.

Concerning the delivery precondition, vector clocks require to piggyback each message with the full vector clock of the participant that made it, yielding a complexity of $O(N_{max})$ per message.

Table 8.5 summarizes the best and worst case communication complexities of optimized SU-Set. Note that a relevant factor is how many triples are affected by each operation, the more triples updated at once, the less messages requiring a piggybacked vector clock will be exchanged. Therefore, in the case of highly dynamic participants, it is possible to pack many updates in one to save bandwidth in exchange of data freshness at the receivers.

Best Case	$O(T_U) + O(N_{max})$
Worst Case	$O(T_U) + O(N_{max})$

Table 8.5: SU-Set ’s best and worst case communication complexity

8.4 Discussion

The use of SU-Set on the Read/Write Linked Data allows to guarantee Strong Eventual Consistency. However, two of the required assumptions: the connectivity of the network and the need to exchange all updates are rather strong, and not aligned with the proposed social vision of the Read/Write Linked Data. For example, if many data consumers copy data from DBpedia, we do not expect neither that DBpedia will consume back all updates from them, nor all data consumers will exchange all update between them.

Nevertheless, if the network is not connected and/or participants are exchanging only subsets of updates, the Strong Convergence part of SEC (def. 7.2.2) property still holds, *i.e.*, the participants that have received and applied the same updates have equivalent state. More formally, if we substitute the Eventual Delivery condition in definition 7.2.1 by the following:

Definition 8.4.1 (Network reliability)**.** *All messages eventually arrive to their intended receiver.*

Strong Convergence still holds.

However, to only have the guarantee of full equivalence for participants that have applied the same updates is too weak, as only in very few cases two participants will consume and apply the same updates. In chapter 9, we develop a consistency criterion stronger than SEC, focused on guaranteeing the consistency of the fragments of data copied from other participants.

8.5 Conclusion and perspectives

In this chapter we presented SU-Set, a CRDT for RDF-Graphs operated with SPARQL 1.1 Update operations. SU-Set solves the consistency problem for the Read/Write Linked Data, guaranteeing Strong Eventual Consistency when the network is connected and all stores consume all updates, otherwise, it assures Strong Convergence: stores that have received the same updates have equivalent state.

Complying with the requirements of the problem stated in definition 6.3, SU-Set does not require consensus, does not introduce any single point of failure and has linear complexity in time, space and communication. SU-Set's highest price to pay is in space, the worst case being when all N participants insert concurrently the same triple, generating N duplicates of the triple. Nevertheless, we provide an algorithm to erase duplicates in exchange of augmenting the amount of traffic in the network.

SU-Set has been implemented into the SPARQL engine Corese [2] and is freely available at `https://code.google.com/p/live-linked-data/`.

Perspectives include:

– The study of improvements that minimize the needed causality tracking information, perhaps using probabilistic methods that miss a causal relationship with a negligible probability.

2. `http://wimmics.inria.fr/corese`

- An experimental study on a grid platform or a simulation of the performance of SU-Set under different dynamics conditions to estimate the average case complexity.
- The development of a protocol that allows to discover the subset of participants that have equivalent state in the network.
- Look for a stronger criterion that takes in account the particular fragments of data being copied. We develop this idea in chapter 9.

9

Fragment Consistency and Col-Graph

In this chapter, we give a second solution to the problem of consistency in Read/Write Linked Data. We propose a criterion stronger than SEC to be able to assert some guarantees in the case of participants receiving only subsets of updates, and an update exchange algorithm to maintain it named Col-Graph. We show that Col-Graph's complexity is the same than the solution described in chapter 8 except in space, where is much higher in the worst case but only slightly higher in the best and average cases. Nevertheless, experiments suggest that in the case of synthetically generated social networks, the overhead is much less than in randomly generated ones.

The main limitation of the solution presented in chapter 8 is that, despite its low complexity, SEC may be too weak for the Read/Write Linked Data. Imagine a participant that wants to perform data cleansing on a subset of DBpedia, e.g., the triples having as subject the entity *DBpedia:France*. Copying the entire DBpedia is a waste of resources, thus, the participant copies only the *fragment* of data he is interested to, and receives only the updates from DBpedia that concern such fragment. With SU-Set, there is nothing we can assert on the consistency of such fragment, because both participants have not applied the same updates. In order to support this use case, we propose *Fragment Consistency*, a criterion focused on the consistency of the fragments of data copied instead of on the state equivalence of all participants

9.1 Fragment Consistency

Definition 9.1.1 (Fragment)**.** *Let S be a SPARQL endpoint of a participant, a fragment of the RDF-Graph made accessible by S, $F[S]$, is a SPARQL CONSTRUCT federated query [123] where all graph patterns are contained in a single SERVICE block with S as the remote endpoint. We denote as $eval(F[S])$ the RDF-Graph result of the evaluation of $F[S]$.*

We specialize the *Consume Updates* tie of the Read/Write Linked Data (Definition 6.2.6) to use definition 9.1.1.

Definition 9.1.2 (Concerns)**.** *Let U be an update and GP a Graph Pattern. The subset of U that concerns GP, $Concern(U, GP)$, is an update constructed as follows:*

1. *Its del-graph is the subset of triples in U's del-graph that match GP.*
2. *Its add-graph is the subset of triples in U's add-graph that match GP.*
3. *Its metadata graph is the same as U's, plus the fact that this is a subset of another update.*

Definition 9.1.3. *Let S be an RDF-GraphStore, we say that an RDF-GraphStore T* Materializes a Fragment of *S, if T evaluates $F[S]$ and unions the result with its own data. The subset of updates made by S delivered to T are the ones that concern $F[S]$. We call S the* source *of the fragment and T its* target.

Figure 9.1 illustrates how updates are propagated on Read/Write Linked Data using fragments. $P1$ starts with data about the *nationality* and *KnownFor* properties of M_Perey (prefixes are omitted for readability). $P2$ materializes from $P1$ all triples with the *knownFor* property. With this information and its current data, $P2$ inserts the fact that M_Perey discovered Francium. On the other hand, $P3$ materializes from $P1$ all triples with the *nationality* property. $P3$ detects a mistake (nationality should be $French$, not $French\_People$) and promptly corrects it. $P4$ constructed a dataset materializing from $P2$ the fragment of triples with the property *discoverer* the fragment of triples with the property *nationality* from $P3$. $P1$ trusts $P4$ about data related to M_Perey, so she materializes the relevant fragment, indirectly consuming updates done by $P2$ and $P3$.

Triples updated on materialized fragments are not necessarily integrated by the source, *e.g*, the deletion done by $P3$ did not reach $P1$, therefore, equivalence between source and materialized fragment cannot be used as consistency criterion. Intuitively, each materialized fragment must be equal

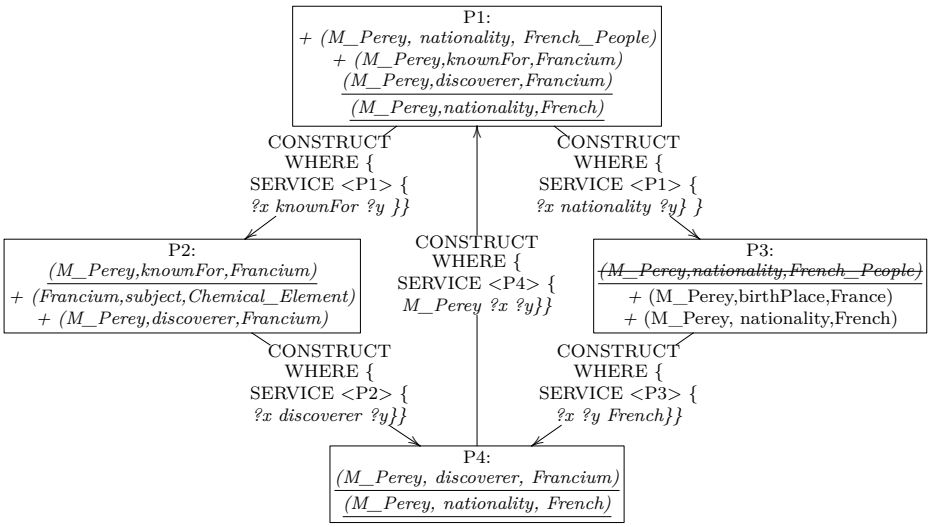


Figure 9.1: Read/Write Linked Data with Fragments. Underlined triples are the ones coming from fragments, triples preceded by a '+' are the ones locally inserted, struck-through triples are the ones locally deleted.

to the evaluation of the fragment at the source after applying *local* updates, *i.e.*, the ones executed by the participant itself and the ones executed during synchronization with other fragments.

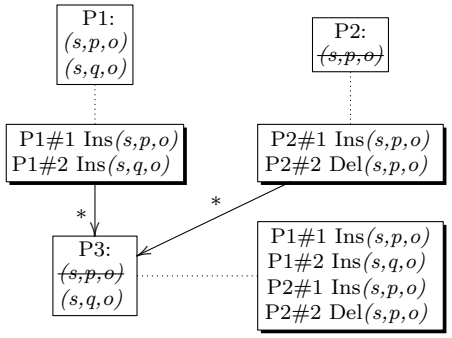
Definition 9.1.4 (Fragment Consistency)**.** *Let $RWLD = (P, E)$ be the Read/Write Linked Data. Assume each $P_i \in P$ maintains a sequence of uniquely identified updates Δ_{P_i} with its local updates and the updates it has consumed from the sources of the fragments $F[P_j]@P_i$ it materializes. Given a Δ_P, let $\Delta_P^{F[S]}$ be the ordered subset of Δ_P such that all updates* concern *$F[S]$,* i.e.*, that match the graph pattern in $F[S]$. Let $apply(P_i, \Delta)$ be a function that applies a sequence of updates Δ on P_i.*

$RWLD$ is consistent *iff when the system is idle, i.e., no participant executes local updates or fragment synchronization, then:*

$$(\forall P_i, P_j \in P : F[P_i]@P_j = apply(eval(F[P_i]), \Delta_{P_j}^{F[P_i]} \setminus \Delta_{P_i})$$

The $\Delta_{P_j}^{F[P_i]} \setminus \Delta_{P_i}$ term formalises the intuition that we need to consider only local updates when evaluating the consistency of each fragment, *i.e.*, from the updates concerning the fragment, remove the ones coming from the source.

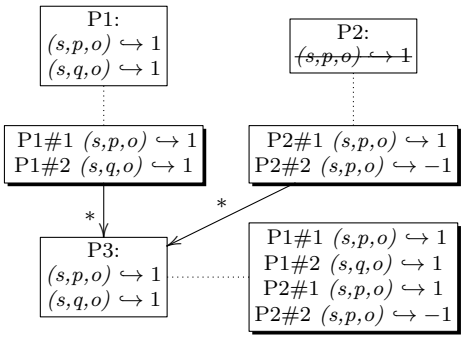
Unfortunately, applying remote operations as they come does not always comply with Definition 9.1.4 as shown in Figure 9.2a: $P3$ synchronizes with $P1$, applying the updates identified as $P1\#1$ and $P1\#2$, then with $P2$, applying the updates identified as $P2\#1$ and $P2\#2$, however, the fragment materialized from $P2$ is not consistent. Notice that, had $P3$ synchronized with $P2$ before



$F[P1]@P3 = apply(\{(s,p,o),(s,q,o)\}, (P2\#1\ Ins(s,p,o), P2\#2\ Del(s,p,o)))$

$F[P2]@P3 \neq apply(\emptyset, (P1\#1\ Ins(s,p,o))\ P1\#2\ Ins(s,q,o)))$

(a) Applying updates as they come does not comply with the correction criterion.



$F[P1]@P3 = apply(\{(s,p,o) \hookrightarrow 1, (s,q,o) \hookrightarrow 1\}, (P2\#1\,(s,p,o) \hookrightarrow 1\ (P2\#2\,(s,p,o) \hookrightarrow -1))$

$F[P2]@P3 = apply(\emptyset, (P1\#1\,(s,p,o) \hookrightarrow 1, P1\#2\,(s,q,o) \hookrightarrow 1))$

(b) The Annotated RDF-Graph enables a consistent Collaboration Network

Figure 9.2: Illustration of Fragment Consistency. Plain boxes represent RDF-Graphs, shaded boxes simplified sequences of updates. $*$ represents a full fragment.

than with $P1$, its final state would be different ((s,p,o) would exist) and the fragment materialized from $P1$ would not be consistent.

SU-Set, the algorithm described in section 8.1 also cannot achieve Fragment Consistency due to its requirement of causal delivery of updates. Suppose a fragment of P_1 materialized at P_2 $F[P_1]@P_2$, and suppose that P_1 performs two updates, u_1 not concerning F and u_2 concerning F. In our model, u_1 will not be delivered to P_2, meaning that when u_2 is delivered, it will be considered as not causally ready to be executed and put on hold indefinitely.

9.2 Col-Graph: A Protocol for Synchronization of Basic Fragments

To achieve Fragment Consistency, we propose, in the spirit of [44], to count the number of insertions and deletions of a triple, *i.e.,* we annotate each RDF-triple with positive or negative integers, positive values indicate insertions and negative values deletions. This allows for a uniform representation of data and updates, yielding a simple way to synchronize fragments.

Incrementally synchronizing a materialized fragment using only the updates published by a data source and the locally materialized fragment without issuing another query on the data source requires to exclude join conditions from fragments [48], therefore, to not compromise the availability of sources, we restrict to *basic fragments* [120], *i.e.,* fragments where the query is comprised by only one triple

pattern.

Definition 9.2.1 (Annotated RDF-triple, Graph and Update). *1. Let t be an RDF-triple and $z \in \mathbb{Z}^*$. $t \hookrightarrow z$ is an* annotated RDF-triple*, t is called the* triple *and z the* annotation.

2. An annotated RDF-Graph *G^A is a set of annotated RDF-triples such that $(\forall t, z | t \hookrightarrow z \in G^A : z > 0)$*

3. An annotated update *u^A is represented by an annotated RDF-triple. More precisely, $t \hookrightarrow 1$ for insertion of t and $t \hookrightarrow -1$ for deletion of t.*

Annotations in RDF-Graphs count the number of *derivations* of a triple in the *RWLD*.

Definition 9.2.2 (Derivation). *Let t be a triple stored in a participant RWLD P_0, a* derivation *of t is a simple path from the participant that inserted t, P_1, and P_0, such that the insertion of t concerns each edge of the path.*

An annotation value higher than 1 indicates that the triple exists in more than one source or there are several simple paths in *RWLD* leading from the participant that inserted the triple to the participant. Annotations in updates indicate, if positive, that z derivations of t were inserted; if negative, that z derivations of t were deleted. For example, an annotated RDF-triple $t_1 \hookrightarrow 2$ means that either t_1 has been inserted by two different sources or the same insert arrived through two different paths in *RWLD*. The annotated update $t_2 \hookrightarrow -1$ means that t_2 was deleted at one source or by some participant in the path between the source and the target; $t_3 \hookrightarrow -2$ means that either t_3 was deleted by two sources or by some participant in the path between two sources and the target.

To apply annotated updates to annotated RDF-Graphs, we define an *Update Integration* function:

Definition 9.2.3 (Update Integration). *Let A be the set of all annotated RDF-Graphs and B the set of all annotated updates. Assume updates arrive and are executed from source to target in FIFO order. The* Update Integration *function $\uplus : A \times B \to A$ takes an annotated RDF-Graph $G^A \in A$ and an annotated update $t \hookrightarrow z \in B$:*

$$G^A \uplus t \hookrightarrow z = \begin{cases} G^A \cup \{t \hookrightarrow z\} & if\ (\nexists w : t \hookrightarrow w \in G^A) \\ G^A \setminus \{t \hookrightarrow w\} & if\ t \hookrightarrow w \in G^A \wedge w + z \leq 0 \\ (G^A \setminus \{t \hookrightarrow w\}) \cup \{t \hookrightarrow w + z\} & if\ t \hookrightarrow w \in G^A \wedge w + z > 0 \end{cases}$$

The first piece of the Update Integration function handles incoming updates of triples that are not in the current state. As we are assuming FIFO in the update propagation from source to target, insertions always arrive before corresponding deletions, therefore, this case only handles insertions. The second piece handles deletions: only if the incoming deletion makes the annotation zero the triple is deleted from the current state. The third piece handles deletions that do not make the annotation zero and insertions of already existing triples by simply updating the annotation value.

We now consider each participant has an annotated RDF-Graph G^A and an sequence of annotated updates U^A. SPARQL queries are evaluated on the RDF-Graph $\{t \,|\, t \hookrightarrow z \in G^A\}$. SPARQL Updates are also evaluated this way, but their effect is translated to annotated RDF-Graphs as follows: the insertion of t to the insertion of $t \hookrightarrow 1$ and the deletion of t to the deletion of the annotated triple having t as first coordinate. Specification 9.1 details the methods to insert/delete triples and synchronize materialized fragments. To avoid the infinite forwarding of updates, each time an update is processed, the protocol checks if it has walked a cycle, if so, it is ignored. Figure 9.2b shows the fragment synchronization algorithm in action.

To materialize fragments for the first time, a SPARQL extension that allows to query the annotated RDF-Graph and return the triples *and* their annotations is needed, for example the one implemented in [131]. To check when an update has cycled, we propose to add a second annotation to updates, containing a set of participant identifiers ϕ_u representing the participants that have already received and applied the update. When an update u is created, ϕ_u is set to the singleton containing the ID of the author, when u is pushed downstream, the receiving participant checks if his ID is in ϕ_u, if yes, u has already been received and is ignored, else, it is integrated, and before pushing it downstream it adds its ID to ϕ_u. Of course, there is a price to pay in traffic, as the use of ϕ increases the size of the update. The length of ϕ_u is bounded by the length of the longest simple path in the Collaboration-Network, which in turn is bounded by the number of participants.

After the publication of [62], we found that the check for cycling introduces a limitation in the topologies that our algorithm can handle: Only network graphs where for each strongly connected component, the number of paths between each pair of nodes belonging to the component is the same, *i.e.*, simple cycles and complete subgraphs. The problem is that deletions performed by a node that is not the author of the triple may be ignored when they need to be applied. Figure 9.3 illustrates

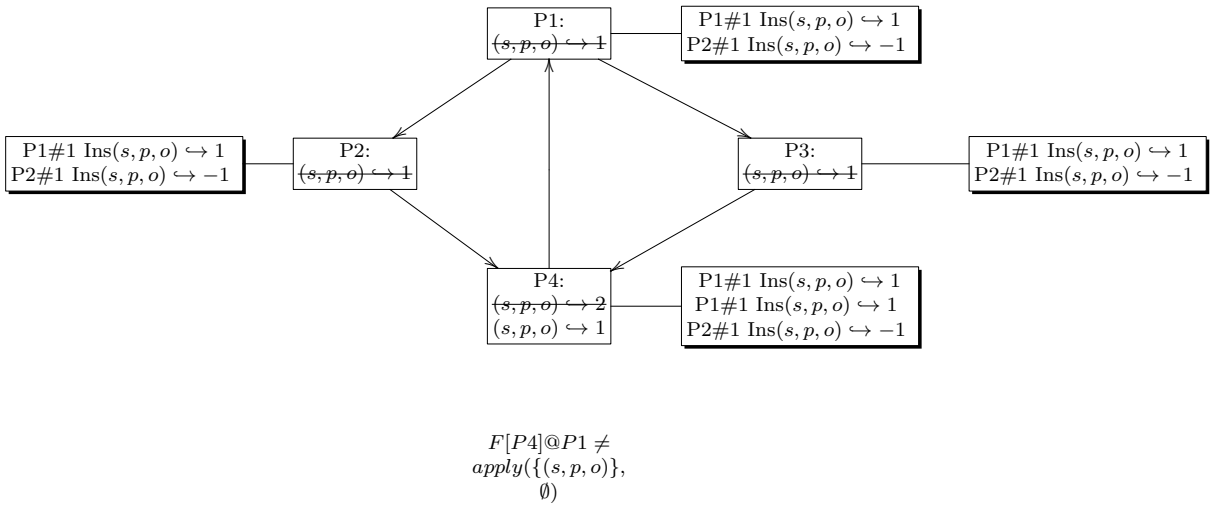


Figure 9.3: In certain Read/Write Linked Data topologies, deletions may be incorrently ignored

the issue: $P1$ inserts (s, p, o) and $P2$ deletes it, the deletion walks through $P4, P1, P3$ and is ignored because a cycle was detected. In this case, the deletion must have been applied and the fragment $F[P4]@P1$ is not consistent.

Fortunately, the issue described in the previous section can be solved if we make the deletions stop when they do not affect the current state instead of when cycles are detected, in a similar way to the fixpoint semantics of datalog. Specification 9.3 shows the modified version of the algorithm. Figure 9.4 illustrates how this versions fixes the problem. The core of the fix lies on the *sync* procedure, the check for cyclic updates is only done for insertions (line 17). For deletions the stop condition is that the triple is not anymore there.

9.2.1 Provenance for Conflict Resolution

In section 9.2 we solved the problem of consistent synchronization of basic fragments. However, Fragment Consistency is based on the mere existence of triples, instead of on the possible conflicts between triples coming from different fragments and the ones locally inserted. Col-Graph's strategy in this case is that each participant is responsible for checking the semantic correctness of its dataset, as criteria often varies and what is semantically wrong for one participant, could be right for another. Participants can delete/insert triples to fix what they consider wrong. Participants that receive these updates can edit in turn if they do not agree with them.

In the event that a participant wants to choose between two triples, the main criteria to choose which one of them delete is their *provenance*. With this information, the decision can be made based on the trust on their authors. As in [69], we propose to substitute the integer annotations of the

```
1  Annotated Graph G^A,
2  Sequence ΔP_ID
3
4  void insert(t):
5      pre: t ∉ {t'|t ↪ x ∈ G^A}
6      G^A := G^A ∪ t ↪ 1
7      Append(ΔP_ID, t ↪ 1)
8
9  void delete(t):
10     pre: t ∈ {t'|t' ↪ x ∈ G^A}
11     G^A := G^A ⊎ t ↪ −z
12     Append(ΔP_ID, t ↪ −z)
13
14 void sync(F[P_x], ΔP_x):
15     for t ↪ z ∈ ΔP_x:
16         if t ↪ z has not cycled:
17             G^A := G^A ⊎ t ↪ z
18             Append(ΔP_ID, t ↪ z)
```

Specification 9.1: Class Participant when triples are annotated with elements of Z.

```
1  IRI P_ID,
2  Annotated Graph G^A,
3  Sequence ΔP_ID
4
5  void insert(t):
6      pre: t ∉ {t'|t ↪ x ∈ G^A}
7      G^A := G^A ∪ t ↪ P_ID
8      Append(ΔP_ID, t ↪ P_ID)
9
10 void delete(t):
11     pre: t ∈ {t'|t' ↪ x ∈ G^A}
12     G^A := G^A ⊎ t ↪ −m
13     Append(ΔP_ID, t ↪ −m)
14
15 void sync(F[P_x], ΔP_x):
16     for t ↪ m ∈ ΔP_x :
17         if t ↪ m has not cycled:
18             G^A := G^A ⊎ t ↪ m
19             Append(ΔP_ID, t ↪ m)
```

Specification 9.2: Class Participant when triples are annotated with elements of the monoid M.

triple by an element of a commutative monoid that embeds $(Z, +, 0)$.

Definition 9.2.4 (Commutative Monoid)**.** *A Commutative Monoid is is an algebraic structure comprised by a set K, a binary, associative, commutative operation $\oplus$ and an identity element $0_K \in K$ such that*

$$(\forall k \in K \mid k \oplus 0_K = k)$$

Definition 9.2.5 (Embedding)**.** *A monoid $M = (K, \oplus, 0_K)$ embeds another monoid $M' = (K', \oslash, 0_{K'})$ iff there is a map $f : K \to K'$ called* homomorphism *such that $f(0_K) = f(0_{K'})$ and $(\forall a, b \in K : f(a \oplus b) = f(a) \oslash f(b))$.*

If we annotate with elements of a monoid that embeds $(Z, +, 0)$, all the properties of our synchronization algorithm maintain. Formally, the semantics of the querying commutes with the application of the homomorphism, a fundamental theorem proved in [45] for the more general case of rings instead of monoids. The use of symbolic expressions that can be morphed to the basic $(Z, +, 0)$ allows the encoding of useful information, for instance, the provenance of triples.

Definition 9.2.6. *Assume each participant in the RWLD has an unique ID, and let X be the set of all IDs. Let $M = (Z[X], \oplus, 0)$ be a monoid with:*

```
1  Annotated Graph G^A,
2  Sequence ΔP_ID
3
4  void insert(t):
5       pre: t ∉ {t'|t ↪ x ∈ G^A}
6       G^A := G^A ∪ t ↪ 1
7       Append(ΔP_ID, t ↪ 1)
8
9  void delete(t):
10      pre: t ∈ {t'|t' ↪ x ∈ G^A}
11      G^A := G^A ⊎ t ↪ −z
12      Append(ΔP_ID, t ↪ −z)
13
14 void sync(F[P_x], ΔP_x):
15      for t ↪ z ∈ ΔP_x:
16          if z > 0:
17              if t ↪ z has not cycled:
18                  G^A := G^A ⊎ t ↪ z
19                  Append(ΔP_ID, t ↪ z)
20          if z < 0:
21              if G^A ⊎ t ↪ z ≠ G^A:
22                  G^A := G^A ⊎ t ↪ z
23                  Append(ΔP_ID, t ↪ z)
```

Specification 9.3: Class Participant with Z annotations modified for all topologies

```
1  IRI P_ID,
2  Annotated Graph G^A,
3  Sequence ΔP_ID
4
5  void insert(t):
6       pre: t ∉ {t'|t ↪ x ∈ G^A}
7       G^A := G^A ∪ t ↪ P_ID
8       Append(ΔP_ID, t ↪ P_ID)
9
10 void delete(t):
11      pre: t ∈ {t'|t' ↪ x ∈ G^A}
12      G^A := G^A ⊎ t ↪ −m
13      Append(ΔP_ID, t ↪ −m)
14
15 void sync(F[P_x], ΔP_x):
16      for t ↪ m ∈ ΔP_x :
17          if m > 0:
18              if t ↪ m has not cycled:
19                  G^A := G^A ⊎ t ↪ m
20                  Append(ΔP_ID, t ↪ m)
21          if m < 0:
22              if G^A ⊎ t ↪ m ≠ G^A:
23                  G^A := G^A ⊎ t ↪ m
24                  Append(ΔP_ID, t ↪ m)
```

Specification 9.4: Class participant with M annotations modified for all topologies

1. *The identity* 0.
2. *The set* $Z[X]$ *of polynomials with coefficients in* Z *and variable in* X.
3. *The polynomial sum* $\oplus$*, for each monomial with the same indeterminate:* $aX \oplus bX = (a+b)X$
4. M *embeds* $(Z, +, 0)$ *through the function* $f(a_1X_1 \oplus \cdots \oplus a_nX_n) = \sum_1^n a_i$

Each time a participant inserts a triple, she annotates it with its ID with coefficient 1. The only change in definition 9.2.3 is the use of $\oplus$ instead of $+$. Specifications 9.2 and 9.4 describes the algorithm to insert/delete triples and synchronize fragments with triples annotated with elements of M.

When annotating with Z, the only information encoded in triples is their number of derivations. M adds (i) Which participant is the *author* of the triple. A triple stored by a participant P with an annotation comprised of the sum of n monomials indicates that the triple was inserted *concurrently* by n participants from which there is a path in CN to P. (ii) The number of simple paths in the Collaboration Network in which all edges concern the triple, starting from the author(s) of the triple

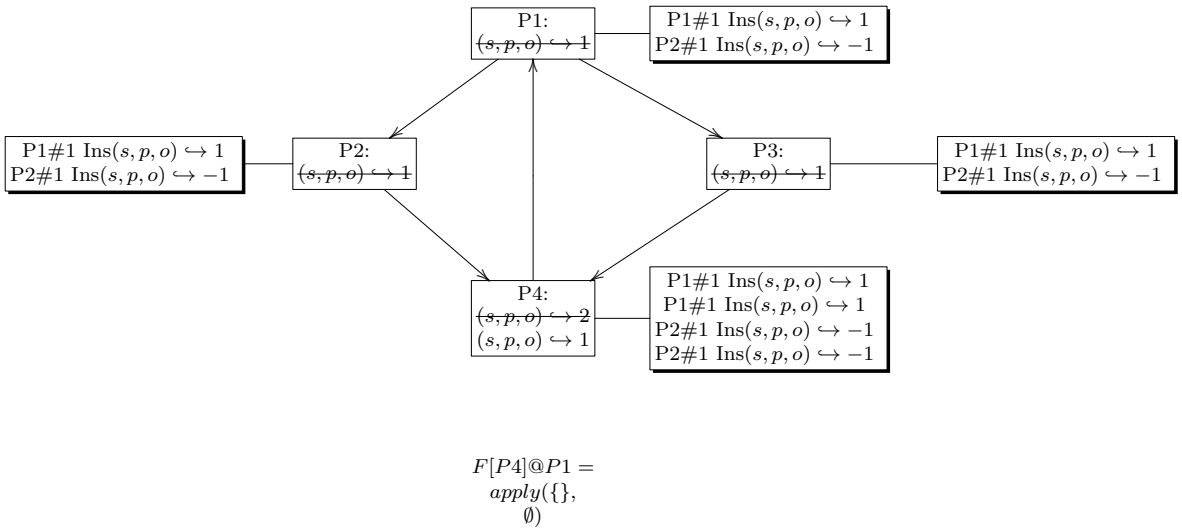
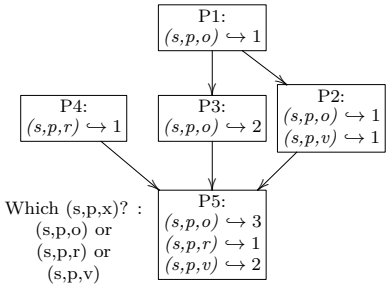
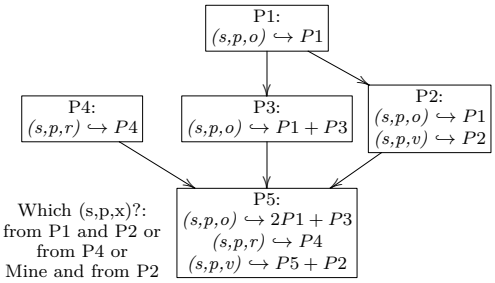


Figure 9.4: Iterating deletions until no effect allows support for any network topology



(a) Without provenance, P5 only information is the number of derivations. She does not know the author of the facts.



(b) With provenance, P5 also knows who inserted what and if it was concurrent, enabling trust based decisions to solve conflicts.

Figure 9.5: Difference between annotating with Z (9.5a) versus annotating with M (9.5b).

to this participant, indicated by the coefficient of the author's ID.

Figure 9.5 compares annotations with Z versus annotations with M. In the depicted collaboration network, the fact *(s,p,o)* is inserted concurrently by P1 and P3, *(s,p,v)* is inserted concurrently by P2 and P5 and *(s,p,r)* inserted only by P4. When the synchronization is finished, P5 notices that it has three triples with s and p as subject and predicate but different object values. If P5 wants to keep only one of such triples based on trust, the Z annotations (9.5a) do not give enough information, while the M annotations (9.5b) give more information for $P5$ to take the right decision. She can know that the triple (s, p, o) was inserted by two participants $P1$ and $P3$, while (s, p, r) was only inserted by $P4$ and that (s, p, v) was inserted by $P2$ and $P5$.

9.3 Complexity Analysis

In this section, we analyze the complexity in time, space and traffic of Col-Graph on RDF-Graphs annotated with Z and M to answer the question: *how much does it cost to guarantee Fragment Consistency on the Read/Write Linked Data?*.

We will use the following notation:

- $RWLD$ for the Read/Write Linked Data.
- N, for the number of participants currently online in $RWLD$.
- N_{max}, for the number of participants in $RWLD$.
- T_p, for the number of triples stored in a participant p.
- T_N, for the number of triples stored in all the $RWLD$.
- T_U, for the number of triples updated (inserted or deleted) by an update operation U.
- $sizeOf(X)$, for the size in memory of an annotation X.

9.3.1 Time Complexity

From specifications 9.1 and 9.2, we can see that time complexity for the insert and delete methods is constant. For the synchronization of a fragment $F[P_x]@P_y$, the complexity is $T_U * (x_1 + x_2)$ where x_1 is the complexity of checking if an update is in the log ΔP_y (which can be considered linear on the size of the log) and x_2 the complexity of the $\uplus$ function. For Z annotations, $\uplus$ is constant, for M is linear on the size of the longest polynomial. As with SU-Set, the amount of concurrent insertions impacts the approach, in this case, the maximum number of terms a polynomial can have is N_{max}, when all participants inserted concurrently the same triple, there is a path from all participants to one participant such that the insertion of the triple concerns all fragments.

Table 9.1 summarizes Col-Graph's best and worst case complexities for Z and M annotations.

	Z annotations	M annotations
Best Case	$O(T_U) + O(T_U)$	$O(T_U) + O(T_U)$
Worst Case	$O(T_U) + O(T_U)$	$O(T_U) + O(T_U * N_{max})$

Table 9.1: Best and worst case time complexities for Col-Graph for Z and M annotations.

9.3.2 Space Complexity

Concerning space complexity, the overhead is the size of the annotations. For an annotated triple t at a participant P, the relevant factors are: (i) the set of participants that concurrently inserted t from which there is a path to P such that all edges concern t, that we will denote β_t (ii) the number of simple paths to P in the collaboration network from the participants $P_1 \dots P_n$ that concurrently inserted t such that all edges concern t. For a participant P_i, we denote this number as $\rho_{t \leftarrow P_i}$. Assume that the cost of storing ids is a constant ω. Then, for $t \hookrightarrow z, z \in Z[x]$ we have $sizeOf(z) = |\beta_t|\omega + \sum_{P_i \in \beta_t} sizeOf(\rho_{t \leftarrow P_i})$. Therefore, for each triple we need to keep a hash map from ids to integers of size $|\beta_t|$. The worst case for $|\beta_t|$ is when the network is *strongly connected* and all participants insert t concurrently, yielding an array of size N_{max}. The worst case for $\rho_{t \leftarrow P_i}$ is a *complete network*, as the number of different simple paths is maximal and in the order of $N_{max}!$

Table 9.2 summarizes the best and worst case space complexities for Z and M annotations. We add the following observations about the worst cases: (i) In the case of Z, as the counter equals to the sum of the number of paths and the number of concurrent insertions, the factorial grows faster (by a polynomial) than in the case of M. (ii) In both Z and M the factorial term of the complexity represents the storage of a number or of an array of numbers that can attain *values* in the order of $N_{max}!$, as such, the practical impact on storage can be limited by using BigInt arithmetics.

	Z annotations	M annotations
Best Case	$O(T_p)$	$O(T_p)$
Worst Case	$O(T_p * sizeOf(N_{max}!))$	$O(T_p * N_{max} * sizeOf(N_{max}!))$

Table 9.2: Best and worst case space complexities for Col-Graph for Z and M annotations.

9.3.3 Communication and traffic complexity

In terms of the number of messages exchanged Col-Graph is optimal, only the messages with the update from each source are required to guarantee fragment consistency, without any extra round of communication.

In terms of message size, there are some differences between Z and M annotations and between insertions an deletions. For both Z and M the best case (besides a completely disconnected network) is when no concurrent insertions occur and there are few paths between participants. The worst case is a network forming a *Complete Graph* where all participants insert concurrently the same triples.

- In the case of Z annotations, in the best case the annotation can be considered having the constant value of storing an integer number. In the worst case, the annotation can attain a factorial value that will need to be transmitted in the case of deletion. Note that for insertions, the transmitted value is constant and equal to 1.
- In the case of M annotations, the best case is the same as with Z. In the worst case, what we have is for each triple an array of size N_{max} with values in the order of factorial. Note that due to the connection between Z and M through the morphism defined in definition 9.2.6, the sum of the values in the array of an M annotation equals to the value of the Z annotation. This means that the value of the Z annotation grows faster than each of the individual values of the array of the M annotation.
- Note that in both cases, the factorial is attained by a *value* or an array of values, therefore, the real impact on space is lowered. For instance, a C++ signed long long of 64 bytes can hold values up to 9223372036854775807, *i.e.*, up to that amount of paths between the author of a triple and a participant.

A $O(N_{max})$ factor needs to be added to account for the cost of the cycle check in the worst case: when the network has a hamiltonian circuit. Concerning the maintenance of a point-to-point FIFO channel between sources and targets, there is only need for a sequence number, *i.e.*, a constant factor.

Table 9.3 summarizes Col-Graph's best and worst case communication complexities for Z and M annotations.

	Z annotations	M annotations
Best Case	$O(T_U)$	$O(T_U)$
Worst Case	$O(T_U * N_{max}!) + O(N_{max})$	$O(T_U * N_{max} * N_{max}!) + O(N_{max})$

Table 9.3: Best and worst case communication complexities for Col-Graph for Z and M annotations.

9.3.4 Summary

To summarize, Col-Graph's performance is mainly affected by the following properties of the RWLD:

- The probability of concurrent insertion of the same data by many participants. The higher this probability, the number of terms of the polynomials is potentially higher.
- Its *connectivity*. The more connected, the more paths between the participants and the potential values of ρ are higher. If the network is poorly connected, few updates will be consumed

and the effects of concurrent insertion are minimized.

– The *overlapping* between fragments. If all fragments copy all data, all incoming updates will be integrated by every participant, maximizing the effects of connectivity and concurrent insertion. If all fragments are disjoint, then all updates will be integrated only once and the effects of connectivity and concurrent insertion will be neutralized.

Table 9.4 compares the complexities of the SU-Set version with Interval Vectors and Col-Graph with M annotations. Col-Graph is less expensive in terms of communication but potentially more expensive in space with a rather high factorial worst case complexity.

	SU-Set best case	SU-Set worst case	Col-Graph best case	Col-Graph worst case
Rounds	1	1	1	1
Communication	$O(\#messages)$	$O(T_U)$	$O(T_U) + O(N_{max})$	$O(T_U * N_{max} * N_{max}!)$
Time	$O(T_U)$	$O(T_U)$	$O(T_U * \|\Delta P_y\|)$ $+O(T_U)$	$O(T_U * \|\Delta P_y\|)+$ $O(T_U * N_{max})$
Space	$O(T_p) + O(N_{max})$	$O(N_{max} * T_p)+$ $O(N_{max} * T_p)$	$O(T_p)$	$O(T_p * N\{max\}$ $+N\{max\}!)$

Table 9.4: Comparison of SU-Set and Col-Graph complexities

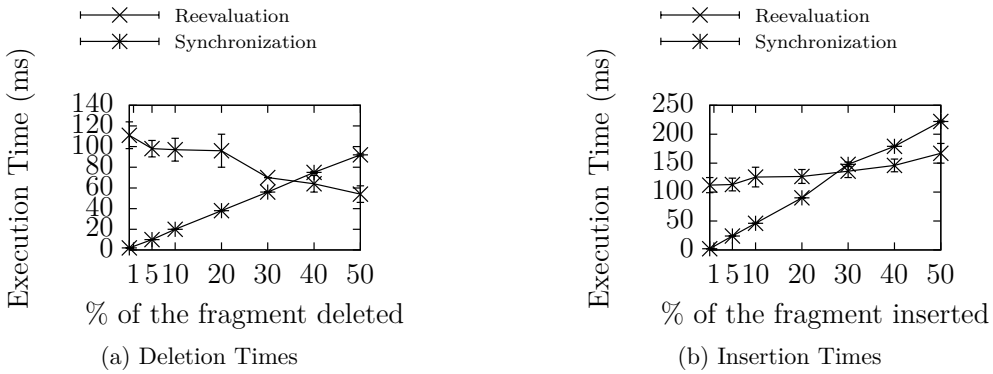
9.4 Experimentations

In this section, we make an empirical evaluation of Col-Graph with the goal of experimentally confirm the theoretical complexities and to estimate the complexity in the average case. The main objective is to know if the potential factorial complexity in space is a concern for average cases.

We implemented specification 9.2 on top of the SPARQL engine Corese [1] v3.1.1. The update log was implemented as a list of updates stored in the file system. We also implemented the ϕ annotation described in section 9.3 to check for double reception. We constructed a test dataset of 49999 triples by querying the DBpedia 3.9 public endpoint for all triples having as object the resource `http://dbpedia.org/resource/France`. Implementation, test dataset, and instructions to reproduce the experiments are freely available [2].

Our first experiment studies the execution time of our synchronization algorithm. The goal is to confirm the linear complexity derived in section 9.3 and to check its cost w.r.t fragment re-evaluation. We defined a basic fragment with the triple pattern *?x :ontology/birthPlace ?z* (7972 triples 15% of

1. http://wimmics.inria.fr/corese
2. https://code.google.com/p/live-linked-data/wiki/ColGraphExperiments



(a) Deletion Times (b) Insertion Times

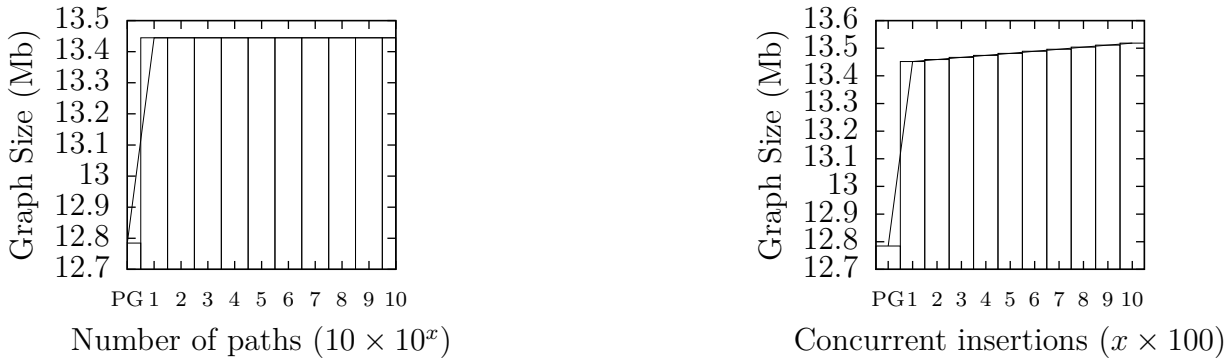
Figure 9.6: Comparison of execution time (ms) between synchronization and fragment reevaluation. Error bars show the error at 95%.

the test dataset's size). We loaded the test dataset in a source, materialized the fragment in a target and measured the execution time when inserting and when deleting 1, 5, 10, 20, 30, 40 and 50% of triples concerning the fragment. As baseline, we set up the same datasets on two RDF-Graphs and measured the time of clearing the target and the re-evaluation of the query that defines the fragment on the source. Both source and target were hosted on the same machine to abstract from latency.

We used the Java MicroBenchmark Harness [3] v. 0.5.5 to measure the average time of 50 executions across 10 JVM forks with 50 warm-up rounds, for a total of 500 samples. Experiments were run on a server with 20 hyperthreaded cores with 128Gb of ram an Linux Debian Wheezy. Figure 9.6 shows a linear behaviour, consistent with the analysis in section 9.3. Synchronization is less expensive than re-evaluation up to approx. 30% of updates. We believe that a better implementation that takes full advantage of streaming, as Corese does by processing data in RDF/XML, could improve performance. Fragments comprised of only one triple pattern are also very fast to evaluate, we expect than in future work, when we can support a broader class of fragments, making our protocol faster in most cases.

Our second experiment compares the impact on annotation's size produced by two of the factors analyzed in section 9.3: concurrent insertions and collaboration network connectivity, in order to determine which is more significant. We loaded the test dataset in: (i) An RDF-Graph. (ii) An annotated RDF-Graph, simulating n concurrent insertions of all triples, at n annotated RDF-Graphs with id `http://participant.topdomain.org/$i$`, with $i \in [0, n]$ (iii) An annotated RDF-Graph, simulating the insertion of all triples in an RDF-Graph with id "http://www.example.org/participant", arriving through m different simple paths, and measured their size in memory on a Macbook Pro

3. http://openjdk.java.net/projects/code-tools/jmh/



(a) Simulation of one insertion arriving through multiple paths.

(b) Simulation of concurrent insertions arriving through one path.

Figure 9.7: Space Overhead of the Annotated Graph w.r.t a plain graph (PG). Both Concurrency and Connectivity represent approx. 6% of overhead each.

running MacOS Lion with java 1.7.0_10-ea-b13 and Java HotSpot(TM) 64-Bit Server VM (build 23.6-b04, mixed mode).

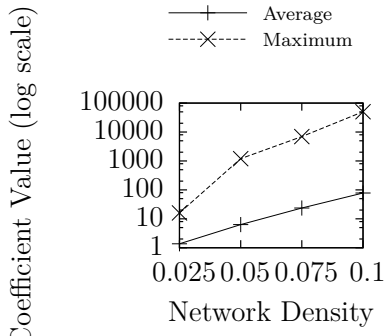
Figure 9.7 shows the results. Both cases represent nearly the same overhead, between 5 and 6 percent of size. Concurrency makes annotation's size grow sub-linearly. With respect to path number, annotation's size grows even slower , however, after 10×10^{17} paths, the *long* type used in our implementation overflows, meaning that in scenarios with this level of connectivity, the implementation must use BigInt arithmetics. In conclusion, after paying the initial cost of putting annotations in place, Col-Graph can tolerate a high number of concurrent inserts and a high network connectivity.

The goal of our final experiment is to study the effect of network's topology on Col-Graph's annotation's size. We argue that the act of materializing fragments and sharing updates is socially-driven, therefore, we are interested in analyzing the behaviour of Col-Graph on social networks. We generated two sets of 40 networks with 50 participants each, all edges defining full fragments, one following the random Ërdos-Renyi model [38] and other following the social-network oriented Forest Fire model [77]. Each set of networks is comprised of 4 subsets of 10 networks with densities $\{0.025, 0.05, 0.075, 0.1\}$. Table 9.5 shows the average of the average node connectivity of each network set. Social networks in the experiment are less connected than random ones, thus, we expect to have better performance, as we found network connectivity an impact factor in section 9.3.2.

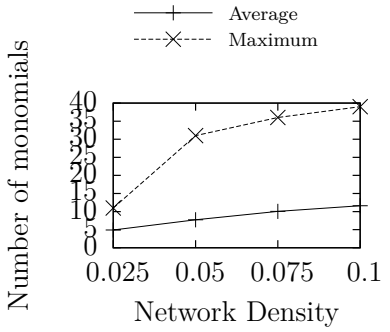
We loaded the networks on the Grid5000 platform (`https://www.grid5000.fr/`) and made each participant insert the same triple to introduce full concurrency, thus, fixing the overlapping and concurrency parameter in their worst case. Then, we let them synchronize repeatedly until quiescence with a 1 hour timeout. To detect termination, we implemented the most naive algorithm:

	density=0.025	density=0.05	density=0.075	density=0.1
Forest Fire	0.0863	0.2147	0.3887	0.5543
Ërdos-Renyi	0.293	1.3808	2.5723	3.7378

Table 9.5: Average node connectivities of the experimental network sets as a function of their density.



(a) Average and maximum coefficient value



(b) Average and maximum number of terms

Figure 9.8: Performance of the synchronization algorithm when applied on networks generated with the Forest Fire model

a central overlord controls the execution of the whole network . We measured the maximum and average coefficient values and the maximum and average number of terms of annotations.

Figure 9.8 shows the results for Forest Fire networks. The gap between the average and maximum values indicates that topology has an important effect: only few triples hit high values. From the Ërdos-Renyi dataset, only networks with density 0.025 finished before timeout. The difference with respect to the networks in the ForestFire dataset with the same density was not significant. These results suggest that high connectivity affects the time the network takes to converge, and, as the number of rounds to converge is much higher, the coefficient values should also be much higher. We leave the study of convergence time and the implementation of a better termination detection strategy for future work.

9.5 Conclusion and Perspectives

We defined *Fragment Consistency* (FC), a criterion strictly stronger than Strong Eventual Consistency. Participants copy subsets of data or *fragments*, defined by SPARQL CONSTRUCT queries, from each others. FC means that the evaluation of the query that defines the fragments at the source plus the updates locally made by the copying participant equals the state of the copying participant.

We presented Col-Graph, a coordination-free protocol based on annotated RDF-Graphs and updates to achieve FC. We analyzed the complexity of our algorithm in time, space and traffic, and

determined that the main factors that affect performance are the probability of concurrent insertion, the connectivity of the collaboration network and the fragment overlapping.

We evaluated experimentally the incurred overhead using a 50k real dataset on our open source implementation, finding that space, concurrency and connectivity represent approximately 6% of overhead each, with a sublinear grow.; in time, our algorithm is faster than the reevaluation of the fragment up to 30% of updated triples without taking in account latency. We also found that our algorithm performs better in socially generated networks than in random ones.

Compared to SU-Set (cf chapter 8), Col-Graph has the same complexities except in two cases: first, when the network is complete and fragments are full the coefficient of the annotations will reach $O(N!)$; second, the number of messages transmitted in the whole network before quiescence is higher.

Perspectives include:

- Estimate the average complexity of Col-Graph with a large scale evaluation focused on the effect of fragment overlapping and with different dataset dynamics. Compare this with the same evaluation for SU-Set (cf section 8.5).
- Investigate the number of messages needed to reach quiescence and compare them with the ones needed by SU-SET.
- Benchmark replication-aware federated query engines on collaboration networks using Col-Graph to quantify the gain in availability.
- Extend Col-Graph to handle dynamics in the fragment definitions themselves, *i.e.*, what happens if a participants wants to change the fragment he is copying from other participant.
- Study strategies to augment the expressiveness of the fragments that can be materialized, *i.e.*, allow more general SPARQL queries in their definitions. A possible research direction is the decomposition of a complex query in self-maintainable fragments.

III

Overall Conclusion and perspectives

10

Conclusions and perspectives.

In this thesis we answer the following research questions:

- *How to integrate format-heterogeneous data sources to the Web of Linked Data? How to query semantic-heterogeneous data sources in the Web of Linked Data?*
- *How to allow Linked Data participants to write each other's data and turn the Linked Data into Read/Write?. Which consistency criteria are suitable for a Read/Write Linked Data? How to maintain them respecting the autonomy of the participants and without compromising their availability and scaling in large number of participants and large datasets?*

For the first research question, we casted the problem to a Local-as-View Data Integration problem and identified the rewriting explosion issue as a bottleneck to obtain results: the execution of a large number of rewritings may take a large amount of time. We developed GraphUnion (GUN), a rewriting execution strategy for the LAV paradigm that takes advantage of RDF graph data model to improve the number of answers produced and the execution time with respect to traditional execution techniques, at the cost of higher memory usage. GUN opens the door to feasible implementations of the LAV paradigm for the Web of Linked Data.

Concerning the second question, we modeled the Read/Write Linked Data as a social network of RDF Graph Stores tied by the relationship *Consume Updates*, *i.e.*, participants copy all or some of the

data from others and subscribe to their updates to keep in sync. On their local copies, participants make updates; other participants, including the original data sources can in turn consume these updates if they consider them correct. However, if this *update exchange* is not correctly managed, data and knowledge may diverge, even in non-deterministic ways. The absence of consistency guarantees for queries and updates severely undermines the interaction between participants.

Under the proposed social network model we formulated the problem of finding a criterion and an update exchange protocol to maintain it such that the scalability, availability and autonomy restrictions imposed by the Web of Linked Data conditions get respected. We proposed two solutions: SU-Set, that guarantees *Strong Eventual Consistency*, and Col-Graph, that guarantees a criterion developed by us called Fragment Consistency.

SU-Set's guarantees can be summarized in the sentence: *Participants that have received the same updates, have the same state.* SU-Set has linear complexity in time and space independently of the topology, and optimal cost in communication. However, when the copies are partial or *fragments* of data, there is nothing that can be assured on the consistency of fragments with respect to their sources.

To overcome SU-Set's limitation, we proposed *Fragment Consistency* as criterion and Col-Graph as algorithm to maintain it. Each participant can copy or materialize from other participants a fragment of data defined by a SPARQL CONSTRUCT Federated query and receives the updates that concern the fragment. Col-Graph's guarantee can be summarized as follows: *each materialized fragment is equal to the evaluation of the fragment at its source modulo the locally made updates.* Col-Graph's complexity is the same as SU-Set's except in two aspects: (i) In space, where it depends on the connectivity of the network with a worst case of factorial (complete graph). (ii) In number of total messages exchanged in the network to converge. Nevertheless, our experiments suggest that for social networks, the performance is much better than for random networks, meaning that Col-Graph is applicable for the Read/Write Linked Data.

One interesting conclusion of our work is the very close relation between the solutions used for two very different visions of consistency: Conflict-Free Replicated Data Types (CRDTs) in distributed systems and Collaborative Data Sharing Systems (CDSSs) in databases. Both resort to annotate data with elements of an algebraic structure, lattices in the case of CRDTs, and commutative semirings in the case of CDSSs. The main difference is the idempotency of lattices and the non-idempotency

of semi-rings. Idempotency is required in replication scenarios to tolerate network disorder, on the other hand, semi-rings are required to support relational algebra operators but this feature requires coordination in the update exchange. We showed that for the special case of fragments, we can use semi-rings and still have the coordination freeness of idempotent solutions.

Finally, the connection between provenance and consistency maintenance is also worth to highlight. The data annotations used to model concurrency in CRDTs equal to one of the basic types of provenance semi-rings, therefore, if provenance information about triples is maintained in a semi-ring transformable format, then the consistency criteria proposed in this thesis can be attained.

10.1 Perspectives

In this section we detail the perspectives of our work

10.1.1 Querying Heterogeneous Data Sources on the Web of Data

GraphUnion (Chapter 5) uses query rewritings as input to create an aggregate graph of the views used in those rewritings and execute the original query. An interesting improvement is to avoid the rewriting phase and use only the component that selects relevant views to create the aggregated graph. This idea has been developed in [85].

Another perspective is to extend Graph-Union to be able to execute SPARQL Updates on the format heterogeneous sources. This is interesting if the original data format is used as input for other processes.

10.1.2 Consistency Criteria for the Web of Linked Data

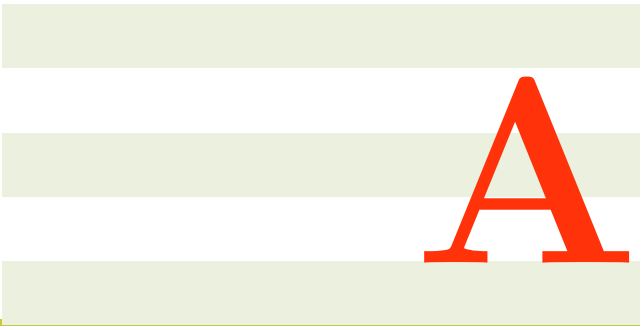
The first perspective is a large scale experimentation with different topologies to better characterize Col-Graph's average case. Very large social networks tend to *densify*, *i.e.*, to form highly connected communities [77] maybe challenging Col-Graph's performance. We would also like to analyze the complexity in terms of total messages exchanged in the network.

The submission of the annotated RDF-Graph to the W3C as a standardization proposal is closely related to the current provenance standardization efforts. The main idea is that a fourth attribute of an RDF-triple could store provenance in semi-ring format and be used for consistency maintenance

in a less expensive way than reification.

An important step would be to support fragments beyond single Basic Graph Patterns. This is equivalent to explore stronger consistency criteria than Fragment Consistency, for example, union of fragments or joins between fragments. The more expressive fragments, the closer to a CDSSs setup we will be. An interesting direction is to determine what is the maximum fragment expressiveness achievable without coordination.

Finally, we would like to extend our study to consistency criteria that take account entailment and/or the preservation of links across different datasets.



Queries, Views and detailed results of Graph Union

A.1 Queries

The queries are taken from [24]:

Query 1

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Product rdfs:label ?Label .
    ?Product rdf:type ?ProductType .
    ?Product bsbm:productFeature ?ProductFeature1 .
    ?Product bsbm:productFeature ?ProductFeature2 .
    ?Product bsbm:productPropertyNumeric1 ?Value1 .
}
```

Query 2

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
```

```
SELECT *
WHERE {
    ?Product rdfs:label ?Label .
    ?Product rdfs:comment ?Comment .
    ?Product bsbm:producer ?P .
    ?P rdfs:label ?Producer .
    ?Product dc:publisher ?P .
    ?Product bsbm:productFeature ?F .
    ?F rdfs:label ?ProductFeature .
    ?Product bsbm:productPropertyTextual1 ?PropertyTextual1 .
    ?Product bsbm:productPropertyTextual2 ?PropertyTextual2 .
    ?Product bsbm:productPropertyTextual3 ?PropertyTextual3 .
    ?Product bsbm:productPropertyNumeric1 ?PropertyNumeric1 .
    ?Product bsbm:productPropertyNumeric2 ?PropertyNumeric2 .
}
```

Query 3

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX rev:<http://purl.org/stuff/rev#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
    ?Product rdfs:label ?ProductLabel .
    ?Offer bsbm:product ?Product .
    ?Offer bsbm:price ?Price .
    ?Offer bsbm:vendor ?Vendor .
    ?Vendor rdfs:label ?VendorTitle .
    ?Vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
    ?Offer dc:publisher ?Vendor .
    ?Offer bsbm:validTo ?Date .
    ?Review bsbm:reviewFor ?Product .
    ?Review rev:reviewer ?Reviewer .
    ?Reviewer foaf:name ?RevName .
    ?Review dc:title ?RevTitle .
    ?Review bsbm:rating1 ?Rating1 .
}
```

Query 4

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
    ?VendorURI rdfs:label ?Vendorname .
    ?VendorURI foaf:homepage ?Vendorhomepage .
}
```

Query 5

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
```

```
SELECT *
WHERE {
    ?Offer bsbm:vendor ?VendorURI .
    ?Offer bsbm:offerWebpage ?OfferURL .
    ?VendorURI rdfs:label ?Vendorname .
    ?VendorURI foaf:homepage ?Vendorhomepage .
}
```

Query 6

```
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rev:<http://purl.org/stuff/rev#>
SELECT *
WHERE {
    ?Review bsbm:reviewFor ?Product .
    ?Review rev:reviewer ?Reviewer .
    ?Review bsbm:rating1 ?Rating1 .
}
```

Query 7

```
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
SELECT *
WHERE {
    ?Offer bsbm:product ?Product .
    ?Offer bsbm:vendor ?Vendor .
    ?Offer dc:publisher ?Vendor .
    ?Vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#US> .
    ?Offer bsbm:deliveryDays ?DeliveryDays .
    ?Offer bsbm:price ?Price .
    ?Offer bsbm:validTo ?Date .
}
```

Query 8

```
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Offer2 bsbm:offerWebpage ?OfferURL2 .
    ?Offer2 bsbm:price ?Price .
    ?Offer2 bsbm:deliveryDays ?DeliveryDays .
}
```

Query 9

```
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Product bsbm:productPropertyNumeric1 ?Value1 .
}
```

Query 10

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

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Product rdfs:label ?Label .
    ?Product rdf:type ?ProductType .
    ?Product bsbm:productFeature ?ProductFeature1 .
}
```

Query 11

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Product rdf:type ?ProductType .
    ?Product bsbm:productFeature ?ProductFeature1 .
}
```

Query 12

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
SELECT *
WHERE {
    ?Product bsbm:producer ?P .
    ?P rdfs:label ?Producer .
    ?Product dc:publisher ?P .
    ?Product bsbm:productFeature ?F .
}
```

Query 13

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Product bsbm:productFeature ?F .
    ?F rdfs:label ?ProductFeature .
}
```

Query 14

```
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Product bsbm:producer ?P .
    ?Offer bsbm:product ?Product .
    ?Offer bsbm:vendor ?VendorURI .
}
```

Query 15

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

```
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX rev:<http://purl.org/stuff/rev#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
    ?Product rdfs:label ?ProductLabel .
    ?Review bsbm:reviewFor ?Product .
    ?Review rev:reviewer ?Reviewer .
    ?Reviewer foaf:name ?RevName .
    ?Review dc:title ?RevTitle .
}
```

Query 16

```
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX rev:<http://purl.org/stuff/rev#>
SELECT *
WHERE {
    ?Review bsbm:reviewFor ?Product .
    ?Review dc:title ?Title .
    ?Review rev:text ?Text .
}
```

Query 17

```
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Review bsbm:reviewFor ?Product .
    ?Review bsbm:rating1 ?Rating1 .
}
```

Query 18

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Offer bsbm:product ?ProductURI .
    ?ProductURI rdfs:label ?Productlabel .
    ?Offer bsbm:vendor ?VendorURI .
    ?Offer bsbm:price ?Price .
}
```

A.2 Views

We took 10 views from [24]

View 1

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

```
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT * WHERE {
    ?Product rdfs:label ?Label .
    ?Product rdf:type ?ProductType .
    ?Product bsbm:productFeature ?ProductFeature1 .
}
```

View 2

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Product rdf:type ?ProductType .
    ?Product bsbm:productFeature ?ProductFeature1 .
}
```

View 3

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
SELECT *
WHERE {
    ?Product bsbm:producer ?P .
    ?P rdfs:label ?Producer .
    ?Product dc:publisher ?P .
    ?Product bsbm:productFeature ?F .
}
```

View 4

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Product bsbm:productFeature ?F .
    ?F rdfs:label ?ProductFeature .
}
```

View 5

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
SELECT *
WHERE {
    ?Product rdfs:label ?Label .
    ?Product rdfs:comment ?Comment .
    ?Product bsbm:producer ?P .
    ?P rdfs:label ?Producer .
    ?Product dc:publisher ?P .
    ?Product bsbm:productPropertyTextual1 ?PropertyTextual1 .
    ?Product bsbm:productPropertyNumeric1 ?PropertyNumeric1 .
}
```

View 6

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Product rdfs:label ?ProductLabel .
    ?Offer bsbm:product ?Product .
    ?Offer bsbm:price ?Price .
    ?Offer bsbm:vendor ?Vendor .
}
```

View 7

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX rev:<http://purl.org/stuff/rev#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
    ?Product rdfs:label ?ProductLabel .
    ?Review bsbm:reviewFor ?Product .
    ?Review rev:reviewer ?Reviewer .
    ?Reviewer foaf:name ?RevName .
    ?Review dc:title ?RevTitle .
}
```

View 8

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX rev:<http://purl.org/stuff/rev#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
    ?Offer bsbm:product ?Product .
    ?Offer bsbm:price ?Price .
    ?Offer bsbm:vendor ?Vendor .
    ?Vendor rdfs:label ?VendorTitle .
    ?Vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
    ?Offer dc:publisher ?Vendor .
    ?Review bsbm:reviewFor ?Product .
    ?Review rev:reviewer ?Reviewer .
    ?Reviewer foaf:name ?RevName .
}
```

View 9

```
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX rev:<http://purl.org/stuff/rev#>
SELECT *
WHERE {
    ?Review bsbm:reviewFor ?Product .
```

```
    ?Review dc:title ?Title .
    ?Review rev:text ?Text .
}
```

View 10

```
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Review bsbm:reviewFor ?Product .
    ?Review bsbm:rating1 ?Rating1 .
}
```

We defined 5 additional views to cover all predicates in the queries:

View 11

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
SELECT *
WHERE {
    ?Product rdfs:label ?Label .
    ?Product rdfs:comment ?Comment .
    ?Product bsbm:producer ?P .
    ?P rdfs:label ?Producer .
    ?Product dc:publisher ?P .
    ?Product bsbm:productPropertyTextual2 ?PropertyTextual2 .
    ?Product bsbm:productPropertyNumeric2 ?PropertyNumeric2 .
}
```

View 12

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
SELECT *
WHERE {
    ?Product rdfs:label ?Label .
    ?Product rdfs:comment ?Comment .
    ?Product bsbm:producer ?P .
    ?P rdfs:label ?Producer .
    ?Product dc:publisher ?P .
    ?Product bsbm:productPropertyTextual3 ?PropertyTextual3 .
    ?Product bsbm:productPropertyNumeric3 ?PropertyNumeric3 .
}
```

View 13

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
    ?Product rdfs:label ?ProductLabel .
    ?Offer bsbm:product ?Product .
```

```
    ?Offer bsbm:price ?Price .
    ?Offer bsbm:vendor ?Vendor .
    ?Offer bsbm:offerWebpage ?OfferURL .
    ?Vendor foaf:homepage ?Vendorhomepage .
}
```

View 14

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT *
WHERE {
    ?Product rdfs:label ?ProductLabel .
    ?Offer bsbm:product ?Product .
    ?Offer bsbm:price ?Price .
    ?Offer bsbm:vendor ?Vendor .
    ?Offer bsbm:deliveryDays ?DeliveryDays .
    ?Offer bsbm:validTo ?Date .
}
```

View 15

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm:<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX dc:<http://purl.org/dc/elements/1.1/>
PREFIX rev:<http://purl.org/stuff/rev#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
    ?Offer bsbm:product ?Product .
    ?Offer bsbm:price ?Price .
    ?Offer bsbm:vendor ?Vendor .
    ?Vendor rdfs:label ?VendorTitle .
    ?Vendor bsbm:country ?Country .
    ?Offer dc:publisher ?Vendor .
    ?Review bsbm:reviewFor ?Product .
    ?Review rev:reviewer ?Reviewer .
    ?Reviewer foaf:name ?RevName .
}
```

Bibliography

[1] Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *Principles of Distributed Database Systems (PODS)*, 1998. 27

[2] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. *Web Data Management*. Cambridge University Press, 2011. 22, 34

[3] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *24th International Conference on Very Large Databases*, VLDB, 1998. 73

[4] Maribel Acosta, Maria Esther Vidal, Tomás Lampo, Julio Castillo, and Edna Ruckhaus. Anapsid: An adaptive query processing engine for sparql endpoints. In *International Semantic Web Conference (ISWC)*, pages 18–34, 2011. 25

[5] Parag Agrawal, Adam Silberstein, Brian F Cooper, Utkarsh Srivastava, and Raghu Ramakrishnan. Asynchronous view maintenance for vlsd databases. In *SIGMOD*, 2009. 73

[6] Emilien Antoine. *Distributed data management with the declarative rule-based language: WebDamLog*. PhD thesis, Université Paris-Sud, 2013. 73

[7] Yolifé Arvelo, Blai Bonet, and Maria Esther Vidal. Compilation ofquery-rewriting problems into tractable fragments of propositional logic. In *AAAI*, 2006. 27, 28, 34

[8] Khaled Aslan, Hala Skaf-Molli, Pascal Molli, and Stéphane Weiss. C-set : a commutative replicated data type for semantic stores. In *RED: Fourth International Workshop on Resource Discovery, At the 8th Extended Semantic Web Conference (ESWC)*, pages 123–130, 2011. 70

[9] Sören Auer and Heinrich Herre. A versioning and evolution framework for rdf knowledge bases.

In *Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference (PSI)*, pages 55–69. Springer, 2007. 63

[10] Jean-François Baget, Madalina Croitoru, Alain Gutierrez, Michel Leclère, and Marie-Laure Mugnier. Translations between rdf(s) and conceptual graphs. In *International Conference on Conceptual Structures*, ICCS, pages 28–41, 2010. 35

[11] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)*, 6(4), 1981. 73

[12] Tim Berners-Lee and Dan Connolly. Delta: an ontology for the distribution of differences between rdf graphs. http://www.w3.org/DesignIssues/Diff, 2001. 63, 75

[13] Tim Berners-Lee and Kieron O'Hara. The read-write linked data web. *Philosophical Transactions of the Royal Society*, 2013. 56

[14] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *International Symposium on Distributed Computing (DISC)*, 2012. 69

[15] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. Technical report, INRIA, October 2012. 71

[16] C. Bizer, T. Heath, and T. Berners-Lee. Linked data-the story so far. *International Journal on Semantic Web and Information Systems*, 4(2):1–22, 2009. 58

[17] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *International Journal of Semantic Web Information Systems*, 5(3):1–22, 2009. 5

[18] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. Dbpedia - a crystallization point for the web of data. *Journal of Web Semantics*, 7(3):154–165, 2009. 77

[19] Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *International Journal of Semantic Web Information Systems*, 2009. 29, 32, 36, 43

[20] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: A language for updatable views. In *Symposium on Principles of Database Systems*, PODS. ACM, 2006. 73

[21] Eric Brewer. Cap twelve years later: How the "rules" have changed. *IEEE Computer*, 45(2), 2012. 69

[22] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. Sparql web-querying infrastructure: Ready for action? In *International Semantic Web Conference (ISWC)*, 2013. 56

[23] Steve Cassidy and James Ballantine. Version control for rdf triple stores. In *International Joint Conference on Software Technologies (ICSOFT)*, 2007. 64, 76

[24] Roberto Castillo-Espinola. *Indexing RDF data using materialized SPARQL queries*. PhD thesis, Humboldt-Universitat zu Berlin, 2012. 32, 43, 121, 125

[25] Surajit Chaudhuri. An overview of query optimization in relational systems. In *ACM SIGACT-SIGMOD-SIGART*, pages 34–43, New York, NY, USA, 1998. 36

[26] Songting Chen, Jun Chen, Xin Zhang, and Elke A. Rundensteiner. Detection and correction of conflicting source updates for view maintenance. In *International Conference on Data Engineering (ICDE)*, 2004. 73

[27] Songting Chen, Bin Liu, and Elke A. Rundensteiner. Multiversion-based view maintenance over multiversion-based view maintenance over distributed data sources. *ACM Transactions on Database Systems*, 29(4):675–709, 2004. 73

[28] Rada Chirkova and Jun Yang. Materialized views. *Foundations and Trends in Databases*, 4(4), 2011. 73

[29] Cédric Coulon, Esther Pacitti, and Patrick Valduriez. Consistency management for partial replication in a high performance data cluster. In *International Conference on Parallel and Distributed Systems (ICPADS)*, 2005. 72

[30] Richard Cyganiak and Anja Jentzch. Linking open data cloud diagram. `http://lod-cloud.net`, 2011. 62, 151

[31] C.J. Date. *View Updating and Relational Theory*. O'Reilly Media, 2012. 73

[32] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems (TODS)*, 7(3), 1982. 73

[33] Andrei Deftu and Jan Griebsch. A scalable conflict-free replicated set data type. In *International Conference on Distributed Computing Systems*. IEEE, 2013. 71

[34] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Erik Mannens, and Rik Van de Walle. Extending r2rml to a source-independent mapping language for rdf. In *International Semantic Web Conference, Poster Session*, ISWC, 2013. 29

[35] Paul Dourish. The parting of the ways: Divergence, data management and collaborative work. In *European Conference on Computer-Supported Cooperative Work (ECSCW)*, 1995. 68

[36] C.A. Elllis and S.J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on management of data*, 1989. 69

[37] Douglas Engelbart and Harvey Lehtman. Working together. *Byte*, 13(13), 1988. 56

[38] Paul Erdös and Alfréd Rényi. On the evolution of random graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl*, 5, 1960. 112

[39] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 2005. 74

[40] Wenfei Fan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 38(3), 2013. 73

[41] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary rdf representation for publication and exchange (hdt). *Journal of Web Semantics*, 19:22–41, 2013. 83

[42] Pauline Folz, Gabriela Montoya, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. Semlav: Querying deep web and linked open data with sparql. In *The Semantic Web: Trends and Challenges - 11th International Conference (Demo Session)*, ESWC, 2014. 51

[43] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 2007. 73

[44] Todd J. Green, Zachary G. Ives, and Val Tannen. Reconcilable differences. *Theory of Computer Systems*, 49(2), 2011. 100

[45] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Principles of Database Systems (PODS)*, 2007. 75, 104

[46] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer, 1993. 83

[47] Dominique Guinard and Vlad Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*, Madrid, Spain, April 2009. 58

[48] Ashish Gupta, H.V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *International Conference on Extending Database Systems (EDBT)*, 1996. 100

[49] Christian Halaschek-Wiener and Vladimir Kolovski. Syndication on the web using a description logic approach. *Journal of Web Semantics*, 6(3), 2008. 77

[50] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 2001. 22, 34

[51] Alon Y. Halevy, Zachary G. Ives, Jayant Madhavan, Peter Mork, Dan Suciu, and Igor Tatarinov. The piazza peer data management system. *IEEE Transactions on Knowledge and Data Engineering*, 16(7), 2004. 76

[52] Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *12th international conference on World Wide Web*, WWW, 2003. 76

[53] Andreas Harth, Craig A. Knoblock, Steffen Stadtmüller, Rudi Studer, and Pedro A. Szekely. On-the-fly integration of static and dynamic sources. In *Proceedings of the Fourth International Workshop on Consuming Linked Data*, COLD, 2013. 29

[54] Olaf Hartig. Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference*, ESWC, 2011. 25

[55] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan and Claypool, 2011. 5

[56] Maurice Herlihy and Jeanette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.

[57] Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Edit lenses. In *Symposium on Principles of Programming Languages*, POPL. ACM, 2012. 73

[58] Katja Hose, Armin Roth, Andre Zeitz, Kai-Uwe Sattler, and Felix Naumann. A research agenda for query processing in large-scale peer data management systems. *Inf. Syst.*, 33(7-8), 2008. 74

[59] Edward Hung, Yu Deng, and V.S. Subrahmanian. Rdf aggregate queries and views. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2005. 73

[60] Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Olivier Corby. Synchronizing semantic stores with commutative replicated data types. In Alain Mille, Fabien L. Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab, editors, *WWW (Companion Volume)*, pages 1091–1096. ACM, 2012. 58, 82, 92

[61] Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Olivier Corby. Live linked data: Synchronizing semantic stores with commutative replicated data types. *International Journal of Metadata, Semantics and Ontologies*, 8(2):119–133, 2013. 8, 58

[62] Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Olivier Corby. Col-graph: Towards writable and scalable linked open data. In *13th International Semantic Web Conference*, ISWC, 2014. 8, 57, 58, 102

[63] Zachary G. Ives, Todd J.Green, Grigorios Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. The orchestra collaborative data sharing system. In *SIGMOD Record*, 2008. 74, 75

[64] Zachary G. Ives, Nitin Khandewal, Aneesh Kapur, and Murat Cakir. Orchestra: Rapic, collaborative sharing of dynamic data. In *Conference on Innovative Data Systems Research (CIDR)*, 2005. 74

[65] Daniel Izquierdo, Maria Esther Vidal, and Blai Bonet. An expressive and efficient solution to the service selection problem. In *9th International Semantic Web Conference*, pages 386–401. Springer, 2010. 28

[66] P. Johnson and R. Thomas. Rfc677: The maintenance of duplicate databases, 1976. 71

[67] Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, and Spyros Zoupanos. Vip2p: Efficient xml management in dht networks. In *Web Engineering - 12th International Conference*, ICWE, 2012. 76

[68] Grigoris Karvounarakis and Todd J. Green. Semiring-annotated data: Queries and provenance. *SIGMOD Record*, 41(3), 2012. 75

[69] Grigoris Karvounarakis, Todd J. Green, Zachary G. Ives, and Val Tannen. Collaborative data sharing via update exchange and provenance. *ACM Transactions on Database Systems*, 38(3), August 2013. 74, 75, 103, 152

[70] Bettina Kemme, Ganesan Ramalingam, André Schiper, Marc Shapiro, and Kapil Vaswani. Consistency in distributed systems. *Dagstuhl Reports*, 3(2):92–126, 2013. 69

[71] Craig A. Knoblock, Pedro A. Szekely, José Luis Ambite, Aman Goel, Shubham Gupta, Kristina Lerman, Maria Muslea, Mohsen Taheriyan, and Parag Mallick. Semi-automatically mapping structured sources into the semantic web. In *The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference*, ESWC, 2012. 29

[72] George Konstantinidis and José-Luis Ambite. Scalable query rewriting: a graph-based approach. In *SIGMOD*, 2011. 23, 27, 28, 32, 34

[73] Lucja Kot and Christoph Koch. Cooperative update exchange in yhe youtopia system. In *International Conference on Very Large Data Bases (VLDB)*, 2009. 74

[74] Günter Ladwig and Thanh Tran. Sihjoin: Querying remote and local linked data. In *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference*, ESWC, 2011. 25

[75] Wangchao Le, Songyun Duan, Anastasios Kementsietsidis, Feifei Li, and Min Wang. Rewriting queries on sparql views. In *WWW*, 2011. 28

[76] P. Leach, M. Mealling, and R. Salz. A universally unique identifier (uuid) urn namespace. *Internet RFCs*, RFC 4122, 2005. 71, 82

[77] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1), march 2007. 112, 119

[78] Alon Levy, Anand Rajaraman, and Joann Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, pages 251–262, 1996. 28

[79] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 95–104. ACM, 1995. 6, 34

[80] Tok Wan Ling and Eng Koon Sze. Materialized view maintenance using version numbers. In *International Conference on Database Systems for Advanced Applications*, 1999. 73

[81] Alejandro Mallea, Marcelo Arenas, Aidan Hogan, and Axel Polleres. On blank nodes. In *International Semantic Web Conference (1)*, pages 421–437, 2011. 65

[82] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989. 71, 80

[83] Keita Mikami, Shinji Morishita, and Makoto Onizuka. Lazy view maintenance for social networking applications. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, 2010. 73

[84] Gabriela Montoya, Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. Gun: An efficient execution strategy for querying the web of data. In *24th International*

Conference on Database and Expert Systems Applications, DEXA, pages 180–194, 2013. 7, 10, 23

[85] Gabriela Montoya, Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. Semlav: Local-as-view mediation for sparql queries. *T. Large-Scale Data-and Knowledge-Centered Systems*, 13:33–58, 2014. 51, 119

[86] Gianluca Moro and Claudio Sartori. Incremental maintenance of multi-source views. In *Australasian Database Conference (ADC)*, 2001. 73

[87] Mohamed Morsey, Jens Lehmann, Sören Auer, Claus Stadler, and Sebastian Hellmann. Dbpedia and the live extraction of structured data from wikipedia. *Program: electronic library and information systems*, 46(2):157–181, 2012. 77

[88] Madhavan Mukund, Gautham Shenoy R., and S. P. Suresh. Optimized or-sets without ordering constraints. In *15th International Conference on Distributed Computing and Networking*, ICDCN, 2014. 71

[89] Brice Nédelec, Pascal Molli, Achour Mostéfaoui, and Emmanuel Desmontils. Lseq: an adaptive structure for sequences in distributed collaborative editing. In *ACM Symposium on Document Engineering*, DocEng, 2013. 70

[90] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, and Michael Sintek. Edutella: a p2p networking infrastructure based on rdf. In *11th international conference on World Wide Web*, WWW, 2002. 76

[91] Luís Eufrasio T. Neto, Vânia Maria P. Vidal, Marco A. Casanova, and José Maria Monteiro. R2rml by assertion: A semi-automatic tool for generating customised r2rml mappings. In *Extended Semantic Web Conference, Demo session*, ESWC, 2013. 29

[92] Damian Ognyanov and Atanas Kiryakov. Tracking changes in rdf(s) repositories. In *International Conference on Knowledge Engineering and Knowledge Management (EKAW)*, 2002. 63

[93] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 2011. 72

[94] Esther Pacitti, Cédric Coulon, and M. Tamer Özsu. Preventive replication in a database cluster. *Distributed and Parallel Databases*, 2005. 72

[95] Alexandre Passant and Pablo N. Mendes. sparqlpush: Proactive notification of data updates in rdf stores using pubsubhubbub. In *Sixth Workshop on Scripting and Development for the Semantic Web (SFSW)*, 2010. 77

[96] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems*, 34(3), August 2009. 12

[97] Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. *ACM SIGOPS Operating Systems Review*, 31(5):288–301, 1997. 69

[98] Rachel Pottinger and Alon Y. Levy. Minicon: A scalable algorithm for answering queries using views. In *VLDB*, pages 484–495, 2001. 28

[99] Nuno M. Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, *CoopIS/DOA/ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 38–55. Springer, 2003. 69

[100] Andrea Pugliese, Matthias Bröcheler, V.S. Subrahmanian, and Michael Ovelgönne. Efficient multi-view maintenance under insertion in huge social networks. *ACM Transactions on the Web (TWEB)*, 8(2), 2014. 73

[101] Charbel Rahhal, Hala Skaf-Molli, Pascal Molli, and Stéphane Weiss. Multi-synchronous collaborative wikis. In *Web Information Systems Engineering (WISE)*, 2009. 69

[102] Armin Roth and Sebastian Skritek. Peer data management. In *Data Exchange, Information and Streams. Dagstuhl Follow-Ups*, volume 5. Dagstuhl Publishing, 2012. 74

[103] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Survey*, 37(1):42–81, 2005. 69

[104] Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, Josiane Xavier Parreira, Helena F. Deus, and Manfred Hauswirth. Daw: Duplicate-aware federated query processing over the web of data. In *International Semantic Web Conference (ISWC)*, 2013. 57

[105] Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Sam Coppens, Erik Mannens, and Rik Van de Walle. R&wbase:git for triples. In *Linked Data on the Web Workshop (LDOW)*, 2013. 64, 76

[106] Bernhard Schandl. Replication and versioning of partial rdf graphs. In *European Semantic Web Conference (ESWC)*, 2010. 77

[107] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *International Semantic Web Conference (ISWC)*, 2011. 25

[108] Kunal Sengupta, Peter Haase, Michael Schmidt, and Pascal Hitzler. Editing r2rml mappings made easy. In *International Semantic Web Conference, Demo Session*, ISWC, 2013. 29

[109] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report 7506, INRIA, January 2011. 70

[110] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 386–400, 2011. 7, 8, 70, 71

[111] Hala Skaf-Molli, Gérôme Canals, and Pascal Molli. Dsmw: Distributed semantic media wiki. In *European Semantic Web Conference (ESWC) Demo Track*, 2010. 69

[112] Steffen Stadtmüller, Sebastian Speiser, Andreas Harth, and Rudi Studer. Data-fu: A language and an interpreter for interaction with read/write linked data. In *Proceedings of the 22nd International World Wide Web Conference*, WWW, 2013. 77

[113] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998. 68

[114] Nicholas E. Taylor and Zachary G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, 2006. 74

[115] Giovanni Tummarello and Christian Morbidoni. The dbin platform: A complete environment for semantic web communities. *Journal of Web Semantics*, 6(4), 2008. 76

[116] Giovanni Tummarello, Christian Morbidoni, Reto Bachmann-Gmür, and Orri Erling. Rdfsync: Efficient remote synchronization of rdf models. In *6th International and 2nd Asian Semantic Web Conference (ISWC + ASWC)*, pages 537–551, 2007. 76

[117] Giovanni Tummarello, Christian Morbidoni, Joackin Petersson, Paolo Puliti, and Francesco Piazza. Rdfgrowth, a p2p annotation exchange algorithm for scalable semantic web applications. In *Proceedings of the MobiQuitous'04 Workshop on Peer-to-Peer Knowledge Management (P2PKM 2004)*, 2004. 76

[118] Jeffrey D. Ullman. Information integration using logical views. *Theoretical Computer Science*, 239(2), 2000. 26

[119] Jürgen Umbrich, Marcel Karnstedt, Josiane Xavier Parreira, Axel Polleres, and Manfred Hauswirth. Linked data and live querying for enabling support platforms for web dataspaces. In *Third International Workshop on Data Engineering Meets the Semantic Web (DESWEB 2012)*, 2011. 57

[120] Ruben Verborgh, Miel Vander Sande, Pieter Colpaert, Sam Coppens, Erik Mannens, and Rik Van de Walle. Web-scale querying through linked data fragments. In *Linked Data on the Web Workshop (LDOW)*, 2014. 77, 100

[121] Maria-Esther Vidal, Jean Carlo Rivera, Luis-Daniel Ibáñez, Louiqa Raschid, Guillermo Palma, Héctor Rodríguez-Drummond, and Edna Ruckhaus. An authority-flow based ranking approach to discover potential novel associations between linked data. *Semantic Web*, 5(1), 2014.

[122] Maria-Esther Vidal, Edna Ruckhaus, Tomás Lampo, Amadís Martínez, Javier Sierra, and Axel Polleres. Efficiently joining group patterns in sparql queries. In *The Semantic Web: Research and Applications*, ESWC, 2010. 23, 28

[123] W3C. *SPARQL 1.1 Federated Query*, march 2013. 14, 98

[124] W3C. *SPARQL 1.1 Protocol*, march 2013. 14

[125] W3C. *SPARQL 1.1 Query Language*, march 2013. 12

[126] W3C. *SPARQL 1.1 Update*, march 2013. http://www.w3.org/TR/sparql11-update/. 14

[127] W3C. *RDF 1.1: Concepts and Abstract Syntax*, February 2014. http://www.w3.org/TR/rdf-concepts/. 11, 65

[128] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*. Cambridge University Press, 1994. 58, 60

[129] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel Distributed Systems*, 21(8):1162–1174, 2010. 69

[130] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In IEEE, editor, *International Conference on Distributed Computing Systems*, pages 464–474, 2000. 67

[131] Marcin Wylot, Philippe Cudré-Mauroux, and Paul Groth. Tripleprov: Efficient processing of lineage queries in a native rdf store. In *WWW*, 2014. 102

[132] Hafed Zarzour and Mokhtar Sellami. B-set: A synchronization method for distributed semantic stores. In *International Conference on Complex Systems (ICCS)*, 2012. 70

[133] Xin Zhang and Elke A. Rundensteiner. Integrating the maintenance and synchronization of data warehouses. *Information Systems*, 27(4):219–243, 2002. 73

[134] Yue Zhuge and Héctor García-Molina. Graph structured views and their incremental maintenance. In *14th International Conference on Data Engineering*, ICDE, 1998. 73

[135] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Distributed and Parallel Databases*, 6(1):7–40, 1998. 73

[136] Spyros Zoupanos. *Efficient peer-to-peer data management*. PhD thesis, Université Paris-Sud, 2009. 76

Contents

List of Tables

List of Figures

l'université nantes angers le mans
COMMUNAUTÉ D'UNIVERSITÉS ET ÉTABLISSEMENTS

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Luis Daniel IBÁÑEZ

Vers une Web des Données en Lecture-Écriture

Towards a Read/Write Web of Linked Data

Résumé

L'initiative «Web des données» a mis en disponibilité des millions des données pour leur interrogation par une fédération de participants autonomes. Néanmoins, le Web des Données a des problèmes de hétérogénéité et qualité. Nous considérons le problème de hétérogèneité comme une médiation «Local-as-View» (LAV). Malheureusement, LAV peut avoir besoin d'exécuter un certain nombre de « reformulations » exponentiel dans le nombre de sous-objectifs d'une requête. Nous proposons l'algorithme «Graph-Union» (GUN) pour maximiser les résultats obtenus á partir d'un sous-ensemble de reformulations. GUN réduit le temps d'exécution et maximise les résultats en échange d'une utilisation de la mémoire plus élevée. Pour permettre aux participants d'améliorer la qualité des données, il est nécessaire de faire évoluer le Web des Données vers Lecture-Écriture, par contre, l'écriture mutuelle des données entre participants autonomes pose des problèmes de cohérence. Nous modélisons le Web des Données en Lecture -Écriture comme un réseau social où les acteurs copient les données que leur intéressent, les corrigent et publient les mises à jour pour les échanger. Nous proposons deux algorithmes pour supporter cet échange : SU-Set, qui garantit la Cohérence Inéluctable Forte (CIF), et Col-Graph, qui garantit la Cohérence des Fragments, plus forte que CIF. Nous étudions les complexités des deux algorithmes et nous estimons expérimentalement le cas moyen de Col-Graph, les résultats suggèrant qu'il est faisable pour des topologies sociales.

Mots clés

Web des Données, Integration de Données, Cohérence des Données

Abstract

The Linked Data initiative has made available millions of pieces of data for querying through a federation of autonomous participants. However, the Web of Linked data suffers of problems of data heterogeneity and quality. We cast the problem of integrating heterogeneous data sources as a Local-as-View mediation (LAV) problem, unfortunately, LAV may require the execution of a number of "rewritings" exponential on the number of query subgoals. We propose the Graph-Union (GUN) strategy to maximise the results obtained from a subset of rewritings. Compared to traditional rewriting execution strategies, GUN improves execution time and number of results obtained in exchange of higher memory consumption. Once data can be queried data consumers can detect quality issues, but to resolve them they need to write on the data of the sources, i.e., to evolve Linked Data from Read/Only to Read-Write. However, writing among autonomous participants raises consistency issues. We model the Read-Write Linked Data as a social network where actors copy the data they are interested into, update it and publish updates to exchange with others. We propose two algorithms for update exchange: SU-Set, that achieves Strong Eventual Consistency (SEC) and Col-Graph, that achieves Fragment Consistency, stronger than SEC. We analyze the worst and best case complexities of both algorithms and estimate experimentally the average complexity of Col-Graph, results suggest that is feasible for social network topologies.

Key Words

Linked Data, Data Integration, Data Consistency.