University of Crete
Computer Science Department

On Computing Deltas of RDF/S Knowledge Bases with Blank Nodes

Christina Lantzaki
Master's Thesis

Heraklion, January 2013

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Υπολογισμός Διαφορών μεταξύ RDF Βάσεων Γνώσης με Ανώνυμους Κόμβους

Εργασία που υποβλήθηκε απο τον

Χριστίνα Α. Λαντζάκη

ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

Χριστίνα Λαντζάκη, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Ιωάννης Τζίτζικας, Επίκουρος καθηγητής, Επόπτης

Δημήτρης Πλεξουσάκης,Καθηγητής, Μέλος

Γιώργος Γεωργακόπουλος,Αναπληρωτής Καθηγητής ,Μέλος

Δεκτή:

Πάνος Τραχανιάς, Καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών
Ηράκλειο, Ιανουάριος 2013

On Computing Deltas of RDF/S Knowledge Bases with Blank Nodes

Christina Lantzaki
Master's Thesis
Computer Science Department, University of Crete

Abstract

The Semantic Web (SW) is an evolving extension of the World Wide Web in which the content can be expressed not only in natural language, but also in formal languages (e.g. RDF/S) that can be read and used by software agents, permitting them to find, share and integrate information more easily. The statement of Heraclitus "Everything flows, nothing stands still" holds also in the context of the SW since everything changes (the resources themselves, the ontologies, the resource descriptions, etc). Consequently, the ability to compute the differences that exist between two RDF/S Knowledge Bases (KBs), hereafter Delta, is very important. In particular, Deltas can be employed to (a) aid humans understand the evolution of knowledge, and (b) reduce the amount of data that need to be exchanged and managed over the network in order to build SW synchronization, versioning and replication services.

The comparison problem is not simple because RDF allows anonymous nodes. A anonymous node, else called blank node, is a node in an RDF graph which is not identified by a URI and is not a literal. Several RDF KBs rely heavily on blank nodes (e.g. 7.5% of Linked Data are estimated to be blank nodes) as they are convenient for representing complex attributes or resources whose identity is unknown but their attributes (either literals or associations with other resources) are known. Considering blank nodes as "constants" unique to both graphs does not help either in detecting equivalence between graphs nor in reducing the Delta. On the contrary, matching the blank nodes of the two graphs can significantly reduce the produced delta. This work is the first that focuses on methods of matching the blank nodes of the one graph with the blank nodes of the other graph, by approaching it as an optimization problem aiming at finding the mapping that yields the minimum in size Delta (with the least number of triples to delete or add to make the graphs equivalent). We prove that finding the optimal blank node mapping is NP-Hard in the general case by reducing to the sub-graph isomorphism problem. When graphs do not contain directly connected blank nodes (no triples with more than one blank nodes exists), we show that the polynomial Hungarian algorithm can be used to find the optimal blank node mapping. For the general case we present various polynomial algorithms returning approximate solutions. For making the application of our method feasible also to very large KBs we present a signature-based mapping algorithm with $NlogN$ time complexity.

Finally, for the proposed algorithms we report extensive comparative experimental results, over real and synthetic KBs, regarding delta reduction (and its deviation from the optimal), equivalence detection, and computational requirements. The results are very interesting. Indicatively the signature-based algorithm can match KBs with up to 150,000 bnodes in a few seconds with $Y\%$ deviation from the optimal.

Supervisor: Yannis Tzitzikas
Assistant Professor

Σύγκριση RDF/S Βάσεων Γνώσης με Ανώνυμους κόμβους

Χριστίνα Λαντζάκη
Μεταπτυχιακή Εργασία
Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Ο Σημασιολογικός Ιστός (ΣΙ) είναι μια εξελισσόμενη επέκταση του Παγκόσμιου Ιστού στην οποία το περιεχόμενο μπορεί να εκφραστεί όχι μόνο με φυσική γλώσσα αλλά και με τυπικές γλώσσες (π.χ. RDF/S), που επιτρέπουν την παροχή προηγμένων υπηρεσιών αναζήτησης, διαμοιρασμού και ολοκλήρωσης πληροφορίας. Η ρήση του Ηράκλειτου «τα πάντα ρει» ισχύει και στον ΣΙ, αφού οι πόροι (resources), οι οντολογίες, τα μεταδεδομένα διαρκώς αλλάζουν και εξελίσσονται. Εκ τούτου η ικανότητα υπολογισμού των διαφορών μεταξύ δύο Βάσεων Γνώσεων (ΒΓ) RDF/S , στο εξής Δέλτα, είναι πολύ σημαντική. Συγκεκριμένα, τα Δέλτα μπορούν (α) να βοηθήσουν τους χρήστες στην κατανόηση της εξέλιξης της γνώσης, και (β) να μειώσουν τον όγκο των δεδομένων που χρειάζεται να ανταλλαχθούν και διαχειριστούν στο δίκτυο για την ανάπτυξη υπηρεσιών συγχρονισμού, διαχείρισης εκδόσεων και αντιγράφων.

Το πρόβλημα της σύγκρισης δεν είναι απλό διότι η RDF υποστηρίζει ανώνυμους κόμβους. Ένας ανώνυμος κόμβος είναι ένας κόμβος σε ένα ΡΔΦ γράφο, ο οποίος δεν είναι ούτε URI, ούτε literal και επομένως δεν έχει εξωτερική ταυτότητα. Αρκετές βάσεις γνώσεων περιλαμβάνουν μεγάλο ποσοστό ανώνυμων κόμβων (π.χ. το 7.5% των Διασυνδεδεμένων Δεδομένων εκτιμάται ότι αντιστοιχεί σε ανώνυμους κόμβους), καθώς είναι κατάλληλοι για την αναπαράσταση σύνθετων γνωρισμάτων ή κόμβων των οποίων η ταυτότητα είναι άγνωστη, αλλά οι ιδιότητες τους (είτε literals ή συνδέσεις τους με άλλους κόμβους) είναι γνωστές. Η θεώρηση των ανώνυμων κόμβων σαν μοναδικές σταθερές στους δύο γράφους δε βοηθάει ούτε στον εντοπισμό ισοδύναμων γράφων, ούτε στη μείωση του Δέλτα. Αντιθέτως, με την αντιστοίχιση των ανώνυμων κόμβων του ενός γράφου με τους ανώνυμους κόμβους του άλλου γράφου μπορούμε να μειώσουμε σημαντικά το παραγόμενο Δέλτα. Η παρούσα εργασία είναι η πρώτη εργασία που εστιάζει σε μεθόδους αντιστοίχισης των ανώνυμων κόμβων προσεγγίζοντας το πρόβλημα ως πρόβλημα βελτιστοποίησης με στόχο την εύρεση της αντιστοίχισης που οδηγεί στο ελάχιστο σε μέγεθος Δέλτα (ελάχιστο πλήθος τριπλετών που πρέπει να διαγραφούν και να προστεθούν για να γίνουν οι δύο γράφοι ισοδύναμοι). Αποδεικνύουμε ότι η εύρεση της βέλτιστης αντιστοίχισης είναι NP-Hard στη γενική περίπτωση ανάγοντας το πρόβλημα της αντιστοίχισης στο πρόβλημα του ισομορφισμού υπογράφων. Επιπλέον αποδυκνείουμε ότι όταν οι γράφοι δεν περιέχουν ανώνυμους κόμβους που συνδέονται άμεσα μεταξύ τους (ήτοι, καμία τριπλέτα δεν περιέχει περισσότερους από έναν ανώνυμους κόμβους), τότε ο πολυωνυμικής πολυπλοκότητας Ουγγρικός αλγόριθμος μπορεί να δώσει τη βέλτιστη λύση. Για τη γενική περίπτωση προτείνουμε διάφορους πολυωνυμικούς αλγορίθμους, που μπορούν να λύσουν το πρόβλημα προσεγγιστικά. Για να είναι εφικτή η εφαρμογή των μεθόδων μας ακόμα και σε πολύ μεγάλες βάσεις γνώσεων, που περιέχουν μεγάλο πλήθος ανώνυμων κόμβων, δίνουμε έναν προσεγγιστικό αλγόριθμο με NlogN πολυπλοκότητα ο οποίος βασίζεται σε υπογραφές. Τέλος, για τους προτεινόμενους αλγόριθμους, αναφέρουμε εκτεταμένα συγκριτικά πειραματικά αποτελέσματα επί πραγματικών και συνθετικά παραγμένων ΒΓ, σχετικά με τις επιδόσεις τους στη μείωση του Δέλτα (και την απόκλιση τους από το βέλτιστο), στον εντοπισμό ισοδύναμων γράφων, και τις υπολογιστικές τους

απαιτήσεις. Τα αποτελέσματα είναι πολύ ενδιαφέροντα. Ενδεικτικά, ο αλγόριθμος που βασίζεται σε υπογραφές μπορεί να αντιστοιχίσει βάσεις που έχουν μέχρι και 150 χιλιάδες ανώνυμους κόμβους σε λίγα δευτερόλεπτα και $U\%$ απόκλιση από το βέλτιστο.

Επόπτης Καθηγητής: Γιάννης Τζίτζικας
Επίκουρος Καθηγητής

Acknowledgements

First of all, I would like to thank my supervisor, professor of the University of Crete, Yannis Tzitzikas, initially for trusting me and then for his continuous support and his valuable advice. His organizational capability was really helpful for me, too. I am also grateful to the professors Dimitris Pleksousakis and Giorgos Georgakopoulos for participating in the supervisory committee.

I would also like to thank the Computer Science Department of Greece for offering a high level of academic education and the Information-Systems Laboratory of ICS-FORTH for its support in terms of graduate fellowship and for providing a high-levelled research environment.

Last but not least, I would like to thank my friends and my parents for supporting me all over this period.

This work was partly supported by the NoE *APARSEN* (Alliance Permanent Access to the Records of Science in Europe, FP7, Proj. No 269977, 2011-2014), and the FP7 Research Infrastructures projects *SCIDIP-ES* (SCIence Data Infrastructure for Preservation - Earth Science, 2011, 2014), and *iMarine* (FP7 Research Infrastructures, 2011-2014).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Life is a series of natural and spontaneous changes.
Don't resist them, that only creates sorrow.
Let reality be reality.
Let things flow naturally forward in whatever way they like.
Lao Tzu

1.1 The Semantic Web: Concepts and Vision

The content of the World Wide Web is currently formatted in a natural language, mainly through HTML. Even though such a language is human-readable and human-understandable, the machines or else the software agents are only able to read this information. The machine-intelligibility cannot be achieved with the current technology.

This gap is called to be solved through the Semantic Web, an extension of the World Wide Web (WWW). The term was coined by the Tim Berners-Lee, the inventor of the WWW and the director of the Word Wide Web Consortium (W3C), and aims at converting unstructured and semi-structured documents into semantically structured knowledge, that can be processed directly and indirectly by machines. The promoted formats give the ability to the machines to interpret the content of the web page and find, share and integrate information more easily.

The Semantic Web is standardized, in the context of Web 3.0, with a set of new languages organized in a layered architecture enabling users and applications to write and share information in a machine-readable way. This layered architecture of the Semantic Web is often referred to as the Semantic Web stack.

Current research proves that the promoted technology is already in use and starts to reach the market.

1.1.1 The Semantic Web Stack: Components

The Semantic Web stack (Figure 1.1) is a work in progress, where the layers are developed in a bottom-up manner. The so far well-established technologies are specified as W3C standards and include RDF, RDF Schema, SKOS, SPARQL and OWL.

The basic components are given below:

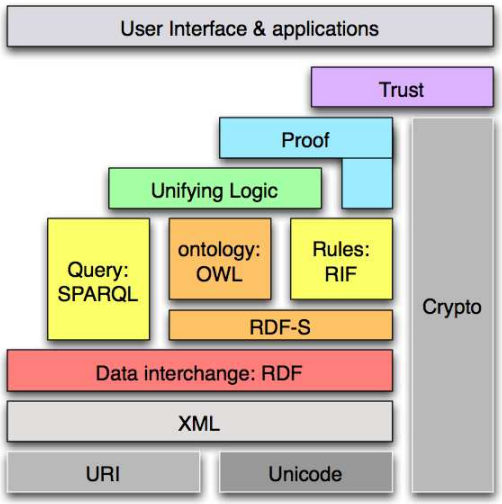

Figure 1.1: The Semantic Web Stack

- XML provides an elemental syntax for content structure within documents, yet associates no semantics with the meaning of the content contained within. XML is not at present a necessary component of Semantic Web technologies in most cases, as alternative syntaxes exists (such as Turtle).

- RDF (Resource Description Framework) is a simple language for expressing data models, which refers to resources (web resources) and asserts binary relationships between them. Such assertions have the form of triples, called statements. The elements of a triple are called subject, predicate and object. A collection of RDF statements intrinsically represents a labeled, directed multi-graph. As such, an RDF-based data model is more naturally suited to certain kinds of knowledge representation than the relational model and other ontological models. An RDF-based model can be represented in a variety of syntaxes, such as RDF/XML, Notation3, Turtle, NTriples and RDFa. As this thesis focuses on this layer, more details are given in the following sections.

- RDFS (RDF Schema) is a vocabulary that extends RDF for describing properties and classes of RDF-based resources, with semantics for generalized-hierarchies. In particular, the statements of RDF Schema (RDFS) make it possible to define hierarchies of classes, hierarchies of properties and to describe domains and ranges of the properties.

- OWL (Web Ontology Language) builds-up on RDFS introducing more expressive description constructs. In particular, OWL adds relations between classes, cardinality, equality, richer typing of properties, characteristics of properties, and enumerated classes. OWL has three increasingly expressive sublanguages: OWL Lite, OWL DL and OWL Full. OWL Lite supports those users primarily needing a classification hierarchy and simple constraints. OWL DL supports users who want the maximum expressiveness while retaining computational completeness and decidability. OWL Full is

meant for users who want very high expressiveness and the syntactic freedom of RDF with no computational guarantees. In OWL 2, there are three sublanguages of the language. OWL 2 EL is a fragment that has polynomial time reasoning complexity; OWL 2 QL is designed to enable easier access and query to data stored in databases; OWL 2 RL is a rule subset of OWL 2.

- SPARQL is the most prevalent RDF query language for semantic web data resources, able to retrieve and manipulate data stored in Resource Description Framework format. In general, querying allows fine-grained data access.

The rest of the components are not yet fully standardized or realized, such as the Rule Interchange Format (RIF) in the Rule Layer or the Unifying Logic and Proof Layer.

In all cases the intention of the semantic web is to enhance the usability and usefulness of the Web and its interconnected resources through servers which expose existing data systems using the RDF and SPARQL standards, through converters to RDF, through documents "marked up" with semantic information, through common metadata vocabularies (ontologies) and maps between vocabularies, through automated agents to perform tasks for users or finally through web-based services to supply information specifically to agents.

Alternatively, Figure 1.2 offers a 3D presentation of the Semantic Web stack, providing more information without sacrificing compactness and simplicity [1]. The one side gives the concepts and the abstraction of the Semantic Web and the other side gives the specification and the solutions for each concept.

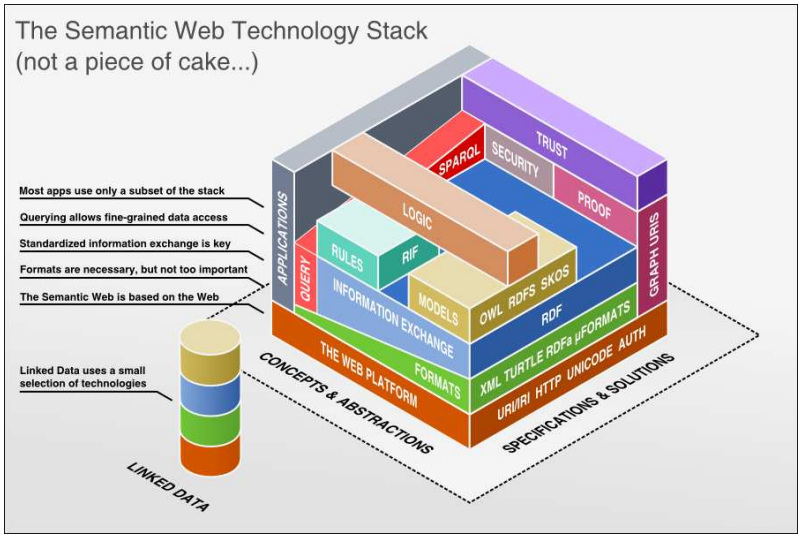

Figure 1.2: The Semantic Web Stack in 3D presentation

Most applications just use only a subset of the described stack. Indicatively, the Linked Data (see more in section 1.2) use a small selection of the technologies and in particular the RDF and OWL technology.

[1] http://bnode.org/

1.2 Linked Data

Linked Data is about using the Web to connect related data that wasn't previously linked, or using the Web to lower the barriers to linking data currently linked using other methods. More specifically, Wikipedia defines Linked Data as "a term used to describe a recommended best practice for exposing, sharing, and connecting pieces of data, information, and knowledge on the SW using URIs and RDF."

Tim Berners-Lee coined the term in a design note discussing issues around the SW project. However, the idea is very old and is closely related to concepts including database network models, citations between scholarly articles, and controlled headings in library catalogs.

The goal of the W3C SW community project is to extend the Web with a data commons by publishing various open datasets as RDF on the Web and by setting RDF links between data items from different data sources. In 2007, datasets consisted of over two billion RDF triples, which were interlinked by over two million RDF links. By 2011 this had grown to 31 billion RDF triples, interlinked by around 504 million RDF links. There is also an interactive visualization of the Linked datasets to browse through the cloud. Here we provide a static visualization of the cloud, seen in Figure 1.3.

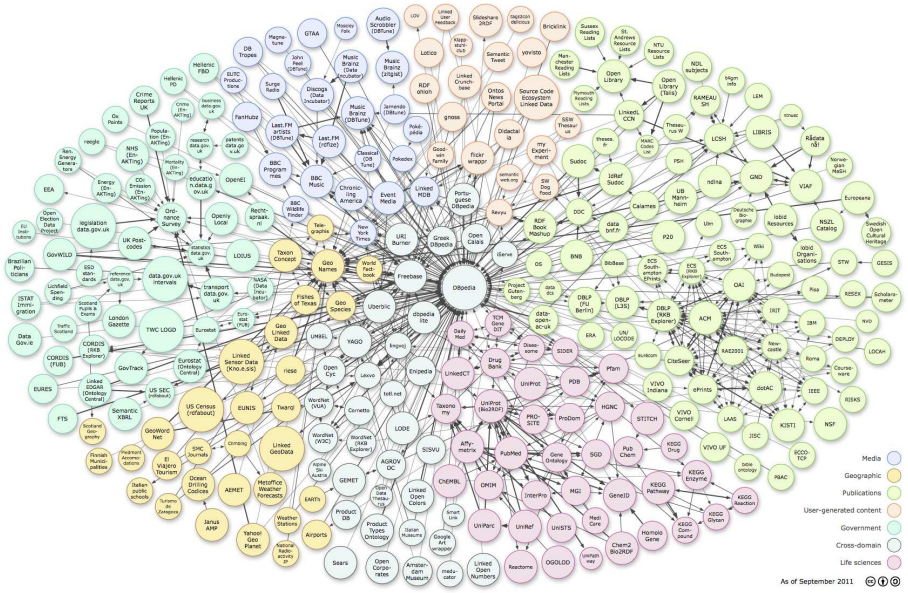

Figure 1.3: The Linked Open Data Cloud

It follows a list with some main datasets inside or outside LOD cloud.

- CKAN: registry of open data and content packages provided by the Open Knowledge Foundation

- DBpedia: a dataset containing extracted data from Wikipedia. It contains about 3.4 million concepts described by 1 billion triples, including abstracts in 11 different languages

- GeoNames: provides RDF descriptions of more than $7,500,000$ geographical features worldwide
- UMBEL: a lightweight reference structure of $20,000$ subject concept classes and their relationships derived from OpenCyc, which can act as binding classes to external data. It also has links to 1.5 million named entities from DBpedia and YAGO
- FOAF: a dataset describing persons, their properties and relationships

1.3 RDF and Blank Nodes

As we have already mentioned, RDF is a data model that represents knowledge in the form of simple statements, called RDF triples, which consist of a subject, a predicate and an object, like a simple sentence in a human language. The subject is a thing(resource) that a statement describes, the predicate of a statement identifies a property or a relation, while the object is a value of a property or a target of a relation.

In RDF, there are three types of nodes, URI references, blank nodes and literals. URI references identify resources, blank nodes represent anonymous resources that are not assigned a URI, and literals denote values such as numbers or dates. The subject of an RDF triple may be a URI reference or a blank node, the predicate must be a URI reference, and the object may be all three kinds of all three kinds (URI references, literals, blank nodes). When combined together, RDF triples form a direct labeled graph (RDF graph). Subjects and objects of RDF triples become nodes in an RDF graph and predicates become arcs connecting them.

Although RDF is based on a simple idea, there are some problems that make it complicated. One main problem is the existence of blank nodes.

1.3.1 Theoretical Perspective

Nodes without a name represent a special kind of nodes, called blank nodes (for short bnodes). These nodes simply indicate the existence of a thing, without using or saying anything about the name of that thing. Therefore, according to the standard, they are referred to as existential variables of an RDF graph.

Due to the absence of a name (URI), manipulating data containing blank nodes is much harder. They make otherwise trivial operations (like the comparison of two KBs or the simple entailment checking) far more complex or even intractable.

On the other hand, they enable a great flexibility in expressing.

In this flexibility a more profound reason is hidden, which perhaps can explain how blank nodes have survived as a part of the RDF model all these years despite all the headaches they have caused. The thing is, blank nodes reflect a human way of referencing things. Let us show an example:

If I want to talk about my left arm, it is quite unnatural to invent a new identifier for it. I will just say "my left arm", describing it relative to myself, and a listener will understand. This is possible due to human's ability to understand the context. He or she knows that the pronoun "my" refers to me as something unique, the arm being part of me, and the "left" finally specifying the exact arm. So, my left arm, unique in the universe, is referenced quite simply and elegantly.

In RDF, it can be expressed with the two statements as: "I have an arm. It (has a property that) is left". Let us assume that we know that the blank node is of a type "ex:Arm" implicitly through the property:

```
ex:hasArm rdfs:range ex:Arm
```

Given that the URI of me is http://foaf/id/3243435, and assuming the relevant properties are defined in ex ontology, we can express it with the following triples:

```
http://foaf/id/3243435 ex:hasArm [
    ex:hasProperty ex:Left
]
```

The left arm is represented by the blank node, which is the object in the first triple and the subject in the second, thus chaining them and forming a rather readable code.

1.3.2 Practical Perspective

While in theory blank nodes do not have a name, in practice, when publishing data, they can be assigned an ID in a local graph/document scope, in order to enable several RDF triples to reference the same unidentified resource. This local identifier is called a "blank node identifier" and it is different from URIs or literals, because it does not provide a unique name in the global context. This identifier is denoted by the characters ("_ : ") followed by a string. It follows an example for better understanding of the benefits of this assignment.

Figure 1.4 shows some triples in RDF-XML syntax, where the blank node can is represented by nested elements.

```
< foaf : Person  rdf : about = "http : //example.org/Person#John" >
    < foaf : knows >
        < foaf : Person  foaf : birthDate = "04 − 21" >
    < /foaf : knows >
< /foaf : Person >
```

Figure 1.4: Blank node with nested elements

If the same blank node is used more than once in the same RDF graph, it can be identified by giving an identifier, like in Figure 1.5. So, now we can express the fact that John and Mary have a common friend.

```
< foaf : Person  rdf : about = "http : //example.org/Person#John" >
    < foaf : knows >
      < foaf : Person  rdf : nodeID = "b1" / >
    < /foaf : knows >
< /foaf : Person >
< foaf : Person  rdf : about = "http : //example.org/Person#Mary" >
    < foaf : knows >
      < foaf : Person  rdf : nodeID = "b1" / >
    < /foaf : knows >
< /foaf : Person >
```

Figure 1.5: Blank node with nested elements

According to a current paper [9], the usage of blank nodes is attributed to the following aspects:

- Blank nodes have the capability to describe "Multi-component Structures" or else "Complex Attributes".
  Figure 1.6 is an example of an RDF graph with such functionality. Complex attributes (e.g. an attribute `address` of Figure 1.6) can be represented without having to name explicitly the auxiliary node that is used for connecting together the values that constitute the complex value (i.e. the particular `street, number` and `postal code` values).

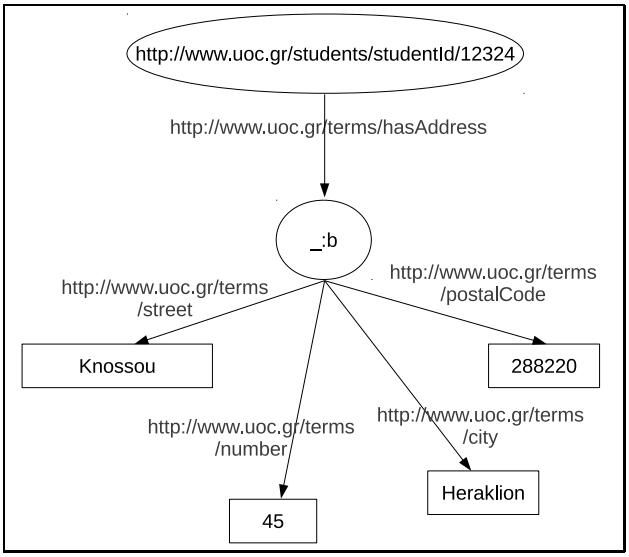

Figure 1.6: Examples of blank nodes with complex attributes

Thus, the information is described in a multi-component structure and we use bnodes to express the existence of the information. Also in RDF, we need to describe groups of things, for example several authors of a book or some students of a team. The container is a structural concept in RDF data model. There are three types of containers: the Bag (group of resources or literals), the Sequence (group of resources or literals, where their order is significant) and the Alternative (group of resources or literals that are alternatives).

These structures can be described with blank nodes, like Figure 1.7, where the characters of a book are given through a blank node.

http://www.library.com/bookids/3243	hasTitle	"From Whom The Bell Tolls"
http://www.library.com/bookids/3243	hasCharacters	_:chars
_:chars	rdf:type	rdf:Bag
_:chars	hasCharacter	"Robert Jordan"
_:chars	hasCharacter	"Pilar"
_:chars	hasCharacter	"Pablo"
_:chars	hasCharacter	"Maria"

Figure 1.7: Example of blank nodes with a Bag container

- Blank nodes have the capability to describe the Refication.
  Refication or else provenance is used to describe other RDF statements using the RDF format, for instance to record information about when statements were made, who made them or other similar information. For example, we have an RDF triple as:

```
exproducts:No001  exterms:weight  "2.4"^^xsd:decimal
```

Then we get the refication of the triple in a graph, whose triples are shown below:

```
_:b    rdf:type        rdf:Statement
_:b    rdf:subject     exproducts:No001
_:b    rdf:predicate   exterms:weight
_:b    rdf:object      "2.4"^^xsd:decimal
```

The blank node $\_:b$ is intended to refer to the original (first) triple and it is called, rather confusingly, a reified triple. We can use another triple to describe the information about the original triple, like the following:

```
exstore:No010    exterms:publish  _:b
```

In other words, the statement is itself an object that can be talked about in RDF, we can associate information to this very basic atom of data, such as who made this particular statement and when. The so-called provenance trails are used in many fields of science and business to support the needs of capturing, processing, presenting and preserving data in the digital object's life cycle.

Figure 1.8 shows a small provenance trail with two subsequent events.

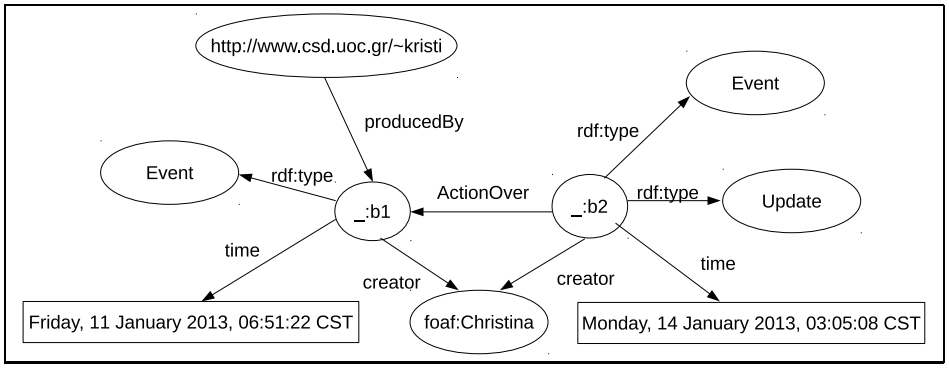

Figure 1.8: Example provenance trail with blank nodes

- Blank nodes can hide the Unexposible Information
  In many senses the publishers may not want to expose their data completely, so the blank nodes can help them to shield some sensitive information. For example, if a shopping center wants to publish some shopping information, we can replace the real costumer's identity with the blank node:

```
_:a1    exterms:buy         exproduct:No001
_:a1    exterms:buyTime     "2011-6-11"^^sxd:date
```

  Thus the browsers outside just only can get the information about the shopping of the shopping center, but can not know any other information about the identity of the costumer. The blank nodes actually protected the inner information in a good manner.

- Blank nodes can express the "Multi-relationship"
  The idea behind this functionality is that RDF can only describe binary relationships directly like $p(s, o)$. As to multi-relationship $p(s_1, s_2, .., s_n)$ the RDF must express this using an indirected form with the help of bnodes. We do it by choosing one participant (s_1) as the subject of the relationship p and a bnode as the object, then we create a group of relationships $p_2, p_3, .., p_n$ to express the relationships and the participants $s_2, s_3, .., s_n$. Figure 1.6 also describes the "multi-relationship" between a person and his address. As depicted there, the blank node is used to connect all the participants like a bridge.

As there have been developed various SW frameworks, there are different parsers that treat blank nodes differently. Some parsers use methods for automatically assigning URIs (skolemization), which complicates things further. In first-order logic, Skolemization is way of removing existential quantifiers from a formula in prenex normal form (a chain of quantifiers followed by a quantifier-free formula). The central idea of Skolemization is to replace existential quantified variables for "fresh" constants that are not used in the original formula. When the original formula does not have universal quantifiers, only constants are needed in the Skolemization process. Since only existential quantifiers are found in simple

RDF graphs, we need only to talk about Skolem constants. However, if Skolemization was used to study the satisfiability of logical formulae in more expressive languages (e.g. OWL), Skolem functions would be needed. Consequently, in terms of RDF, Skolemization refers to replacing existential variables with unique constants or simply a way of assigning URIs to blank nodes.
Other parsers just assign local identifiers/ labels in a systematical way. This labelling is based on the explicit blank node labels or the order of appearance (serialization order) of the blank nodes inside the imported set of triples or a combination of both. All the above methods are preferable than a completely arbitrary labelling.

1.3.3 Blank Nodes in published data

In this section we give some information on the use of blank nodes in RDF data published on the Web, according to [21]. These information were collected over a domain-agnostic sample of RDF data containing 965 MB of unique triples.

Looking at terms in the data-level position of triples, they found that the 57.8% of the unique terms were blank nodes, and only 32.2% were URIs, and 10% were literals. Each blank node had on average 5.2 data-level occurrences. Each blank node occurred, on average, 0.99 times in the object position of a non-rdf:type triple, with 3.1 MB blank nodes (1.9of all blank nodes) not occurring in the object position. Conversely, each blank node occurred on average 4.2 times in the subject position of a triple, with 0.04% not occurring in the subject position. Thus, we get that (i) blank nodes are prevalent on the Web; (ii) most blank nodes appear in both the subject and object position, but occur most prevalently in the former. Both their functionality and possibly the tree-based RDF/XML syntax can verdict in favour of these results.

#domain	bnodes	%bnodes	LOD?
hi5.com	14,409,539	87.5%	no
livejournal.com	8,892,569	58.0%	no
ontologycentral.com	2,882,803	86.0%	no
opiumfield.com	1,979,915	17.4%	no
freebase.com	1,109,485	15.6%	yes
vox.com	843,503	58.0%	no
rdfabout.com	464,797	41.7%	yes
opencalais.com	160,441	44.9%	yes
soton.ac.uk	117,390	19.1%	yes
bbc.co.uk	101,899	7.4%	yes

Figure 1.9: Top publishers of blank nodes in the corpus

Moreover, Table 1.9 lists the top ten domains in terms of publishing unique blank nodes found in their corpus. “LOD?” indicates whether the domain features in the LOD cloud diagram. Of the 783 domains contributing to our corpus, 345 (44.1%) did not publish any blank nodes. The average percentage of unique terms which were blank nodes for each domain was 7.5%, indicating that although a small number of high-volume domains publish many blank nodes, many other domains publish blank nodes more infrequently. The analogous figure including only those domains appearing in the LOD cloud diagram was 6.1%.

Treewidth	#components
1	518,831
2	8,134
3	208
4	99
5	23
6	-
7	1

Figure 1.10: Tree Width distribution

Regarding the structure of the blank nodes inside the graphs of the published data, we get the following information. They found 23% of all documents containing blank nodes and a 9% of all documents contained "non-reflexive" blank triples (triples with only one blank node). As for the treewidth they give Table 1.10. Notably, 98.4% of the components are acyclical, but a significant number are cyclical (treewidth greater than 1).

1.3.4 The future of blank nodes

Recent bibliography ([9], [21]) propose the reduction of blank nodes in the RDF graphs by clearing all the blank nodes that represent redundant information and the blank nodes which can be mapped into some URI references. This would help to make leaner and cleaner RDF graphs.

In Linked Data publication, it is not encouraged to describe information using blank nodes, but the common mechanisms of publishing Linked Data cannot avoid the usage of blank nodes, as it gives the developers the aforementioned conveniences.

Discussion on blank nodes is still open, but as the amount of published data grows rapidly, a consensus is very much needed. At the same time, because of the absence of an undisputed solution, the problems of blank nodes are an inevitable reality and research, like this thesis, is orientated on their solution.

1.4 Managing the evolution of Knowledge Bases

As already mentioned, Semantic Web is an evolving extension of the World Wide Web. This means that a lot of data need to be stored or exchanged over the network. Three kinds of services are exploited to cope with those needs:

- Versioning services are used to control multiple versions of the same unit of information.
- Synchronization services are used to keep remote data consistent (i.e. both of them contain the same information although something changed at one side).
- Replication services are used to ensure consistency between redundant resources (stored in multiple storage devices)

1.4.1 Versioning Services

Version control is the management of multiple revisions of the same unit of information. It is commonly used in engineering and software development to manage ongoing development of digital documents like application source code and other critical information that may be worked on by a team of people. Changes to these documents are identified by incrementing an associated number or letter code, termed the "revision number", "revision level", or simply "revision" and associated historically with the person making the change. A simple form of revision control, for example, has the initial issue of a drawing assigned the revision number "1". When the first change is made, the revision number is incremented to "2" and so on.

In computer software engineering, revision control is any practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and configuration files as well as source code. Most revision control software can use delta encoding, which retains only the differences between successive versions of files. This allows more efficient storage of many different versions of files.

Delta encoding is a way of storing or transmitting data in the form of differences between sequential data rather than complete files. Delta encoding is sometimes called delta compression, particularly where archival histories of changes are required.

The differences are recorded in discrete files called "deltas" or "diffs". Because changes are often small, delta encoding greatly reduces data redundancy. Collections of unique deltas are substantially more space-efficient than their non-encoded equivalents. From a logical point of view the difference between two data values is the information required to obtain one value from the other. The difference between identical values (under some equivalence) is often called 0 or the neutral element. A good delta should be minimal, or ambiguous unless one element of a pair is present.

A delta can be defined in two ways, symmetric delta and directed delta. A symmetric delta can be expressed as: $\Delta(v_1, v_2) = (v_1 \ v_2) \cup (v_2 \ v_1)$, where v_1 and v_2 represent two successive versions.

A directed delta, also called a change, is a sequence of (elementary) change operations which, when applied to one version v_1, yield another version v_2 (note the correspondence to transaction logs in databases).

In delta encoded transmission over a network, where only a single copy of the file is available at each end of the communication channel, special error control codes are used to detect which parts of the file have changed since its previous version.

The nature of the data to be encoded influences the effectiveness of a particular compression algorithm. Delta encoding performs best when data has small or constant variation.

1.4.2 Synchronization Services

In computer science, synchronization refers to one of two distinct, but related concepts:

- Process synchronization refers to the idea that multiple processes are to join up or handshake at a certain point, so as to reach an agreement or commit to a certain sequence of action.

- Data synchronization refers to the idea of keeping multiple copies of a dataset in coherence with one another, or to maintain data integrity. Process synchronization primitives are commonly used to implement data synchronization.

At this thesis we study the problem of data synchronization. Data synchronization is the process of establishing consistency among data on remote sources and the continuous harmonization of the data over time. It is fundamental to a wide variety of applications, including file synchronization [32], Personal Digital Assistant synchronization [2], and Public Key Server synchronization.

Several theoretical models of data synchronization exist in the research literature. The models are classified based on how they consider the data to be synchronized. Some models consider the data to be unordered while others consider the data to be ordered.

The problem of synchronizing **unordered** data (also known as the set reconciliation problem) is modelled as an attempt to compute the symmetric difference $S_A \oplus S_B = (S_A - S_B) \cup (S_B - S_A)$ between two remote sets S_A and S_B [23]. Some solutions to this problem are typified by:

- *Wholesale transfer.* In this case all data is transferred to one host for a local comparison.

- *Timestamp synchronization.* In this case all changes to the data are marked with timestamps. Synchronization proceeds by transferring all data with a timestamp later than the previous synchronization.

- *Mathematical synchronization.* In this case data are treated as mathematical objects and synchronization corresponds to a mathematical process.

Considering **ordered** data two remote strings σ_A and σ_B need to be reconcilied. Typically, it is assumed that these strings differ by up to a fixed number of edits (i.e. character insertions, deletions, or modifications). Some solution approaches to this problem include:

- shingling - splitting the strings into shingles in order to reduce this problem into an unordered synchronization problem.[8]

- synchronizing files and directories from one location to another while minimizing data transfer using delta encoding.

1.4.3 Replication Services

Replication is one of the most important topics in the area of distributed systems. It involves sharing information so as to ensure consistency between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance, or accessibility.

There are two kinds of replication:

- data replication if the same data is stored on multiple storage devices
- computation replication if the same computing task is executed many times

In the context of this thesis we focus on data replication. These processes are passive and operate only to maintain the stored data, reply to read requests, and apply updates.

Knowledge Base replication can be seen as an evolution of database replication. Database replication can be used on many database management systems usually with a master/slave relationship between the original and the copies. The master logs the updates, which then ripple through to the slaves. The slave outputs a message stating that it has received the update successfully, thus allowing the sending (and potentially resending until successfully applied) of subsequent updates. Multi-master replication, where updates can be submitted to any database node, and then ripple through to other servers, is often desired, but introduces substantially increased costs and complexity which may make it impractical in many situations. The most common challenge that exists in multi-master replication is transactional conflict prevention or resolution. Most synchronous or eager replication do conflict prevention, while asynchronous solutions have to do conflict resolution. For instance, if a record is changed on two nodes simultaneously, an eager replication system would detect the conflict before confirming the commit and abort one of the transactions. A lazy replication system would allow both transactions to commit and run a conflict resolution during resynchronization. The resolution of such a conflict may be based on a time-stamp of the transaction, on the hierarchy of the origin nodes or on much more complex logic, which decides consistently on all nodes.

In terms of the mobile domain it is very unlikely that an application will access all of them online, since mobile network connectivity is not always available for reasonable prices. To overcome this problem,and to decrease response times, data from remote sources can be replicated locally, and applications can operate on these local copies. However, it is not practical to duplicate several billions of RDF triples to a mobile device with limited computing power and memory capacity, and often this is not required at all for a specific application.

The replication of data from external sources, which can be selected based on the specific application context, is relatively straightforward for read-only data.

1.5 Motivation for comparing of RDF KBs with blank nodes

*RDF Delta*s can be employed to aid humans understand the evolution of knowledge, and to reduce the amount of data that need to be exchanged and managed over the network in order to build Semantic Web synchronization [33, 4], versioning [18, 19, 4, 11, 36] and replication [30] services.

Although RDF Knowledge Bases can be serialized in various text formats, a straightforward application of existing version control systems for software code (such as RCS and CVS) or for XML data (such as [22], [12] and [10]) is not a viable solution for computing RDF Deltas. This is mainly due to the fact that

RDF Knowledge Bases essentially represent graphs which (a) may feature several possible serializations (since there is no notion of edge ordering) and (b) can be enriched with semantics of a particular specification (also including inferred triples). For these reasons several non text-based tools have been developed for comparing graphs produced autonomously on the Semantic Web. We are going to refer later to examples of such tools. However, existing RDF differential tools have not yet focused enough on the size of the produced deltas, a very important aspect for building versioning services over Semantic Web repositories. [37] focuses on applying various differential functions that minimize the delta size, but they treat the blank nodes as named nodes.

As we have previously mentioned the existence of blank nodes makes the whole procedure much more complex. The prevalence of the blank nodes becomes clear through empirical study on the published data. Looking at terms in the data-level position they found that 57.8% of the unique terms were blank nodes. Indicatively, two domains of their corpus with a high number of publishing unique blank nodes are the following (a) the "hi5.com foaf" domain consisting of 87.5% of blank nodes and (b) the "opencalais.com" domain, which is part of LOD (Linked Open Data) cloud, with 44.9% of blank nodes. The authors also state that the inability to match blank nodes increases the delta size and does not assist in detecting the changes between subsequent versions of a Knowledge Base.

Previous works on comparing RDF Knowledge Bases have not elaborated on this issue thoroughly. There are works (e.g. [36, 37]) proposing differential functions that yield reduced in size deltas (in certain cases) but treat blank nodes as named nodes. Other works and systems (specifically Jena [8]) focus only on deciding whether two KBs that contain blank nodes are equivalent or not, and do not offer any delta size saving for the case where the involved KBs are not equivalent. There are other works that compare non equivalent KBs that neither aim directly at reducing the delta size nor treat the blank nodes for all the cases. [4] is able to match blank nodes only if they have functional term labels. [35] creates an identity for each blank node but is only restricted in finding the accurately same blank nodes. [26] proposes a blank node matching that presupposes that blank nodes are part of uniquely identified triples.

In brief, and to the best of our knowledge, our work is the first one that attempts to establish a blank node mapping:

- for reducing the delta size for the case of equivalent and not equivalent KBs
- that focuses on minimizing the delta size
- that can be applied in all the cases of KBs with blank nodes

Note that finding such a mapping can be considered as a preprocessing step, a task that is carried out before a differential function (like those described in [30, 35, 27, 25, 19, 36]) is applied.

1.5.1 Motivating Scenarios

At this subsection we are going to give a motivating scenario that will be used in the rest of this work.

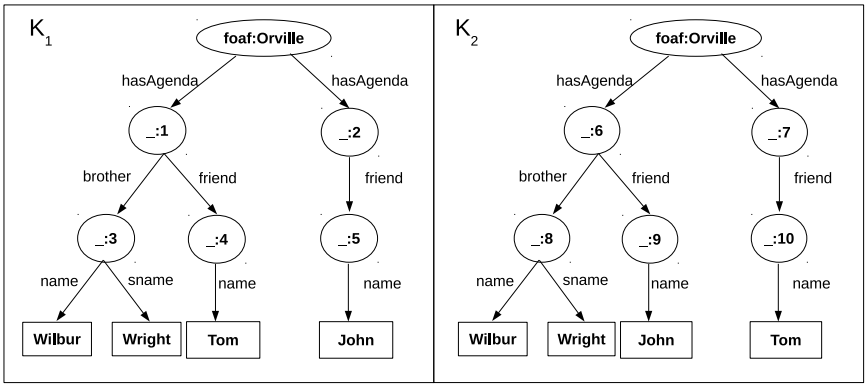

Figure 1.11: Two Knowledge Bases with directly connected blank nodes

Consider the Knowledge Bases K_1 and K_2 in Figure 1.11. K_2 can be considered as a subsequent updated version of K_1. It is clear that all the triples of the two graphs contain at least one blank node. As a result, if we do not apply any blank node matching, then all the triples of the first graph are considered different from the triples of the second graph. The delta will contain 18 change operations.

On the other hand, if we match the blank nodes of the first graph with the blank nodes of the second graph, delta can be reduced. Table 1.1 shows different matching options and the exported delta for each case.

Blank Node Matching	Delta (Deleted triples)	Delta (Added triples)	Delta Size
$\emptyset$	$Del(foaf:Orville, hasAgenda, \_:1)$ $Del(foaf:Orville, hasAgenda, \_:2)$ $Del(\_:1, brother, \_:3)$ $Del(\_:1, friend, \_:4)$ $Del(\_:2, friend, \_:5)$ $Del(\_:3, name, Wilbur)$ $Del(\_:3, sname, Wright)$ $Del(\_:4, name, Tom)$ $Del(\_:5, name, John)$	$Add(foaf:Orville, hasAgenda, \_:6)$ $Add(foaf:Orville, hasAgenda, \_:7)$ $Add(\_:6, brother, \_:8)$ $Add(\_:6, friend, \_:9)$ $Add(\_:7, friend, \_:10)$ $Add(\_:8, name, Wilbur)$ $Add(\_:8, sname, Wright)$ $Add(\_:9, name, John)$ $Add(\_:10, name, Tom)$	18
$(\_:1 -> \_:6)$ $(\_:2 -> \_:7)$ $(\_:3 -> \_:8)$ $(\_:4 -> \_:9)$ $(\_:5 -> \_:10)$	$Del(\_:3, name, Wilbur)$ $Del(\_:4, name, Tom)$ $Del(5, name, John)$	$Add(\_:4, name, John)$ $Add(5, name, Tom)$	4
$(\_:1 -> \_:8)$ $(\_:2 -> \_:10)$ $(\_:3 -> \_:6)$ $(\_:4 -> \_:7)$ $(\_:5 -> \_:9)$	$Del(foaf:Orville, hasAgenda, \_:1)$ $Del(foaf:Orville, hasAgenda, \_:2)$ $Del(\_:1, brother, \_:3)$ $Del(\_:1, friend, \_:4)$ $Del(\_:2, friend, \_:5)$ $Del(\_:3, name, Wilbur)$ $Del(\_:3, sname, Wright)$ $Del(\_:4, name, Tom)$ $Del(5, name, John)$	$Add(foaf:Orville, hasAgenda, \_:3)$ $Add(foaf:Orville, hasAgenda, \_:4)$ $Add(\_:3, brother, \_:1)$ $Add(\_:3, friend, \_:5)$ $Add(\_:4, friend, \_:2)$ $Add(\_:1, name, Wilbur)$ $Add(\_:1, sname, Wright)$ $Add(\_:5, name, John)$ $Add(2, name, Tom)$	18
$(\_:1 -> \_:7)$ $(\_:2 -> \_:6)$ $(\_:3 -> \_:8)$ $(\_:4 -> \_:10)$ $(\_:5 -> \_:9)$	$Del(\_:1, brother, \_:3)$	$Add(2, brother, 5)$	2

Table 1.1: Delta on different matching of the blank nodes

The third row of the Table shows the worse case, which can come from a completely random matching of the blank nodes. In such a case the delta size equals to that of no blank node matching. The second row has a matching that is more physical and coherent with the serialization order. However, the third

matching of the fourth row gives the minimum delta.

In the next chapters we are going to focus on how we should theoretically formulate the problem so as to always get the blank node matching with the minimum delta and then comprehend the difficulties that arouse in the practical application of this problem.

1.6 Contributions

In a nutshell the main contributions of this thesis are:

This work focuses on defining theoretically the blank node matching problem, as the problem of mapping the blank nodes from one graph to another in an optimal way. The optimality refers to the size of the produced delta (the number of triples to delete or add to make the graphs equivalent) considering the mapping of their blank nodes. We prove that finding the optimal blank node mapping is NP-Hard in the general case by framing it in terms of the sub-graph isomorphism problem. When the graphs do not contain directly connected blank nodes (no triples with more than one blank nodes), we show that the polynomial(cubic) Hungarian algorithm can be used directly to find the optimal bnode mapping.

When the blank nodes are connected (there are triples with more than one blank nodes), we propose to use polynomial heuristics algorithms returning approximate solutions. One algorithm uses the Hungarian algorithm but considers blank nodes as variables with no cost in matching. For making the application of our method feasible also to large Knowledge Bases we present a signature-based algorithm with n-logn complexity. This algorithm computes and orders signatures (encoding edges of blank-nodes) to find blank nodes with similar neighbourhoods.

The experimental results over real and synthetic datasets showed that the proposed algorithms significantly reduce the sizes of the produced deltas, while the time required is affordable (just indicatively the $n \log n$ algorithm requires a few seconds for KBs with up to 150,000 bnodes). We provide comparative results regarding their time efficiency, their potential for delta reduction and equivalence detection, their scalability and their deviation from the optimal solution.

1.7 Organization of the thesis

The rest of this work is organized as follows. Chapter **??** gives an introduction on the field of comparing RDF KBs and on blank nodes by discussing on change operations, *RDF Delta*s and RDF KBs with blank nodes. Chapter 3.4 elaborates on the problem of finding the optimal blank node mapping and proposes approximate bnode matching algorithms. Finally, Chapter **??** reports experimental results, discusses the applicability of the method at the presence of inference rules and various semantics. Chapter **??** gives the related work and finally, Chapter **??** concludes the work and identifies issues for further research.

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

- Blank Node Matching and RDF/S Comparison Functions, $YannisTzitzikas$, $ChristinaLantzaki$ and $DimitrisZeginis$, ISWC 2012

- Demonstrating BlankNode Matching and RDF/S Comparison Functions, *ChristinaLantzaki*, *YannisTzitzikas* and *DimitrisZeginis*, DemoPaper ISWC 2012

Chapter 2

Related Work

This section examines state of the art tools with the same or similar orientation and points out their differences in relation to our work.

2.1 Introduction

Several non text-based tools have been recently developed for comparing RDF graphs produced autonomously on the SW as for example:

- Ontoview [19]. Is an ontology management system, able to compare two ontology versions and highlight their differences.
- PromptDiff [27, 28, 25]. Is an ontology-versioning environment, that includes a version-comparison algorithm (based on heuristic matchers)
- SemVersion [35]. Proposes two Diff algorithms one *structure-based* and one *semantic-aware*
- RDF_Utils [13]. Introduce the notion of RDF molecules as the finest components to be used when comparing RDF graphs.
- CWM of w3c [4]. Is a general-purpose semantic web data processing tool which can compare two RDF files. It uses a functional or inverse functional properties to identify a blank nodes.
- Jena [1]. Is a Java framework for building Semantic Web applications. It provides a tool for checking isomorphism between two RDF graphs.
- Powl [3]. Is a web based ontology management tool that tracks the editing actions that are made using the system.

Existing RDF comparison tools have neither focused on the size of the produced Deltas, nor in the blank nodes matching, two very important aspects for building versioning services over SW repositories. Furthermore, the output of these tools is exploited by humans, and thus an intuitive presentation of the comparison results (and other related issues) has received considerable attention.

[1] http://jena.sourceforge.net/

Finally, tracking the evolution of ontologies when changes are preformed in more controlled environments (e.g. collaborative authoring tools) has been addressed in [20, 29, 38].

It follows an analysis over the most important tools and some annotations about the way they treat blank nodes.

2.2 Ontoview

OntoView [19] is a web-based system[2] inspired by CVS [6] that helps users to manage changes in ontologies. OntoView stores the contents of the versions, metadata, conceptual relations between constructs in the ontologies and the transformations between them. The internal version management is partly based on change specifications and the versions of ontologies themselves, but also uses additional human input about the meta-data and types of changes (as described below). It allows users to differentiate between ontologies at a conceptual level and to export the differences as adaptations or transformations.

Two types of change are distinguished. There can be changes in the logical definition of a concept which are not meant to change the concept, and, the other way around, a concept can change without a change in its logical definition. An example of the first case is attaching a slot "$fuel - type$" to a class "Car". Both class-definitions still refer to the same ontological concept, but in the second version it is described more extensively. On the other hand, a natural language definition of a concept might change, e.g. the new definition of "*chair*" might exclude "$reclining - chairs$" without a logical change of the concept. The former kind of change is referred in the literature as explication change, while the latter conceptual change. Since at the syntactic level, the same data can be the result of any of these types of change, more (human) input is needed to classify the change.

OntoView provides a web "diff" view for comparing two versions of an ontology (see Figure 2.1) at a structural level. The comparison function is inspired by UNIX diff, but the implementation is quite different. The UNIX diff compares file version at line-level, highlighting the lines that textually differ in two versions. Ontoview, in contrast, compares versions of ontologies at a structural level, showing which definitions or properties are changed. So as to produce such meaningful difference for ontologies (where there is no inherent ordering), the ontology is canonicalized at the syntactic level before being given to the diff tool.

The comparison function used by the Ontoview distinguishes between the following types of change:

- Non-logical change (conceptual change). A change at the natural language defintion. e.g. changes in the rdfs:label of a concept or property, or in a comment inside a definition.
- Logical definition change (explication change). This is a change in the definition of a concept or property that affects its formal semantics. Examples of such changes are alterations of subClassOf, domain, or range statements. Additions or deletions of local property restrictions in a class are also logical changes.

[2] currently there is no link working for this tool

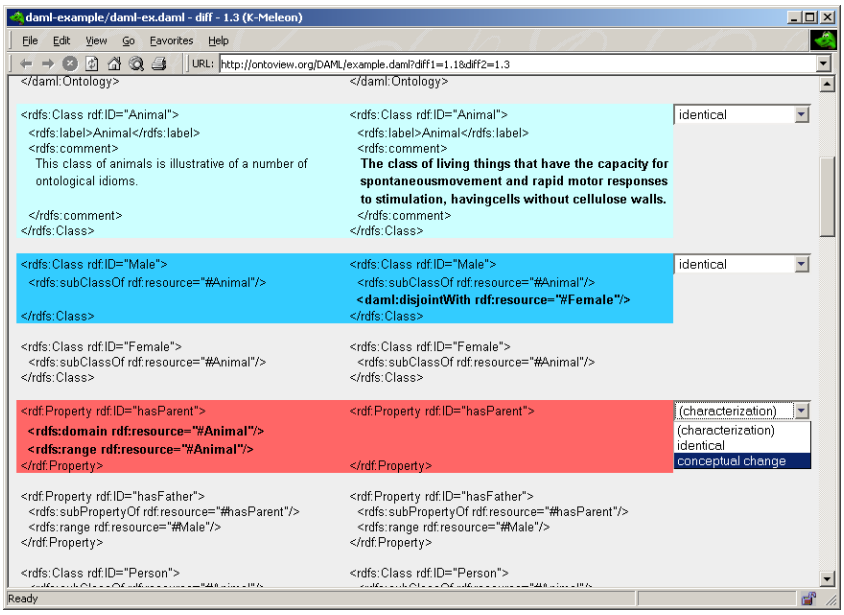

Figure 2.1: Comparing two ontologies in OntoView

- Identifier change. This is the case when a concept or property is given a new identifier, i.e. a renaming.
- Addition of definitions.
- Deletion of definitions.

Most of these changes can be detected completely automatically, except for the identifier change, because this change is not distinguishable from a subsequent deletion and addition of a simple definition. In this case, the system uses the location of the definition in the file as a heuristic to determine whether it is an identifier change or not.

2.2.1 Rules for Changes

The algorithm uses the fact that the RDF data model underlies a number of popular ontology languages, including RDF Schema and DAML+OIL. First, it splits the document at the first level of the XML document. This groups the statements by their intended "definition". The definitions are then parsed into RDF triples, which results in a set of small graphs. Each of these graphs represent a specific definition of a concept or a property, and each graph can be identified with the identifier of the concept or the property that it represents.

Then, the algorithm locates for each graph in the new version the corresponding graph in the previous version of the ontology. Those sets of graphs are then checked according to a number of rules. Those rules specify the "required" changes in the triples set for a specific type of change.

The rules have the following format:

IF exist:old

$\langle A, Y, Z \rangle *$
exist:new
$\langle X, Y, Z \rangle *$
not-exist:new
$\langle X, Y, Z \rangle *$
THEN change-type A

They specify a set of triples that should exist in one specific version, and a set that should not exist in another version (or the other way round) to signal a specific type of change. With this rule mechanism, the tool is able to specify almost all types of changes, apart from the "identifier change".

The rules are specific for a particular RDF-based ontology language because they encode the interpretation of the semantics of the language for which they are intended. For another language other rules would have been necessary to specify other differences in interpretation. The semantics of the language are thus encoded in the rules. The mechanism relies on the "materialization" of all rdf:type statements that are encoded in the ontology. In other words, the closure of the RDF triples according to the used ontology language has to be computed.

Regarding the blank nodes, we realize that they can be determined as "identifier changes" through their blank node identifiers. As we have already mentioned the proposed rule mechanism is not applicable to them. The blank node matching is described and applied in an indirect way, by using their location in the file as a heuristic to determine their matching (or not) with a blank node in the second file. No more information are given on this issue.

2.3 Promptdiff

`Prompt` is an ontology-management framework that brings together different ontology-management tools and provides an infrastructure for other related tools. The key components of the framework are: `iPrompt` an interactive ontology-merging tool, `AnchorPrompt` a graph-based tool for finding related concepts in different ontologies, `Protege` a tool that provides access to a library of ontologies, giving users meta-information about an ontology and `PromptDiff`.

PromptDiff is an ontology-versioning tool that determines what has changed between two versions. It finds a structural diff between versions i.e. compares the structure of ontology versions and not their text serialization.

Figure 2.2 shows the overall architecture of the PromptDiff ontology-versioning system. Two versions of an ontology, v_1 and v_2, are inputs to the system. The heuristic-based algorithm for comparing ontology versions analyzes the two versions and automatically produces a diff between v_1 and v_2 called a structural diff (Figure 2.3). The post-processing module uses the diff to identify complex changes. The results are presented to the user through the intuitive interface. The user then has the option of accepting or rejecting changes and these actions are reflected in the updated diff.

Given two versions of an ontology O, v_1 and v_2, a structural diff between v_1 and v_2, is a set of pairs $\langle r_1, r_2 \rangle$ where:

- $r_1 \in v_1$ or $r_1 = null$, $r_2 \in v_2$ or $r_2 = null$

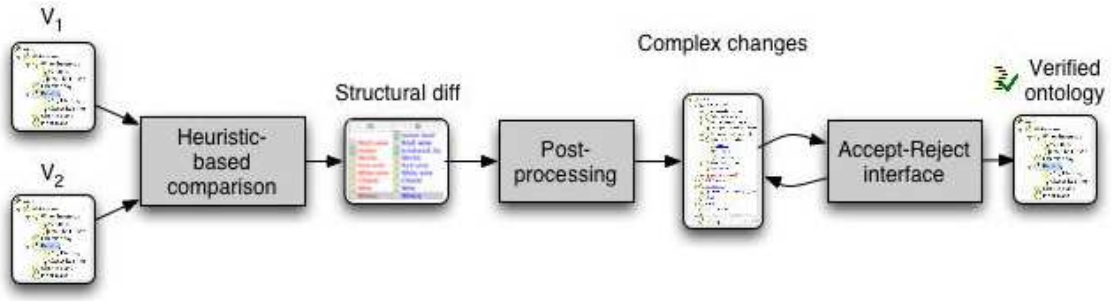

Figure 2.2: The architecture of the PromptDiff

f1	f2	renamed	operation	map level
	tannin level	No	Add	
Blush wine	Rosé wine	Yes	Map	Isomorphic
maker	produced_by	Yes	Map	Isomorphic
Chianti	Chianti	No	Map	Isomorphic
Merlot	Merlot	No	Map	Changed
Red wine	Red wine	No	Map	Changed
White wine	White wine	No	Map	Changed
Wine	Wine	No	Map	Isomorphic
Winery	Winery	No	Map	Unchanged

Figure 2.3: The structural diff showing the difference between two versions

- r_2 is an image of r_1 (matches r_1), that is, r_1 became r_2. If r_1 or r_2 is null, then we say that r_2 or r_1 respectively does not have a match.
- Each resource from v_1 and v_2 appears in at least one pair.
- For any resource r_1, if there is at least one pair containing r_1, where $r_2 \neq null$, then there is no pair containing r_1 where $r_2 = null$. The same is true for r_2.

The PromptDiff algorithm consists of two parts: (1) an extensible set of heuristic matchers and (2) a fixed-point algorithm to combine the results of the matchers to produce a structural diff between two versions. Each matcher employs a small number of structural properties of the ontologies to produce matches. The fixed-point step invokes the matchers repeatedly, feeding the results of one matcher into the others, until they produce no more changes in the diff. Then the differences found by the algorithm are presented to the user who is responsible to accept or reject them.

2.3.1 Heuristic Matchers

The PromptDiff algorithm combines an arbitrary number of heuristic matchers, each of which looks for a particular property in the unmatched frames. The heuristic matchers compare two ontology versions looking for the following situations:

- **Resources of the same type with the same name.** In general, if $r_1 \in K_1$ and $r_2 \in K_2$ and r_1 and r_2 have the same name and type, then r_1 and r_2 match.

- **Single unmatched sibling.** In general, if $c_1 \in K_1$ and $c_2 \in K_2$, c_1 and c_2 match, and each of the classes has exactly one unmatched subclass, $subC_1$ and $subC_2$, respectively, then $subC_1$ and $subC_2$ match.

- **Siblings with the same suffixes or prefixes.** In general, if $c_1 \in K_1$ and $c_2 \in K_2$, c_1 and c_2 match, and the names of all subclasses of c_1 are the same as the names of all subclasses of c_2 except for a constant suffix or prefix, then the subclasses match.

- **Single unmatched Property.** In general, if $c_1 \in K_1$ and $c_2 \in K_2$, c_1 and c_2 match, and each of the classes has exactly one unmatched property, p_1 and p_2 respectively, and p_1 and p_2 have the same domain and range, then p_1 and p_2 match.

- **Unmatched inverse properties.** If a knowledge model allows definition of inverse relationships, then those relationships can be used to create matches as well. In general, if $p_1 \in K_1$ and $p_2 \in K_2$, p_1 and p_2 match, $invP_1$ and $invP_2$ are inverse properties for p_1 and p_2 respectively, and $invP_1$ and $invP_2$ are unmatched, then $invP_1$ and $invP_2$ match.

- **Split classes.** In general, if $c_0 \in K_1$ and $c_1 \in K_2$ and $c_2 \in K_2$, and for each instance of c_0, its image is an instance of either c_1 or c_2, then c_0 was split into c_1 and c_2. A similar matcher identifies classes that were merged.

The above heuristic matchers are mainly focusing on the matching of the nodes according to their structural difference or similarities. As a result, they can be applied for the matching of the blank nodes. However, no special attention is paid in their case.

2.4 Semversion

SemVersion [35] is a Java library for providing versioning facilities to RDF data. It is based on RDF/RDFS, so it can be used for any ontology language built or adapted to this data model.

Semversion offers an easy to use (and thus, integrate with) API that closely follows the usual functions and concepts of CVS [6]. To commit a new version, a user can either provide the complete contents of the version (which is an RDF model, i.e., simply a set of triples), or a *diff*, that is, the *change* that is to be applied on a preexisting version to create the new one.

At the implementation level (Figure 2.4), persistence is handled by RDF2Go [3], which provides common storage interfaces over triple- and quad-stores (SemVersion uses the abstraction of the latter), such as Jena [4], Sesame [5], YARS [6], NG4J [7], etc. SemVersion stores each version of an RDF model as a unique independent graph that contains the whole model.

[3] http://ontoware.org/projects/rdf2go/
[4] http://jena.sourceforge.net/
[5] http://www.openrdf.org/
[6] http://sw.deri.org/2004/06/yars/
[7] http://sites.wiwiss.fu-berlin.de/suhl/bizer/ng4j/

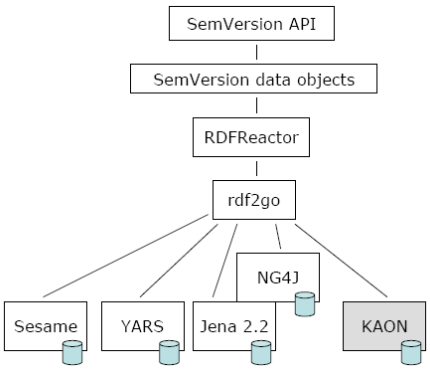

Figure 2.4: The Layered Architecture of SemVersion

Diffs serve two purposes: First, SemVersion allows to compute (structural and semantic) diffs between two arbitrary chosen models, to inform the user about changes. This allows collaborative ontology engineering. Second, diffs can be used in an update command to apply changes to a remotely stored model. When dealing with very large models, it might not be feasible (nor efficient) to transfer the complete model, if only a small fraction has changed.

Semversion provides three type of *diff*s analyzed at the next sections.

2.4.1 Set-based Diff

For versioning, the set-based diff is simply the set-theoretic difference of two RDF triple sets. Such diffs can be computed by simple set arithmetics for triple sets that contain only URIs and literals.

2.4.2 Structural Diff and Blank nodes

Without the presence of blank nodes, the set-based diff is the same as the structural diff. With blank nodes, the set-based diff considers all blank nodes to be different and reports all statements involving blank nodes both as added and as removed.

SemVersion also handles the problem of uniquely identifying blank nodes. Blank nodes cannot be globally identified, as they lack a URI, and this poses a challenge at diff algorithms. This is overcome by adding functional properties to each blank node leading to a URI, effectively treating them, from that point on, as normal nodes. This procedure is called *blank node enrichment*. Other tools that process the RDF data are expected not to remove this property, so this will survive the roundtrip "extract a version from the repository, manipulate it in some ontology editor, reinsert the changes at the repository to create a new version", so that SemVersion can understand whether two blank nodes are the same. If this URI is missing, then SemVersion treats the node as new (since creating a new node from an external tool would be missing this, of course).

However, blank nodes are matched only in the case where they participate in exactly the same triples. In the context of a system where changes arise between subsequent versions, this blank node matching technique is a bit restrictive, as no closest matching is applied.

2.4.3 Semantic Diff

The semantic difference has to take the semantics of the used ontology language into account.

An intuitive way to understand the concept of a sematic diff goes like this: Let's assume we use RDF Schema as our ontology language, and have two versions (K and K') of an RDFS ontology. Now, in order to compute the semantic RDFS diff, we use the closure of K ($C(K)$). Then we do the same for K' ($C(K')$). Now we calculate a structural (syntactical, set-based) diff on $C(K)$ and $C(K')$. This is not the same as the structural diff between K and K'. If the structural diff of two models is empty, then the semantic diff must also be empty. The inverse is not necessarily true: There might be two different RDF Knowledge Bases which encode the same semantic model, resulting in an empty semantic diff, but a nonempty structural diff.

2.5 x-RDF-3X

The work presented in presents an extension of the RDF-3x system that supports versioning and some other services as time-travel access and transactions on RDF databases. Versioning is achieved by maintaining versions of individual triples (updates are considered as pairs of insertions and deletions, so two timestamps fields are used, the created and deleted to denote the life of each triple version). The [created, deleted] interval is the lifespan of the triple version, where deleted has a null value for versions that are presently alive. The database state of a given point in time t can be reconstructed by returning all triples for which t falls into the corresponding lifespan interval. Ideally, timestamps reflect the commit order of transactions, but unfortunately the commit order is not known when inserting new data. In order to cope with this problem each transaction is assigned a write timestamp once it starts updating the differential index, and this timestamp is then used for all subsequent operations. Ideally the migration is performed at transaction commit only, which means that the timestamps perfectly reflect the commit order and need no further updates. Moreover, a transaction inventory is used, that tracks transaction ids, their begin and commit times (BOT and EOT), the version number used for each transaction, and the largest version number of all committed transactions (highCV #) at the commit time of a transaction (Figure 2.14). This inventory serves to efficiently decide if a transaction committed before another one. Also, it relates the relative time of transaction ordering and version numbering to wall clock times. This is needed for supporting time-travel queries and snapshot isolation. Regarding the experimental evaluation synthetic workloads were constructed based on real data (from the LibraryThing book-tagging web site) and the x-RDF-3x system was compared with two other systems: PostgreSQL and Jena. The results show that the first system was 3 times more space efficient than the others systems.

2.6

2.7 RDF_utils

RDF_utils is developed as part of the KnoBot project. KnoBot is an RDF-based content management system which stores all its data in a Jena Model. It is designed to allow decentralized exchange of information founded on trust relationships between individual persons/agents. While it is designed to maximally comply with standards and best practices it offers novel features (e.g. relevance based aggregation).

RDF_utils is a utility tool for dealing with RDF data, it provides the following features:

- **Leanify**: Remove redundant statements (and anonymous nodes) from rdf-graphs
- **Diff**: Show the difference between two rdf-graphs

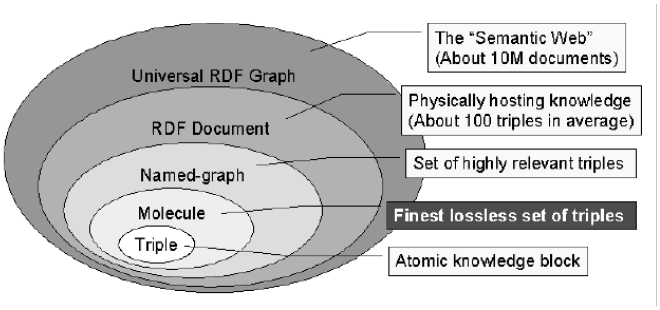

Figure 2.5: The granularity of the Semantic Web ranges from the universal graph to triple.

The main difference of this work to others is the choice of the level of granularity (Figure 2.5). The problem of granularity is well explained in [13]. RDF documents and named graphs are too coarse for some particular application needs, such as in tracking provenance of an RDF graph. In this case, the overlap of the graph at hand with other graphs is a key to identify its provenance. But a named graph can't be used to express an overlap, as it will generally contain irrelevant triples too, unless explicitly calculating the intersection. On the other hand, triple-level is too fine-grained, due to the case of blank nodes. For example, see the RDF graphs of Figure 2.6. The first one shows an unnamed resource (blank node) with surname 'Ding' and first name 'Li'. The second graph is identical, while the third described another 'Ding' person, in particular 'Zhongli Ding'. If the triple-based overlap was meant to be used, the first and the third graph would appear that they share a common triple, while in fact the triples describe different people. This is due to the lack of universal identity of blank nodes; their identity is only derived by the named resources or literals connected to them. Clearly, when blank nodes are involved, equality of triples can't reliably be used as identification of equal RDF content.

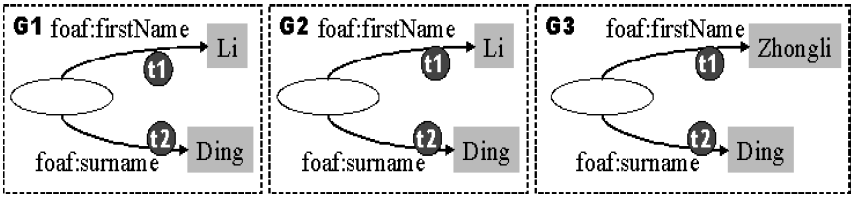

Figure 2.6: Three RDF graphs that show personal information from three sources. The first one asserts that a person who has first name 'Li' and surname 'Ding'.

In [14], the decomposition is defined as follows. An RDF graph decomposition consists of three elements (W, d, m): the background ontology W, the decompose operation $d(G, W)$ which breaks an RDF graph G into a set of sub-graphs $G* = G_1, G_2, ..., G_n$ using W, and the merge operation $m(G*, W)$ which combines all elements in $G*$ into the a unified RDF graph G' using W. In addition, a decomposition must be lossless such that
$for\ any\ RDFgraph\ G, G = m(d(G, W), W)$.

RDF molecules are defined as the finest and lossless subgraphs of a graph G according to a decomposition (W, d, m). Worth of note is that this concept is very similar to the notion of *Minimum Self-contained Graphs* (MSG), described in [34], one of the differences being that molecules also consider an arbitrary reasoning -the "background ontology"- while MSG deals only with RDF).

2.7.1 Labeling RDF Nodes

As mentioned by

2.8 RDF Sync

2.9 CWM of w3c

CWM is part of SWAP, a Semantic Web Application Platform. SWAP consists of tools and applications to manipulate RDF graphs much like traditional tools manipulate text files. CWM is a command-line tool, written in python, for processing RDF in both the standard XML encoding and an experimental encoding, Notation3 [5].

CWM offers a utility that allows the user to compute the delta between two Knowledge Bases and then to apply the delta on the first Knowledge Base to obtain the second.

The authors state that in case which all the nodes are names, computing the difference between two graphs is simple and straightforward:

If G_1 and G_2 are ground RDF graphs, then the ground graph delta of G_1 and G_2 is a pair of $(inserions, deletions)$ where $insertions$ is the set difference $G_2 - G_1$ and deletions is $G_1 - G_2$. This form of delta is reasonably economical: the storage cost is linear in the size of the difference between the graphs.

2.9.1 Patch file format

By analogy to the text diff, there is a need not only for a difference-finding algorithm, but for a patch file format. Such a format needs:

- A way to uniquely identify what is changing.
- A way to distinguish between the pieces added and those subtracted.

It is straightforward to pinpoint the parts of the Knowledge Base that have changed when all nodes are named, but less so in the presence of anonymous nodes. To identify what is changing, Notation3 expressions are used and three new terms are introduced. For example:

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

This one new property replacement can express any change. Deletions can be written {...} `diff:replacement` {} and additions can be written {} `diff:replacement` {...}.

The second alternative is very similar but involves two properties, one for inserting and one for deleting:

```
{ ?x bank:accountNo "1234578"}
    diff:deletion { ?x bank:balance 4000};
diff:insertion { ?x bank:balance 3575}.
```

The form using `diff:insertion` and `diff:deletion` is implemented in CWM.

2.9.2 Weak and Strong Deltas

CWM distinguishes two types of RDF deltas:

- **Weak delta**. Gives enough information to apply it to exactly the Knowledge Base it was computed from.
- **Strong delta**. Specifies the changes in a context independent manner. The difference is not in the format of the output but in the information a particular delta gives.

For example, if bank account numbers are globally unique, then a blank node that represents a bank account can be identified by a particular bank account. In OWL terms, if $bank : accountNumber$ is an $owl : InverseFunctionalProperty$, then the node must be the $owl : sameAs$ any other node with the same account number. In that case, the delta will be **strong**. If however, many accounts can have the same number, applying that delta to another knowledge base may inadvertently alter the wrong account. The delta is **weak**.

In order to produce strong deltas CWM uses the $owl : FunctionalProperty$ and $owl : InverseFunctionalProperty$ to assign labels to blank nodes in order to uniquely identify them. Strong deltas are provided if sufficient information can be found in the Web to fully label the input graphs.

It becomes clear that the blank nodes cannot be matched in the general case. The blank node matching is successfully built only in the case of ontologies with functional datatype properties, which are compatible in OWL/Full but not in OWL/DL, let alone in RDF.

2.10 Jena

Jena is a Semantic Web toolkit for Java programmers. The heart of the Semantic Web recommendations is the RDF Graph as a universal data structure. Jena similarly has the Graph as its core interface around which the other components are built.

Jena provides tools, including: a Java model/graph API, an RDF Parser (supporting an N-Triples filter), a query system based on RDQL , support classes for DAML+OIL ontologies and persistent/in-memory storage on BerkeleyDB or various other storage implementations. Due to its storage abstraction, Jena enables new storage subsystems to be integrated. To facilitate querying, Jena provides statement-centric methods for manipulating an RDF model as a set of RDF triples and resource-centric methods for manipulating an RDF model as a set of resources with properties, as well as built-in support for RDF containers. Jena contains Joseki RDF server, a server accepting SOAP and HTTP requests to query RDF resources. The latest version of Jena and Joseki support SPARQL.

Jena does not provide a mechanism to compare two RDF graphs and find the differences between them (i.e. compute the delta). But it provides a mechanism that decides whether two RDF graphs are isomorphic or not.

It follows the approach intoduced at [8] to decide isomorphism between graphs. A signature is created for each node (named or unnamed) assuming its position in the graph. Nodes that have the same signatures between the two graphs are matched. If all the nodes of the two graphs match then the graphs are isomorphic.

Although blank nodes are treated here in a more sophisticated and general way than the aforementioned papers, no attention is paid in the case of not isomorphic Knowledge Bases, as no delta can be computed.

2.11 pOWL

Powl, a web based ontology management tool. Its capabilities include parsing, storing, querying, manipulating, versioning, serving and serializing RDF and OWL knowledge bases for different target audiences.Powl is implemented in the web scripting language PHP.

Powls architecture consists of 4 stacked tiers, while trying to minimize dependencies and supplying clean interfaces between tiers. It consists of the following tiers:

- Powl store: SQL compatible relational database.
- RDFAPI, RDFSAPI, OWLAPI: layered APIs for handling RDF, RDFS and OWL.
- Powl API: containing classes and functions to build web applications on top of those APIs.

- User interface: a set of PHP pages combining widgets provided by Powl API for accessing (browsing, viewing, editing) model data in a Powl store.

To enable domain experts to collaboratively develop shared conceptualizations based on the Ontology Web Language a key requirement is to support a versioning strategy. In order to suport versioning, pOWL does provide a mechanism to compare Ontologies and find the differences between them, but supposes that all the changes were made through the pOWL platform and so they are tracked. One editing action by the user may be complex, but every editing action can be decomposed into smaller editing actions (Figure 2.7) and finally into adds and removes of RDF triples to or from the RDF model.

Powl enables rollback of every particular editing action by determining if the involved triples are still present (if added) or still missing (if removed). A parent action thus may only be rolled back if all sub-actions may be rolled back as well.

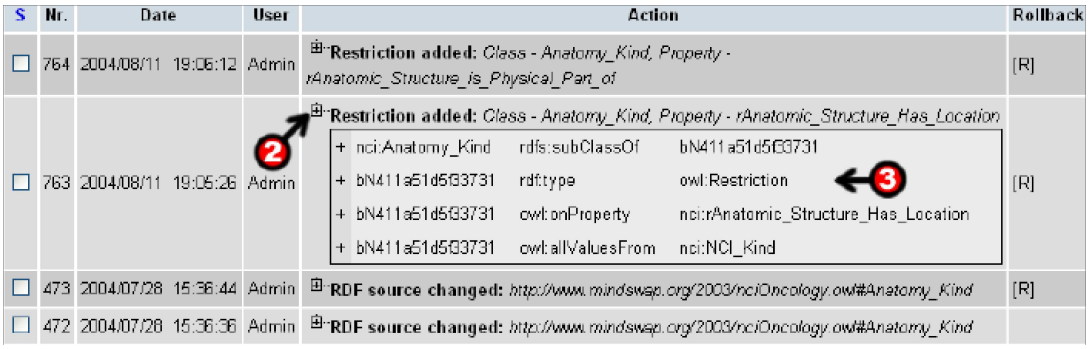

Figure 2.7: pOWL Versioning.

2.12 Summary Comparison

Table **??** presents a number of basic features which are then used for providing an overview of the functionality offered by each of the previously described systems.

Features	Description
Δ_e	Use Δ_e as differential function
Δ_c	Use Δ_c as differential function
Δ_d	Use Δ_d as differential function
Δ_{dc}	Use Δ_{dc} as differential function
Δ_{ed}	Use Δ_{ed} as differential function
$\mathcal{U}_p$	Use $\mathcal{U}_p$-semantics
$\mathcal{U}_{ir}$	Use $\mathcal{U}_{ir}$-semantics
Heuristic matchers	Use heuristic matchers to detect changes
Isomorphism	Decide isomprphism between two RDF graphs
Change Log	Maintain the sequence of applied changes

Table 2.1: Features of the comparison

The first five features correspond to the differential functions already introduced (i.e. Δ_e, Δ_c, Δ_d, Δ_{dc} and Δ_{ed}) and indicate if a system uses the comparison function or not. The next two features correspond to the two different change operations semantics introduced (i.e. $\mathcal{U}_p$ and $\mathcal{U}_{ir}$).

Heuristic matchers are a kind of rules that are used in order to decide if something has changes at a Knowledge Base. Unfortunately, the heuristic matchers are language specific i.e. depend on the semantics of the underlying language. For example the heuristic matchers that need to be used for RDF differ from those used for OWL.

The *Isomorphism* is the ability to decide whether two RDF graphs, which may contain blank nodes, have the same structure. The tools that implement this feature may not produce a delta as a result, but just decide isomorphism between the graphs.

The *Change Log* is a registry that records the actions that occurred in the versioning system, for example the steps taken to create a new version from an older one. If the actions are completely recorded, one could traverse this log and apply the actions in the order that they occurred and reach the same result. The change log is also useful for a user that wants to understand the changes made by someone else, to see the way they work and possibly spot errors.

Features	Systems							
	OntoView	**PromptDiff**	**SemVersion**	**RDF_Utils**	**CWM**	**Jena**	**Powl**	**SWKM**
Δ_e	no	no	yes	yes	yes	no	no	yes
Δ_c	no	no	yes	no	no	no	no	yes
Δ_d	no	no	no	no	no	no	no	yes
Δ_{dc}	no	no	no	no	no	no	no	yes
Δ_{ed}	no	no	no	no	no	no	no	yes
$\mathcal{U}_p$	no	no	yes	yes	yes	no	no	yes
$\mathcal{U}_{ir}$	no	no	no	no	no	no	no	yes
Heuristic matchers	yes	yes	no	no	no	no	no	no
Isomorphism	no	no	no	yes	yes	yes	no	not yet
Change Log	no	no	no	no	no	no	yes	no

Table 2.2: Features comparison of versioning systems and tools

The differential function Δ_e is utilized by SemVersion, RDF_Utils, CWM and SWKM, while Δ_c is used by SemVersion and SWKM. The other differential functions (i.e. Δ_d, Δ_{dc} and Δ_{ed}) are used only by SWKM. Considering the change operation semantic, $\mathcal{U}_p$ is used by SemVersion, RDF_Utils, CWM and SWKM, while $\mathcal{U}_{ir}$ is used only by SWKM. PromptDiff and OntoView detect changes using heuristic matches. RDF_Utils, CWM and Jena can detect if two RDF graphs are isomorphic. Finally, Powl uses a change log to keep track of the changes in order not to have to compute the delta.

Chapter 3

On reducing *RDF Delta* in KBs with blank nodes

3.1 Introduction

Versioning, Synchronization and Replication services need a method to compare two RDF KBs with blank nodes and then transform the first KB to the second (Figure 3.1). For this reason two modules are required:

- A differential function to report the differences between two RDF KBs
- A change operation semantics that indicates the way the differences must be applied to the first RDF KB to get the second one.

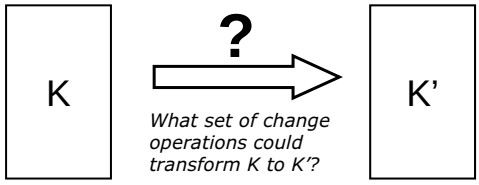

Figure 3.1: What set of change operations could transform K to K'?

Obviously, a differential function that yields the smallest in size result is preferred. Furthermore, a pair of (differential function & change operation semantics) must at least satisfy correctness when synchronizing remote Knowledge Bases.

3.2 Preliminaries

In this chapter we initially give some basic notation, that is followed in the rest of this work. Then we give the basic definition of RDF KBs and determine the differential functions and change operations.

3.2.1 Basic Notation

Let t be an RDF triple of the form (s, p, o), where s is called the *subject*, p the *predicate* and o the *object* of the triple. Let T be the set of all possible RDF triples

that can be constructed from an infinite set of URIs (for resources, classes and properties) as well as literals. In general, an RDF Knowledge Base can be seen as a finite subset K_1 of T , i.e. K ? T .

Another presentation of a Knowledge Base K_1 is as a directed labelled graph $G_1 = (N_1, R_1)$. The nodes of the graph are the URIs, the literals and the blank nodes that appear as subjects or objects in the triples of K, while the edges of the graph are labelled (with URIs) arcs that connect the corresponding nodes. We can partition the nodes of N into three sets $N = U_1?L_1?B_1$, where U_1 is the set of all URI nodes, L_1 is the set of all literals nodes, and B_1 is the set of all blank nodes of G.

3.2.2 RDF Knowledge Bases

Apart from the explicitly specified triples of a K, other triples can be inferred based on the RDF/S semantics [17]. For this reason, we introduce the notion of closure and reduction of RDF/S KBs.

The *closure* of a K, denoted by $C(K)$, contains all the triples that either are explicitly asserted or can be inferred from K by taking into account class or property assertions made by the associated RDFS schemas. Thus, we can consider that $C(K)$ is defined (and computed) by taking the reflexive and transitive closures of RDFS binary relations such as subClassof and type. It should be stressed that our work is orthogonal to the consequence operator of logic theories [15] actually employed to define the closure operator C. Specifically, if P denotes the powerset of all possible sets of triples of $\mathcal{T}$, then the closure operator can be defined as any function $C : P \rightarrow P$ that satisfies the following properties:

- $K \subseteq C(K)$ for all K, i.e. C is *extensive*
- If $K \subseteq K'$ then $C(K) \subseteq C(K')$, i.e. C is *monotonically increasing*
- $C(C(K)) = C(K)$ for all K, i.e. C is an *idempotent* function

If it holds $C(K) = K$, then we will call K *completed*. The elements of K will be called *explicit* triples, while the elements of $C(K) - K$ will be called *inferred*. We can now define an equivalence relation between two knowledge bases.

Def. 1 Two knowledge bases K and K' are *equivalent*, denoted by $K \sim K'$, iff $C(K) = C(K')$.

The *reduction* of a K, denoted by $R(K)$, is the smallest in size set of triples such that $C(R(K)) = C(K)$. Let Ψ denote the set of all knowledge bases that have a unique reduction. Independently of whether the reduction of a K is unique or not, we can characterize a K as (semantically) *redundancy free*, and we can write $RF(K) = True$ (or just $RF(K)$), if it does not contain explicit triples which can be inferred from K. Formally, K is redundancy free if there is not any proper subset K' of K (i.e. $K' \subset K$) such that $K \sim K'$. Figure 3.2 illustrates the above sets of triples ($R(K)$ is enclosed in a dashed box because it is not always unique).

It is worth noticing that the reduction of a K is not always unique. In general, uniqueness of the transitive reduction of a binary relation R is guaranteed only when R is antisymmetric and finite. Unfortunately, this is not the case of RDF/S KBs allowing cycles in the subsumption relations. For example, in Figure 3.3 we have $K \sim K_1 \sim K_2$, moreover $RF(K_1), RF(K_2)$, but $K_1 \neq K_2$.

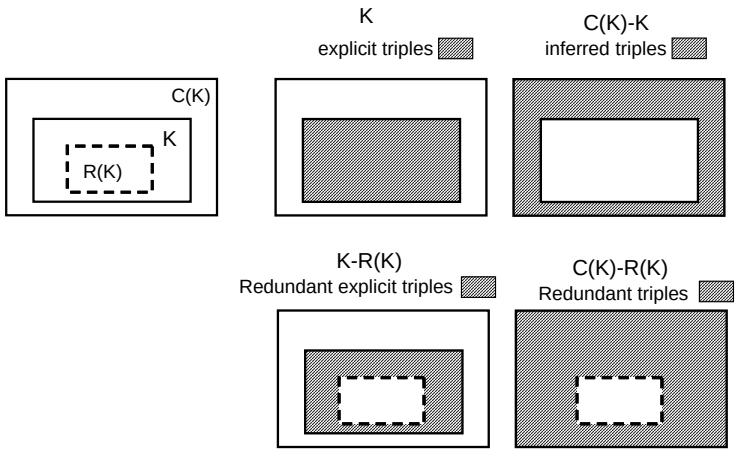

Figure 3.2: Distinctions of triples sets

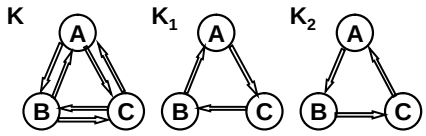

Figure 3.3: KB without unique reduction

3.2.3 Differential Function and Change Operations

In the context of this work, we focus on two *basic* change operations allowing to transform one KB to another, namely triple *addition* $Add(t)$ and *deletion* $Del(t)$ where $t \in T$. In this respect, a triple *update* is "split" into an addition and a deletion of triplets having the same *subject* and *predicate* (and thus keep both "old" and "new" values usually ignored by updates).

[37] introduced five differential functions of RDF/S KBs. They are given in Figure 3.4, namely, Δ_e, Δ_c, Δ_d, Δ_{dc} and Δ_{ed}.

$$
\begin{aligned}
\Delta_e(K \to K') &= \{Add(t) \mid t \in K' - K\} \cup \{Del(t) \mid t \in K - K'\} \\
\Delta_c(K \to K') &= \{Add(t) \mid t \in C(K') - C(K)\} \cup \{Del(t) \mid t \in C(K) - C(K')\} \\
\Delta_d(K \to K') &= \{Add(t) \mid t \in K' - C(K)\} \cup \{Del(t) \mid t \in K - C(K')\} \\
\Delta_{dc}(K \to K') &= \{Add(t) \mid t \in K' - C(K)\} \cup \{Del(t) \mid t \in C(K) - C(K')\} \\
\Delta_{ed}(K \to K') &= \{Add(t) \mid t \in K' - K\} \cup \{Del(t) \mid t \in K - C(K')\}
\end{aligned}
$$

Δ_e (where e stands for *e*xplicit) actually returns the set difference over the explicitly asserted triples, while Δ_c (where c stands for *c*losure) returns the set difference by also taking into account the inferred triples [1]. Three novel differential functions namely Δ_d (where d comes from *d*ense), Δ_{dc} (dc comes from *d*ense & *c*losure) and Δ_{ed} (ed comes from *e*xplicit & *d*ense) are introduced in [36]. It results that Δ_d produces the smallest in size set of change operations.

[1] Mention that Δ_e and Δ_c define a symmetric set difference.

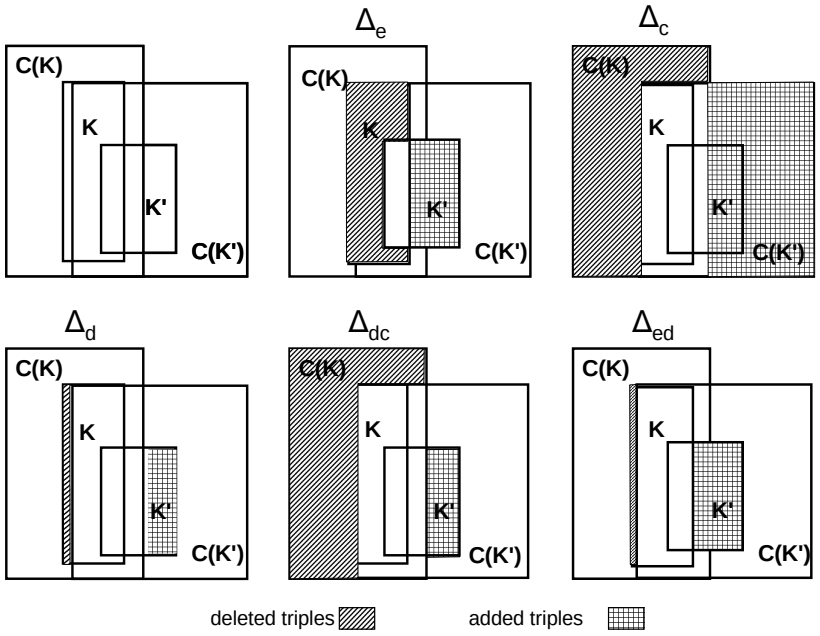

Figure 3.4: Added and deleted triples of the differential functions

These five differential functions yield essentially *sets of atomic change operations*. More formally, for a differential function $\Delta_x(K \rightarrow K'))$ where $x \in \{e, c, d, dc, ed\}$, Δ_x^+ is used to denote the corresponding set of triple additions (i.e. *incremental* changes) and Δ_x^- the set of triple deletions (i.e. *decremental* changes).

Obviously, Δ_x contains only sets of useful change operations reflecting the *net effect* of successive modifications over the same (explicit or inferred) triple of two KB versions. In other terms, Δ_x does not contain both $Add(t)$ and $Del(t)$ operations for a given $t \in T$.

Def. 2 A Delta $\Delta(K \rightarrow K')$ is *useful* if it holds $\Delta^+ \cap \Delta^- = \emptyset$.

By defining RDF Deltas as sets of atomic change operations, we avoid to specify an execution order as in an edit-script (i.e. a sequence of triple additions or deletions). This design choice amends to simpler computation requirements for RDF/S Deltas while it provides the opportunity of applying alternative semantics of changes when transforming one KB to another (i.e. with or without side-effects on the KB closure). e introduce two sets of change operations:

- *Atomic* change operations
  Here we have operations of the form $Add(t)$ and $Del(t)$ where $t \in T$.

- *Bulk* change operations
  Here we consider the operation $AddDel(A, D)$ where A and D are disjoint sets of triples (i.e. $A \subseteq \mathcal{T}$, $D \subseteq \mathcal{T}$, $A \cap D = \emptyset$).

It is not hard to see that the result of a differential function $\Delta(K \rightarrow K')$ could be applied on K by issuing one single call to $AddDel(\cdot, \cdot)$ (i.e. $AddDel(\Delta^+, \Delta^-)$), or by issuing $|\Delta^+|$ calls to $Add(\cdot)$ and $|\Delta^-|$ calls to $Del(\cdot)$ in any possible order.

For the needs of this work we mostly use the differential function Δ_e. For short, its output is called *delta*.

3.3 RDF KBs with Blank Nodes

3.3.1 RDF graph equivalence

The equivalence of *RDF graphs* that contain *blank nodes* is defined in [1] as:

Def. 3 (Equivalence of RDF Graphs that contain Blank nodes)
Two RDF graphs G_1 and G_2 are *equivalent* if there is a bijection[2] M between the sets of nodes of the two graphs (N_1 and N_2), such that:

- $M(uri) = uri$ for each $uri \in U_1 \cap N_1$
- $M(lit) = lit$ for each $lit \in L_1$
- M maps blank nodes to blank nodes (i.e. for each $b \in B_1$ it holds $M(b) \in B_2$)
- The triple (s, p, o) is in G_1 if and only if the triple $(M(s), p, M(o))$ is in G_2.

$\diamond$

It follows that if two graphs are equivalent then it certainly holds $U_1 = U_2$, $L_1 = L_2$ and $|B_1| = |B_2|$.

Let us now relate the problem of equivalence with *edit distances*.

Def. 4 (Edit Distance over Nodes given a Bijection)
Let o_1 and o_2 be two nodes of G_1 and G_2, and suppose a bijection between the nodes of these graphs, i.e. a function $h : N_1 \to N_2$ (obviously $|N_1| = |N_2|$). We define the edit distance between o_1 and o_2 over h, denoted by $dist_h(o_1, o_2)$, as the number of additions or deletions of triples which are required for making the "direct neighborhoods" of o_1 and o_2 the same (considering h-mapped nodes the same). Formally, $dist_h(o_1, o_2) =$
$|\{(o_1, p, a) \in G_1 \mid (o_2, p, h(a)) \notin G_2\}| + \quad |\{(a, p, o_1) \in G_1 \mid (h(a), p, o_2) \notin G_2\}| +$
$|\{(o_2, p, a) \in G_2 \mid (o_1, p, h^{-1}(a)) \notin G_1\}| + |\{(a, p, o_2) \in G_2 \mid (h^{-1}(a), p, o_1) \notin G_1\}| \quad \diamond$

Now recall that if G_1 is equivalent to G_2 then there exists a bijection h such that $(a, p, b) \in G_1 \Leftrightarrow (h(a), p, h(b)) \in G_2$. We will denote this by $G_1 \equiv_h G_2$. It follows that:

Theorem 1 (RDF Graph Equivalence and Edit Distance)
$G_1 \equiv_h G_2 \Leftrightarrow dist_h(o, h(o)) = 0$ for each $o \in N_1$.

Obviously the above theorem is useful for the case where the bijection h respects the constraints of Def. 3 (i.e. maps named elements to named elements, and anonymous elements to anonymous).

[2] A function that is both one-to-one (injective) and onto (surjective).

3.3.2 Bnode Name Tuning

The basic idea for reducing the delta is the following: if we match a blank node b_1 (of B_1) to a blank node b_2 (of B_2), through a bijecion M, then these blank nodes can be considered as equal at the computation of delta. For example, if K_1 contains a triple $(b_1, name, Joe)$ and K_2 contains a triple $(b_2, name, Joe)$ and we match b_1 to b_2, then these two triples will be considered equal and thus no difference will be reported. However we should note that in the context of versioning or synchronization services the change operations derived by a differential function *should not* be used as they are. For example, consider $K_1 = \{(b_1, name, Joe)\}$ and $K_2 = \{(b_2, name, Joe), (b_2, lives, UK)\}$ and suppose that we match again b_1 to b_2. In this case a mapping-aware comparison function will return the delta $\{Add((b_2, lives, UK))\}$. If we want to apply it on K_1 then we have to replace b_2 by b_1, i.e we should apply on K_1 the operation $Add((b_1, lives, UK))$, and in this way, we will obtain $K_1' = \{(b_1, name, Joe), (b_1, lives, UK)\}$ which is equivalent to K_2. We call this step *Bnode Name Tuning*, and it actually replaces (renames) in the delta the local names of the blank nodes of B_2 by the local names of the matched blank nodes in B_1. In this way the delta does not need any *rename* operation (i.e. $rename(b_1, b_2)$) and hence not any particular execution order.

3.3.3 Delta reduction size

Blank node matching cannot increase the delta size. Without blank node matching any pair of blank nodes from different Knowledge Bases is considered different, and thus all triples to which they participate will be different and reported as change operations in the delta. In particular, we denote D_a as the average number of direct edges of the blank nodes (i.e. average number of triples to which a blank node participates) and $n_1 = |B_1|$, $n_2 = |B_2|$. Without blank node matching the produced delta will contain at least $(n_1 + n_2) * D_a$ change operations.
On the other hand, if two blank nodes are matched then the delta size is reduced if they participate to triples with the same predicate and the same other node (i.e. the same *subject* or *object*). In the worst case where all predicates/nodes of these triples are different, the delta size that will be reported is what will be reported without blank node matching.

3.4 Bnode Matching as an Optimization Problem

Let us now focus on the case where two Knowledge Bases, K_1 and K_2, are *not necessarily equivalent* and do contain blank nodes. We would like to find a mapping over their blank nodes that reduces the size (i.e. the number of change operations) of their delta and allows detecting whether K_1 is equivalent to K_2. Furthermore we want an efficient (tractable at least) method for finding such a mapping.

3.4.1 Problem Formulation

Here we formulate the problem of finding a mapping between the blank nodes of two Knowledge Bases as an optimization problem. Let $n_1 = |B_1|$, $n_2 = |B_2|$ and $n = min(n_1, n_2)$. We have to match n elements of B_1 with n elements of B_2, i.e. our objective is to find the unknown part of the bijection M. To be

more precise, M a priori contains the mappings of all the URIs and literals of the Knowledge Bases (URIs and literals are mapped as an identity function as in Def. 3), and its unknown part concerns B_1 and B_2. Suppose that $n = n_1 < n_2$. Let $\mathcal{J}$ denote the set of all possible bijections between B_1 and a subset of B_2 that comprises n elements. The number of all possible bijections (i.e. $|\mathcal{J}|$) is $n_2 * (n_2 - 1) * ... * (n_2 - n_1 + 1)$, i.e. the first element of B_1 can be matched with n_2 elements of B_2, the second with $n_2 - 1$ elements, and so on. Consequently, the set of candidate solutions is exponential in size.
Since our objective is to find a bijection $M \in \mathcal{J}$ that reduces the delta size (as regards the "unnamed" parts of the Knowledge Bases), we define the cost of a bijection M as follows:

$$Cost(M) = \sum_{b_1 \in B_1} dist_M(b_1, M(b_1)) \tag{3.1}$$

Def. 5 (The bijection yielding the less delta size) The best solution (or solutions) is defined as the bijection with the minimum cost, i.e. we define:

$$M_{sol} = \arg_M \min_{M \in \mathcal{J}} (Cost(M)) \qquad \diamond$$

The notation arg_M returns the M in $\mathcal{J}$ that gives the minimum cost.

Theorem 2 (Equivalence and Mapping Cost) *If $G_1 \equiv_{M_{sol}} G_2$ (according to Def. 3) then $Cost(M_{sol}) = 0$.*

The proof follows easily from the definitions. It is also clear that the inverse of Th. 2 does not hold (i.e. $Cost(M_{sol}) = 0 \not\Rightarrow G_1 \equiv_{M_{sol}} G_2$) because the cost is based on the distance between the direct neighborhoods of the blank nodes *only*, and not between the named parts of the graphs.

From the algorithmic perspective, one naive approach for finding the best solution (i.e. M_{sol}) would be to examine the set of all possible bijections. That would require at least $n!$ examinations (true if $n_1 = n_2 = n$, while if $n_1 < n_2$ then their number is higher than $n!$). However, the problem is intractable in general:

Theorem 3 Finding the optimal bijection (according to Def. 5) is NP-Hard.
Proof*:*
We will show that subgraph-isomorphism *(which is NP-complete problem) can be reduced to the problem of finding the optimal bijection (meaning that our problem is at least as hard as* subgraph-isomorphism*). Let us make the* hypothesis *that we can find the optimal bijection in polynomial time. We will prove that if that hypothesis were true, then we would be able to solve the subgraph isomorphism in polynomial time. The* subgraph isomorphism decision problem *is stated as: given two plain graphs G_1 and G_2 decide whether G_1 is isomorphic to a subgraph of G_2. Let $G_1 = (N_1, R_1)$ and $G_2 = (N_2, R_2)$. We can consider these graphs as two RDF graphs such that: all of their nodes are blank nodes and all property edges have the same label. Assume that $|N_1| \leq |N_2|$ and let $n = \min(|N_1|, |N_2|)$. If we can find in polynomial time whether there is a bijection between the n nodes of G_1 and n nodes of G_2 such that $Cost(M_{sol}) = 0$, then this means that we have found whether G_1 is isomorphic to a subgraph of G_2. Specifically, to decide whether there is a subgraph isomorphism, (a) we compute the optimal bijection, say M_{sol}, and (b) we compute its cost. If the cost returned by step (b) is 0 then we return YES, i.e. that there is a subgraph isomorphism. Otherwise we return NO (i.e. there is no subgraph isomorphism). Note*

that step (a) is polynomial by hypothesis*, while step (b) relies on Def. 4 and its cost is again polynomial. Regarding the latter, note that M_{sol} contains n pairs, and to compute $dist_M(b_1, b_2)$ for each (b_1, b_2) pair of M, we consider only the direct neighborhoods of the two nodes in the two graphs (for G_2 we have to consider only those that connect nodes that participate in M_{sol})*[3]*. It follows that its computational cost is analogous to the number of edges of the graphs, and thus polynomial. Therefore given a bijection M_{sol}, to compute $Cost(M_{sol})$ requires polynomial time. Also note that Th. 1 holds also for plain graphs assuming a distance function over not labeled edges. We conclude that if our hypothesis were true, then we would be able to decide subgraph isomorphism in polynomial time.*

We conclude that finding the optimal bijection is NP-Hard.$\diamond$

Below we will show that there are algorithms of polynomial complexity for a frequently occurring case. For the general case, we will propose algorithms of polynomial complexity that return an approximate solution.

3.4.2 Polynomially-solved (and Frequently Occurring) Cases

3.4.2.1 No directly connected blank nodes

Consider the Knowledge Bases in Figure 3.5 and suppose that we want to compute $dist_h(\_:1, \_:6)$ (according to Def. 4). It is not hard to see that this distance depends on the mappings (by h) of the blank nodes that are connected to $\_:1$ and $\_:6$, i.e. on the mappings of $\_:3, \_:4, \_:8$ and $\_:9$. However several datasets do not have directly connected blank nodes. For this reason, here we study a variation of the problem that is appropriate for this case. The key point is that the distance between two blank nodes does not depend on how the rest blank nodes are mapped.

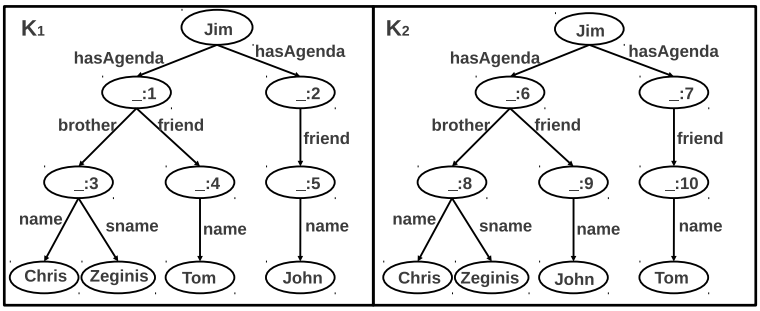

Figure 3.5: Two Knowledge Bases with directly connected blank nodes

This is very important because in this case we can solve the optimization problem (as defined in Definition 5) using the *Hungarian algorithm* [24] (for short Alg_{Hung}, an algorithm for solving the *assignment problem*. Here the elements (blank nodes) of B_1 play the role of *workers*, the elements (blank nodes) of B_2 play the role of *jobs*, and the edit distances of the pairs in $B_1 \times B_2$ play the role of the *costs*. Consider for the moment that $|B_1| = |B_2|$. If we compute the edit distances between all possible n^2 pairs, then Alg_{Hung} can find the optimal assignment at the cost of $O(n^3)$ time. This means that finding the optimal solution

[3] *Alternatively, if $Cost(M_{sol}) \neq 0$ (using the distance as defined in the main paper), we return YES only if $\Delta_e(G_1 \rightarrow G_2)$ as defined in section* **??***, after blank node name tuning, contains only triples each containing one blank node in B_1 and one not in B_1.*

costs polynomial time. An extension of Alg_{Hung} giving the ability to assign the problem in rectangular matrices (i.e. when $|B1| \neq |B2|$) is already provided in [7]. We conclude that if there are not directly connected blank nodes then the optimal mapping can be found in polynomial time.

Theorem 4 Finding the optimal bijection (according to Def. 5) is a polynomial task if there are no directly connected blank nodes.$\diamond$

3.4.2.2 BNode Neighborhoods with bounded tree width

[21] also surveyed the structure of blank nodes in published data. According to their results almost the 58% of their documents containing blank nodes had directly connected blank nodes. However, the 98.4 of those blank node structures were acyclic. The acyclicity of a structure is equivalent with a tree width up to 1.

Taking the above statistics into account, it is worth investigating the existence of tractable solutions for the wider case of RDF graphs with acyclic bnode neighborhoods.

Bounding the tree width makes many intractable (in the general case) problems tractable.

3.5 Approximation Algorithms

At section 3.5.1 we present a variation of Alg_{Hung} for getting an *approximate* solution for the general case, then at Section 3.5.5 we present a *signature*-based algorithm appropriate for larger datasets.

3.5.1 Hungarian BNode Matching Algorithm

We have already stated that Alg_{Hung} can find the optimal mapping in polynomial time if no directly connected bnodes exist in the compared KBs. For the cases where there are directly connected bnodes, Alg_{Hung} enriched with an assumption regarding how to treat the connected bnodes at the computation of $dist_h$, could be used for producing an *approximate* solution. Also in this case the algorithm will make $n_1 \times n_2$ distance computations (where $n_1 = |B_1|$ and $n_2 = |B_2|$), and the complexity of the algorithm will be again $O(n^3)$.

Regarding connected bnodes, at the computation of $dist_h$, one could either assume that all of the connected bnodes are different, or all of them are the same. The first assumption does not require any bijection (h contains only the identity functions of the URIs and literals). According to Definition 4, the fact that all the bnodes are different means by extension that the triples in the direct neighborhoods connecting blank nodes are different too, even in the case where these triples have the same properties. For instance, applying the Definition 4 between bnodes (_ : 1, _ : 6) and (_ : 1, _ : 7) of Figure 3.5, we get that $dist_h($_ : 1, _ : 6$) = 4$ and $dist_h($_ : 1, _ : 7$) = 3$ respectively. However, bnodes _ : 1, _ : 6 have two outgoing triples with exactly the same properties, while bnodes _ : 1, _ : 7 have only one. We observe that this assumption is not very good because we would prefer _ : 1 to be "closer" to _ : 6 than to _ : 7.

According to the alternative assumption, when comparing bnodes (_ : 1, _ : 6) in Figure 3.5, bnode _ : 3 can be matched either with bnode _ : 8 or with bnode

_ : 9, depending on the existence of a common property between them. This yields $dist_h(\_ : 1, \_ : 6) = 0$ since both bnodes have two outgoing triples with common properties (i.e. $(\_ : 1, brother, \_ : 3)$ is matched with $(\_ : 6, brother, \_ : 8)$ and $(\_ : 1, friend, \_ : 4)$ is matched with $(\_ : 6, friend, \_ : 9)$). Regarding $\_ : 1$ and $\_ : 7$, we get $dist_h(\_ : 1, \_ : 7) = 1$ because of the deleted triple $(\_ : 1, brother, \_ : 3)$. It follows that the results of this assumption are better over this example, as $\_ : 1$ is "closer" to $\_ : 6$ than to $\_ : 7$. In general it is better because it exploits common properties, and therefore we adopt this assumption in our experiments.

In other words, the above assumptions do not compute the "likelihood" of the blank nodes to be matched. They arbitrarily suppose that there is no likelihood (0) of matching (first assumption) by charging with 2 the edit distance (one deletion and one addition) or there is probability equal to 1 of the blank nodes to be matched, by not charging at all the edit distance. The computation of the "likelihood" would require an increase in the time and memory demands.

3.5.2 A Fast ($O(N \log N)$) Signature-based Algorithm

The objective here is to devise a faster mapping algorithm that could be applied to large KBs (with a heavy load of bnodes), at the cost of probably bigger deltas and less chances to detect equivalences if they exist [4]. We propose a *signature-based* mapping algorithm, for short Alg_{Sign}, which consists of two phases: the signature construction and the mapping construction phase. Regarding the signature construction phase, for each bnode b we produce a string based on the direct neighborhood of b. This string is called the *signature* of bnode b. At the end of this phase we get two lists of signatures, one for the bnodes of each KB. These lists should be considered as bags rather than sets, as there is a probability that two or more bnodes get the same signature. This probability and by extension the deviation of this algorithm from the optimal solution depends on the way the signature is built (we discuss this later).

```
Alg. SignatureMapping
Input: two sets of bnodes B1 and B2,
     where |B1| < |B2|
Output: a bij. M between B1 and B2
(1)  M = ∅
(2)  BS1 = BS2 = emptybag
(3)  for each b1 ∈ B1
(4)      BS1 = BS1 ∪ {Signature(b1)}
(5)  for each b2 ∈ B2
(6)      BS2 = BS2 ∪ {Signature(b2)}
(7)  sort(BS1)
(8)  sort(BS2)
(9)  for each bs1 ∈ BS1
(10)     bs2 = Lookup(BS2, bs1)
(11)     if (bs2 == bs1)
(12)         M = M ∪ {(bn1[bs1], bn2[bs2])}
// bn1[str] returns the b ∈ B1 corresponding to str
(13)         BS2.remove(bs2)
(14)         BS1.remove(bs1)
(15) for each bs1 ∈ BS1
(16)     bs2 = Lookup(BS2, bs1)
(17)     M = M ∪ {(bn1[bs1]), bn2[bs2])}
(18)     BS2.remove(bs2)
(19) return M
```

Figure 3.6: Alg. The Signature-based bnode matching algorithm

The mapping phase takes these two bags of strings and compares the elements of the first bag with those of the second. To make *binary search* possible, both bags (lists) are sorted lexicographically. Subsequently, we start from the smaller list,

[4] however we theoretically and experimentally show that it gets the same chance to detect equivalence

say BS_1, and for each string bs_1 in that list we perform a lookup in the second list BS_2 using *binary search*. If an exact match exists (i.e. we found the string bs_1 also in BS_2) we produce a bnode mapping, i.e. the pair $(bn_1[bs_1], bn_2[bs_1])$. Since more than one bnodes may have the same signature we select one. We prefer the order as provided by the managing software, which in many cases reflects the order by which bnodes appear in files. As there is a high probability for subsequent versions to keep the same serialization, using the original order increases the probability of matches in case of same signatures[5]. More information about the role of serialization are given in Section **??**. We continue in this way for all strings of BS_1. For each element bs_1 of BS_1 for which no exact match was found in BS_2 we perform a second lookup over the remainder part of BS_2, say BS'_2, which will produce a mapping based on the *closest* element of BS'_2 to the bs_1 element. Specifically we will match bs_1 to the element of BS'_2 to which binary search stopped, i.e. to the *lexicographically closer* element. Note that we perform the closest matches after finishing with the exact matches in order to avoid the situation where an approximate (closest) match deters an exact match at a later step.

The algorithm is shown in Figure 3.10 and relies on an algorithm *Signature* for producing signatures, and on an algorithm *Lookup*, that are analyzed below. The complexity of this algorithm is $O(n \log N)$ where $N = \max(n_1, n_2)$ and $n = \min(n_1, n_2)$, assuming that the average graph degree of bnodes (and thus signature size) does not depend on N.

3.5.2.1 Signature Construction

A crucial issue is how the signature of each bnode is derived. We would like to devise a signature construction method producing signatures with high discrimination power. Our objective is to derive a string that will allow good matches in the *Lookup* algorithm even if the bnodes do not match exactly. The more resources we capture inside the signature of a bnode, the more we increase its discrimination power. However, the need to keep the signature size independent of N forces us to remain only in the direct neighborhood.

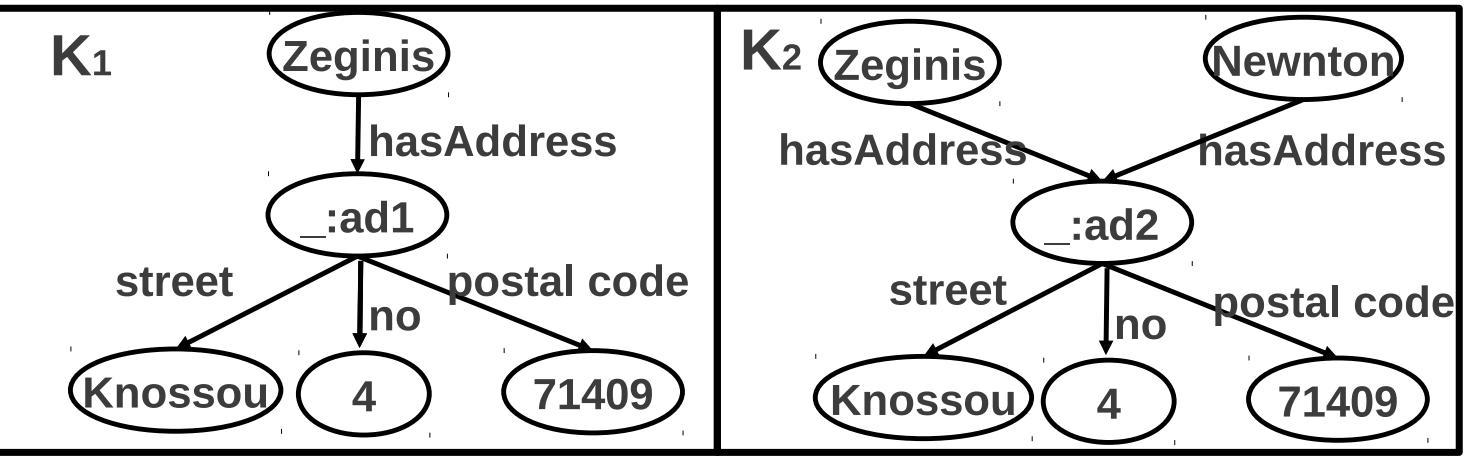

Figure 3.7: Two Knowledge Bases of an address ontology

Figure 3.11 shows two subsequent KBs of the Address Ontology. Table 3.2 gives the signatures of all the bnodes in Figure 3.11. The namespaces of the resources

[5]We do the same in Alg_{Hung} in case of ties in costs.

are omitted here just for needs of space.

Table 3.1: *Signatures* on bnodes of K_1 and K_2 of Fig. 3.11 according to the given option

Local Name	Signature
_ : 1	$ChristinahasAddress\diamondsuit typeAddress\diamondsuit cityLondon * no14 * streetOxfordStreet$
_ : 3	$ChristinahasAddress\diamondsuit typeAddress\diamondsuit cityLondon * no14 * streetOxfordStreet$
_ : 2	$YannishasAddress\diamondsuit typeAddress\diamondsuit cityNewYork * no445 * streetBroadway$
_ : 4	$YannishasAddress\diamondsuit typeAddress\diamondsuit cityChicago * no132 * streetMichiganAvenue$

Consider bnode _ : 1 of Figure 3.11 which is involved in the following triples:
Incoming triples: {(Christina, hasAddress, _:1)},
Outgoing triples: {(_:1, street, Oxford Street), (_:1, No, 14), (_:1, city, London)},
Class Type triples: {(_:1, typeOf, Address)}.

The set *Class Type* contains the triples with the `rdf:type` property of the respective bnode. Each of these triples will be mapped to a substring (i.e. "ChristinahasAddress" for the triple $(Christina, hasAddress, \_ : 1)$).

For each one of the three different sets of triples (Incoming, Outgoing, Class Type) we are going to construct a concatenation of substrings (i.e. "$cityLondon * no14 * streetOxfordStreet$" for the Outgoing set of triples). The substrings inside each set are sorted lexicographically and separated by a special character, here denoted by $*$.

These sets of substrings are then concatenated and separated by a special character, here denoted by $\diamondsuit$. This final concatenation yields the signature. In case there is a blank node as subject in the set of incoming triples or object in the set of outgoing triples, we replace it by a special character, here denoted by ♣. In other words, we treat them as equal, as we did in the second assumption of approximation version of Alg_{Hung}. The reasoning of the structure of the signature is given later.

The exact steps of the signature construction algorithm are shown in algorithm $Signature1$ at Figure 3.12. The method $name(o)$ gives us the uri reference if o is a *uri* and the label if o is a literal. The method $isBNode(n)$ returns a boolean that is true if n is a bnode or false if it is not. The strings $inSet$, $outSet$, $classSet$ are the three sets that are gradually built by concatenating the information from all the triples in the direct neighborhood.

3.5.2.2 The Lookup algorithm

This algorithm is actually responsible for the high time efficiency of this approximation algorithm, as it is a slightly modified version of *Binary Search*. Instead of the classical $BinarySearch$, this algorithm *Lookup* always returns a string. If the *Lookup* succeeds, it returns the matching (i.e. the string that we search), otherwise it returns the closest (in lexicographical order) string. The closest string is defined as the string located either in the position the $BinarySearch$ stopped or in the exact previous (if the last iteration decreased the high value) or in the next (if the last iteration increased the low value) position. In order to make a decision among the two strings (signatures), we define the elements of a signature,

Alg. Signature1
Input: a blank node b
Output: a string bs signature of the blank node
(1) $bs = null$
(2) $inSet = outSet = classSet = \emptyset$
//construction of the Set of Incoming Triples
(3) for $each\{(sub, pr, ob) \mid ob = b\}$
(4) if $(isBNode(sub))$
(5) $inSet = inSet \cup \{name(pr) + "\clubsuit"\}$
(6) else
(7) $inSet = inSet \cup \{name(pr) + name(sub)\}$
(8) for $each\{(sub, pr, ob) \mid sub = b\}$
//construction of the Set of rdf:type Triples
(9) if (isTypeOf(pr))
(10) $classSet = classSet \cup \{name(pr) + name(ob)\}$
(11) else if $(isBNode(ob))$
(12) $outSet = outSet \cup \{name(pr) + "\clubsuit"\}$
//construction of the Set of Outgoing Triples
(13) else
(14) $outSet = outSet \cup \{name(pr) + name(trim(ob))\}$
(15)sort($inSet$)
(16)sort($outSet$)
//concatenation of the three Sets
(17)for $(each\ strings \in inSet)$
(18) $bs = bs + s + "\diamondsuit"$
(19)for $(each\ strings \in classSet)$
(20) $bs = bs + s + "\diamondsuit"$
(21)for $(eachstrings \in outSet)$
(22) $bs = bs + s + "\diamondsuit"$
(23)return bs

Figure 3.8: Signature Construction Algorithm

say $elems(bs)$, as the number of the building blocks (i.e. substrings split by the characters $\diamondsuit$ or $*$) included in its string. The returned string among these two strings is the one that has as *elems* a value closer to the *elems* of the searching string. The exact steps of the algorithm *LookUp* are shown in Figure 3.13.

Alg. Lookup
Input: a string bs and a sorted bag of strings BS
Output: the best matching string bs_2
(1) $a = NULL$
(2) $low = 0$
(3) $high = BS.length$
(4) while $(low < high)$
(5) $\quad mid = \lfloor (low + high)/2 \rfloor$
(6) $\quad bmid = BS[mid]$
(7) $\quad$ if $(bs < bmid))$
(8) $\quad\quad high = mid$
(9) $\quad$ else if $(bs > bmid)$
(10) $\quad\quad low = mid + 1$
(11) $\quad$ else return $bmid$ // exact match found
(12)end while
(13)if $(bs < bmid \wedge (|elems(bs) - elems(bmid)| > |elems(bs) - elems(BS(mid-1))|)$
(15) $\quad$ return $BS[mid-1]$
(16)else if $(bs > bmid) \wedge (|elems(bs) - elems(bmid)| > |elems(bs) - elems(BS(mid+1))|)$
(18) $\quad$ return $BS[mid+1]$
(19)else
(20) $\quad$ return $bmid$

Figure 3.9: Lookup algorithm

Applying the *SignatureMapping* algorithm in the simple example of Figure 3.11 we get the final mapping: _:1 ↔ _:3 and _:2 ↔ _:4. Going to a more complex example (i.e. with connected bnodes), like the one of Figure 3.5 the final mapping would be _:1 ↔ _:6, _:2 ↔ _:7, _:3 ↔ _:8, _:4 ↔ _:10, _:5 ↔ _:9.

3.5.3 More about the Signature Construction

A key point in the efficiency of this algorithm is the order by which the sets of triples are concatenated. The prevailing option is to give a first priority to the set of the incoming triples, a second priority to the set with the rdf:type properties, and the last priority to the set of the outgoing triples. Let us remind that at this phase we are looking for a signature construction method that is completely general and domain-agnostic.

Some intuition for the proposed ordering stems from the evidence that the probability for the outgoing statements to change is higher than the ingoing (i.e. like in Figure **??**(a) where updating the address of a person is more probable than changing his/her name).

A closer approach to the functionality of the blank nodes, could persuade the reader for the prevalence of their outgoing triples over their incoming triples. This fact entails that the probability for an outgoing triple to change is higher than an incoming triple. Based on the analysis of functionality of bnodes, described

in Section 1, we get the following: (a) the multi-component structure yields that bnodes are arising more times as subjects than objects (see Figure **??**(a)), (b) the functionality of provenance imposes the usage of bnodes in the subject position more than in the object position and (c) in the "Multi-relationship expression" the bnode occurs once in the object position and more than once in the subject position.

Moreover, taking into account the empirical surveys of bnodes in Linked Data, we get that, according to [21], surveying a corpus of 1G quadruples, they got that each bnode occured 0.99 times in the object position of a non-rdf:type triple (with 1.9% of all bnodes not occuring at all in the object position), whereas each bnode occurred on average 4.2 times in the subject position of a triple (with 0.04% not occuring at all in the subject position).

A reason that urges the above results is the tree-based RDF/XML syntax.

We conclude that mainly the functionality purposes and secondly the prevailed RDF format (RDF/XML) indicate the dominance of bnodes in the subject position, justifying that the Incoming Set has more probability to stay stable from the one version to the other, than the Outgoing Set.

3.5.4 Comparing the approximation algorithms

In this section we are going to compare the algorithms Alg_{Hung} and Alg_{Sign} at a theoretical level. An analysis over their complexities has already been offered in the previous sections and made clear that the Alg_{Sign} offers a much better time and main memory complexity. Now we are going to focus on the equivalence detection and delta reduction potential.

3.5.4.1 On Equivalence Detection Potential

As regards the equivalence detection potential we get the following proposition:

Theorem 5 (Comparing on Equivalence Detection Potential) The Alg_{Sign} has the same equivalence detection potential with the Alg_{Hung}. Proof*:*

Let us suppose that we have two equivalent KBs (K_1 and K_2). We have to investigate the following cases.
1st Case: The KBs do not have any directly connected bnodes. In this case, the constructed signatures do not contain any blank node (as no bnode belongs to the direct neighborhood of a bnode). This means that during signature matching no assumptions have to be made about the treatment of the directly connected bnodes (in equivalence with the Optimal Hungarian). Let us focus on the signature matching phase and suppose that exact and closest signature mappings occur. Suppose a closest signature mapping occurs i.e. the signature s_1 of a bnode b_1 in K_1 is mapped to the signature s_2 of a bnode b_2 in K_2 and $s_1 \neq s_2$. The fact that s_1 was mapped to s_2 entails that there is no signature of K_2 that equals s_1, or that there are more signatures that are the same with s_1 in K_1 than there are in K_2 (so the rest of the same bnodes are already mapped and removed from the signature lists according to lines 9 - 14 of the Signature Mapping Algorithm). From the signature construction method we know that all triples of the direct neighborhood of b_1 are represented inside the s_1 in a deterministic manner, which is independent of the serialization order of these triples. In other words, two same signatures correspond to two bnodes with the

same direct neighborhoods. As a result, we get that the closest signature matching entails that there is a bnode in K_1 with a direct neighborhood D_1 that does not exist in K_2 or that there are more bnodes in K_1 equivalent with D_1 than there are in K_2. Both entailments are wrong because K_1 and K_2 are equivalent. We conclude that the signature matching phase of two equivalent KBs performs only exact signature mappings. For this subcase, this fact is equivalent to a total cost equals to zero, as an exact signature matching is equivalent with a zero edit distance and additionally is independent of the other blank node pairs. We conclude that for equivalent KBs the Alg_{Sign} *always finds the solution that has a zero total cost for the* Alg_{Hung} *and by extension it always finds the optimal bijection (according to Theorem 2 and Theorem 4).*
2nd Case: The KBs contain directly connected bnodes. Recall that signature construction phase represents each bnode in the direct neighborhood of a bnode with the same symbol. Consequently, all the connected bnodes are considered the same during the signature matching phase, exactly like in the Alg_{Hung}*. This means that if we focus on the signature matching phase, it is clear that we only have exact signature mappings (similarly with the 1st Case). Each exact signature matching is equivalent to a zero edit distance between the bnodes of these signatures in the* Alg_{Hung}*. As a result, the same bnode matchings are extracted for the* Alg_{Sign} *and the* Alg_{Hung}*. Remind that each KB remains exactly the same (same serialization of bnodes) during the comparison with both the algorithms.* $\diamond$

3.5.4.2 On Delta Reduction Potential

The lower delta reduction potential is the price to pay for the low time complexity of the Alg_{Sign}. Recall that the binary search produces a bnode mapping giving priority to the first part of the signatures. Consequently, its delta reduction potential is built upon a quite general probabilistic model based on predictions/statistics about the changes that we expect to be made on the direct neighborhoods of the bnodes. This is the extra approximation factor in relation to the Alg_{Hung}, that justifies a probable bigger delta.

Thus, in case of KBs with no directly connected bnodes the Alg_{Sign} gives the same (optimal) or bigger delta than the Alg_{Hung}. However, in case of KBs with directly connected bnodes there are scenarios, where the Alg_{Sign} exports bigger deltas than the Alg_{Hung}, but there are other scenarios where the Alg_{Sign} exports smaller deltas than the *hung*. In other words, the extra approximation factor of the Alg_{Sign} may impact in a positive way to the exported delta. The percentage of how smaller or bigger the delta could be cannot be determined generally, only under specific domain statistics.

3.5.5 On the serialization of the Knowledge Bases

As we have already mentioned, the serialization of the two files may play an important role in the delta reduction or the equivalence detection potential of both the algorithms. In particular, in case of equivalent KBs with directly connected blank nodes the only factor that restricts the Alg_{Sign} and the Alg_{Hung} algorithms from getting the optimal bijection is the serialization order of their blank nodes. In other words, the serialization order is the order of appearance of the blank nodes inside the file. Formally, we get:

Theorem 6 (The serialization order as an approximation factor) If two equivalent KBs have the same serialization order (for their bnodes), both the Alg_{Sign} and Alg_{Hung} find the optimal bnode mapping.
Proof*:*
We will prove that if two equivalent KBs (K_1 and K_2) have the same serialization order for their bnodes, then the optimal bijection between their bnodes will be built. Suppose that K_1 and K_2 have the same serialization fr their bnodes and that they do not give the optimal bnodes bijection. The Alg_{Hung} *will calculate the edit distances between the bnodes of K_1 and K_2. From the Theorem 1 we get that if two RDF graphs are equivalent, then there is a bijection M between their bnodes such that the pairs of bnodes of this bijection have a zero edit distance. As the* Alg_{Hung} *makes an assumption about the treatment of the connected bnodes that assumes all the connected bnodes the same, it will compute as zero at least the edit distances of the pairs of bnodes in M. However there is a probability that a bnode b_1 of B_1 has a zero edit distance not only with b_2 of B_2 s.t. $(b_1, b_2) \in M$, but also with other bnodes of B_2 and inversely. So, we should further investigate this case and in particular we give the two different scenarios for the existence of this case.*
Scenario A': There are more than one bnodes (let's say 2*) in K_1 with the same direct neighborhoods (D_1) and respectively there are* 2 *bnodes with the same direct neighborhoods D_1 in K_2 (as K_1 and K_2 are equivalent). Their direct neighborhoods do not contain bnodes, only URIs and literals. As a result, whatever kind of bijection can occur between the* 2 *bnodes of K_1 and the* 2 *bnodes of K_2 the total cost will remain zero and the optimal solution will be given in any case.*
Scenario B': There are more than one bnodes in K_1 (let's say 2*) with exactly the same named parts and the same number of bnodes in their direct neighborhoods (D_1, D_2). Respectively there are* 2 *bnodes in K_2 with exactly the same features. Let us denote with b_1, b_2 the bnodes $\in K_1$ and b_3, b_4 the bnodes $\in K_2$. Bnodes b_{1a}, b_{1b} are connected to b_1 and respectively b_{2a}, b_{2b} are connected to b_2 and so on. Let us keep the same serialization order of these bnodes for both the files of K_1 and K_2, or in other words if the order that the bnodes are displayed inside the first file is b_1, b_{1a}, b_{1b}, b_2, b_{2a}, b_{2b}, then the order in the second file will be b_3, b_{3a}, b_{3b}, b_4, b_{4a}, b_{4b}. Subsequently, we get that the* Alg_{Hung} *gives an array of edit distances with many zero values, because of the assumption that all the connected bnodes are the same. However, because the serialization remained the same, we get that the pairs (b_1, b_3) and (b_2, b_4) will be given in the final bijection and the assumptions that are made during the computation of the edit distances will also be part of the final bijection. It similarly goes, if instead of 2 bnodes structures we had* 3 *and so on. So the same serialization order of the bnodes ensures the fact that no assumptions made for the edit distances of the final bijection, is going to be violated and by extension the total cost will remain* 0 *or else we get the optimal solution. We get that our assumption does not stand.* $\diamond$

The objective here is to devise a faster mapping algorithm that could be applied to large KBs, at the cost of probably bigger deltas and less chances to detect isomorphisms if they exist [6]. We propose a *signature-based* mapping algorithm, for short Alg_{Sign}, which consists of two phases: the signature construction and the mapping construction phase. For each bnode b we produce a string based on the direct neighborhood of b. This string is called the *signature* of bnode b. This phase gives us two lists of signatures, one for the bnodes of each KB. These lists

[6] theoretically and experimentally show that gets the same chance to detect isomorphism

should be considered as bags rather than sets, as there is a probability that two or more bnodes get the same signature. The probability depends on the way the signature is built (we discuss this later).

Alg. SignatureMapping	(9) for *each* $bs_1 \in BS_1$
Input: two sets of bnodes B_1 and B_2, where $\|B_1\| < \|B_2\|$	(10) $bs_2 = Lookup(BS_2, bs_1)$
	(11) if $(bs_2 = bs_1)$
Out: a bij. M between B_1 and B_2	(12) $M = M \cup \{(bn_1[bs_1], bn_2[bs_2])\}$
(1) $M = \emptyset$	// $bn_1[str]$ returns the $b \in B_1$ corresponding to *str*
(2) $BS_1 = BS_2 = emptybag$	(13) $BS_2.remove(bs_2)$
(3) for *each* $b_1 \in B_1$	(14) $BS_1.remove(bs_1)$
(4) $BS_1 = BS_1 \cup \{Signature(b_1)\}$	(15) for *each* $bs_1 \in BS_1$
(5) for *each* $b_2 \in B_2$	(16) $bs_2 = Lookup(BS_2, bs_1)$
(6) $BS_2 = BS_2 \cup \{Signature(b_2)\}$	(17) $M = M \cup \{(bn_1[bs_1]), bn_2[bs_2])\}$
(7) $sort(BS_1)$	(18) $BS_2.remove(bs_2)$
(8) $sort(BS_2)$	(19) return M

Figure 3.10: Alg. The Signature-based bnode matching algorithm

The mapping phase takes these two bags of strings and compares the elements of the first bag with those of the second. To make *binary search* possible, both lists are sorted lexicographically. Subsequently, we start from the smaller list, say BS_1, and for each string bs_1 in that list we perform a lookup in the second list BS_2 using *binary search*. If an exact match exists (i.e. we found the string bs_1 also in BS_2) we produce a mapping, i.e. the pair $(bn_1[bs_1], bn_2[bs_1])$. Since more than one bnodes may have the same signature we select one. We prefer the order as provided by the managing software, which in many cases reflects the order by which bnodes appear in files. As there is a high probability for subsequent versions to keep the same serialization, using the original order increases the probability of matches in case of same signatures[7]. We continue in this way for all strings of BS_1. For each element bs_1 of BS_1 for which no exact match was found in BS_2 we perform a second lookup over the remainder part of BS_2, say BS_2', which will produce a mapping based on the *closest* element of BS_2' to the bs_1 element. Specifically we will match bs_1 to the element of BS_2' to which binary search stopped, i.e. to the *lexicographically closer* element. Note that we perform the closest matches after finishing with the exact matches in order to avoid the situation where an approximate match deters an exact match at a later step.

The complexity of this algorithm is $O(n \log N)$ where $N = \max(n_1, n_2)$ and $n = \min(n_1, n_2)$, assuming that the average graph degree of bnodes (and thus signature size) does not depend on N. The algorithm is shown in Figure 3.10 and relies on an algorithm *Signature* for producing signatures, and on a algorithm *Lookup*. These are analyzed below.

3.5.5.1 Signature Construction

A crucial issue is how the signature of each bnode is derived. We would like a signature construction method producing signatures with high discrimination power. The objective is to derive a string that will allow good matches (i.e. matches that will yield small deltas) even if the bnodes do not match exactly. To this end, we

[7] We do the same in Alg_{Hung} in case of ties in costs.

should give priority (i.e. bring to the front part of the string) the items of the bnode that have lower probability to be changed from one version to the other.

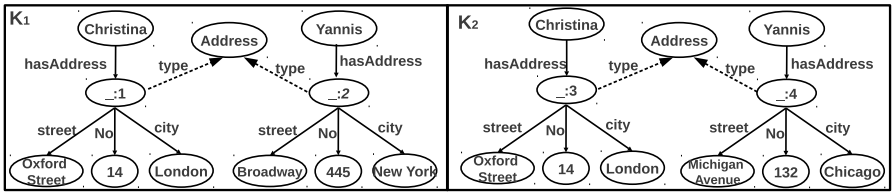

Figure 3.11: Two versions of an address Knowledge Base

Table 3.2: *Signatures* on bnodes of K_1 and K_2 of Fig. 3.11 according to the given option

Local Name	Signature
_ : 1	$ChristinahasAddress\diamondsuit typeAddress\diamondsuit cityLondon * No14 * streetOxfordStreet$
_ : 3	$ChristinahasAddress\diamondsuit typeAddress\diamondsuit cityLondon * No14 * streetOxfordStreet$
_ : 2	$YannishasAddress\diamondsuit typeAddress\diamondsuit cityNewYork * No445 * streetBroadway$
_ : 4	$YannishasAddress\diamondsuit typeAddress\diamondsuit cityChicago * No132 * streetMichiganAvenue$

Consider bnode _ : 1 of Figure 3.11 which is involved in the following triples: *Incoming*: $\{(Christina, hasAddress, \_:1)\}$, *Outgoing*: $\{(\_:1, street, OxfordStreet)$, $(\_:1, No, 14)$, $(\_:1, city, London)\}$, *Class Type*: $\{(\_:1, typeOf, Address)\}$. Each of these triples will be mapped to a string (e.g. "ChristinahasAddress" for the triple $(Christina, hasAddress, \_:1)$). The set *Class Type* contains the triples with the `rdf:type` ("type" in the figure) property of the respective bnode. For the three different sets of triples (Incoming, Outgoing, Class Type) we are going to construct three sets of substrings respectively (i.e. "$cityLondon\diamondsuit No14\diamondsuit streetOxfordStreet\diamondsuit$" for the Outgoing set of triples).The substrings inside each set are sorted lexicographically and separated by a special character, here denoted by $*$.

The triples inside these sets are sorted lexicographically. The concatenation of these sets of substrings will yield the signature.

A key point is the order by which the sets are concatenated. One option is to give a first priority to the set of the incoming triples, a second priority to the set with type information (i.e. "typeAddress"), and the last priority to the set of the outgoing triples. We should also mention that inside the signature the sets are separated by a special character, here denoted by $\diamondsuit$. Table 3.2 shows the signatures of all the bnodes of Figure 3.11 according to this option. The proposed ordering of the substrings inside the signature stems from the assumption that the probability for the outgoing statements to change is higher than the ingoing (i.e. like in Figure 3.11 where updating the address of a person is more probable than changing his/her name). A more stable argument for the proposed ordering is the fact that, according to [21], blank nodes occur most prevalently in the subject position than the object position due to the tree-based RDF/XML syntax. This structure yields the existence of a resource (i.e. the subject of the incoming triple) and its properties (i.e. the objects of the outgoing triples). There are more chances to face a change of properties of a resource than of the name itself of the resource (probably a URI). However it is worth investigating other signature construction methods. Under this assumption the ingoing statements should precede the outgoing inside the signature. Similarly for the class type of

the bnode, it is not usual to be changed from the one version to the other. [8] We represent the blank nodes which are subjects of incoming statements or objects of the outgoing statements, by a special character ♣ (i.e. we treat them as equal, as we did in the 2nd assumption of approximation version of Alg_{Hung}). The exact steps of the signature construction algorithm are shown in algorithm $Signature1$ at Figure 3.12.

Alg. Signature1
Input: a blank node b
Output: a string bs signature of the blank node
(1) $bs = null$
(2) $inSet = outSet = classSet = \emptyset$
(3) for $each\{(sub, pr, ob) \mid ob = b\}$
(4) if $(isBNode(sub))$
(5) $inSet = inSet \cup \{name(pr) + "\clubsuit"\}$
(6) else
(7) $inSet = inSet \cup \{name(pr) + name(sub)\}$
(8) for $each\{(sub, pr, ob) \mid sub = b\}$
(9) if (isTypeOf(pr))
(10) $classSet = classSet \cup \{name(pr) + name(ob)\}$
(11) else if $(isBNode(ob))$
(12) $outSet = outSet \cup \{name(pr) + "\clubsuit"\}$
(13) else
(14) $outSet = outSet \cup \{name(pr) + name(trim(ob))\}$
(15)sort$(inSet)$
(16)sort$(outSet)$
(17)for $(each\ strings \in inSet)$
(18) $bs = bs + s + "\diamondsuit"$
(19)for $(each\ strings \in classSet)$
(20) $bs = bs + s + "\diamondsuit"$
(21)for $(eachstrings \in outSet)$
(22) $bs = bs + s + "\diamondsuit"$
(23)return bs

Figure 3.12: Signature Construction Algorithm

3.5.5.2 The Lookup algorithm

The algorithm $SignatureMapping$ also uses a *Lookup* method, which is actually a slightly modified version of *Binary Search*. Instead of the classical $BinarySearch$, this algorithm *Lookup* always returns a string. If the *Lookup* succeeds, it returns the matching (i.e. the string that we search), otherwise it returns the closest (in lexicographical order) string. The closest string is defined as the string located either in the position the $BinarySearch$ stopped or in the exact previous (if the last iteration decreased the high value) or in the next (if the last iteration increased the low value) position. In order to make a decision among the two strings (signatures), we define the elements of a signature, say $elems(bs)$, as the number of the building blocks (i.e. substrings split by the character $\diamondsuit$) included in its string. The returned string among these two strings is the one that has as *elems* a value closer to the

[8] **Note that the assumption that the outgoing statements change more frequently than the ingoing is an assumption.**

elems of the searching string. The exact steps of the algorithm *LookUp* are shown in Figure 3.13.

Alg. Lookup
Input: a string bs and a sorted bag of strings BS
Output: the best matching string bs_2
(1) $a = NULL$
(2) $low = 0$
(3) $high = BS.length$
(4) while $(low < high)$
(5) $\quad mid = \lfloor (low + high)/2 \rfloor$
(6) $\quad bmid = BS[mid]$
(7) $\quad$ if $(bs < bmid))$
(8) $\quad\quad high = mid$
(9) $\quad$ else if $(bs > bmid)$
(10) $\quad\quad low = mid + 1$
(11) $\quad$ else return $bmid$ // exact match found
(12)end while
(13)if $(bs < bmid \wedge (|elems(bs) - elems(bmid)| > |elems(bs) - elems(BS(mid - 1))|)$
(15) $\quad$ return $BS[mid - 1]$
(16)else if $(bs > bmid) \wedge (|elems(bs) - elems(bmid)| > |elems(bs) - elems(BS(mid + 1))|)$
(18) $\quad$ return $BS[mid + 1]$
(19)else
(20) $\quad$ return $bmid$

Figure 3.13: Lookup algorithm

Applying the *SignatureMapping* algorithm in the simple example of Figure 3.11 we get the final mapping: _:1 ↔ _:3 and _:2 ↔ _:4. Going to a more complex example (i.e. with connected bnodes), like the one of Figure 3.5 the final mapping would be _:1 ↔ _:6, _:2 ↔ _:7, _:3 ↔ _:8, _:4 ↔ _:10, _:5 ↔ _:9.

Chapter 4

Experimental Evaluation

We performed experiments for evaluating the potential for delta reduction, equivalence detection and time efficiency, deviation from the optimal delta and scalability.

They were performed using Sesame RDF/S Repository (main memory), using a PC with Intel Core i3 at 2.2 Ghz, 3.8 GB Ram, running Ubuntu 11.10.

to be done

4.1 TestBed

In order to provide an integrated experimental evaluation experiments were performed over real and synthetic datasets.

Real Datasets. We used two real datasets available in the LOD cloud: the *Swedish open cultural heritage* dataset[1], and the *Italian Museums* dataset[2], published from LKDI[3]. From each one we downloaded two versions with a time difference of one week or month. A preprocessing was necessary for corrections (e.g. missing URIs for some classes) and for merging the files. The features of these two datasets are given in Table 4.1. In both datasets there are *no directly connected bnodes*.

Synthetic Datasets. Although semantic data generators already exist in the

[1]http://thedatahub.org/dataset/swedish-open-cultural-heritage used from http://kringla.nu/kringla/ for providing information on cultural data of Sweden.

[2]http://thedatahub.org/dataset/museums-in-italy

[3]http://www.linkedopendata.it/

Table 4.1: Features of two real LOD datasets

	Swedish		Italian	
	File 1	File 2	File 1	File 2
Date	15/10/11	22/10/11	2/11/11	4/12/11
$\|Triples\|$	3,750	3,589	49,897	49,897
$\|BNodes\|$	535	509	6,390	6,390
$\|Triples$ with $bnodes\|$	77.7%	77.2%	43.85%	43.85%
Total Size	378 KB	365 KB	5.49 MB	5.46 MB

to be done

Table 4.2: Experimental results over real datasets

	Swedish				Italian			
	without BM	*with BM (bnode matching)*			*without BM*	*with BM (bnode matching)*		
		Random	Alg_{Hung}	Alg_{Sign}		*Random*	Alg_{Hung}	Alg_{Sign}
$\|Added\|$	2,805	2,726	75	127	21,885	19,762	3	3
$\|Deleted\|$	2,966	2,887	236	288	21,885	19,762	3	3
$\|\Delta_e\|$	5,771	5,613	311	419	43,770	39,524	6	6
BLoad Time(ms)	-	631	630	634	-	428	423	421
SC Time(ms)	-	-	-	210	-	-	-	840
BM Time(ms)	-	1.3	5,391	130	-	4.9	576,592	82.5
Diff Time(ms)	55	64	30	47	145	166	169	163
Tuning Time(ms)	-	15	0.2	0.5	-	3,332	9.4	9.5
Total Time(ms)	57	715	5,931	1,024	147	3,935	577,197	1,521

bibliography, none of them deals with the blank node connectivity issues. Therefore we designed and developed a synthetic generator over the UBA (Univ-Bench Artificial data generator) [16] that provides control over the bnode structures of the generated datasets. Each dataset corresponds to an RDF graph G.

4.2 Evaluation: not directly connected bnodes

Experiments were conducted *with* and *without* bnode matching. Regarding matching we tested: (a) the *random*, (b) the *Hungarian*, and (c) the *Signature*-based mapping methods. The results are shown in Table 4.2. The first rows show the size of the yielded deltas and the last rows the time required for loading the bnodes (BLoad), constructing signatures (SC), bnode maping (BM), delta computation (Diff), bnode name tuning (Tuning Time), and the total time. We observe that the algorithms provide a delta of 12.7 to $7,294$ times smaller than without bnode mapping. Alg_{Hung} yields an equal (for the Italian) or smaller (0.34 times smaller for the Swedish) delta than Alg_{Sign}, but it requires more time (from 15 to 624 times).

4.3 Evaluation: directly connected bnodes

4.3.1 On Complex Bnode structures

Let $Nodes$ be the set of all nodes in the graph, B be the set of bnodes ($B \subseteq Nodes$), and $conn(o)$ be the nodes of G that are directly connected with a node $o \in Nodes$. We define $b_{density}$ as $b_{density} = \text{avg}_{b \in B} \frac{|conn(b) \cap B|}{|conn(b)|}$. Note that if there are no directly connected bnodes then $b_{density} = 0$. The extended generator can create datasets with the desired $b_{density}$ and the desired maximum length of paths that consist of edges that connect bnodes (we denote by b_{len} their average).

Using the synthetic generator, we created a sequence of 9 pairs of KBs (each pair has two subsequent versions of a KB). For instance, the first KB is K_0 and its pair is K_0'.

Each time we compare the subsequent versions of a pair with respect to mapping time and yielded delta size. From now on we express the delta size as a

Table 4.3: Blank node Features of the synthetic dataset

K	$\|triples\|$	$\|B\|$	D_a	$b_{density}$	b_{len}	Optimal delta size	Variation
K_{0a}	5,846	240	13.4	0	0	1%	No connected blank nodes
K_{1a}	5,025	240	10.5	0.1	1	0.5%	*b_Neighborhoods* of 2 bnodes, reduced *b_named* triples
K_{2a}	2,381	240	7	0.15	1	1.5%	Reduced *b_named* triples
K_{3a}	1,628	240	5	0.2	1	1.5%	Reduced *b_named* triples
K_{4a}	1,636	240	5	0.2	1.15	1%	*b_Neighborhoods* of up to 8 bnodes
K_{5a}	1,399	240	4	0.25	1.15	1.7%	Reduced *b_named* triples
K_{6a}	919	240	3	0.32	1.15	3.2%	*b_Neighborhoods* of up to 15 bnodes, reduced *b_named* triples
K_{7a}	909	240	3.25	0.4	1.35	2.7%	Connect *b_Neighborhoods*, reduced *b_named* triples
K_{8a}	1,001	240	3.94	0.5	21.5	2.5%	Connect *b_Neighborhoods*

percentage of the number of triples of the KB, i.e. as $\frac{|\Delta_e(K,K')|}{\frac{|K|+|K'|}{2}}$. Table 4.3 shows the blank node properties of each pair of KBs, its optimal delta size over its subsequent version (known by construction) and its variation over the next pair of KBs (we call *b_Neighborhood* every subgraph having as nodes only bnodes, and we call *b_named* triple every triple that contains one bnode). With D_a we denote the average number of direct edges of the bnodes (i.e. average number of triples to which a bnode participates). to be done

4.3.2 Delta Reduction Potential

Figure 4.1(left) gives the delta reduction potential of each algorithm in logarithmic scale. Without bnode mapping the delta size ranges from 95% (for the second pair of KBs) to 143% (for the ninth pair of KBs). Instead for Alg_{Hung} it ranges from 0.47% to 10.67% and for Alg_{Sign} it ranges from 1% to 11.5%. Notice that Alg_{Sign} does not reduce the delta to the optimal for any pair of datasets, while Alg_{Hung} achieves the optimal delta for most of the pairs.
Figure 4.1 (right) shows the delta reduction potential for the same pairs with the difference that the two bnode lists are not scanned in the original order (as in the left figure), but the second list is reversed. We notice that as the areas of directly connected bnodes become bigger (after the sixth pair of datasets), we get different (here higher) deltas. In such areas the direct neighborhoods lose their discrimination ability and thus the delta reduction potential becomes more unstable, increasing the probability to get a bigger delta.

If we use the optimal delta as baseline, and compute the percentage $\frac{|\Delta_x|-|\Delta_{opt}|}{|\Delta_{opt}|}$, in the first diagram this percentage for Alg_{Hung} falls in [0, 2.88], while the Alg_{Sign}'s percentage falls in [0.4,3.2] (in the second diagram they fall in [0,8] and [0.4,8] resp.).

4.3.3 Time Efficiency

Figure 4.2 (left) shows the mapping times of each algorithm in logarithmic scale. Alg_{Sign} gives two orders of magnitude lower mapping times.

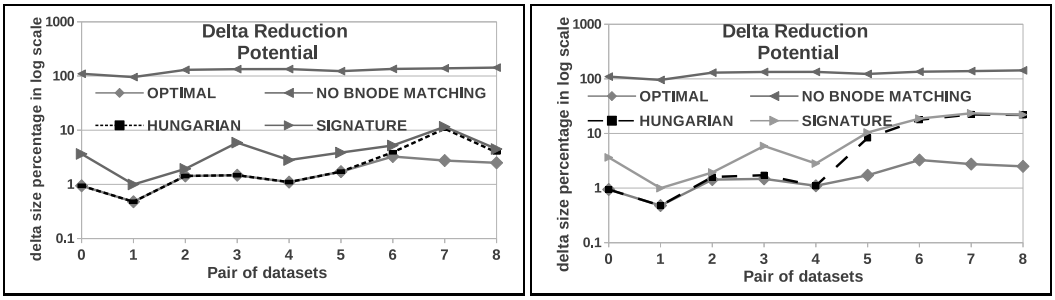

Figure 4.1: Delta Reduction over the synthetic datasets

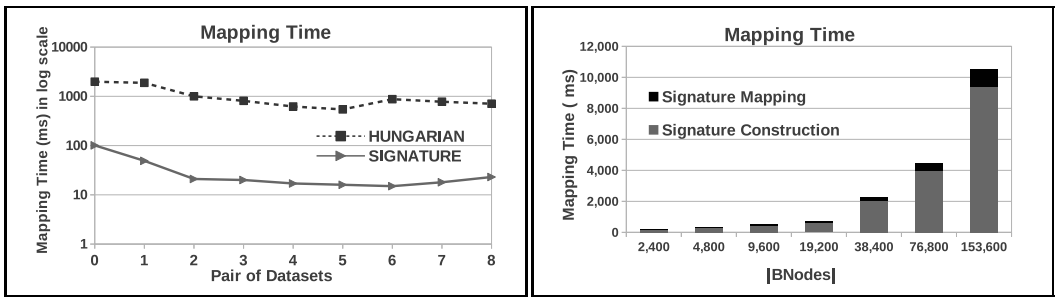

Figure 4.2: Mapping times over the synthetic datasets

4.3.4 Equivalence Detection Potential

Regarding equivalent KBs, if there are no directly connected bnodes then Alg_{Hung} detects them at polynomial time (recall Th. 4). To investigate what happens if there are directly connected bnodes we compared the pairs (Kia, Kia) for i=0 to 8 of the synthetic KBs. In case of similarly ordered bnode lists both Alg_{Hung} and Alg_{Sign} detected equivalences for all the KBs, while for reverse scanned bnodes lists they detected 5 of the 9 equivalences. They did not detect equivalences for the KBs with $b_{density} \geq 0.25$.

4.4 Scalability

To investigate the efficiency of Alg_{Sign} in bigger datasets, we created 7 pairs of KBs: the first pair contains 23,827 triples and 2,400 bnodes, the second pair has the double number of triples and bnodes, and so on, until reaching the last pair containing 153,600 bnodes. From Fig. 4.2 (right) we can see that the mapping time for Alg_{Sign} was only 10.5 seconds for the seventh pair of KBs (153,600 bnodes). Alg_{Hung} could not be applied even to the third pair of KBs due to its high (quadratic) requirements in main memory space.
The results are summarized in the concluding section.

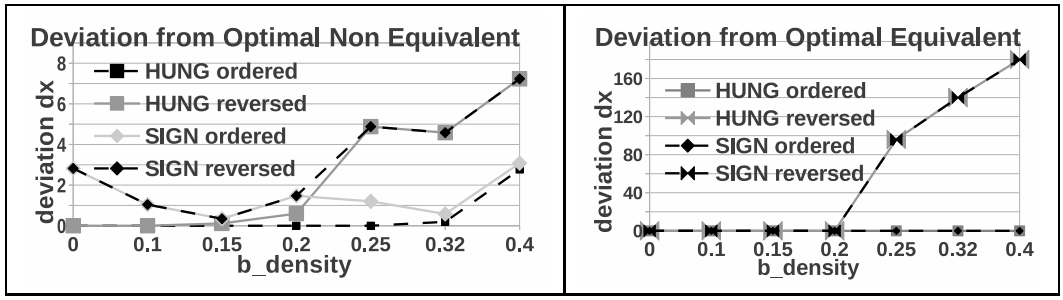

Figure 4.3: d_x over non equivalent (left) and equivalent (right) KBs

4.5 Measuring the approximation

The upper bound of the reduction of the delta size that can be achieved with bnode matching is the min number of bnodes of the two KBs multiplied by their average degree. Experimentally we have investigated whether $b_{density}$ (which is zero if there are no directly connected bnodes, and equal to 1 if all nodes are bnodes as in the proof of Th. 3), is related with the deviation from the optimal delta $d_x = \frac{|\Delta_x|-|\Delta_{opt}|}{|\Delta_{opt}|+1}$. Results over equivalent and non-equivalent KBs are shown at Figure 4.3. Both algorithms give a much smaller deviation from optimal than without bnode matching (its d_x ranges [47,114]). We also observe that keeping the original order of the bnodes is beneficial for both algorithms. For the non equivalent KBs the Alg_{Hung} gives always equal or (mostly) smaller delta than the Alg_{Sign}, while for the equivalent both algorithms give exactly the same deviation.

Chapter 5

System and Applications

Here we discuss the system that we have implemented and various applications.

5.1 Functionality

`BNodeDelta` supports all the bnode matching algorithms that have been described.

This tool contains two different versions. The first version is a command line version and the second one is a web based application. At this point we are going to give a brief presentation of these two versions.

Regarding the web based version, initially the user/client gives the two KBs to be compared (which can be given either as paths to local files of the client or through URLs). Then the server uploads the KBs through FTP and imports them in the local main memory repository or fetches them from the network through HTTP respectively.

Figure 5.1 shows a screen shot, where the user has uploaded the two KBs and selected the RDF format of these KBs.

Afterwards, the user/client specifies the bnode matching algorithm to be used and selects some extra output options. Figure 5.2 shows such a screen shot.

After these two steps the user/client submits the request and the `BNodeDelta` outputs:

- a file with the deleted and the added statements (or one file with the deleted statements and one file with the added statements) in the selected RDF format
- statistics regarding the delta
- a file with statistics regarding the KBs (optional)
- a static visualization of the delta as a graph, where the red parts are the deleted triples and the green parts are the added triples (only in case the delta size is less than sixty triples)

Figure 5.3 and 5.4 show the exported results from the BNode matching over the Address KBs using the Alg_{Hung}.

An optional step can take as input a *namespace mapping table* (if a namespace $nm1$ is mapped to a $nm2$ then they are considered equal at the comparison

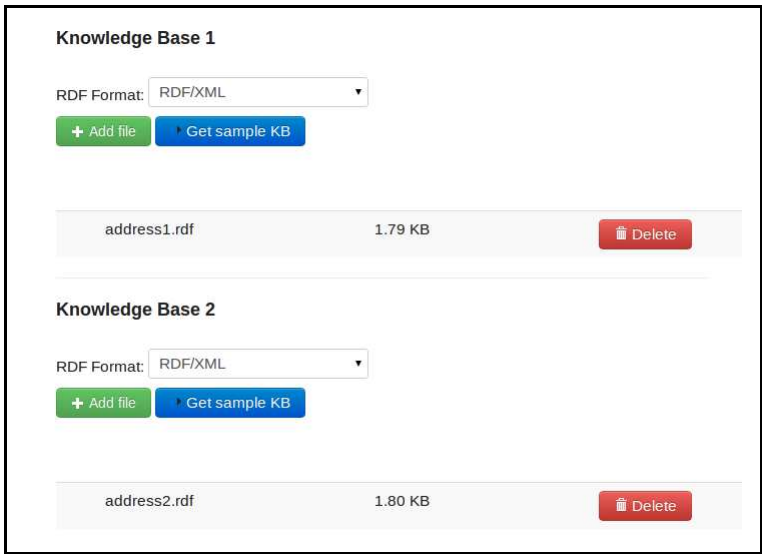

Figure 5.1: Importing the KBs on `BNodeDelta`

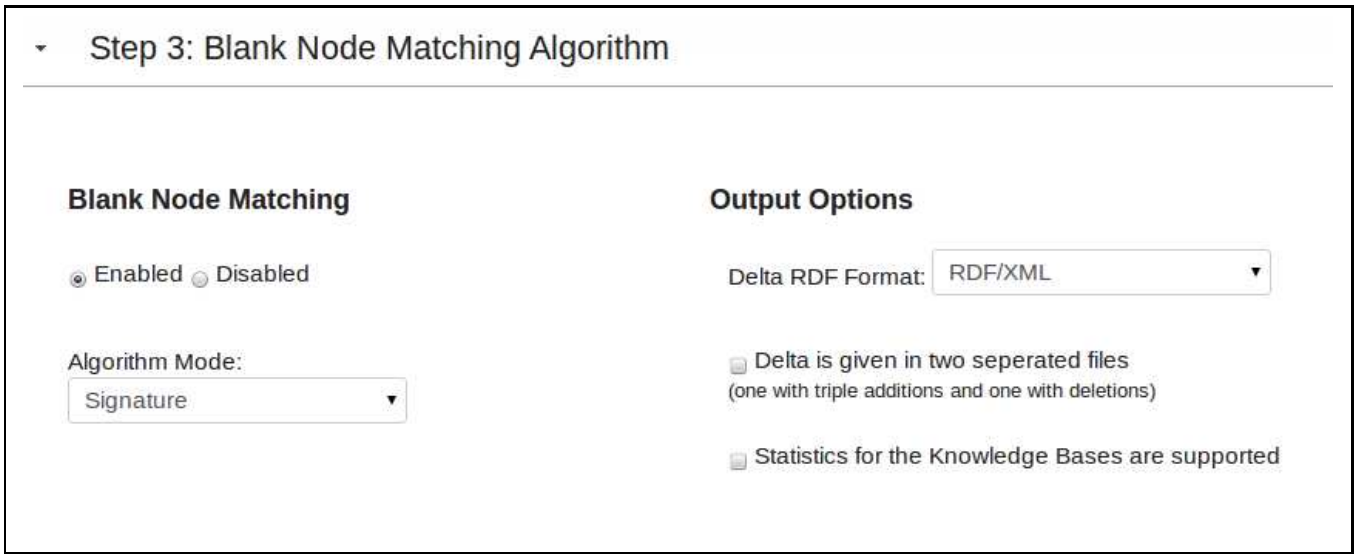

Figure 5.2: Selecting the BNode matching algorithm on `BNodeDelta`

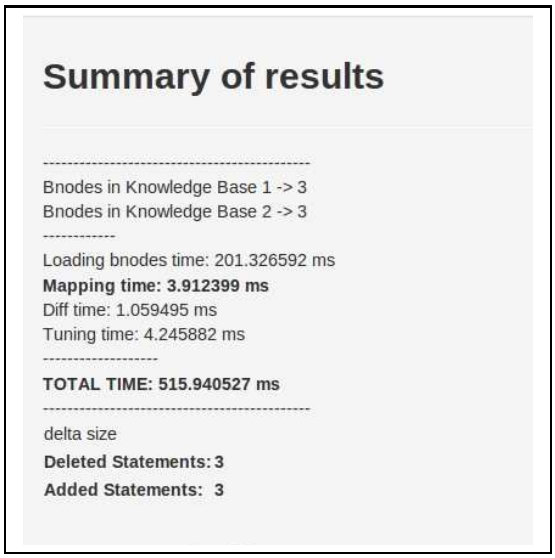

Figure 5.3: Basic Statistics of the BNode matching

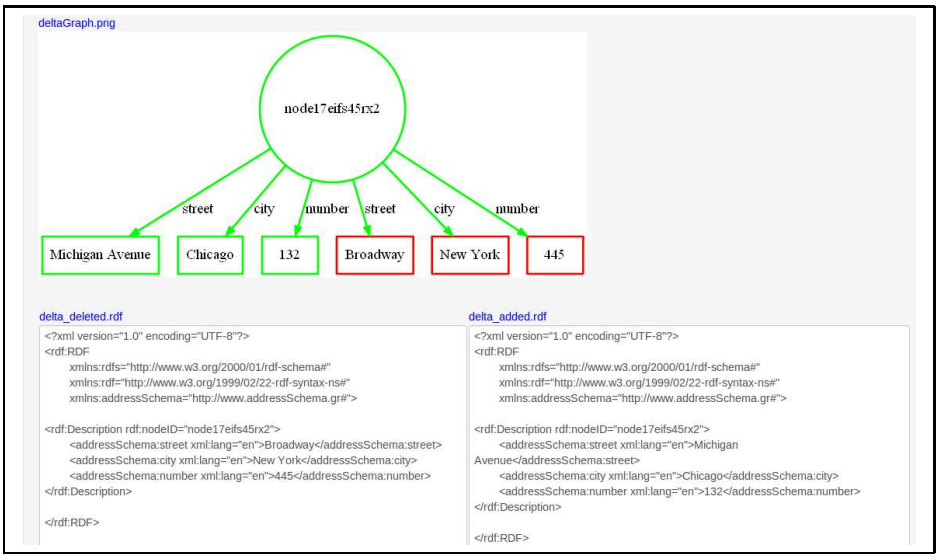

Figure 5.4: The exported files of `BNodeDelta`

phase). Such a phase can be really useful, in case the namespace URI references has changed from the one version to the other, but we do not to this kind of change to impact on the exported detla.

Regarding the command line version of `BNodeDelta`, the user (human or other program) can execute the jar file through the command line. The input is now given as parameters of the jar file.

5.2 Architecture

`BNodeDelta` was developed over the Sesame Virtuoso provider, which is a fully operational Native Graph Model Storage Provider for the Sesame Framework, allowing users of Virtuoso to leverage the Sesame Framework to modify, query, and reason with the Virtuoso quad store using the Java language. Sesame Framework's purpose is to provide a Java-friendly access point to Virtuoso. For the needs of `BNodeDelta` Sesame is used as a Java library. It provides us with the necessary tools to parse, interpret, query and store all this information, embedded in `BNodeDelta`, while abstracting the details of the underlying machinery. Figure 5.5 shows an overview of Sesame's architecture.

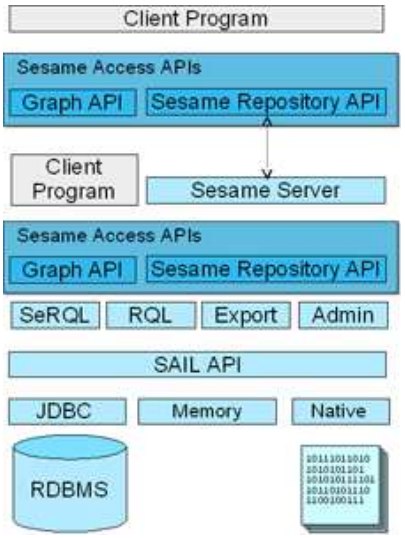

Figure 5.5: The Sesame component Stack

Starting at the bottom, the Storage And Inference Layer, or SAIL API, is an internal Sesame API that abstracts from the storage format used (i.e. whether the data is stored in an RDBMS, in memory, or in files, for example), and provides reasoning support. Each Sesame repository has its own SAIL object to represent it. For the storage of `BNodeDelta`'s data, we used the Main Memory object with its two different implementations. The first implementation is the one that operates directly on top of a SAIL object and the second operates as a proxy to a Sesame repository available on a remote server, accessible through HTTP.

On top of the SAIL, we find Sesame's functional modules, such as the SeRQL, RQL and RDQL query engine, the admin module, and RDF export. `BNodeDelta`

uses the SeRQL language to extract the blank nodes as well as the triples containing blank nodes. Access to these functional modules is available through Sesame'As Access APIs, consisting of two separate parts: the Repository API and the Graph API. The Repository API provides high-level access to Sesame repositories, such as querying, storing of rdf files, extracting RDF, etc. The Graph API provides more fine-grained support for RDF manipulation, such as adding and removing individual statements, and creation of small RDF models directly from code.

Sesame'As main memory model is primarily graph-based, where URIs are nodes, and triples are a pair of edges (an edge from subject to predicate, and an edge from predicate to object) each. This provides a quite simple view of RDF data, that allows someone to navigate uniformly the logical RDF graph, ignoring RDF intricacies where they are irrelevant. Sesame'As main memory model is triple-based view oriented, a choice that at least affords its API to be very concise.

5.2.1 Applications

At this section we focus on applications that could take advantage of our work in order to integrate their systems.

5.2.1.1 Jena rdfcompare

Jena is an open source Semantic Web framework for Java, grown out of work with the HP Labs Semantic Web Programme. It provides an API to extract data from and write to RDF graphs. The graphs are represented as an abstract "model that can be sourced with data from files, databases, URLs or a combination of these. Jena is actually similar to Sesame; though, unlike Sesame, Jena provides support for OWL (Web Ontology Language).

The Jena's *rdfcompare* is a command line tool written in java which loads two RDF files into Jena RDF models and uses an API call to check if the models are isomorphic. Although it seems to correctly tell whether two graphs are isomorphic, Can compare two files in different RDF formats, it doesn't give any analysis of the difference between the files, like someone would expect from UNIX diff.

The existence of blank nodes is the main reason that frameworks have not elaborated integrated versioning systems. However, this tool could make use of the Alg_{Sign}in order to provide in a tolerable time a quite small delta.

5.2.2

Chapter 6

Conclusion and Future Work

In this paper we showed how we can exploit bnode anonymity to reduce the delta size when comparing RDF/S KBs. We proved that finding the optimal mapping between the bnodes of two KBs, i.e. the one that returns the smallest in size delta regarding the *unnamed* part of these KBs, is NP-Hard in the general case, and polynomial in case there are not directly connected bnodes. To cope with the general case we presented polynomial algorithms returning approximate solutions.

In real datasets with no directly connected bnodes Alg_{Sign} was two orders of magnitude faster than Alg_{Hung} (less than one second for KBs with 6,390 bnodes), but yielded up to 0.34 times (or 34%) bigger deltas than Alg_{Hung}, i.e. than the optimal mapping. Alg_{Hung} also identified all equivalent KBs.
For checking the behavior of the algorithms in KBs with directly connected bnodes, we created synthetic datasets, over which we compared Alg_{Sign} and the Alg_{Hung} *approximation* algorithm. Alg_{Hung} yielded from 0 to 3 times smaller deltas than Alg_{Sign}, but the latter was from 18 to 57 times faster. Alg_{Sign} requires only 10.5 seconds to match 153,600 bnodes.

This is the first work on this topic. Several issues are interesting for further research. For instance, it is worth investigating other special cases where the optimal mapping can be found polynomially (e.g. directly connected bnodes that form graphs of bounded tree width). Another direction is to comparatively evaluate various (probabilistic) signature construction methods and greedy approximation algorithms.

Software and datasets are available to download and use from:
`http://www.ics.forth.gr/isl/BNodeDelta`. Finally, in the case of very large KBs that do not fit in memory it would be possible to have a diff implementation over RQL.

Bibliography

[1] Resource Description Framework (RDF):Concepts and Abstract Syntax.

[2] S. Agarwal, D. Starobinski, and A. Trachtenberg. "On the Scalability of Data Synchronization Protocols for PDAs and Mobile Devices". In *IEEE Network (Special Issue on Scalability in Communication Networks), Vol. 16, No.4, pp.2228*, 2002.

[3] S. Auer. "Powl - A Web Based Platform for Collaborative Semantic Web Development". In *Proc. of 1st Workshop Workshop Scripting for the Semantic Web (SFSW'05)*, Hersonissos, Greece, May 2005.

[4] T. Beners-Lee and D. Connoly. "Delta: An Ontology for the Distribution of Differences Between RDF Graphs", 2004. http://www.w3.org/DesignIssues/Diff (version: 2006-05-12).

[5] T. Beners-Lee, D. Connoly, and S. Hawke. Semantic web tutorial using n3. In *Twelfth International World Wide Web Conference*, Budapest, Hungary, May 2003.

[6] B. Berliner. "CVS II: Parallelizing Software Development". In *Procs of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990.

[7] François Bourgeois and Jean-Claude Lassalle. An extension of the Munkres algorithm for the assignment problem to rectangular matrices. *Commun. ACM*, 1971.

[8] J. J. Carroll. "Matching RDF graphs". In *Procs of the ISWC'02*, pages 5–15, Italy, Oct. 2002.

[9] Lei Chen, Haifei Zhang, Ying Chen, and Wenping Guo. "Blank Nodes in RDF". ISWC'12, 2012.

[10] S. Chien, V. Tsotras, and C. Zaniolo. Eficient Management of Multiversion Documents by Object Referencing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 291–300, Roma, Italy, 2007.

[11] R. Cloran and B. Irwin. "Transmitting RDF graph deltas for a Cheaper Semantic Web". In *Procs. of SATNAC'2005*, South Africa, September 2005.

[12] G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Heidelberg, Germany, 2001.

[13] L. Ding, T. Finin, A. Joshi, Y. Peng, P. da Silva, and D. McGuinness. "Tracking RDF Graph Provenance using RDF Molecules". In *Procs of ISWC'05*, Galway,Ireland, November 2005.

[14] Li Ding, Tim Finin, Yun Peng, Paulo Pinheiro da Silva, and Deborah L. McGuinness. Tracking RDF Graph Provenance using RDF Molecules. Technical report, UMBC, April 2005.

[15] G. Flouris. *"On Belief Change and Ontology Evolution"*. PhD thesis, Computer Science Department, University of Crete, Greece, 2006.

[16] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. In *Selected Papers from the Intern. Semantic Web Conf. ISWC, 2004*.

[17] P. Hayes. "RDF Semantics, W3C Recommendation", 2004.

[18] J. Heflin, J. Hendler, and S. Luke. "Coping with Changing Ontologies in a Distributed Environment". In *Procs of AAAI-99 Workshop on Ontology Management*, Florida, July 1999.

[19] M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. "Ontology versioning and change detection on the web". In *Procs of EKAW'02*, pages 197–212, Siguenza, Spain, Oct 2002.

[20] M. Klein and N. Noy. "A component-based framework for ontology evolution". In *InWorkshop on Ontologies and Distributed Systems at IJCAI-03*, Acapulco, Mexico, 2003.

[21] A. Mallea, M. Arenas, A. Hogan, and A. Polleres. On blank nodes. In *Procs of the 10th Intern. Semantic Web Conference (ISWC 2011)*. Springer, October 2011.

[22] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-Centric Management of Versions in an XMLWarehouse. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 581–590, Roma, Italy, 2001.

[23] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. In *International Symposium on Information Theory*, page 232, 2001.

[24] J. Munkres. Algorithms for the assignment and transportation problems. *J-SIAM*, 5(1), 1957.

[25] M. Klein N. F. Noy, S. Kunnatur and M. A. Musen. "Tracking Changes During Ontology Evolution". In *Procs of ISWC'04*, pages 259–273, Hisroshima, Japan, November 2004.

[26] A. Newman, YF Li, and J. Hunter. A scale-out rdf molecule store for improved co-identification, querying and inferencing. In *Intern. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2008.

[27] N. F. Noy and M. A. Musen. "PromptDiff: A Fixed-point Algorithm for Comparing Ontology Versions". In *Procs of AAAI-02*, pages 744–750, Edmonton, Alberta, July 2002.

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

[29] P. Plessers and O. De Troyer. "Ontology Change Detection Using a Version Log". In *Procs of ISWC'05*, pages 578–592, Galway,Ireland, November 2005.

[30] B. Schandl. Replication and versioning of partial rdf graphs. ESWC'10, 2010.

[31] Y. Theoharis, V. Christophides, and G.Karvounarakis. "Benchmarking Database Representations of RDF/S Stores". In *Procs of ISWC'05*, pages 685–701, Galway, Ireland, Nov 05.

[32] A. Tridgell. "Efficient algorithms for sorting and synchronization". Phd thesis, The Australian National University, February 1999.

[33] G. Tummarello, C. Morbidoni, R. Bachmann-Gmur, and O. Erling. RDFSync: efficient remote synchronization of RDF models. In *(ISWC-07)*, 2007.

[34] Giovanni Tummarello, Christian Morbidoni, Paolo Puliti, and Francesco Piazza. Signing individual fragments of an rdf graph. In *Proceedings of the World Wide Web conference*, 2005.

[35] M. Volkel, W. Winkler, Y. Sure, S. Ryszard Kruk, and M. Synak. "SemVersion: A Versioning System for RDF and Ontologies". In *Procs of ESWC'05.*, Heraklion, Crete, May 2005.

[36] D. Zeginis, Y. Tzitzikas, and V. Christophides. "On the Foundations of Computing Deltas Between RDF Models". In *Proceedings of the 6th International Semantic Web Conference (ISWC-07)*, Busan, S. Korea, 2007.

[37] D. Zeginis, Y. Tzitzikas, and V. Christophides. "On Computing Deltas of RDF/S Knowledge Bases". *ACM Transactions on the Web (TWEB)*, 2011.

[38] Z. Zhang, L. Zhang, C. Lin, Y. Zhao, and Y. Yu. "Data Migration for Ontology Evolution". In *Poster Proceedings ISWC'03*, Sanibel Island, Florida, USA, October 2003.

Chapter 7

Appendix: Proofs