Updating RDF in the Semantic Web

A thesis presented for the degree of

Doctor of Philosophy

Sana Al Azwari

2016

Department of Computer and Information Sciences

University of Strathclyde

Glasgow

DECLARATION

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

signed:

Date: 13/1/2016

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

First and formost, I would like to thank God, whose many blessings have made me who I am today.

I am forever grateful to my supervisor Dr. John Wilson for his patience, guidance and continuse support.

I would like to thank my husband and my daughters, for their patience and their understanding when I am not always there for them.

I also thank my parents, my brothers and my sister for their wise counsel and for being always there for me.

Finally, there are my friends for their encourgement and support.

Sana Al Azwari

May 2016

PUBLICATIONS

This work has resulted in the publication of the following papers:

1. Al Azwari, S., & Wilson, J. N. (2015, February). The cost of reasoning with RDF updates. In Semantic Computing (ICSC), 2015 IEEE International Conference on (pp. 328-331). IEEE.

2. Al Azwari, S., & Wilson, J. N. (2015). Consistent RDF updates with correct dense deltas. In Data Science (pp. 74-86). Springer International Publishing.

3. Al-Azwari, S., & Wilson, J. N. (2015, September). Updating OWL2 ontologies using pruned rulesets. In Proceedings of the 11th International Conference on Semantic Systems (pp. 105-112). ACM. (Winner of the best paper at SEMANTiCS 2015).

4. Al Azwari, S., & Wilson, J. (2016). Change Detection Techniques in the Semantic Web-Detailed Analysis. In Proceedings of the Eighth Saudi Students Conference in the UK (pp. 239-244). (Winner of the best presentation at SSC8 2015).

ABSTRACT

RDF is widely used in the Semantic Web for representing ontology data. Many real world RDF collections are large and contain complex graph relationships that represent knowledge in a particular domain. Such large RDF collections evolve as a consequence of their representation of the changing world.

Evolution in Semantic Web content produces difference files (deltas) that track changes between ontology versions. These changes may represent ontology modifications or simply changes in application data. An ontology is typically expressed in a combination of OWL, RDFS and RDF knowledge representation languages.

A data repository that represents an ontology may be large and may be duplicated over the Internet, often in the form of a relational data store. Although this data may be distributed over the Internet, it needs to be managed and updated in the face of such evolutionary changes. In view of the size of typical collections, it is important to derive efficient ways of propagating updates to distributed data stores.

The deltas can be used to reduce the storage and bandwidth overhead involved in disseminating ontology updates. Minimising the delta size can be achieved by reasoning over the underlying knowledge base. OWL 2 is a development of the

OWL 1 standard that incorporate new features to aid application construction. Among the sub languages of OWL 2, OWL 2 RL/RDF provides an enriched ruleset that extends the semantic capability of the OWL environment. This additional semantic content can be exploited in change detection approaches that strive to minimise the alterations to be made when ontologies are updated. The presence of blank nodes (i.e. nodes that are neither a URI nor a literal) in RDF collections provides a further challenge to ontology change detection. This is a consequence of the practical problems they introduce when comparing data structures before and after an update.

The contribution of this thesis is a detailed analysis of the performance of RDF change detection techniques. In addition, the work proposes a new approach to maintaining the consistency of RDF by using knowledge embedded in the structure to generate efficient update transactions. The evaluation of this approach indicates that it reduces the overall update size, at the cost of increasing the processing time needed to generate the transactions.

In the light of OWL 2 RL/RDF, this thesis examines the potential for reducing the delta size by pruning the application of unnecessary rules from the reasoning process and using an approach to delta generation that produces a small number of updates. It also assesses the impact of alternative approaches to handling blank nodes during the change detection process in ontology structures. The results indicate that pruning the rule set is a potentially expensive process but has the benefit of reducing the joins over relational data stores when carrying out the subsequent inferencing.

1. INTRODUCTION

Since its introduction in 1991[1], the World Wide Web (WWW) has expanded considerably in terms of size and number of users. The Web is a large repository for documents that may consist of text, images and other multimedia. Users over the Internet can simply access the content via both search engines and hyperlinks. The simplicity and universality of the WWW have been important factors in the growth of its content and as a consequence the WWW currently contains more than 7 billion pages[2].

More recently, Web 2.0 has become a popular phenomenon. Web 2.0 is a term given to describe the second generation of the World Wide Web that is focused on the ability for people to collaborate and share information online [Cro09]. Web 2.0 started with the publication of social sites such as Facebook[3], YouTube[4], LinkedIn[5], Twitter[6] and others. Web 2.0 has dramatically changed the way of publishing information and the Web has become a medium for human communication and interaction with no software skills required for exposing information. This feature of the Web has contributed a lot to the growth of its content. How-

[1] w3c.org
[2] http://www.worldwidewebsize.com/
[3] http://www.facebook.com/
[4] http://www.youtube.com/
[5] http://www.linkedin.com/
[6] twitter.com

ever, this explosion in the quantity of Web content has made it difficult for users to find and process the information they require. As information in the Web is typically presented in natural language, the use of search engines to find information may retrieve many irrelevant results or may fail to retrieve many relevant results. For example, when searching for the keyword 'Protein', indexes do not contain the fact that 'chain of amino acid' may be used as synonym for protein, and as a result, search engines will only retrieve Web pages that contain the keyword 'Protein'. Most search engines will fail to retrieve other Web pages that contain other synonyms of the search word.

Finding relevant information in the Web is not the only limitation of current Web solutions. Extracting information from different Web resources adds further difficulties. This problem goes back to the fact that Web contents are often in different file formats, which require different information extraction tools for each format.

Another task that adds limitations is the task of information integration on the Web. Combining information from different Web sites to fulfill users requirements and requests is a big challenge for the current Web [FFST11]. Human users find it difficult to access Web content and process information from different Web sites. However, it is a much more difficult task for machines or automated processes, as most Web content is presented in human natural languages and designed to be used and consumed by human users.

In order to take advantage of the capabilities of machines to process information on the Web, these machines need first to be able to capture the semantics of this information. In this thesis, semantics is defined as the expression of relationships between terms. This proposed solution is called the ‘Semantic Web’, also called Web 3.0, which aims to overcome some of the above mentioned problems by enabling machines to recognise and process Web content by adding semantics to its content in a machine readable language.

The initial vision of the Semantic Web by its inventor Tim Berners-Lee is that ”The Semantic Web is not a separate Web but an extension of the current one, in which information is given a well-defined meaning, better enabling computers and people to work in cooperation” [BLHL01].

Furthermore, the Semantic Web is an infrastructure technology that will help in linking Web content together and provide more information that can be recognised and processed by computers. This will provide a revolution in capabilities and improvements in functionality of the existing Web in many aspects such as search, information extraction and integration. However, Web browsers will still be used to access Web content and Web applications [Hen11].

For moving the current Web towards this level of service, it needs to be accompanied by a combination of machine-readable semantics of data and a knowledge representation scheme for information. Technologies for the Semantic Web have already been developed by the AI and W3C communities. This has led to the emergence of a number of issues related to the Semantic Web such as ontologies, Semantic Web languages, Semantic Web applications and services.

1.1 The Semantic Web Stack

Semantic Web technologies provide a means of formally describing the concepts, vocabularies and relationships for a certain domain. The standard Semantic Web technologies as specified by the W3C are RDF, RDFS, OWL and SPARQL[7]. These technologies are built one on top of another, forming layers of technologies known as Semantic Web Stack (Figure 1.1).

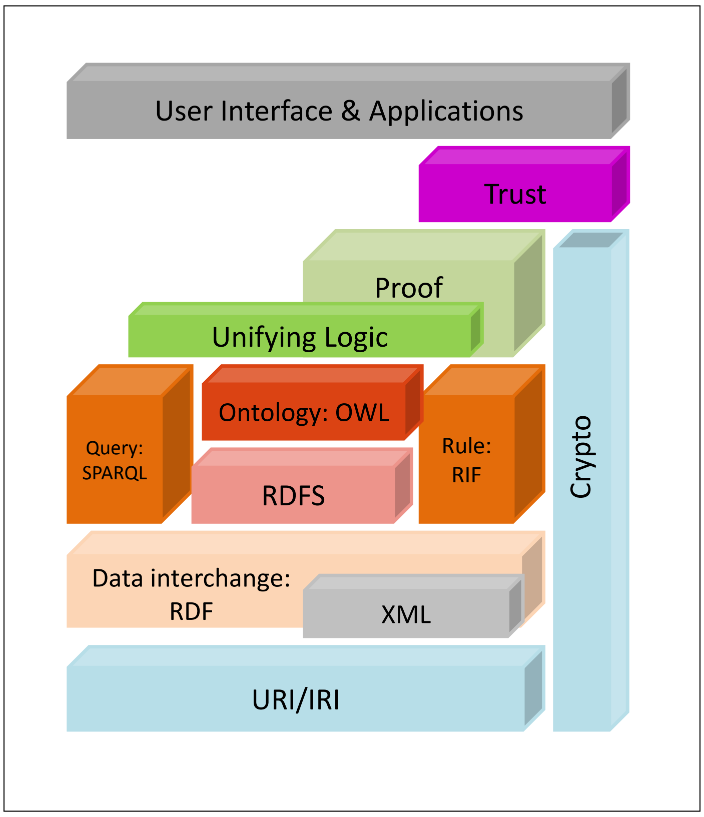


Fig. 1.1: Semantic Web Stack

As shown in Figure 1.1, Semantics are delivered by RDF (the Resource Description Framework) a generic data structure for machines to exchange machine readable data. RDF can be serialized using different serialization syntax such as

[7] www.w3c.org

RDF/XML, N-Triple and Turtle. These documents can then be exchanged across the Web and reused by other applications.

RDF provides a flexible mechanism for adding semantic annotation to structured documents on the Web. These annotations describe Web documents and state the relationships between them. RDF is a simple language; it encodes annotations in a set of triples. Each triple consists of three components known as subject, predicate and object. These components are respectively, like the subject, verb and object in human natural language. An RDF data structure is a labeled directed graph, each subject and object in a triple represent a pair of nodes, and the predicate represents the relationship between them. Figure 1.2 shows the relationship between a person and his email address identified by the relation 'has_email'. The subject of a triple can be either a blank node[8] or an IRI (Internationalized Resource Identifiers), the object can be a blank node, an IRI or a literal value such as string or integer. The predicate of a triple is always an IRI. Although RDF is a flexible language for adding annotations to Web documents, it provides limited support for processing the semantics of the terms used in annotations.

In order to model schema information, the Semantic Web offers RDFS Schema (RDFS), the lightweight ontology language [BG04], and more expressive Web ontology languages. RDFS provides special vocabularies to model class hierarchies (rdfs:subClassOf), and property hierarchies (rdfs:subPropertyOf). It also pro-

[8] A blank node holds no textual data but act as a means of gathering together other RDF nodes.

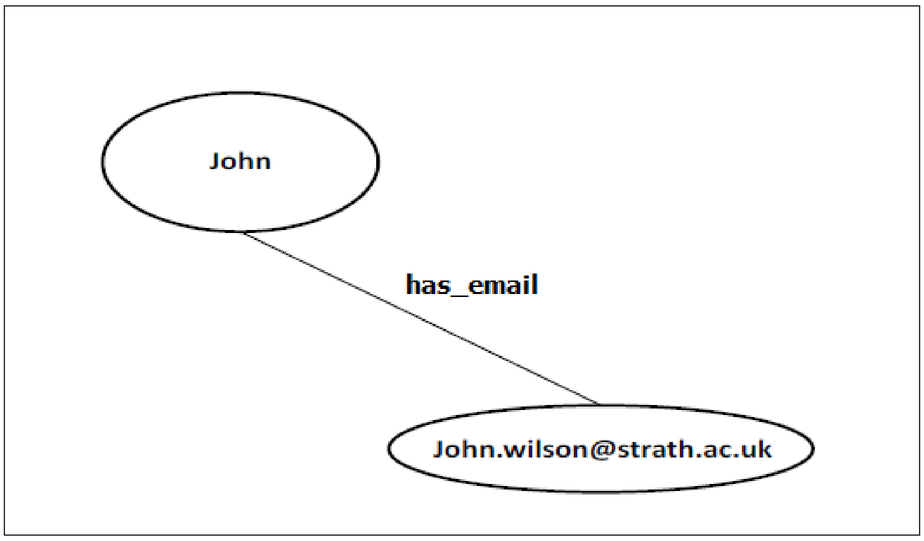


Fig. 1.2: A simple RDF graph describing the relationship between a person and an email address

vides the means to define domains (rdfs:domain) and ranges (rdfs:range) that allow for assigning a class as the subject and object of a relation with a given property respectively.

The need for a more expressive ontology language has resulted in the introduction of several languages including RDFS[9] SHOE[10], DAML+OIL [11], and OWL (Web Ontology Language) [PSHH+04]. The W3C standard and the broadly accepted Web ontology language of the Semantic Web is OWL which exploited the earlier work on OIL and DAML+OIL and integrated these languages with RDF. OWL has an RDF based syntax, which makes it hard to read, but that is a trade-off for its direct accessibility to Web applications. OWL is part of the ontology layer in the Semantic Web Stack introduced in Figure 1.1. OWL

[9] Resource Description Framework Schema language provides a basic class structure for RDF
[10] Simple HTML Ontology Extensions (SHOE) [HHL99]
[11] DARPA (US Defense Advanced Research Projects Agency) Agent Markup Language (DAML) and the Ontology Inference Layer [MFHS02]

has major benefits over RDF in that it is built on the basis of Description Logic (DL) which provides a logical formalism for ontologies and the Semantic Web by ensuring accuracy and consistency of data in certain domains.

The next two layers in the Semantic Web Stack are logic and the proof layer that establishes the truth of logical statements given in the lower layers, to allow automated reasoners to deduce unstated facts and conclusions. At the top of the Semantic Web Stack is the trust layer which is the high-level concept of the Semantic Web: the Semantic Web will achieve its full potential when users have trust in the data and the quality of information provided. Cryptography is an element that can be used to provide security throughout the Web stack. Each layer in the Semantic Web Stack is built on the layer below and tends to be more complex than the layers below it [BZC10].

1.2 Motivations

As RDF annotations (also known as RDF triples) will be shared between different agents[12], a common interpretation of the terms used in annotations is required, and this is the role of ontologies. An Ontology is a collection of definitions of concepts and their interrelationships that are used by different agents to provide a shared interpretation of the ontology content. Using ontologies for automatic processing in computers (or for the Semantic Web) requires an expressive and well-defined knowledge representation language that can provide a formal defini-

[12] In this work an agent is considered to be automated processes on the Web.

tion of the elements they contain. The Web Ontology Language (OWL) provides such a platform [AVH04].

Any change in a domain, in the conceptualization or in the specification requires a change in the ontology. Changes in the specification mean changes in how the conceptualization of the domain is represented (i.e. changes in the ontology representation language). Changes in the conceptualization of the domain occur as a result of a new requirement, observation or measurement, or a change in the usage of the ontology. The domain may also change at any time since the real world itself is dynamic. Ontology evolution is the process of modifying an ontology in response to a domain or a conceptualization change [FP05]. Data that represents instances described by the ontology is often stored as RDF. Collections of RDF are often very large[13]. An example of frequently updated RDF knowledge bases includes the Gene Ontology (GO)[14] which has grown to incorporate many databases including the world's repositories for plants, animals and microbial genomes. GO annotations are of very large sizes of more than 10^8 triples. This knowledge base is updated every month. Another example of large RDF knowledge bases is the Foundational Model of Anatomy[15], an ontology for the domain of human anatomy. It contains more than 85,000 classes, 140 relationships connecting the classes, and more than 120,000 terms, and has a frequent incremental updates [GGD13]. DBPedia[16] is another example of large RDF

[13] See http://www.w3.org/wiki/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics accessed on Saturday, 27 June, 2015
[14] http://geneontology.org/
[15] http://si.washington.edu/projects/fma
[16] http://wiki.dbpedia.org/

knowledge bases that has a frequent update. DBPedia is a large multi-domain ontology which has been derived from Wikipedia. DBPedia describes 3.77 million concepts with 400 million facts, and it is updated twice a year. Appendix A.1 contains foundational work carried out to generate RDF data from information retrieved in XML format which shows the verbosity of RDF data compared to XML.

Since the nature of ontologies is to evolve over time, rebuilding a complete collection of RDF when this change happens is going to be a problem particularly when these collections tend to be large. Therefore, managing ontology evolution is one of the fundamental issues to be addressed in the development of technologies to support the Semantic Web.

RDF collections can be stored and manipulated persistently in RDF triple stores. In triple stores, RDF data is typically stored as rows of triples. RDF can be viewed in different ways such as an in-memory model or as serialized RDF files which also can be in different RDF textual formats. There are two common types of RDF serialization in practice; triple-based and XML-based serialization.

Update propagation between two knowledge bases M and M' can be carried out by generating the deltas Δ (i.e. the differences) between the two knowledge bases and then applying it to M in order to update M to M'. In the context of RDF data structures the knowledge base M may be large and may be distributed to a number of different remote locations. It is therefore important to minimize the delta size since it may need to be distributed to the sites holding the knowledge base M.

1.3 Hypothesis and research questions

This work focuses on reducing the delta size, which is important for reducing the storage required to store these updates and also for reducing the bandwidth required to transfer these updates through the network.

Reducing the delta size in this work is based on the reduction of both the set of delete operations needed to propagate the update and the set of insert operations. It is important to maintain ontology consistency during this process. in order to produce minimized correct deltas.

Work in the realm of updates to RDFS knowledge bases indicates that it is possible to use rulesets that define the semantics for such knowledge bases to restrict the size of the data volume that is needed to update to successive versions of a knowledge base [ZTC11].

The hypothesis of this work is that improving RDF change detection techniques by exploiting both sets of updates (i.e. insertions and deletions) while retaining correctness when transforming one knowledge base (KB) into another will support reduction in the size of updates. Moreover, exploiting the hidden semantics in RDF KBs by applying complex rulesets such as OWL 2 (in our work, OWL 2 RL/RDF, a subset of OWL 2) could further support the production of small sized deltas. The process of reducing the delta size relies on reasoning with the OWL 2 RL/RDF ruleset [MGH$^+$09] which is more complex than RDFS rules as the application of these rules exploits more inferred information in the dataset, which in turn could avoid unnecessary updates and hence reduce the delta size.

To this end, the hypothesis leads to the following research questions:

1. What are the performance consequences of update strategies that can be applied to Semantic Web data collections? It is important to measure accurately the costs of different approaches because delta generation may need to be carried out 'on demand' or in a distributed peer context.

2. What is the best way of minimizing delta sizes using RDFS rules? The process of minimising deltas using RDFS rules need to be clearly defined so that the principles can be applied to more complex rulesets.

3. How can the richness of OWL 2 be exploited for the reduction of the delta size? The complexity of the OWL 2 ruleset suggests that new approaches to minimizing the deltas will be possible.

1.4 Contributions

This thesis makes several contributions:

1. The first contribution of this work is a comparative study of different strategies for using semantic content to generate RDF updates in the context of the overall performance of the process. These strategies are: the explicit change detection EC, the Backward-chaining change detection BC, the pruning and backward-chaining change detection and the forward-chaining

change detection. The evaluation has been performed using different versions of Uniprot Taxonomy, a real dataset from the bioinformatics domain.

2. the second contribution of this work is an approach for using the smallest deltas that will maintain the consistency of an RDF knowledge base. This new method is called correct dense delta (ΔD_c), which improves the unsound dense delta method (ΔD). The performance of this work is evaluated using real-world datasets from both Gene Ontology (GO) and Uniprot Taxonomy which were enhanced by synthetic data using our RDF generator.

3. The third contribution of this work is to assess the impact of pruning rules in the OWL 2 ruleset in the context of their use in reducing the size of updates needed to transform ontologies between versions. This part of the work considers blank nodes when computing the deltas. The evaluation of this work performed using two benchmark datasets LUBM and UOBM.

1.5 Organization of the thesis

The remainder of the thesis proceeds as follows: Chapter 2 provides the necessary background that will be of use in the subsequent chapters and to simplify the understanding of the proposed approaches. This will include a literature review on the different types of ontology change and an introduction to the existing change detection techniques. Chapter 3 gives a review of the related work. Chapter 4 is the main core of this thesis. It (a) analyzes and investigates the process of updating RDF views (b) provides a detailed analysis of the different change detection techniques and investigates the cost of pruning in these techniques (c) proposes a correction method for dense delta that retains ontology consistency (d) extends the ruleset used in change detection by applying more complex rules in the context of OWL 2 ontologies and proposes a pruning method that prunes unnecessary rules. Chapter 5 reports and discusses experimental results. Chapter 6 summarizes the results of this thesis and identifies topics that are worth further work and research.

2. PRELIMINARIES

The Semantic Web is the new generation of the Web that enables machines to automate, integrate and exchange information. In this thesis, four main categories of the issues related to the Semantic Web are addressed: Ontologies, which are the way of bridging between human knowledge and machine representation, languages for representing ontologies in the Semantic Web, managing and detection of change, and reasoning with OWL.

2.1 Ontologies

Since the Semantic Web is a transformation of the current Web of information and data into the Web of Knowledge [Dev06], this knowledge needs to be captured, processed, integrated and reused. Ontologies are a knowledge representation scheme which supports all these tasks. Ontologies have become common on the World Wide Web. Many disciplines now develop standardized ontologies (see for example schema.org) that domain experts can use to annotate information in their field in a machine-interpretable language for software agents to process and share.

Ontologies define a common vocabulary that is used for sharing information in a certain domain. They consist of machine-interpretable definitions of concepts in the domain and the relationships among them. Developing ontologies is useful for the sharing of common understanding of the structure of information among people and software agents. Ontologies also enable the reuse of knowledge in certain domains. For example, the FOAF ontology[1] provides a sharable definition of personal details that can be used in a number of different contexts. Another common use of ontologies is separating domain knowledge from functional knowledge.

In philosophy, Ontology is a theory about the nature of existence [BLHL01]. In the literature of Artificial Intelligence there are many definitions of ontology. The most widely used definition is by Gruber [Gru93] "an ontology is a specification of conceptualization". That is, an ontology is a formal description of the concepts and relationships that can exist in a domain of discourse. Common components of ontologies include classes (also called concepts), properties that describe the variety of features of the classes (also called attributes or roles) and restrictions on properties (also called role restrictions). Instances of individual classes are entities of the same class, but individuals are not an explicit part of the ontology. An ontology that includes these components together with a set of individual instances of classes constitutes a knowledge base.

For Web researchers, an ontology is a vocabulary, semantic interconnections, and some simple rules of inference and logic for a particular topic [Hen01]. Informally, an ontology of a certain domain consists of a vocabulary which includes concepts, or classes of objects. It helps to understand what different things are in

[1] http://www.foaf-project.org/

a domain and how they are classified. Semantic interconnections, which are links between concepts help to show the relation between the different things in that domain. In addition, inference and logic rules enable some forms of reasoning and validation.

An ontology is not a stand alone structure or a goal in itself. It is typically text-based files that can be stored somewhere on the Web and consulted by different services and applications as necessary. Ontologies can be represented using machine-readable languages which typically are XML-based files. They also can be represented graphically using visual languages and tools [Dev06].

An example of a simple ontology that describes the knowledge of a schools system is given in Figure 2.1. The classes are Person, StaffMember, Teacher, Student, and, Course. The properties are the is-a property which identifies the subsumption relationship between classes and the two user-defined properties assigned, and enrolled. These properties are shown as arrows connecting two classes. The source and destination ends of an arrow are called the domain and the range respectively.

Classes are the main component of ontologies. They describe and classify concepts in the domain. For example, the class of StaffMember includes instances of all staff members. Specific staff members are instances of this class. For example, John is an instance of the StaffMember class. Classes can have subclasses that include concepts that are more specific than their superclasses. The StaffMember class can have a Teacher class as its subclass. The Teacher class includes instances

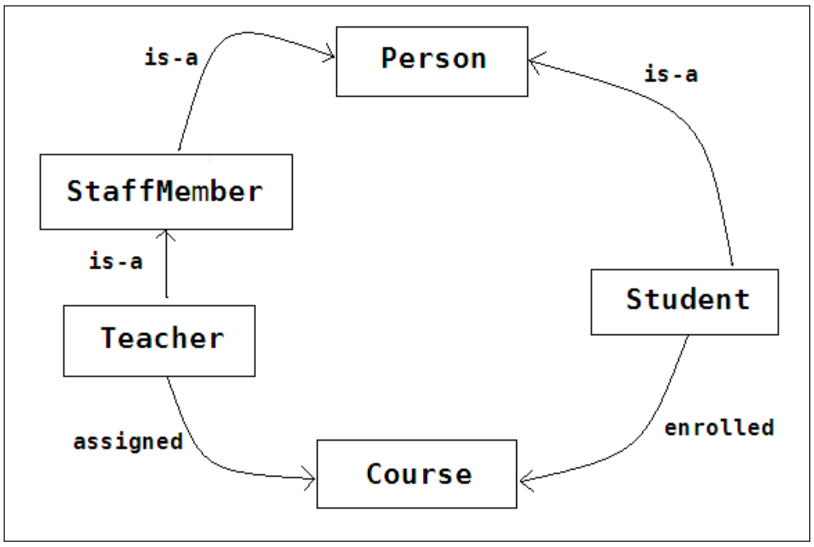


Fig. 2.1: School ontology

of all members of staff who do teaching. Classes also can have superclasses that include concepts that are more general than their subclasses. For example, the StaffMember class can be subclass of a more general class called Person. The same thing applies to the Student class, which is also a subclass of the Person class. In typical real world ontologies, all classes are derived from the superclass Thing. Properties describe features of classes and instances and the relationships between classes. In the school example, instances of the class Teacher have a property ‘assigned’, the range of which is an instance of the class Course. Another property in our example is the ’enrolled’ property which describes the class Student and its range is an instance of the class Course. The is-a relationship describes the subclass-superclass relationship described earlier.

2.1.1 OWL as an ontology language

OWL is the most recent development in standard ontology languages from the World Wide Web consortium (W3C). OWL ontologies have similar components to ontologies built using other data modeling languages such as the UML (Unified Modeling Language) but in OWL the terminology used to describe these components is different. OWL consists of classes, properties, and individuals. Whereas in UML the components are things, relationships and diagrams. This section describes how to build the school ontology in OWL using Protégé[2]; an open-source platform that provides tools for constructing domain models and knowledge-base applications with ontologies.

Creating a new ontology includes defining classes and the class hierarchy, defining properties of classes, and describing property restrictions.

As mentioned earlier, classes are the main building blocks of an OWL ontology. In Protégé 4, the initial class hierarchy tree view contains one class called Thing as the superclass for all other classes. In the school example, the classes are Person, StaffMember, Teacher, Student, and Course. The Teacher class is a subclass of the StaffMember class, and both the StaffMember class and the Student class are subclasses of the Person class.

Classes can be disjoint from each other which means that an individual can not be an instance of more than one of these classes. For example, in the school ontology (student, course), (Teacher, Course), (Teacher, Student), (StaffMember,

[2] http://protege.stanford.edu

Student), (Staffmember, Course) and (Person, Course) are all disjoint classes.

Properties in ontologies represent relationships between classes. OWL can define three types of properties, Object properties, Datatype properties and Annotation properties. Object properties are relationships between two individuals, Datatype properties describe the relationship between individuals and data values such as an XML Schema Datatype value or RDF literal. Annotation properties are used to add information to classes, individuals and object/datatype properties. In our school ontology we have three object properties; 'assigned' which links the Teacher class and the Course class, 'hasTutor' links the Course class to the Teacher class, and 'enrolled' links Student and Course.

OWL object properties can have various characteristics to provide information concerning properties and their values. These properties include:

- Inverse Properties: One property can be the inverse of another property. If Property P1 is the inverse of Property P2, and if property P1 links individual X to individual Y then the inverse property P2 will link individual Y to individual X.

- Functional Properties: If a property is functional, then it may have at most one value for each individual.

- Inverse Functional Properties: This characteristic for a property means that the inverse property is functional.

- Transitive Properties: If a property P is transitive, and it links individual X to individual Y, and also links individual Y to individual Z, then property P will link individual X to individual Z.

- Symmetric Properties: If a property P is symmetric, and it links individual X to individual Y, then the same property P will link individual Y to individual X.
- Asymmetric Properties: If a property P is asymmetric, and individual X is linked to individual Y via property P, then property P cannot link property Y to property X.
- Reflexive Properties: A property P is stated to be reflexive if the property links individual X to itself.
- Irreflexive Properties: A property P is irreflexive if it links individual X to individual Y, where X and Y are not the same.

Properties in the school ontology may be characterized from the above definitions, for example, assigned property may be characterised as asymmetric, irreflexive, and hasTutor is its inverse property. hasTutor on the other hand is functional, asymmetric and irreflexive. The last property in our example is enrolled which links Student to Course and is characterised as asymmetric and irreflexive.

2.2 OWL 2 Web Ontology Language

Since the introduction of OWL, a number of reasoners, such as FacT++[TH05], Pellet[SPG$^+$07], RACER[HM01] and HermiT[SMH08], and a number of ontology editors, such as Protégé and Swoop[KPS$^+$06], have been developed. Despite the success of OWL, users and designers of OWL tools and APIs have identified certain problems with OWL such as expressivity and practical limitations of the language. These problems produced a need for a revision of OWL 1.

OWL 2 has received considerable attention leading to the development of tool support for the language. For example, Protégé has been extended to support the additional features provided by OWL 2, and also reasoners such as FaCT++ and Pellet systems provide support for OWL 2 inference.

Computing all possible conclusions of an OWL 2 ontology can be a challenging and sometimes undecidable problem. To address this problem, OWL 2 has also come in three lightweight sublanguages called profiles (summarized in Table 2.1): OWL 2 EL, OWL 2 QL and OWL 2 RL. These profiles are designed by applying syntactic restrictions on the full features of OWL 2 in order to simplify the reasoning process. The choice of which profile to use in practice depends on the reasoning task and the structure of the ontology.

OWL 2 EL is designed particularly for application employing ontologies with a very large number of properties and/or classes. In this subset of OWL 2, the basic reasoning problems can be performed in polynomial time with respect to the size of the ontology.

OWL 2 QL is used for applications that contain very large volumes of instance data and where query answering is the most important reasoning task. Sound and complete query answering can be performed in polynomial time with respect to the size of the assertions.

OWL 2 RL is aimed at applications that require scalable reasoning and a limited extension of both RDF and RDFS is desired. Reasoning with OWL 2 RL can be implemented using rule-based reasoning engines. Reasoning and query answering problems can be performed in polynomial time with respect to the size of the ontology. OWL 2 RL/RDF is different from the rest of the profiles. With OWL 2 RL/RDF we no longer talk about syntactic restriction of OWL 2, but a semantic restriction of OWL 2 RDF-based Semantics.

Users of OWL 2 can choose between two slightly different semantics: The Direct semantics and the RDF-based semantics. There are two way of assigning meaning to ontologies in OWL 2. Direct semantics can be applied to ontologies in OWL 2 DL, a subset of OWL 2 and can be regarded as a syntactically restricted version of OWL 2 Full which is designed to allow more RDF graphs or OWL 2 full ontologies to be valid OWL 2 DL ontologies. However, OWL 2 DL could also be expressed using RDF-based semantics. In general, conclusions drawn using the Direct semantics are still valid under the RDF-based semantics while not all conclusions drawn by RDF-based semantics are valid under Direct semantics as under the RDF-based semantics there are some extra conclusions are derived from viewing the ontologies as RDF graphs.

On the other hand ontologies in OWL 2 Full, another subset of OWL 2, can only be interpreted under the semantics of RDF-based Semantics. Direct semantics allow OWL 2 ontologies to be expressed using description logic (DL), a fragment of first-order logic. RDF-based semantics is an extension of RDFS semantics and is based on viewing OWL 2 ontologies as RDF graphs. The differences between direct semantic and RDF-based semantics is that Direct semantics is not applicable to all RDF databases that use OWL features i.e. cannot be used with arbitrary RDF graphs.

Regarding decidability when designing a reasoner for OWL 2 ontologies, OWL 2 DL under the Direct semantics is decidable which make designing a reasoner that answers all yes-no questions possible. On the other hand, OWL 2 Full under RDF-based semantics is undecidable, thus there are no such reasoners for OWL2 Full under RDF-based semantics.

OWL 2 profiles, (i.e. OWL 2 EL, OWL 2 QL and OWL RL) are syntactic fragment of OWL 2 DL. Thus, every conforming OWL 2 DL reasoner is also a conforming reasoner for OWL 2 QL, OWL 2 EL and OWL 2 RL. However, none of OWL 2 profiles is a subset of another. Both OWL 2 semantics, the direct semantics and the RDF-based semantics, can be used for any of these profiles, but the use of Direct semantics for OWL 2 QL and OWL 2 EL, and RDF-based Semantics for OWL 2 RL is most common in practice.

Unlike other OWL 2 profiles, OWL 2 RL/RDF, is not a syntactic restriction of OWL 2 DL, it is rather a semantic restriction of OWL 2 RDF-based Semantics.

Tab. 2.1: OWL 2 languages

Language	Semantics	Syntax	Complexity
OWL 2 DL	OWL 2 Direct semantics/OWL 2 RDF-based Semantics	Syntactical restriction of OWL 2 Full	Decidable
OWL 2 Full	OWL 2 RDF-based Semantics. An extension of RDFS semantics	RDF graphs	Undecidable
OWL 2 EL	Typically, not exclusively, interepreted under OWL 2 Direct semantics	Syntactic restriction of OWL 2 DL	Decidable
OWL 2 QL	Typically, not exclusively, interepreted under OWL 2 Direct semantics	Syntactic restriction of OWL 2 DL	Decidable
OWL 2 RL	Typically, not exclusively, interepreted under OWL 2 RDF-based Semantics	RDF graphs	Decidable
OWL 2 RL/RDF	Semantic restrection of OWL 2 Full semantics	ruleset applied to generalized RDF graphs	Decidable

However, reasoners are allowed to extend the semantics of the ruleset up to the level of RDF-based semantics. In [MGH$^+$09], OWL 2 RL/RDF is defined as a partial axiomatization of the OWL 2 RDF-based semantics in the form of first-order implications [3]. This axiomatization contains a large set of triple rules where each antecedent and consequent in these rules consists of an RDF triple pattern. Thus, the syntax of OWL 2 RL/RDF is a generalized RDF graphs. Work in this thesis focuses on the implications of OWL 2 RL/RDF rules in the generation of KB updates.

[3] The axiomatization is called OWL 2 RL/RDF rules which is given as universally quantified first-order implications over a ternary predicate T.

2.3 Managing ontology changes

Ontologies can become large structures, and due to this fact, many research problems may appear during the development and maintenance of ontologies. Such problems include the challenge of modifying an ontology in order to meet a certain need or a specific requirement. This problem is referred to as ontology change in [FMK+08]. This process includes any type of change to the ontology whether this change was in response to external events, such as changes in order to meet the requirements of an enhanced system, or internal events, such as changes forced by heterogeneity considerations (i.e. differences in terminology, language or syntax between ontologies).

Several overlapping research topics have appeared in order to deal with the problem of ontology change, such as (ontology evolution, alignment, merging, mapping, integrating etc.), causing a confusion in understanding the meaning and the uses of the terms in these areas. The ontology change problem can be classified into four groups according to the need for change [FMK+08]:

- heterogeneity resolution, which deals with resolving heterogeneity of ontologies which in turn contains five subfields: ontology mapping, alignment, articulation, morphism and translation.

- ontology editing that consists of any type of modification to the ontology such as modification in order to resolve inconsistencies or incoherences (ontology debugging) or changes in response to a change request (ontology evolution).

- ontology fusion which deals with the problem of combining two ontologies. The source ontologies can be from similar domains (ontology integration) or identical domains (ontology merging).
- ontology versioning which copes with the different versions of ontology by creating and managing different variants of it.

2.3.1 Heterogeneity resolution

Heterogeneity of the Semantic Web means that different ontologies use different terminologies, languages and syntax to refer to the same entities. Work related to the heterogeneity resolution tries to solve this problem by providing a set of translation rules to make two ontologies use the same name for similar entities and different names for different entities [ELBB^{+}04] [KS03] [SE05].

Although research fields related to solving the heterogeneity problem by providing a set of translation rules do not seem to make any direct change to the ontology, they are considered subfields of ontology change for two reasons: First, changing an ontology in response to a new information requires both the ontology and the new information to use the same terminology, language and syntax. Second, the change of an ontology in heterogeneity resolution is implicitly performed as each new collection of information (i.e. new ontology) is changed locally to fit the source ontology.

The research fields under the heterogeneity resolution group are: ontology mapping, morphism, alignment, articulation and translation.

Ontology mapping Given an ontology as a pair $< S, A >$ where S is the signature[4] and A is the set of ontological axioms, ontology mapping is defined as the task of relating the signature of two ontologies that share the same domain of discourse in such a way that the mathematical structure of ontological signatures and their intended interpretations, as specified by the ontological axioms, are respected. As a result a set of functions on the ontology signature is produced to map between the ontologies' vocabularies [KS03].

Ontology morphism Since ontology mapping is restricted to signatures, the approach that deals with both signatures and axioms of the ontologies is called ontology morphism [KS03].

Ontology alignment It is also called ontology matching. In ontology alignment two ontologies are related to each other using a set of relations. It is defined as the process of finding relationships between ontologies' signatures [Flo07] [SGS$^+$06].

Ontology articulation Each binary relationship in ontology alignment could be decomposed into a pair of ontology mappings from a common intermediate source. This process is called ontology articulation and is defined as the process of determining the intermediate ontology and the two mappings to the initial ontologies [Flo07] [KS03].

[4] An ontology signature is the ontology vocabulary that is modeled as a simple set containing the names of all concepts, properties or individuals that are related to a particular domain [Flo07].

Ontology translation The term ontology translation refers to two different processes [Flo07]. The first one changes the syntax of the axiom part of an ontology but not the signature, whereas the second process translates the signature part of an ontology. Translating the signature part may lead to a confusion when thinking of ontology mapping, but to avoid this confusion, it is worth mentioning that ontology mapping provides a set of functions that relates two ontologies' signatures, while the translation of the signature implements the mapping by applying these functions.

2.3.2 Ontology editing

Ontology editing is the process of modifying the ontology to resolve inconsistencies or incoherences that may occur in an ontology, which in this case is called ontology debugging, or to modify the ontology in response to a change request and this type of modification is called ontology evolution.

The purpose of ontology debugging is to find techniques that would remove contradiction from an ontology which may occur as a result of modeling errors or changing the ontology [HQ07].

Ontology evolution on the other hand refers to the process of incorporating new knowledge in an ontology. Six phases of ontology evolution have been identified in [SMMS02]:

- change detection: in this phase changes in the ontology are identified.
- change representation: changes are formally represented.

- semantics of change: the consequence of implementing any change is identified to guarantee the validity of the ontology.
- change implementation: involves the implementation of the changes using appropriate tools.
- change propagation: this phase ensures that all changes are propagated to all dependent agents.
- change validation: this phase allows the ontology engineer to review the implemented changes and undo any changes if required.

In [PDT05] a similar approach which identifies five phases for ontology evolution:

- change request: this phase involves the identification of required changes.
- change implementation: this phase executes the requested changes on the ontology. All applied changes are stored into an evolution log to keep track of these changes.
- change detection: this phase involves the identification of any changes that may occur as a consequence of modifying the ontology in the previous phase.
- change recovery: in this phase, changes from the change request phase are checked and revised if necessary.
- change propagation: changes in the evolution log are propagated to dependent parties.

2.3.2.1 Ontology versioning

Ontology versioning is considered a variation of ontology evolution [HS04]. The only difference between them is that in the ontology versioning process the original ontology and all its versions are stored after changes are performed. Thus, allowing access to the different versions of the ontology.

2.3.3 Ontology fusion

Ontology fusion refers to the construction of a new ontology by fusing the information found in two or more ontologies. Based on the level of similarity between the source ontologies, ontology fusion is classified into two different research areas: ontology merging and ontology integration [Flo07].

2.3.3.1 Ontology merging

Ontology merging is the process of combining two or more ontologies containing identical or overlapped information in order to create a one large ontology. Ontology merging and ontology alignment are often understood as the two sides of the same coin, but the only difference between them is that the result of ontology merging is one new and large ontology, whereas in ontology alignment the source ontologies remain with links of relations between them. Example of the tools used for ontology merging are PROMPT and Chimaera, which are based on a semi-automatic approach that proposes a set of suggestions on how source ontologies should be combined together leaving the final choice to the ontology engineer.

2.3.3.2 Ontology integration

Ontology integration is the process of constructing a global ontology by combining a number of local ontologies covering loosely related domains. The domain of the resulting ontology is more general than the domain of any of the source ontologies. In contrast, the domain resulting from applying ontology merging process is more specific and gives detailed information on a given topic as the source ontologies are highly overlapped. Fused ontologies are generally heterogeneous in terms of vocabulary, syntax, representation languages etc., and considering heterogeneity issues is a common practice when combining different ontologies, but modeling differences need to be taken into account specially when dealing with highly interconnected (i.e. identical) ontologies as the meaning of the terms are different even if the same terminology is used, which might lead to an invalid or inconsistent ontology. Ontology merging (or fusion in the general sense) can be done by applying five steps:

- the identification of the semantic overlap between the source ontologies.
- a transformation (heterogeneity resolution) agreement in terms of the terminology, syntax and representation.
- design and implementation.
- the union of the sources is taken.

- the evaluation of the resulting ontology in terms of consistency, redundancy and validity.

2.4 Change detection

In the Semantic Web, data that represents instances described by an ontology is typically stored as RDF. On the Web, RDF collections often contain an extremely large number of triples, and the size of the collection may increase rapidly. Changes in the ontology may require changes in the underlying RDF data. Since RDF is, in general, designed to easily integrate information from diverse data sources, changes to RDF may occur in distributed environments. Therefore, in order to cope with the evolving nature of the Semantic Web, it is important to efficiently detect these changes and update RDF accordingly. In particular where RDF collections are replicated, it is important to be able to distribute updates quickly.

The straightforward algorithms for version control systems such as CVS [B$^+$90] and the traditional change detection techniques based on structured data such as XML are not able to handle RDF data. This is because RDF models represent graphs which can be serialized into different text formats and these graphs are enriched with semantic information including inferred knowledge [ZTC07]. An RDF graph is a set of triples (subject, predicate, object) each triple represents a relation between two nodes. The set of triples can be serialized in different formats, which means that two RDF graphs can be identical in terms of the information they represent but have different text representation. In addition, RDF data is enriched with semantic information.

Berners-Lee *et al.* proposed a non text-based change detection scheme for RDF based on comparing two RDF graphs and computing the differences between them, which generates a set of differences called deltas. These differences are a set of change operations (i.e. insertions and deletions) which transform one RDF graph into another [BLC04]. This change detection method generates the deltas using set arithmetic for RDF graphs [BLC04]. For example, If M and M' are RDF models, then the delta that transforms M to M' is modeled as a set of triple insertions and triple deletions where insertions is the set difference $M' - M$ and deletions is $M - M'$.

However, this approach does not handle the semantic level of RDF and does not exploit the inferred knowledge for the reduction of the produced deltas but generates the explicit differences between two RDF models. In addition, the cost of storing the delta or exchanging it through the network using this method is linear with the size of the differences between two RDF models [BLC04]. Therefore, several approaches for calculating the delta have been proposed based on methods to minimize the delta size in order to reduce the required bandwidth and storage space for updating RDF data collections [NM02, VG06, BK03, ILK13]. These approaches aim to minimize the delta size by exploiting the semantics of RDF data. This can be carried out by applying inference rules under the RDFS specification to the triple set [HM04]. These inference rules are applied to the

M	M'
(Graduate subClassOf Person),	(Head_Teacher subClassOf Teacher),
(Student subClassOf Person),	(Teacher subClassOf Staff),
(Head_Teacher subClassOf Staff),	(Staff subClassOf Person),
(Teacher subClassOf Staff),	(Graduate subClassOf Student),
(Staff subClassOf Person),	(Student subClassOf Person),
(John type Student).	(Teacher subClassOf Person),
	(Head_Teacher subClassOf Person),
	(John type Person).

Fig. 2.2: Sample data structure before and after update

existing RDF triples in order to derive new ones, a process known as the RDF closure.

Definition 2.4.1 (Closure)**.** Let t be a triple with subject, predicate, object (SPO). The closure of M is defined as M extended by those triples that can be inferred from the graph M. The closure of an RDF graph M is denoted by:

$$C(M) = M \cup \{t \in (SPO) \mid M \models t\}$$

Example 1. *Let $M = \{a\ subClassOf\ b, b\ subClassOf\ c\}$ then the closure of M will contain these triples and a further triple $\{a\ subClassOf\ c\}$.*

Considering this closure (i.e. inferred set of triples) when calculating the set-differences between two RDF models may reduce the size of the produced delta because adding triples to a delta when these triples can be inferred in the updated knowledge base is unnecessary.

The basic operations in change detection techniques are the set-difference operation and the inference operation. Different change detection techniques can be classified based on the inference strategy (i.e. the computation of the closure). The existing strategies are the Explicit delta, the Closure delta, the

$$\begin{aligned}\Delta E = \; & \{\text{Del (Graduate subClassOf Person)},\\ & \text{Del (Head\_Teacher subClassOf Staff)},\\ & \text{Del (John type Student)}\}\\ \cup \; & \{\text{Ins (Head\_Teacher subClassOf Teacher)},\\ & \text{Ins (Graduate subClassOf Student)},\\ & \text{Ins (Teacher subClassOf Person)},\\ & \text{Ins (Head\_Teacher subClassOf Person)},\\ & \text{Ins (John type Person)}\}\end{aligned}$$

Fig. 2.3: The explicit delta

$$\begin{aligned}\Delta ED = \; & \{\text{Del (John type Student)}\}\\ \cup \; & \{\text{Ins (Head\_Teacher subClassOf Teacher)},\\ & \text{Ins (Graduate subClassOf Student)},\\ & \text{Ins (Teacher subClassOf Person)},\\ & \text{Ins (Head\_Teacher subClassOf Person)},\\ & \text{Ins (John type Person)}\}\end{aligned}$$

Fig. 2.4: The explicit dense delta

Dense delta, the Dense & Closure delta and the Explicit & Dense delta, ΔE, ΔC, ΔD, ΔDC and ΔED, respectively. The computation of the deltas between two RDF datasets M and M' in ΔC and ΔDC requires computing the closure of both sets and then in ΔC the set-difference operation is performed between the two closures for producing the deletion set (i.e. $C(M) - C(M')$) and the insertion set (i.e. $C(M') - C(M)$), while in ΔDC, the set-difference operation is performed between the two closures for producing the deletion set (i.e. $C(M) - C(M')$) while the insertion set is produced using the set-difference operation between M' and the closure of M (i.e. $M' - C(M)$). Thus, both ΔC and ΔDC may produce larger delta sizes compared to the other change detection techniques because they produce deltas that not only include the differences between the datasets but also the differences between the inferred triples from both datasets. In this thesis, we focus on inference-based change detection which produces the smaller delta [ZTC11] [ILK13]:ΔED and ΔD_c (a method that is proposed in this thesis that is an improved version of ΔD. This improved method produces a correct delta). However, it is worth mentioning that ΔC, ΔD, ΔDC and ΔED are inference based strategies apart from ΔE which is syntactic-based. The ΔE, ΔED and ΔD are explained here in the context of the two example RDF models

M and M' in Figure 2.2

As explained above, the naïve way of generating the delta involves computing the set-difference between the two versions using the explicit sets of triples forming these versions. This is called explicit delta (ΔE) which calculates the delta at the syntactic level of RDF and provides a set of triples to be deleted from and inserted into M in order to transform it into M'.

Definition 2.4.2 (Explicit delta)**.** Given two RDF models M and M', let t denote a triple in these models, Del denote triple deletion which is calculated by $M - M'$, and Ins denote triple insertion which is calculated by $M' - M$. The explicit delta is defined as:

$$\Delta E = \{Del(t) \mid t \in M - M'\} \cup \{Ins(t) \mid t \in M' - M\}$$

From the example in Figure 2.2, the delta obtained by applying the above change detection function is shown in Figure 2.3. Executing these updates against M will correctly transform it to M'. However, this function handles only the syntactic level of RDF and does not exploit its semantics. In the latter context, executing some of the updates in ΔE is not necessary as they can still be inferred from other triples. For instance, we can observe from the example in Figure 2.2 that deleting *(Graduate subClassOf Person)* from M, in order to transform it into M', is not necessary as this triple can still be inferred from the triples *(Graduate subClassOf Student)* and *(Student subClassOf Person)* in M'. Since this update is not necessary, it is useful to remove it from the delta. RDF data is rich in semantic content and exploiting this in the process of updating RDF models can minimize the delta size and therefore the storage space and the time to synchronize changes between models. In particular this approach handles differences between two versions and does not handle aggregated differences between multiple versions.

Unnecessary updates can be avoided by applying a differential function that supports reasoning over the closure of an RDF graph. In RDF inference, the closure can be calculated in order to infer some conclusions from explicit triples. This process is carried out by applying entailment rules against the RDF knowledge base. In this work, we consider the RDFS entailment rules provided by the RDFS semantics specification [HPS14]. This specification contains 13 RDFS entailments rules. However not all these rules are used in this work, only rules that derive triples with subClassOf, subPropertyOf and type relations. These rules are shown in Table 2.2. Other rules that do not have an effect on minimizing the delta size are excluded from the current approach for change detection [ILK13]. An example of these rules in the RDFS entailment rules is the rule: $(xrdf : typerdfs : Class \Longrightarrow xrdfs : subClassOfrdfs : Resource)$ which cannot reduce the delta size.

Thus, in contrast with ΔE, ΔED obtains a set of insertions by performing

	If KB contains	**Then add to KB**
rdfs1	s rdf:type x *and* x rdfs:subClassOf y	s rdf:type y
rdfs2	x rdfs:subClassOf y *and* y rdfs:subClassOf z	x rdfs:subClassOf z
rdfs3	p rdfs:subPropertyOf q *and* q rdfs:subPropertyOf r	p rdfs:subPropertyOf r

Tab. 2.2: Relevant rules

the set-difference operation $M' - M$. While a set of deletions is obtained by computing the closure of M' (denoted as $C(M')$) first and then performing the set-difference operation $(M - C(M'))$.

Definition 2.4.3 (Explicit dense delta)**.** Let M, M', $Del(t)$, $Ins(t)$ be as stated in Definition 2.5.2. Additionally let $C(M')$ denote the closure of M'. ΔED is defined as:

$$\Delta ED = \{Del(t) \mid t \in M - C(M')\} \cup \{Ins(t) \mid t \in M' - M\}$$

Applying this function to the example in Figure 2.2 produces the delta shown in Figure 2.4. The inserts in this delta are achieved by explicitly calculating the set difference $M' - M$ to provide the set of triples that should be inserted to M in order to transform it into M'. On the other hand, the set of deleted triples is achieved by calculating the closure of M' using the RDFS entailment rules to infer new triples and add them to M'. From the example, the inferred triples in M' are:

(Teacher subClassOf Person)

(Head_Teacher subClassOf Person)

(Head_Teacher subClassOf Staff)

(Graduate subClassOf Student)

These inferred triples are then added to M' to calculate the set difference $M - C(M')$ which results in only one triple to delete: *(John type Student)*. The number of updates produced by this delta is smaller than the one produced by the ΔE as a result of the inference process.

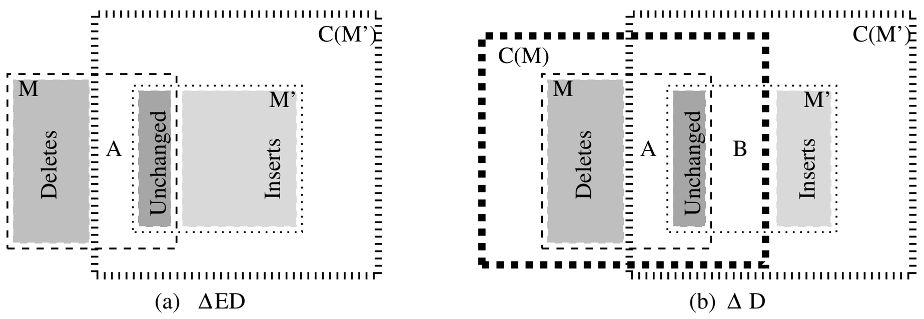
The effect of the inference process in minimising ΔED is limited to applying the inference rules when computing the deleted set of triples only. Applying inference rules for computing the inserted triples may further reduce the number of updates. For example, inserting the three triples *(Teacher subClassOf Person)*, *(Head_ Teacher subClassOf Person)* and *(John type Person)* into M may not be necessary because these triples implicitly exist in M and can be inferred in M using the RDFS entailment rules. In this example, applying *rdfs1* to M would infer *(John Type Person)* while the other two triples could be inferred using *rdfs2*. The application of inference over both the insert and delete sets produces the *dense delta* (ΔD).

Definition 2.4.4 (Dense delta)**.** Let M, M', Del(t), Ins(t) be as stated in Definition 2.5.2. The dense delta is defined as:

$$\Delta D = \{Del(t) \mid t \in M - C(M')\} \cup \{Ins(t) \mid t \in M' - C(M)\}$$

Figures 2.5(a) and 2.5(b) illustrate the distinction between ΔED and ΔD. In the former only the deletes that are not in $C(M')$ need to be carried out. In this case, $C(M)$ is not checked to see whether all of the planned inserts need to be applied. In the case of ΔD, deletes are handled in the same way as in ΔED however inserts are only applied if they are not in $C(M)$. This results in minimising both delete and insert operations.

From the example in Figure 2.2, the updates generated by applying (ΔD) are



A – deletes that are still in C(M') once M' has been generated
B – inserts that are already in C(M) before it is updated

Fig. 2.5: The distinction between ΔED and ΔD.

ΔD = {Del (John type Student)}
∪ {Ins (Head_Teacher subClassOf Teacher),
Ins (Graduate subClassOf Student) }

Fig. 2.6: The dense delta (ΔD)

shown in Figure 2.6. ΔD is smaller than either ΔE or ΔED with only three updates to transform M to M'. However, in contrast to ΔE and ΔED, ΔD does not always provide the correct delta to carry out the transformation. In this case, applying ΔD to transform M into M' will transform M as shown in Figure 2.7. This delta function does not correctly update M to M' because when applying the updates, *(John type Person)* is not inserted into M and cannot be inferred in M after the triple *(John type Student)* has been deleted.

By carrying out this approach, minimizing the delta size could be extended to handle inference over the delta itself. For example, the difference between two models M and M' is:

$\Delta E = Ins(A\, subClassOf\ B), Ins(B\ subClassOf\ C), Ins(A\ subClassOf\ C)$

which are the triples that exist in M' and do not exist in $C(M)$. Handling inference over ΔE can reduce the triple $(A\, subClassOf\ C)$ which can be derived from the other two triples in the delta.

	M
Original triples	(Graduate subClassOf Person), (Student subClassOf Person), (Head_Teacher subClassOf Staff), (Teacher subClassOf Staff), (Staff subClassOf Person), ~~(John Type Student).~~
Inserted triples	(Head_Teacher subClassOf Teacher), (Graduate subClassOf Person).

Fig. 2.7: Incorrect updates

RDF change detection techniques can also be classified based on the order of the basic operations; the set-difference operation and the inference operation, into forward-chaining change detection and backward-chaining change detection.

The forward-chaining approach follows the *inference-then-difference* strategy, which computes the entire closure first and then calculates the set-differences. Unlike the forward-chaining approach, the backward-chaining approach uses the *difference-then-inference* strategy, which instead of computing the entire closure, this method applies first the set-difference operations and then the inferencing on the result of the set-difference operation. This would be expected to improve the time and space required in change detection because only a small part of the closure is computed, not the full closure as in the case of forward-chaining approach (the two approaches are further explained in Chapter 4).

As explained above, the calculation of RDF closure is based on applying the RDFS entailment rules provided by the RDFS semantics specification. The entailment rules infer new RDF statements based on the presence of other statements.

However, although the backward-chaining method is applied to infer only relevant triples, some triples might be unnecessary for change detection. Therefore, the authors in [ILK13] proposed a pruning method in addition to the backward-chaining strategy. The proposed pruning method is applied prior to invoking backward inference. This allows for the skipping of some irrelevant triples derived from the set-difference operation ($M - M'$) during change detection. After pruning, the remaining triples are checked to see if they can be inferred in M'. By contrast to the approach presented in [BK03], the proposed change detection technique follows *difference-pruning-inference strategy.*

The irrelevant triples are pruned and removed from $M - M'$ using the following rules:

(1) When $Del(t(SsubClassOfO))$: if $t(SsubClassOfT') \notin M'$ or $t(T'subClassOfO) \notin M'$, then $Del(t(SsubClassOfO))$ is pruned.

(2) When $Del(t(SsubPropertyOfO))$: if $t(SsubPropertyOfT') \notin M'$ or $t(T'subPropertyOfO) \notin M'$, then $Del(t(SsubpropertyOfO))$ is pruned.

(3) When $Del(t(SBO))$: if $t(SAO) \notin M'$ or $t(AsubPropertyOfB) \notin M'$, then $Del(t(SBO))$ is pruned.

(4) When $Del(t(StypeO))$: if $t(StypeT') \notin M'$ or $t(T'subClassOfB) \notin M'$, then $Del(t(StypeO))$ is pruned.

The general rule for pruning is that if the subject or object of a triple does not exist in M' then this triple cannot be inferred in M', therefore, this triple is pruned before the inference process begins.

Results in [ILK13] show that the pruning and backward-chaining method can prune 10-60% of triples prior to the backward-chaining, and that the inference time using that method is 10-80 times faster than that using the forward-chaining method and about 1.5-4 times than backward-chaining method. However, the pruning time and the overall reasoning time (i.e. both the pruning time and the inference) is not provided.

RDF is used to represent information such as bibliographies, medical and biological terms and other large scale data. This information is created and updated from distributed sources, and as already noted, RDF data collections are typically large and awkward to maintain. It is consequently important for RDF versioning and synchronization services to employ efficient strategies to compare two RDF graphs and transform one graph into another. These strategies need to provide sum and difference functions for RDF graphs [BLC04] and a change operations able to exploit the semantics of RDF. It is also important to make the right decisions when optimizing update strategies.

The work presented in this thesis analyses in detail the performance of such strategies for the ontology evolution process as it is important to provide efficient methods for detecting changes in RDF collections and updating them. Since most of the data in RDF versions remains unchanged as reported by [NM04], an alternative approach to forward-chaining inference is the backward-chaining inference [BK03] which instead of computing the full closure of RDF model, computes only the relevant triples. Choosing backward inference over forward inference is a trade-off between storage space and query processing, respectively. A work in [SQ06] proposed a flexible framework that combines the two strategies.

2.5 Reasoning with OWL

The previous section discusses reasoning under a subset of the RDFS semantics. Work presented later shows that exploiting the hidden information in RDF structures by applying inference rules could reduce the size of the delta between two different RDF structures. Further reduction to the delta size is possible if more hidden information is exploited by applying more complex rules. The Web Ontology Language has more facilities for expressing semantics than RDFS, which in the context of our work, can allow change detection techniques to perform useful reasoning tasks that aim to reduce delta sizes.

In this section, we move to more complex inference rules under OWL 2 semantics considering the OWL 2 RL/RDF set of entailment rules [Sch09] which is aimed at applications that require the expressivity of OWL 2 full as well as scalable reasoning[5]. These rules represent a partial axiomatization of the complete OWL 2 RDF-Based semantics which is designed to be useful for implementation using rule-based technologies such as RDBMSs, Prolog, Jess, etc.

The OWL 2 RL/RDF ruleset provides a simple set of rules of RDF-based semantics as a set of first-order logic formulas but the formulas for these rules generally have a more restricted form which make the resulting language less expressive than OWL 2 Full. However, it is allowed for compliant OWL 2 RL/RDF reasoners to produce additional reasoning results up to the full expressivity of the OWL 2 RDF-based semantics. Therefore, it allows tool providers to create

[5] Scalable reasoning is reasoning performed on large scale datasets

reasoners with an OWL 2 RL/RDF ruleset as a base but with additional power if needed by customers.

The official OWL 2 RL/RDF ruleset contains 78 rules compared to the 13 rules defined for RDFS. OWL 2 RL/RDF supports all RDFS constructs; however, rules for axiomatic triples, such as *rdf:type rdf:type rdf:Property*, in RDF and RDFS are omitted in OWL 2 RL/RDF for performance reasons as those triples must be satisfied by, respectively, every RDF and RDFS interpretation [MGH+09]. In addition OWL 2 RL/RDF supports rules for defining the semantics for: equality, property axioms, classes, class axioms, datatypes and schema vocabulary. The semantics of these rules take a variety of forms:

- Semantics of equality which define the equality relation owl:sameAs and its equality properties such as reflexive, symmetric and transitive.
- Semantics of axioms about properties.
- Semantics of classes.
- Semantics of class Axioms.
- Semantics of Datatypes which include rules for a special processing for datatypes; e.g. the equality dt-eq and inequality dt-diff rules for literals that assert that two literals with the same value are equal or two literals with different values are different, respectively.
- Semantics of schema vocabulary which specify the semantic restrictions on the vocabulary used to define the schema.

The simplicity of the rule-based language specification, the unrestricted RDF graphs that these rules can handle and the flexibility to extend OWL 2 RL/RDF reasoners makes this language specification a promising candidate for practical reasoning in the semantic web [SM09]. Moreover, regarding the syntax of OWL 2 RL/RDF ruleset, there are no limitations to which ontologies this ruleset can be applied since they are defined upon arbitrary RDF graphs therefore it can be applied to unrestricted or generalized RDF datasets.

The number of rules in reasoning with OWL 2 RL/RDF and the type of relations supported in these rules have led to several challenges in reasoning with OWL 2 RL:

- The support of equivalent relations in OWL 2 RL/RDF such as owl:sameAs. A sameAs relation is a relation of the form (s, owl:sameAs, o), which asserts that s denotes the same resource as o. This relation may increase the complexity for a full materialization of the inferences as it implies the computation of the transitive closure across the entire KB. This complexity could be $O(N^2)$ with respect to the size of the original KBs using a naïve representation of closure. This complexity becomes very expensive with evolving large scale data sets as the inference closure needs to be maintained continuously.

- OWL 2 RL/RDF ruleset has large number of inference rules compared to the RDF(S) ruleset, and these rules may trigger the application of other rules. A naïve approach to performing reasoning against large-scale datasets may take hours to finish [KWE10].

- Generally, it is not possible to compute all class subsumptions without taking into account assertional information (i.e. data) [Krö12] which in turn may complicate the reasoning process with regard to the size of ABox[6] in the used dataset.

Some rules are excluded in a typical implementation for two reasons:

1. They have no effect on minimizing the delta size.
2. They may yield an exponential number of inference cycles.

These rules are:

- Inconsistency checks rules. These rules check if a contradiction is derived and that consequently,the RDF structure is inconsistent.
- The rules that exploit the owl:sameAs transitivity and symmetry. These rules produce cycles in the inference process as they can be applied to every term of the triples because of the interaction among owl:sameAs inferences with other rules in OWL 2 RL/RDF.
- Data list relationships to be asserted between chains of properties such as the rule prp-spo2 for propertyChainAxiom which says, for example, that uncle is precisely a parent's brother.

[6] An ABox contains data associated with a knowledge base

- Datatype checking using rules for supporting datatype reasoning which provides type checking and value equality and inequality checking for typed literals.
- General schema rules which say, for example, that a class is a subclass of itself or a property is a subproperty of itself.

In this thesis a subset of OWL 2RL/RDF rules (shown in the table in Appendix B) is used. This set of rules differ from the RDFS rules, for example, there are multiple antecedents in some of these rules. Moreover, most rules in OWL 2 RL/RDF include assertional triples and generally the number of assertional triples exceed that of schema triples[7]. The application of these rules involves calls to the application of other rules and so on until saturation (i.e. no more triples are inferred). Another challenge with these rules are the use of blank nodes when defining restrictions on classes such as the use of intersectionOf and unionOf relation.

2.6 Blank nodes

Blank nodes, are a special kind of nodes without a name. They indicate the existence of a thing for which a URI reference or literal value is not given. Since they are anonymous, blank nodes require special treatment when matching ontologies. Despite the problems involved in processing data with these anonymous nodes, the use of blank nodes in RDF data models is an important feature, which adds

[7] Schema triples are triples used in describing an ontology such as triples with predicate rdf:subClassOf. In contrast, assertional triple is any triple that is not a schema triple [WH09]

flexibility when expressing information in RDF model.

The first stage in delta construction is the computation and production of the explicit delta (i.e. the syntactical differences) between the two stored models. After the computation of the syntactical differences, the blank node matching begins, although no order is required for the two processes as they do not overlap. Blank nodes are arranged in chains and the matching of these nodes can make use of both the ID of the node as well as the triple count in its chain. The equivalence of RDF graphs that contain blank nodes is defined as [KC06] :

Definition 2.6.1 (Equivalence of RDF Graphs with blank nodes).
Two generalized RDF graphs G_1 and G_2 are equivalent if there is a bijection f between the sets of triples of the two graphs, such that:
f(uri) $=$ uri for all uri $\in U_1 \subseteq G_1$
f(lit) $=$ lit for each lit $\in L_1 \subseteq G_1$
Where $U1$ is the set of URIs and $L1$ is the set of literals in $G1$. For each b $\in B_1$ f maps blank nodes to blank nodes, such that f (b) $\in B_2$, where b is a blank node and B1 and B2 are the set of blank nodes in G1 and G1, respectively.
The triple (s, p, o) is in G_1 if and only if the triple (f(s), p, f(o)) is in G_2

It follows that if two graphs are equivalent then it certainly holds $U_1 = U_2$, $L_1 = L_2$ and $\|B_1\| = \|B_2\|$. Since uri and lit have global identifiers in both graphs, this bijection matches the uri or lit in one graph to the same uri or lit in the other graph. Unlike uri and lit, blank nodes have only local identifiers, thus, f shows how each blank node identifier in G_1 can be replaced by a new identifier in order to give G_2

Without blank node matching, any pair of blank nodes from different knowledge bases is considered as a difference between these data structures. If $|T_{b1}|$ and $|T_{b2}|$ are the blank node counts in M and M' respectively then without blank node matching the delta for two graphs will contain at least $|T_{b1}| + |T_{b2}|$ change operations.

Matching these blank nodes may reduce the size of the delta. The worst case of blank node matching is when all blank nodes in the participating triples are not matched. In this case, the delta size with blank node matching is equal to the delta size without blank node matching. Thus, if blank node matching does not reduce the delta size, it will never increase it.

2.7 Summary

In order to cope with the evolving nature of the Semantic Web we need an efficient change detection approaches for building effective Semantic Web synchronization and versioning services. In this thesis we are interested in computing RDF deltas as a set of update operations (i.e. inserts and deletes) which enable the correct transformation from one RDF knowledge base into another.

As these update operations are required for synchronization through the network, it is important to reduce the deltas size and avoid unnecessary updates. This reduction of the delta will also reduce the storage overhead required to store different deltas between different versions of the knowledge base.

This thesis is focused on change detection approaches that exploit implicit information in RDF knowledge bases for producing small RDF deltas (i.e. with a small number of insertions and deletions). It also aims at matching blank node chains for further reduction of the delta which is considered as a preprocessing step that is carried out before the application of the change detection approaches.

The next chapter looks at work that has been carried out on approaches to manage ontology changes with the same or similar orientation as the work described in this thesis.

3. RELATED WORK

This section gives a discussion of the principals involved in ontology change and the state of the art tools that have the same or similar goals to the work explained in this thesis. At the end of this Section, the differences between these tools and the work of this thesis is characterized.

3.1 Practicality of ontology evolution

In the context of Semantic Web systems, there are several non-text based tools that have been proposed for detecting changes between RDF graphs. For example:

- OntoView [KFKO02]: An ontology management system that compares two RDF graphs and produces the differences between them.
- PromptDiff [NM02]: An ontology versioning system that compare two different versions of an ontology based on heuristic matchers.
- SemVerion [VG06]: A tool that produce differences between ontologies using two different comparison methods; syntax-based and semantic-based.
- CWM [BL$^+$00]: Is a general purpose tool for comparing RDF data in the Semantic Web. It uses functional and inverse functional properties for iden-

tifying blank nodes.

- x-RDF-3X [NW10]: A system for the management and querying of RDF datasets based on a deferred indexing method with integrated versioning.
- Jena [CDD$^+$04]: Is a Java framework for developing Semantic Web applications. It provides a tool for checking if two RDF files are isomorphic.
- pOWL [Aue04]: A web-based ontology management system. It includes a tool for tracking changes made when editing the ontology using the system.
- Work of Flouris et al. [FPA06]: Introduces a logic-based approach to ontology evolution. This work is inspired by the general belief revision principles for considering ontology changes in instance and schema levels data.
- OUL [LRVS09] [ILK13]: An event-driven automated ontology updating approach that makes use of SPARQL and SPARQL/Update statements.
- Work of Dong-Hyuk et al. [ILK12]: A version framework for the management of RDF versions in relational databases and a pruning method for the computation of RDF deltas.
- RDF/S Diff [ZTC07]: An approach that proposed five differential functions which take into account the implicit knowledge in RDF graphs by means of RDF/S inference rules.
- BNodeDelta [TLZ12]: A proposed method for the blank node matching problem that aims at finding a mapping that yields a minimal sized delta.

- SQOWL 2 [LM15]: is an approach that performs transactional and incremental reasoning over OWL 2 RL ontologies stored in RDBMS for query answering. It also supports update for instances level data.

These existing comparison tools typically do not focus on the size of the delta produced, nor on blank nodes matching. These two aspects are important for an efficient versioning system for Semantic Web repositories where versions are stored in remote sites on the Internet. In addition, these tools are essetially based on high-level changes between RDF graphs. They describe the meaning of the different change operations that are detected by the underlying change detection algorithm in such a way that they can be interpreted and exploited by humans [PFF$^+$13]. For example, changing the name of a class by deleting it and inserting it with the new name can be represented in a high-level language as a rename operation instead of the equivalent (delete and insert pair) low-level change operations. Therefore, these tools give more attention to other aspects of the delta such as presentation of the delta in a way that is interpreted by humans. This includes highlighting the differences with different colors [KFKO02] or representing the differences in human language rather than a language that is interpreted by machines [NM02].
The rest of this Section gives a detailed analysis of these tools, with more attention to ontology languages supported by them and the way they deal with blank nodes when producing deltas.

3.1.1 OntoView

OntoView [KFKO02] is a Web-based system that provides users with tools to aid the task of ontology versioning. It is able to compare two ontologies and recognise their differences. Moreover, it allows users to specify the conceptual relations such as subsumptions and equivalence. It stores the content of a version, metadata, and conceptual relations in the ontology and the differences between versions. OntoView distinguishes two types of changes: changes in the conceptual level and changes in the syntactic level. It provides a diff view tool that compares two versions of an ontology at the structural level. This tool is inspired by the UNIX diff tool, however, the UNIX tool compares two text files at a line-level and produces lines that are textually different, while OntoView compares two versions of an ontology at the structural level highlighting changes in the definitions of ontological concepts and properties. The tool distinguishes between several types of change:

- Non-logical changes: changes in the natural language such as changes in RDF:label or changes in a comment.
- Logical changes: changes in the definition of a class or property such as changes in the subsumption hierarchy or in the domain or range of a property.
- Identifier changes: this is when a change is made to the identifier of a class or a property such as renaming a class or property.

- Inserting definitions: are changes to the ontology made by adding a new class or property.
- Deleting definitions: changes that are made to the ontology by deleting a class or property.

It is worth noticing here that all changes are detected automatically apart from identifier changes as this change is not distinguishable from deletions and insertions. Therefore, the system tends to use the location of the definition in the file to determine whether or not it is a changed identifier.

Regarding the languages that Ontoview uses for the production of deltas, it supports an RDF-based ontology language which includes RDFS and DAML+OIL. First, it splits the ontology to separate definitions. These definitions are then parsed into a group of RDF triples. Each group of RDF triples represents a definition of a concept or a property.

The algorithm then locates each small group of triples in the new version with the corresponding group in the previous version of the ontology. The changes between these groups are then calculated according to a number of rules which specify the required changes in the triple set for a specific type of change.

These rules are based on a specific mechanism that identifies a set of triples that should exist in one specific version, and the set should not exist in another version. These rules could then identify all types of changes apart from the identifier changes as mentioned earlier. This mechanism relies on the materialization of all the RDF:type triples in the ontology.

Blank nodes are determined as identifier changes, and blank node matching is used indirectly by using the location in the file as a heuristic to determine whether it matches or not with a blank node in the other file.

3.1.2 PromptDiff

PromptDiff [NM02] is an ontology versioning tool integrated into Protégé. This tool detects changes between two versions of an ontology. It detects structural changes between two ontologies based on the structure of the ontologies rather than their text serialization.

The algorithm uses the a knowledge model compatible with the Open Knowledge Base Connectivity (OKBC) protocol [CFF$^+$98], which states that an ontology consists of different elements: classes, class hierarchy, instance of classes, slots, slots attached to a class to specify class properties, and facets to specify constraints on slot values. All these elements can be represented by different representation formalisms (i.e. ontology languages) such as RDFS, OWL, DAML+OIL and other formalisms. Thus, PromptDiff applies to ontologies regardless the representation formalism used.

The PromptDiff algorithm is based on the integration of different heuristic matchers for comparing and analysing two versions of an ontology and producing the structural differences between them. This approach consists of two parts: (1) an extensible set of heuristic matchers (2) an alignment algorithm to combine the results of the matchers for producing the structural difference between two versions.

Each matcher employs a specific number of structural properties of the ontologies to produce matches. The alignment algorithm combines these matchers so that the result of one matcher is an input into the next and so until this process produces no more changes in the diff. This diff is then presented to the user to accept the application of these changes or reject them.

PromptDiff combines an arbitrary number of matchers, each matcher looks for a particular situation in the unmatched frames, which can be of any type; class, slot, facet or instance. These situations are: frames of the same type with the same name, single unmatched sibling, siblings with the same suffix or prefix, single unmatched slot, unmatched inverse slot, and split classes.

Moreover, as each of the matchers is a heuristic matcher the results produced by the matcher may not always be correct. Therefore, these results are presented to a human expert to analyse them and examine the results of these matchers to check whether they are correct or not. Although the heuristic matchers generally focus on matching the structure of the nodes in the ontology which can also be applied to blank nodes, no special focus was given to this problem.

3.1.3 Semversion

Semversion [VG06] is a structural and semantic versioning system for RDF models that supports RDF-based ontology languages such as RDFS. This system separates the management aspects from the core versioning functions.

This approach was inspired by CVS [B^+90], the text-based versioning system. The core feature of Semversion is the separation of the language-specific feature (i.e. the diff) from the structural differences. The system offers a simple API. For example, to commit a new version, users can either provide the complete content of the new version of an RDF model or the diff which is a set of changes that needs to be applied to the existing version to update it to the newer one.

This versioning system is based on a layered approach that follows the Semantic Web architecture. These layers are: the RDF syntax layer (e.g. RDF/XML, Turtle, N-triple), the RDF structure layer and the ontology layer. This is to provide a general versioning system independent of the ontology language used. At the same time, the ontology versioning layer of the system can be extended to support functions for specific ontology languages. These functions take the semantics of the language into account in order to produce semantic diffs or merging with semantic conflict detection.

Diff in Semversion is used in two ways; (1) to calculate the semantic and structural differences between two RDF models. (2) it can be used as an update command that can apply changes to a remote RDF model because the transfer of updates is more efficient than the transfer of the complete new RDF version.

Semversion identifies three types of diff methods:

- Set-based Diff, is the process of computing the differences between the explicit triples in the two models using set arithmetic for the triple set. Given we have two versions of an RDF model M and M', the set based diff between these versions are calculated using $(M - M')$ for deleted triples and $(M' - M)$ for inserted triples. The diff result is then the union of the results of these two operations. In this method, only triples containing URI or literal and not blank nodes are considered. Considering blank nodes when calculating the diff using the set-based method will report all statements involving blank nodes in the first model to be deleted and all statements involving blank nodes in the second version to be inserted.

- The second diff method is the structural diff. In this diff, Semversion gives a special focus to blank nodes. Without the presence of blank nodes in the models the structural diff will be the same as the set-based diff. The problem of blank nodes is overcome by a process called blank node enrichment. This process involves adding functional properties to each blank node leading to a URI and thus treating them as normal nodes. Thus, Semversion can decide whether two blank nodes are the same or not. However, blank nodes are matched when they participate in the same triples.

- The third diff method is the semantic diff. In this diff the semantics of the underlying ontology language are considered. Semversion uses forward inference to compute the differences between RDF models. It materializes the full closure for both models (i.e. $C(M)$ and $C(M')$) in advance and then performs the structural diff between $C(M)$ and $C(M')$.

3.1.4 CWM

Closed World Machine (CWM) [BL+00] is a Semantic Web general-purpose data processor for manipulating RDF data in RDF/XML, NTriple and Notation3 format. It is used to store RDF triples in a triple table for querying using a forward-chaining inference engine. CWM also supports the computation of the explicit delta between two knowledge bases and then transforms one knowledge base into the other using the resulting delta.

3.1.5 x-RDF-3X

x-RDF-3X [NW10] is a system designed and implemented mainly for the management and querying of RDF datasets. Each dataset is stored in six duplicates of ternary projections that constitute an RDF triple (i.e. Subject, Predicate, and Object), one duplicate per index (SPO, SOP, OPS, OSP, POS, PSO). In addition to the six indexes, x-RDF-3X adds another set of 9 aggregates indexes that consists of binary projections (SP, SO, OP, OS, PO, PS) and three unary projections (S,P,O). These indexes, x-RDF-3X provides a performance improvement as they eliminate the problem of expensive self-joins in database tables which are required for answering SPARQL query patterns which are executed against RDF databases. However, storing and accessing these indexes with increasing volumes of datasets may come at a high cost due to the duplicates of these indexes. Not only storage, but also performing updates on these datasets is a challenge as it also requires updating all the indexes.

Versioning is achieved by maintaining versions of individual triples using two timestamp fields; created and deleted, to denote the life of each triple version. This timestamp interval contains the life span for a triple version. For triples that have not changed, the deleted field has a null value.

The update system is built upon a number of assumptions. First, it assumes that the number of updates to RDF models are infrequent compared to the number of queries over these models. The second assumption is that the insertions are more frequent than deletions in RDF models.

Based on these assumptions, the authors have designed and implemented a differential indexing method for RDF-3X. This method has considered updates as either inserts or deletes. For inserts, it is more challenging as the system is based on indexing and each RDF dataset is stored in 15 different tables, inserting a triple requires the inserts to these different tables. To overcome this problem, the system uses a staging architecture through the use of deferred index updates. First all updates are collected in workspaces and all their triples are indexed in differential indexes in an isolated manner from the main database. These changes remain separate for a specific period or until the changes exceeded a specific time limit, then all the changes are integrated into the main indexes in the database.

The system handles deletions in the same way as insertions. So in the case of deleting triples, these triples are inserted into a separate workspace and differential indexes with a deletion flags are generated. However, this system applies the updates unaware of reasoning.

3.1.6 Jena

In Jena the handling of RDF data is supported by a Semantic Web toolkit for Java programmers [CDD$^+$04]. It is mainly centered on RDF graphs as the universal data structure and as the heart of the different ontology representation languages such as RDF, RDFS and OWL. The single unit of a graph is the binary relationship(S,P,O)where P is the binary relationship that links (S,O). This graph is the core interface for Jena around which the other components are built.

Jena provides a rich API for manipulating RDF graphs. This API provides different tools to process RDF graphs. For example, it provides tools for input and output of various serialization of RDF data such as RDF/XML. N-Triple, and N3. It also supports the query language RDQL [MSR02]. Moreover, it gives the users the option to store their RDF data either in memory or in database-backed stores. It also supports reasoning in OWL FULL, DAML+OIL, and RDFS.

The design of Jena is built upon two key architectures goals:

- The Presentation layer: RDF graphs can be presented using a multiple and flexible presentations for programmers.
- The graph layer: the goal of this layer is to offer a simple view to the programmer in terms of RDF triples, as this is particularly useful for reasoning. The first layer is layered on top of the second.

In terms of calculating the differences between two models, Jena offers a method that can compare two RDF files and reports whether they are isomorphic or not. However, this method does not actually tell what these differences are or what updates can be done in order to update a model from one version into another.

RDF model theory has a vision that the use of blank nodes is within the specific scope in the file and not an arbitrary use. This idea is the basis of handling blank nodes in the work reported in this thesis. This model theory specifies blank nodes as existential quantifiers over the set of resources in which the identifiers of these blank nodes are not significant. This contrasts with other resources, which have identifiers with global scope outside the local file (i.e. URI identifiers). Blank nodes in RDF can play three roles: in the subject, in the object, or both.

The handling of blank nodes follows the approach introduced in [M+81] to check if two graphs are isomorphic. For this method, it creates a signature for all nodes in the graph either anonymous or named nodes based on their position in the graph. Nodes with the same signature are matched. If all the nodes in the graph are matched then the two graphs are isomorphic structures.

A blank node in this method is treated in a more general way and it does not provide any information on the differences between these files that include blank nodes in case of non-isomorphic.

3.1.7 pOWL

pOWL is a Web based ontology management tool [Aue04] implemented using the Web scripting language PHP. It provides different tools for managing ontologies, which include parsing, storing, querying editing and versioning and comparing RDF and OWL knowledge bases. The design of pOWL is built upon a four-layered architecture:

- RDF store layer which stores RDF data in an SQL based databases.
- The ontology language API layer which consists of RDFAPI, RDFSAPI and OWLAPI.
- The third layer on top of the ontology layer contains all the classes and functions required to build Web application using the underlying ontology layer.
- The fourth layer is the user interface layer. This user interface contains a set of PHP pages for browsing, storing and editing RDF data.

The support for versioning in pOWL provides a method for comparing RDF versions and reports to the user the differences between them. However, this support is limited to ontologies edited by pOWL only as these changes are tracked while editing the ontology. Moreover, pOWL supports the rollback of editing actions. The parent action is rolled back only if the sub-actions are rolled back.

3.1.8 Work of Flouris et al.

Flouris et al. [FPA06] studied the problem of ontology evolution considering changes in instance level and in schema level with respect to the integrity constraint associated with the ontology. Their work is inspired by the general belief revision principles[Gär03] of Success[1], Validity[2] and Minimal change[3]. Their proposed framework is responsible for addressing invalidities caused by change request. However, this is mainly a theoretical work.

3.1.9 OUL

OUL (Ontology Update Language) [LRVS09] detects frequent domain changes and updates the ontology. It is built on top of SPARQL and SPARQL Update and is inspired by database triggers. However, the language requires a user to manually execute an update in order to detect and handle changes. An extension to OUL is done in [SHF12] to address some of its drawbacks such as applying immediate updates rather than the deferred updating. In addition, the extension also includes automatic change detection which make use of change handlers and performs SPARQL and SPARQL/UPDATE actions whenever a certain change event occur. This process is inspired by active database triggers.

[1] Every change operation is implemented
[2] The resulting ontology is valid
[3] The appropriate result of changing an ontology should be as close as possible to the original

3.1.10 Work of Dong-Hyuk et al.

Dong-hyuk et al. [ILK12] proposed a version framework for managing RDF versions in relational databases. They introduced an optimized delta-based scheme for managing RDF versions called Aggregated Delta. This scheme stores the most recent version of RDF while older versions are stored as a backward delta between two specific versions which have more than one in-between delta and storing them in advance. Older versions are then created on the fly by directly executing these deltas, instead of executing all the in-between deltas in sequence. Thus, reducing the increasing version reconstruction time and the storage space required to store all versions. In addition, for better performance and storage space, the aggregated approach eliminates the duplicated deltas using the compression algorithm that removes the duplicate triples from the deltas. In addition to saving storage space, this compression algorithm reduces version construction time by avoiding unnecessary computation at version reconstruction.

The performance of this approach was evaluated in terms of storage overhead, logical version construction time, delta computation time, compression ratio, and query processing time. For the experiment, nine versions of a dataset from Uniprot Taxonomy RDF were used.

Results from their experiments show that the proposed framework maintains multiple versions of RDF efficiently compared to two other popular approaches in RDF version management systems, the All Snapshots approach and the Sequential approach. The benefit comes in terms of storage space, version construction time and query construction time.

The proposed scheme presents the basis for storing RDF versions in relational databases. It is designed to handle RDF in both instance and schema levels. However, it is does not handle the OWL model and its associated inference rules.

In [ILK13] a scalable change detection tool for RDF data is proposed. The tool computes the delta (i.e. the difference between two RDF files) based on backward-chaining inference strategy. In addition, in order to improve the performance of change detection, their tool is supported by a pruning method that minimizes the delta size. To handle RDF data sets too large to fit in the available RAM, this tool was implemented using relational database and SQL queries for computing the delta.

3.1.11 RDF/S Diff

In [ZTC07] three RDF/S differential delta functions are proposed to transform one knowledge base into another. These deltas are interpreted as sets of change operations (deletions and insertions) and are calculated based on the inferred knowledge from RDFS knowledge bases. These deltas are explicit dense delta (ΔED), dense delta (ΔD) and dense & closure delta (ΔDC). These deltas vary in the application of inference to reduce their size. ΔED and ΔD are explained in Section 2.5 (Definition 2.5.3 and 2.5.4, respectively).

3.1.12 BNodeDelta

In [TLZ12] the authors proposed methods to exploit blank nodes in order to identify subgraph-isomorphism and reduce the delta size when comparing two knowledge bases using the explicit information only. The proposed methods are based on mapping between the blank node of the compared knowledge bases and they reduce delta size.

The authors represent algorithms that are polynomial in case there are not directly connected blank nodes. These algorithms return approximate solutions in the general case. The first algorithm is based on the Hungarian method [Mun57] which produces smaller deltas but at additional cost. A Hungarian method is an algorithm which finds an optimal assignment for a given cost matrix. Suppose we have n resources to which we want to assign to n tasks on a one-to-one basis. Also suppose that the cost of assigning a resource to a task is known. The assignment problem is to find an optimal assignment that minimizes total cost. In the context of mapping between blank nodes of the compared graphs, the blank nodes in one graph, B1, play the role of resources, blank nodes in the other graph, B2, play the role of tasks, and the edit distances of the pairs in $B1 \times B2$ plays the role of the cost. Considering that $\|B1\| = \|B2\|$, the Hungarian algorithm computes the edit distance between all possible n^2 pairs and can find the optimal assignment at the cost of $O(n^3)$ time. The second algorithm for mapping between blank nodes of two knowledge bases is signature-based which has reduced cost but produces larger deltas.

3.1.13 SQOWL 2

SQOWL 2 [LM15] is an approach that is built as a separate layer over an RDBMS. It performs transactional and incremental reasoning over OWL 2 RL ontologies stored in RDBMS and adopts a materialised approach that preserves full ACID properties (i.e. Atomicity, Consistency, Isolation and Durability [HR83]). In transactional reasoning new data becomes available at the commit of any transaction that inserts or deletes data in the database. In incremental reasoning the materialised views are incrementally maintained as a result of transactions. The incremental reasoning could be recursive when considering OWL 2 RL.

This approach is based on the Delete & Rederive (DRed) algorithm [GMS93]. In this algorithm, all derived facts are over deleted from the view when deleting explicit facts. The approach then rederives new facts that are referable from the remaining explicit facts in the database. This algorithm is inefficient as some of the inferred facts are then reinserted or when some derived facts have many different derivations and have relations to other inferred facts. The SQOWL 2 approach presents a variant of DRed that uses RDBMS triggers which support transactional and incremental type inference. The way the triggers are used is as follows:

- Each derived fact in the materialised data is assigned a state.
- Deletions over the explicit facts invoke RDBMS triggers to update the state of related implicit facts.

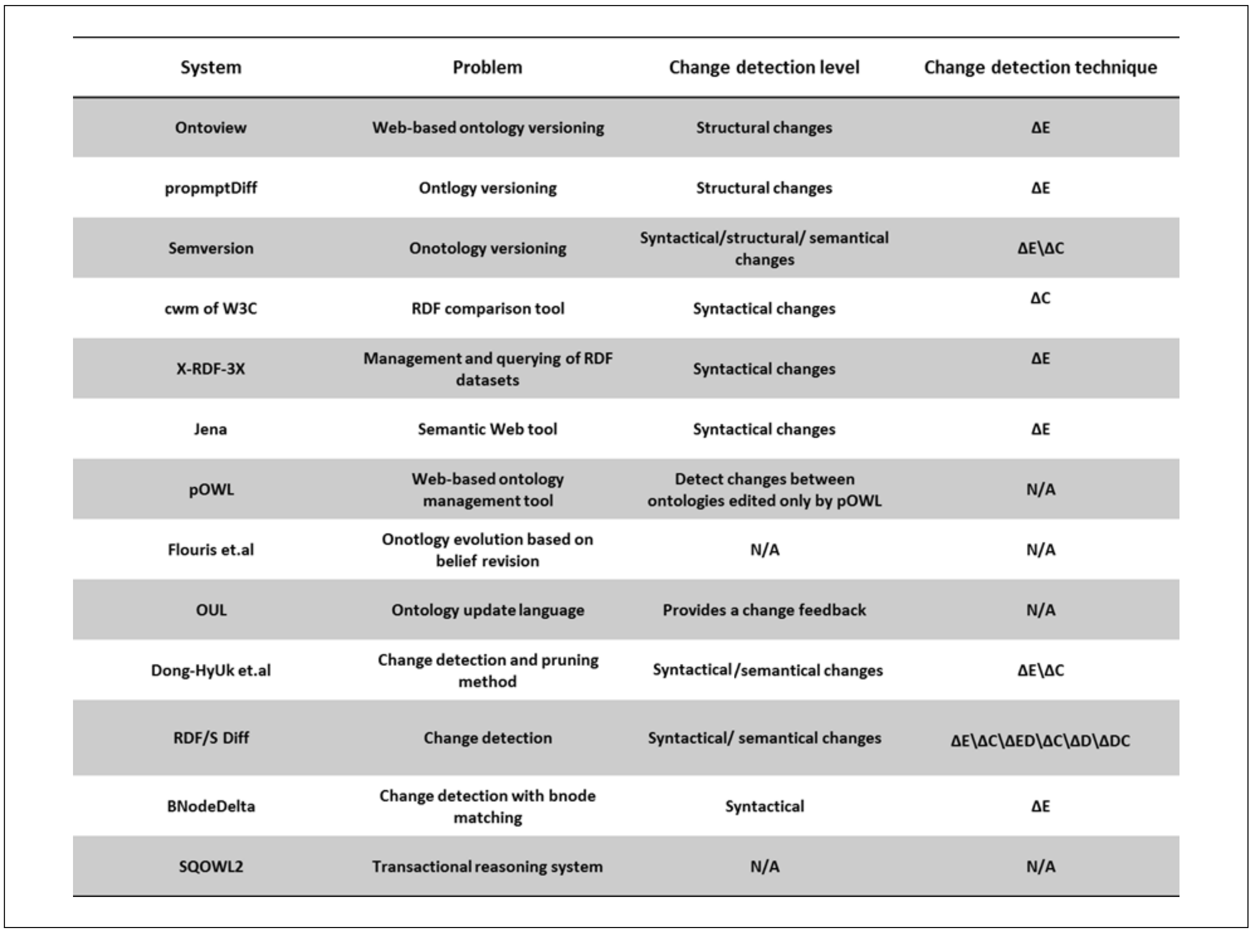
- This reduces the number of real deletes from the implicit facts.

Since this approach materialises the results of reasoning, it is more efficient for query answering than the non-materialising approaches. This approach can also be incorporated into any RDBMS application. This approach is restricted to updates to the ontology A-Box (i.e. assertional knowledge within the knowledge base) and assumes that the T-Box (i.e. terminological knowledge that describes the conceptualization) of the ontology remains unchanged. Thus. it performs type reasoning which derives for each instance its membership of classes and properties. It is also built on the assumption that the number of queries exceeds the number of updates and is less suitable for applications where the number of updates is relatively higher than the number of queries.

3.2 Conclusion

The Semantic Web has undergone significant advancement in change detection. A synopsis of change detection tools is presented in Table 3.1. Work has been carried out on characterising the deltas between ontology versions [ZTC11] but existing RDF change detection tools have not yet focused on producing a small correct delta using the power of inference and blank node matching that could efficiently transform one version of an ontology into another. In addition, although some studies have dealt with inferencing under OWL ontologies in RDF query processing systems [UVHSB11] [UKM$^+$12][KWE10], no work is found for detecting RDF changes under OWL 2 ontologies, that gives attention to not only producing the correct and small delta under the semantics of OWL 2 ontologies but also reducing the unnecessary inferencing and application of the OWL 2 ruleset.

The work of this thesis focuses on change detection between two versions of an ontology, to detect changes between them and also to characterise update commands that can be sent through the network to update remotely stored versions of the ontology. This work gives considerable attention to both the size and correctness of the produced delta. The size of the delta in our work is further reduced by applying a general blank node matching algorithm that preprocesses blank node chains before the calculation of deltas. This algorithm matches blank node chains between the two versions and only adds different chains of blank nodes to the produced delta where these are necessary. For the production of deltas, this work eliminates unnecessary reasoning through the introduction of a rule pruning method in the context of rule-based reasoning with OWL 2 RL/RDF ruleset.



System	Problem	Change detection level	Change detection technique
Ontoview	Web-based ontology versioning	Structural changes	ΔE
propmptDiff	Ontlogy versioning	Structural changes	ΔE
Semversion	Onotology versioning	Syntactical/structural/ semantical changes	ΔE\ΔC
cwm of W3C	RDF comparison tool	Syntactical changes	ΔC
X-RDF-3X	Management and querying of RDF datasets	Syntactical changes	ΔE
Jena	Semantic Web tool	Syntactical changes	ΔE
pOWL	Web-based ontology management tool	Detect changes between ontologies edited only by pOWL	N/A
Flouris et.al	Onotlogy evolution based on belief revision	N/A	N/A
OUL	Ontology update language	Provides a change feedback	N/A
Dong-HyUk et.al	Change detection and pruning method	Syntactical /semantical changes	ΔE\ΔC
RDF/S Diff	Change detection	Syntactical/ semantical changes	ΔE\ΔC\ΔED\ΔC\ΔD\ΔDC
BNodeDelta	Change detection with bnode matching	Syntactical	ΔE
SQOWL2	Transactional reasoning system	N/A	N/A

Fig. 3.1: Existing systems

4. EXPERIMENTAL METHODS

RDF is widely used in the Semantic Web for representing ontology data. Many real world RDF collections are large and contain complex graph relationships that represent knowledge in a particular domain. Such large RDF collections evolve in consequence of their representation of the changing world. Although this data may be distributed over the Internet, it needs to be managed and updated in the face of such evolutionary changes. In view of the size of typical collections, it is important to derive efficient ways of propagating updates to distributed data stores.

Minimising the size of updates can be achieved by reasoning over the underlying knowledge base. This could be accomplished by reasoning over both insertions and deletions and by exploiting the semantic content of the underlying ontology language.

Optimizing the reduction of RDF updates and the need to measure the cost of this process has led to four sets of experiments:

1. An initial experiment that highlights the problem of view updates and the cost of applying inserts and deletes over RDF collections.

2. To cope with the evolving nature of the Semantic Web, it is important to

understand the costs and benefits of the different change detection techniques. This experiment provides a detailed analysis of the overall process of RDF change detection techniques and the cost of reasoning and pruning unnecessary updates.

3. A change detection technique that yields a small delta is proposed. However, this delta must satisfy correctness when propagating these updates to remote knowledge bases. This experimental work describes a new approach to maintaining the consistency of RDF by using knowledge embedded in the structure to generate efficient update transactions.

4. The final set of experiments evaluates the potential for reducing the delta size by pruning the application of unnecessary rules from the reasoning process. It also assesses the impact of handling blank nodes during the change detection process in ontology structures.

4.1 RDF views update

An RDF view is an abstract of underlying data where that abstract is represented as an RDF graph. Three possible ways of applying updates to RDF views and their underlying triple stores are described in Figure 4.1. (a) View may be regenerated after applying updates to the underlying triple store, (b) after updating the triple store, rather than regenerating the whole view, updates are applied to the view to fix it. In this case it is called view maintenance. (c) Updates are applied directly to the view and updates to the underlying triple store are postponed, this case is called view update.

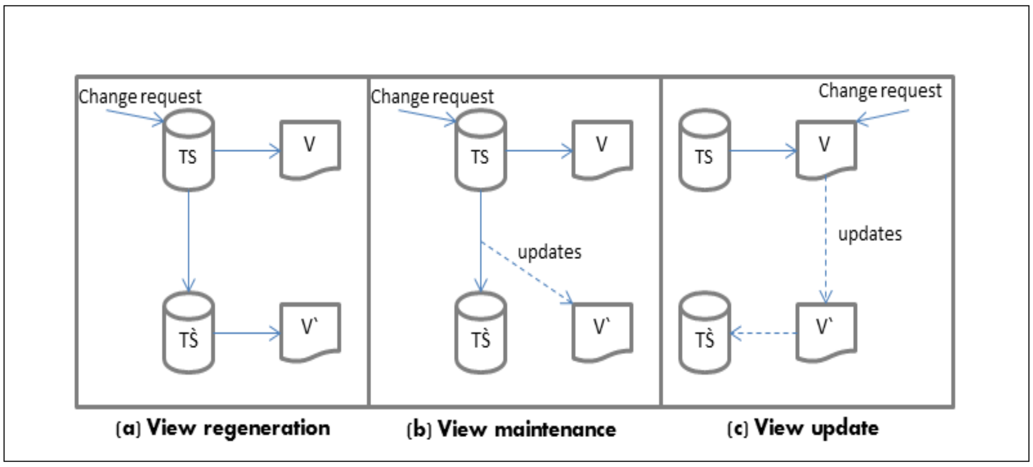


Fig. 4.1: Possible alternatives for applying changes to RDF Data. TS: Triple Store. V: RDF view. The sold arrow between the triple store and the view indicates a regeneration process, while the dotted line indicates updating process.

The aim of the first set of experiments is to assess the performance of basic view operations on RDF data structures using the view update model ((c) above). In these experiments, six different but incremented portions of the Uniprot RDF dataset were stored in RDF/XML format. Table 4.1 summarizes the characteristics of the data set.

RDF data	Model1	Model2	Model3	Model4	Model5	Model6
Number of triples	80,472	162,587	253,081	540,918	755,579	1,287,856
Size (MB)	8.12	16.2	32.5	56.8	73.1	130

Tab. 4.1: Uniprot RDF

The architecture of this work is shown in Figure 4.2. By using libraries in the Jena Framework, RDF structures stored in XML/RDF format are loaded as in-memory models of the RDF structures. These in-memory models can be seen as

Update	Random selection	Specific seletion	Model
Fixed insert	10,000 triples	10,000 triples	All six models
Fixed delete			
Variable insert	10- 100- 1000- 100,000- 1000,000 triples	10- 100- 100- 1000- 100,000- 1000,000 triples	Only model 6
Variable delete			

Tab. 4.2: Number and type of updates performed

RDF views of the underlying RDF structures. The initial work of the experiment involves two types of RDF view updates: multiple random updates and multiple specific updates. These updates are applied to the instance level of RDF and do not handle the schema level, which is the ontology level that identifies complex constraints on RDF data. The view maintenance approach is classified as an updating method. It is not a method for managing multiple versions.

Both random and specific updates perform updates to an in-memory view of the underlying RDF structures. The implementation of both approaches uses Jena as the underlying framework for RDF update as it provides an implementation of SPARQL and SPARUL. These updates were applied against the RDF dataset shown in Table 4.1. Each type of update involves four operations: fixed[1] deletes, fixed inserts, variable[2] deletes, and variable inserts. Table 4.2 shows the details of each of these operations in terms of the exact number of updated triples and on which models these updates are applied. The performance of these updates was compared and evaluated in terms of update CPU time.

In the Random approach, updates were applied to the RDF structure without

[1] Fixed in the sense that the number of updates was fixed but the size of the triple store varied

[2] Variable in the sense that the number of updates varied but the size of the triple store was a constant

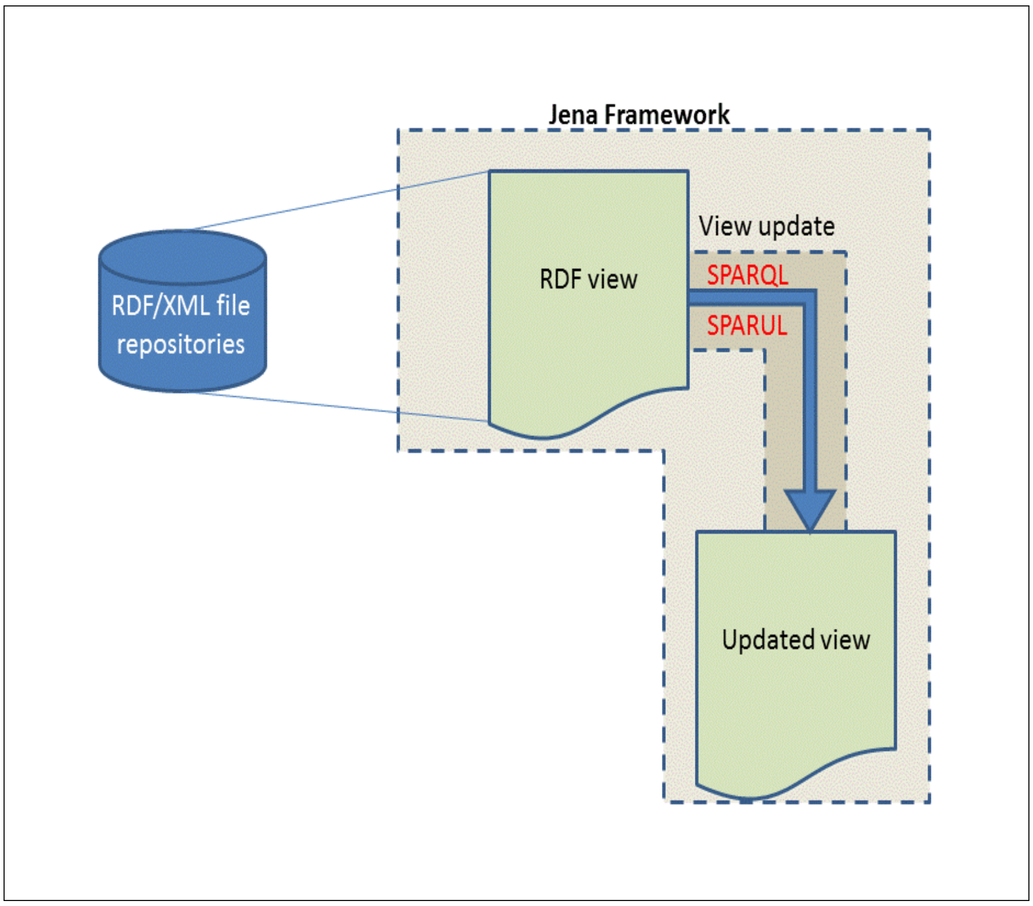


Fig. 4.2: RDF view update.

initially locating specific triples to update. In contrast, in the specific update approach, a list of triple subjects was separately prepared by selecting distinct triple subjects and storing them in a separate text file using Jena framework and Java. The subject entries in this list were used to identify and select the triples that have the same subject or to add new triples having these entries as a subject to update using SPARQL queries. Whilst the random update will give the time needed to carry out the update operation, the difference between the random and specific operations will give the additional time needed to locate each of the triples to be updated.

Within these two overall approaches, the impact of varying the size of the model was evaluated by using data structures ranging from 8MB to 130MB as shown in the table. In these cases 10,000 updates were applied to each of the models. In addition, the impact of varying the number of updates was assessed by applying between 10 and 1,000,000 updates. In both variable updates and variable model sizes, the performance of both insert and delete operations were measured in terms of update time. Results for these experiments are presented in Section 5.1.

4.2 The cost of pruning in change detection techniques

The work in Section 4.1 assesses the overall cost of updating RDF structures. The work described in this section provides a detailed analysis of different change detection techniques[3] including: explicit change detection (denoted as EC), forward-chaining change detection (denoted as FC), backward-chaining change detection (denoted as BC) and pruning-and-backward-chaining change detection (denoted as PBC). EC performs ΔE (Definition 2.5.2) for computing the deltas, while FC, BC and PBC perform ΔED (Definition 2.5.3).

The RDF triple store was designed based on the properties subClassOf, subPropertyOf, Type, and Triple which holds triples that contain any other property type. In addition, there are two extra tables which store the result of performing the set-difference operations $M - M'$ and $M' - M$, these tables are Del table and

[3] Defined in Section 2.4 on page 36

Ins table respectively, and a third table (Inf) stores inferable triples. Using these tables, changes between two RDF models are detected and updated as follows:

First, the differences between M and M' are computed using the set-difference operations $M - M'$ and $M' - M$, the result is stored in the Del table and the Ins table, respectively.

Next, in the case of ΔE, the triples in the Del table and the Ins table represent the delta. Therefore, all the triples in the Del table are removed from M, and all the triples in the Ins table are inserted into M. This step transforms M to M'. In the case of the reasoning-based approach, ΔED, after computing the explicit differences between the two versions M and M' one of the inference strategies FC, BC or PBC is used to produce the delta based on the semantic differences between the versions. In the case of ΔED, the reasoning process is applied only on the Del table and not the Ins table as after the application of this approach the differences in the Ins table will remain explicit differences and not semantic differences.

The computation of delta in FC follows the *inference-then-difference strategy*. Therefore, before computing the differences and updating M, there is an inference process that involves the calculation of the full closure in M'.

In the case of BC, a *difference-then-inference strategy* is followed so the differences between the two models are computed first and instead of calculating the full closure of M' the inference process checks only the triples in the Del table to see if they can be inferred in M' by applying the inference rules in Table 2.2. If

Algorithm 1: Generation of the correct dense delta ΔD_c

```
   Data: $M$,$M'$
   Result: $\Delta D_c$
1  Del = $M - M'$;
2  Ins = $M' - M$ ;
3  for $a \in Del$ do
4      if inferable(a, M') then
5          remove a from Del;
6  for $b \in Ins$ do
7      if (inferable(b, M)) and (all antecedents of $b \notin$ Del) then
8          remove b from Ins;
9  $\Delta D_c$ = Del $\cup$ Ins;
```

ΔD = {Del (John type Student)}
$\cup$ {Ins (Head_Teacher subClassOf Teacher), Ins (Graduate subClassOf Student) }

ΔD_c = { Del (John type Student)}
$\cup$ {Ins (Head_Teacher subClassOf Teacher), Ins (Graduate subClassOf Student), Ins (John type Person)}

Fig. 4.3: The dense delta ΔD $(left)$.The correct dense delta ΔD_c $(right)$ calculated from the example in Figure 2.2

any triple is inferred in M' this triple is removed from the Del table.

In contrast, in PBC, prior to the inference process, some of the triples in the Del table may be pruned. The pruning process checks every triple in the Del table. If both the subject and the object of the triple exist in M' as a subject and object, respectively, then the triple may be inferable in M', and therefore this triple is inserted to the Inf table. After the pruning process, only the triples in the Inf table are included in the inference process and not all the triples in the Del table. It is worth mentioning that not all the rules are applied to each triple, but only the rules that correspond to the property of the triple. The result of this experiments are represented in Section 5.2.

4.3 Correct dense delta

An important requirement of change detection tools is their ability to produce a small but correct delta that will efficiently transform one RDF model to another. This thesis contributes a correct dense delta called ΔD_c. This is achieved by producing a solution to the correctness of ΔD (ΔD is characterized in Definition 2.4.4). This is a particularly important problem when RDF collections are large and dynamic. In this context, propagation between server and client or between nodes in a peer-to-peer system becomes challenging as a consequence of the potentially excessive use of network bandwidth. In a scenario where RDF update is carried out by push-based processes, the update itself needs to be minimised to restrict network bandwidth costs. In addition, in pull-based scenarios, it is important to limit server processing so that updates can be generated with maximum efficiency. Both of these scenario will benefit from using the smallest deltas that will maintain the consistency of an RDF knowledge base.

Definition 4.3.1 (Correct dense delta)**.** Let ΔE, $C(M)$ and $C(M')$ be as defined in Section 2.5 and additionally let $ante(s,t) \rightarrow s$ indicate that s is an antecedent of t. The correct dense delta ΔD_c is defined as:

$$\Delta D_c = \Delta E - (\{Del(t) \mid t \in C(M')\} \cup \{Ins(t) \mid t \in C(M) \wedge \{ante(s,t) \rightarrow s \notin Del\}\})$$

Where Del is the delete set.

Under the semantics of the subset of RDFS rules in Table 2.2 all deltas are unique [4] with respect to the difference between $C(M)$ and $C(M')$. ΔD_c does not require M or M' to be closed and consequently it is not unique.

The correct dense delta is produced by checking triples in both the insert and delete sets of ΔE. Firstly, the delete set should be calculated before the insert set. Secondly, all antecedents for each inferred triple must be checked to see whether they exist in the delete set. If one or both antecedents exist in the delete set then this triple cannot be inferred.

To calculate the closure for M in order to compute the insert set, if two triples in M point to a conclusion based on the rules, then these triples are checked against the deleted set. The conclusion cannot be true if at least one of the two triples exists in the delete set, otherwise, the conclusion is true and the triple can be inferred in M. This process (Algorithm 1) produces the correct dense delta ΔD_c.

Returning to the example in Figure 2.2 in Chapter 2, Figure 4.3 provides an example of the operation of ΔD_c. Because the delete set is calculated first, the triple *(John Type Person)* will not be inferred from *(John Type Student)* and *(Student SubclassOf Person)* given that the former is included in the delete set. The delta will result in the updates shown in Figure 4.3 (the updates on the left). Applying these updates to M will result in the model in Figure 4.4. This model is identical to M', indicating the correctness of ΔD_c. The number of updates after fixing the incorrectness problem is increased but it produces a correct delta. However, this number is smaller than the number of updates produced by ΔED

[4] Uniqueness means there is only one set of update that transform one model into the other

	M
Original triples	(Graduate subClassOf Person), (Student subClassOf Person), (Head_Teacher subClassOf Staff), (Teacher subClassOf Staff), (Staff subClassOf Person), ~~(John Type Student).~~
Inserted triples	(Head_Teacher subClassOf Teacher), (Graduate subClassOf Person). (John Type Person)

Fig. 4.4: Correct updates

or equal to it in the worst case. In such a worst case, none of the inserted triples in ΔD_c can be inferred in M because either there are no triples that can be inferred or at least one of the antecedents of every inferable triple is included in the delete set.

Both ΔED and ΔD_c functions discussed above apply an *inference-then-difference* strategy. This implies that the full closure of the RDF models should be calculated and all the possible conclusions under the RDFS entailment rules are stored in these models. By contrast, a backward-chaining approach uses the *difference-then-inference* strategy. That is, instead of computing the entire closure of M', in the case of ΔED, this method calculates first the set-differences $M - M'$ and $M' - M$, and then checks every triple in $M - M'$ and removes it if it can be inferred in M'. The operation becomes:

$$\text{Remove } t \text{ from } (M - M') \text{ if } t \in C(M')$$

Instead of pre-computing the full closure in advance, this method infers only triples related to the result of $M - M'$. This would be expected to improve the time and space required in change detection by comparison with the forward-chaining approach.

In the example dataset shown in Figure 2.2, to calculate ΔED using the backward-chaining strategy, the sets of inserted and deleted triples are calculated using set-difference operation in the same way as when calculating ΔE. After calculating the changes at the syntactic level, each triple in the delete set is checked to see if it can be inferred in M' using the RDFS entailment rules in Table 2.2. For example, the triple *(Graduate subClassOf Person)* in $M - M'$ is checked to see if it can be derived in M'. Using the RDFS entailment rules this triple can be derived from the two triples *(Graduate subClassOf Student)* and *(Student subClassOf Person)*, therefore, this triple is removed from $M - M'$. Rather than checking all the triples in M', only the three triples in $M - M'$ are checked.

For applying the backward-chaining in ΔD_c, first the set of deleted triples in $M - M'$ is inferred as explained above, then the set of inserted triples in $M' - M$ is also checked to see if it can be derived in M. However, to guarantee the correctness of the delta, before removing the inferable triples from the delta, antecedents of each inferable triple in $M' - M$ are checked to see if at least one of them exists in $M - M'$. If this is the case, this triple cannot be removed from the delta. Algorithm 1 describes the generation of ΔD_c by backward-chaining.

Both forward-chaining and backward-chaining produce the same delta, but the latter applies the inference rules on only the necessary triples. However, although the backward-chaining method is applied to infer only relevant triples, applying the inference on some of these triples might be unnecessary allowing pruning to be applied before backward-chaining [ILK13]. The general rule for pruning is that if the subject or object of a triple in $M - M'$ or $M' - M$ does not exist in M' or M, respectively, then this triple cannot be inferred, consequently the triple can be pruned before the inference process begins. Although pruning may reduce the workload for inferencing, it carries a potential performance penalty [AAW15]. A proof of the correctness of ΔD_c is presented in Appendix C. Results for generating deltas using ΔD_c are presented in Section 5.3.

4.4 Delta generation using pruned ruleset

The use of pruning triples in the context of RDFS knowledge bases typically follows the process of checking the subject and object of each triple to see if it exists in the knowledge base. If it does exist then it is needed for the inferencing process. If not, the triple can be pruned from the inferencing set. This works well when there is a large number of triples and few rules. Where the rules are more complex, as in the case of OWL2 RL/RDF, pruning the ruleset rather than the triples becomes more important. This section describes the process of pruning the OWL 2 ruleset in the context of repeated rounds of rule application. This contrasts with previous approaches to the problem that focused on pruning the triples themselves.

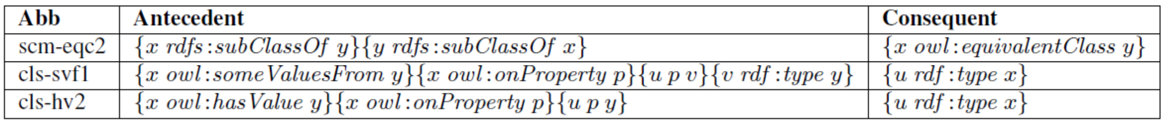


Abb	Antecedent	Consequent
scm-eqc2	$\{x\ rdfs{:}subClassOf\ y\}\{y\ rdfs{:}subClassOf\ x\}$	$\{x\ owl{:}equivalentClass\ y\}$
cls-svf1	$\{x\ owl{:}someValuesFrom\ y\}\{x\ owl{:}onProperty\ p\}\{u\ p\ v\}\{v\ rdf{:}type\ y\}$	$\{u\ rdf{:}type\ x\}$
cls-hv2	$\{x\ owl{:}hasValue\ y\}\{x\ owl{:}onProperty\ p\}\{u\ p\ y\}$	$\{u\ rdf{:}type\ x\}$

Fig. 4.5: Entailment rules in OWL2 RF/RDF

4.4.1 OWL 2 RL/RDF rules

The RDFS ruleset provides limited scope for entailment and most of the rules are not relevant to inference over RDF updates. The OWL 2 RL/RDF ruleset is more extensive and provides considerable scope for reasoning over ontology updates[MGH+09]. Examples of the rules are shown in Table 4.5. The full ruleset used in this thesis is shown in Appendix B.

OWL 2 rules form an OR tree and as can be seen from Table 4.5, there are multiple possibilities for establishing a single consequence such as (x *rdf:type* y). Furthermore, the structure of these rules allows for iterative inference over a triple set. That is, each rule may produce new triples that can be added to the triple set and impact further rounds of rule application.

The application of these rules can be used to find the smallest delta that can unambiguously represent the difference between two ontologies. A significant challenge in reasoning over such differences comes from the presence of blank nodes in ontologies.

4.4.1.1 Rule execution

Simple database implementations of OWL 2 RL rules perform poorly in ontologies with large ABoxes as the application of the rules requires execution of joins on the arbitrary large ABox [HD09]. However, optimization such as the parallelisation of backward-chaining can improve the performance of rule implementations. This work focuses on backward-chaining for the reduction of RDF deltas.

Definition 4.4.1 (Delta reduction using chaining).
Given two RDF models M, M' and a set of entailment rules R, the reduced delta Δ_R is defined as: a reduced set of triples $t_I \mid t_I \notin \Delta_R$ *are entailed in* M, M' using the rules in R.

Regardless of the set of considered rules, for each update (i.e. inserted or deleted triple) in the delta, backward-chaining first searches all the rules for a conclusion that is compatible with this update. After this, it will look at the body of these rules trying to find antecedent patterns that contain values in the same position as specified in the body of the rule. Only triples that contain properties of the type:
rdfs:subClassOf, *rdfs:subPropertyOf* or *rdf:type* are inferred and are checked in this way.

A subset of the OWL 2 RL/RDF rules can be categorised into three groups based on these properties. Each group contains a set of rules that have a property of these values as a conclusion and a body consisting of one or more antecedent patterns that lead to that conclusion. Figure 4.6 shows the resulting OR tree.

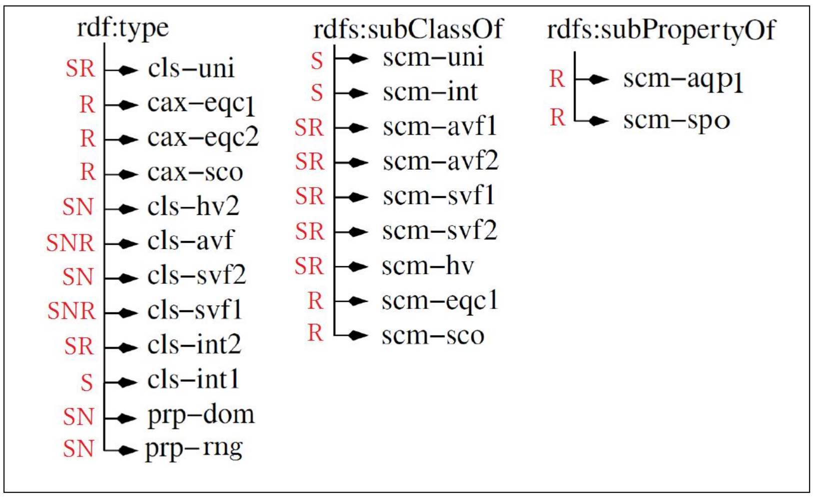


Fig. 4.6: OWL 2 OR trees used to derive conclusions about rdfs:subPropertyOf, rdfs:subClassOf and rdf:type. (In red) The type of each rule based on the type of its patterns: Selective(S), Non-selective(N), Recursive(R) or a combination of two or all of them.

To check if an update of a particular property type is inferable in the knowledge base, the set of rules in the appropriate tree are applied sequentially until the update is inferred in the knowledge base or no more rules remain to apply.

Implementation of these rules can be simplified by decomposing the antecedents into multiple database searches which are terminated when one component fails to return a value. Further simplification can be achieved by executing rule patterns in a particular order starting with simpler patterns first.

Rule patterns in OWL 2 RL/RDF are either selective, non-selective or recursive. The different types of patterns indicate how these patterns are executed against the dataset in order to find a matching triple. A selective pattern does not require further execution of the set of rules in order to find a matching triple in the dataset. If no triple in the knowledge base matches the selective pattern then no further rules can be applied to infer that pattern. This contrasts with the recursive pattern which will generate repeated calls until the desired conclusion is found or until no more patterns can be executed. Non-selective pattern may trigger the execution of further rules but are not in themselves recursive.

In OWL 2 RL/RDF ruleset (Appendix B), rules can consists of selective patterns, non-selective patterns, recursive patterns or a combination of two or all of them. Figure 4.6 shows the type of each rule based on the type of its patterns.

Example 2. *As an example of the process, the rule cls-svf1 has antecedents*

(*?x owl:someValuesFrom ?y*)

(*?x owl:onProperty ?p*)

(*?u ?p ?v*)

(*?v rdf:type ?y*)

and consesquent (*?u rdf:type?x*)

The rule in the above example is of the type SNR, as shown in Figure 4.6, because it has four patterns (i.e. antecedents) the first two is selective (i.e. S), the third pattern is non-selective (i.e. N) and the last pattern is recursive (i.e. R). Similarly, the type of the other rules in Figure 4.6 is based on their patterns which can be found in Appendix B.

One step in reaching the consequent is to establish a list of triples that match the selective triple pattern (*?x owl:someValuesFrom ?y*), which will bind only to triples containing *owl:someValuesFrom* as a predicate. A further step to reach the consequent of cls-svf1 is to bind triples matching the non-selective pattern $(?u, ?p, ?v)$. Rule cls-svf1 also requires the recursive antecedent (*?v rdf:type ?y*).

This antecedent can be established by consulting any of the rules in the rdf:type OR tree shown in Figure 4.6 which also includes a call to cls-svf1.

For each triple in the delta, the purpose of executing the rules is to see if the triple can be inferred in the updated set (M or M'). At this point the OR tree for that triple can be terminated. To achieve this, the order of executing these patterns starts with selective patterns, followed by the non-selective patterns and finally the recursive pattern because they are the most complex in terms of execution.

Recursive patterns are potentially expensive in terms of their execution because they generate further calls until the desired conclusion is found or until no more patterns can be executed. In contrast, the execution of the selective pattern is relatively simple, because a conclusion such as (?*u* *rdf:type* ?*x*) can be derived from the knowledge base simply by finding that the object of the triple (?*x*) exists in the someValuesFrom table. In the case where this object does not exist, the rest of the patterns in the rule no longer require further execution. Thus, in this example, the patterns (?*x* *owl:someValuesFrom* ?*y*) and (?*x* *owl:onProperty* ?*p*) are executed first because they are both selective patterns and can save the execution of the other patterns if no triple in the knowledge base matches one of these patterns. Subsequently the pattern (?*u* ?*p* ?*v*) and finally the pattern (?*v* *rdf:type* ?*y*) are matched because they may require further execution of rules if no triples in the knowledge base match the pattern. Decomposing the execution of these patterns in this way may avoid the searches required in executing

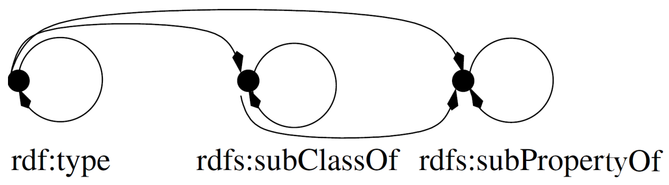


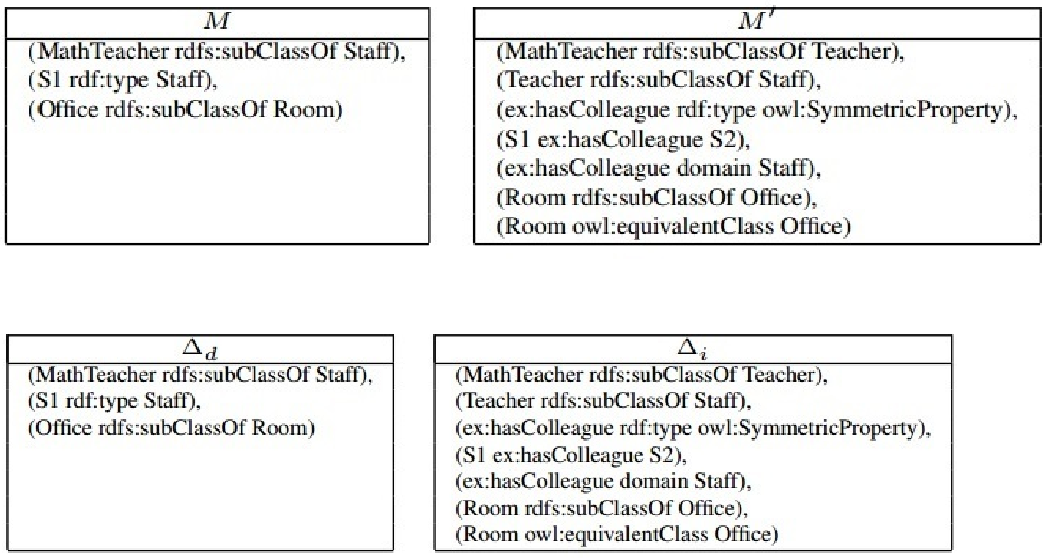
Fig. 4.7: Overlapped OR trees. The round arrows indicate a recursive call from within the OR tree

them as a single query and consequently reduce the execution time [KWE10]. Decomposed sections can then be executed separately following the order described above.

As a result of the recursive patterns, the different OR trees overlap because recursive patterns in one OR tree may require further application of other rules which may be in other OR trees. Figure 4.7 shows the overlapped OR trees as concluded from the rules they contain. All OR trees (i.e. rdf:type, rdfs:subclassOf, and rdfs:subpropertyOf trees) have a recursive call from within the OR tree. In addition, from the rules in the OR trees in Figure 4.6, rdf:type OR tree requires a call to execute rules from other OR trees (i.e. rdfs:subClassOf and rdfs:subPropertyOf), and rdfs:subClassOf OR tree requires a call to execute rules from rdfs:subPropertyOf OR tree, while rdfs:subPropertyOf OR tree has only a recursive call to itself.

4.4.2 Pruning OR trees

The process of pruning OR trees starts with the generation of ΔE. Each triple in the delta set is checked against the dataset to determine whether it is inferable,



M
(MathTeacher rdfs:subClassOf Staff),
(S1 rdf:type Staff),
(Office rdfs:subClassOf Room)

M'
(MathTeacher rdfs:subClassOf Teacher),
(Teacher rdfs:subClassOf Staff),
(ex:hasColleague rdf:type owl:SymmetricProperty),
(S1 ex:hasColleague S2),
(ex:hasColleague domain Staff),
(Room rdfs:subClassOf Office),
(Room owl:equivalentClass Office)

Δ_d
(MathTeacher rdfs:subClassOf Staff),
(S1 rdf:type Staff),
(Office rdfs:subClassOf Room)

Δ_i
(MathTeacher rdfs:subClassOf Teacher),
(Teacher rdfs:subClassOf Staff),
(ex:hasColleague rdf:type owl:SymmetricProperty),
(S1 ex:hasColleague S2),
(ex:hasColleague domain Staff),
(Room rdfs:subClassOf Office),
(Room owl:equivalentClass Office)

Fig. 4.8: Sample data structure before and after update with the insert and delete sets

which would allow it to be removed from the delta set and hence reduce the delta size. This process requires the execution of each rule in the OR tree for the corresponding triple (i.e. triples with the a property: *rdf:type*, *rdfs:subclassOf*, or *rdfs:subPropertyOf)*. In a relational data store implementation, the execution of these rules involves joins between tables in the database that match the patterns in these rules. However, some execution of these rules, and therefore joins between tables in the database, are unnecessary and can be avoided as they will not lead to the desired conclusion.

In the example shown in Figure 4.8, M and M' are two different versions of an OWL knowledge base with M' being a newer version of M. The explicit differences (ΔE) between the two versions are shown in the same figure. This example focuses only on the deletion set of triples because the process of reducing this set does not require further checking to perform correct and valid reduction of the delta, as would be the case if the insertion set was involved. Reducing the deletion set requires the application of OWL inference rules against M', the newer version of the dataset. The deletion set in the delta contains a triple (*MathTeacher rdfs:subClassOf Staff*). In order to reduce the delta size, the triple needs to be checked to see if it is inferable in M'. This involves executing the rules in the OR tree for the *subClassOf* property shown in Figure 4.6 until this triple is inferred by the execution of one of these rules or until no more rules can be applied. In the first case, the triple is removed from the delta. In the second case the triple should remain in the delta. The other rules used in this process are also identified in Figure 4.6. Using as an example the recursive rule scm-sco, the execution of this rule requires a recursive call to the rule until the triple is inferred or no more recursive calls can be applied.

Each time a recursive call is made, a self-join to the *subClassOf* table is required in order to infer the triple (*MathTeacher rdfs:subClassOf Staff*). Initially, a search is carried out to find if the patterns (*MathTeacher rdfs:subClassOf ?c*) and (*?c rdfs:subClassOf Staff*) exist. If they can be found, the triple is inferable and can safely be removed from the delta. However, if triples matching these patterns do not exist then the straightforward approach is to find all the patterns that have *MathTeacher* in the subject position and apply a recursive call to this rule until the main triple (i.e. (*MathTeacher rdfs:subClassOf Staff*)) is inferred or no more patterns can be generated from the dataset which will terminate the recursive execution of the rules as RDF datasets contains a finite number of triples.

If triples such as (*MathTeacher rdfs:subClassOf C1*), (*MathTeacher rdfs:subClassOf C2*) etc. exist in M' then these triples are added to a list of those in M' that have *MathTeacher* in the subject position. In the context of the scm-sco rule and consideration of the triple (*MathTeacher rdfs:subClassOf C1*), a recursive call is made to the rule in order to infer (*C1 rdfs:subClassOf y*) by searching for the patterns (*C1 rdfs:subClassOf ?x*) and (*?x rdfs:subClassOf y*). If these patterns do not exist in M', then all the patterns that have $C1$ in the subject position are generated and this process continues until the triple is inferred or no further patterns are generated.

This approach requires successive self-joins in the triple store, which it may not be possible to infer the triple in order to reduce the delta size. There is potential advantage in avoiding unnecessary rule execution since this will result in potentially multiple self-joins in the triple store.

We proposed a rule pruning method that prune unnecessary rules in the OR tree. This approach is based on initially checking whether both the subject and object of a triple exist in the appropriate positions as defined by the patterns in each rule before executing that rule. If both subject and object exist then the rule is applied otherwise it is pruned from execution. The checking avoids the use of joins in the triple store, thereby reducing the effort involved in further processing. Algorithm 2 describes this method.

Finding a subject or an object of a triple can be done in two ways based on the type of the pattern of the rule. Finding a matching triple for a selective pattern requires checking whether the subject and the object of a triple exist in the exactly the same positions as defined by the patterns of the rule. This is shown in the algorithm as *FindMatchInFixedPosition*.

The other way of finding a subject or an object of a triple is to check whether the subject and the object of a triple exist in any position (i.e. the subject position or the object position). Finding matching for non-selective and recursive patterns is done using this approach which is shown in the algorithm as *FindMatchInAnyPosition*. The evaluation of the pruning algorithm described in this work is based on a relational triple store which is explained in Section 5.4.

The relational representation of triple data used by the RDF model is decomposed into a number of tables as described in Section 4.2. The complexity of Algorithm 2 is based on (μ) which is the number of triple in any of these tables and thus normally very much less than (N), being the total number of triples in the dataset. The algorithm step in line 6 is consequently of complexity $O(\mu)$. The step in line 11 is similarly of $O(\mu)$. The last step in the algorithm (in line 14) requires joining of the table that represent the decomposed triples. The worst case of this process will be $O(\mu^2)$ which gives an indication of the time complexity of the algorithm as a whole. The actual performance can be improved by indexing.

Generally, to infer the triple (x *rdfs:subClassOf* y), both x and y are checked to see if they exist in the *subClassOf* table in the subject position of one triple and the object position of another triple respectively. No joins are needed in this step. If the method returns true then the rule can be executed, otherwise this rule is pruned and no further checking of the consequent takes place. If the rule is pruned, the other rules in the OR tree are checked in the same way until a true value is applied or no more rules remain in the OR tree.

For example, a triple such as ($S1$ *rdf:type Staff*) in the deletion set, reduction in the delta size and particularly the deletion set can be achieved by checking whether the triple is inferable in M' or not by applying rules in the *rdf:type* OR tree. Before proceeding with the execution of these rules, the subject ($S1$) and the object (*Staff*) of this triple are checked against all patterns within the rules of the *rdf:type* OR tree. The rule prp-dom, for instance, has two patterns ($?p$ *rdfs:domain* $?c$) and ($?x$ $?p$ $?y$) in its body which derive the conclusion ($?x$ *rdf:type* $?c$). To check if this rule can be pruned, we need to check if the subject of the triple ($S1$ *rdf:type Staff*) exists in either the subject column or the object column of the general triple table. Furthermore, it is necessary to check the object column in the rdfs:domain table to ascertain whether value *Staff* exists as the object of that triple.

Checking the existence of the subject $S1$ in either position of the triple table is an exceptional case that appears in all rules containing a non-terminological pattern (i.e. the property of the triple is a user-defined property) such as ($?x$ $?p$ $?y$). The reason for checking the existence of the value in either the subject or the object columns of the general triple table is because triples matching non-terminological patterns can be inferred by other rules which include non-terminological patterns in their bodies that reverse the positions of the values of the subject and object. An example of such a rule is prp-symp, which has two antecedents in its body: (*?p rdf:type owl:SymmetricProperty*) and (*?y ?p ?x*), and derives a conclusion (*?x ?p ?y*).

Checking the value $S1$ of the triple in only the subject column of the triple table is not enough to decide if the rule can be pruned as triples matching the non-terminological pattern (*?x ?p ?y*) can be concluded by other rules having non-terminological patterns with the value of the subject in the object position (*?y ?p ?x*).

According to the rule prp-dom, checking the subject of the triple ($S1$ *rdf:type Staff*) in only the subject column of the general triple table will result in pruning this rule as $S1$ does not exist in the subject column in this table as shown in Figure 4.8. However, M' contains the triples (*ex:hasColleague rdf:type owl:SymmetricProperty*) and ($S2$ *ex:hasColleague* $S1$) which according to rule prp-symp can produce as a conclusion the triple ($S1$ *ex:hasColleague* $S2$). This has $S1$ in the subject position which is necessary for the execution of the rule prp-dom.

To summarise, pruning a rule involves checking whether the value of the subject and the object of the corresponding triples in the delta set exist in the same position as stated in the patterns of the body of that particular rule. Only when a rule contains a non-terminological pattern then the existence of a particular value is checked against either the subject position or the object

Algorithm 2: Reasoning with pruned rules

```
Data: $t \in \Delta$, orTree
Result: true if the update is inferable in the knowledge base otherwise
        false
1  rules = orTree.getRules(t)//get the rules from the corresponding orTree
2  result = false
3  while rule in rules AND result == false do
4      if ruleType == SNR or SN or SR or S then
           /* SNR - Selective, Non-Selective and Recursive      */
           /* SN - Selective and Non-Selective                  */
           /* SR - Selective and Recursive                      */
           /* S - Selective                                     */
5          selectivePatterns = rule.getSelectivePatterns()
6          if not{selectivePatterns.FindMatchInFixedPosition(t)} then
7              rule.prune()
8              continue //to the next rule in rules
9      if ruleType == SNR or NR or SN or SR then
           /* NR - Non-Selective and Recursive                  */
10         nonSelectivePatterns = rule.getNonSelectivePatterns()
11         if not{nonSelectivePatterns.FindMatchInAnyPosition(t)} then
12             rule.prune()
13             continue //to the next rule in rules
14     result = rule.apply()
15 return result
```

position.

4.4.3 Blank node pre-processing

RDF model theory [HM04] characterises blank nodes as having local scope within the file that contains them. Such nodes act as existential quantifiers over a set of resources in which the identifiers of the blank nodes are not significant. In practice blank nodes are used to describe multi-component structures represented by RDF containers, to describe reification (i.e. triples about triples) or to represent complex information. Given the local scope of blank nodes, it is not possible to rely on their identifiers being consistent between successive ontology versions. However, chains of blank nodes hold information that may be useful in the process of reasoning about updates between ontology versions. Loading blank nodes involves pre-processing these nodes to trace graphs of triples that contain them. This step is useful for matching blank nodes when computing the differences between two versions of an ontology and the reduction of the delta size.

Tracing blank nodes is based on the assumption that such chains start with a non-blank node (i.e. a URI) in the subject position of a triple. These blank node chains may form a tree such as that shown in Figure 4.9 that represents the N-triple shown in Figure 4.10. If a triple with a non-blank node in the subject position and a blank node in the object positon is encountered, the tracing process begins tracing all connected blank nodes until no more related triples are found. The length of the chain is equal to the number of triples containing the connected blank node. The example in Figure 4.10 gives a chain length of 9. Each chain of blank nodes is held in the triple store along with the length of the chain in order to use it in the matching process. Effectively, each chain now has an ID to distinguish the group of triples that belongs to it. Results from experiments on blank node processing are contained in Section 5.4.

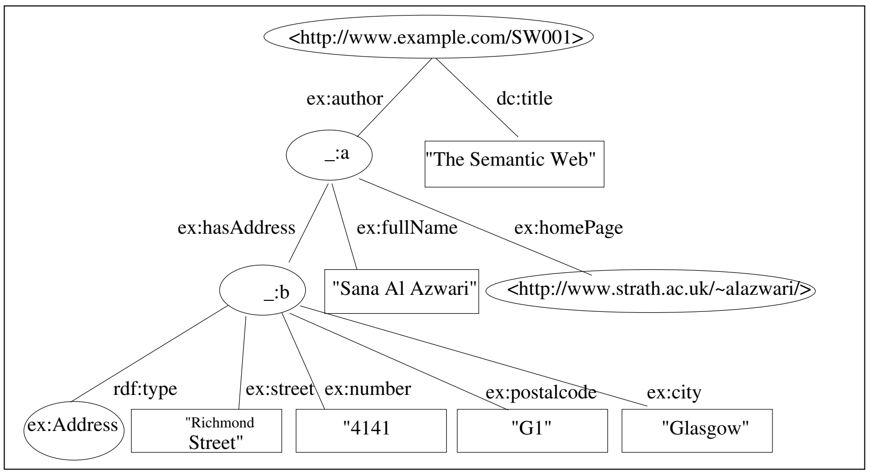


Fig. 4.9: Blank nodes tree structures

```
<http://www.example.com/SW001> dc:title “The Semantic Web"
<http://www.example.com/SW001> ex:author _:a .
_:a ex:fullName “Sana Al Azwari" .
_:a ex:homePage <http://www.strath.ac.uk/~alazwari/> .
_:a ex:hasAddress _:b .
_:b rdf:type ex:Address .
_:b ex:street “Richmond Street“ .
_:b ex:number “4141“ .
_:b ex:postalcode “G1" .
_:b ex:city “Glasgow" .
```

Fig. 4.10: Blank nodes chain example

4.5 Conclusion

This chapter has explained the methods used to evaluate the hypothesis that exploiting the richness of complex ontology languages will support reduction in the size of updates. Initially it explains work to asses the performance of updates to RDF structures. It then considers the ways in which this process can be simplified by inference and pruning in the context of RDFS and OWL 2 semantics. There are three main contributions in this work. The first contribution is to provide an understanding of the effect of pruning method in the context of RDF updates. Second, this work proposes a novel method for correcting the unsoundness of dense delta which produce a small in size number of updates. Third, in this work, the reduction of the delta is evaluated in the context of complex ontology languages, the OWL 2 RL/RDF. We propose a rule pruning method that prunes the application of unnecessary OWL 2RL/RDF rules during the process of delta reduction. The next chapter presents the experimental results from this process and discusses their meaning.

5. RESULTS AND DISCUSSION

Experimental work reported in this chapter aims to evaluate the overall hypothesis that exploiting ontology rules when generating RDF updates results in reduced delta sizes. Details of hardware and software configuration of systems used in the evaluation is listed in Appendix E. All change detection techniques were implemented using Java with Jena. For the triple store, a MySQL database was used to store RDF versions and deltas. This allows for large data sets to be processed independently of main memory restrictions.

5.1 RDF views update

The hypothesis of the first set of experiments is that updating RDF structures is an expensive process in terms of the work required to maintain real-world data structures. This section includes a comparison between the two update approaches discussed in the Section 4.1. These approaches are evaluated in terms of view update time.

For this experiment a full database containing both the GO vocabulary and associations between GO terms and gene products including the Uniprot Taxonomy from the bioinformatics domain was used. This data set was chosen because it is frequently updated and a new version of it is released every month. The data set includes ten versions dated from 2005 to 2014. That is a first release of the data set for each year. A yearly version was chosen to increase the gap between versions and therefore get more differences between the RDF KBs. Moreover, the transformation from one version to another was as folows: the oldest version (i.e.the 2005 version) was transformed to the other different versions starting from the 2006 version and ending with the 2014 version. This also gradually increases the number of differences between the versions when measuring the delta size and the performance of the different change detection methods.

As discussed in Section 4.1, RQ2 is concerned with assessing the update performance in both random and specific update approaches. To address this issue, the CPU time for applying the update was measured. Figure 5.1 shows the update times when a fixed number of triples (10,000 triples) are updated using two types of updates: deletion and insertion performed against all six models using both approaches for selecting the triples: random selection and specific selection. The x-axis represents the size of the RDF models in triples as described in Table 4.1, while the y-axis represents the CPU time in milliseconds.

Figure 5.2 shows the update times when a variable number of triples (10, 100, 1000, 10,000, 100,000, and 1,000,000 triples) are randomly and specifically selected to be deleted from and inserted to model6 (130MB), which is the largest model in the dataset. The x-axis in Figure 5.2 represents the number of updated triples, while the y-axis represents the CPU time in milliseconds.

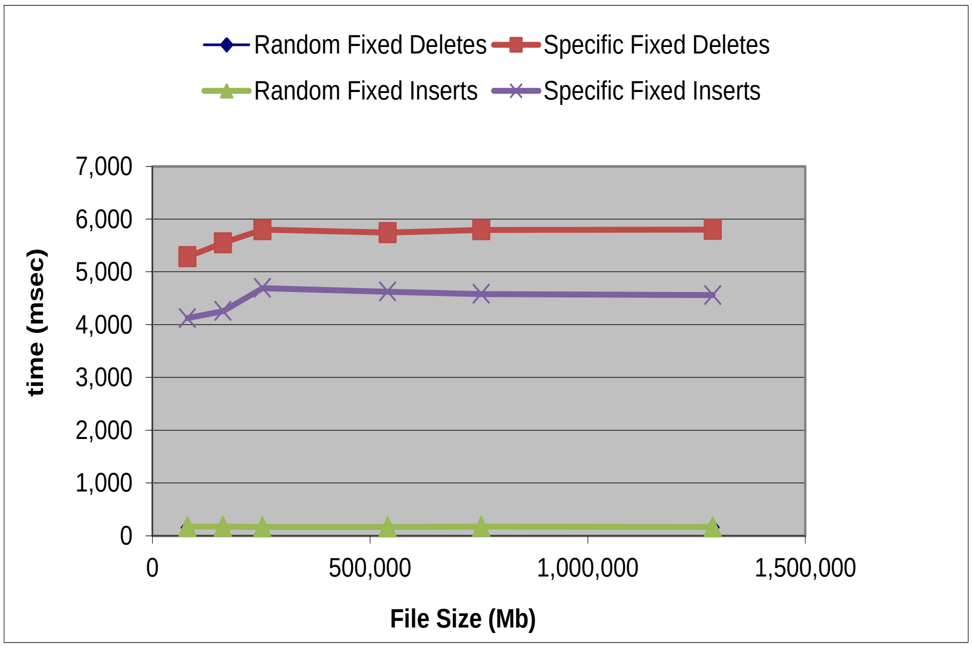


Fig. 5.1: Random and Specific Fixed Update of 10,000 triples. Random fixed deletes is superimposed by random fixed inserts

In addition, it can be seen from the results that the delete operation in both types of update is slower than the insert operation, which means that these two operations do not use equivalent CPU time.

As clearly shown in both figures, specific update is slower compared to random update as the latter involves only the time for adjusting RDF structures while most of the update time in the specific update is taken up by searching the structure. Moreover, this time increases as the number of updates increases and consequently it is useful to prune these updates to minimize the time taken to update RDF structures.

In the light of the above results, suppose a scenario where there are multiple applications concurrently using a materialized view of an RDF triple store (Figure 5.3). If one of these applications generates a large number of updates that

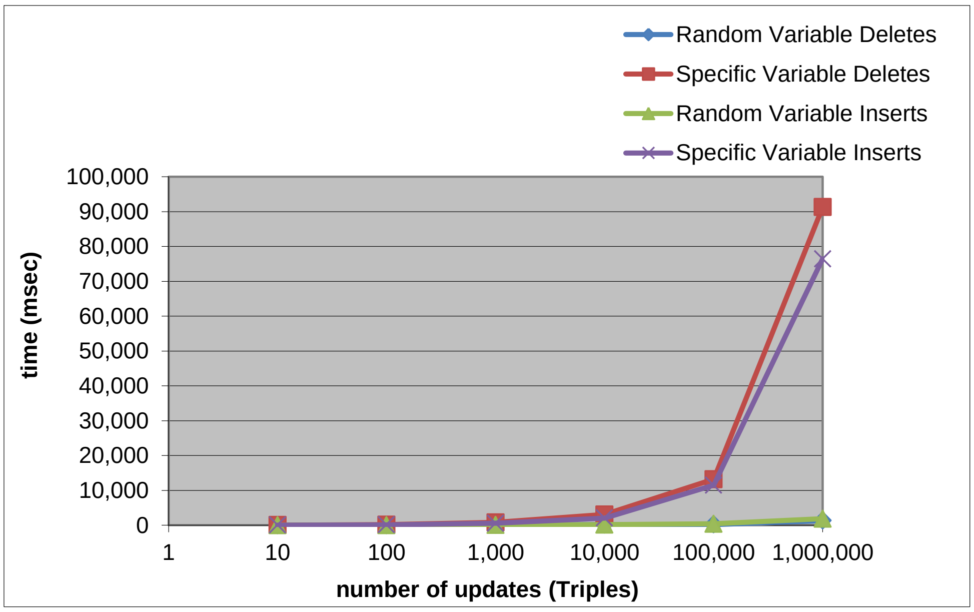


Fig. 5.2: Random and specific variable updates to the largest (130MB) data structure. Random variable deletes is superimposed by random variable inserts.

are to be applied to the view, the other applications need to wait until the view is updated. From the results in Figure 5.2, 100,000 updates will result in a delay of approximately 10 seconds while the view is adjusted. Clearly, this delay may not be acceptable for the other applications that are sharing the data structure.

5.1.1 Validation and performance evaluation

To validate the framework, results of the update process which include inserts and deletes using SPARUL and SPARQL queries in the experiment are compared to SQL queries over a non-memory resident data store (Figure 5.4). To do this, each RDF dataset in Table 4.1 is stored in a corresponding triple table using MySQL database. Each triple table consists of three columns: Subject, Property

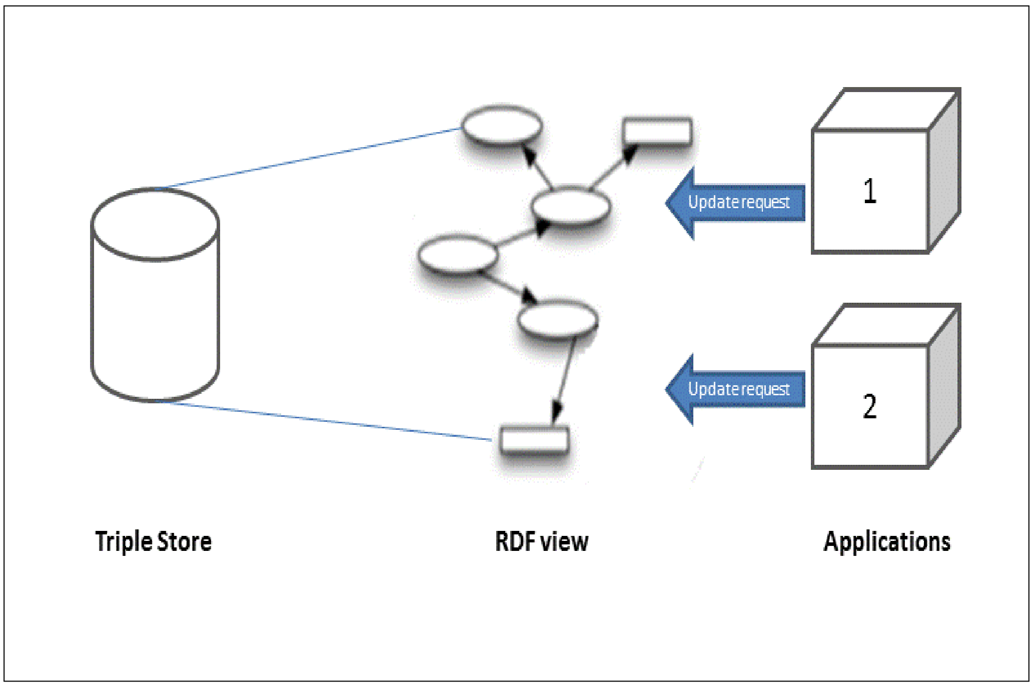


Fig. 5.3: Update requests from application to RDF views.

and Object to store the triples and a column to determine the type of the Object in each triple whether it is a URL or a literal. These tables are updated using SQL queries and for each table a list of triple subjects was separately prepared and the entries in this list were used to identify the triples to update the table. After updating the tables an RDF file is generated for each updated table and an in-memory view is created for the purpose of comparing the results. The same list of triple subjects is used to update views of the RDF models in Table 4.1 using SPARUL and SPARQL in our framework. Views from the two different approaches are then compared to see whether they are identical or different. When comparing the results of updating all six RDF views using the two approaches, the maintained view was the same as the regenerated view which validates the framework.

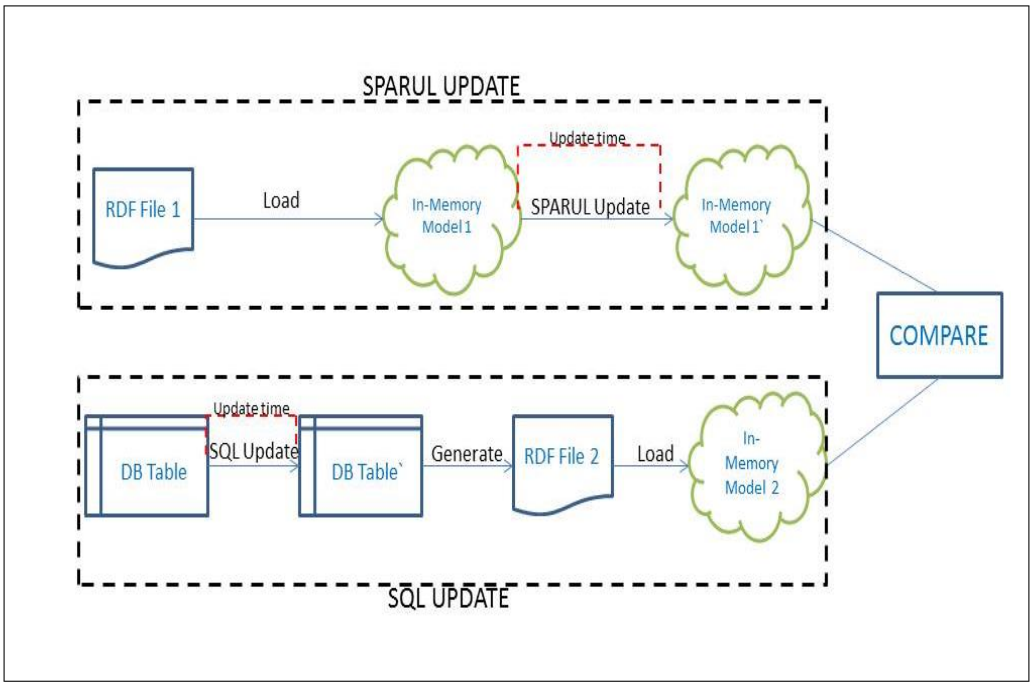


Fig. 5.4: Validation process for view update.

The CPU time for both types of updates; inserts and deletes in the two approaches were recorded as highlighted with the dotted red line in Figure 5.4. The times for each type of update were compared to evaluate the performance of both approaches. As shown in Figure 5.5, inserting triples to a table using SQL inserts is more CPU intensive than inserting to the same number of triples to an in-memory view using SPARUL and SPARQL queries, this is reasonable as inserting to a database table is actually saving information to a disk which requires writing this information to a disk and creating indexes for it. On the other hand, SQL deletes of triples from a table appears to be faster than deleting the same triples from an in-memory view using SPARUL and SPARQL queries. This may be because updating from in-memory view requires the upload of both models into in-memory which is memory expensive. Also deletions in DBMSs are

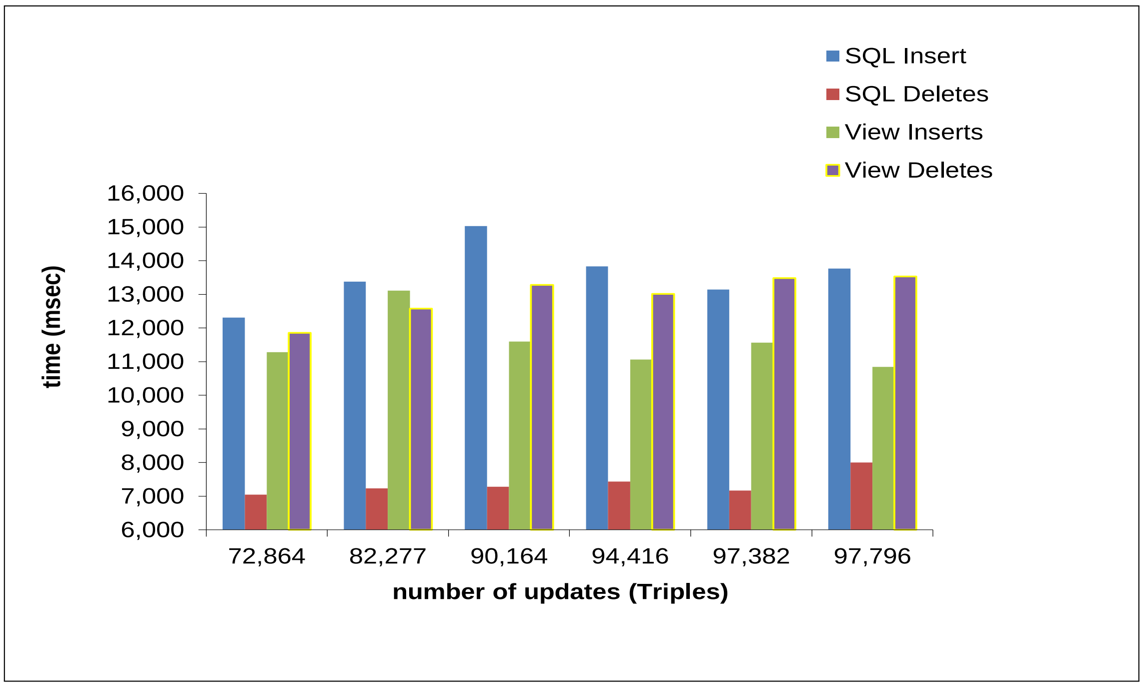


Fig. 5.5: Performance Evaluation of RDF update using SQL queries and SPARUL and SPARQL queries.

handled by only marking the record as being deleted.

5.2 Change detection techniques

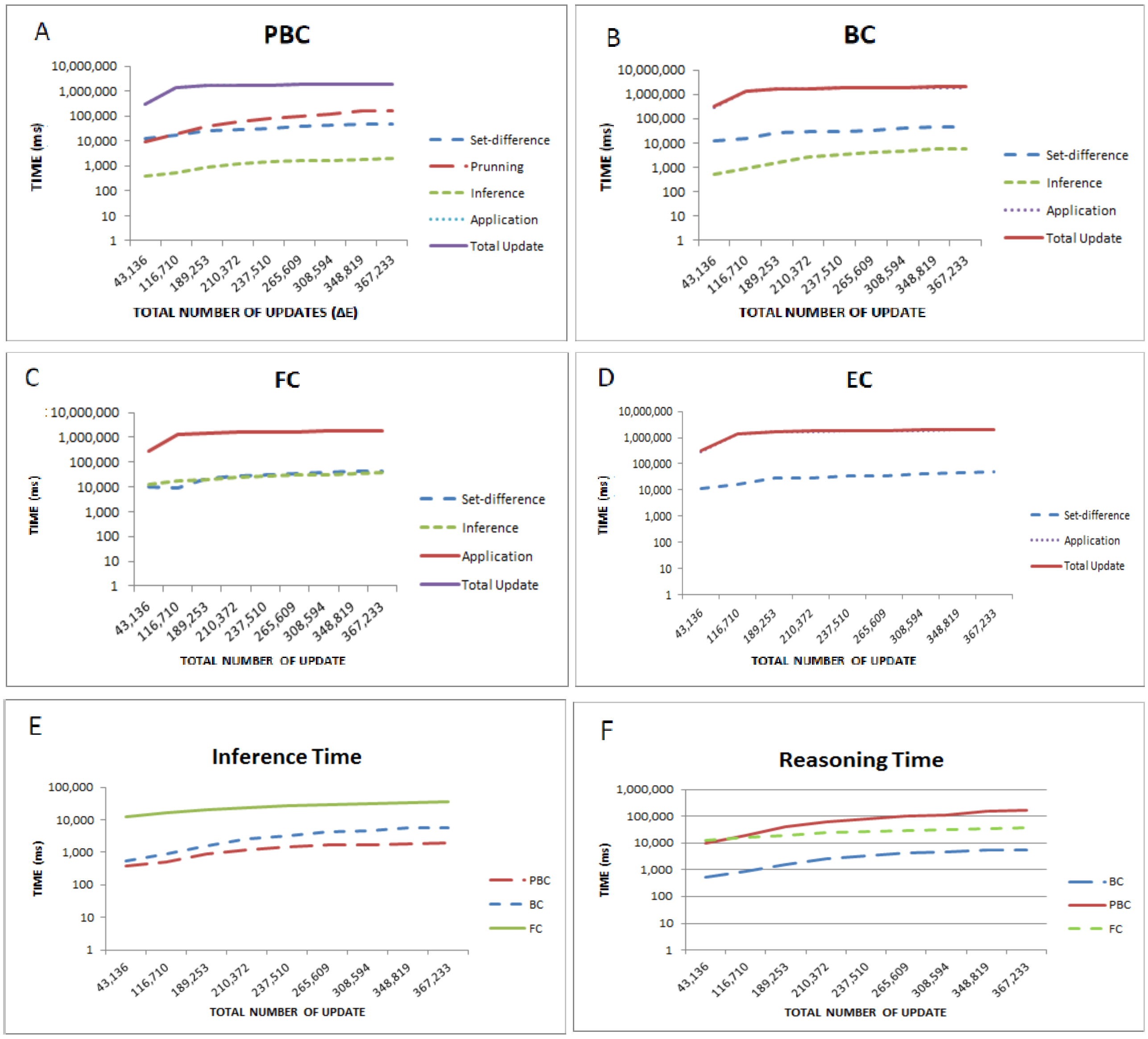


Fig. 5.6: The results of the performance measurement in change detection techniques. In (A), (B), (C) and (D) the application line is superimposed by the total update line.

	(1)		(2)			(3)	
	Delta Size		Triples used in reasoning			Triples excluded from reasoning	
year-range	ΔE	ΔED	FC	BC	PBC	BC	PBC
2005-2006	43,136	42,770	45,400	817	562	~98%	~99%
2005-2007	116,710	116,228	52,449	1,457	888	~97%	~98%
2005-2008	189,253	188,512	59,995	2,880	1,434	~95%	~98%
2005-2009	210,372	209,334	66,264	4,086	1,969	~94%	~97%
2005-2010	237,510	236,190	74,389	4,754	2,433	~94%	~97%
2005-2011	265,609	264,221	81,538	5,513	2,790	~93%	~97%
2005-2012	308,594	307,163	87,751	5,895	2,883	~93%	~97%
2005-2013	348,819	347,292	99,425	6,735	3,471	~93%	~97%
2005-2014	367,233	365,629	104,209	7,080	3,638	~94%	~97%

Tab. 5.1: Triple statistics. (1) The total number of changes collected using ΔE and ΔED. (2) The count of triples participated in the inference process. (3) The percentage of triples excluded from reasoning.

The previous section describes the results of experiments that characterize the performance which is encountered when updating RDF collections either in memory or on disk. One way of limiting these performance problems is to reduce the number of updates that are needed (i.e. reduce the delta size). The hypothesis being evaluated in these experiments is that the update size can be reduced by pruning but this is in itself a potentially expensive process.

Since the inference-based approaches (FC, BC and PBC) use the ΔED, the delta reductions produced by these methods are identical[1], the sizes of the delta produced by these methods are equivalent. The delta sizes for the FC, BC and PBC are smaller compared to the delta produced by the explicit method where no inference is applied and therefore no reductions from the set of differences are made (Table 5.1).

[1] As explained in Section 4.2

Table 5.1 column 2 shows the number of triples that participated in the inference process. FC has the highest number of triples used in the inference process as a result of calculating the full closure. This number is reduced by ∼94% when changes were detected using BC. PBC, on the other hand, can further prune ∼47% of these triples in BC.

In addition to the delta size and the number of triples in the inference process, the performance was evaluated by separately measuring the execution time for each core operation in change detection. These core operations include the total execution time, the time taken to produce the deltas (set-difference), the pruning time (in the case of PBC only), the inference time (in the case of FC, BC and PBC only), the reasoning time (which consists of the time to complete both inferencing and pruning in the case of PBC only) and the set-operation time (i.e. the application of the delta). The execution times for this experiment are shown in Appendix D.

The graphs in Figure 5.6, which are plotted in logarithmic scale in the y-axis, show the results of comparing the performance of the three techniques with an increasing number, i.e. 50,000 to 1M, of both types of updates; insertions and deletions. From Figure 5.6 (E) it can be seen that the inference time in PBC is faster by about 1.5-3 times than that in BC , and of FC by about 19-33 times as a result of the pruning process. However, although pruning RDF has efficiently reduced the inference time, the pruning operation in PBC takes most of the overall time in change detection as shown in Figure 5.6 (A). Adding the pruning time to the inference time, because they are both part of the reasoning process, gives a more realistic view of the overall performance shown in Figure 5.6 (F). Moreover, the pruning time appears to increase significantly with the increase in the number of pruned triples.

Results from these experiments show that change detection using PBC has a penalty of about 15-30 times by comparison with BC when pruning time is counted as part of the reasoning process.

5.3 Correct dense delta

In ΔE (Definition 2.5.2) the delta is calculated using the explicit triples only for both set of updates; the delete set and the insert set. It does not exploit the inferred triples for the reduction of the delta. While ΔED (Definition 2.5.3) calculated the delta by exploiting inferred triples for reducing the deletion set only while the insertion set is calculated explicitly. ΔD (Definition 2.5.4) on the other hand, calculates the delta by exploiting inferred triples for producing both sets of updates. However, as explained in Section 2.5, this delta does not produce a sound (i.e. correct) deltas.

In the previous section, the delta size is reduced by the inference and pruning processes which are applied only on the set of deletions and not the insertions (i.e. ΔED). This is due to the lack of guarantee of correctness of the delta produced by inferencing over both sets of update (i.e. ΔD). Thus, in this thesis we proposed a correction method that can improve the correctness of the deltas produced by backward-chaining change detection approaches (BC and PBC) as explained in Section 4.3. This method is called correct dense delta ΔD_c

The hypothesis being tested in this part of the work is that the correct dense delta ΔD_c will in general produce a smaller delta than the explicit dense delta ΔED and at the same time result in a correct update of M to M'.

To evaluate the correction method, the processing time for computing the delta and delta size of updates to enhanced RDF KBs of different sizes are assessed. The objective of this evaluation is to compare the performance of the different delta computation methods (i.e. ΔE, ΔED, ΔD_c) and approaches (i.e. forward-chaining (FC), backward-chaining (BC) and pruned backward-chaining (PBC)) by measuring and comparing their delta computation times over synthetic datasets and by validating their effect on the integrity of the resulting RDFS KBs.

As with the experiment described in Section 4.1, the dataset contains both GO vocabulary and associations between GO terms and gene products including the Uniprot Taxonomy. The reason for chosing this dataset is provided on page 102 . The dataset includes five versions selected to show a range of values over the period 2005 and 2014. Using this dataset, the oldest version (i.e. the 2005 version) was transformed to five versions released between 2006 and 2014. This gradually increases the delta size with a consequent effect on the performance of the different change detection methods. The real-world data was enhanced by synthetic data prepared by incorporating 20% additional triples representing *sub-*

						Reduction strength		
Versions	**M**	ΔE	ΔED	ΔD_c	ΔD	ΔED	ΔD_c	ΔD
$(M - M1')$	121,374	48,136	47,270	44,270	44,212	1.8%	8.0%	8.2%
$(M - M2')$	127,374	126,710	125,228	119,228	119,098	1.2%	5.9%	6.0%
$(M - M3')$	139,374	230,372	227,334	215,334	214,926	1.3%	6.5%	6.7%
$(M - M4')$	157,374	343,594	338,663	317,662	317,109	1.4%	7.5%	7.7%
$(M - M5')$	169,374	412,233	406,129	379,129	378,482	1.5%	8.0%	8.2%

Tab. 5.2: Triple counts used in evaluation.

Abbr.	**delta**
E	explicit
EDFC	explicit dense, forward chaining
EDBC	explicit dense, backward chaining
EDPBC	explicit dense, pruned,backward chaining
D_cFC	corrected dense,forward chaining
D_cBC	corrected dense, backward chaining
D_cPBC	corrected dense, pruned, backward chaining

Tab. 5.3: Change detection techniques.

Class, *subProperty* and *type* properties. Synthetic data was added to ensure that subProperty rule was exercised and to arrange for the model to contain redundant triples (i.e. explicit data that can also be inferred from antecedents). The level of enhancement was chosen to secure a measurable effect without obscuring the structure of the original data.

Using the enhanced datasets, change detection techniques shown in Table 5.3 were implemented. A triple store was constructed in MySQL to handle the RDF collections and the deltas. Indexing was excluded to preserve the validity of the use-case. The Jena framework was used to read the RDF dataset into the triple store and to validate change detection techniques by comparing the updated RDF dataset with the target RDF dataset.

The consistency of M' after delta application was evaluated by comparing the in-memory M' produced by applying the delta to M in the database with the original in-memory M' using the Jena isIsomorphic method. Applying ΔD_c using the approach described above was found to result in the same M' as that used to generate the delta. By contrast, tests carried out to assess the consistency of applying the uncorrected ΔD indicate that in all the models tested, this approach always failed to produce consistent updates.

Table 5.2, Table 5.3 and Figures 5.7-5.10 report the delta sizes and the delta computation times, respectively. From Table 5.2, the deltas produced by ΔE exceed those of ΔED and ΔD. The latter deltas are smaller than those produced by ΔE as a consequence of applying inference on the delete set of triples ΔED. ΔD_c further reduces the deltas as a result of inferring both the delete and insert set of triples when calculating the deltas. ΔD in turn may be smaller than ΔD_c but its application in the update process may lead to an inconsistent result.

Table 5.2 reports the reduction strength, which is defined as the percentage reduction achieved by inference i.e. $\frac{|\Delta E|-|\Delta ED|}{|\Delta E|}$ or $\frac{|\Delta E|-|\Delta D_c|}{|\Delta E|}$. Reduction strength indicates when the size of ΔE, ΔED and ΔD_c are different i.e. when inference is capable of making a difference to the size of the delta. Reduction strength in ΔD_c is greater than that in ΔED as a result of considering both sets of updates. The reduction strength in ΔD is slightly greater than that of ΔD_c, however, these deltas produced by ΔD are not always correct.

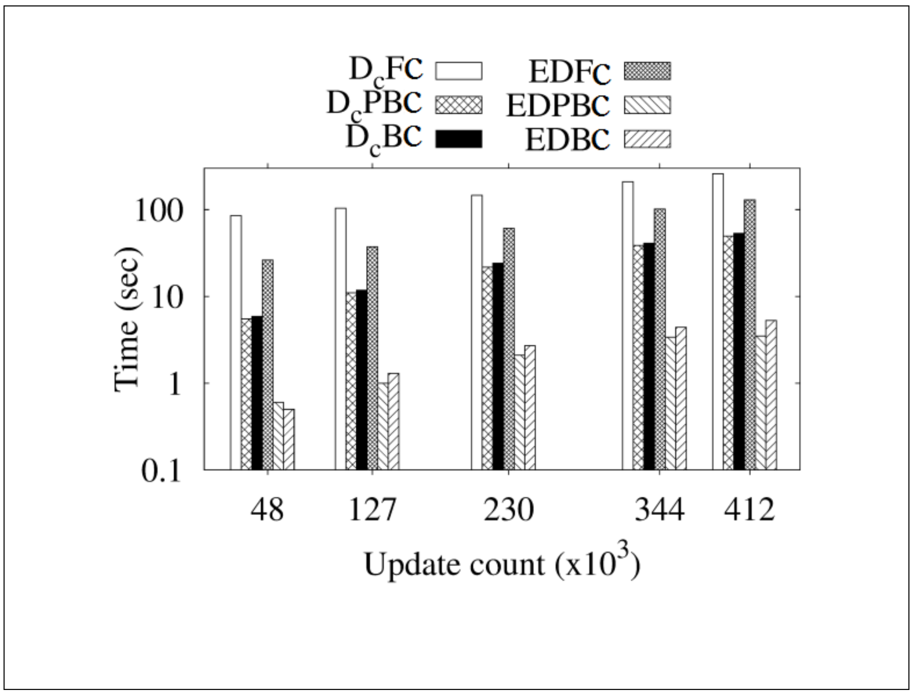


Fig. 5.7: Inference time

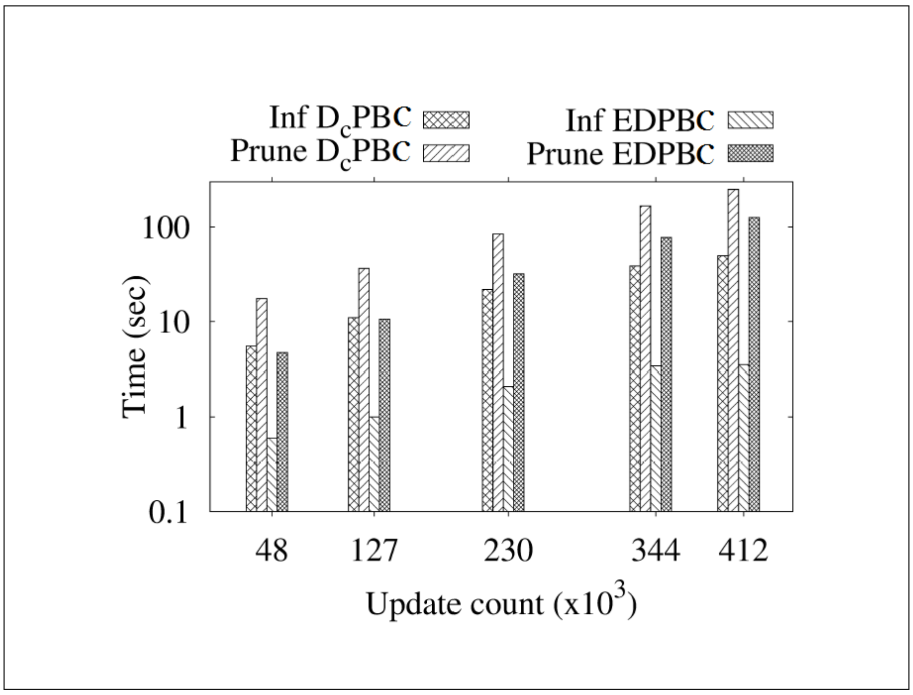


Fig. 5.8: Reasoning times

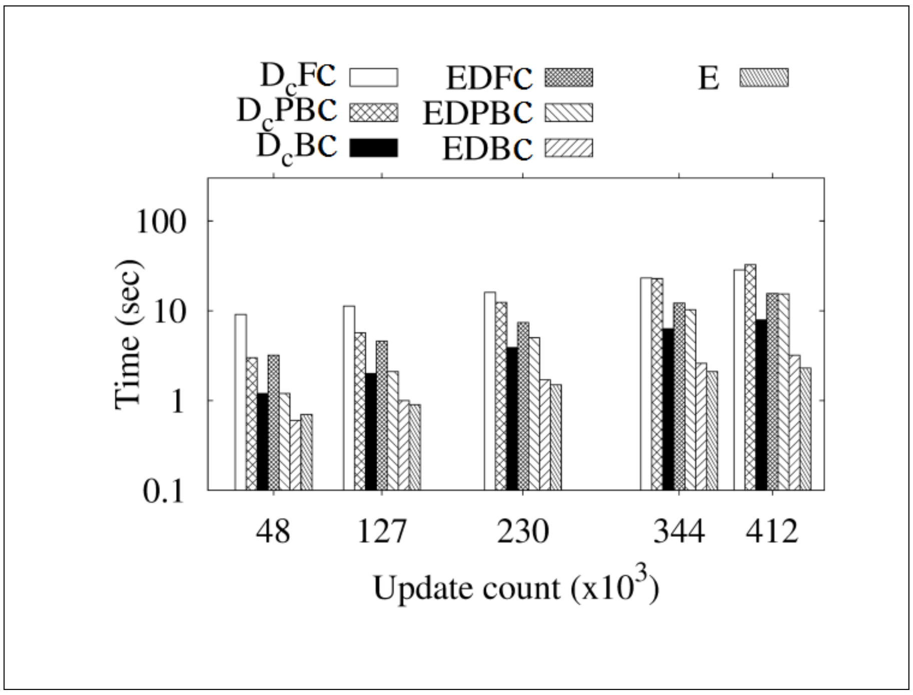


Fig. 5.9: Delta time

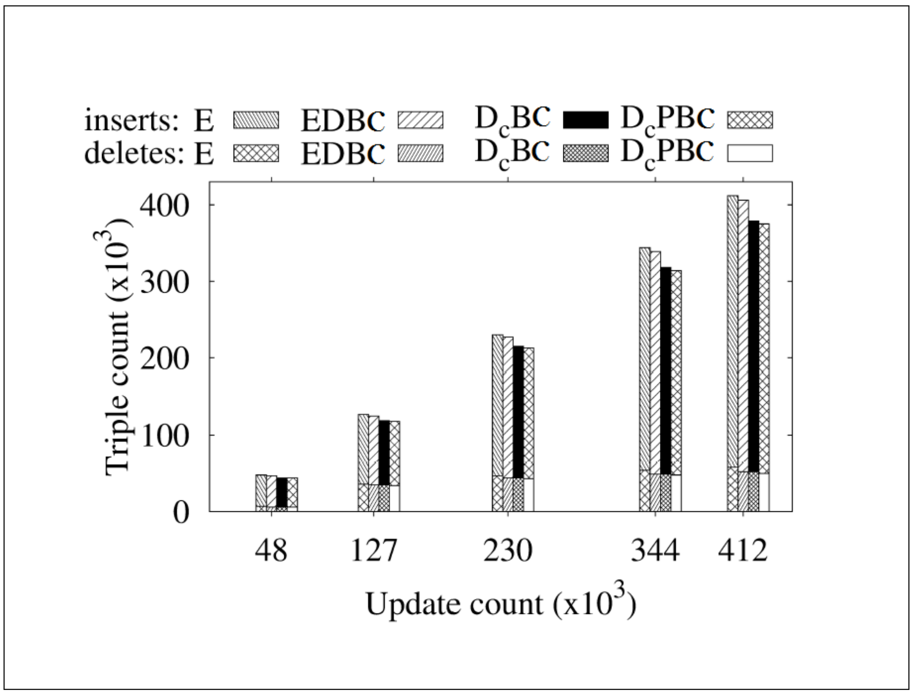


Fig. 5.10: Delta size

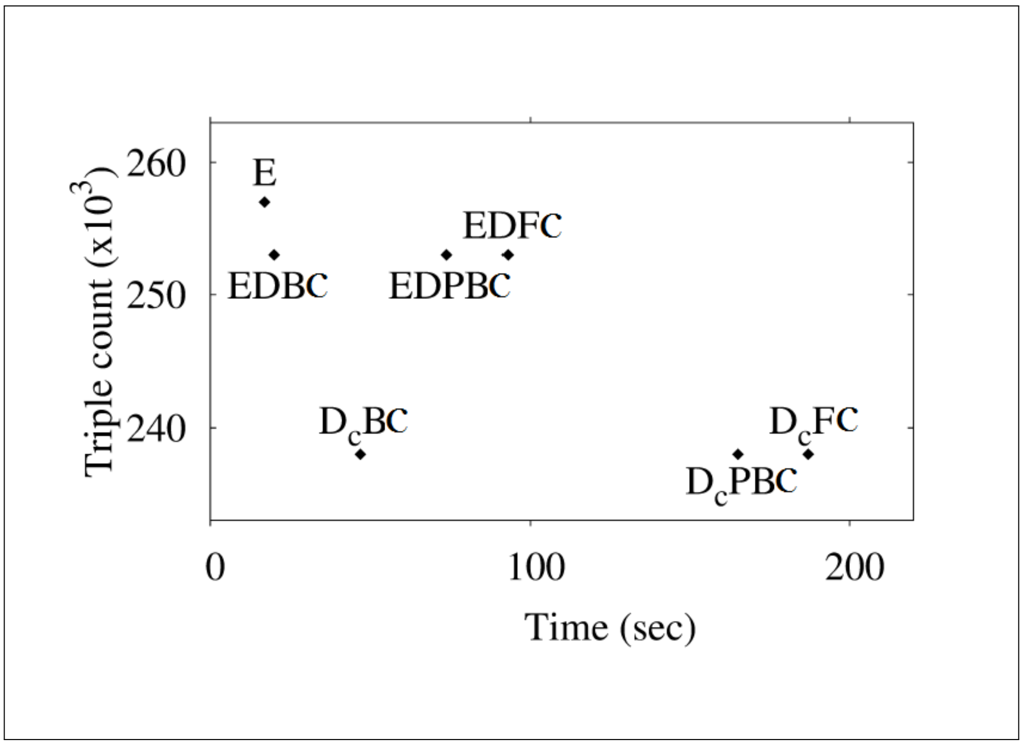


Fig. 5.11: Comparison of delta approaches.

In Figure 5.7 it can be seen that of the deltas evaluated in these experiments, EDBC and the pruned version of the same approach can be generated with the lowest inference time. This is a consequence of both the efficiency of backward-chaining and the application of inference only to the delete set. At the other end of the spectrum, forward-chaining methods are slower, as a consequence of the time needed to produce the closure for both models. Forward-chaining is expensive but becomes useful where models are being queried. However since the focus of this work is updating models, backward-chaining is a more appropriate approach where only a limited number of triples are inferred (i.e. triples in the delta) and not the full closure as the case in the forward-chaining approach.

Pruning generally helps to further reduce the inference time however the process adds further expense. Figure 5.8 shows the reasoning time (i.e the time taken up by both inferencing and pruning). This indicates that for the data structure used, the time required to carry out pruning exceeds the inference time both for ΔD_c and ΔED. This is consistent with previous findings [AAW15]. The overall delta time shown in Figure 5.9 indicates that taking account of set difference operations, inferencing and pruning, approaches that prune the delta set tend to require significantly more processing power than non-pruning approaches. Overall, the ΔE is the fastest process since no pruning or inferencing is carried out. The delta sizes shown in Figure 5.10 indicate that applying inference on this data set reduces the updates that need to be executed, particularly when it is applied to both the insert and delete sets.

The relationship between Figures 5.9 and 5.10 is summarised in Figure 5.11, which is based on the average delta size and average generation time for all the data models. Figure 5.11 shows the interaction between the degree of inference (i.e. the delete set and/or the insert set or no inference at all) and the approach to inferencing (i.e. inferring all triples or only necessary triples) and their impact on the delta size and the delta computation time. It can be seen that ΔD_c has the smallest delta size compared to ΔED and ΔE. It can also be seen that the approach to inferencing affects the delta computation time. Figure 5.11 indicates that the combination of ΔD_c and backward-chaining approach is more efficient (i.e smaller delta size and faster generation) than the other methods tested. Overall, Figure 5.11 shows that the computation time increases in the sequence of explicit, backward chaining, pruned backward chaining, forward chaining whereas the delta size increases in the sequence ΔD_c, ΔED, ΔE.

The overall effect of these results is to indicate that ΔD_c provides a viable route to limiting the data that would need to be transferred from a server to a client in order to update copies of an RDF data store. Pruning may assist this process but comes at a cost of additional processing time, which may be unacceptable in a peer-to-peer context or where updates need to be generated on demand.

5.4 Pruning the OWL 2 ruleset

The hypothesis being evaluated in this section of the work is that OWL 2 RL/RDF rules can be used to reduce the size of updates to OWL 2 ontologies. The process of pruning rules used in ontology updates with OWL 2 RF/RDF rules that is explained in Section 4.4 has been evaluated experimentally using the Lehigh University Benchmark (LUBM) [GPH05] and the University Ontology Benchmark (UOBM) [MYQ+06]. These Semantic Web benchmarks allow the generation of datasets of different sizes. LUBM facilitates the evaluation of Semantic Web tools and is accepted as a standard evaluation platform for OWL ontology systems. Despite this, it does not fully support the inference of either OWL lite or OWL DL profiles of OWL 2. For example, inferencing the *allValuesFrom* restrictions and the cardinality constraints cannot be tested using LUBM datasets. Furthermore, the generated instance data lacks inter-linkage between isolated subgraphs. In this context, instance data can be generated to represent individuals for a number of universities but individuals in one university do not have relations with individuals from other universities. This limits the benchmark's value for scalability tests as inference on connected subgraphs is harder than that on isolated subgraphs. As a consequence of this, LUBM is weaker in measuring the capability of inference engines as it does not trigger all the inference rules supported by these engines.

For these reasons, UOBM was developed to extend LUBM and overcome its limitations with full support for both OWL lite and OWL DL as well as the generation of more complex instance datasets by establishing links between individuals from different universities.

In the experimental work reported here both LUBM and UOBM benchmark generators were used to produce three versions nominally of 1000, 10,000 and 100,000 triples respectively. Three change ratios on each of these different sizes of datasets were produced. This involved changing the subsumption hierarchy as well as the addition of inferable triples. These inferable triples were obtained by materializing ontology versions and selecting a number of the inferred triples to be added to the corresponding dataset. The change ratio is defined as the size of explicit differences between two versions divided by the size of the original version. Using this manipulation, four versions for each size of the datasets were generated: the original version; 5% change ratio version; 10% change ratio version and 15% change ratio version. Table 5.4 represents the feature of the different versions generated using both LUBM and UOBM benchmarks.

	LUBM				UOBM			
Nominal size	**Original size**	**∼5%**	**∼10%**	**∼15%**	**Original size**	**∼5%**	**∼10%**	**∼15%**
1000	1,391	1,380	1,418	1,445	965	970	962	967
10,000	10,149	10,348	10,553	10,853	10,097	9,956	10,696	10,729
100,000	100,448	102,165	109,377	113,002	101,133	101,894	103,354	107,703

Tab. 5.4: Triple count in the LUBM and UOBM ontologies used for evaluation.

The triple store was implemented in MySQL to handle the RDF collections and the deltas. Each predicate was represented in a separate table. Indexing was excluded to preserve the validity of the use-case i.e. delta that may need to be generated on demand without allowing time for index production. The triple store was loaded and updates were validated using the Jena framework.

Computation of the syntactic differences between successive ontology versions starts with the generation of ΔE. This step takes into account non-blank node triples (i.e. triples that do not contain blank identifiers in any position). After the calculation of the explicit difference between the two versions and the blank node matching, these differences enter a reduction phase where reasoning under the semantics of OWL 2 RL/RDF is employed for the purpose of minimizing unnecessary change operations (i.e. insertions or deletions).

In addition to the change detection approaches explained in Section 4.3; the explicit delta ΔE, the explicit dense delta ΔED and the correct dense delta ΔD_c, two pruning-based approaches as proposed in [ILK13] are also employed; pruned explicit dense delta and pruned correct dense delta, ΔPED and ΔPD_c, respectively. These approaches combine the change detection approaches in [ZTC11] with pruning methods to reduce unnecessary computation during the reasoning process. The inference process for the production of these deltas are carried out using the backward-chaining strategy.

	Delta sizes		
change	$\|\Delta E\|$	$\|\Delta ED\|$	$\|\Delta D\|$
$\sim$ 5%	5,001	4,398	3,534
$\sim$ 10%	10,001	9,154	7,457
$\sim$ 15%	15,000	7,457	11,165

Tab. 5.5: Delta sizes in the 10,000 triple structure in UOBM triple sets using the different change detection techniques.

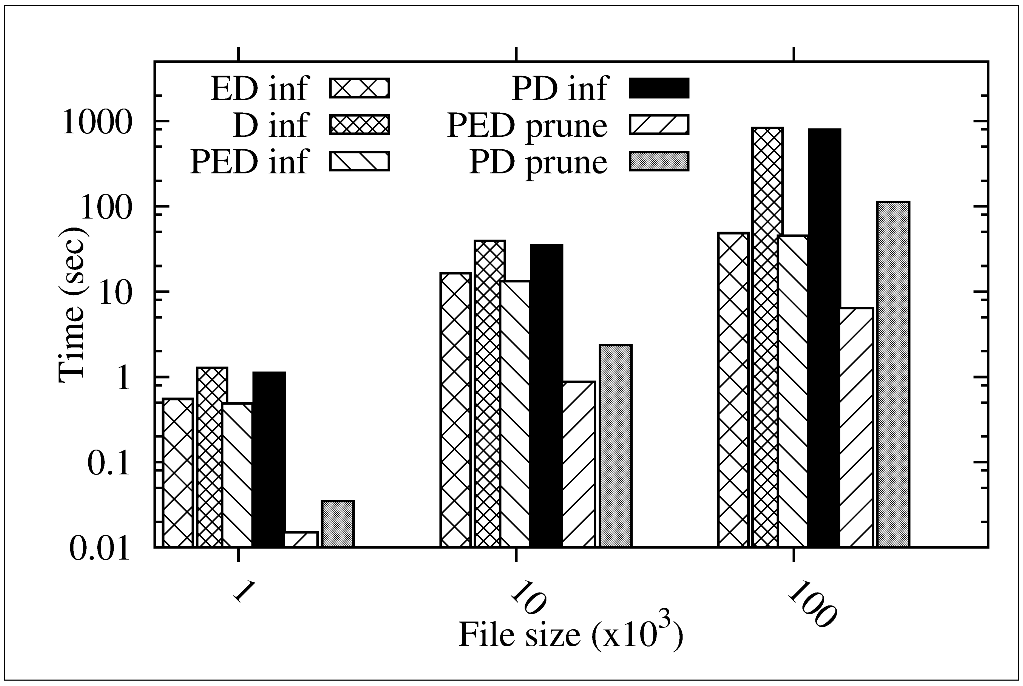


Fig. 5.12: Reasoning time for 10% updates LUBM data set with no blank node support

Updates were calculated for each of the sample datasets and indicate that the inference load for ΔD_c exceeded that for ΔED in consequence of the latter approach only carrying out inference over the delete set (Figure 5.12). Similarly, Figure 5.12 shows that the process of pruning rules in the ΔD_c approach is more costly than pruning rules for ΔED because the former, being a larger set, presents more pruning opportunities.

The distinction between the UOBM and LUBM benchmarks is evident from Figure 5.13. It can be seen here that UOBM data triggers the execution of more rules than the simpler LUBM set. Both data sets benefit from the reduction in rule operation that is supported by the pruning process described in Section 4.4 although the benefit is more pronounced in LUBM. The pruning method reduced $\sim$17% of unnecessary rules in LUBM, while this method reduced $\sim$4% of unnecessary rules in UOBM. This, in turn indicates that the benefits produced by pruning rules are influenced by the data distribution within a particular dataset. Benchmark data can be deficient in this respect because the distribution of values may not reflect real world data very accurately. This result also indicates that the UOBM set presents more of a challenge to the inference process because of its richer structure.

The variation of inferencing in LUBM and UOBM for the 100,000 triple set is shown in Figure 5.14. In both change detection approaches ΔED and ΔD_c, the inference time in the UOBM dataset is higher than that in the LUBM by $\sim$85% and $\sim$36% respectively using these two approaches.

Delta sizes for the 100,000 triple sets with different change ratios in UOBM are shown in Table 5.5. These deltas are calculated using different change detection approaches: the no-inference approach ΔE, the explicit dense delta ΔED and the correct dense delta ΔD_c. The table shows a reduction of the delta as a result of inferencing over the set of deletes (i.e. ΔED) and this reduction is increased further by inferencing over both sets of updates; deletes and inserts (i.e. ΔD_c). All these deltas are tested for their soundness and they correctly updated the models.

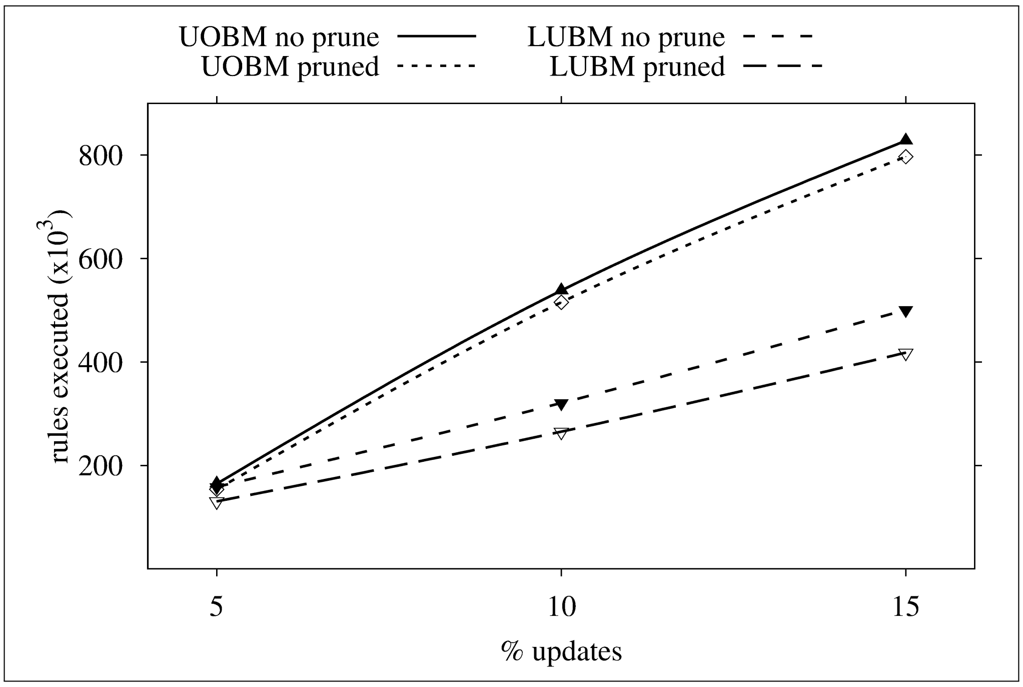


Fig. 5.13: Reduction in rules assessed as a consequence of pruning in the 100,000 triple structure

The impact of blank node reduction in LUBM is shown in Figure 5.15. This process saves additional triples in the delta, which has consequences for the performance time of inferencing. Where blank node matching is unsupported, the cost of inferencing is ~35% higher, using ΔD_c, than with the support of blank node matching. This Figure also shows the performance of rule pruning in LUBM using the two change detection approaches; ΔED and ΔD_c. The time for pruning rules using ΔD_c is ~90% higher than the pruning time using ΔED. This very significant difference in pruning time between the two approaches is because ΔD_c applies pruning in both sets of updates; the delete set and the insert set, while ΔED considers the delete set only.

Overall the results indicate that both rule pruning and blank node matching have the potential for reducing the processing required for generating compact deltas.

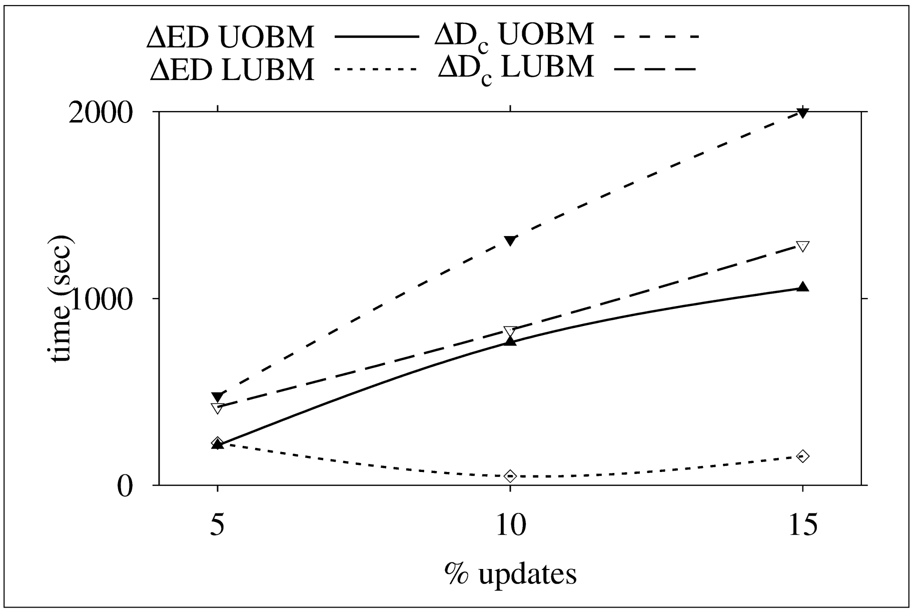


Fig. 5.14: Inference time for ΔED and ΔD_c in the 100,000 triple for both LUBM and UOBM

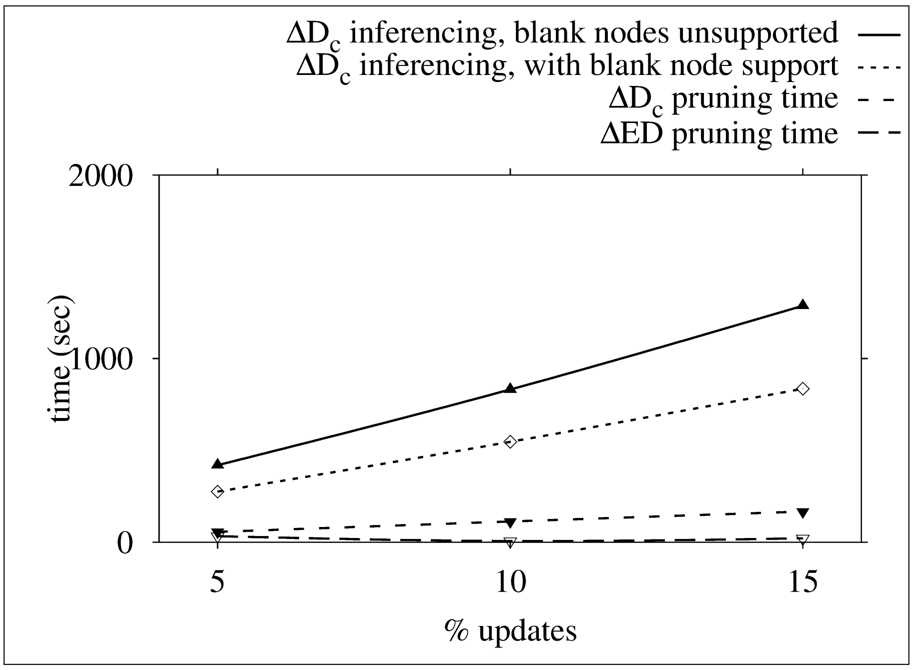


Fig. 5.15: Performance time for 100,000 triple set in LUBM with and without support for blank nodes

5.5 General discussion

Updating RDF is an important problem and as the understanding of the update process grows, new approaches for its optimization also emerge. The work presented in this thesis contributes to this growing understanding by presenting an in depth analysis of the update process. Based on our experimental results, we found that pruning RDF has a significant effect on reducing the inference time. However, pruning the RDF requires additional time for applying the pruning rules, and this time increases as the size of the structural differences between the RDF versions increases. This time is about 20-90 times higher than the inference time (Appendix D). Therefore, since the pruning method is applied prior to computing the closure using the backward-chaining strategy, the time for pruning the RDF needs to be counted for measuring the inference and the overall performance of the backward-chaining change detection approach. Regardless of the time taken to prune triples, the inference process has a clear effect in reducing unnecessary updates. Extending this process to reduce updates that involve inserting a triple and at the same time producing a consistent update can further minimize the size of the delta that needs to be synchronized to other remote devices.

In this chapter we experimentally evaluated a correction method for dense deltas that results in consistent update of RDF datasets. We have eliminated the need for conditions on the dataset by checking the antecedents of inferable triples in the insert set. If at least one such antecedent is found in the delete set then the inferable triple in the insert set cannot be removed from the delta. Otherwise, this triple can be safely removed from the delta to minimize its size.

A summary of our results is shown in Figure 5.11, which characterises the interaction between the degree of inference (i.e. applying inference on the delete set and/or on the insert set or no inference at all) and the approach to inferencing (i.e. inferring all triples or only necessary triples) and their combined impact on the delta size and computation time. It can be seen that ΔD_c has the smallest delta size compared to ΔED and ΔE. It can also be seen that the approach to inferencing affects the delta computation time. Figure 5.11 indicates that backward-chaining is more efficient (i.e smaller delta size and faster generation) than the other methods tested.

In this chapter we have investigated the effect of inference degree and inference approach on both the delta computation time and storage space over RDF datasets. Also, it is worth exploring different inference strengths to further evaluate the delta sizes and performance of the different approaches to producing these deltas. Inference strength is defined in [ZTC11, p 14:20] as the difference between the full closure of a model and the original model divided by the original model (i.e. $inference\ strength = \frac{|C(M)|-|M|}{|M|}$). In particular while backward-chaining may be efficient, combining it with pruning may be expensive in terms of computation time where data is characterised by large inference strengths.

By contrast with inference strength, reduction strength shown in Table 5.2 indicates when inference is capable of making a difference to the size of the produced delta. When the inference strength is zero, there are no inferences to be made and the model is closed. Under these circumstances, $|\Delta E| = |\Delta D_c|$. However, $|\Delta E|$ may still be equal to $|\Delta D_c|$ when the inference strength is greater than zero. This occurs when, for example, none of the triples in the delta are inferable in M.

Example 3. *Let* $M = \{w\ \textit{subClassOf}\ x, x\ \textit{subClassOf}\ y, y\ \textit{subClassOf}\ z\}$ *and* $M' = \{w\ \textit{subClassOf}\ x, x\ \textit{subClassOf}\ y, y\ \textit{subClassOf}\ z, n\ \textit{subClassOf}\ r\}$*. Under these circumstances,* $\Delta E = \{ins\{n\ \textit{subClassOf}\ r\}\}$ *and since this triple can not be inferred in M,* $\Delta D_c = \{ins\{n\ \textit{subClassOf}\ r\}\}$*.*

The inference strength has a value of 1 but $|\Delta E| = |\Delta D_c|$ i.e. the inference strength is significantly different from zero but there are no inferred triples. This contrasts with the definition provided by [ZTC11, p 14:20], which states that inference strength is proportional to the count of reduced triples. Alternatively, the reduction strength in this example is zero, thereby providing an effective guide to indicate when $|\Delta E| = |\Delta D_c|$, which is not clearly shown by the inference strength.

Both inference strength and reduction strength also give an indication of the work load of pruning. High values for these parameters indicate that a large number of triples can be inferred. However, adding such inferable triples provides a large collection of data that needs to be checked for possible pruning before inference can take place.

Example 4. *Let* $M = \{w\ subClassOf\ x, x\ subClassOf\ y, y\ subClassOf\ z\}$ *and* $M' = \{w\ subClassOf\ x, x\ subClassOf\ y, y\ subClassOf\ z, n\ subClassOf\ r,$ $w\ subClassOf\ z, w\ subClassOf\ y, x\ subClassOf\ z\}$*.*
Here, $\Delta E = \{ins\{n\ subClassOf\ r\}, ins\{w\ subClassOf\ z\}, ins\{w\ subClassOf\ y\},$ $ins\{x\ subClassOf\ z\}\}$*.*

Pruning this list will involve checking every entry to ensure that the subject or object does not occur in M in order to prune that triple from the list to be entered into the inference process. Of the four triples added in this example, all must be checked for pruning but only one triple ($ins\{n\ subClassOf\ r\}$) will be removed before the remaining three triples will enter the inference process.

In general terms, reduction strength appears to be a better indication of the differences between ΔE and ΔD_c than inference strength. Similar arguments apply to establishing the difference between ΔE and ΔED.

Reduction of the delta size could be larger when the number of inferred triples in the delta between two datasets is large. For example, comparing a dataset with its materialized (i.e full closure) version results in a large reduction of the delta, which means that the difference between ΔE and ΔD_c is big.

Since inference has a clear effect on reducing the delta size, exploiting inferred triples may provide further improvements in update performance, thus, similar methods can be applied to ontologies that are represented in OWL 2. Here the RL ruleset [MGH+13], as a subset of OWL 2, is much richer than the ruleset for RDFS with consequent potential for benefits to delta generation performance and size.

However, the rich ruleset of complex ontology languages, like OWL 2, may provide a challenge to change detection techniques. In particular, the repeated application of a large ruleset that may be necessary to produce the desired conclusion can result in performance problems. Blindly applying rules will result in many such applications being void as a result of consequents that can not contribute to the desired outcome. Advance knowledge of which rules are applicable and which are not is very important in avoiding their unnecessary application. This thesis proposes a change detection technique using backward-chaining inference. It produces a small delta (Table 5.5) using a pruning method that eliminates unnecessary inference rules during the reduction of the delta size (Figure 5.14).

A further reduction in delta size is possible through the blank node matching method. This method matches chains of blank nodes between ontology versions. Excluding matched blank nodes from the delta is beneficial in reducing the delta size and hence the network bandwidth when synchronizing ontology versions as well as the storage overhead for deltas (Figure 5.13). .

5.6 Experimental limitations

All experiments were carried out using benchmarks datasets or databases that have been manipulated. In real world data collections we may get different results since value distribution in artificial data is often different form the real world. Performance has been assessed by relatively small data collections but we may get different results when bigger data collections are used. In our work, the evaluation of the different change detection techniques and the proposed methods involve time measurements which were carried out by controlling garbage collection and Just in Time (JIT) compilation in order to achieve results with improved accuracy.

The problem for code-execution time measurement is that Java implementations have a complex performance life cycle. This life cycle involves a warm-up phase which has little to do with the actual code and more to do with Java Virtual Machine (JVM) behavior. In this phase JVM typically loads classes it uses which involves disk I/O, initializations, parsing and verification. This phase usually takes place in the first execution of the code; however, some JVMs continue gathering profiling information in the next few executions of the code. Consequently, the initial performance is relatively slow which means that multiple iterations of the code within a single execution of the virtual machine are required to eliminate or exclude the warm-up time and to reach the steady state application performance which is achieved when the Just-In-Time (JIT) compilation occurs.

However, even when the application reaches the steady state, the Java Virtual Machine (JVM) behavior involves other concerns that can affect the measurement of the performance. One of these concerns is the automatic memory management (Garbage collection). Garbage collection (GC) is a VM feature that attempts to reclaim memory occupied by objects that are no longer in use by the program. GC is undetermined and can occur at any time during the execution of the program depending on the task itself and on the status of the underlying resources and this could have a cost inside each measurement. However, one thing that could be done is to clean up the VM by calling the garbage collection method System.gc() before the start of any measurement. This would ensure that the VM is free from objects created by other code in the same application. In addition, ensuring sufficient memory is available by increasing the VM heap size (-Xmx) could eliminate the GC calls by the VM.

As the control of the JIT in our experiments was handled by executing multiple iterations of the code with a single run of the virtual machine, this process consumes more processing power as the size of the RDF collection increases. In our final experiments using OWL 2 RL ruleset, JIT was not controlled due to the intensive consumption of processing power as a result of the complexity of the ruleset.

5.7 Summary

The work presented in this chapter initially draws attention to the performance penalty of updating RDF data collections. The first finding is that pruning triples from the inference process required for the reduction of the delta helps with this process but contributes significantly to the overall process and cost. The second contribution is the proposed method for correcting dense delta, ΔD_c. Our method results in the smallest correct delta for transformation between different versions. The third contribution of the work is a method that we proposed for pruning rules in the context of updating OWL 2 ontologies. The application of this method results in a reduction of the number of rules applied during the production of the delta in both datasets; LUBM and UOBM. In addition to rule pruning the third contribution involves a simple method for blank node matching which is used to avoid incorporating such content into the differences that are detected between these versions.

6. CONCLUSIONS AND FUTURE WORK

The work presented in this thesis explores approaches for reducing the costs of updates on RDF data sets. The initial experiments involved comparing two different update approaches. In the first approach, both fixed and variable numbers of triples were selected randomly and updates were performed against the in-memory view of the RDF structures. This establishes the basic update time. While in the second approach, specific triples are selected to be updated. The specific approach requires finding the triples before updating them. The time measured in these experiments includes both the search and the update time. Searching for the triples is done by using SPARQL queries while updating them is done by using SPARUL queries. Results from the experiment show that the random approach is faster than the specific approach. However, in the random approach no searching is involved in the update process as is the case in the specific approach. This work includes changes to the instance level of RDF and does not exploit the semantic content of OWL.

The number of updates required for updating RDF collections can be reduced by inferencing over the data to see if a particular update is really necessary. Inferencing itself can be reduced by pruning triples from the inferencing process if there is no prospect of them taking part. Both inferencing and pruning are costly processes in terms of performance time.

This thesis proposes a correction method for dense deltas that results in consistent update of RDF datasets. This works by checking the antecedents of inferable triples in the insert set. If at least one such antecedent is found in the delete set then the inferable triple in the insert set cannot be removed from the delta. Otherwise, this triple can be safely removed from the delta to minimize its size.

This work investigated the effect of inference degree and inference approach on both the delta computation time and storage space over RDF datasets. The results suggest that while backward inference may be efficient, combining it with pruning may be expensive in terms of computation time where data is characterised by large inference strengths. Exploiting the inferred triples to infer new information may provide further improvements in update performance.

These experiments indicate that the semantics of RDF can be exploited in order to reduce the differences between RDF versions. However, change detection techniques that rely on powerful rules defined by ontology languages, such as OWL, may be complicated by applying a large number of rules and by the repeated application of these rules in order to get the desired conclusion from the underlying dataset. Knowing in advance which rules are applicable and which are not is very important to avoid unnecessary application of rules. Experimental work in this thesis evaluates a pruning method that prunes unnecessary inference rules for the reduction of delta size. Experimental results on two datasets; LUBM and UOBM show that both datasets benefit from pruning rules. Rule reduction in LUBM is more pronounced and this is a consequence of the fact that this dataset is not enriched with OWL 2 constructs when compared to UOBM, and thus more irrelevant rules can be pruned.

To further reduce the delta size, the thesis explores a blank node matching method. This method matches chains of blank nodes between RDF versions. Excluding matched blank nodes from the delta is beneficial in reducing the delta size and hence the network bandwidth when synchronizing RDF versions as well as the storage overhead for deltas.

6.1 Overall conclusions

The research questions that drove this thesis were:

RQ1: What are the costs of different approaches for generating updates that need to be carried out on demand in a distributed environment?

RQ2: What is an efficient way of minimizing delta sizes using RDFS rules so that the principles can be applied to more complex rule sets?

RQ3: How can we exploit the complexity of the OWL 2 rule set for the reduction of delta size?

These research questions were motivated by the fact that reducing the delta size by exploiting the semantics hidden in RDF data is important for reducing the storage required to hold these updates and also for reducing the bandwidth required to transfer these updates through a network. In this thesis we answered these research questions by a number of experiments that tackle specific issues of this problem. RQ1 is addressed by establishing a starting point for understanding and analyzing the costs and benefits of the different change detection techniques. Knowing that the update is expensive, in these experiments we investigate the cost of pruning with respect to the cost of inference. We also evaluate the costs and benefits of the different change detection techniques.

The results for these experiments showed how implicit knowledge can be exploited to reduce the number of deletions in the updates. RQ2 is addressed by expanding this process to involve the insertion set of updates that can further reduce the update size. However, a naïve approach to achieve this may produce incorrect updates that lead to an inconsistent ontology. The focus for this experiment was to produce a small collection of updates with correctness of these updates in mind. This work proposed a technique to reduce the update size considering both sets; insertions and deletions, and to produce a correct dense delta.

The inference process can be empowered by applying more complex ontology languages that can exploit more implicit knowledge and derive new knowledge which may participate in the reduction of the update size. This work addresses RQ3. The application of the OWL 2 RL/RDF rule set in this part of our work has revealed the complexity of the application of these rules in the inference process which results from the overlapped rules. This problem led to our contribution of a technique to prune unnecessary rules during the inference process and thus only rules that may derive the desired conclusion are applied. Moreover, blank nodes are considered in this part of our work and a matching technique based on tracing blank nodes chains is proposed to further reduce the delta size rather than considering all blank nodes as differences between the models.

Overall the answers to these research questions indicate that the original hypothesis that " improving RDF change detection techniques by exploiting both sets of updates (i.e. insertions and deletions) while retaining correctness when transforming one knowledge base (KB) into another will support reduction in the size of updates." is a valid statement.

6.2 Contribution

The contribution of this thesis is to provide an in depth analysis of the strengths and weaknesses of different RDF update strategies that use semantic content to generate these updates. As a result of this analysis this work proposed an algorithm for a change detection tool ΔD_c which is able to produce a small delta that can correctly transform one RDF data set into another. We proposed an approach for generating the correct dense delta that will maintain the consistency of RDF data sets. Finally, this thesis contributes to the problem of ontology evolution by using the extended rule set of OWL 2 RL/RDF. In this contribution we proposed a pruning method for this complex rule set. We also proposed a simple blank node matching technique for matching blank node chains between RDF versions.

6.3 Future work

The main direction of future work is to improve the performance of our change detection technique which may be achieved by distributing the execution of rules in the OR tree between a number of machines instead of a single machine. Moreover, more complicated rules could be considered to investigate their effect on delta reduction. An example of these rules are the rules that exploit the owl:sameAs relation which have been excluded from the current work due to their execution complexity.

The work presented in this thesis indicates ways in which delta reduction can be carried out. There may be circumstances in real world data collections when delta reduction is unnecessary and bandwidth would be sufficient to transfer the unreduced delta. These circumstances are yet to be explored.

BIBLIOGRAPHY

[AAW15] S. M. M. Al Azwari and J.N. Wilson. The cost of reasoning with RDF updates. In *Proc 9th IEEE ICSC*, pages 328–331, 2015.

[Aue04] Sören Auer. A web based platform for collaborative ontology management. In *Proc. International Semantic Web Conference (ISWC)*, 2004.

[AVH04] Grigoris Antoniou and Frank Van Harmelen. Web ontology language: Owl. In *Handbook on ontologies*, pages 67–92. Springer, 2004.

[B$^+$90] Brian Berliner et al. Cvs ii: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, volume 341, page 352, 1990.

[BG04] Dan Brickley and Ramanathan V Guha. {RDF vocabulary description language 1.0: RDF schema}. 2004.

[BK03] Jeen Broekstra and Arjohn Kampman. Inferencing and truth maintenance in rdf schema. *PSSS*, 2003.

[BL$^+$00] Tim Berners-Lee et al. Cwm: A general purpose data processor for the semantic web. *URL http://www. w3. org/2000/10/swap/doc/cwm. html*, 2000.

[BLC04] Tim Berners-Lee and Dan Connolly. Delta: an ontology for the distribution of differences between rdf graphs, 2004.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web: Scientific American. *Scientific American*, May 2001.

[BZC10] Aurélien Bénel, Chao Zhou, and Jean-Pierre Cahier. Beyond web 2.0âĂę and beyond the semantic web. In *From CSCW to Web 2.0: European Developments in Collaborative Design*, pages 155–171. Springer, 2010.

[CDD+04] Jeremy J Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83. ACM, 2004.

[CFF+98] Vinay K Chaudhri, Adam Farquhar, Richard Fikes, Peter D Karp, and James P Rice. Okbc: A programmatic foundation for knowledge base interoperability. In *AAAI/IAAI*, pages 600–607, 1998.

[Cro09] John J Cronin. Upgrading to web 2.0: An experiential project to build a marketing wiki. *Journal of Marketing Education*, 2009.

[Dev06] Vladan Devedzic. Introduction to the semantic web. In *Semantic Web and Education*, pages 29–69. Springer-Verlag, 2006.

[ELBB$^+$04] Jerome Euzenat, Thanh Le Bach, J Barrasa, P Bouquet, J De Bo, R Dieng-Kuntz, M Ehrig, M Hauswirth, Mustafa Jarrar, R Lara, et al. State of the art on ontology alignment. *Knowledge Web Deliverable D*, 2:2–3, 2004.

[FFST11] Dieter Fensel, Federico Michele Facca, Elena Simperl, and Ioan Toma. Semantic web. In *Semantic Web Services*, pages 87–104. Springer, 2011.

[Flo07] Giorgos Flouris. On the evolution of ontological signatures. In *Proceedings of the workshop on ontology evolution*, pages 67–72, 2007.

[FMK$^+$08] Giorgos Flouris, Dimitris Manakanatas, Haridimos Kondylakis, Dimitris Plexousakis, and Grigoris Antoniou. Ontology change: Classification and survey. *The Knowledge Engineering Review*, 23(02):117–152, 2008.

[FP05] Giorgos Flouris and Dimitris Plexousakis. Handling ontology change: Survey and proposal for a future research direction. *Institute of Computer Science, FORTH., Greece, Technical Report TR-362 FORTH-ICS*, 2005.

[FPA06] Giorgos Flouris, Dimitris Plexousakis, and Grigoris Antoniou. Evolving ontology evolution. In *SOFSEM 2006: Theory and Practice of Computer Science*, pages 14–29. Springer, 2006.

[Gär03] Peter Gärdenfors. *Belief revision*, volume 29. Cambridge University Press, 2003.

[GGD13] Christine Golbreich, Julien Grosjean, and Stefan Jacques Darmoni. The foundational model of anatomy in owl 2 and its use. *Artificial intelligence in medicine*, 57(2):119–132, 2013.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining views incrementally. *ACM SIGMOD Record*, 22(2):157–166, 1993.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[Gru93] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[HD09] Aidan Hogan and Stefan Decker. On the ostensibly silent ’w’ in owl 2 rl. In *Web Reasoning and Rule Systems*, pages 118–134. Springer, 2009.

[Hen01] J. Hendler. Agents and the semantic web. *Intelligent Systems, IEEE*, 16(2):30 – 37, mar-apr 2001.

[Hen11] James Hendler. The semantic web 10th year update. In *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*, WIMS '11, pages 1:1–1:3, New York, NY, USA, 2011. ACM.

[HHL99] Jeff Heflin, James Hendler, and Sean Luke. Shoe: A knowledge representation language for internet applications. 1999.

[HM01] Volker Haarslev and Ralf Müller. Racer system description. In *Automated Reasoning*, pages 701–705. Springer, 2001.

[HM04] Patrick Hayes and Brian McBride. Rdf semantics, 2004.

[HPS14] Patrick J Hayes and Peter F Patel-Schneider. Rdf 1.1 semantics. *W3C Recommendation, Online at http://www. w3. org/TR/rdf11-mt*, 2014.

[HQ07] Peter Haase and Guilin Qi. An analysis of approaches to resolving inconsistencies in dl-based ontologies. In *Proceedings of International Workshop on Ontology Dynamics (IWODâĂŹ07)*, pages 97–109, 2007.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[HS04] Peter Haase and York Sure. State-of-the-art on ontology evolution. 2004.

[ILK12] Dong-Hyuk Im, Sang-Won Lee, and Hyoung-Joo Kim. A version management framework for rdf triple stores. *International Journal of Software Engineering and Knowledge Engineering*, 22(01):85–106, 2012.

[ILK13] Dong-Hyuk Im, Sang-Won Lee, and Hyoung-Joo Kim. Backward inference and pruning for rdf change detection using rdbms. *Journal of Information Science*, 39(2):238–255, 2013.

[KC06] Graham Klyne and Jeremy J Carroll. Resource description framework (rdf): Concepts and abstract syntax. 2006.

[KFKO02] Michel Klein, Dieter Fensel, Atanas Kiryakov, and Damyan Ognyanov. Ontology versioning and change detection on the web. In *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, pages 197–212. Springer, 2002.

[KPS⁺06] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, Bernardo Cuenca Grau, and James Hendler. Swoop: A web ontology editing browser. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(2):144–153, 2006.

[Krö12] Markus Krötzsch. The not-so-easy task of computing class subsumptions in owl rl. In *The Semantic Web–ISWC 2012*, pages 279–294. Springer, 2012.

[KS03] Yannis Kalfoglou and Marco Schorlemmer. Ontology mapping: the state of the art. *The knowledge engineering review*, 18(01):1–31, 2003.

[KWE10] Vladimir Kolovski, Zhe Wu, and George Eadon. Optimizing enterprise-scale owl 2 rl reasoning in a relational database system. In *The Semantic Web–ISWC 2010*, pages 436–452. Springer, 2010.

[LM15] Yu Liu and Peter McBrien. Transactional and incremental type inference from data updates. In *Data Science*, pages 206–219. Springer, 2015.

[LRVS09] Uta Lösch, Sebastian Rudolph, Denny Vrandečić, and Rudi Studer. Tempus fugit. In *The Semantic Web: Research and Applications*, pages 278–292. Springer, 2009.

[M$^+$81] Brendan D McKay et al. *Practical graph isomorphism.* Department of Computer Science, Vanderbilt University, 1981.

[MFHS02] Deborah L McGuinness, Richard Fikes, James Hendler, and Lynn Andrea Stein. Daml+ oil: an ontology language for the semantic web. *Intelligent Systems, IEEE*, 17(5):72–80, 2002.

[MGH$^+$09] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. Owl 2 web ontology language: Profiles. *W3C recommendation*, 27:61, 2009.

[MGH$^+$13] Boris Motik, B Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 Web ontology language profiles, W3C recommendation 11, 2013.

[MSR02] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three implementations of squishql, a simple rdf query language. In *The Semantic WebâĂŤISWC 2002*, pages 423–435. Springer, 2002.

[Mun57] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[MYQ+06] Li Ma, Yang Yang, Zhaoming Qiu, Guotong Xie, Yue Pan, and Shengping Liu. *Towards a complete OWL ontology benchmark*. Springer, 2006.

[NM02] Natalya Fridman Noy and Mark A Musen. Promptdiff: A fixed-point algorithm for comparing ontology versions. *AAAI/IAAI*, 2002:744–750, 2002.

[NM04] Natalya F Noy and Mark A Musen. Ontology versioning in an ontology management framework. *Intelligent Systems, IEEE*, 19(4):6–13, 2004.

[NW10] Thomas Neumann and Gerhard Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases. *Proceedings of the VLDB Endowment*, 3(1-2):256–263, 2010.

[PDT05] Peter Plessers and Olga De Troyer. Ontology change detection using a version log. In *The Semantic Web–ISWC 2005*, pages 578–592. Springer, 2005.

[PFF+13] Vicky Papavasileiou, Giorgos Flouris, Irini Fundulaki, Dimitris Kotzinos, and Vassilis Christophides. High-level change detection in rdf (s) kbs. *ACM Transactions on Database Systems (TODS)*, 38(1):1, 2013.

[PSHH+04] Peter F Patel-Schneider, Patrick Hayes, Ian Horrocks, et al. Owl web ontology language semantics and abstract syntax. *W3C recommendation*, 10, 2004.

[Sch09] Michael Schneider. Owl 2 web ontology language: Rdf-based semantics. *W3C Recommendation (October 27 2009)*, 2009.

[SE05] Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. In *Journal on data semantics IV*, pages 146–171. Springer, 2005.

[SGS+06] Pavel Shvaiko, Fausto Giunchiglia, Marco Schorlemmer, Fiona McNeill, Alan Bundy, Maurizio Marchese, Mikalai Yatskevich, Ilya Zaihrayeu, Bo Ho, Vanessa Lopez, et al. Dynamic ontology matching: a survey. *Techn. Report DIT-06-046, University of Trento, Available on: http://eprints. biblio. unitn. it/archive/00001040*, 2006.

[SHF12] Jordy Sangers, Frederik Hogenboom, and Flavius Frasincar. Event-driven ontology updating. In *Web Information Systems Engineering-WISE 2012*, pages 44–57. Springer, 2012.

[SM09] Michael Schneider and Kai Mainzer. A conformance test suite for the owl 2 rl/rdf rules language and the owl 2 rdf-based semantics. In *OWLED*, 2009.

[SMH08] Rob Shearer, Boris Motik, and Ian Horrocks. Hermit: A highly-efficient owl reasoner. In *OWLED*, volume 432, page 91, 2008.

[SMMS02] Ljiljana Stojanovic, Alexander Maedche, Boris Motik, and Nenad Stojanovic. User-driven ontology evolution management. In *Knowledge engineering and knowledge management: ontologies and the semantic web*, pages 285–300. Springer, 2002.

[SPG$^+$07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[SQ06] Wennan Shen and Yuzhong Qu. An rdf storage and query framework with flexible inference strategy. In *Frontiers of WWW Research and Development-APWeb 2006*, pages 166–175. Springer, 2006.

[TH05] Dmitry Tsarkov and Ian Horrocks. Fact++. *School of Computer Science, University of Manchester, Recuperado el*, 28, 2005.

[TLZ12] Yannis Tzitzikas, Christina Lantzaki, and Dimitris Zeginis. Blank node matching and rdf/s comparison functions. In *The Semantic Web–ISWC 2012*, pages 591–607. Springer, 2012.

[UKM$^+$12] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank Van Harmelen, and Henri Bal. Webpie: A web-scale parallel inference engine using mapreduce. *Web Semantics: Science, Services and Agents on the World Wide Web*, 10:59–75, 2012.

[UVHSB11] Jacopo Urbani, Frank Van Harmelen, Stefan Schlobach, and Henri Bal. Querypie: Backward reasoning for owl horst over very large knowledge bases. *The Semantic Web–ISWC 2011*, pages 730–745, 2011.

[VG06] Max Völkel and Tudor Groza. Semversion: An rdf-based ontology versioning system. In *Proceedings of the IADIS international conference WWW/Internet*, volume 2006, page 44. Citeseer, 2006.

[WH09] Jesse Weaver and James A Hendler. *Parallel materialization of the finite rdfs closure for hundreds of millions of triples.* Springer, 2009.

[ZTC07] Dimitris Zeginis, Yannis Tzitzikas, and Vassilis Christophides. On the foundations of computing deltas between rdf models. In *The Semantic Web*, pages 637–651. Springer, 2007.

[ZTC11] Dimitris Zeginis, Yannis Tzitzikas, and Vassilis Christophides. On computing deltas of rdf/s knowledge bases. *ACM Transactions on the Web (TWEB)*, 5(3):14, 2011.

APPENDIX

Appendix A

FOUNDATIONAL WORK

An initial application-based experiment was conducted to build data mashup and extended it to produce RDF data.

A.1 Background

Mashups are basically applications that combine information retrieved from different web services through their open APIs to create a useful output of them. Open APIs are open interfaces that allow other applications or users to access their data which is usually in an XML (eXtensible Markup Language) format which provides operating system and programming-language independent data. Web services, in many cases, provide access to their APIs through different protocols such as REST and SOAP. The mashup combines information retrieved through APIs offered by eBay and Google Maps and can be accessed from: http://devweb2011.cis.strath.ac.uk:8180/sana/MashupClientServlet. It is built in Java and based on an online tutorial provided by IBM: `http://www.ibm.com/developerworks/xml/tutorials/x-ultimashup1/`. Both APIs retrieve information in XML which is then integrated to produce a simple web page displaying information about product from eBay and static map of the location of that product. The Semantic Web relies on XML and RDF as the base languages for other Semantic Web technologies. XML enables everyone to structure their documents by creating their own tags, but it says nothing about the meaning of the structure. The flexibility of XML in creating user-defined tags led to the development of various domain-specific XML models such as CML (Chemical Markup Language) and MathML. As XML has nothing to do with the meaning of the tags used in these models, there is a lack of standardization in describing the resources on the Web. Hence, a common model for describing data in these various models is required and this is the role of RDF which is the next layer in the Semantic Web stack. RDF has become a W3C standard for expressing the semantics of the data so that this data can be shared across different applications on the Web. Therefore, since RDF is a descriptive data model for representing and exchanging data on the Web which serves the goals of the Semantic Web, it would be useful to enable reuse of XML data in RDF.

A.2 Methodology

An extension to the experiment was carried out to generate RDF data from the information retrieved in XML format. This information was duplicated in order to gradually increase the size of the XML file and transform XML file to RDF each time.

The transformation process was done using a hand-coded eXtensible Stylesheet Language Transformations (XSLT). XSLT is an XML-based language for transforming XML documents.

An ontology for book store is created in order to extend the work to examine how ontology evolution may affect the generated RDF triples, and if the ontology change is limited, how possible it is to update the RDF instead of regenerating the whole RDF. Figure A.1 shows the architecture of the mashup and the transformation process.

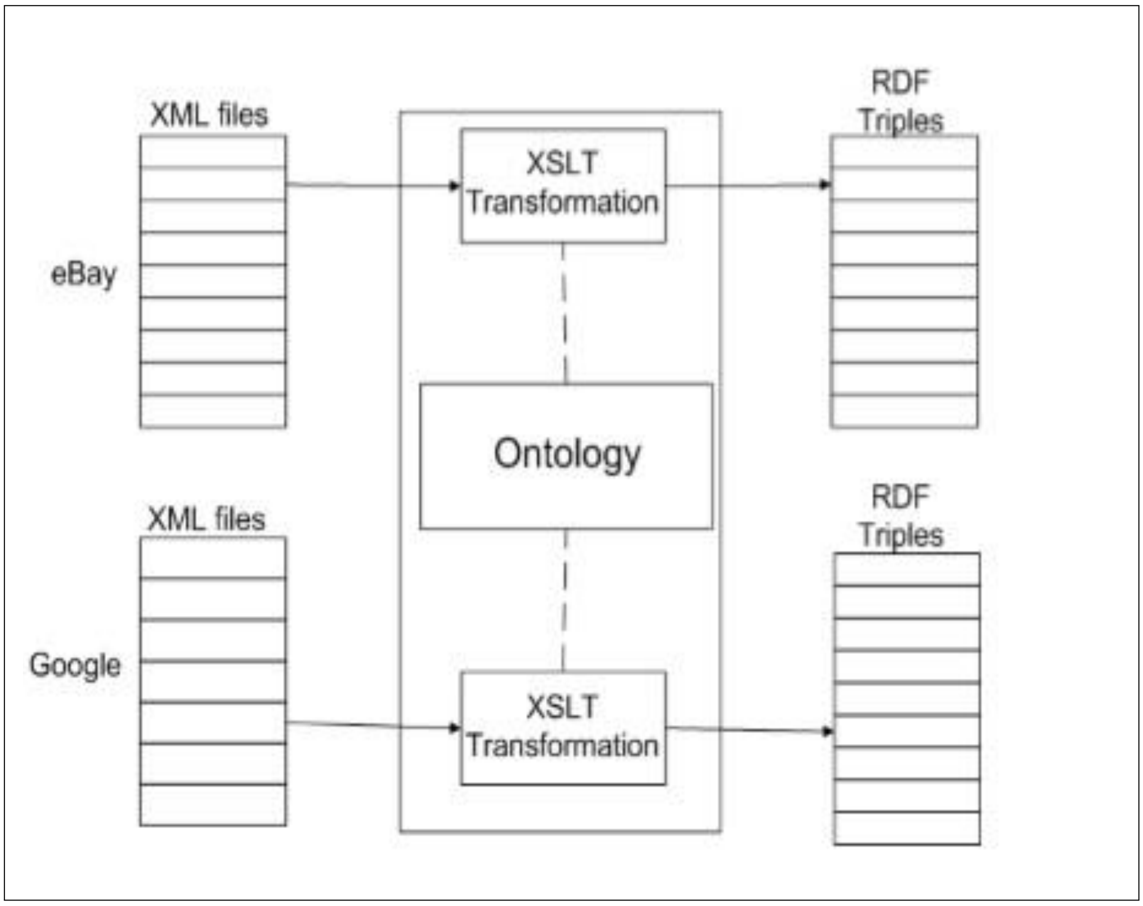


Fig. A.1: Architecture of the mashup

A.3 Results and discussion

It can be seen clearly from the graph in Figure A.2 that there is a considerable increase in the size of RDF compared to the corresponding XML. For example, the RDF file size is 70.4MB when XML file is 60.8 MB, which represents the same information in RDF but in XML format. This means that the increase in RDF file size is greater than the increase in the corresponding XML file size.

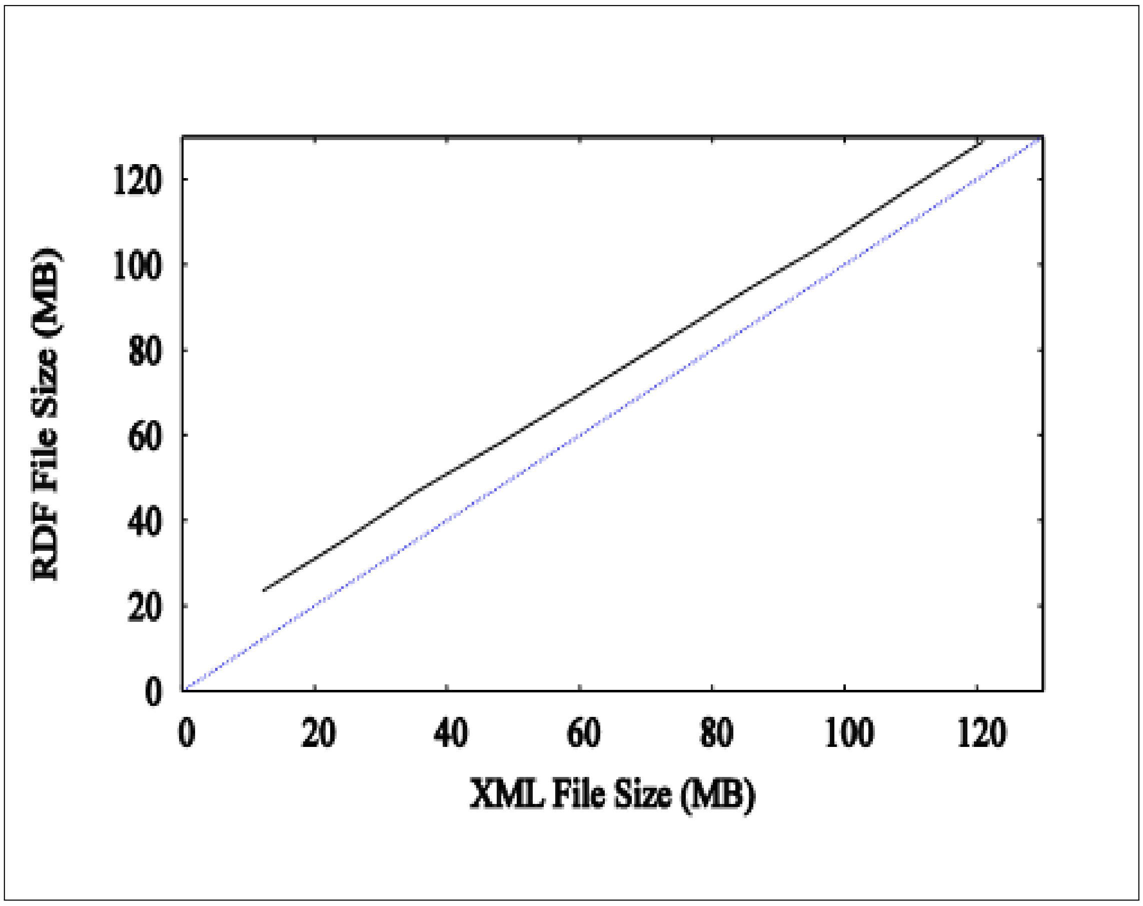


Fig. A.2: Sizes of RDF files generated from XML files. The solid line is the experimental line while the dashed line represents the unity line, which would occur if both sizes of XML and RDF are similar.

The linearity of the RDF/XML file size may be a product of duplicating the

XML data but shows that RDF is even more verbose then XML for the same information content. The time needed to produce RDF also increases more or less linearly with the size of the source data (Figure A.3).

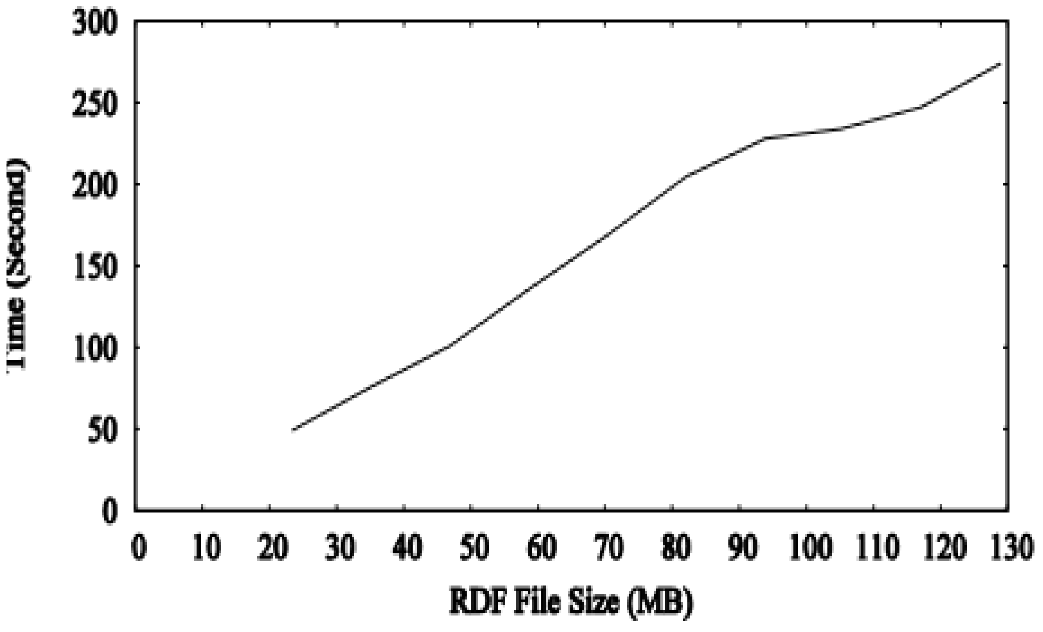


Fig. A.3: RDF generation time.

The collections of RDF are typically very large. On the web, RDF collections contain an extremely large number of triples (as evidenced in Figure A.4)), and this number is increasing rapidly. Moreover, these collections are not static they change overtime, and to be forced to rebuild a complete collection of RDF when this change happens is going to be a problem because these collections tend to be large.

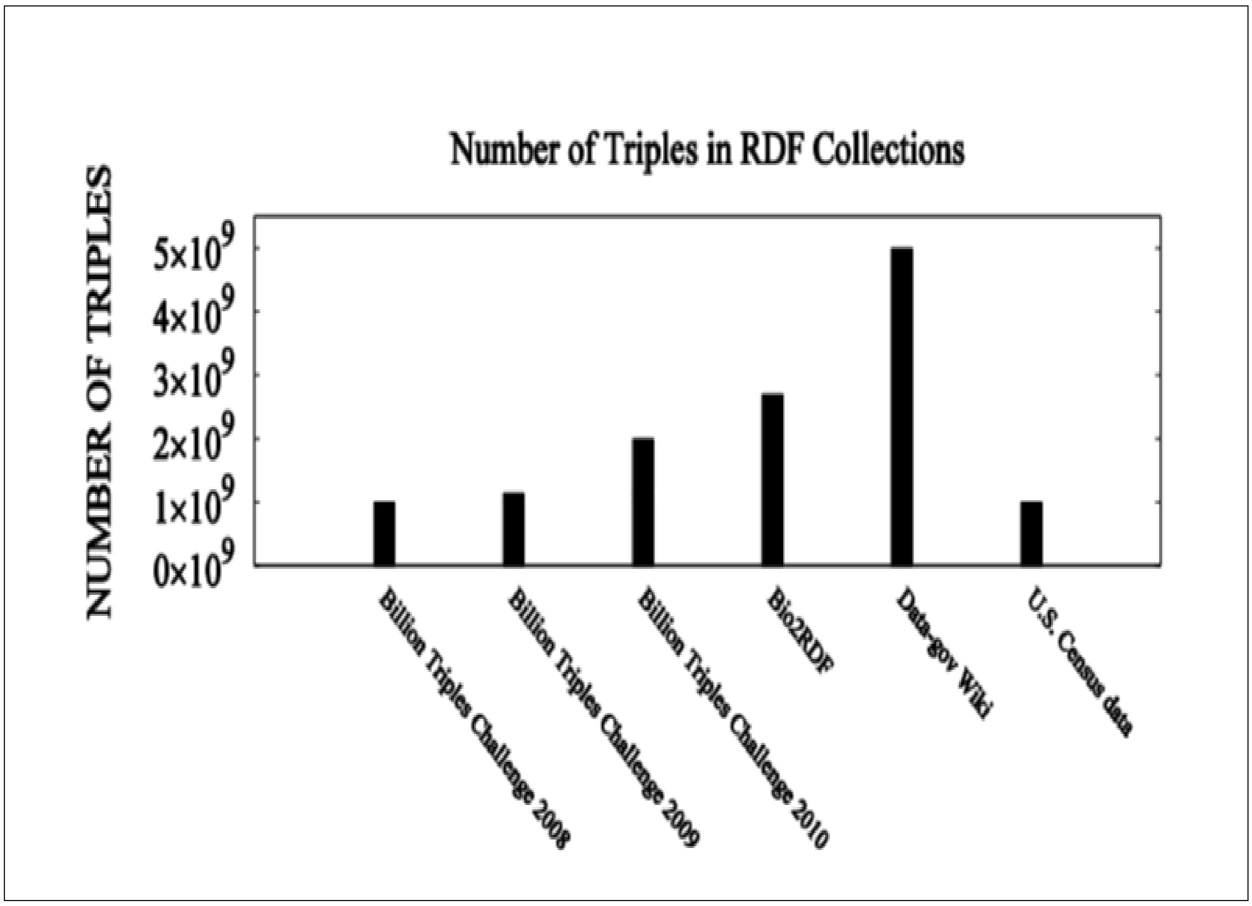


Fig. A.4: Number of triples in RDF collections.

From these results we concluded that it would be useful to find a way to 'repair' RDF structures rather than regenerate the whole structure each time there is a change.

Appendix B

A SUBSET OF OWL 2 RL/RDF RULESET

Abb	Antecedent	consequent

scm-sco	T(?c1, rdfs:subClassOf, ?c2)
	T(?c2, rdfs:subClassOf, ?c3)
		T(?c1, rdfs:subClassOf, ?c3)

scm-eqc1	T(?c1, owl:equivalentClass, ?c2)	T(?c1, rdfs:subClassOf, ?c2)
		T(?c2, rdfs:subClassOf, ?c1)

scm-eqc2	T(?c1, rdfs:subClassOf, ?c2)
	T(?c2, rdfs:subClassOf, ?c1)
		T(?c1, owl:equivalentClass, ?c2)

scm-hv	T(?c1, owl:hasValue, ?i)
	T(?c1, owl:onProperty, ?p1)
	T(?c2, owl:hasValue, ?i)
	T(?c2, owl:onProperty, ?p2)
	T(?p1, rdfs:subPropertyOf, ?p2)
		T(?c1, rdfs:subClassOf, ?c2)

Abb	Antecedent	consequent

scm-svf1	T(?c1, owl:someValuesFrom, ?y1)
	T(?c1, owl:onProperty, ?p)
	T(?c2, owl:someValuesFrom, ?y2)
	T(?c2, owl:onProperty, ?p)
	T(?y1, rdfs:subClassOf, ?y2)
		T(?c1, rdfs:subClassOf, ?c2)

scm-svf2	T(?c1, owl:someValuesFrom, ?y)
	T(?c1, owl:onProperty, ?p1)
	T(?c2, owl:someValuesFrom, ?y)
	T(?c2, owl:onProperty, ?p2)
	T(?p1, rdfs:subPropertyOf, ?p2)
		T(?c1, rdfs:subClassOf, ?c2)

scm-avf1	T(?c1, owl:allValuesFrom, ?y1)
	T(?c1, owl:onProperty, ?p)
	T(?c2, owl:allValuesFrom, ?y2)
	T(?c2, owl:onProperty, ?p)
	T(?y1, rdfs:subClassOf, ?y2)
		T(?c1, rdfs:subClassOf, ?c2)

scm-avf2	T(?c1, owl:allValuesFrom, ?y)
	T(?c1, owl:onProperty, ?p1)
	T(?c2, owl:allValuesFrom, ?y)
	T(?c2, owl:onProperty, ?p2)
	T(?p1, rdfs:subPropertyOf, ?p2)
		T(?c2, rdfs:subClassOf, ?c1)

Abb	Antecedent	consequent

scm-int	T(?c, owl:intersectionOf, ?x)
	LIST[?x, ?c1, ..., ?cn]
		T(?c, rdfs:subClassOf, ?c1)
		T(?c, rdfs:subClassOf, ?c2)
		...
		T(?c, rdfs:subClassOf, ?cn)

scm-uni	T(?c, owl:unionOf, ?x)
	LIST[?x, ?c1, ..., ?cn]
		T(?c1, rdfs:subClassOf, ?c)
		T(?c2, rdfs:subClassOf, ?c)
		...
		T(?cn, rdfs:subClassOf, ?c)

cax-sco	T(?c1, rdfs:subClassOf, ?c2)
	T(?x, rdf:type, ?c1)
		T(?x, rdf:type, ?c2)

cax-eqc1	T(?c1, owl:equivalentClass, ?c2)
	T(?x, rdf:type, ?c1)
		T(?x, rdf:type, ?c2)

cax-eqc2	T(?c1, owl:equivalentClass, ?c2)
	T(?x, rdf:type, ?c2)
		T(?x, rdf:type, ?c1)

cls-svf1	T(?x, owl:someValuesFrom, ?y)
	T(?x, owl:onProperty, ?p)
	T(?u, ?p, ?v)
	T(?v, rdf:type, ?y)
		T(?u, rdf:type, ?x)

cls-svf2	T(?x, owl:someValuesFrom, owl:Thing)
	T(?x, owl:onProperty, ?p)
	T(?u, ?p, ?v)
		T(?u, rdf:type, ?x)

Abb	Antecedent	consequent

cls-avf	T(?x, owl:allValuesFrom, ?y)
	T(?x, owl:onProperty, ?p)
	T(?u, rdf:type, ?x)
	T(?u, ?p, ?v)
		T(?v, rdf:type, ?y)

cls-hv1	T(?x, owl:hasValue, ?y)
	T(?x, owl:onProperty, ?p)
	T(?u, rdf:type, ?x)
		T(?u, ?p, ?y)

cls-hv2	T(?x, owl:hasValue, ?y)
	T(?x, owl:onProperty, ?p)
	T(?u, ?p, ?y)
		T(?u, rdf:type, ?x)

cls-int1	T(?c, owl:intersectionOf, ?x)
	LIST[?x, ?c1, ..., ?cn]
		T(?y, rdf:type, ?c1)
		T(?y, rdf:type, ?c2)
		...
		T(?y, rdf:type, ?cn)
		T(?y, rdf:type, ?c)

cls-int2	T(?c, owl:intersectionOf, ?x)
	LIST[?x, ?c1, ..., ?cn]
	T(?y, rdf:type, ?c)
		T(?y, rdf:type, ?c1)
		T(?y, rdf:type, ?c2)
		...
		T(?y, rdf:type, ?cn)

cls-uni	T(?c, owl:unionOf, ?x)
	LIST[?x, ?c1, ..., ?cn]
	T(?y, rdf:type, ?ci)
		T(?y, rdf:type, ?c)

Abb	Antecedent	consequent

prp-inv1	T(?p1, owl:inverseOf, ?p2)
	T(?x, ?p1, ?y)
		T(?y, ?p2, ?x)

prp-inv2	T(?p1, owl:inverseOf, ?p2)
	T(?x, ?p2, ?y)
		T(?y, ?p1, ?x)

prp-eqp1	T(?p1,owl:equivalentProperty, ?p2)
	T(?x, ?p1, ?y)
		T(?x, ?p2, ?y)

prp-eqp2	T(?p1,owl:equivalentProperty, ?p2)
	T(?x, ?p2, ?y)
		T(?x, ?p1, ?y)

prp-symp	T(?p, rdf:type,owl:SymmetricProperty)
	T(?x, ?p, ?y)
		T(?y, ?p, ?x)

prp-trp	T(?p, rdf:type,owl:TransitiveProperty)
	T(?x, ?p, ?y)
	T(?y, ?p, ?z)
		T(?x, ?p, ?z)

prp-spo1	T(?p1, rdfs:subPropertyOf, ?p2)
	T(?x, ?p1, ?y)
		T(?x, ?p2, ?y)

prp-dom	T(?p, rdfs:domain, ?c)
	T(?x, ?p, ?y)
		T(?x, rdf:type, ?c)

prp-rng	T(?p, rdfs:range, ?c)
	T(?x, ?p, ?y)
		T(?y, rdf:type, ?c)

scm-dom1	T(?p, rdfs:domain, ?c1)
	T(?c1, rdfs:subClassOf, ?c2)
		T(?p, rdfs:domain, ?c2)

Abb	Antecedent	consequent

scm-dom2	T(?p2, rdfs:domain, ?c)
	T(?p1, rdfs:subPropertyOf, ?p2)
		T(?p1, rdfs:domain, ?c)

scm-rng1	T(?p, rdfs:range, ?c1)
	T(?c1, rdfs:subClassOf, ?c2)
		T(?p, rdfs:range, ?c2)

scm-rng2	T(?p2, rdfs:range, ?c)
	T(?p1, rdfs:subPropertyOf, ?p2)
		T(?p1, rdfs:range, ?c)

scm-dp	T(?p, rdf:type, owl:DatatypeProperty)
	T(?p, rdfs:subPropertyOf, ?p)
		T(?p, owl:equivalentProperty, ?p)

scm-spo	T(?p1, rdfs:subPropertyOf, ?p2)
	T(?p2, rdfs:subPropertyOf, ?p3)
		T(?p1, rdfs:subPropertyOf, ?p3)

scm-eqp1	T(?p1, owl:equivalentProperty, ?p2)	T(?p1, rdfs:subPropertyOf, ?p2)
		T(?p2, rdfs:subPropertyOf, ?p1)

scm-eqp2	T(?p1, rdfs:subPropertyOf, ?p2)
	T(?p2, rdfs:subPropertyOf, ?p1)
		T(?p1, owl:equivalentProperty, ?p2)

Tab. B.1: Selected rules from OWL 2 RL/RDF ruleset [MGH+09]

Appendix C

PROOF OF THE CORRECTNESS OF ΔD_C

Proposition 1. The correctness of ΔD_c is established by conditional proof[1]. Let M and M' denote two RDF models, let Del and Ins denote the triples to be deleted from and inserted into M to update it to M' under ΔE, let Del' and Ins' denote the triples to be deleted from and inserted into M to update it to M' under ΔD_c, let $C(M)$ and $C(M')$ denote the closure of M and M' respectively. It is claimed that

$$C(M \backslash Del' \cup Ins') = C(M') = C(M \backslash Del \cup Ins)$$

[1] This proof was contributed by Dr Clemens Kupke, University of Strathclyde.

Proof.

Suppose $b \in M \backslash Del' \cup Ins'$

Subcase $b \in M \backslash Del'$

If $b \in M \backslash Del$ then $b \in M'$ and $b \in C(M')$

If $b \in Del \backslash Del'$ then by definition of Del' we have $b \in C(M')$

Subcase $b \in Ins'$

But $Ins' \subseteq Ins \subseteq M'$ so clearly $b \in C(M')$

Suppose $b \in M \backslash Del \cup Ins$

Subcase $b \in M \backslash Del \Rightarrow b \in M \backslash Del' \Rightarrow b \in C(M')$

Subcase $b \in Ins$

Suppose $b \in Ins'$ then $b \in C(M')$

Suppose $b \notin Ins'$ by definition of Ins' we know

$b \in C(M)$ and does not depend on Del'

$\Rightarrow b \in C(M \backslash Del') \subseteq C(M \backslash Del' \cup Ins')$

$\square$

Appendix D

CHANGE DETECTION: EXPERIMENTAL DATA

FC Processing Time

UPDATES	TOTAL	INF	DELTA	APP
43,136.0	288,115.2	12,526.2	10,535.4	265,039.0
116,710.0	1,360,343.2	16,785.2	8,855.6	1,334,701.2
189,253.0	1,521,340.8	19,800.0	21,750.4	1,479,759.6
210,372.0	1,633,272.4	23,660.0	26,232.0	1,583,379.2
237,510.0	1,743,232.0	26,785.0	31,820.4	1,684,615.8
265,609.0	1,787,331.4	29,284.0	34,499.4	1,723,541.2
308,594.0	1,857,939.0	31,058.0	38,890.6	1,787,984.4
348,819.0	1,910,102.0	34,369.0	42,988.2	1,832,734.8
367,233.0	1,931,490.8	36,913.0	44,630.8	1,849,940.0

BC Processing Time

UPDATES	TOTAL	DELTA	INF	APP
43,136.0	311,423.4	11,859.2	547.6	299,015.6
116,710.0	1,370,769.4	16,023.2	907.6	1,353,837.6
189,253.0	1,635,292.4	25,907.0	1,596.6	1,607,787.8
210,372.0	1,742,764.4	29,210.4	2,653.0	1,710,900.2
237,510.0	1,815,300.0	30,615.0	3,316.2	1,781,367.8
265,609.0	1,889,555.8	34,414.6	4,314.0	1,850,826.4
308,594.0	1,906,702.8	43,289.0	4,726.6	1,858,686.6
348,819.0	1,972,542.6	45,382.0	5,552.8	1,921,607.2
367,233.0	1,989,457.2	47,853.8	5,479.4	1,936,123.0

EC Processing Time

UPDATES	TOTAL	DELTA	APP
43,136.0	310,213.0	11,748.0	298,464.4
116,710.0	1,405,034.4	16,834.0	1,388,199.6
189,253.0	1,648,819.6	27,408.4	1,621,410.6
210,372.0	1,772,373.6	28,904.4	1,743,468.8
237,510.0	1,848,989.0	33,195.6	1,815,792.8
265,609.0	1,914,169.2	35,612.0	1,878,557.0
308,594.0	1,950,972.2	41,406.2	1,909,565.2
348,819.0	1,998,234.6	46,750.0	1,951,483.8
367,233.0	2,036,642.0	47,917.4	1,988,723.8

PBC Processing Time

UPDATES	TOTAL	DELTA	PRUN	INF	APP	REASONING
43,136.0	309,785.6	12,024.8	9,102.4	386.2	288,270.8	9,488.6
116,710.0	1,415,910.2	17,024.3	18,965.7	522.3	1,383,904.8	19,488.0
189,253.0	1,685,152.0	25,212.4	38,303.8	888.2	1,620,746.0	39,192.0
210,372.0	1,800,994.6	28,435.4	58,348.0	1,163.4	1,713,046.0	59,511.4
237,510.0	1,888,765.8	32,547.4	78,961.6	1,511.6	1,775,744.0	80,473.2
265,609.0	1,964,734.6	37,248.4	100,540.4	1,641.2	1,825,303.0	102,181.6
308,594.0	2,029,878.0	42,092.8	113,391.6	1,641.0	1,872,751.2	115,032.6
348,819.0	2,149,090.2	48,545.2	157,389.2	1,867.0	1,941,287.8	159,256.2
367,233.0	2,155,001.0	49,704.2	168,394.4	1,971.2	1,934,929.2	170,365.6

Appendix E

SOFTWARE CONFIGURATION

Experiment	**RDF views update**	**Change detection techniques**	**Correct dense delta**	**Pruning OWL 2 ruleset**
Platform	Intel® Core™ i7-3520M CPU @ 2.90GHz, 2901 Mhz, 2Core(s), 4 Logical Processor(s), Windows 7 Pro operating system and 12GB memory	Intel® Xeon® CPU X3470,@ 2.93GHz – 1CPU with 4 Cores and hyperthreading, Ubuntu 12.04 LTS operating system and 16GB memory		
Java Platform	Eclipse Java EE IDE. Version: Juno Service Release 2 Heap space: -Xms4096M -Xmx10240M			
Data store	MySQL. Software version: 5.5.27			
Dataset	Gene Ontology (GO)	Gene Ontology (Go) Uniprot Taxonomy	Gene Ontology (GO) and Uniprot Taxonomy enhanced by synthetic data	LUBM and UOBM

Tab. E.1: Software configuration