Data source synchronization in cloud-based triple stores

Owusu, E.B.

Award date:
2014

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Data source synchronization in cloud-based triple stores

Master's Thesis

Evans Boateng Owusu

Supervisor:

dr. G. H. L. Fletcher

Assessment Committee:

dr. G. H. L. Fletcher
dr. ir. Joaquin Vanschoren
dr. Dirk Fahland

Eindhoven, August 2014

Abstract

Data integration and warehousing have many advantages for enterprises such as: allowing a central view of data across all branches, providing a means to draw meaningful relations among data, providing consistency in all data and allowing a single standard model for all data. As organizations grow and expand, large quantities of data are produced and if not properly managed, an organization may miss out on the benefits of data integration and management. Apart from the extra cost of data integration, warehousing and management, an organization may not wish to take on such extra responsibility, which requires expertise often different from the primary focus of the organization. It is in this light that Semaku B.V. is building a platform to provide a means for organizations to unleash their data into repositories and enjoy the benefits that come with data integration and warehousing. The platform will also provide a means to synchronize an organizations' source data with the repository to allow easy propagation of updates. This thesis focuses on selected challenges in realizing the platform: data transformation, change detection and update execution. We study existing data format transformation, and change detection approaches and propose new and improved algorithms. We propose generic means to transform data, which is completely automated and preserves structure of data to allow round tripping. To allow organizations to customize and restructure their data, we propose a semi-automated semantic transformation process. We also propose a means to execute updates between synchronized data. Furthermore, we implement a data transformation platform which allows both generic and semantic transformations of selected popular data formats (XML, JSON, CSV) to the W3C's Resource Description Framework (RDF) format. We also propose a simple vocabulary for describing changes in data and based on this vocabulary, an update engine for executing updates, between synchronized data sources, was implemented.

Preface

This thesis is the result of my final graduation project for the master program in Embedded Systems at Eindhoven University of Technology. This project was conducted in the Web Engineering group, Department of Mathematics and Computer Science, in cooperation with Semaku B.V.

I would like to express my greatest gratitude, first and foremost, to George Fletcher for his supervision, advice and critical feedback.

Secondly, I would like to express my gratitude to my supervisors at Semaku B.V, John Walker, Tim Nelissen and Evgeny Knutov for their feedback. I would also like to thank the assessment committee members, Joaquin Vanschoren and Dirk Fahland, for reviewing my thesis and attending my presentation and, for their critical feedback.

Finally, I would like to thank my family and friends, especially Kadian Davis, for their help and support.

Contents

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Enterprises produce data in diverse formats and in different locations. Systems that generate data are usually not the same ones that consume it. Getting all of an enterprise's data together and managing a single data silo or repository is important in providing users a unified view of the data. Semaku B.V is developing a corporate semantic platform that will enable enterprises to transform, load and manage their data in a flexible way. The semantic platform will provide a means for enterprises to integrate and store their data in a repository in the cloud to provide the combined advantages of data integration and warehousing. Some of the advantages are:

1. it enables a central view of data across all branches of enterprise.
2. gathering data from multiple sources into a single repository allows a single query engine to be used to present data.
3. It provides consistency in the way an organization's data is presented.
4. It provides a single standard data model for all data irrespective of the data source and
5. it allows data to be restructured to fit the needs of users.

The semantic platform will also provide a means to synchronize source data with the repository, monitor changes and propagate changes when they occur. Consider an application that queries and presents data in a repository; when the source data, which was transformed and stored in the repository, changes, the change is detected and propagated to the repository in order to provide data that is up to date to users. In the same way, if the application makes changes to the data in the repository, the changes are detected and propagated back to the source data so that the source and the target are consistent. Data synchronization provides consistency and freshness among connected data sources. It also provides a way to monitor the evolution of data. Realization of this platform requires knowledge areas as data extraction and transformation, data propagation and synchronization, change detection and update execution.

In this work, we deal with two main aspects of the platform. The first is *data transformation*, which consists of applying a series of rules to a source data to convert it into the format of a target storage system. For the central repository, all source data will be transformed into one standard format, Resource Description Framework (RDF). This choice of data format was not random as RDF in recent times has become a popular standard for data representation and exchange. RDF is supported by its own query language called SPARQL. In this work, we consider a subset of the most popular enterprise data formats: XML, JSON and CSV and how they can be efficiently transformed to RDF. We analyze and propose solutions for efficient data transformation in relation to automation of the transformation process, round-tripping of data, semantic relevance and flexibility. With respect to automation, little or no human intervention in the transformation process is necessary for an autonomous systems. Also we investigate how semantically relevant the transformation could be and how data structure and content could be preserved in the process. The platform within which the transformation takes place should be flexible enough to integrate with existing data systems.

The second aspect of the semantic platform considered is *change detection and update execution*. Change detection is the process of detecting, calculating and representing differences (delta) between two documents or data mostly of the same format. Data extracted and loaded into the repository may be used to feed applications such as corporate websites, etc. and these applications need to stay up to date all the time. A change in the source data should therefore be efficiently detected and propagated to the repository in order to ensure freshness of data used by the applications. An important part of the change detection and update execution is the format in which data is represented. In this work, we examine existing algorithms for change detection in XML and propose a modified version of X-diff, an algorithm for detecting changes in XML documents Wang et al. (2003). We also examine change detection in RDF knowledge bases and propose applicable algorithms. We further propose a language for representing changes and then develop a prototype tool for executing updates assuming changes have been successfully detected.

1.2 Contributions

This project presents a data transformation process that incorporates generic and semantic transformations of XML, JSON and CSV to RDF. The transformation process presents two two novel things: the property of the generic transformation to preserve structure of data and to allow round-tripping and the use of a list of functions for semantic transformations of RDF triples generated from the generic transformation process. Also, we present a design and implementation of the data transformation platform. This project also discusses algorithms for change detection and proposes a language for describing the changes. An implementation of an update engine based on the vocabulary is demonstrated.

1.3 Outline of Thesis

The remainder of the thesis is organized as follows. Chapter 2 explains some major concepts and terms that will be used throughout the thesis. Related works are examined in the same chapter. Chapter 3 elaborates the theories and algorithms for the transformation of XML, JSON and CSV to RDF. In Chapter 4, we discuss change detection in XML and RDF and give elaborate algorithms. Also a language for representing data difference (delta) and how updates are executed are discussed in Chapter 4. Chapter 5 explains the development and implementation of a prototype data transformer and an update engine. In Chapter 6, we test and measure the performance of the software tools developed in Chapter 5. Chapter 7 summarizes the work done and gives conclusions.

Chapter 2

Background and Related Work

2.1 Introduction

In this chapter, we present brief explanations of concepts and terms used throughout this thesis. Where necessary, examples and diagrams are given to illustrate concepts to avoid in-depth explanations of broad terms. We also discuss related work.

2.2 RDF

Resource Description Framework (RDF), is a standard model for data exchange on World Wide Web and a W3C recommendation w3c (2014b). This data model is based on the representation of resources by uniform resource identifiers (URIs) and the description of the relations betweeen resources using a set of RDF statements referred to as *triples*. A triple consists of a *subject*, *predicate* and an *object*. A subject could either be a URI or a blank node, a predicate could only be a URI and an object could be a URI, a blank node or a literal. A blank node is used to represent a resource, which does not have a URI. Basically, the predicate of a triple shows a relation between the subject and the object. A set of triples represents a labelled directed graph where subjects and objects are nodes and predicates are edges. A statement such as *"Nelson is a driver who drives buses"* can be represented by two triples as shown in Listing 2.1

```
<ex:Nelson rdf:type ex:Driver> .
<ex:Nelson ex:drives ex:Buses> .
```

Listing 2.1: Example of triples

where *ex* and *rdf* are prefixes for the URIs *http://example.org/* and *http://www.w3.org/1999/02/22-rdf-syntax-ns#* respectively. A simple directed graph representing these triples is shown in figure 2.1.

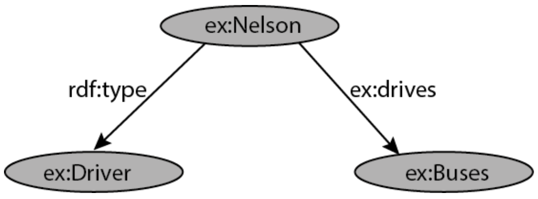


Figure 2.1: Sample rdf graph

Commonly used serialization formats for RDF are: Turtle, N-Triples, RDF/XML and JSON-LD. The RDF/XML and Turtle representation of the example are shown in Listing 2.2 and Listing 2.3.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns/#"
xmlns:ex="http://example.org/" >
  <rdf:Description rdf:about="http://example.org/Nelson" >
    <rdf:type rdf:resource="http://example.org/Driver" />
    <ex:drives rdf:resource="http://example.org/Buses" />
  </rdf:Description>
</rdf:RDF>
```

Listing 2.2: RDF data in RDF/XML format.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns/#> .
@prefix ex: <http://example.org/> .

ex:Nelson rdf:type ex:Driver .
ex:Nelson ex:drives ex:Buses .
```

Listing 2.3: RDF data in Turtle format.

2.3 SPARQL

SPARQL is a language for querying data stored in RDF format. It is a W3C recommedation for querying RDF endpoints w3c (2008). SPARQL has a close resemblance to SQL. There are five query forms which use pattern matching to form result sets or RDF graphs. The query forms are:

1. SELECT: returns variables bound in a pattern match.
2. CONSTRUCT: creates new RDF triples or graphs.
3. ASK : tests whether a query pattern has a solution or not.
4. UPDATE: provides operations to update, create and remove triples from an RDF endpoint.
5. DESCRIBE: returns a description and other potentially relevant details of RDF triples.

Listing 2.4 shows a sample SPARQL query.

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

select ?x
where { ?x rdf:type ex:Driver . ?x ex:drives ex:Buses }
```

Listing 2.4: SPARQL query expression

2.4 XML

XML, Extensible Markup Language, defines a simple set of rules for encoding data that is both human and machine readable w3c (2003). It specifies neither semantics nor tags and as such, users are free to define custom tags and the structural relationships between them. The success of XML as a model for data exchange on the web is largely attributed to its flexibility. The structure of an XML document can however be predefined in a Document Type definition (DTD) or an XML Schema. An example xml document is shown in Listing 2.5

```
<?xml version="1.0" encoding="UTF-8"?>
 <Demoboard identifier="OM7622">
  <DemoboardInformation>
   <Path>Development kits/Small signal RF/LNA</Path>
   <Ranking/>
   <DescriptiveSummaries>
    <DescriptiveSummary>
     <Header>Evaluation board content</Header>
    </DescriptiveSummary>
   </DescriptiveSummaries>
  </DemoboardInformation>
 </Demoboard>
```

Listing 2.5: Sample XML document

2.5 XQUERY

XQUERY is a query and functional programming language designed to query data stored in XML format w3c (2010). It consists mainly of FLWOR expressions where F stands for the *For* clause, which can be used to declare variables that iterate over XML sequences. The L stands for *let* and allows to bind values to variables. A condition for filtering variable bindings can be specified using the *where* (W) clause. *Order by* (O) orders the result set according to a given condition and *return* always come last in an XQUERY expression and defines the items that are included in the result. With a large number of built-in functions, XQUERY is a powerful language for arbitrary XML transformations. A simple XQUERY expression is shown in Listing 2.6.

```
for $x in doc("movies.xml")/genre/superhero
where $x/price > 10
order by $x/title
return $x/title
```

Listing 2.6: XQUERY expression

2.6 JSON

JSON, JavaScript Object notation, is a lightweight data format that uses human-readable text to transmit data consisting of key-value pairs or an array ECMA (2013). A key is a string in double quotes and a value can be a string, number, true, false, null, array or a nested object. The structure is as shown in figure 2.2. It is easy for humans to read and write. It is used as an alternative to XML. It is comparatively much easier for machines to parse and generate because the format is already available in many programming languages. A sample JSON data is shown in Listing 2.7.

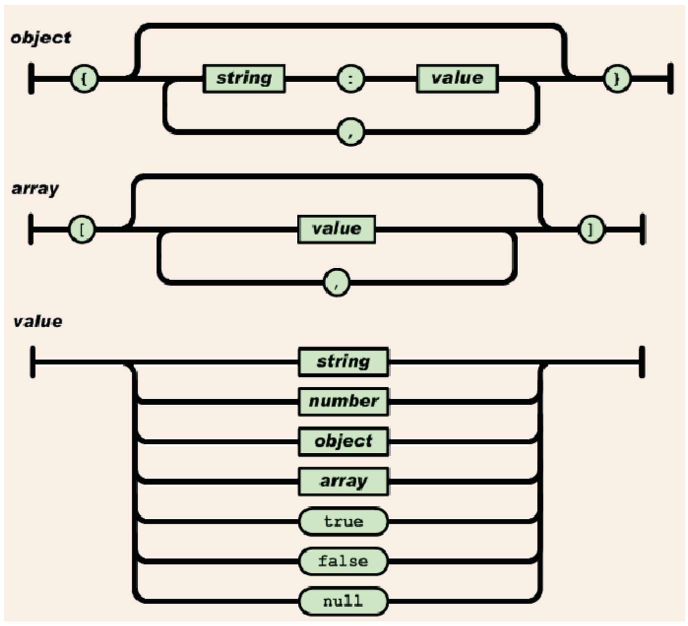


Figure 2.2: JSON structure

```
{
  "person":
  {
    "age": "40",
    "name": "John Smith"
  }
}
```

Listing 2.7: Sample JSON data

2.7 CSV

Comma-separated values or character-separated values (CSV) files store tabular data in plain-text form Shafranovich (2005). The file format is used in Microsoft excel and is popular even among non-Microsoft platforms. CSV has an advantage over XML specification in terms of overhead. CSV has much lower overhead, thereby using much less bandwidth and storage than XML. An example of CSV file is shown in listing 2.8.

```
Year,Make,Model,Description,Price
1997,Ford,E350,ac abs moon,3000.00
1999,Chevy,Venture Extended Edition,"",4900.00
```

Listing 2.8: CSV file

2.8 Related Work

2.8.1 XML to RDF

Several approaches for transforming XML to RDF have been proposed. Most of these approaches require human intervention and require different mapping rules for each XML transformation. Very few approaches have tried to completely automate the transformation process.

Melnik (1999) proposed a simplistic approach for transforming XML to RDF by a direct mapping of XML elements to RDF. In Melnik's approach, every XML element is transformed to RDF with child nodes as properties of parent nodes using a simplified RDF syntax. The approach applies the subject-predicate-object model of RDF preserving the XML structure in the final RDF model. The almost one-to-one mapping from XML to RDF helps preserve data and this makes transforming the generated RDF model back to XML much easier. Moreover, the approach requires no human intervention. However, it draws little semantic information from the XML files.

Thuy et al. (2008) exploits XML schema to transform XML documents to RDF. In their approach, element definitions stored in an XML Schema are mapped to RDF Schema ontology. The mapping is then used to transform XML files that conform to the XML Schema. The authors' claim of capturing implicit semantics in an XML document is not entirely true since an XML Schema is used to specify the structure prescriptions for XML documents and does not provide meaning to XML tags. Data type information is however relevant knowledge that may be captured in an XML Schema. The result of this approach is therefore not so different from Melnik's approach with the exception of the extra data type information.

Klein (2002) introduces a procedure for transforming XML to RDF by annotating the XML documents via an RDF-Schema specification. In his approach, an ontology is used to specify which elements and attributes in an XML document are interesting and what role they have, i.e., whether they specify a property or class. Not every syntactic structure is therefore converted into a statement in the RDF model. The tranformation of an XML document to RDF is based on the generated ontology, which did not follow a structured procedure but was more reliant on intuition, which requires human intervention. Though the approach draws meaningful information from XML documents, data is not preserved as many XML elements are absent in the final RDF model. The approach is not completely automated and hardly round-trippable.

To make the XML-RDF-XML transformation process completely automated, error-free and round-trippable, we propose a hybrid approach which uses a simplistic mappings to preserve data, and the use of XML schema, if available, to preserve meaningful information such as datatypes and namespaces. If necessary, more meaning will be added to the triples using a set of semantic mappings.

2.8.2 JSON to RDF

Despite JSON's popularity for exchanging data on the web and its serialization for RDF, not so much has been work done to transform an arbitrary JSON data to RDF and vice versa.

Izquierdo & Cabot (2013) undertook a project to discover the implicit schema of an arbitary JSON data. They proposed a model-based process composed of three phases: (1) pre-discovery phase extracting low-level JSON models out of JSON documents, (2) single-service discovery phase aimed at obtaining the schema information for a concrete service (inferred from a set of low-level JSON models output of different consecutive calls to the service), and (3) multi-service discovery phase in charge of composing the schema information obtained in the previous phase in order to get an overall view of the application domain. The resulting JSON schema is represented by the Ecore model. The Ecore metamodel can be mapped to the RDF schema making is easy to transform the discovered JSON to RDF. Their work however focused only on discovering the implicit JSON schema and nothing more.

In October 2011, John Boyer (2011) from IBM made a submission to W3C workshop in data and services integration. Their presentation highlighted their experience with transformations between XML and JSON. The authors grouped JSON into two categories: friendly JSON and unfriendly JSON and how to transform them to XML. A friendly JSON is characterized by: 1. Friendly JSON does not include repeating variable names. 2. Data types are directly associated with each variable. 3. Friendliness provides easy data structure consumption by JavaScript programmers, authors.

Listing 2.9 shows an example of a friendly JSON.

```
{
  "person":
    {
      "age": "40",
      "name": "John Smith"
    }
}
```

Listing 2.9: CSV file

Listing 2.9 shows an example of an unfriendly JSON.

```
{
  "book" : { "children :
  [{ "title" : { "attributes" :
  { "isbn": "15115115" },
  "children" :
  [ "This book is ", { "emph" : "bold" }] }
  }] }
}
```

Listing 2.10: CSV file

The JSON-LD project provides a JSON serialization for RDF data and also facilitates easy transformation of existing JSON into JSON-LD w3c (2014a). JSON-LD is a format for transporting RDF data as JSON. The transformation of an arbitrary well-formed JSON to JSON-LD revolves around the concept of context. A context specification provides mappings from JSON to RDF model by linking keys and values in JSON to concepts in the RDF ontology. In simple terms, a context is used to map terms in the JSON to IRIs. Listing 2.11 shows an example of JSON-LD with a context.

```
{
  "@context":
  {
    "name": "http://schema.org/name",
    "image": {
      "@id": "http://schema.org/image",
      "@type": "@id"
    },
    "homepage": {
      "@id": "http://schema.org/url",
      "@type": "@id"
    }
  },
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

Listing 2.11: JSON-LD context

The project provides a set of tools for transforming from different RDF serializations to JSON-LD and back.

2.8.3 Change detection

A plethora of studies have explored the possibility of calculating the difference between XML documents and RDF knowledge bases. We present a brief literature study of the works and tools developed in this regard.

Wang et al. (2003) proposed an algorithm for detecting changes between unordered XML models. Two XML documents are parsed as Xtrees and hash values are generated for the Xtrees. The detection process is then applied on the trees rather than on the document itself. The algorithm then generates a minimum-cost matching edit script. S. K. Lee & Kim (2006) also employ Xtree and hash value generation to detect changes between XML documents where order of data is taken into account. The approaches examined require the XML trees to reside in main memory therefore making them unscalable when applied to large data files. Chen et al. (2004) and Sundaram & Madria (2012) proposed algorithms for detecting changes in XML data sets stored in relational databases. The former assumes ordered XML data while the latter study assumes unordered XML data.

Volkel et al. (2005) proposed two algorithms. (1) A structure based algorithm, which compares the explicit set of triples of two RDF graphs and return the difference and (2) a semantic aware algorithm, which additionally considers inferred triples in the RDF graphs. Noy & Musen (2002) proposed a differential algorithm based on heuristic matching RDF concepts. Klein et al. (2002) allows users to specify conceptual relations that exist between ontology concepts to aid in an efficient recognition of the difference between two onotology versions. Zeginis et al. (2011) give a theoretical analysis of the various difference algorithms(RDFs differential functions). They give a detailed study of the deltas produced by comparing their size, semantic identity, redundancy, reversibility and composability. They then propose a more efficient algorithm for calculating RDFs deltas. Papavasileiou et al. (2013) proposed a change language capable of concisely and unambigously describing any possible change that could be encountered in an RDF. D.-H. Lee et al. (n.d.) suggested a more efficient approach in finding delta by partitioning triples based on their predicate. Triples with common predicates were grouped together and compared making it more efficient than a naive comparison of all triples.

2.9 Summary

We have explained some basic concepts that are used throughout this thesis. Extensive explanations are given where necessary in the subsequent sections. We have also discussed related work in data transformation and change detection.

Chapter 3

Data Transformation

3.1 Introduction

Data transformation is the process of converting data from one format (XML, JSON, CSV, RDF, etc) to another. Enterprises produce data in different formats and at different locations. Getting all the data together and managing a single data repository is important in providing users a unified overview of the data. The different source data are collected, transformed into a single standard format before being loaded into a repository. Data transformation involves mapping a source data to a target data and converting the source data based on this mapping. Data mapping defines the relationship between elements of the source and target data. In other words, data mapping defines the rules for data transformation.

In this project, we consider the transformation of XML, JSON and CSV to RDF and back. Most of the data produced today are in XML, JSON or CSV formats. These data sources feed many enterprise applications and websites. It is essential to transform data from their different formats to a chosen standard format so that applications that use these data can operate on a common data format to enhance application interoperability. Also, merging of data becomes much easier when data are in a common format. The chosen standard format is W3C's RDF data format.

In this section we describe a transformation process that will transform source data in XML, JSON or CSV to RDF and back. The transformation consists of two processes: a completely automated generic transformation and a semi-automated semantic transformation. Figure 3.1 gives an illustration.

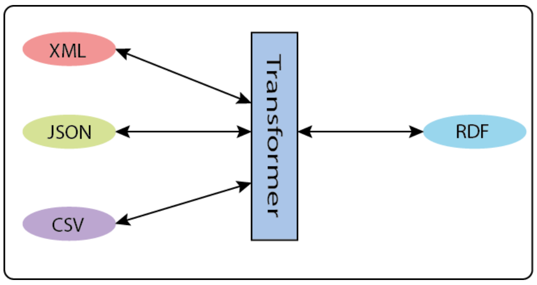


Figure 3.1: Data transformer

A source file in XML, JSON or CSV format is initially transformed into a generic RDF format and then specific semantic mappings are defined to further transform the triples.

3.2 Generic transformation

A generic set of rules are applied to the data source to transform it to RDF. This transformation process requires no knowledge of the source schema. The same rules are applied to all documents of the same format in the transformation.

3.2.1 XML to RDF

We assume every XML model has a default corresponding RDF model as proposed by Melnik (1999). XML child elements and attribute names are treated as properties of parent elements and consequently represented as predicates in a triple. Text elements and attribute values are treated as RDF literals. To preserve the structure and ordering of XML data, we extend the RDF/S vocabulary to include rdfx:pos, which identifies the positions of elements within a document. The root element is identified by *root*. The Listing 3.1 displays a sample XML document whose RDF model is shown in Figure 3.2.

```
<?xml version="1.0" encoding="UTF-8"?>
 <Demoboard identifier="OM7622">
  <DemoboardInformation>
   <Path>Development kits/Small signal RF/LNA</Path>
   <Ranking/>
   <DescriptiveSummaries>
    <DescriptiveSummary>
     <Header>Evaluation board content</Header>
    </DescriptiveSummary>
    <DescriptiveSummary>
     <Header>Applications</Header>
    </DescriptiveSummary>
   </DescriptiveSummaries>
  </DemoboardInformation>
 </Demoboard>
```

Listing 3.1: Sample XML document

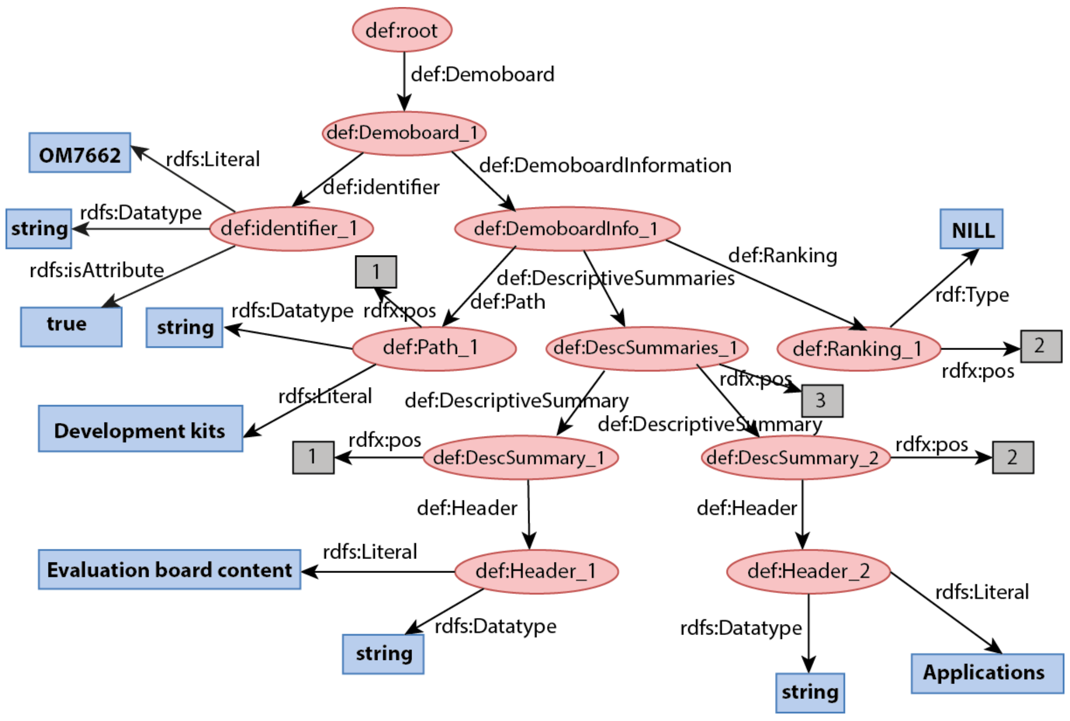


Figure 3.2: XML-RDF graph

The transformation process adds a little semantic information for software by deducing basic data types of the literals as shown in Figure 3.2. The generic transformation algorithm is described in Algorithm 1.

Algorithm 1 XML to RDF

```
if element is a complex type then
  if element is root then
    create the first statement:
    default_namespace:root element_name :id1
  else
    create statement:
    :id1 default_namespace:element_name :id2
  end if
  if element is child and its parent has more than one child then
    add position statement:
    :id2 rdfx:pos position_value
  end if
  if element has attributes then
    create these statements:
    :id1 default_namespace:attribute_name :id3
    :id3 rdfx:isAttribute "true"
    :id3 rdfs:literal literal_value
    :id3 rdfs:datatype datatype
  end if
end if
if element is a simple type then
  create these statements:
  :id1 default_namespace:element_name :id2
  :id2 rdfs:literal literal_value
  :id2 rdfs:datatype datatype
end if
if element has siblings then
  add position statement:
  :id2 rdfx:pos position_value
end if
Recursively apply algorithm on all child elements of the root element.
```

3.2.2 JSON to RDF

JSON has a key-value format and keys are treated as predicates in a triple. Since JSON is an unordered set of elements the order of values are not relevant except elements within an array. To preserve the order of elements within and array, rdfx:pos is used. Unlike JSON-LD, which requires a context specification to map keys and values to IRIs, we map all keys to a default IRI. The semantic transformation allows more meaningful mapping definitions. Data type information is also captured in this transformation. Listing 3.2 and Figure 3.3 illustrates the transformation process.

```
{
 "FileID": 12566,
 "BasicTypeNumbers": "BUK754R0-55B;BUK764R0-55B" ,
 "BasicTypes": ["BUK9Y12-55B", "BUK7Y12-55B"]
}
```

Listing 3.2: Sample JSON document

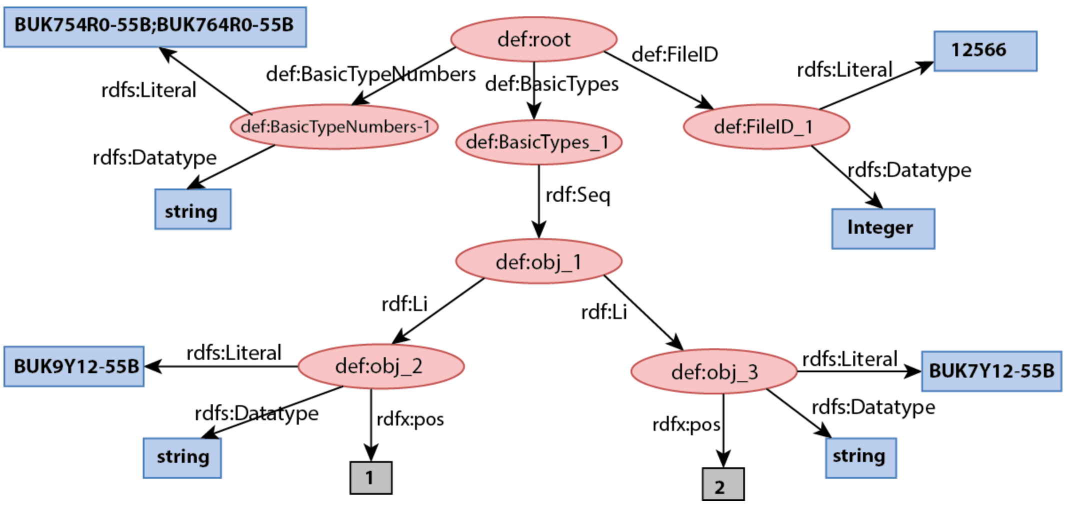


Figure 3.3: JSON-RDF graph

Algorithm 2 describes the algorithm

Algorithm 2 JSON to RDF

```
Represent parent JSON object or array by :root
# Keys are predicates
# Step 2:
On encountering a value create statement:
*:root default_namespace:key :id1*
# Step 3:
**if** value is a simple value (string, boolean, number) **then**
  create statements:
  *:id1 rdfs:Literal literal_value*
  *:id1 rdfs:Datatype datatype*
**end if**
**if** value is null **then**
  create statement:
  *:id1 rdf:Type nill*
**end if**
**if** value is an object **then**
  go to step 2 (use the appropriate subject)
**end if**
**if** value is an array **then**
  create statement:
  *:id1 rdf:Seq :id2*
  **for all** array values **do**
    create statement:
    *:id2 rdf:li :id2,...,number of array values*
    **if** array has more than one element **then**
      add positions to each element:
      *:id rdfx:pos position_value*
    **end if**
  **end for**
**end if**
Go to step 3 for each value
```

3.2.3 CSV to RDF

CSV stores data in a tabular format. Each row in the table is represented as a unique subject with the header titles as predicates and column values as objects of the triples. If the CSV document has no header, a custom header is generated. The positions of rows of the document is captured by the rdfx:pos. Data type information is also captured in the transformation. Listing 3.2 and Figure 3.4 illustrates the transformation.

```
Year, Model
1997, E350
1999, Venture Exteneded Edition
```

Listing 3.3: Sample CSV document

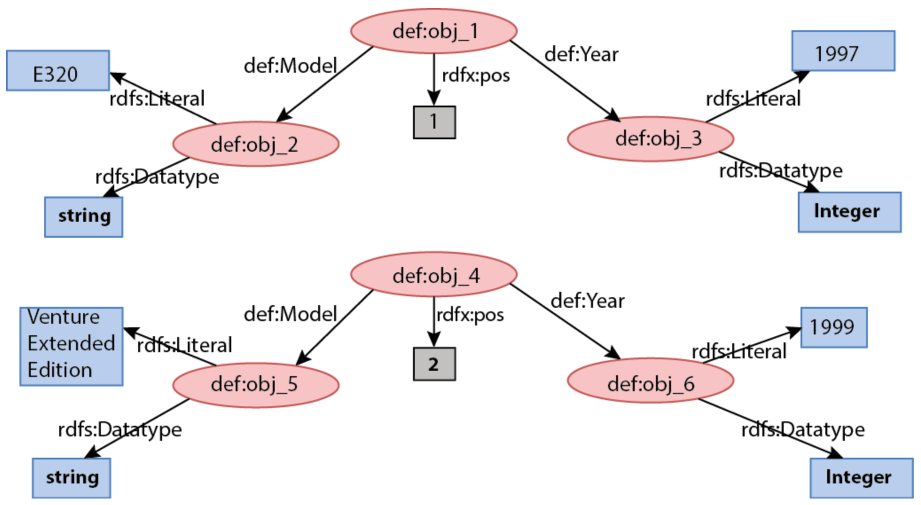


Figure 3.4: CSV graph

Algorithm 3 describes the algorithm for the transformation

Algorithm 3 CSV to RDF

```
for all rows do
    create a unique object as the subject of the statement:
    :id1
    add position statement:
    :id1 rdfx:pos position
    for all values in a row do
        create these statements:
        :id1 default_namespace:column_header :id2
        :id2 rdfs:literal literal_value
        :id2 rdfs:Datatype literal_datatype
    end for
end for
```

3.3 Semantic or model specific transformation

This phase of the transformation consists of a list of functions, which transforms an input set of triples into a desired format. The set of functions may consist of generic functions for data restructuring tasks such as string splitting, string concatenation, word replacement, language translation, exporting of data in specified formats, etc. It may also consist of custom functions defined to restructure specific data sets. So, given a set of triples T and a list of functions $< f_1, f_2, \ldots, f_n >$, we compute $T^{'} = f_n(f_{n-1}, \ldots, f_n(T))$, where $T^{'}$ is the newly transformed set of triples.

The list of functions are executed on the triples in the order in which they are specified in the *mapping rules.* Each function takes a reference to a local triple store and an optional list of string parameters. The implementation details of functions as a plugin is discussed in Section 5.3 Examples of functions and their definitions are shown in Examples 1, 2 and 3.

Example 1 (adding a namespace)**.**
Given a set of triples T a prefix, p and a URI, u
$add\_namespace(T, p, u) = T \bigcup \{p, u\}$

Example 2 (replacing a predicate)**.**
Given a set of triples T, predicates, p and $p^{'}$,
then for all triples $s\ p\ x \in T$,
$replace\_predicate(T, p, p^{'}) = (T \bigcup \{s, p^{'}, x\}) - \{s, p, x\}$

Example 3 (splitting a value)**.**
Given a set of triples T, a predicate, p and a separator, k,
then for all triples $s\ p\ x \in T$,
$split\_value(T, p, k) = (T \bigcup \{s, p, x_1, s, p, x_2, \ldots, s, p, x_n, \}) - \{s, p, x\}$
where $x = x_1 + x_2 + \cdots + x_n$.

3.4 Summary

We have presented a generic transformation process that converts a source data in XML, JSON or CSV into RDF. This transformation is based on a fixed set of mapping rules that can be applied automatically without any human intervention. The transformation captures data type information, and position of elements. A semi-automated semantic transformation using a list of functions further transforms data into a desired format to make data more meaningful. We have also presented algorithms for each transformation process.

Chapter 4

Change detection and Update execution

4.1 Introduction

The ability to automatically detect changes between different versions of data is very important in data synchronization. It is important for synchronized data to stay up-to-date and consistent. The difference between synchronized data sources needs to be detected, calculated and then propagated to enhance freshness and consistency of data. It is also essential for monitoring the evolution of data.

In this section, we assume an XML source, transformed with our proposed transformation engine to RDF, is synchronized with a triple store i.e an RDF graph. We discuss how changes in an XML source document can be efficiently calculated and transformed as an RDF change to update the corresponding RDF graph in the triple store. In much the same way, we consider how changes in the RDF graph can be efficiently calculated and transformed as an XML change to update the XML source document. Most of the ideas presented in this chapter are adapted from existing works. We however introduce novel propositions and algorithms for a more efficient change detection process. We also propose a simple vocabulary for describing and representing changes within XML and RDF. We then discuss how delta, based on the vocabulary, can be used to update an XML document or an RDF graph.

4.2 XML change detection

An XML tree may be ordered or unordered. In an ordered tree, both ancestor (parent-child) relationship and left-to-right ordering among siblings are significant whilst only ancestor relationships are significant in an unordered XML tree. Accordingly, algorithms for detecting changes can be grouped into two broad categories depending on whether they treat XML trees as ordered or unordered. In this context, we will recognize XML trees as unordered since they are applicable to a wide variety of XML documents. Unordered XML trees are problematic to round tripping since sibling elements' positions are not preserved. However, transforming RDF back to the original XML may not be feasible after many semantic transformations have been performed on the RDF. The semantic transformations may not be invertible therefore preventing round tripping. Applications that require the original XML documents will connect directly to the source systems. The change detection method proposed in this section is adopted from Wang et al. (2003). We are interested in three kinds of nodes in an XML tree:

1. Element node - Non-leaf node identified by a name label.
2. Text node - leaf node with a value.
3. Attribute node - Non-leaf node identified by a name and a value.

These edit operations are allowed on a tree:

- $insert(x(name, value), y)$: inserts x with name and value as a child node of y.
- $delete(x)$: deletes node x.
- $update(x, new\_value)$: changes the value of leaf node x to new_value. Only text nodes and attributes values can be updated. An update cannot modify a node's name.
- $insert(T_x, y)$: insert a subtree T_x as a child of y.
- $delete(T_x)$: delete subtree T_x.

$insert(T_x, y)$ and $delete(T_x)$ are compositions of the first three basic edit operations. Changes between two XML documents are specified as a sequence of edit operations referred to as edit script or an XML delta. The following definitions and theories are essential to understand the change detection algorithm proposed in Wang et al. (2003).

Definition 1.
Given an edit script, E that transforms tree T_1 to T_2, E is the minimum-cost edit script for $(T_1 \mapsto T_2)$ iff $\forall$ E' of $(T_1 \mapsto T_2)$, $Cost(E') \geq Cost(E)$. $Cost(E) = n$ where n is the number of basic edit operations in E.

Definition 2.
Editing distance between T_1 and T_2, $Dist(T_1, T_2) = Cost(E)$ where E is a minimum-cost edit script for $(T_1 \mapsto T_2)$

Definition 3.
The signature of a node x in a tree T, is given by $signature(x) = /Name(x_1)/.../Name(x_n)/Name(x)/Type(x)$, where x_1 is the root of T and $(x_1, \ldots, x_n, x)$ is the path from root to x. The signature is obtained by concatenating all the names and the type. Only nodes of the same signature are matched during the change detection process.

Definition 4.
A pair of nodes (x, y) is matching, M from T_1 to T_2 iff,

1. *$(x, y) \in M$, $x \in T_1$, $y \in T_2$, $signature(x) = signature(y)$*
2. *$\forall (x_1, y_1) \in M$, and $(x_2, y_2) \in M$, $x_1 = x_2$ iff $y_1 = y_2$.*
3. *M is prefix closed, i.e given $(x, y) \in M$, suppose x' is the parent of x and y' is the parent of y, then $(x', y') \in M$.*

The matching, M which generates a minimum-cost edit script is referred to as the *minimum-cost matching*, M_{min} from T_1 to T_2.

Lemma 1.
Suppose both x and y are leaf nodes, $x \in T_1$, $y \in T_2$; θ denotes null.

1. $Dist(x, y) = 0$ iff $Signature(x) = Signature(y), Value(x) = Value(y)$ *(identical);*

2. $Dist(x, y) = 1$ iff $Signature(x) = Signature(y), Value(x) \neq Value(y)$ *(update);*

3. $Dist(x, \theta) = Dist(\theta, y) = 1$ *(delete & insert).*

Given two XML documents doc_1 and doc_2, let T_1 and T_2 be their tree representations. The algorithm consists of three main steps.

1. *Parsing and hashing*: doc_1 and doc_2 are parsed into xtrees T_1 and T_2. The hash values of every node in the trees are then calculated.

2. *Matching*: The hash values of the trees are compared to find equivalences and differences between the trees. This is done to find the minimum-cost matching between T_1 and T_2.

3. *Generating minimum cost edit scripts*: based on the minimum-cost matching, a minimum cost edit script E for $(T_1 \mapsto T_2)$ is generated.

These steps are illustrated in algorithms 4, 5 and 6.

Algorithm 4 X-Diff Algorithm Wang et al. (2003)

```
Input: $(DOC_1,\ DOC_2)$
# Parsing and Hashing
Parse $DOC_1$ to $T_1$ and hash $T_1$;
Parse $DOC_2$ to $T_2$ and hash $T_2$;
# Checking and Filtering
if $(XHash\ (Root(T_1))\ =\ XHash\ (Root(T_2)))$ then
  $DOC_1$ and $DOC_2$ are equivalent, stop.
else
  Do Matchng:
  Find a minimum-cost matching $M_{min}(T_1,\ T_2)$ from $T_1$ to $T_2$.
end if
# Generating minimum-cost edit script
Do EditScript:
Generate the minimum-cost edit script E from $M_{min}(T_1,\ T_2)$.
```

Algorithm 5 X-Diff Matching Algorithm Wang et al. (2003)

```
Input: $Tree\ T_1\ and\ T_2$
Output: a minimum-cost matching $M_{min}(T_1,\ T_2)$
Initialize: set initial working sets
$N_1\ =\ all\ leaf\ nodes\ in\ T_1,\ N_2\ =\ all\ leaf\ nodes\ in\ T_2$
Set the Distance Table $DT\ =\ \{\}$
# Step 1: Reduce matching space
Filter out next-level subtrees that have equal XHash values
# Step 2: compute editing distance for $(T_1\ \mapsto T_2)$
DO {
for all nodes x in $N_1$ do
  for all nodes y in $N_2$ do
    if Signature(x) = Signature(y) then
      Compute $Dist(x,\ y)$;
      Save matching $(x,\ y)$ with $Dist(x,\ y)$ in $DT$
    end if
  end for
end for
Set:
$N_1\ =\ parent\ nodes\ of\ previous\ nodes\ in\ N_1$;
$N_2\ =\ parent\ nodes\ of\ previous\ nodes\ in\ N_2$
} WHILE($both\ N_1\ and\ N_2\ are\ not\ empty$)
# Step 3: mark matchings on $T_1$ and $T_2$
Set $M_{min}(T_1,\ T_2)\ =\ \{\}$
if $Signature(Root(T_1))\ \neq Signature(Root(T_2))$ then
  Return; # $M_{min}(T_1,\ T_2)\ =\ \{\}$
else
  Add $(Root(T_1),\ Root(T_2))\ to\ M_{min}(T_1,\ T_2)$
  for all non-leaf nodes mapping $(x,\ y)\ \in Mmin(T_1,\ T_2)$ do
    Retrieve matchings between their child nodes that
    are stored in DT.
    Add child node matchings into $M_{min}(T_1,\ T_2)$
  end for
end if
```

Algorithm 6 X-Diff EDIT-SCRIPT Algorithm Wang et al. (2003)

```
Input: Tree $T_1$ and $T_2$, a minimum-cost matching $M_{min}(T_1, T_2)$, the distance table DT.
Output: an edit script E
Initialize: set $E = Null$;
$x = Root(T_1), y = Root(T_2)$
if $(x, y) \notin M_{min}(T_1, T_2)$ then
  Return "$Delete(T_1), Insert(T_2)$"
else if $Dist(T1, T_2) = 0$ then
  Return "";
else
  for all node pairs $(x_i, y_j) \in M_{min}(T_1, T_2)$, $x_i$ is a child node of x, $y_j$ is a child node
  of y do
    if $x_i$ and $y_j$ are leaf nodes then
      if $Dist(x_i, y_j) = 0$ then
        RETURN "";# Subtree deletion and insertion
      else
        Add $Update(x_i, Value(y_j))$ to E; # Update leaf node
      end if
    else
      Add $EditScript(Tx_i, Ty_j)$ to E;# Call subtree matching
      RETURN E;
    end if
  end for
  for all nodes $x_i$ not in $M_{min}(T_1, T_2)$ do
    Add "$Delete(Tx_i)$" to E;
  end for
  for all nodes $y_j$ not in $M_{min}(T_1, T_2)$ do
    Add "$Insert(Ty_j)$" to E;
  end for
end if
return E;
```

The representation of the nodes in the edit script should uniquely and unambiguously identify the specified nodes. Section 4.4 describes how this change can be unambiguously represented. The change detection approach presented above is fast and efficient for small XML files but inefficient in handling large XML files since:

1. the entire trees of both XML documents have to reside in memory during the detection process and

2. an XML DOM tree is twice as large as the XML document Sundaram & Madria (2012).

The detection process is therefore constrained by the system's available main memory. For a more scalable change detection, XML documents should be stored in a database and queried. This approach would be more scalable as large documents can be stored in databases and efficiently queried and manipulated.

4.3 RDF change detection

Changes between two RDF knowledge bases are represented as a set of triple additions and deletions. These set of changes are referred to as RDF delta. An update to a part of a triple without explicit deletion of the triple will consist of deleting the triple and adding a new updated triple. In the same vein, a triple replacement will consist of a deletion and an addition of a new triple. An RDF delta will contain:

1. *del(x, y , z)* - delete a triple

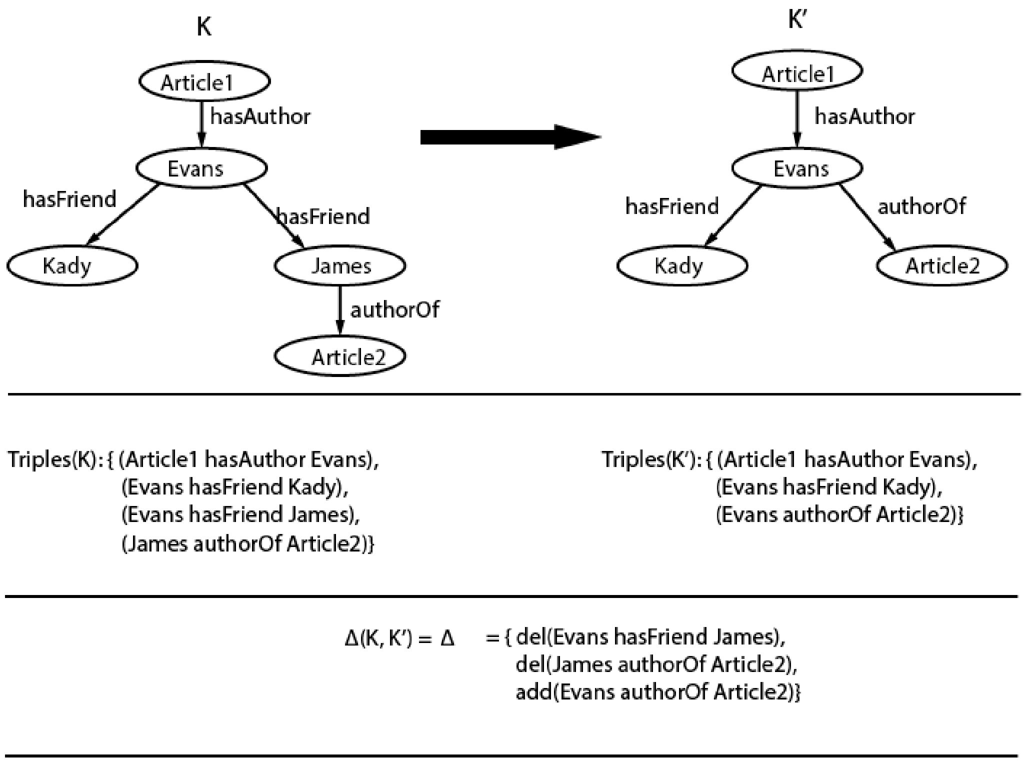
2. *add(u, v, w)* - add a triple.

Note that we do not consider RDF Schema since it introduces a certain type of ”non-determinism” as data changes and the schema evolves Chirkova & Fletcher (2009).

RDF graphs may contain blank nodes, which makes it more difficult to compare two graphs since the scope of blank nodes is restricted to the local graph in which they belong. We will first discuss computing delta between graphs without blank nodes. Computing the explicit difference between two RDF graphs with no blank nodes is simple and straight forward. As defined by Berners-Lee & Connolly (2004):

Definition 5.
If G_1 and G_2 are ground RDF graphs, then the ground graph delta of G_1 and G_2 is a pair (insertions, deletions) where insertions is the set difference $G_2 - G_1$ and deletions is $G_1 - G_2$.

Since we have to translate a set of additions and deletions of triples (RDF delta) to a set of edit operations (XML delta), we will consider only explicit RDF deltas. Figure 4.1 shows two RDF graphs and their delta.



Triples(K): { (Article1 hasAuthor Evans),
(Evans hasFriend Kady),
(Evans hasFriend James),
(James authorOf Article2)}

Triples(K′): { (Article1 hasAuthor Evans),
(Evans hasFriend Kady),
(Evans authorOf Article2)}

Δ(K, K′) = Δ = { del(Evans hasFriend James),
del(James authorOf Article2),
add(Evans authorOf Article2)}

Figure 4.1: Delta between K and K’

The RDF delta is defined as

$$\triangle = (K, K^{'}) = (\{add(t)|t \in K^{'} - K\} \bigcup \{del(t)|t \in K - K^{'}\})$$

$\triangle$ is reversible and composable according to the definitions below Zeginis et al. (2011)

Definition 6 (Reversibility).
The inverse of an atomic change operation is defined as: $Inv(Add(t)) = Del(t)$ *and* $Inv(Del(t)) = Add(t)$ *A set of change operations* D *can be reversed as follows:* $Inv(D) = U\ \{Inv(d)|d \in D\}$
$\triangle$ *is reversible if:* $Inv(\triangle(K, K^{'})) = \triangle(K^{'}, K)$

Definition 7 (Composition).
A differential function $\triangle$ *is composable if:* $\triangle(K_1, K_2) \circ \cdots \circ \triangle(K_{n-1}, K_n) = \triangle(K_1, K_n)$ *where* $\triangle_1 \circ \triangle_2 = \triangle_1 \bigcup \triangle_2$.

For graphs containing blank nodes, the blank nodes need to be renamed in such a way that the graphs will be comparable. In most practical RDF graphs, we notice that each blank node has one incoming edge and many outgoing edges. So a blank node is used as an object of a triple only once. RDF graphs with this property is known to have a nested form D.-H. Lee et al. (n.d.). We propose a renaming scheme for blank nodes. The following defines the renaming scheme.

Definition 8 (Blank node renaming)**.**
Suppose x is a blank node and the object of the triple s_1, p_1, x, then the new name of x, $name(x) = s_1.p_1$ if s_1 is a URI or a renamed blank node. If s_1 is a blank node and not renamed, then s_1 should be renamed first before x. The new name is transformed into a URI using a chosen default URI.

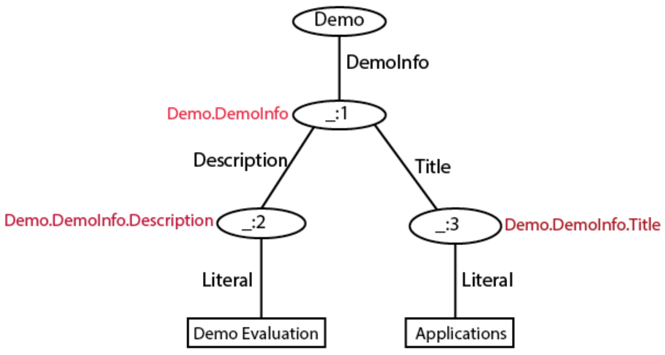


Figure 4.2: Blank node renaming

Figure 4.2 illustrates the renaming scheme. After applying the renaming scheme, the RDF graphs become ground graphs(graphs with no blank nodes) and definition 5 can be applied to compare and find the difference between the two graphs. To prevent naive comparison and make the delta calculation more efficient, we adopt predicate-grouping and triple partition as proposed by D.-H. Lee et al. (n.d.). In this approach, triples are grouped according predicates and comparison is done between groups of the same predicate. As a result, this reduces the amount of comparisons. Figure 4.3 illustrates predicate grouping.

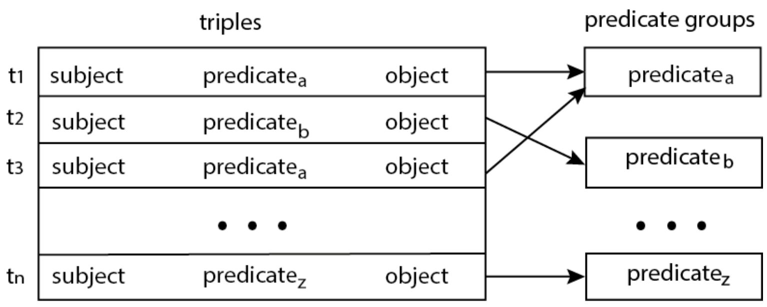


Figure 4.3: Predicate grouping

The steps to find an RDF delta are presented in algorithm 7.

Algorithm 7 RDF delta algorithm

```
Input: Graphs $G_1$ $and$ $G_2$
Apply blank node renaming to $G_1$ $and$ $G_2$
Apply predicate grouping and partition $G_1$ $and$ $G_2$
# Find $G_2 - G_1$ $and$ $G_1 - G_2$
for same predicate groups in $G_1$ $and$ $G_2$ do
  # Compare triples
  if $s\ p\ x \in G_2$ $and$ $s\ p\ x \notin G_1$ then
    add "$ADD(s\ p\ x)$" to delta
  else if $s\ p\ x \in G_1$ $and$ $s\ p\ x \notin G_2$ then
    add "$DEL(s\ p\ x)$" to delta.
  end if
end for
```

The delta can be represented using the vocabulary defined in the next section.

4.4 Delta representation and Update execution

To be able to propagate changes in an XML document to its corresponding RDF graph, the basic edit operations (XML delta) should be uniquely and unambiguously represented as an RDF delta. In the reverse direction, RDF delta should also be uniquely represented as an XML delta. Figure 4.4 illustrates the idea of finding the difference two XML or RDF versions of a document, propagating only the delta which is then used to update the synchronized target.

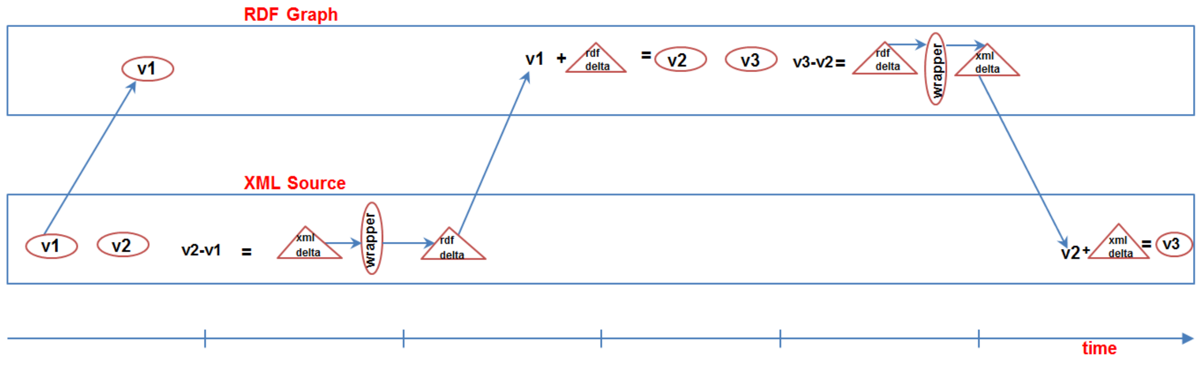


Figure 4.4: Change detection and update

We propose a simple vocabulary that captures all possibles changes in an XML tree and an RDF graph. The format for representing deltas using the vocabulary may be XML or RDF. Recall that, in this work, we confine change detection and update between XML documents and RDF graphs. The format for delta representation is relevant as deltas may not only be used for updates but may be stored to monitor data evolution and also for version control. Before discussing the delta vocabulary, we reiterate the notion of XML path expression.

Definition 9 (XML path expression).
Path expressions are used to find nodes in trees. In this work, we define the path of a node as the concatenation of all nodes names in the path from the root to the node. Given a node x *in a tree* T,
$path(x) = /Name(x_1)/Name(x_2)/.../Name(x_n)/Name(x)$ *where* x_1 *is the root of* T *and* $(x_1, ..., x_n, x)$ *is the path to* x.

In our generic transformation of XML to RDF, node names are represented as predicates in RDF triples. We therefore define path of a triple as *the concatenation of all predicates from the root of the RDF graph to the subject of the triple.* The path value is a literal value. The following vocabulary may be used to represent all deltas:

1. deltaType - identifies the type of delta representation. It has two literal values: "XML"and "RDF".

2. add - denotes a simple insert operation at leaf level (insert node name and value).

3. delete - denotes a simple remove operation at leaf level (remove node name and value),

4. path - gives the path expression of a node or a triple.

5. position - gives the location of an element. It gives the position of an element to be removed or the location to insert an element.

6. id - identifies an XML document or an RDF graph to which a delta belongs.

To get a clear understanding of how deltas are represented with this vocabulary, we will illustrate how the basic XML edit operations are represented. Figure 4.5 shows an XML tree and its corresponding RDF graph.

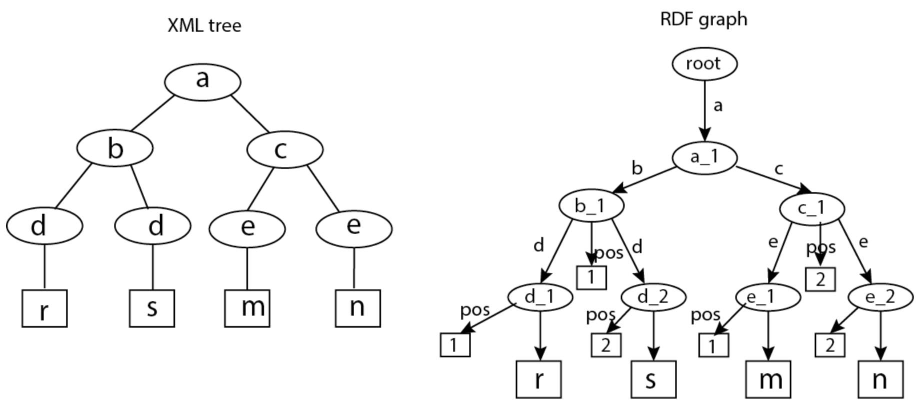


Figure 4.5: XML tree and RDF graph structure

A node insertion at the leaf level (insert() operation) is illustrated in Figure 4.6. The corresponding delta in RDF is shown in Figure 4.7.

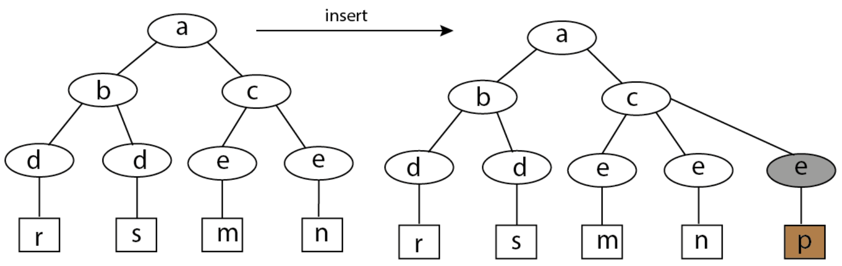


Figure 4.6: Node insertion

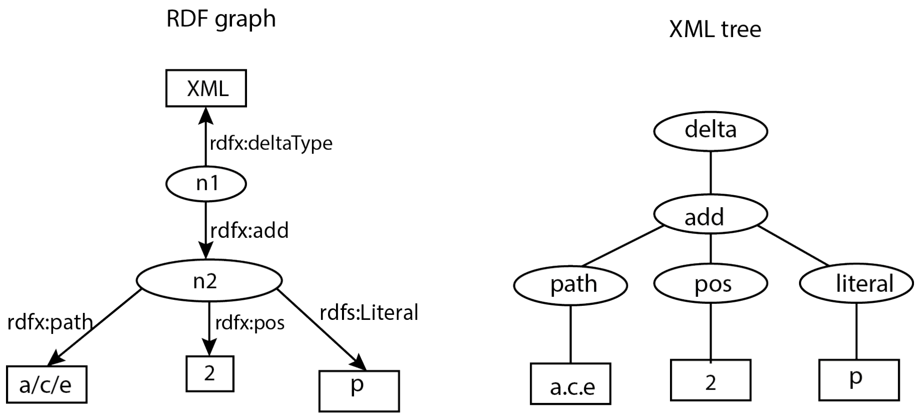


Figure 4.7: Delta for node insertion

A simple SPARQL query of the delta for the path, literal and position values is shown in Listing 4.1.

```
select ?path ?value ?pos
  where { ?s rdfx:add ?o .
      ?o rdfx:path ?path .
    ?o rdfs:Literal ?value .
    OPTIONAL { ?o rdfx:pos ?pos }
  }
```

Listing 4.1: SPARQL to get path, value and position

The last name in the path value gives the name of the node to be inserted and the literal value gives the node's value. The rest of the path literal value and the position values are used to locate where to insert the set of triples. To update the corresponding RDF graph, the query in Listing 4.2 finds the point of insertion (the subject of the new triples to be added).

```
select ?sub
  where { ?m :a ?n .
          ?n :c ?sub .
          ?sub :pos pos .
         }
```

Listing 4.2: SPARQL to get subject of triples to insert

Figure 4.8 shows a node deletion at the leaf level.

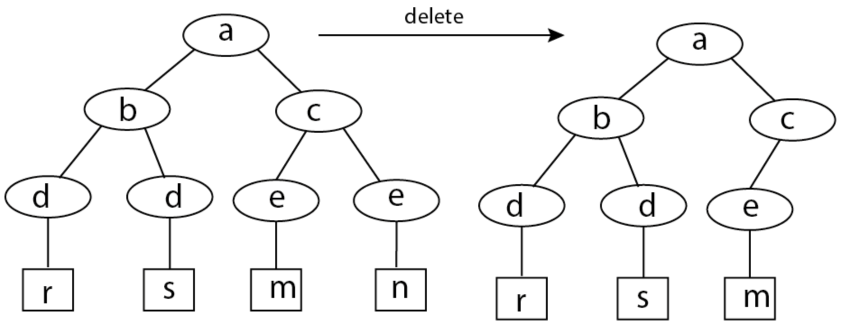


Figure 4.8: Node deletion

The corresponding delta representations are shown in Figure 4.9.

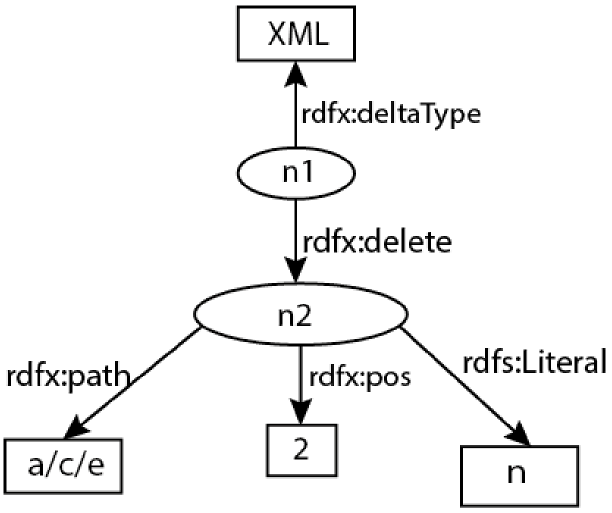


Figure 4.9: Delta for node deletion

The path value together with the literal and the position values uniquely identify the node to be deleted. The query in Listing 4.3 updates the corresponding RDF graph.

```
delete  { ?a :e ?b .
      ?b ?c ?d
    } where { ?m :a ?n .
              ?n :c ?a .
              ?a :e ?b .
              ?b :Literal value .
              ?b :pos pos .
              ?b ?c ?d
             }
```

Listing 4.3: SPARQL to delete triples

An update is treated as a sequence of deletes and insertions. Figure 4.10 shows an updated node at leaf level.

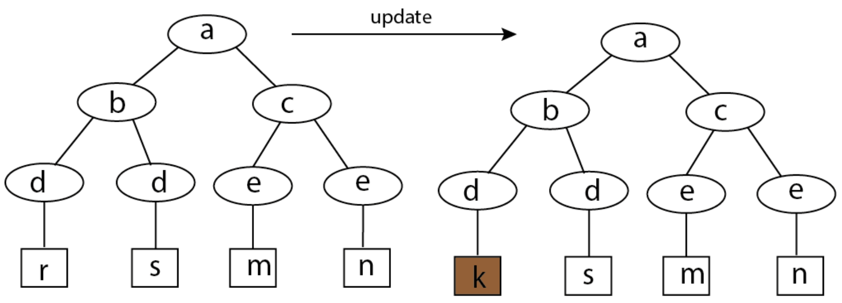


Figure 4.10: node value update

The corresponding delta is shown in Figure 4.11.

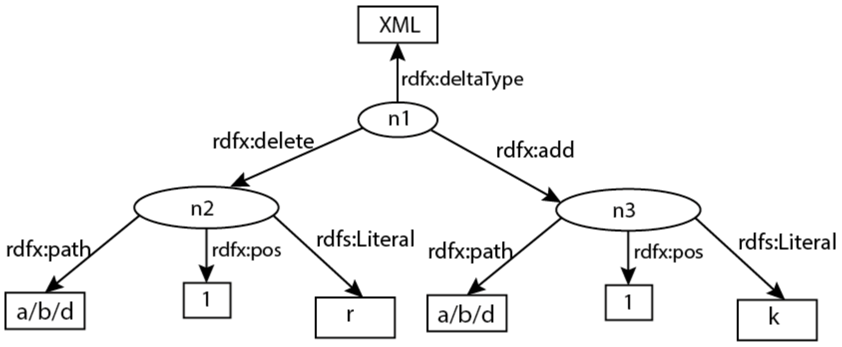


Figure 4.11: Delta for update

An update of the corresponding RDF graph will first execute delete operations followed by add operations. Insertion and deletion of subtrees are compositions of the three basic edits above. To be able to properly update triples, generated nodes and blank nodes in an RDF graph should be uniquely identifiable and traceable to specific elements in the source XML during their transformation to RDF.

To transform an RDF delta to a corresponding XML edit operation, we apply this rule: *A set of additions and deletions of triples in an RDF delta will be propagated as an XML delta if and only if they form a set of valid XML edit operations.* Triples in the RDF delta will be transformed to XML trees and if a tree corresponds to a valid XML update, that update will be propagated. In the generic transformation, it is quite straight forward to transform triples to XML by treating predicates as element names, literals as text or attribute values. Figure 4.12 shows an RDF graph, a valid and an invalid RDF deltas.

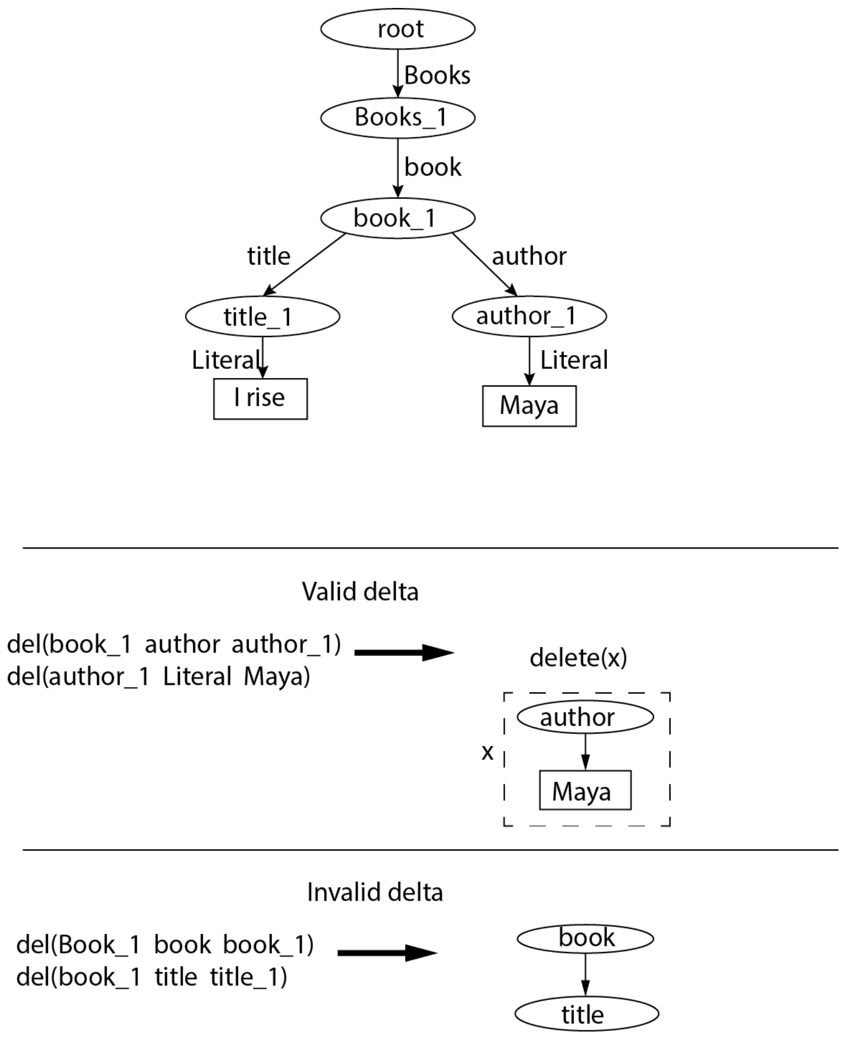


Figure 4.12: Delta transformation

Assuming these triples are deleted:
($book\_1$ $title$ $title\_1$)
($title\_1$ $Literal$ $"Irise"$)
and these triples added.
($book\_1$ $distributor$ $distributor\_1$)
($distributor\_1 Literal$ $"Semaku"$)
The first set of triples form a valid XML delete operation and the second forms a valid XML insert operation. The representation of the delta is shown in Figure 4.13.

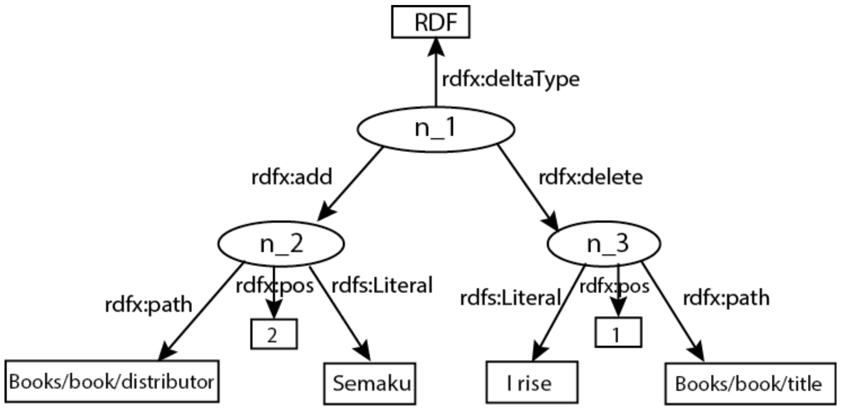


Figure 4.13: Example RDF delta

To update a corresponding XML tree, we query for the path, literal and position values. The XQUERY query in Listing 4.4 performs an insert update.

```
insert node <distributor>Semaku</distributor> into doc("doc.xml")//Books/book[
    fn:position() = pos]
```

Listing 4.4: XQUERY to insert node

Listing 4.5 shows a delete update.
The position value identifies the element in which to insert the new node and not the exact position to insert the node

```
delete node doc("doc.xml")//Books/book/title[. = 'Semaku' and fn:position() =
    pos]
```

Listing 4.5: XQUERY to delete node

The delta transformation and representation is sound and complete since every XML edit operation has a corresponding RDF delta. The corresponding RDF delta is unique. Also, every valid RDF delta also transforms uniquely and unambiguously to corresponding XML edit operations.

4.5 Summary

We have presented an XML detection algorithm, X-Diff and proposed an RDF change detection algorithm. We have proposed a blank node renaming scheme to make our RDF change detection easier and more efficient. Furthermore, we have proposed a vocabulary to represent all valid deltas in XML and RDF. We have also discussed a novel delta transformation and representation from XML to RDF and vice versa.

Chapter 5

Software Development and Implementation

5.1 Introduction

This section is divided into two subsections. The first subsection describes the design and implementation of the data transformer. The second subsection describes the implementation of the update engine. We discuss assumptions, properties and requirements for the realization of the data transformer and the update engine.

5.2 Data transformer

5.2.1 Purpose

The purpose of this tool is to transform data from XML, JSON, and CSV formats to RDF. It provides a two-stage transformation process: A generic transformation and a semantic transformation, which provides a means to add more meaning and structure to a source RDF. The resulting RDF can be serialized into a different format(N-triples, Turtle, etc) and then loaded into a remote triple store. The platform provides an automated means to transform a source file in XML, JSON, CSV to RDF without any human intervention. To further transform a set of triples to a desired form, the platform provides a semi-automated semantic transformation through a list of operations to perform on the source RDF.

5.2.2 Definitions

5.2.2.1 Pluggable application framework

Application plugins are software components that provide a means to extend an application's functionality. A software application is pluggable if it supports addition of plugins.

5.2.2.2 Reflection

Reflection is a property of a software application to modify its structure and behavior at runtime. Using reflection, the software application can inspect classes within packages, load, instantiate and invoke methods of these classes at runtime. Reflection is mostly possible with high-level virtual machine programming languages such as Java, PHP, etc.

5.2.3 System overview

The platform consists of several components. Figure 5.1 shows the major components. The generic transformer handles the transformation of the XML, JSON or CSV to RDF. The dynamic plugin loader reads and loads the plugins at runtime. The semantic transformer reads mapping rules and executes them on triples in the semantic transformation phase. Triples are stored in the local triple store. The local triple store also supports SPARQL queries. The transport component sends the triples to a remote triple store. Figure 5.1 demonstrates the system overview.

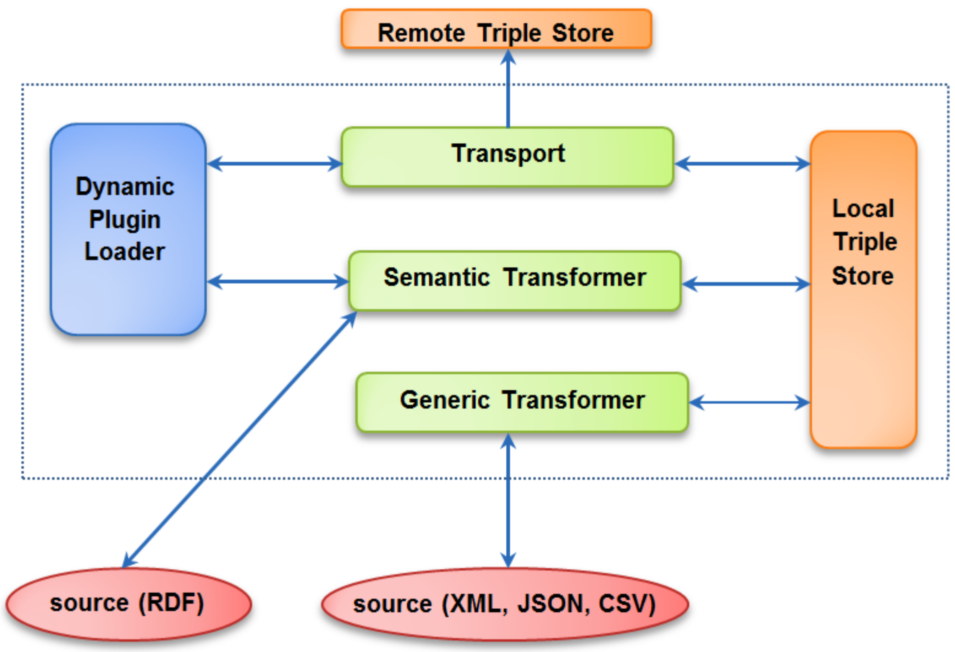


Figure 5.1: System overview

5.2.4 Specific requirements

5.2.4.1 Logical database requirements

5.2.4.1.1 XML database

For easy processing and querying of XML documents, the tool employs an XML database. An XML database allows the storage of XML documents and query them using the powerful XQUERY language. Sedna XML database is deployed in this tool. Sedna is an open source native XML database, which supports the full XQUERY language as specified by W3C. It provides easy backup to files and provides APIs for all the major programming languages.

5.2.4.1.2 RDF triple store

A local RDF triple store or simply triplestore is required to store and query the generated triples. Triplestore is a database for triples. OpenRDF Sesame triplestore is deployed in this tool. OpenRDF Sesame provides RDF database (triplestore) with in-memory and file storage. It offers an easy to use JAVA API that provides means to connect the database and query using W3C's SPARQL.

5.2.5 Other requirements

The set of plugins and functions for semantic transformations and transport are described in an XML file. An XML parser is required to parse and read the file. This tool uses a Xerces XML parser.

A JSON parser is required to parse a source JSON document in order to transform it to RDF. Also the mapping rules for semantic transformations are specified as JSON. This tool uses Java API for JSON processing.

5.2.6 System Structure

The class diagram in Figure 5.2 gives the structural model of the system. The diagram only shows the main classes without other helping classes. As their names suggest, the XML-DatabaseManager instantiates a Sedna XML database, loads XML documents and returns a reference to the XMLTripleGenerator class. The XMLTripleGenerator class is responsible for transforming XML to RDF and loading in the local RDF triple store. The JSONTriple-Generator class parses JSON documents and transforms them to RDF before loading them in the local triple store. The CSVTripleGenerator class likewise transform CSV files. The PluginPropertiesLoader class reads the plugins specified in the plugin properties file. The MappingRulesLoader class reads the mapping rules applied during the semantic transformations. The dynamic loading of plugins and calling of functions on triples are done by the MainEngine Class.

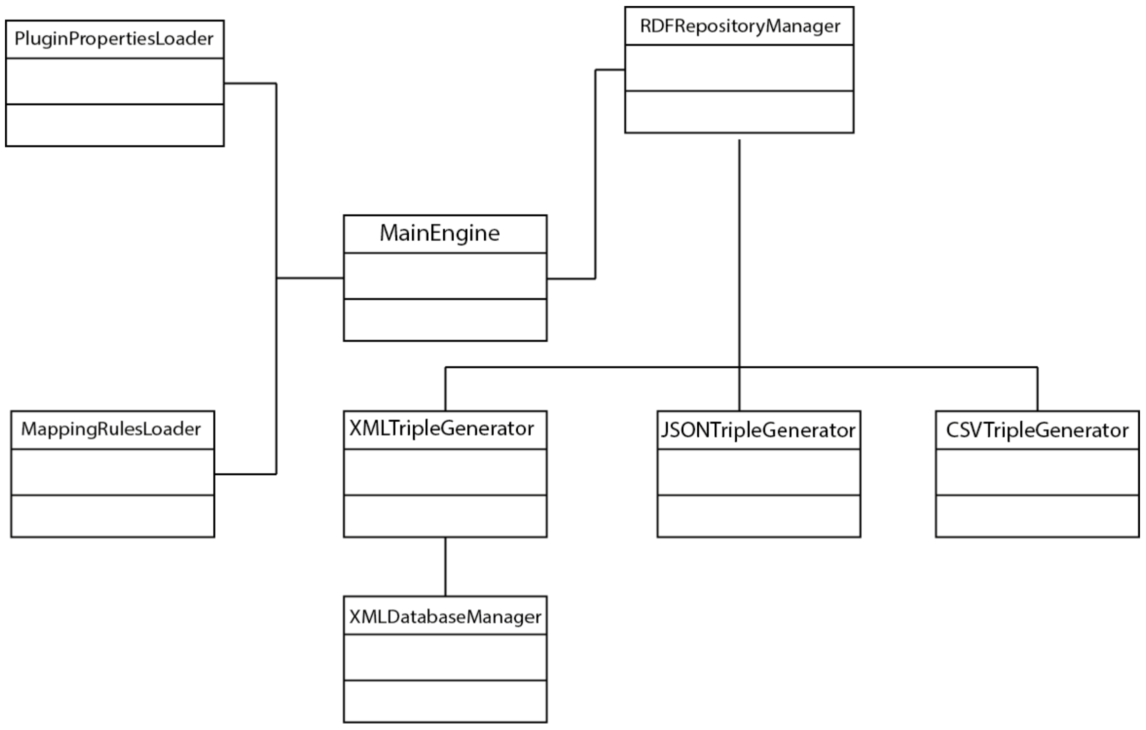


Figure 5.2: System structure

5.2.7 Software System Attributes

5.2.7.1 Portability

The software prototype is written in Java, which is a cross-platform programming language. The application can therefore be deployed on all platforms where the Java Virtual Machine can run. Porting from source to other programming languages is possible if the language supports reflection i.e loading and instantiation of classes and invoking methods of these classes at runtime.

5.2.8 Implementation

The software prototype was implemented following the Agile Software development process. The agile software development process is based on "inspect and adapt" where developers gather requirements at the same time software is in development. Regular meeting between the developer and Semaku B.V (customer) was conducted to explain algorithms, features, development environments. Feedback and desired changes from the customer were then incorporated into the software prototype. The software prototype is implemented in Java programming language.
The platform depends on these libraries.

1. OpenRDF sesame Java API: OpenRDF Sesame provides RDF database (triplestore) with in-memory and file storage. It offers an easy to use JAVA API that provides means to connect the database and query using W3C's SPARQL.

2. sedna XQJ Java API: Sedna provides a free native XML database with persistent storage, ACID transactions and backup. It comes with a JAVA API that allows to connect, add and delete documents from the database. It supports W3C's XQUERY for querying XML documents.

3. Xerces XML parser: Provides a library for parsing, generating, manipulating and validating XML documents using DOM and SAX APIs.

4. JAVA API for JSON processing: The standard JAVA API for JSON processing provides a streaming API and an Object Model API for parsing and manipulating JSON documents. The streaming generates events as an XML document is parsed whiles Object Model API creates a tree-like structure of the JSON data in memory.

5.3 Semantic transformation API

The data transformer consists of a generic transformer and a semantic transformer. The generic transformer uses a fixed set of rules to transform source data to RDF triples. These rules cannot evolve without explicit changes at the source code level. The semantic transformer on the other hand is a list of functions defined if and when needed. These functions are public class functions which are exported as Java jar files and added to the application. A function may accept a connection to a repository and two string lists. These parameters are optional and could be replaced by null values. The lists are to provide a means to pass data to the functions if necessary. Listing 5.1 show a sample class with one public function for semantic transformation.

```java
public class ReplacePredicate {

  public void replace_predicate(Repository rep, List<String> left, List<String>
      right)
  {
    . . .
  }

}
```

Listing 5.1: Java function for semantic transformation

The Java class is exported as a jar file and added to the application. The complete class name and function names are added to the application's configuration file as shown in Listing 5.2.

```xml
<?xml version="1.0" ?>
<plugins>
  <plugin>
    <class-name>plugins.ReplacePredicate</class-name>
    <functions>replace_predicate</functions>
  </plugin>
</plugins>
```

Listing 5.2: Plugins configuration file

To use this function in the semantic transformation, add the function name to the list of functions defined in the semantic rules. Note that the platform only provides a means to add functions to extend the semantic transformation and not what the function actually does. Listing 5.3 shows a sample list of semantic functions.

```
{
  "add_namespace":{
    "left":["evans"],
    "right":["http://wiout.com/"]
    },
  "export_ntriples":{
    "left":["ntriples"],
    "right":[]
  }
}
```

Listing 5.3: Sample list of semantic functions

5.4 Change Update Engine

5.4.1 Purpose

The change detection framework consists of three main components; an XML change detector, an RDF change detector and an update engine. The XML change detector generates delta between two versions of an XML document, which is then passed to the update engine to update a corresponding RDF graph in the repository. On the other hand, the RDF change detector generates RDF delta which is used to update a corresponding XML document. Figure 5.3 gives an illustration of the framework.

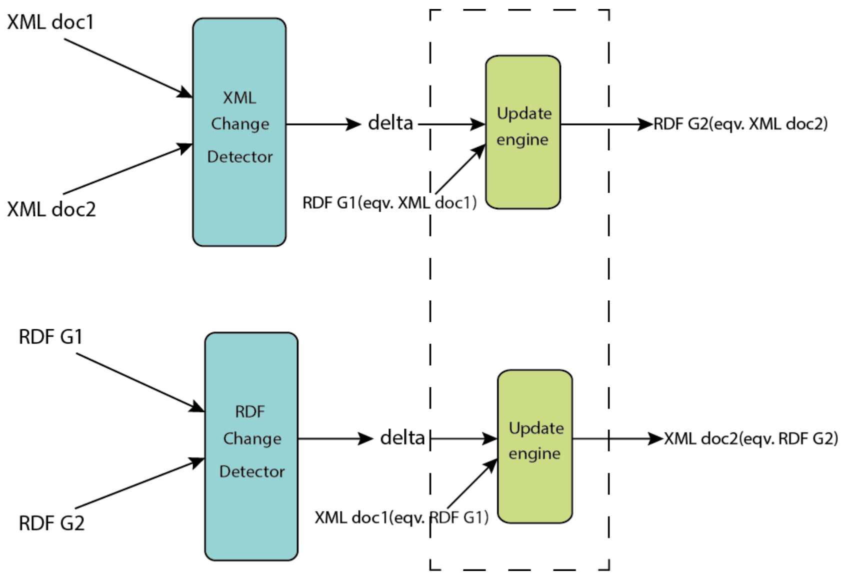


Figure 5.3: Change detection platform

In this section we describe a prototype implementation of the update engine. Implementation of the change detectors are out of scope for this project.

5.4.2 Assumptions

We assume both XML and RDF deltas are in RDF format.

5.4.3 System Structure

Figure 5.4 illustrates the structural model of the update engine. The RDFQueryDelta class queries the RDF delta, which is managed by DeltaStore class. The UpdateXML and UpdateRDF classes build queries with the results returned by the RDFQueryDelta class. The XMLDatabaseManager and RDFRepoManager classes manage connections to the XML and RDF databases respectively.

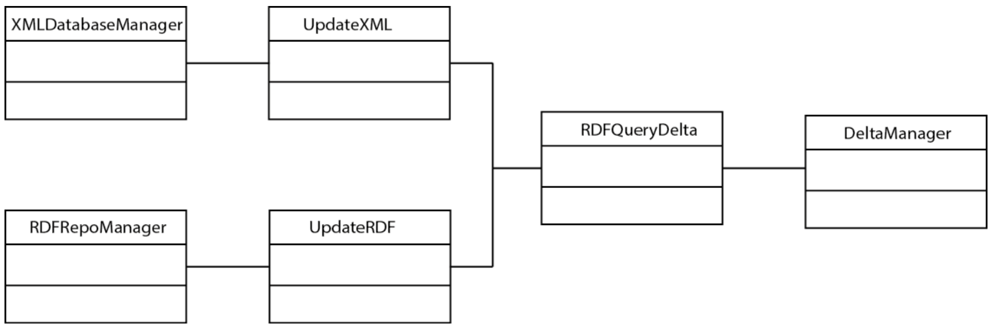


Figure 5.4: Structural model of the Update Engine

5.4.4 Implementation

The update engine is implemented within a similar platform as the data transformer. It is implemented in Java programming language using Agile Software development process.

5.5 Summary

We have discussed the implementation of generic XML, JSON and CSV to RDF transformer. We have also presented an implementation of an Update Engine.

Chapter 6

Performance Testing

In this section we describe and show test results of both the generic transformer and the update engine. The generic transformation process was required to be completely automated and round-trippable. Execution times of the data transformation process is dependent on data size and structure. No strict time requirement was given. In Section 6.1 however, we measure execution times for some loads to give an estimate of the execution times.

6.1 Testing of generic transformer

Tests with different file sizes were performed to show that the implementation works correctly and also to give estimates of how long it takes to transform different file sizes. Only generic transformations from (XML, JSON and CSV) to RDF were tested and measured. Few semantic transformations were tested only to verify correctness. Execution times were not measured since semantic transformations are document specific and based on the requirements of an organization and we cannot foresee which semantic transformations will be performed on which data. Testing was carried out on an HP EliteBook 8560w (Intel Core i7-2670QM) by increasing load up to approximately 2MB. Typical document sizes used in this project range from 0 to 1MB. The XML data was generated using XMark benchmark suit, a benchmark for XML data management CWI (2003). It provides a scalable XML database modeling an internet auction website. XML documents were generated with the XMark command line tool. Different sizes of documents were obtained by varying the -f *factor*. The graphs show a plot of execution times against data sizes.

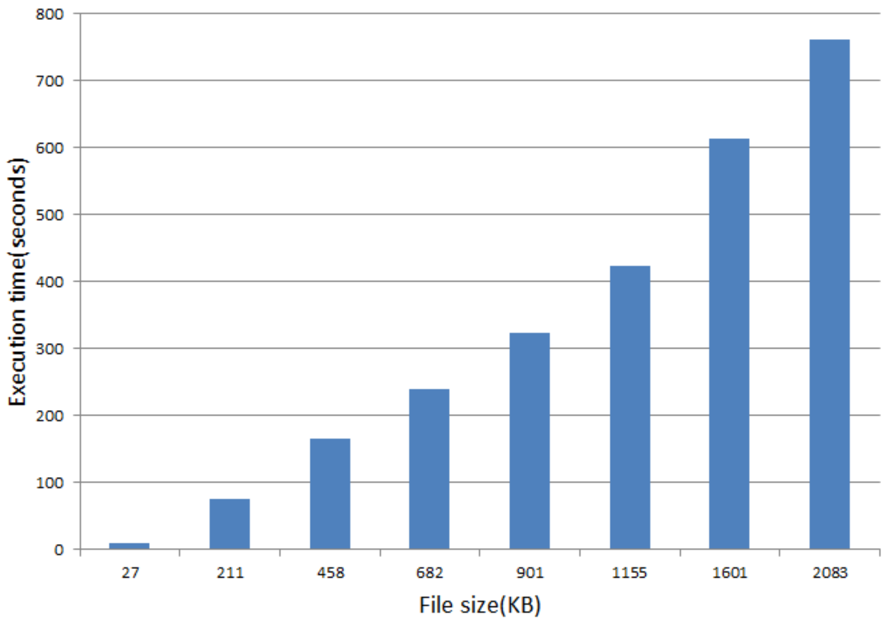


Figure 6.1: XML to RDF transformation

JSON data was generated using the online tool at *json-generator.com*. Different sizes of the data was obtained by varying the repeat parameter, which corresponds to how many times the keys and value pairs are repeated.

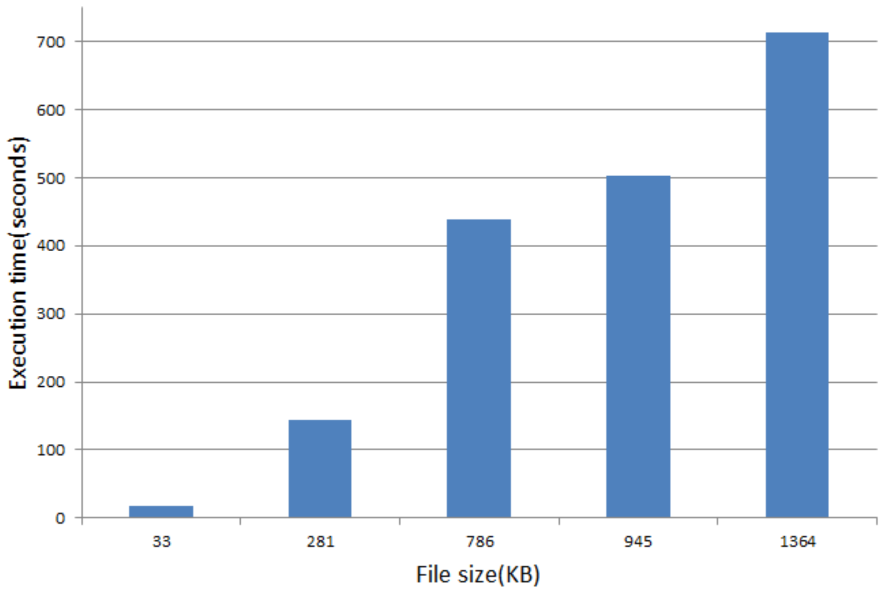


Figure 6.2: JSON to RDF transformation

CSV data was generated using the online tool at *mackaroo.com*.

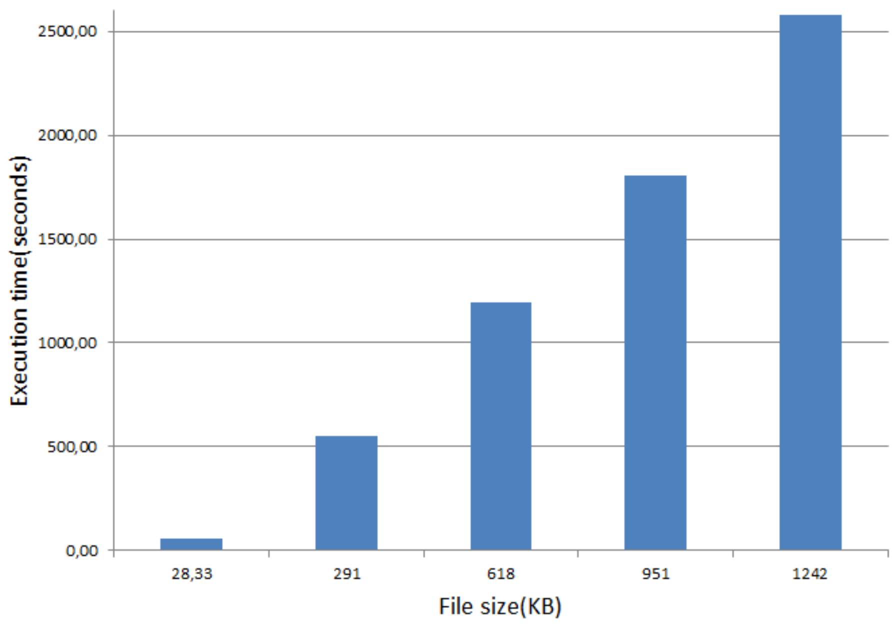


Figure 6.3: CSV to RDF transformation

6.2 Testing update Engine

Testing was done to ensure correctness of the update process. An XML document was edited (few nodes were deleted and new nodes added) and the corresponding delta manually generated since change detectors were not implemented in this project. The original XML document was transformed to RDF using the generic transformer. With the generated delta, the update engine successfully updated the RDF graph to reflect all changes. In a same way, the RDF graph was edited and its delta was generated manually. The update engine successfully updated the corresponding XML document with the RDF delta.

6.3 Summary

We have presented tests and results of execution times of our generic transformer for certain sizes of data. The results of the tests are used to give estimates of how long it may take to transform a document from XML, JSON or CSV to RDF. Successful test results also indicates correctness of our implementations.

Chapter 7

Conclusions

In this section, we discuss the contributions and limitations of this work. We also discuss desirable features that could be implemented in the future.

7.1 Contributions

We have presented a data transformation process that incorporates generic and semantic transformations of XML, JSON and CSV to RDF. The transformation process incorporates two novel things:

- the property of the generic transformation to preserve structure of data and to allow round-tripping.
- the use of a list of functions for semantic transformations of RDF triples generated from the generic transformation process.

Also, we have presented an implementation of the platform providing a completely automated generic transformation and a semi-automated semantic transformation. The platform provides a means to add more functions to the pool of functions for semantic transformations. This was made possible with the platform's plugin architecture. The ability to add functions when needed is essential for flexibility in the transformation and to cater for unforeseen custom data modifications that may be required.

We also presented a theoretical study and proposed algorithms for change detection methods in XML and RDF. We further proposed a simple vocabulary for describing changes in XML and RDF, and based on this vocabulary, an update engine was implemented. The implementation demonstrated how updates between synchronized XML and RDF data can be represented and executed. An initial study of related work and relevant knowledge was also presented.

Finally, the implementations were tested to ensure expected results and also to give estimates of execution times for generic transformations of files of different sizes.

7.2 Limitations

Due to time constraints, XML and RDF change detection engines were not implemented. Changes in XML and RDF had to be detected manually during the test of the update engine.

We could therefore only test the performance of the update engine with a small set of data generated manually.

Since a full-fledged system with a means to propagate data between remote systems have not been built yet, the transformation and update engines were tested on one local machine. It was therefore not possible to test how the system scales in a distributed environment and with different data sizes and structures.

7.3 Future work

Implementation of RDF and XML change detection engines
We have presented algorithms for detecting changes in XML and RDF. A change detection engine is an important component of the semantic platform and therefore an implementation in the future is required.

Data propagation
Synchronizing data sources requires an effective means of data propagation. We therefore need to investigate effective means to propagate data between the repository and the data sources. The means of data propagation should be suitable for transferring large amounts of data and also small chunks of updates.

Extensive testing of system performance
In future work, the system has to be extensively tested for scalability to see the system's performance under increased number of connected sources to the repository, and under increased workloads.

References

Berners-Lee, T. & Connolly, D. (2004). *Delta: an ontology for the distribution of differences between RDF graphs.* W3C.
34

Chen, Y., Madria, S. & Bhowmick, S. (2004). DiffXML: change detection in XML data. In *Database systems for advanced applications* (pp. 289–301).
19

Chirkova, R. & Fletcher, G. H. (2009). Towards well-behaved schema evolution. In *Webdb.*
34

CWI. (2003). *Xmark - an XML benchmark project.* Retrieved from `http://www.ins.cwi.nl/projects/xmark/`
53

ECMA. (2013). *The JSON data interchange format.*
16

Izquierdo, J. L. C. & Cabot, J. (2013). Discovering implicit schemas in JSON data. In *Web engineering* (pp. 68–83). Springer.
18

John Boyer, S. M. M. M. R. S. J. S., Sandy Gao. (2011). Experiences with JSON and XML transformations. W3C.
18

Klein, M. (2002). Interpreting XML documents via an RDF schema ontology. In *Database and expert systems applications, 2002. proceedings. 13th international workshop on* (pp. 889–893).
17

Klein, M., Fensel, D., Kiryakov, A. & Ognyanov, D. (2002). Ontoview: Comparing and versioning ontologies. *Collected Posters ISWC 2002*.
19

Lee, D.-H., Im, D.-H. & Kim, H.-J. (n.d.). A change detection technique for RDF documents containing nested blank nodes. 20, 35, 36

Lee, S. K. & Kim, D. A. (2006). X-tree diff+: Efficient change detection algorithm in XML documents. In *Embedded and ubiquitous computing* (pp. 1037–1046). Springer.
19

Melnik, S. (1999). Bridging the gap between RDF and XML. *Retrieved February*, *26*, 2006.
17, 22

Noy, N. F. & Musen, M. A. (2002). Promptdiff: A fixed-point algorithm for comparing ontology versions. *AAAI/IAAI*, *2002*, 744–750.
19

Papavasileiou, V., Flouris, G., Fundulaki, I., Kotzinos, D. & Christophides, V. (2013). High-level change detection in RDF (s) kbs. *ACM Transactions on Database Systems (TODS)*, *38*(1), 1.
19

Shafranovich, Y. (2005). *Common format and mime type for comma-separated values (CSV) files.*
16

Sundaram, S. & Madria, S. K. (2012). A change detection system for unordered XML data using a relational model. *Data & Knowledge Engineering*, *72*, 257–284.
19, 33

Thuy, P. T. T., Lee, Y.-K., Lee, S. & Jeong, B.-S. (2008). Exploiting XML schema for interpreting XML documents as RDF. In *Services computing, 2008. scc'08. ieee international conference on* (Vol. 2, pp. 555–558).
17

Volkel, M., Winkler, W., Sure, Y., Kruk, S. R. & Synak, M. (2005). Semversion: A versioning system for RDF and ontologies. In *Proc. of eswc.*
19

w3c. (2003). *Extensible markup language (XML).* `http://www.w3.org/{XML}/`.
15

w3c. (2008). *Sparql query language for RDF.*
14

w3c. (2010). *Xquery 1.0: An XML query language (second edition).*
15

w3c. (2014a). *A JSON-based serialization for linked data.*
18

w3c. (2014b). *Resource description framework.* Retrieved from `http://www.w3.org/{RDF}/`
13

Wang, Y., DeWitt, D. J. & Cai, J.-Y. (2003). X-diff: An effective change detection algorithm for XML documents. In *Data engineering, 2003. proceedings. 19th international conference on* (pp. 519–530).
10, 19, 30, 31, 32, 33

Zeginis, D., Tzitzikas, Y. & Christophides, V. (2011). On computing deltas of RDF/S knowledge bases. *ACM Transactions on the Web (TWEB)*, *5*(3), 14.
19, 35