University of Crete

Computer Science Department

Semantic Web Middlewares and Versioning Services

Dimitris Andreou

Master's Thesis

Heraklion, December 2007

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Πλατφόρμες Σημασιολογικού Ιστού και Υπηρεσίες Εκδόσεων

Εργασία που υποβλήθηκε απο τον

Δημήτρη Σ. Ανδρέου

ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

Δημήτρης Ανδρέου, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Βασίλης Χριστοφίδης, Αναπληρωτής καθηγητής, Επόπτης

Ιωάννης Τζίτζικας, Επίκουρος καθηγητής, Μέλος

Γρηγόρης Αντωνίου, καθηγητής, Μέλος

Δεκτή:

Πάνος Τραχανιάς, Καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών
Ηράκλειο, Δεκέμβριος 2007

Abstract

The Semantic Web (SW) is an evolving extension of the World Wide Web in which Web content can be expressed not only in natural language but also in a format that can be read and used by software agents, thus permitting them to find, share and integrate information more easily. It comprises a variety of formally specified languages (RDF/S, OWL), associated formats (e.g. RDF/XML, N3, Turtle, N-Triples) and related technologies for their management.

Various platforms have been implemented in the recent years which apart from helping applications process Semantic data, evolve towards providing middleware services. These services are contacted remotely and provide controlled access to a repository on behalf of client applications, which are not needed to know or depend on the internal repository layout.

This thesis reviews such platforms and focuses on the design and implementation of the Semantic Web Knowledge Manager (SWKM) platform specifically. A common requirement in these platforms is that they all need some sort of main memory representation of Semantic data, in order facilitate its processing internally. These representations are also used by external in clients, to develop Semantic-aware applications. In this thesis we experimentally compared the performance characteristics of the main memory representation of each platform, by covering a multitude of possible access patterns. Additionally, we pay particular attention to the various Semantic versioning facilities, since versioning is an ubiquitous requirement for collaborative applications. In general, versioning is the process of assigning unique identifiers to states of artifacts and keeping other historical information and metadata. In this thesis, versioning services for the SWKM platform are introduced and compared to similar existing offerings.

In summary, the contribution of this work lies on the experimental evaluation and comparative analysis of various main memory representations of Semantic data, and on the design and implementation of middleware versioning services for Semantic models, offered as Web Services.

Πλατφόρμες Σημασιολογικού Ιστού και Υπηρεσίες Εκδόσεων

Δημήτρης Ανδρέου
Μεταπτυχιακή Εργασία
Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Πλατφόρμες Σημασιολογικού Ιστού και Υπηρεσίες Εκδόσεων

Ο Σημασιολογικός Ιστός είναι μια εξελισσόμενη επέκταση του Παγκοσμίου Ιστού στην οποία το περιεχόμενο μπορεί να εκφραστεί όχι μόνο με φυσική γλώσσα αλλά και με τυπικές γλώσσες που επιτρέπουν την παροχή προηγμένων υπηρεσιών αναζήτησης, διαμοιρασμού και ολοκλήρωσης πληροφορίας. Περιλαμβάνει ποικιλία τυποποιημένων γλωσσών (RDF/S, OWL) και σχετικών μορφοτύπων (RDF/XML, N3, Turtle, N-Triples), καθώς και σχετικές τεχνολογίες για τη διαχείριση τους.

Διάφορες πλατφόρμες έχουν υλοποιηθεί τα τελευταία χρόνια που εκτός της παροχής υπηρεσιών για την επεξεργασία σημασιολογικών δεδομένων, εξελίσσονται προς την κατεύθυνση της παροχής υπηρεσιών ενδιάμεσου λογισμικού. Οι υπηρεσίες αυτές προσφέρουν απομακρυσμένη και ελεγχόμενη πρόσβαση σε μία ή περισσότερες βάσεις δεδομένων εκ μέρους εφαρμογών-πελάτη, οι οποίες δεν χρειάζεται να γνωρίζουν ή να εξαρτώνται από τις λεπτομέρειες υλοποίησής τους.

Στην εργασία αυτή μελετήσαμε τέτοιες πλατφόρμες και επικεντρωθήκαμε στην σχεδίαση και την υλοποίηση της πλατφόρμας Semantic Web Knowledge Manager (SWKM). Κοινή απαίτηση από αυτές τις πλατφόρμες είναι η παροχή κάποιου είδους αναπαράστασης κύριας μνήμης για σημασιολογικά δεδομένα, η οποία είναι αφενός απαραίτητη για την εσωτερική (από το ενδιάμεσο λογισμικό) επεξεργασία αυτών των δεδομένων, και αφετέρου χρήσιμη για τους πελάτες των υπηρεσιών. Στην παρούσα εργασία συγκρίναμε τις επιδόσεις τέτοιων αναπαραστάσεων (που προσφέρονται από διάφορες πλατφόρμες) χρησιμοποιώντας ένα πλατύ εύρος πιθανών μοτίβων χρήσης. Επιπροσθέτως, δώσαμε ιδιαίτερη έμφαση σε υπηρεσίες και

εργαλεία για τη διαχείριση εκδόσεων (versions), διότι η παροχή τέτοιων υπηρεσιών είναι συνήθως απαραίτητη σε συνεργατικές εφαρμογές. Γενικά, η διαχείριση εκδόσεων συνίσταται στην ανάθεση μοναδικών ταυτοτήτων στα διάφορα τεχνουργήματα (στην προκειμένη σημασιολογικών μοντέλων) και στην διατήρηση ιστορικών πληροφοριών και άλλων μεταδεδομένων. Στην παρούσα εργασία σχεδιάσαμε και υλοποιήσαμε τέτοιες υπηρεσίες για την πλατφόρμα SWKM και τις συγκρίναμε με παρόμοιες υπάρχουσες υπηρεσίες.

Συνοψίζοντας, η συνεισφορά αυτής της εργασίας έγκειται αφενός στην συγκριτική και πειραματική αξιολόγηση διαφόρων αναπαραστάσεων κύριας μνήμης για δεδομένα σημασιολογικού ιστού, κι αφετέρου στον σχεδιασμό και στην υλοποίηση υπηρεσιών διαχείρισης εκδόσεων σημασιολογικών μοντέλων σε επίπεδο ενδιάμεσου λογισμικού με χρήση ιστουπηρεσιών (web services).

Επόπτης Καθηγητής: Βασίλης Χριστοφίδης
Αναπληρωτής Καθηγητής

Ευχαριστίες

Ο παρακάτω άδειος χώρος είναι ειδικά αφιερωμένος για να γράψετε τις ειλικρινείς σας ευχαριστίες που είχατε την τιμή να με γνωρίσετε από κοντά.

Ευχαριστήστε με κάτω από τη γραμμή. (Παρακαλώ όχι ροζ καρδούλες).

_____

Contents

Table of Contents **v**

List of Figures **x**

1 Introduction **1**

1.1 Introduction to the Semantic Web 1

1.2 The notion of a Middleware Platform for the Semantic Web 2

1.2.1 Main Memory Representation 4

1.2.2 Persistence 4

1.2.3 Access 4

1.3 Motivation for Benchmarking Main Memory Representations 5

1.4 Motivation and Application Scenarios for Versioning Services 5

1.4.1 Collaborative Development and Evolution of Ontologies 5

1.4.2 Collaborative Development and Evolution of Concepts 7

1.4.3 Archiving and Preservation 8

1.5 Contributions 9

1.5.1 Organization of the thesis 10

2 Semantic Web Middleware Platforms **11**

2.1 Introduction 11

2.1.1 RDF Data Model 11

2.1.2 Dependencies between RDF spaces 12

2.1.3 Declarative Query/Update support 13

2.1.4 Selection Criteria 14

2.1.5 RDF/S Triple vs. Object-based Views 15
2.2 Sesame 16
2.2.1 Architecture 16
2.2.2 Main Memory Model 17
2.2.2.1 Data Structures 18
2.2.2.2 Access Methods 19
2.2.3 Persistence Storage 19
2.2.4 Sesame Services 20
2.2.4.1 Query Service 20
2.2.4.2 Extract RDF Service 20
2.2.4.3 Upload Data Service 20
2.2.4.4 Clear Repository Service 21
2.2.4.5 Triple Removal Service 21
2.2.5 Summary 21
2.3 Jena 21
2.3.1 Main Memory Model 22
2.3.1.1 Data Structures 22
2.3.1.2 Access Methods 23
2.3.2 Persistence Storage 23
2.3.3 Services 24
2.4 Kowari 24
2.4.1 Architecture 24
2.4.2 Main Memory Model 25
2.4.2.1 Data Structures 26
2.4.2.2 Access Methods 27
2.4.3 Persistence Storage 27
2.4.4 Services 28
2.5 SWKM 29
2.5.1 Architecture 29
2.5.2 Main Memory Model 29

2.5.2.1 Data Structures 29
2.5.2.2 Access Methods 31

3 Benchmarking the Main Memory representations 33
3.1 Introduction 33
3.2 RDF/S Benchmark 34
3.2.1 RDFS version of the LUBM schema 34
3.2.2 Instance Generation 36
3.2.3 Query Workload 36
3.3 Experimental Evaluation 38
3.3.1 Load Time & Memory Requirements 38
3.3.2 Query Response Time 39
3.3.2.1 Schema Queries on Models 39
3.3.2.2 Triple Queries on Models 40
3.3.2.3 Instance Queries on Models 41
3.3.2.4 Triple and Schema Queries on NS and GS 43
3.3.3 Summarizing the Results 44
3.4 Conclusions 45

4 Semantic Web Knowledge Middleware (SWKM) Services 47
4.1 Introduction 47
4.2 Overview and Design Choices 49
4.3 SWKM Services 51
4.3.1 Importer Service 51
4.3.1.1 A note in resolving dependencies for storing 52
4.3.2 Exporter Service 55
4.3.3 Query Service 56
4.3.4 Update Service 57
4.3.5 Comparison Service 58
4.3.6 Change Impact Service 60
4.3.7 Registry Service 63

4.3.8 Versioning Service . . . 73
4.3.8.1 Import Version Operation . . . 73
4.3.8.2 Create and Import Versions Operation . . . 76
4.4 Usage of Services Example . . . 79
4.5 Evaluation of the SWKM platform . . . 87
4.5.1 Strengths . . . 87
4.5.2 Weaknesses . . . 88

5 Related Work **91**
5.1 Related Work on Benchmarking RDF/S Main Memory Models . . . 91
5.2 Related Work on Versioning . . . 93
5.2.1 Ontoview . . . 93
5.2.2 SemVersion . . . 96
5.2.2.1 The MarcOnt Ontology Builder Case . . . 97
5.2.3 Blackboard Collaboration Architecture of ConcepTool . . . 99
5.2.4 MORE . . . 99
5.2.5 DIP . . . 102
5.2.6 GVS . . . 105
5.2.7 Summary Comparison . . . 107
5.2.7.1 Structural Diff . . . 109
5.2.7.2 Semantic Diff . . . 109
5.2.7.3 Change Log . . . 110
5.2.7.4 Change-based Deltas . . . 110
5.2.7.5 Concept Mapping between versions . . . 110
5.2.7.6 Storage Tools . . . 110
5.2.7.7 Storage Approach . . . 111
5.2.7.8 Declarative Query Language . . . 111
5.2.7.9 Branch . . . 112
5.2.7.10 Merge . . . 112
5.3 Open Issues . . . 113

5.4 Evaluation of the SWKM Versioning Service 114

6 Conclusion **119**

6.1 Synopsis and Key Contributions 119

6.2 Directions for Further Research 119

List of Tables

3.1 Benchmark Queries 35

5.1 Features of the comparison 108

5.2 Features comparison of versioning systems and tools 109

List of Figures

1.1 The Semantic Web Overview 2
1.2 The Java Enterprise Edition Architecture 3
1.3 The Microsoft Distributed interNet Applications Architecture (DNA) . . . 3

2.1 Dependencies among namespaces, graphspaces and namespaces and graphspaces 12
2.2 Sesame Architecture Overview 17
2.3 Sesame Data Structures 18
2.4 Jena Data Structures 22
2.5 Kowari Architecture Overview 25
2.6 JRDF Model Design 26
2.7 JRDF Data Structures 26
2.8 SWKM Data Structures 30
2.9 RDF/S Triple vs Object View in SWKM 31

3.1 Univ-Bench Schema 36
3.2 Load Time (left) / Memory Requirements (center) / Q1-Q7 (right) 38
3.3 Q3 (left) / Q7 (center) / Q8-Q15 (right) 40
3.4 Queries 16-21 41
3.5 Queries 22-28 44

4.1 An architectural overview of the SWKM services 48
4.2 Deployment of SWKM middleware and Client interaction 49
4.3 Importer service 51
4.4 Exporter service 55

4.5 Query service . . . 57
4.6 Update service . . . 57
4.7 Comparison service . . . 58
4.8 Comparison service interaction with Exporter and Main Memory Model . . 61
4.9 Change Impact service . . . 61
4.10 Change service interaction with Exporter and Main Memory Model 64
4.11 Registry service . . . 64
4.12 The Registry Schema . . . 67
4.13 Import Version Operation . . . 73
4.14 Create and Import Versions Operation . . . 76
4.15 Versioning service interaction with Exporter, Main Memory Model and Importer . . . 79

5.1 Comparing two ontologies in OntoView . . . 95
5.2 Multiple concurrent ontology branches . . . 98
5.3 Blackboard Architecture . . . 100
5.4 MORE System . . . 102
5.5 A commit dialog in DIP versioning tool. . . . 104
5.6 The three RDF graphs above show personal information from three sources. The first one asserts that a person who has first name 'Li' and surname 'Ding'. . . . 106

Chapter 1

Introduction

1.1 Introduction to the Semantic Web

The Semantic Web is an evolving extension of the World Wide Web in which web content can be expressed not only in natural language, but also in a format that can be read and used by software agents, thus permitting them to find, share and integrate information more easily. It derives from W3C director Sir Tim Berners-Lee's vision of the Web as a universal medium for data, information, and knowledge exchange.

At its core, the semantic web comprises a philosophy, a set of design principles, collaborative working groups, and a variety of enabling technologies. Some elements of the semantic web are expressed as prospective future possibilities that have yet to be implemented or realized. Other elements of the semantic web are expressed in formal specifications. Some of these include Resource Description Framework (RDF), a variety of data interchange formats (e.g. RDF/XML, N3, Turtle, N-Triples), and notations such as RDF Schema (RDFS) and the Web Ontology Language (OWL), all of which are intended to provide a formal description of concepts, terms, and relationships within a given knowledge domain. The proposed, by World Wide Web Consortium (W3C)[1], overview of the Semantic Web can be seen in Figure 1.1.

[1]http://www.w3.org/2006/Talks/1023-sb-W3CTechSemWeb/Overview.html#(19)

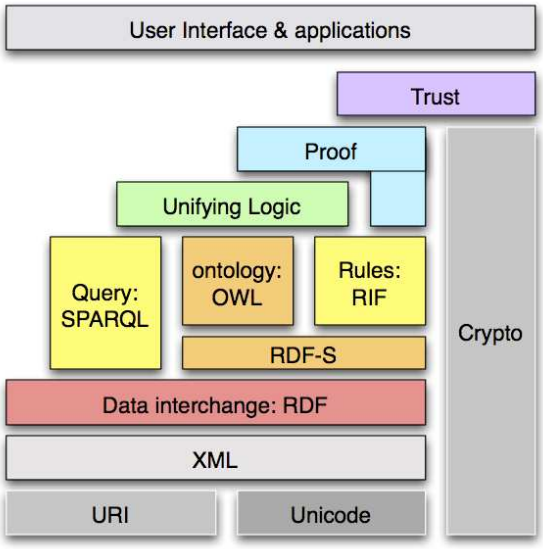


Figure 1.1: The Semantic Web Overview

1.2 The notion of a Middleware Platform for the Semantic Web

In the early years of the Internet, the client-server paradigm (also referred to as "two-tiered architecture") was the common way to develop distributed applications. In this paradigm, most of the application logic is bundled with the client, along with some user interface that issues calls to the server database. Each new application that needs to operate on the same data connects to the same database; so, the database becomes the main integration point between applications, providing them data sharing and (thus) communication.

This approach makes it very problematic to modify the data layer, as that is used directly by the applications, and modification to it cause those applications to break. Also, it is difficult to update the application logic itself (and enforce common updated policies), since this is bundled with each client, so every client installation must be updated independently. An even more serious problem is the increased network traffic; this is a direct consequence of application logic moved far away from the persistence layer, so all data needed by the application need to cross the network.

More recently, another paradigm became popular, known as "n-tier architecture", and

it was pioneered by platforms such as *Java Enterprise Edition* (JavaEE[2]) and Microsoft's Distributed interNet Applications Architecture (DNA[3]). A glimpse of this paradigm, expressed in either of these platforms (almost identically) is provided in Figures 1.2 and 1.3. In this architecture, *middleware* refers to software that is between the client application and the persistence layer, mediating both. In this sense, an application no longer has to deal directly with the database, but can rely instead on a set of services built on top of it. The implementation details of the persistence are hidden, and the middleware can guarantee the consistency of the database (which would not be the case if applications would contact and update it directly). Also, as usually the middleware resides close to the database, this transition entails a win in performance, as more computation is pushed closer to the data, avoiding costly data transmissions over the wire.

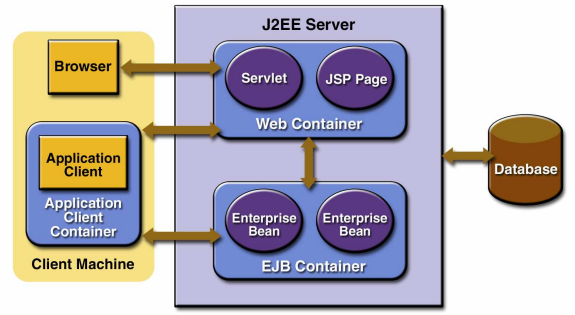


Figure 1.2: The Java Enterprise Edition Architecture

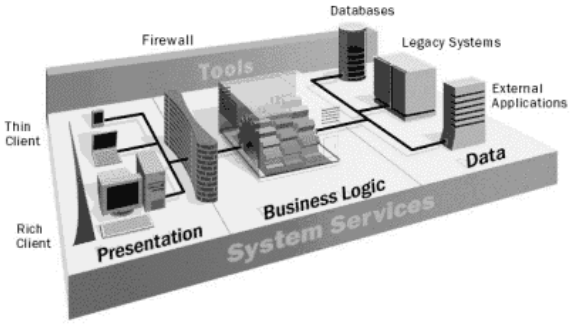


Figure 1.3: The Microsoft Distributed interNet Applications Architecture (DNA)

A middleware for the Semantic Web follows the same general architecture and additionally offers more specific services regarding Semantic Web data. In briefly reviewing the available middleware services provided from each major RDF management system,

[2]java.sun.com/javaee/
[3]http://www.microsoft.com/technet/archive/itsolutions/intranet/plan/introdna.mspx

we focus on the main memory representation of RDF data models, on their long-term persistence, and on access of data.

1.2.1 Main Memory Representation

In general, a main memory representation of SW data is useful for two main purposes: (a) for the internal functioning of the middleware (to support validation, provide services that are able to manipulate them fast, etc), and (b) for providing clients an appropriate programmatic abstraction of SW data (for client-side manipulation) which in addition is consistent with the capabilities of the underlying SW repository.

1.2.2 Persistence

Probably the most important functionality that a middleware offers is the provision of persistence services that hide the details of the particular physical layer used. Typically, SW data have to persist (except the most trivial cases) across multiple application life-cycle periods, and may well outlive the application itself. In addition, it should provide the classical transactional semantics (atomicity, concurrency, isolation, durability) in an efficient manner. Scalability is also an issue, inasmuch available SW data grow worldwide at an adequate volume to bolster the vision of the Semantic Web itself.

1.2.3 Access

The middleware should offer expressive (e.g. declarative) and efficient access services to the SW data stored in the underlying repository. The expressibility of access methods are directly correlated with the effort needed to create application that take advantage of the available data by accessing them in convenient ways. Examples of what access paths can be more "convenient" to developers are given in later chapters (in particular, see Section 2.1.5).

1.3 Motivation for Benchmarking Main Memory Representations

The main memory representation of SW data determines in a large degree the efficiency of applications built over the middleware. This is due to the fact the main memory representation affects the efficiency of both server-side and client-side tasks. Furthermore, in main memory processing it is not too obvious what to optimize (in contradiction with secondary memory, where IO accesses are the major factor for latency), since applications use much more fine-grained APIs to access the main memory representations, thus there is not much of a chance to optimize these very short-lived accesses; and deriving at runtime the big picture of an application's accessed paths is problematic and often ignored in order to avoid the runtime overhead. So, in almost all cases main memory representations choose predefined indexes by assuming that these will be the most cost-effective, and it is particularly interesting to find out if those implementation decisions have been correct or not, to what extend and for which access patterns. By having this information, application developers can make more intelligent decisions when picking main memory representations to develop time-critical applications, and framework designers can get valuable feedback so as to improve problematic areas in their tools.

1.4 Motivation and Application Scenarios for Versioning Services

In this thesis, there is a particular focus on versioning facilities for the Semantic Web. To understand the issue better, we will go through some motivating application scenarios that may benefit from versioning facilities, and examine the relative requirements of each.

1.4.1 Collaborative Development and Evolution of Ontologies

An ontology is a shared knowledge artifact of a community. A matter of fact is that if it is actively used by a community, the community will drive the need for it to *evolve*.

Individuals propose changes to the ontology, that may involve changes proposed by other members, or be completely independent. Without versioning, individuals would either wait for a general consensus to be reached, or would create a copy of the ontology, apply their desired changes, and experiment with it to gain further insight of the consequences. This is a rather ad-hoc process, and members collaboration is prohibited - each member have to keep track of his/her own changes, and changes of other members that interest him/her the most. All these are moving parts, and the risk of human error increases. Furthermore, combining the work of different individuals is not as easy as it should. Finally, each member has to worry about keeping backups before each important change, and then check for compatibility problems, and possibly revert to a backup.

A versioning system should allow its users to freely perform changes, create new versions and experiment with them, incorporate changes of others, recombine changes, without fearing of introducing irreversible problems. The creation of new versions, with under discussion features, enables a richer and less theoretic discussion of the extension, as the changes are accessible to all, and each member can put the proposed ontology under test and produce further feedback. At a later stage, the decision to change the authoritative (aka *trunk*) version of the ontology can be accommodated, either by extending the existing version to include the desired features, or by directly *merging* changes that produces experimental versions.

In such a scenario, functional requirements for a versioning would include:

- Extend an existing version to produce another
- Annotate a version with arbitrary metadata such as author, intension, issues, comments etc.
- Evaluate queries over versions and such metadata
- Select and combine changes, apply them in an existing version to produce another, and detect and resolve possible conflicts
- Analysis of changes and description of their consequences (such as the exact compatibility problems that are introduced)

As per the performance requirements, it is obvious that in a collaborative setting, the most costly factor to consider is the time spent by humans. So, a versioning tool should concentrate on providing the highest performance in tasks like:

- Create a version
- Retrieve a version
- Retrieve change log

Normally, heavier disk usage can be afforded in order to accommodate higher performances for the above tasks. Note, however, that excessive disk usage will deteriorate operation times as well.

1.4.2 Collaborative Development and Evolution of Concepts

A more free-form collaboration setting is one that does not require centralized decision points at all. Imagine a community portal into which there is no need to come to a structured, general consensus of an ontology at all, in a Web 2.0 fashion. Users spontaneously define their concepts or reuse definitions of their peers, and an ontological landscape emerges, where the relative *importance* of concepts is measured by the number of usages or instantiations. An example of such a portal is described in [51]. Note that concepts increase monotonically, and redundant or obsolete concepts need not be removed, so a weaker versioning model that only allows a single branch of evolution is enough [4].

This versioning model results in a slightly different set of functional requirements, where the focus is not so much the management of ontologies, but the management of concepts, and who created/used them, and when. These can be summarized as tracing concept history:

- Annotate a version with arbitrary metadata such as author, intension, issues, comments etc. (as in previous scenario)

[4] An applicable approach for this kind of versioning model is described in [46]

- Evaluate queries over versions and such metadata (as in previous scenario)
- Find instantiations of a concept in a time interval
- Find time interval that a concept is in use (or when a concept was used for the first time)

Queries that can address these requirements can be expressed given an operator that relates concepts to their creation time, which in turn can be built on top of a more primitive operator that correlates triples to versions, which themselves can be associated with time of creation metadata.

Performance-wise, insertions should be performed very fast, which is what the end-user expects as typically insertions happen incrementally in small volumes. Even more importantly, as typically read operations vastly outnumber write operations, what is needed is real time performance for the evaluation of history trace-type queries, such the ones described above. Retrieval of the full contents of an ontology is not at all a critical operation, so a versioning system should be able to choose data structures and algorithms with very low retrieval performance, in order to attain higher performance at more critical operations.

1.4.3 Archiving and Preservation

Another important usage scenario involves information preservation and archiving, where the prime focus is on storing vast amounts of knowledge in a future-proof, searchable way. In the Semantic Web context, such activities can be translated into creating ontologies that can capture the semantics of the data to be archived. These ontologies eventually need to change, especially since archiving focus on the long term rather than the short, to accommodate the representation of new types of knowledge, or generally changes in the real world or its conceptualization. An example of archiving, out of the Semantic Web scope though, is the *Internet Archive WaybackMachine*[5]. Moreover, several scientific communities (e.g. biological, medical etc) produce and use voluminous data

[5]http://www.archive.org/

annotated by ontologies or similar structures, which are periodically extended, refined, or revised, and it would be desirable to store and manage efficiently every available version.

Functionally, this scenario does not bring any new requirement, except perhaps the need to sometimes track the *provenance* of preserved knowledge; but this can be covered by the more general requirement of storing arbitrary metadata, as stated in the first application scenario.

Performance-wise though, archiving impose vastly different priorities than other tasks. The main focus is the reduction of storage space requirements, which may happen in expense of performance of creating new versions and, secondly, in expense of querying capabilities. The performance of the latter though cannot be degraded too much, as data are valuable only so long as they are retrievable in reasonable time constraints.

1.5 Contributions

In a nutshell, the contribution of this thesis lies in:

- benchmarking several different main memory representations, and
- designing and implementing versioning services for SW data

Specifically, several contemporary RDF management system/middleware platforms are analyzed in terms of architecture and supported services. The various abstractions of main memory models of each platform are compared, and conclusions and useful advice about which model is more preferable for typical use cases are given. In addition, this thesis provides a detailed description of the SWKM platform architecture, its available middleware services and implementation, and in particular its versioning services. These services are mainly concerned with the application scenario detailed in Section 1.4.1; the others our out of the scope of the current design goals, and may be the subject of future research. Various design decisions behind the versioning services are discussed, by also comparing with other systems where relevant, and insights are provided on the related problems of implementing versioning facilities.

1.5.1 Organization of the thesis

Chapter 2 describes various currently available middleware platforms in terms of their main memory representations, their persistence-related services and their provided access paths.

Chapter 3 presents the experimental setting and benchmark results for the main memory representation models of the aforementioned middlewares, with the respective drawn conclusions.

Chapter 4 delves into the design, architecture and implementation decisions of the SWKM middleware in particular, as well as its versioning services specification and implementation desicions.

Chapter 5 examines the state of the art in benchmarking main memory representations and in versioning services and tools for the Semantic Web.

Chapter 6 summarizes the results of this thesis and identifies topics that are worth further work and research.

Chapter 2

Semantic Web Middleware Platforms

2.1 Introduction

This section reviews various currently available middleware platforms. As briefly mentioned in Section 1.2, the discussion will focus on three perspectives: (a) on the main memory model the platform uses and offers for RDF data manipulation, (b) on persistence-related services, and (c) on the available access paths to the persisted RDF information. The latter subsections introduce these points

2.1.1 RDF Data Model

In the RDF data model[1], a universe of discourse to be modeled is a set of resources (essentially anything that can have a universal resource identifier, URI). Resources are then described through a set of properties (i.e. binary predicates) while descriptions are statements (i.e. triples) of the form `subject-predicate-object`: a *subject* denotes the described resource, a *predicate* denotes a resource property, and an *object* the corresponding property value. The predicate and object [2] are in turn resources (or literal values). A vocabulary (called *namespace*) of the properties but also of the classes employed in

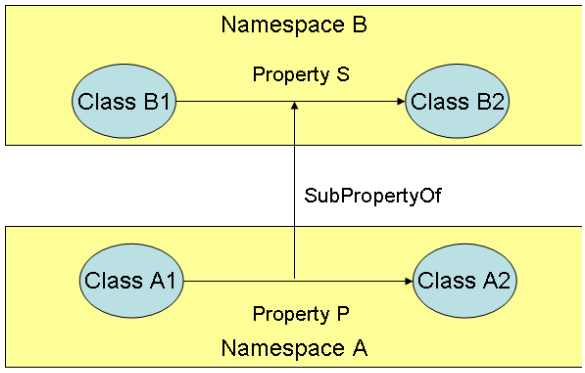
[1] www.w3.org/TR/rdf-concepts

[2] Note that the subject and object of a statement can be also non-universally identified objects, called unnamed resources (or blank nodes), whose URI could not be shared across different RDF descriptions.

resource descriptions can be defined in the RDF Schema (RDFS[3]) language (also represented in the basic RDF model). More formally, a namespace defines uniquely (via its associated namespace URI) a container of schema elements; classes, properties, etc. This provides a means to avoid naming collisions of elements between different organizations which could happen to choose the same simple name, e.g. "Person". Instead, this simple name is transformed into a URI by adding its namespace prefix, i.e. the URI of the namespace that it belongs, for example "$http://www.example.org/\#Person$". The namespace in this example is "$http://www.example.org/\#$". In this document, we distinguish the terminology between "namespace URI", which is a name (string), and "namespace" which is meant as a container of schema elements that share as prefix the same namespace URI.

An important extension to this model is the so-called *Named Graphs*[11], also known as *graphspaces*, the preferred term in this document. Graphspaces provision for the association of a name (URI) to a set of triples; from that point, this particular set of triples can be referred collectively by using its respective URI. A motivating application scenario for graphspaces is provenance tracking; i.e. who asserted what facts.

2.1.2 Dependencies between RDF spaces



(a) Namespace A depends on Namespace B

Figure 2.1: Dependencies among namespaces, graphspaces and namespaces and graphspaces

[3]www.w3.org/TR/rdf-schema

Each RDF space is allowed to depend on declarations found in other RDF spaces. For instance, a property P, defined in namespace A may be a subtype of a property S defined in a namespace B. In this case, we say that namespace A depends on namespace B. Also, data instances in a graphspace C will be instances of a class defined in a namespace D. In this case, graphspace C depends on namespace D. Finally, data instances in a graphspace E may refer to a graphspace F. In this case, graphspace E depends on graphspace F.

For understanding and using an ontology, it may be needed to resolve such a dependency. For instance, before storing a single namespace it needs to be validated (to ensure, for example, that no cyclic class inheritance exists). If it depends on another namespace, it cannot be properly validated without the presence of the other namespace (in the previous example, a class that is part of an inheritance cycle may be defined in a different namespace than the one that is being stored; the cyclic dependency cannot be detected without processing the second namespace).

2.1.3 Declarative Query/Update support

Several query languages (e.g. *RQL*[30], *SPARQL*[4], *iTQL*[5]) have been developed during the last years for supporting declarative access to ontologies and resources descriptions available on the Semantic Web. A comparative discussion about the relative merits and weaknesses of each declarative query/update language is out of the scope of this thesis.

SWKM in particular uses RQL. RQL is a typed declarative query language for RDF. It is defined by a set of basic queries and iterators that can be used to build new ones through functional composition; it can combine schema paths for executing complicated schema navigations; not many languages support this type of queries. However, its major innovation lies in its ability to ask queries both on the schema and data levels. It supports generalized path expressions featuring variables on labels for both classes and properties, i.e. nodes and arcs in the graph representation, respectively. Finally, it provides set-theoretic operators, allows using XML Schema data types, aggregate functions and arithmetic operations on data values.

[4]http://www.w3.org/TR/rdf-sparql-query/
[5]http://www.kowari.org/oldsite/271.htm

One of the unique features of RQL is its ability to match filtering/navigation patterns against RDF/S graphs by taking into account (or ignoring) the semantics (e.g. transitivity of subsumption relationships) of the ontologies employed to describe knowledge artifacts (see [23] for a detailed comparison of SW QL expressiveness). This functionality is useful for abstracting the technicalities of the RDF/S data model while it has been efficiently implemented in secondary memory.

SWKM also offers a declarative language for updating RDF knowledge, called *RUL*[38]. This language ensures that the execution of the update primitives on nodes and arcs neither violates the semantics of the RDF model (e.g. insert a property as an instance of a class) nor the semantics of a specific RDFS schema (e.g. modify the subject of a property with a resource not classified under its domain class). This main design choice has been made given that type safety for updates is even more important than type safety for queries: the more errors catched at update specification time the less costly runtime checks (and possibly expensive rollbacks) are needed. It is also smoothly integrated with RQL.

To support RQL and RUL evaluations, an interpreter has been developed (implemented in C++). This operates directly upon this persistence layer, that can both retrieve knowledge/answer queries (expressed in RQL) or update the knowledge base (with RUL) respectively. Currently, there are several provided mappings between RDF/S and the (object-)relational model that the interpreter can handle, with different performance properties, named as: *Schema Oblivious*, *Schema-Aware*, *Hybrid* [44][7]. RDF/S representations are mapped and stored as relations in the database.

2.1.4 Selection Criteria

We selected four Java-based main memory representation management systems (MM-RMS) for the main memory model benchmark, namely Sesame, Jena, JRDF (used as the main memory API of Kowari), and SWKM , which, to the best of our knowledge, cover the full spectrum of general purpose Java programming frameworks available nowdays to SW developers. In particular, we are interested in understanding the impact of their

various architectural and implementation choices when constructing or accessing programmatically RDF/S schemas and instances. Our choice to stick with RDF/S is motivated by the fact that according to a recent survey [14] 85.45% of the SW relies on RDF/S, while only 14.55% on OWL.

2.1.5 RDF/S Triple vs. Object-based Views

The basic abstraction SW developers have in their disposal to program is that of an *RDF/S Model*, which encapsulated the data model expressed in Section 2.1.1. A model consists of one or more collections of triples and groups all the methods for constructing and accessing RDF/S schemas and instances. All four MMRMS used in our benchmark represent triples as objects, whereas the three of them (Jena, Sesame, JRDF) model also the concept of node, for representing the corresponding *subject*, *predicate* or *object*. Since a node is generic enough to represent almost anything in RDF/S, it cannot carry any type information (so we cannot know what kind of specific RDF/S construct it represents, such as a class, property or individual resource). It is usually used either as a storage facility for triples (in Sesame) or just as a reference object during triple creation (Jena, JRDF). In general the three aforementioned MMRMS support, in a similar manner, the concepts of the model, the triple and the node.

However, SW developers are more akin to program with objects having a concise state (i.e. attribute values) and type information (i.e. classes they instantiate). This is due to the old and good reasons of modern programming paradigms: ease of use, speed of navigation and access, better encapsulation and exploitation of capabilities like inheritance and polymorphism. Such an *object-based view* would map RDFS classes to programming language classes, RDF resources to programming language objects and RDF predicates to data or function members of those objects. Unfortunately most of the so far developed MMRMS do not provide an object-based view of the RDF/S schemas and instances represented in Main Memory. Some efforts to provide a programmatic object creation for RDF/S based schemas, like RDF Reactor [47] and RDF2Go [47] require that the RDF/S schema is parsed in advance and the corresponding Java classes to be created before the actual code can be written, making such approaches at least cumbersome and resilient

to changes. Other efforts, like ActiveRDF [39], rely on the dynamic typing capabilities of scripting languages such as Ruby in order to bridge to some extend the gap between RDF/S and the object oriented world.

2.2 Sesame

Sesame[6]. is an open source Java framework for storing, querying and reasoning with RDF and RDF Schema. It can be used as a database for RDF and RDF Schema, or as a Java library for applications that need to work with RDF internally. Sesame provides the user with the necessary tools to parse, interpret, query and store RDF data, embedded in user applications or in a seperate database on a remote server.

A central concept in Sesame is the *Repository*. A repository is a abstraction of storage container for RDF data. This can mean Java objects in memory, or it can mean a relational database. Virtually all operations in Sesame happens with respect to a repository: the repository is the provider of persistence and querying capability.

Sesame also supports RDF Schema inferencing. This means that given a set of RDF and/or RDF Schema, Sesame can find the implicit information in the data. Sesame supports this by simply adding all implicit information to the repository when data is being added.[7]

2.2.1 Architecture

Figure 2.2 shows an architectural overview of Sesame. Starting at the bottom, the Storage And Inference Layer, or *SAIL* API, is a Sesame API that abstracts Sesame's persistence, and is described in Section 2.2.3

On top of the SAIL, we find Sesame's functional modules, such as the SeRQL[10], RQL[30] and RDQL[41] query engines, the admin module, and RDF export. Access to these functional modules is available through Sesame's Access APIs, consisting of two seperate parts: the Repository API and the Graph API. The Repository API provides

[6]http://www.openrdf.org

[7]As of the upcoming version 2.0, inferencing capabilities will be independent from any specific repository. In prior versions it was up to a repository whether it supported or not inferencing.

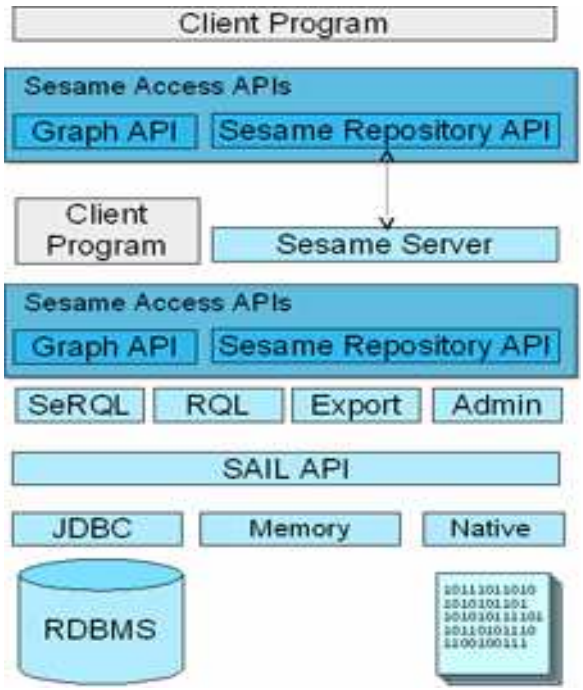


Figure 2.2: Sesame Architecture Overview

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.

The Access APIs provide direct access to Sesame's functional modules, either to a client program (for example, a desktop application that uses Sesame as a library), or to the next layer of Sesame's middleware. The middleware provides HTTP-based access to Sesame's APIs. Then, on the client side of the remote invocation we again find the access APIs, which can again be used for communicating with Sesame, this time not as a library, but as a server running on a remote location.

2.2.2 Main Memory Model

Sesame's 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. As we shall discuss in upcoming sections, Sesame's main memory model is

triple-based view oriented, a choice that at least affords its API to be very concise.

2.2.2.1 Data Structures

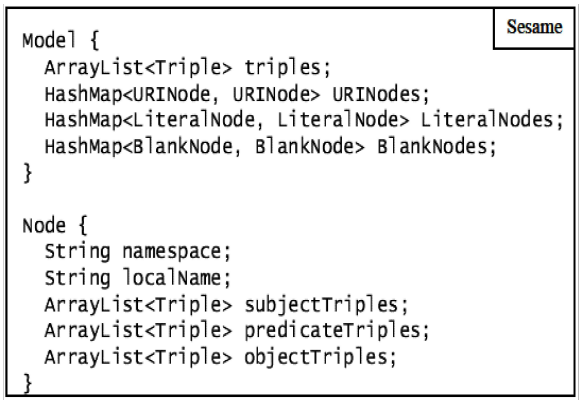


Figure 2.3: Sesame Data Structures

As we can see in Figure 2.3, an RDF/S model in Sesame keeps an `ArrayList` of all its triples (`triples`). Moreover, it also holds three `HashMap`s with all kinds of nodes appearing in the model triples, namely literals, named resources (nodes that carry a URI) and anonymous resources (blank nodes). Both the key and value of these Maps are the node itself (object reference). Moreover, each node contains three `ArrayList`s of triples (`subjectTriples`, `predicateTriples`, `objectTriples`) having the given node as *subject*, *predicate* or *object* respectively.

When a node needs to be created or retrieved from the memory, a new node is created and then looked up in the Maps. If a node exists, the already saved node will be used instead and the newly created one will be garbage collected. On the other hand, when a new triple is to be inserted, at first, three nodes are created (for its subject, predicate and object) and looked up in the Maps described earlier. If one of these new nodes is not found, then it is saved in the corresponding Map. Once the subject, predicate and object nodes are retrieved, the triple is looked up in the `triples` list of the model. When the triple is not found, it is saved in `triples` while the corresponding lists of nodes appearing as its subject, predicate and object are updated.

It should be noted here that `Model` and `Node` are not classes of the Sesame API, i.e. there is no access method allowing the programmer to retrieve them or any other method facilitating the retrieval of related nodes, as for instance, to find the superclasses of a

specific class. They are just used as holders for the containers of the triples, so as the API to be able to find their location. Moreover, only the URI or the literal information contained in a `Node` is actually represented in the triples and not the object instantiating the `Node`.

2.2.2.2 Access Methods

There are only two access methods for model triples in Sesame. The first one, returns an `Iterator` over triple lists, given the desired subject, predicate and/or object. If one or more of the search arguments is null, for instance the subject, this implies that any subject is accepted in the search results. If not null, all method arguments are looked up in the node's Maps described above. Then, the subject, predicate and object lists of a node are actually searched and the one with the fewer triples is selected each time (to minimize the search space). When iterating through the result, the triples of the list returned in each case are acquired one by one, and the subject, predicate and object are checked to see if they match the corresponding arguments employed to invoke the method. This means that most of the searching time for triples is spent on iterating through the results, while time for actually getting the result (iterator) is negligible. The choice of returning an `Iterator` was made for two reasons. Firstly, the user should not be able to change the contents of a returned list to respect encapsulation of the model. Secondly, an iterator allows using the same main memory to return the results. The second method returns an `Iterator` to all the triples of the model. Calling the second method is the same as calling the first one with all three arguments being null.

2.2.3 Persistence Storage

SAIL is Sesame's abstraction 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. In the persistence layer, there are SAIL implementations for PostgreSQL, MySQL, SQL Server and Oracle database. SAIL implementations can also be stacked on top of each other, to provide functionality such as caching or concurrent access handling. Each Sesame repository has its own SAIL object to represent it. The SAIL abstraction defines very

few and basic operations, such as adding and removing triples, starting and committing transactions, clearing the repository etc. Whether this abstraction is rich enough to allow implementations to take advantage of available optimization opportunities (for instance, to implement bulk uploading with a SAIL interface, Sesame partitions the uploading in sequential transactions containing 1000 additions of a triple each, regardless whether the underlying implementation supported streaming) remains to be seen.

2.2.4 Sesame Services

2.2.4.1 Query Service

The Query Service of Sesame's middleware regards the remote evaluation of SeRQL, RQL and RDQL type of queries. Various serialization syntaxes are supported, chosen by user parameters. SeRQL in particular can also be used to construct and return RDF graphs.

2.2.4.2 Extract RDF Service

An "Extract RDF" service is provided, with the purpose to retrieve the (possibly filtered by the parameters explained below) contents of a repository. A client can specify the following parameters when requesting to extract a particular repository: (a) whether schema information should be extracted, (b) whether data instances (not schema) should be extracted, (c) whether only explicit triples will be exporter, or implicit too.

2.2.4.3 Upload Data Service

Another service, which complements adding triples to a SAIL interface, is the "Upload Data" service, which has the purpose of loading into the database entire RDF documents. The parameters a client can control (apart from the RDF content itself): (a) a baseURI (optional) for resolving any relative URI in the RDF document, (b) whether data validation should take place. A request to Upload Data service may result in getting a list of validation errors, with error message, and line and column where the error occured.

2.2.4.4 Clear Repository Service

Sesame offers an easy way to clear a repository from its contents, without removing the initial schema of the underlying database (if applicable). This is particularly useful in testing.

2.2.4.5 Triple Removal Service

This service allows for a triple-pattern (`subject-predicate-object`) to be defined, and all matching triples to be removed. This may become more powerful in upcoming versions; currently the problem of the absence of a general declarative update language is apparently limiting. It should be noted that removals are performed without managing side-effects, that is, triples are removed directly and the end result depends on whether the repository supports inference or not.

2.2.5 Summary

We saw that Sesame offers http-based interfaced based on its underlying APIs, packaged in its middleware, and can act as a centralized mediator of RDF repositories, with a quite respectable array of services. It is also noted that the services themselves are simple wrappers over HTTP requests[8], making them easily accessible with commodity technology.

2.3 Jena

Jena[9], to the best of our knowledge, does not offer a middleware offering per se. But using Jena framework one can tailor custom middleware services. We will describe here the main Jena tools which could be used to create them.

[8] An application of the RESTful approach
[9] jena.sourceforge.net

2.3.1 Main Memory Model

The main memory model of Jena, as Sesame, is based on the concept of a graph, created by URIs/nodes and triples/edges. It seems that there were much influence between these two projects, as large parts of model related APIs are surprisingly similar. A significant design difference exists in the handling of inference. In Jena, inference is provided through a proxy model, that offers inference based on an underlying model that it talks to while in Sesame, inference is placed on the persistence/access layer, namely by a inferencing SAIL implementation that is also based to an underlying SAIL for data access.

2.3.1.1 Data Structures

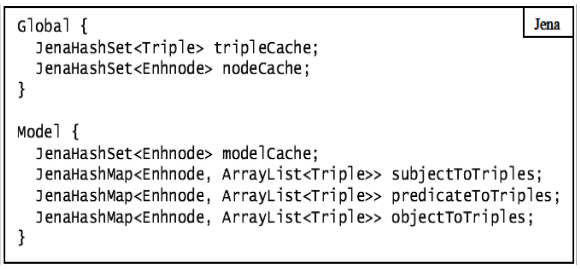


Figure 2.4: Jena Data Structures

As we can see in Figure 2.4 an RDF/S based model in Jena is kept in three `HashMaps` each one containing all triples as values and using as key the *subject*, the *predicate* and the *object* respectively (`subjectToTriples`, `predicateToTriples`, `objectToTriples`). These Maps are updated synchronously when a new triple is inserted, thus assuring consistency among them and minimizing search cost (so if a triple is found in one it is guaranteed to be in the rest). Moreover, it also keeps a `HashSet` of enhanced nodes (`EnhNode`), which is a class that contains references to the node, as well as to the model in which it is contained. This is the only construct that keeps explicit information regarding the available nodes, where a node can again be any subject, predicate or object of any triple.

The size of all the above memory constructs is doubled when capacity reaches 50%, resulting in 2 to 4 times bigger than needed memory consumption; all of them have an initial size of 10. The index used is calculated using the hash code of the (string) label

of the key. When a collision occurs, the first position that is empty, before the one expected, is used, resulting in worst case linear lookups instead of constant ones. As also illustrated in Figure 2.4, Jena introduces the concept of *Global Cache*(s). It uses two caches (implemented as `HashSets`), one for triples and one for nodes. These caches keep only a predefined number of objects (1000 for triples and 5000 for nodes, respectively) and are used to facilitate fast access to recently used triples or nodes, since they do not have to be looked up in the Maps described above.

2.3.1.2 Access Methods

Jena provides also four access methods for model triples. There exists a method that retrieves all triples of a model and one that returns the triples matching the subject, predicate and object or any combination of them passed as arguments. The former returns an `Iterator` of all triples in the model, in which case the map that contains the subjects as keys is used. Additionally there exists a method returning all subjects, all predicates or all objects of the triples of a model and one returning all subjects with a given property, all objects with a given property and so on. These methods along with methods of the form *"find all subclasses of subject X"* choose which of the Maps (`subjectToTriples`, `predicateToTriples`, `objectToTriples`) to work on based on the available input, e.g. if the subject is given then they use `subjectToTriples` and so on. As already mentioned, all triples that are looked up in any model are also stored in the Global Cache. Finally, there are no methods that can return any kind of information regarding RDF/S namespaces and graphspaces.

2.3.2 Persistence Storage

Jena offers a single persistence solution[10], implemented in various RDBMS. It follows the "schema-oblivion" approach, in which there is one and only one relational schema, regardless of whatever RDF/S schema are stored into the database. This approach is not recommendable as has been shown; for an in-depth discussion, refer to [42].

[10]Described here: http://jena.sourceforge.net/DB/layout.html

2.3.3 Services

To the best of our knowledge, Jena does not offer a full suite of middleware services, nor persistence-related services in particular, although it should be pretty easy for one to build such services on top of Jena persistence tools. But a quite important service is implemented by the collaboration of two subprojects: ARQ[11] and Joseki[12]. ARQ is an implementation of SPARQL and Joseki an implementation of the SPARQL Protocol[13], standardized by the World Wide Web Consortium. The SPARQL Protocol defines a query service, bindable to HTTP (GET and POST actions) or SOAP[14]. This service evaluates SPARQL queries, and returns its results in a well defined format, so in theory one can target a query to various implementations of SPARQL Protocol and be able to understand the result without having to know any implementation-specific details.

2.4 Kowari

Kowari[15] is an open source, scalable, transactional database for the storage, retrieval and analysis of metadata. Kowari is written in Java and it supports Resource Description Framework (RDF) and Web Ontology Language (OWL) metadata.

2.4.1 Architecture

Figure 2.5 shows an architectural overview of Kowari[49]. The topmost shows the access APIs to Kowari, that connect (see the layers below) remotely to a query engine that can operate on several types of storage. These include Simple Ontology Framework API (SOFA[16]), Java RDF (JRDF[17]), Jena, interactive Tucana Query Language (iTQL), RDF Data Query Language (RDQL[18]) and Simple Object Access Protocol (SOAP). The JRDF, SOFA, Jena and iTQL APIs can all be used on the client side or in-JVM local to

[11] http://jena.sourceforge.net/ARQ/
[12] http://www.joseki.org/
[13] http://www.w3.org/TR/rdf-sparql-protocol/
[14] http://www.w3.org/TR/soap/
[15] http://kowari.sourceforge.net
[16] http://sofa.projects.semwebcentral.org/
[17] http://jrdf.sourceforge.net/
[18] http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/

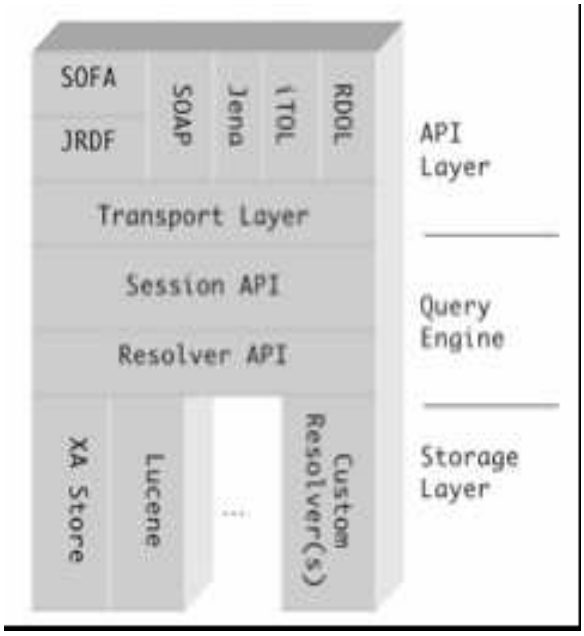


Figure 2.5: Kowari Architecture Overview

the server. The Session and Resolver APIs are responsible for taking access requests and directing them to the storage layer.

The Resolver API is located at the interface between the upper parts of the query engine and the storage layer. An incoming query is broken down to the point where individual constraints are to be resolved against individual RDF models. The Resolver API has been developed to allow these simple operations to be applied against almost any kind of data source. The intention is to allow pluggable graphs implementations other than the native Kowari data store. These alternative graph implementations are called Resolvers. When Kowari starts, a configuration file is read that contains a list of Resolvers to use. As each Resolver implementation is loaded, it registers its ability to handle a type of model. When a constraint is to be resolved against a model on the current server, the query engine looks up the type in the system model, and maps the request to the appropriate Resolver.

2.4.2 Main Memory Model

As already noted, Kowari's primary Main Memory Model is provided by JRDF. JRDF, as Sesame and Jena, offers a graph-based view of RDF data, and in Figure 2.6 we a hierarchy that shows how a node can be mapped to an RDF element.

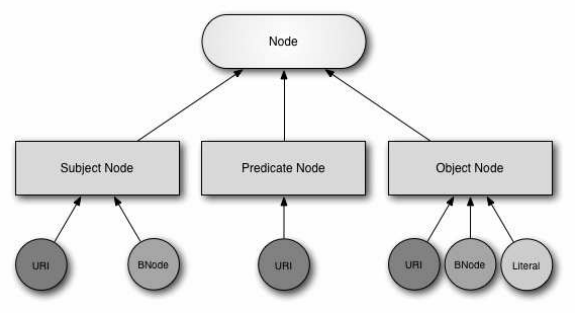


Figure 2.6: JRDF Model Design

Nothing is significantly different in this approach than the aforementioned ones, so we can with the analysis of its internal data structures, which are quite different, something that is also exhibited in performance characteristics.

2.4.2.1 Data Structures

JRDF

```
Model {
  HashMap<Subject, HashMap<Predicate, Object>> spoMap;
  HashMap<Predicate, HashMap<Object, Subject>> posMap;
  HashMap<Object, HashMap<Subject, Predicate>> ospMap;
  HashMap<URI, ID> StringPool;
  HashMap<ID, object> NodePool;
}
```

Figure 2.7: JRDF Data Structures

As we can see in Figure 2.7, JRDF in order to store an RDF/S model keeps three `HashMaps` of all the available triples (`spoMap`, `posMap`, `ospMap`). These Maps use as key one of the *subject*, *predicate* or *object* and as value another `HashMap` with the other two (using the second as key and the third as value respectively). It should be noted here that JRDF provides no explicit programmatic construct of a triple but instead keeps the nodes in the corresponding Maps as described. This implies that if we want to retrieve a triple object we have to construct it every time we need it (even if it is the same), resulting in some more overhead.

Additionally, two `HashMaps` (`StringPool`, `NodePool`) are used in order to hold the nodes (where a node in JRDF can be, as in Sesame, a literal, a URI or a blank node). The first one uses the URIs of the nodes as key and the (long integer) IDs as value whereas the second uses the IDs as key and the (node) objects as value. The node IDs are then used throughout the system to refer to the actual objects.

2.4.2.2 Access Methods

There is only one method supported by JRDF to retrieve triples. Like in all previously described MMRMS, it takes as arguments the *subject*, *predicate* and/or *object* of the inquired triple and returns an `Iterator` over the result. At first, the given nodes are looked up in the string pool, so as to retrieve their ids. Afterwards, depending on the passed arguments, the three maps described previously are used. For instance, if only the subject and object are given as arguments, the Map where the object is the key and the map holding the subjects as keys and the predicates as values will be used. So, when the user iterates through the results, the set of predicates that corresponds to the given subject and object are scanned, and a new triple, holding the given subject and object as well as the retrieved predicate is constructed and returned.

2.4.3 Persistence Storage

Kowari's native data store (a XA[19] enabled statement store) is a native Java transactional data store that stores RDF statements persistently on disk without the use of an external database.

The Statement Store stores statements as quad-tuples consisting of subject, predicate, object and meta nodes. The first three items form a standard RDF statement and the meta node describes which model the statement appears in. Each quad is unique, so a statement that appears in two models will be listed twice, with differing meta node values.

Kowari stores RDF statements in six different orders (indices). This corresponds to the minimum number of ways the four node types can be arranged such that any collection of one to four nodes can be used to find any statement or group of statements that match. Each of these orderings therefore acts as a compound index, and independently contains all of the statements for the RDF store.

Each index in Kowari is an AVL tree that provides addressing information into a large random-accessed file. Using this structure allows the AVL trees to remain relatively small and can often fit into physical memory. In practice, the speed of searching, insertion and

[19]XA is widely used distributed transaction processing protocol. Refer to http://www.opengroup.org/onlinepubs/009680699/toc.pdf for the specification

deletion operations are linear when the depth of the AVL trees is relatively small. As trees may become unbalanced during write operations, they must be rebalanced, often by rotation of a node's children. This is a simple operation on the addressing information tree that does not affect the underlying data.

2.4.4 Services

Kowari at the server side offers various remoting options to the underlying iTQL interpreter, which handles client *sessions*, which can issue database queries and updates. These options are:

- JavaServer Pages [20] (JSP) tag libraries, that is, markup tags that can be embedded directly in JSP pages
- SOAP-based web services

Also, a client is offered that communicates directly with the iTQL interpreter of the server. Apart from exposing iTQL, there are no other server-side services offered, but the intention is clear: instead of having separate services, make available all functionality as expressions in the iTQL language. For example, iTQL has language constructs to allow one to import an RDF/XML or N-Triple file into the database (although this can only be done via URLs; i.e. it is not possible to import a file that is generated by a program but not accessible through a URL), start/commit/rollback transactions, taking backups, etc.

This approach has both its advantages and its disadvantages. By having a uniform way to access all functionality, only one remote endpoint need to be configured and accessed by the client, which simplifies deployments of both server and client. But on the other hand, developers instead of having APIs with methods, parameters etc, only have a generic "input: string, output: string" function, and they have to manually translate/encode requests into a particular syntax, instead of calling a more type-safe method. That is unless developers use Kowari-specific client helper components which could offer such APIs (but note that if developers are free to use Kowari on both server and client endpoints, then it might be the case that the protocol could be a more efficient binary one).

[20] A popular server-side templating solution: http://java.sun.com/products/jsp/

2.5 SWKM

SWKM[21] is developed by ICS-FORTH organization, and partially supported by EU projects KPLab[22](IST-1999-13479) and CASPAR[23]. It's purpose is to provide general and scalable knowledge management services based on Semantic Web technologies.

2.5.1 Architecture

An elaborate technical review of the SWKM platform architecture is given in Chapter 4.

2.5.2 Main Memory Model

SWKM's model is the only MMRMS that supports a fully-fledged object based view of RDF/S. This view allows typing information to be carried along with the objects themselves while provides object methods for storing and accessing RDF/S schemas and instances. Additionally, it offers higher abstractions to SW developers such as *Namespaces* (viewed as a container of classes and properties defined in a schema) and Graphspaces [11] (viewed as a container of edges relating objects through various kinds of properties). In a nutshell, the *subject*, *predicate* or *object* of a triple in SWKM are Java objects whose state and type information is determined by the triples of an RDF/S model. Thus, SWKM supports seamlessly both triple and object views allowing to construct the latter from the former in a transparent to the user way.

2.5.2.1 Data Structures

As we can see in Figure 2.8, SWKM employs two additional abstractions as containers of the RDF/S model information: *NameSpaces* (NS) and *GraphSpaces* (GS). In this context, an RDF/S model consists of NS and GS collections. A NS gathers all the class and property nodes along with the respective triples that are defined in an RDFS schema associated to the model while a GS keeps track of all edges relating nodes through various

[21]http://athena.ics.forth.gr:9090/SWKM
[22]http://www.kp-lab.org//
[23]http://www.casparpreserves.eu/

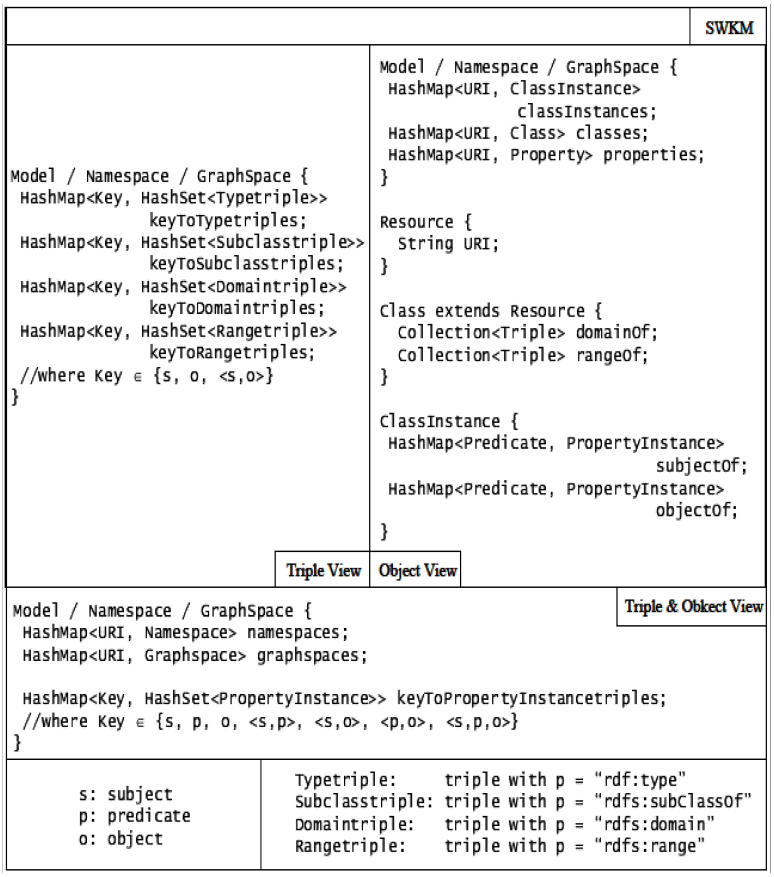


Figure 2.8: SWKM Data Structures

kinds of properties. In SWKM, the *subject*, *predicate* or *object* of a triple are Java objects whose state and type information is determined by the triples of an RDF/S model. In turn a triple is implemented as a Java object holding references to the participating nodes. At the NS level four Maps named `keyToTypetriples`, `keyToSubclasstriples`, `keyToDomaintriples`, `keyToRangetriples` are used to store schema related triples split by their predicate, i.e. using a different Map for each possible property of the RDF/S. The Maps used are actually MultiHashMaps[24], which store the value of the HashMap's $< key, value >$ pair in another HashMap. Moreover these Maps are indexed by *subject*, *object* and by *subject and object* (meaning that actually three `HashMaps` are kept per RDF/S property). Class and property instance related information are kept at the model level in the `classInstances` and `propertyInstances` `HashMaps` respectively. Actually, for the `propertyInstances`, six Maps are kept since *subject* or *predicate* or *object* and all their possible combinations are used as multikeys for faster search. Triples are

[24] commons.apache.org/collections/

treated as objects, subclasses of the root class *Triple*, allowing for programmatic access to them through common interfaces and methods. Class *Triples* (Figure 2.9) instantiations carry also information about the NS and the GS they belong to. Finally, separating the triples and providing object representations for each RDF/S property type make easier the development of sophisticated algorithms for RDFS schema evolution and comparison.

Nodes are also stored as objects either at the NS (when they are schema related) or at the model (when they are instance related) level. Two `HashMaps`, `classes` and `properties`, are used to store the objects representing the classes and the properties of each NS and kept at the NS level. On the other hand, one additional `HashMap` is used to store resources (`classInstances`) at the model level. Nodes carry within available information that is computed at run time, e.g. a node of type *Class* carries information about its subclasses and its superclasses (in both cases immediate and all), a node of type *Class Instance* carries information about its classification, etc; this information is computed from the corresponding triples once. Finally, NS and GS themselves hold dependency information, i.e. to which NS or NS and GS respectively they might depend on.

2.5.2.2 Access Methods

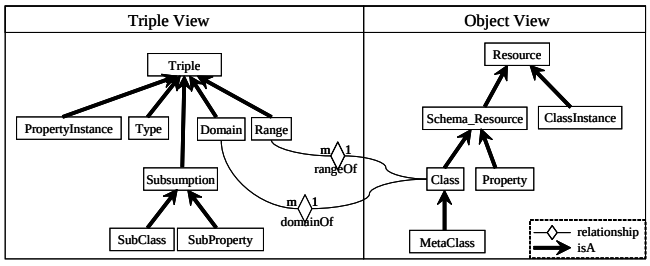


Figure 2.9: RDF/S Triple vs Object View in SWKM

Using the SWKM API the developer has the possibility to choose between the *triple-based* and the *object-based* views to access the same information represented in an RDF/S model. In the triple-based view, similarly to all previously mentioned MMRMS, a retrieval method for all triples of the model, as well as one for all triples with some *subject*, *predicate* and/or *object* are offered. To retrieve all triples of a model, all values of the previously described Maps are returned. For this operation, all the Maps hashed on the subjects are

used. When searching for triples by subject, predicate and/or object, the respective Hash Maps are used. A rich set of variations of these methods is also provided. For instance, a user can choose not to include in the results of the "find all triples" method the triples contained in the default RDF/S namespace or one can retrieve triples searching for them either by their (string) URI or by the object representation of the node. As already stated, we can also retrieve triples that belong to a specific NS or GS.

In the object-based view, the objects carry information firstly through their types (thus they are not just nodes as in the previous MMRMS). Additional information is computed, like *subClassOf*, *superClassOf* and *subPropertyOf*, *superPorpertyOf* information for classes and properties respectively, which then can be retrieved directly from the object's members in constant time by returning an `Iterator` on the structure(s) that holds it. The same way information on which triples the object serves as domain, range or predicate is also kept. The object based view offers an objectification of the RDF information in a consistent way. Additionally in any step this can be combined with information retrieved through the triple based access methods. Actually, one can seamlessly move between the two views since the objects mainly serve as triple nodes and can be reached through the triples and the triples can be found based on the Hash Map keys which are build based on the objects (Figure 2). This combined capability of representing and manipulating RDF information in main memory both as triples and as full blown objects is to the best of our knowledge unique in SWKM.

Chapter 3

Benchmarking the Main Memory representations

3.1 Introduction

The development of end-user applications and middleware services for the Semantic Web (SW) requires suitable data abstractions and structures for engineers. This is the objective of the various SW Programming Environments developed during the last years [9]. A core issue for understanding their functionality and performance is the representation in Main Memory of RDF/S[1] (or OWL[2]) schemas and instances. Although most of the current Main Memory RDF/S Management Systems (MMRMS) are developed in some object-oriented language, they do not abstract at the same level of detail the complexity of the RDF/S data model (i.e. *triple-based* vs. *object-based* views of RDF/S). Furthermore, they do not offer the same programming facilities (e.g. access paths to RDF/S schemas and data) while they do not rely on the same data structures (e.g. HashMaps vs. ArrayLists, etc.) for implementing them. Despite some informal surveys [4, 31], an extensive evaluation of MMRMS functionality and performance is still missing.

More precisely, we compare experimentally the memory requirements and access response times of the data structures employed internally by each MMRMS. In this respect,

[1] www.w3.org/RDF
[2] www.w3.org/2004/OWL

we rely on the widely used LUBM Benchmark [22] to generate synthetic data sets as well as a home-made query workload covering to our opinion a great part of the access functionality requested by existing SW middleware services and end-user applications.

The main conclusion drawn from our experiments is that Sesame, Jena and JRDF best suit to SW applications that stick with the URI-based triple view of an RDF/S model, with Sesame being the best in terms of memory requirements and performance. On the other hand, SWKM best suits to those applications that either need to manage RDF/S schemas and data in the context of a name- or graph-space or demand an object view on the schema and data.

3.2 RDF/S Benchmark

3.2.1 RDFS version of the LUBM schema

In this Section we present the Benchmark that we used for evaluating experimentally the performances of MMRMSs. It is based on LUBM [22], originally proposed for benchmarking the performances of OWL (or DAML [21]) repositories which comprises:

- an OWL schema, called *univ-bench* (*ub*), for describing universities, departments and related activities.
- a synthetic generator of instances of the *ub* schema
- a set of fourteen queries expressed in SPARQL[3].

More precisely, we rely on an RDFS version of the *ub* schema as well as the LUBM synthetic instance generator. As it will be presented in the sequel, we consider a new set of SPARQL queries that are more suited to test the efficiency of the Main Memory representation of RDF/S schema and instances supported in MMRMSs. In particular we take into account a) the RDF/S abstractions offered by each system (i.e. triple vs object-based view) and b) the available access paths to the manipulated RDF/S schema and instances (i.e. find the triples of an RDF/S model vs. the objects of a given namespace). It should be stressed that in our experimental study we are interested in the average

[3]http://www.w3.org/TR/rdf-sparql-query/

#	Inf. Need	SPARQL expression
Q1	classes of the model	SELECT ?x WHERE{?x *rdf:type rdfs:Class*}
Q2	properties of the model	SELECT ?x WHERE{?x *rdf:type rdf:Property*}
Q3	subclasses of class c	SELECT ?x WHERE{?x *rdfs:subClassOf* c}
Q4	superclasses of class c	SELECT ?x WHERE{c *rdfs:subClassOf* ?x}
Q5	subproperties of property p	SELECT ?x WHERE{?x *rdfs:subPropertyOf* p}
Q6	superproperties of property p	SELECT ?x WHERE{p *rdfs:subPropertyOf* ?x}
Q7	Is c_1 subclass of c_2 ?	ASK {c_1 *rdfs:subClassOf* c_2}
Q8	all triples of the model	SELECT ?x, ?y, ?z WHERE {?x ?y ?z}
Q9	triples with subject c	SELECT c, ?y, ?z WHERE {c ?y ?z}
Q10	triples with predicate c	SELECT ?x, c, ?z WHERE {?x c ?z}
Q11	triples with object c	SELECT ?x, ?y, c WHERE {?x ?y c}
Q12	triples with subject c_1 and predicate c_2	SELECT c_1, c_2, ?z WHERE {c_1 c_2 ?z}
Q13	triples with predicate c_1 and object c_2	SELECT ?x, c_1, c_2 WHERE {?x c_1 c_2}
Q14	triples with subject c_1 and object c_2	SELECT c_1, ?y, c_2 WHERE {c_1 ?y c_2}
Q15	triples with subject c_1, predicate c_2 and object c_3	SELECT c_1, c_2, c_3 WHERE {c_1 c_2 c_3}
Q16	all class instances of a model	SELECT ?x WHERE{?x *rdf:type* ?y . ?y *rdf:type rdfs:Class*}
Q17	all property instances of a model	SELECT ?x, ?p, ?y WHERE{?x ?p ?y . ?x *rdf:type* ?z . ?z *rdf:type rdfs:Class* . ?y *rdf:type* ?w . {{ ?w *rdf:type rdfs:Class*} UNION {?w *rdf:type rdfs:Literal*}} }
Q18	class instances of c class	SELECT ?x WHERE{?x *rdf:type* c}
Q19	property instances of p property	SELECT ?x, p, ?y WHERE{?x p ?y . ?x *rdf:type* ?z . ?z *rdf:type rdfs:Class* . ?y *rdf:type* ?w . {{?w *rdf:type rdfs:Class*} UNION {?w *rdf:type rdfs:Literal*}}}
Q20	property instances connecting c_1 and c_2	SELECT c_1, ?x, c_2 WHERE{c_1 ?x c_2}
Q21 (k=1)	class instances connected with c through a path of length k	SELECT ?x WHERE { {{c ?y ?x} UNION {?x ?y c}} . ?x *rdf:type* ?y . ?y *rdf:type rdfs:Class*}
Q22	all namespaces	SELECT prefix(?x) WHERE {{?x *rdf:type rdfs:Class*} UNION {?x *rdf:type rdf:Property*}}
Q23	triples of a given namespace (ub)	SELECT ?x WHERE{?x ?y ?z . FILTER regex(?x, "*ub*")}
Q24	classes of a given namespace (ub)	SELECT ?x WHERE{?x *rdf:type rdfs:Class* . FILTER regex(?x, "*ub*")}
Q25	properties of a given namespace (ub)	SELECT ?x WHERE{?x *rdf:type rdf:Property* . FILTER regex(?x, "*ub*")}
Q26	triples of a given graphspace (gs)	SELECT ?x FROM NAMED ⟨gs⟩ WHERE {?x ?y ?z}
Q27	classes instances of a graphspace (gs)	SELECT ?x FROM NAMED ⟨gs⟩ WHERE {?x *rdf:type* ?y . ?y *rdf:type rdfs:Class*}
Q28	property instances of a graphspace (gs)	SELECT ?x, ?p, ?y FROM NAMED ⟨gs⟩ WHERE {?x ?p ?y . ?x *rdf:type* ?z . ?z *rdf:type rdfs:Class* . ?y *rdf:type* ?w . {{?w *rdf:type rdfs:Class*} UNION {?w *rdf:type rdfs:Literal*}}}

Table 3.1: Benchmark Queries

performance of MMRMSs access methods over all classes and properties of the *ub* schema (vs. specific one as in the original LUBM queries).

The RDFS version of the *ub* schema we use[4] is depicted in Figure 3.1. To simplify the presentation we illustrate only the main classes (from a total of 43 classes) and properties (from a total of 32 properties). For instance, property *degreeFrom* represents the fact that a *Person* has graduated from a *University*. Moreover, every *GraduateStudent* is also a *Person*, while every *Department* is also an *Organization*.

Besides syntax transformations (from OWL to RDFS clas-ses and properties) we have rewritten the *owl:intersectionOf* constructs. Given that the intersection of class

[4]athena.ics.forth.gr:9090/RDF/lubm/schema/univ-bench.rdfs

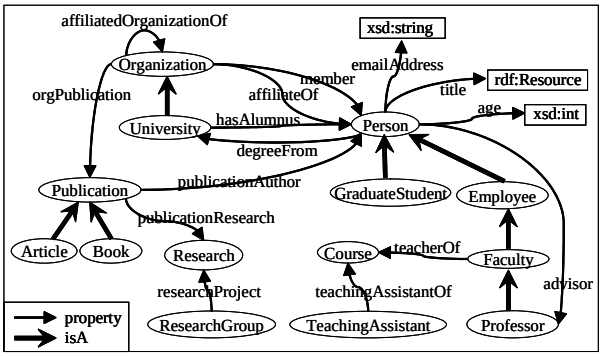


Figure 3.1: Univ-Bench Schema

ub#Person with an anonymous class (derived from an *owl:Restriction*) is always subsumed by *ub#Per-son*, we replaced the *owl:inter-sectionOf* statements with the corresponding class *ub#Person*. Finally, we omitted all *owl:inverseOf* constructs.

3.2.2 Instance Generation

The LUBM synthetic instance generator takes as input the number of universities we want to produce and outputs the description of a number of departments per university. Information regarding each department (i.e. instances of the *ub* schema) is stored in a different RDF/XML file. For example, it generates instances of *Student*s and *Professor*s per university department. For our experiments, we generated incrementally a data set of 10 departments of a single university. More precisely, $Dept(1)$ comprises the data set of the first department, $Dept(2)$ comprises the datasets of both first two departments and so forth. The size, in terms of triples, of the generated datasets ranges between 9,000 and 68,000 (while the schema triples are 226).

3.2.3 Query Workload

Table 3.1 illustrates the 28 queries of our benchmark formulated in SPARQL. Since all queries employ the *ub* schema we omit in queries the corresponding namespace prefix. It should be stressed that all 28 queries are evaluated as Java programs (and not using a SPARQL engine) using the access methods supported by each MMRMS. The queries are organized in four representative groups of the access functionality supported by all systems of our benchmark:

- Schema Queries on a Model (Q1-Q7).

  This group comprises queries asking for the classes (or properties) of the model (as Java objects) or the subsumption relationships between them.

- Triple Queries on a Model (Q8-Q15).

  These queries retrieve the triples of a model using filtering conditions on their subject, predicate, object or any combination of them. Symbols c, c_1, c_2, c_3 in Q8-Q15 denote resource URIs. We should notice that for Q3-Q6 we compute all the subclasses (resp. subproperties) of a class (resp. property) and not only the direct ones. In a similar fashion, Q7 asks if c_1 is a subclass (direct or indirect) of c_2.

- Instance Queries on the Model (Q16-Q21)

  This group comprises queries returning class / property instances (as Java objects). The pattern {?*y rdf:type rdfs:Class*} is needed in order to filter out from the result of Q16 the instances of metaclass like *rdfs:Class*. In similar fashion, we consider in the result of Q17 only the triples whose subject is a class instance (i.e. {?*z rdf:type rdfs:Class*}) and the object is either a class instance or a literal (i.e. {?*w rdf:type rdfs:Class*} $UNION$
  {?*w rdf:type rdfs:Literal*}}). We should stress, that for Q8-Q15, we consider as instances of a class c, the c instances union the instances of every subclass (direct or indirect) of c.

- Triple and Schema Queries on the Namespace or Graph-space (Q22-Q28).

  To formulate Q22 we employed a function *prefix* for obtaining the namespace of the given class or property, not actually supported by SPARQL.

We should notice that we evaluate every query for all possible input values (e.g. class or property of the *ub* schema) and compute the average. For instance, in the case of Q3, that retrieves all the subclasses of c, we assume as c each of the 43 *ub* schema classes. In this manner, we do not need to further investigate the effect of the specific distributions that LUBM datasets exhibit.

3.3 Experimental Evaluation

Experiments were carried out on a Windows XP computer with Intel Pentium IV processor, at 3.0GHz and 1GB main memory. Sun's Java SE 6, with 768MB Java Heap Size was used. As far as the tested MMRMS are concerned, the last stable version of each of them was used (i.e. Sesame 1.2.7, Jena 2.5.3, JRDF 0.5.0 and SWKM 1.0.)

3.3.1 Load Time & Memory Requirements

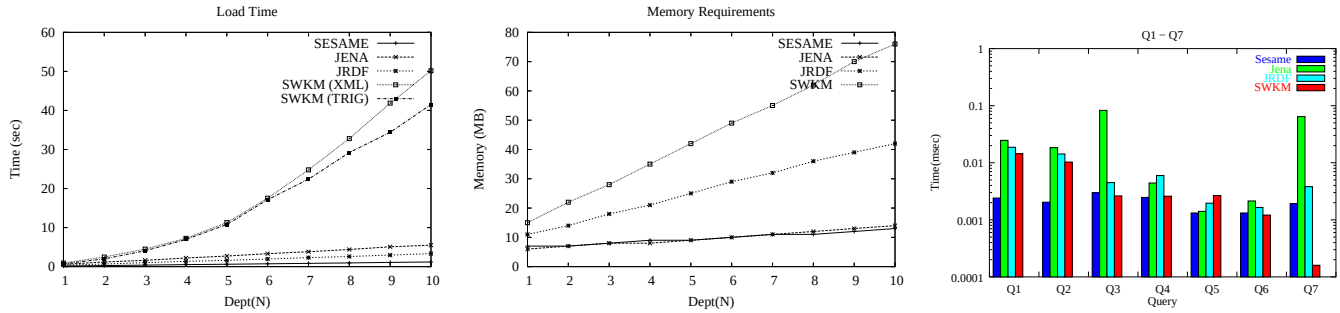


Figure 3.2: Load Time (left) / Memory Requirements (center) / Q1-Q7 (right)

The left (resp. center) part of Figure 3.2 illustrates for each system the time (resp. the memory requirements) to load in Main Memory. Loading time depends on the performances of the employed parsers to analyze the RDF/S input in a specific syntax (XML, TriG, N3) as well as the number and structure of Java objects that need to be initialized. Thus, MMRMS with high memory requirements (as SWKM) have also increased time for loading. As we will see in the following subsections, this is the price to pay in order to provide multiple access paths to RDF/S schemas and instances. The two ends of this functionality/performance tradeoff are Sesame and SWKM. More precisely, SWKM consumes 2-6 times (for Dept(1)-Dept(10) respectively) more memory than Sesame while its loading time is 6-42 times slower (for Dept(1)-Dept(10) respectively) than Sesame. Note also that Sesame performance gains are due to the SAX parser compared to less efficient RDF/XML parsers employed by the other three systems. As we can see in Figure 3.2 while a $55-15\%$ (for Dept(1)-Dept(10) respectively) of the loading time in SWKM is devoted to RDF/XML parsing, it drops to $24-3\%$ when SWKM TriG parser is used. The rest of SWKM loading time is essentially devoted to the construction of the numerous

indices presented in Section 2.5.2.1.

Among the rest, JRDF holds more indices than Sesame and Jena, and thus it requires approximately 3 times more memory, while its loading time is higher than Sesame but lower than Jena. That delay of Jena is due to the invariants of the internal `HashMap` implementation, that cause a decrease of performance for both adding and searching (put and get) methods. Sesame, instead, uses Java `ArrayList`s that offer very fast performance when adding triples. Furthermore, Sesame relies on standard Java `HashMap`s for nodes, which provides a much faster implementation than the one implemented in Jena.

3.3.2 Query Response Time

3.3.2.1 Schema Queries on Models

Q1-Q7 are schema queries and therefore one would expect that their evaluation would be independent of the size of the instance triples. This holds for Sesame, JRDF and SWKM, but not always for Jena. In particular, the Jena performance for Q3 and Q7 depends on the size of instance descriptions. This is due to the fact that Jena does not allow for triple search by predicate. As a matter of fact, in order to respond to Q3, Jena needs first to retrieve all the triples of the model ($\{x\ y\ c\}$) and then to iterate on the result in order to filter out triples not having a *rdfs:subClassOf* predicate (y). The search space in this context essentially comprises both schema and instance triples where the latter are significantly larger than the former. The same is also true for Q7 for Jena. Therefore, in the case of Q3 and Q7 the right part of Figure 3.2 reports the Jena response time when Dept(5) has been loaded (i.e. an average size of the employed datasets). Additionally, the left and the center part of Figure 3.3 illustrates Jena performance for Q3 and Q7 with respect to all loaded departments.

Sesame outperforms all other systems for Q1 and Q2. When all classes (or properties) of the model need to be retrieved, the `objectTriples` (see also Figure 2.3) list of every node is scanned to find those triples matching the predicate *rdf:type*. Since this list does not contain irrelevant triples, the cost of returning the classes (or properties) is essentially the time to obtain the subjects from the triples. This justifies the performance excellence

of Sesame for Q1 and Q2.

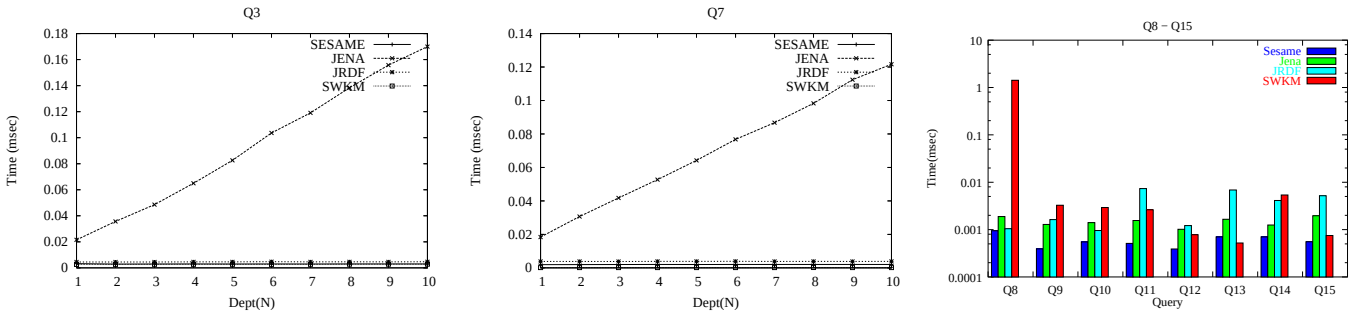


Figure 3.3: Q3 (left) / Q7 (center) / Q8-Q15 (right)

Queries Q3-Q7 require transitive closure computations over the collections of triples in order to retrieve both direct and indirect subsuming classes (or properties). Given that the size of the *ub* schema is small and there are introduced only few subsumption relationships, the effect of these transitive closure computations is not so important in the performance figures of Figure 3.2. The good performance of SWKM for Q7 is due to the encoding of subsumption relationships using an interval based labeling scheme [6] which allows us to avoid costly transitive closures. Then, checking whether a class c_1 is subsumed by c_2 is reduced to a simple interval containment check using two integers representing the start and the end of each interval. Thus, unlike all other systems subsumption check in SWKM takes constant time (2 integer comparisons) irrespective of the schema size and the relative position of classes in the subsumption hierarchy (i.e. direct or indirect subclasses). Of course, the construction of labels is performed during loading (object-based view creation) and it is linear to the number of encoded schema classes (or properties).

3.3.2.2 Triple Queries on Models

One can retrieve from a model all its triples or a subset based on either the subject, the predicate or the object or any combination of the above. The MMRMS under consideration return an iterator over the results. As one can observe in the right part of Figure 3.3 (drawn on log-scale), Sesame features the best overall performance for Q8-Q15. In order to evaluate Q9-Q15 Sesame first searches the `URINodes`, `LiteralNodes` and `BlankNodes` to find the resources given as input. For instance, for Q9, it retrieves the resource c and then returns an `Iterator` over the `subjectTriples` attribute (see also

Figure 2.3).

Instead, Jena searches the `subjectToTriples`, `predicateToTriples`or `objectToTriples` map (see also Figure 2.4) with key the given subject, predicate or object respectively and returns an Iterator over the returned collection.

JRDF's implementation of these queries varies on the type of nodes used as parameters. Two lookup operations are needed for each node (i.e., subject, predicate or object) specified in the query: one in the `NodePool`, to get the id of the node, and one in one of the three (i.e., `spoMap`, `posMap`, `ospMap`) matching indices, using the retrieved id.

As described in Section 2.5.2.1, in SWKM, indices on schema related triples are split by their predicate. So, in order to retrieve triples, the first step is to identify the relevant indices based on the predicate, or use them all if no predicate was specified. The selection of the correct indices and the merging of results found in them creates an overhead that slows SWKM in these types of queries.

3.3.2.3 Instance Queries on Models

Figure 3.4 illustrates the response times for Q16-Q21 (drawn in log-scale). On overall, for this kind of queries, SWKM outperforms all other systems. The second most efficient system is Sesame, while Jena and JRDF follow by far.

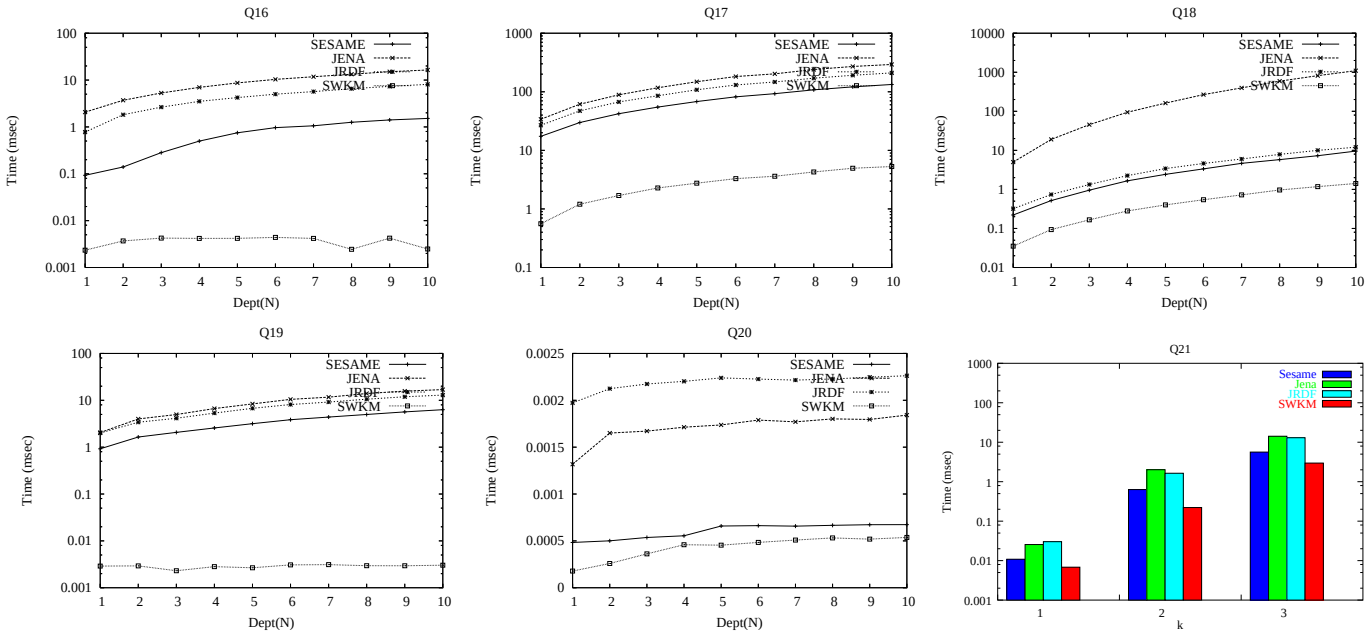


Figure 3.4: Queries 16-21

The performance excellence of SWKM stems from the fact that it relies on different `HashMaps` for different kinds of triples (e.g. with predicate *rdf:type*, *rdfs:subClassOf* etc.). As a consequence, in order to answer Q16, SWKM needs only to return the collection of values of `classInstances` (see Figure 2.8). Instead, Sesame needs first to retrieve two collections, one comprising all the model triples with object the c class and the other comprising all the model triples with predicate *rdf:type*. Then, it iterates over the smaller of the two collections, which in our testbed is the first, to check whether the predicate equals to *rdf:type*. Jena searches triples of the model with object the class c and then iterates over the result to check whether the predicate is *rdf:type*. Finally, JRDF first searches the `posMap` (see Figure 2.7) with key the predicate *rdf:type*. The returned value is a `HashMap` whose key is the object and value the subject of the triple. JRDF iterates over the key set of that `HashMap` and for every search key that is a class it appends the values to the result. In order to find whether a uri is a class or not, JRDF searches the `posMap` with key the predicate *rdf:type*. The returned value is a `HashMap` whose key is the object and value the subject of the triple. JRDF searches that `HashMap` with key the object *rdfs:Class*.

SWKM outperforms Sesame up to 3 orders of magnitude for Q19. In order to retrieve the property instances of the model, SWKM exploits the HashMap with key the property and value the collection of triples with the property instances (i.e. Q19 answer). Instead, the other three systems first retrieve the triples of the given predicate and then iterate over them to check whether the subject is a class instance and the object a class instance or a literal. For Q19 Sesame performance is closer to SWKM (only 27% slower) given that two class instances, i.e. c_1 and c_2 a triple of the form $\{c_1\ ?x\ c_2\}$ is always guaranteed to be a property instance. As a result Sesame, but also Jena and JRDF, do not need to match the additional patterns (i.e., $\{?z$ *rdf:type rdfs:Class*$\}$ and $\{?w$ *rdf:type rdfs:Class*$\}$ $UNION$ $\{?w$ *rdf:type rdfs:Literal*$\}$) of Q19. Note that although Q20 is computationally simpler than Q19, SWKM performance is almost the same since all triples of the corresponding property instances are stored in the specific `HashMap` during loading. Q21 response time is only slightly affected by the size of instance triples. Specifically, all systems search a `HashMap` given the triple subject as key. The effect of the `HashMap` size in the performance

of the `get` method is negligible. As a consequence, we present in the bottom right part of Figure 3.5 the performance of the four systems for Q21 as composite plot. As expected, response times increase as long as k increases, due to the additional patterns that should be considered. However, the relative order of systems remains the same, reaffirming in this manner the qualitative results drawn from Q16-Q20.

3.3.2.4 Triple and Schema Queries on NS and GS

As we can see in Figure 3.5, SWKM performance is orders of magnitude better than other systems for queries Q22-Q25 and therefore the plots are drawn in log scale. SWKM performance gains for queries retrieving triples, classes or properties of a specific namespace stems from the representation of RDFS namespaces as Java objects whose state refer to the contained triples. In addition RDF/S models maintain references to the namespaces by which are composed. For this reason the evaluation by SWKM of Q22 implies only a direct return of the namespace collection and thus is independent of the size of the instance triples. On the other hand, Sesame and Jena only store the namespaces as strings in the nodes of the triples. As a result, the only way to acquire namespace information is to retrieve all the model triples and then, identify the loaded namespaces on the corresponding state of nodes. Consequently, their performance depends on the size of instance triples. Unlike the rest of the systems, JRDF does not hold in any way separate references to namespaces. Thus, the only way to retrieve the namespaces of a model is to directly parse the URI of the contained resources. For these reasons, Sesame (resp. Jena and JRDF) is four (resp. five) orders of magnitude slower than SWKM. Note that Sesame is faster than Jena because, as was described in Section 3.3.2.2, the iteration over all the model triples returned by Sesame is faster than by Jena.

The same qualitative conclusions are drawn from Q23-Q25. SWKM first retrieves the namespace object and then retrieves only the triples stored in it. Instead, the rest of the systems, to answer queries Q23-Q25 need first to retrieve the model triples and then filter them with the given namespace.

Finally, the bottom right part of Figure 3.5 illustrates the performance of SWKM for queries Q26-Q28 retrieving triples, classes or properties of a specific graphspace. The rest

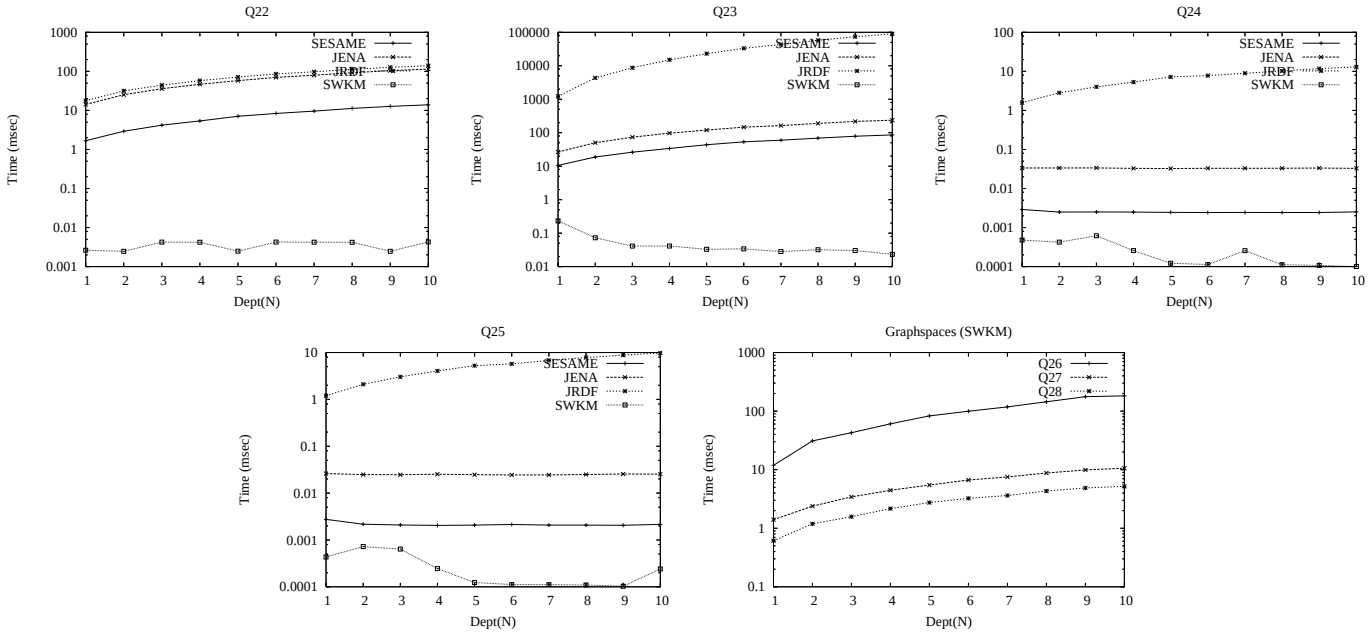


Figure 3.5: Queries 22-28

systems are absent from the plot, since RDF named graphs are not yet supported. As one can observe, SWKM is more efficient when searching for class or property instances rather than for triples of a graphspace.

3.3.3 Summarizing the Results

The basic conclusions drawn from the experimental evaluation of the four MMRMS are directly mapped to the design choices of their creators. SW developers that want to work only on the URI-based triple view of an RDF/S model can rely on Sesame for its performance (tops in Q1 to Q6, right part of Figure 3.2) and low memory requirements (center part of Figure 3.2). If they need to work in the context of individual name- or graphspaces though, they have to rely on SWKM since it exhibits the best performance; moreover SWKM is the only one that supports graphspaces. When the SW developers want to work on top of an object view they should rely on SWKM (Q16 - Q21, Figure 3.3), which is the only one that offers that possibility and exhibits a 2 to 4 times better performance from the second best; this comes with 8 times more memory usage than the other systems. It should also be noted that in contrast to Sesame and SWKM that show their strengths and weaknesses by positioning themselves in first places in various queries,

Jena and JRDF do not take the lead in any category related to performance neither can they compete with Sesame for smaller memory footprints.

3.4 Conclusions

We compared four well known MMRMS with respect to their functionality and performance. The Benchmark we proposed, allowed us to investigate the effect of the design and implementation choices of current MMRMS to their performance. Features and limitations of their programmatic functionality were investigated and their correlation with their efficiency was studied. Additionally, the trade off between the variety of the provided logic views on RDF/S data and the memory requirements was illustrated. The problem of employing the optimal indices that would maximize systems performance on queries assuming both RDF/S triple and object views and minimize as long as possible the memory requirements, deserves further research. The experimental results presented in this work consist the first step towards this direction.

Chapter 4

Semantic Web Knowledge Middleware (SWKM) Services

This section delves into the design, architecture and implementation decisions of the SWKM middleware in particular, as well as its versioning services specification and implementation decisions.

4.1 Introduction

The SWKM platform is a semantic management server-side stack, implemented in *Java* and *C++*. Its purpose is to provide its users scalable middleware services for managing voluminous representations of Semantic Web data (schemata and data expressed in RDF/S).

A major goal of the platform, which influences the design and implementation of almost all layers of the platform is to enable powerful and general declarative querying and update capabilities to the user. We will iterate briefly over this focal point, and then we will enumerate and and specify the SWKM services follow that define the platform's architecture.

The initial database schema (whatever its particular flavor) is created by the *Importer Service*. RDF Schema information is stored into the database through the Importer Service too, as currently RUL can only update RDF data instances, not schema information.

Also, the Importer Service offers bulk uploading of multiple data to the database, for any RDF-relational mapping, instead of having to issue multiple RUL statements to the RQL/RUL interpreter. Currently, PostgreSQL, an Object-Relational DBMS[1], is being used.

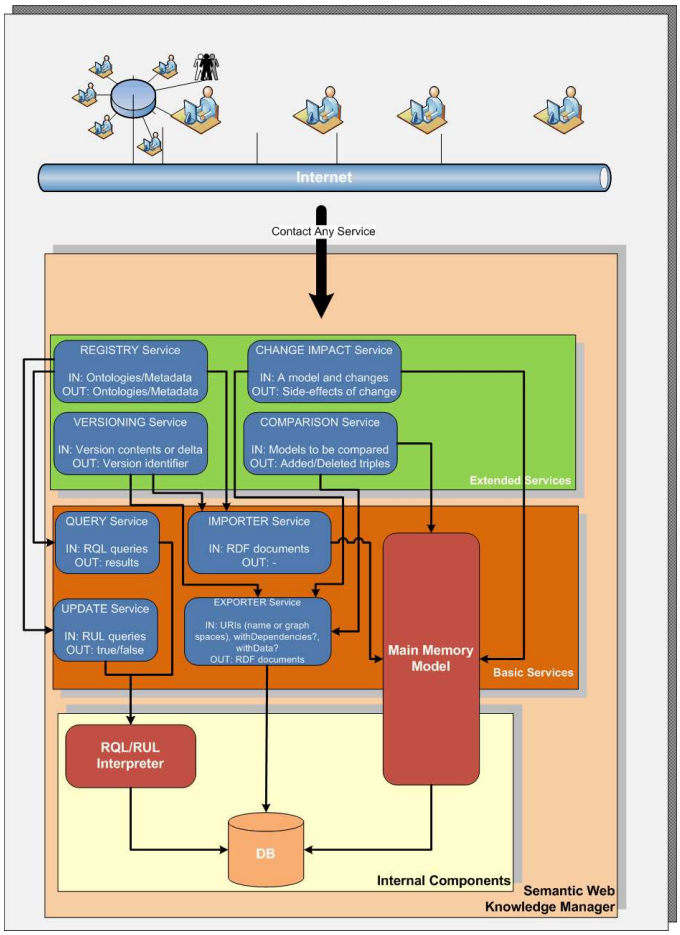


Figure 4.1: An architectural overview of the SWKM services

In Figure 4.1 we see a diagram of the middleware architecture. In Figure 4.2 we see a middleware deployment and the interaction with a client, and an example "learner" application".

[1]http://www.postgresql.org/

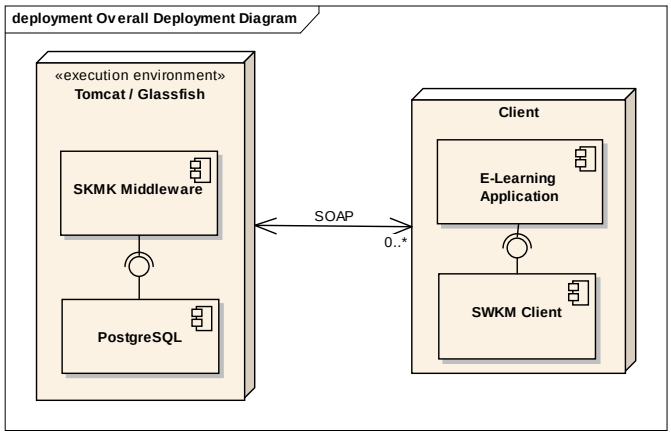


Figure 4.2: Deployment of SWKM middleware and Client interaction

4.2 Overview and Design Choices

For the purpose of giving the platform the maximum interoperability, it was chosen to offer all services as an array of SOAP-enabled[2] Web Services. Here we will provide a detailed reference of the current Web Services architecture, which broadly defines the various ways that the SWKM middleware platform can be integrated with other applications.

First of all, all described signatures and types are based on the ones appearing at the XML (SOAP) layer of the communication stack, and this is an accurate representation of them since the web services of SWKM are based on SOAP over HTTP. For the notation of the types, standard `RELAXNG`[3] is used in this document, in particular its non-XML flavor, as it was deemed much more readable than its equivalent XML Schema counterpart.

A key point of SWKM platform is its (web) statelessness: every user request boils down to a self-contained document that fully describes the request, and no server-side state management is needed. The latter point contributes to the platform's web interface horizontal scalability, that is, scalability gained by adding more web servers. Having no state (per user) in the web server means there is no need to spend time replicating it to every server; each server can interchangeably serve any client. Also, the memory footprint of the platform is lower. On the downside, this means more state moving back and forth between the server and the client, but in our case, this is typically low, and only minimally

[2]http://www.w3.org/TR/soap/
[3]http://www.oasis-open.org/committees/relax-ng/spec-20011203.html

contributes to the networking costs.

A feature that raises the above discussion is the ability of the user to customize database settings; for example, if there are two different databases used as repositories for the SWKM platform, the user can define the one to be targeted by a service call. If the user does not define such settings, the defaults are used. At most one repository can be used by a request, for validity reasons; such constraint are only applied in a single repository-wide manner. These settings can be defined as following (in `RELAXNG`, as mentioned above):

```
element dbSettingsElement {
    element dbName { text }?,
    element host { text }?,
    element port { xsd:integer }?,
    element protocol { text }?,
    element representation { text }?,
    element username { text }?,
    element password { text }?
}
```

This element can be attached to most services (only to the Registry service it is not applicable - we considered the latter case to not add significant value). It is not advisable to send username and password credentials over the network though, from a security point of view. A more recommended approach would be to simply send another database name (possibly also database network address) to be used, which can still be accessed with the default username and password (stored in the server), or even provide a different username which has the same password as the default user (any missing value is supplanted by the defaults, even the password; if the resulting settings are illegal, the database call will fail).

Another data type that appears in several services and would make sense to describe here is the *Delta*. A Delta is conceptually two sets of triples; one set of "added" triples and one of "deleted" triples. Where are these triples added or deleted from (if at all) is service-specific. Each triple set is actually defined by a string following the TriG[4] syntax. This is

[4]http://sites.wiwiss.fu-berlin.de/suhl/bizer/TriG/

less verbose than having a nested XML element for each triple. The Delta type definition:

```
element deltaElement {
    element added{ text },
    element deleted { text }
}
```

All provided services are *synchronous*, or RPC-styled (Remote Procedure Call). That is, a client sends a request and waits for the response (which can be empty, or a failure description). So, each service method can be described by two XML types: its input, and its output (since these are matched one to one). The notation will be that the input element will have the same name as the service method; the output element will have the same name with an appended "Response" to it.

With this common understanding, we can continue with the services themselves.

4.3 SWKM Services

4.3.1 Importer Service

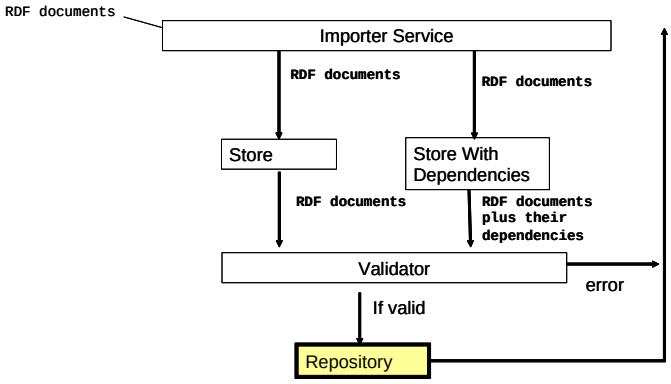


Figure 4.3: Importer service

The Import Service is responsible for loading the contents of a valid and well formed name or graph space (along with their version ID). It is also responsible for creating the necessary database constructs (tables, relationships, indices) that allow for efficient retrieval and manipulation by the RQL/RUL interpreter.

The service uses the main memory representation described earlier. It firstly load the RDF content into the main memory, check validation constraints on it, and if it is deemed valid, commits it to the repository The RDF model is afterwards unloaded from the main memory.

This service consists of two programmatic interfaces, *Store* and *Store with dependencies*.

4.3.1.1 A note in resolving dependencies for storing

There are two processes of resolving dependencies: *passive resolution* and *eager resolution*. The following actions are executed in strict order:

- Try to find the needed RDF space in the supplied parameters of the request
- If failed, try to find it in the underlying storage
- If failed, try to download it from the network (use its URI as the address)
- If not found, raise an error

Passive resolution is used in plain store operation. The following actions are executed in strict order:

- Try to find the needed RDF space in the supplied parameters of the request
- If failed, try to find it in the underlying storage
- If not found, raise an error

Next, we define the *Store* operation, which uses passive resolution, and *Store with dependencies* operation, which uses eager resolution.

The signature of the `Store` operation is:

```
element store {
    element URI { text }*,
    element document { text }*,
    element format { text },
    element dbSettings { dbSettingsElement }?
}
element storeResponse { }
```

It imports in the underlying storage the specified (by their URIs) namespaces and/or graphspaces. Each URI element corresponds to a document element, where as a document we define the contents of the corresponding namespace or graphspace. The format parameter must have the value of either "TRIG" or "RDF/XML", the two supported RDF serialization formats. All documents are strings in the format specified by the format parameter. Henceforth, the corresponding URI element of an element in 'documents' will be referred to as "the URI of the document".

If a document defines an RDF namespace, the URI of that namespace will be the URI of the document. In case of TRIG format, the document may define one or more RDF graphspaces, which are locally identified by a graph id. Each graphspace's URI will be created by concatenating the document's URI and the graph id. These resulting URIs can be used to retrieve back the stored RDF spaces.

In case of TRIG format, at most one namespace is allowed per document (that namespace's URI will be the respective URI specified for the document), but there is no limitation in the number of graphspaces. In case of RDF/XML format, each document can have at most one namespace and at most one graphspace (possibly both). The namespace will get the URI of the document. The graphspace will be unnamed, i.e. there will be no way to fetch it through the Exporter service, but its contents will still be accessible through the Query service. If the intention is to store a graphspace, TRIG format must be used.

Preconditions:

- The array of URIs has exactly the same number of elements as the array of data.
- Format is either "TRIG" or "RDF/XML"

- If an RDF space declares its own URI (in a syntax-specific way), it must be the same as the respective entry in the array of URIs

After the successful execution of the operation, the underlying storage will contain all supplied RDF spaces, identified by their respective URIs (see above).

If the underlying storage already contains a namespace or graphspace identified by a URI that this operation was about to create, the pre-existing RDF space is kept and used and cannot be superseded by any RDF space with the same URI. Each RDF space may depend on other RDF spaces. This operation resolves dependencies *passively* (as explained in a later section).

Validation of all resolved namespace and/or graphspaces will always take place before actual importing. If validation fails, an error will be raised, and the operation will be cancelled. It is also noted that the operation semantics are all-or-nothing, i.e. in case of a failure (for example, due to corrupted data, or a validation failure), no partial storing will take place; either all provided namespaces and graphspaces will be stored in the underlying storage, or none at all.

The signature of the `Store with dependencies` operation is:

```
element storeWithDependencies {
    element URI { text }*,
    element document { text }*,
    element format { text },
    element dbSettings { dbSettingsElement }?
}
element storeWithDependenciesResponse { }
```

This is mostly similar to the above operation, but differs in the way interdependencies between RDF name or graph spaces are handled.

It imports in the underlying storage the specified namespaces and/or graphspaces, along with their dependencies, as explained later. This operation resolves dependencies *eagerly*.

The semantics of the parameters are identical to the ones in the *Store* operation, as well as its preconditions.

After the successful execution of the operation, the underlying storage will contain all supplied RDF spaces, identified by their respective URIs (see above), and all their dependencies.

4.3.2 Exporter Service

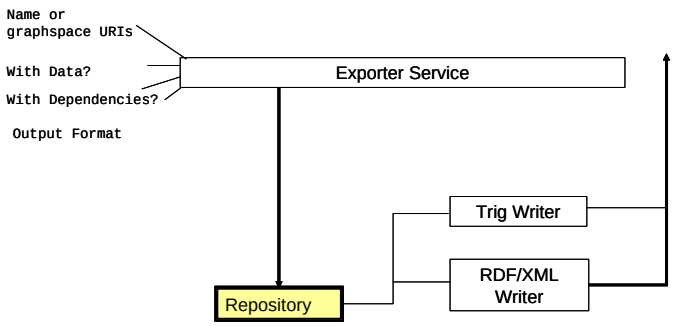


Figure 4.4: Exporter service

The Exporter Service is responsible for dumping into a byte sequence (in RDF/XML serialization or TRIG triple-based formats) the contents of the name or graph spaces given as input. The user of the service needs only to specify which one needs to be exported.

The programmatic signature of this operation is:

```
element fetch {
    element nameOrGraphspaceURI { text }*,
    element format { text },
    element withDependencies { xsd:boolean },
    element withData { xsd:boolean},
    element dbSettings { dbSettingsElement }?
}
element fetchResponse {
    element nameOrGraphspaceURI { text }*,
    element document { text }*
}
```

Preconditions:

- Format is either "TRIG" or "RDF/XML"

- requested URIs exist in the underlying storage

The RDF contents of the requested name or graph spaces are returned in the appropriate format. If *dependencies* is *true*, all transitively name or graphspaces dependent to any returned space are also returned. If *data* is true, all data instances of any schema information returned are returned too (i.e., if a requested namespace contains a class, its instances, whatever the graphspace that contains them, will also be returned).

4.3.3 Query Service

The Query Service is responsible for executing RQL queries. The service will return its results in an RDF/XML or Trig serialization as a bag of resources. The query results can contain both schema and data information from one or several name and graph spaces.

There are two flavors of this operation, *Query* and *QueryMultiple*:

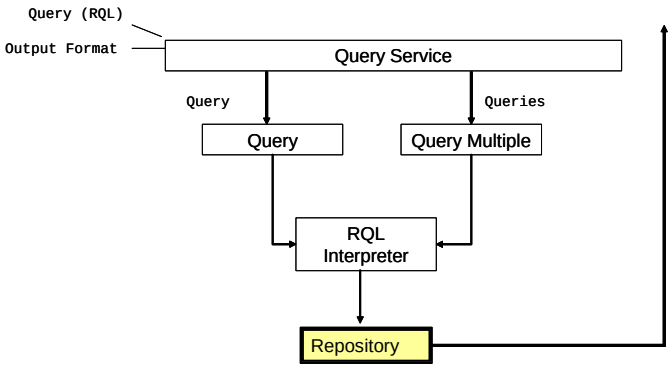


Figure 4.5: Query service

element query {	element queryMultiple {
element RQL { text },	element RQL { text }*,
element format { text },	element format { text },
element dbSettings { dbSettingsElement }?	element dbSettings { dbSettingsElement }?
}	}
element queryResponse {	element queryMultipleResponse {
element result { text }	element result { text }*
}	}

QueryMultiple evaluates multiple queries concurrently, for performance reasons. The Query Service relies on the RQL Interpreter which is used for both parsing and executing the query at hand, as well as on its multithreading capabilities for supporting concurrency.

4.3.4 Update Service

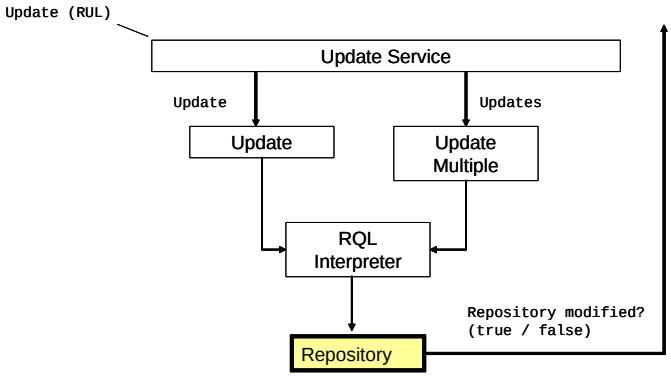


Figure 4.6: Update service

The Update Service is responsible for executing RUL updates involving one or several name or graph spaces. Updating includes construction, modification and deletion of objects in the repository and returns a Boolean value "true" or "false" for successful (commit) or unsuccessful (abort) execution.

As in the Query service, two flavors of this operation are provided, *Update* and *UpdateMultiple*:

element update {	element updateMultiple {
element RUL { text },	element RUL { text }*,
element dbSettings { dbSettingsElement }?	element dbSettings { dbSettingsElement }?
}	}
element updateResponse {	element updateMultipleResponse {
xsd:boolean	element result { xsd:boolean }*
}	}

In the case of the multiple updates signature, the supplied updates run in parallel but they should all finish before the results are returned to the client. Note that if the updates are mutually dependent, the results of this operation are unspecified, as no guarantee of order of update execution can be made.

4.3.5 Comparison Service

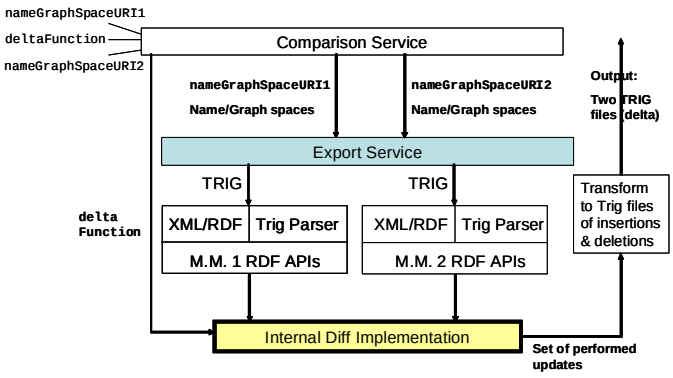


Figure 4.7: Comparison service

The Comparison Service is responsible for comparing two collections of name or graph

spaces already stored in the repository and compute their delta in an appropriate form. The result of the comparison is a "delta" (or "diff") describing the differences between the two collections of name or graph spaces, i.e., the change(s) that should be applied upon the first in order to get to the second. The intended use of the service is the comparison of two different versions of the same name or graph space to identify their differences; comparing unrelated name or graph spaces (i.e., name or graph spaces which are not different versions of the same name or graph space) would give results which have no intuitive meaning.

Note that the problem of comparing two name or graph spaces is very different from the problem of comparing the source files (e.g., TRIG files) which describe them. This is true because (a) a name (or graph) space carries semantics, as well as implicit knowledge which is not part of the source file (i.e., the particular serialization format of some RDF content); (b) there are alternative ways to describe syntactically the same construct (triple) in a name or graph space (for instance, in most cases, order of appearance does not matter), which could result to erroneous differences if resorting to a source file comparison method; and (c) source files may contain irrelevant information, e.g., comments, which should be ignored during the comparison.

The programmatic interface to this service is:

```
element diff {
    element URI1 { text }*,
    element URI2 { text }*,
    element deltaFunction { text },
    element dbSettings { dbSettingsElement }?
}
element diffResponse {
    element delta { deltaElement }
}
```

As shown in Figure 4.7, the Comparison Service exposes a single service which is used

to compare two collections of name or graph spaces and return their delta (diff) according to the selected delta function.

The input of the method is the two collections of the name or graph spaces to be compared, as well as a parameter indicating the mode of the comparison (delta function). These two collections are passed using the nameGraphSpaceURI1[] and nameGraphSpaceURI2[] parameters. Each such parameter is an array of strings, each string containing the URI of a name or graph space (so each of nameGraphSpaceURI1[] and nameGraphSpaceURI2[] represents a collection of name or graph spaces). It should be emphasized that the comparison is not performed upon the name and graph spaces in the input only, but also upon the name and graph spaces that they depend on. In other words, the compared conceptualizations occur by taking the union of the triples in the URIs indicated by nameGraphSpaceURI1[] (and nameGraphSpaceURI2[]) plus the triples in the name or graph spaces that the input name or graph spaces depend on.

The deltaFunction parameter indicates the type of the delta function to be used in the comparison. The semantics of this are described in [50].

The output of the above operation is a pair of strings representing the delta of the two models. In particular, the first string of the pair represents the RDF triples that exist in the second model but don't exist in the first, whereas the second represents the triples that exist in the first but not in the second. This way, the delta can be viewed as an update request (see also the Change Impact Service below), which, when applied to the first model, will (should) result to the second; under this viewpoint, the first string of the output can be viewed as the added triples, while the second can be viewed as the deleted triples. Both strings should encode those triples in TRIG format.

In Figure 4.8 we see the interaction of the Comparison service with Exporter and the Main Memory Model.

4.3.6 Change Impact Service

The Change Service is responsible for determining the changes that should occur on a name or graph space in response to a change request. Given the change request, the change service attempts to apply it to the target name or graph space; in several cases,

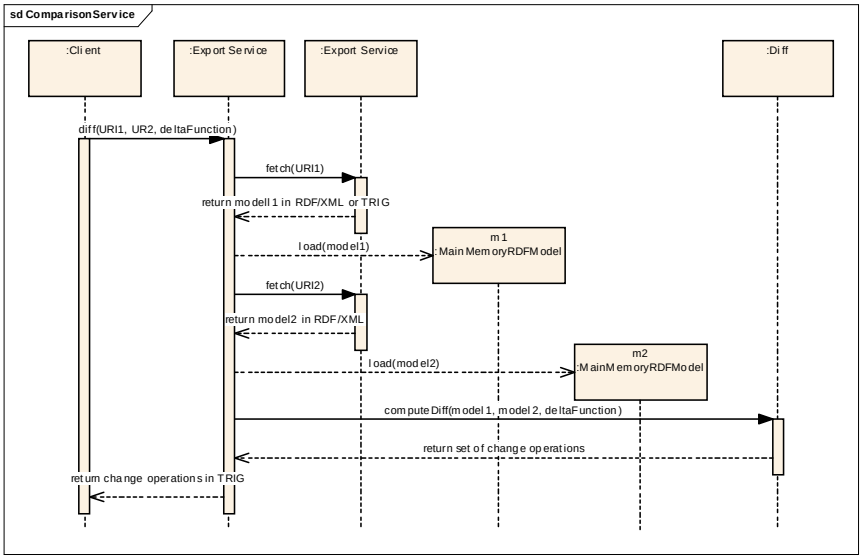


Figure 4.8: Comparison service interaction with Exporter and Main Memory Model

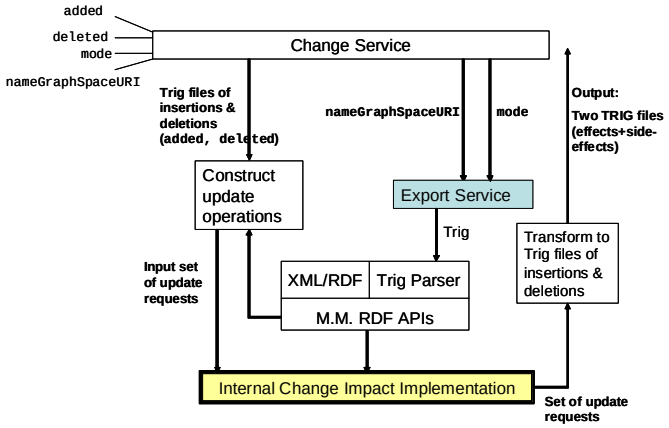


Figure 4.9: Change Impact service

the naive application of a change request upon a name or graph space could potentially result to an RDF KB that is meaningless, invalid or does not obey the RDF formation rules [34].

The programmatic interface of the service is:

```
element changeImpact {
    element delta { deltaElement },
    element nameOrGraphspaceURI { text }*,
    element namespaceClosureMode { text },
    element dbSettings { dbSettingsElement }?
}
element changeImpactResponse {
    element delta { deltaElement }
}
```

The input to this service is an RDF knowledge base and the update request. The RDF knowledge base is specified using any, arbitrarily large, collection of name and/or graph spaces. The change request could affect any of the triples in this collection. However, the side-effects could potentially affect triples in other, depended or depending name or graph spaces; as a result, in order for the change request to be processed in a correct way, all the depended and depending name and graph spaces should be taken into account. Therefore, the RDF knowledge base in this case is the union of all the triples that appear in all the name or graph spaces that are directly or indirectly depending (or are depended) on the given ones.

Having said that, the caller of the service is given the option to restrict changes and side-effects to happen in the given collection of name or graph spaces, plus, of course, those name or graph spaces that the members of this collection depend on; it should be clear that this option may not give the best possible results, as certain side-effects may not be computed. In this case, the RDF knowledge base consists only of the union of all triples that appear in the given name and graph spaces plus those that they depend on.

The update request is specified using the string parameters *added* and *deleted*, representing the set of triples that should be added and deleted respectively from the RDF knowledge base (i.e., the original update request). The triples are encoded using TRIG syntax. The added and deleted triples are combined with the parsed output of the Export Service in order to determine the types of update operations that need to be executed upon the RDF knowledge base and are ultimately fed, along with the RDF knowledge base

that was produced by the parsed output of the Export Service, to the Internal Change Impact Implementation to produce the output. A related restriction is that all the schema resources (classes, properties) that are used inside the *added* and *deleted* parameters (i.e., all the schema resources that appear in the update request) should have the same URI (including version ID - see the versioning service below) as (one of) the URI(s) of the input describing the RDF knowledge base (i.e., one of the URIs in the $nameGraphSpaceURI[]$ parameter); in a different case, an error is reported by the service.

The output of the service is a set of primitive update operations (i.e., another update request) that captures all the effects and side-effects of the original change request upon the target KB. In the example of Figure 2, the output would contain the deletion of B (direct effect), the deletion of the two IsAs (side-effect) and the explicit addition of the previously implicit IsA (side-effect). These effects and side-effects are returned to the caller, in order to be visualized and either accepted or rejected. If the updates are accepted, the Update Service should be called in order to physically execute the updates upon the RDF knowledge base.

Note that, in many cases, there may be more than one possible outputs (i.e., side-effects) that satisfy the above properties; in such cases, the service will select the action that has the minimal possible impact upon the original RDF KB, without negating its validity. In other words, the result of the change should be "as close as possible" to the original KB, according to the "Principle of Minimal Change" [Gar92].

In Figure 4.10 we see the interaction of the Change Impact service with Exporter and the Main Memory Model.

4.3.7 Registry Service

The role of the Registry Service is to record and manage metadata information about ontologies, schemas or namespaces stored in the knowledge repository. Furthermore, the registry offers the possibility to keep track of the development lifecycle of a schema through the support of storing versions, their metadata and the relationships among them. Both schema and version information follow the Ontology Description Schema that is stored in the knowledge repository and is appropriately instantiated for each schema and

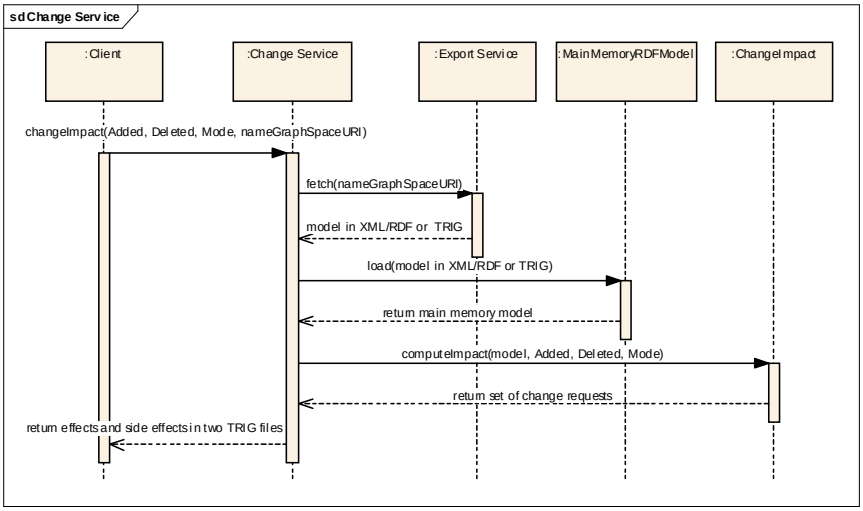


Figure 4.10: Change service interaction with Exporter and Main Memory Model

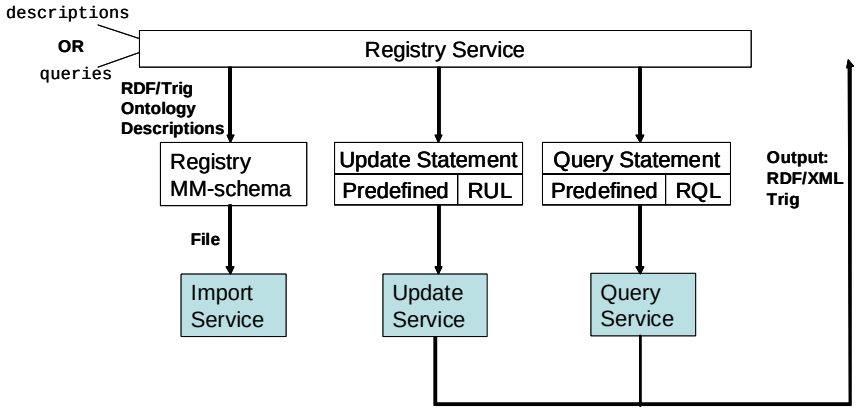


Figure 4.11: Registry service

version stored. Applications using the registry have the possibility to update and retrieve information about the already recorded schemas and their versions by using the available service methods. Note that currently the Registry Service offers support for namespaces only.

A comparison of some of the existing registries is presented in [16]. All of the mentioned systems provide certain searching facilities, but only some of them support editing functions that modify stored information about ontologies and add new ones (such as WebOnto [17], Ontolingua [18] and Ontology Server [2]). Moreover, only a few provide reasoning mechanisms that make it possible to derive a query-answering mechanism such as WebOnto and Ontolingua. Furthermore, only one of the systems, SHOE[24], supports

a versioning mechanism in order to maintain the changes of ontologies in the registry. Our ontology registry provides all of the aforementioned functionalities, since it is using a query/update service based mechanism. Furthermore, it supports versioning in its more general sense as it will be described later.

The Registry Service is implemented as a web service and the different functionalities offered by it are implemented as web methods. However, this web service is not a self-contained module but rather depends on and uses the services provided by the knowledge repository, such as the Import, Update and Query Services. In particular, the Import Service is used to persistently store ontological descriptions, the Update Service is used to update the metadata information on the ontologies (which is stored in the Ontology Registry Schema, which is an ontology itself and described below) and the Query Service is used to query the metadata information stored in the Ontology Registry Schema (for retrieval purposes). The dependencies between the Registry Service and the aforementioned services are schematically depicted in Figure 4.11.

As already mentioned, the Registry Service is using its own ontology, encoded in RDF/S, in order to explicitly describe every other ontology stored in the Knowledge Repository. This ontology is called the Ontology Registry Schema and is described in detail later in this section (see Figure 4.12). For every ontology stored in the Knowledge Repository, an instance of the proper type is created and stored under the Ontology Registry Schema. The Registry is also supporting versioning of schemas by allowing for each ontology the creation of multiple instances of the corresponding class Version and relating these instances to the proper instance of the class Schema. Thus, the metadata stored for each namespace are divided into two main categories regarding to whether their values are changing with each version (e.g., the number of classes or the related namespaces) or they are permanent characteristics of the namespace (e.g., the encoding or the URI prefix). This, in turn, imposes the rule that at least one version should exist in the Knowledge Repository for any stored namespace and its instance should be correctly related to the instance representing the namespace in the registry.

Since keeping track of versions has a significant role in the lifecycle of a schema, the registry includes a sophisticated versioning mechanism, accounting for and supporting

the fact that different versions of a schema can be developed in parallel. Thus, during the lifecycle of a schema its versions can create a Direct Acyclic Graph (DAG). This means that a version might depend on more than one versions, which might be considered as merging two or more versions. Similarly, two or more versions might depend on a single one, which might be considered as forking or parallel development. This way the maximum possible flexibility is provided and all known versioning schemes can be easily supported. Besides the versioning mechanism, the registry additionally offers the possibility to document the changes that occur on a schema when moving from one version to the next one(s). These changes have the format of the results of the Comparison Service that compares two RDF models.

Finally, as mentioned above, the Registry Service offers the possibility to retrieve ontology metadata information from the repository and also update the information that is already stored. In order to retrieve data from the registry, one can either type an RQL query, or use a query from a set of predefined ones. The latter type (the predefined queries) are exposed through a set of web service methods and are highly configurable by the developer of the service allowing for the necessary flexibility and taking advantage of the knowledge of the Ontology Registry Schema. Similarly, in order to update the information stored in the Registry a set of implemented web methods is exposed accounting for most actions that might be needed by the user and assuring the necessary consistency of the information in the Registry, imposing for example the rule of necessitating at least one version per schema; nevertheless, the user can always post updates in RUL, in which case (s)he bares also the responsibility for keeping the consistency rules.

The schema of Ontology Registry consists of five basic classes: Schema, Version, Change, foaf#Person and foaf#Organization.

- The Schema class represents a stored namespace (or ontology or schema) and includes, besides the URI of the schema, information about the creator, the title, the purpose, the keywords etc. Regarding the organization of the concepts described by a namespace, the kind of their interrelations and the level of conceptualization,

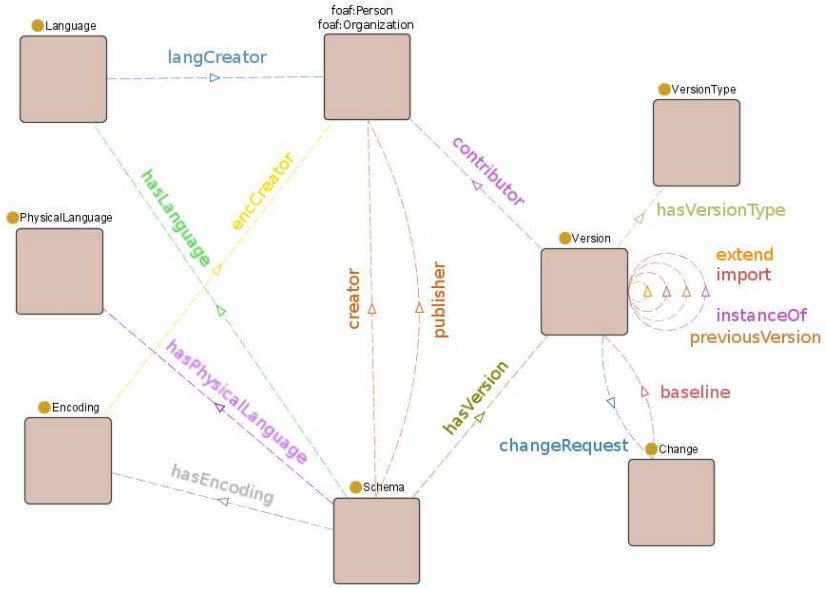


Figure 4.12: The Registry Schema

further classification is offered through the subclasses of class Schema. These subclasses are the following: Ontology, Thesaurus, Taxonomy, SemanticNetwork, DomainOntology, UpperOntology, TaskOntology, CoreOntology, ApplicationOntology, FederatedThesaurus, FacetedThesaurus and NetworkedThesaurus.

- The Version class is correlated to class Schema by the property hasVersion and describes attributes of a schema that might change between versions such as statistical characteristics of a schema (number of classes, number of properties, maximum length of a hierarchy). As one might see, this class also contains properties that correlate one schema to another with the relationships import, extend and instanceOf. Moreover, class Version has a property with predefined values that is used to indicate the intended uses of a version regarding its evolution during the version lifecycle. The predefined values are instances of VersionType class. The evolution can be seen in two ways: versions that are going to be developed in parallel and versions that are developed sequentially and depend on one another. Thus, the VersionType class can take the form of one of the following subclasses: Permanent (not to be merged in the future), Temporal (might be merged in the future) and Revision (replacing its previous versions).

- The Change class is correlated with class Version through the property changeRequest and describes the insertions/deletions of RDF statements that have led to the creation of this version (in the form of add/delete statements like the ones produced by the Comparison Service).

- The (FOAF#)Person and (FOAF#)Organization classes from the schema FOAF are correlated to both classes Schema and Version through the properties creator, publisher and contributor respectively.

Moreover, some additional classes have been specified that are related to the language, encoding, and physical language used in the document describing a specific namespace. The main classes and properties of Ontology Registry Schema are illustrated in Figure 9. The recording of a schema namespace by the Registry Service might also include the storing into the registry not only of the instances of classes Schema and Version but also instances of the classes Change, foaf#Person, foaf#Organization, Language, Encoding and PhysicalLanguage.

The Registry Service offers functionalities for:

- Storing information into the Ontology Registry Schema

- Updating information in the Ontology Registry Schema

- Retrieving information related to any object stored under the Ontology Registry Schema

As already mentioned, the methods exposed by the Registry Service are using the underlying methods offered by the SWKM platform, more specifically the Import, Update and Query Services. The Registry Service builds on top of these services in order to provide a more intuitive interface between the Knowledge Repository and the applications using the registry. These methods try to hide the possible complexity of producing the right (and optimized) RQL queries or RUL updates by predefining the correct ones, account for the consistency and imposing the necessary rules (which otherwise would have to be imposed manually) and exploit on the knowledge of the Ontology Registry Schema which the application need not know in detail.

So the available methods (web services) of the Ontology Registry API for inserting information into the Registry are:

element insertPersonURI {
element classURI { text },
element personURI { text }*,
element property { text }
}
element insertPersonURIResponse { }
	element insertSchemaURI {
	element className { text },
	element instanceURI { text },
	element versionId { text }
	}
	element insertSchemaURIResponse { }

element existInstanceURI {
element className { text },
element instanceURI { text }
}
element existInstanceURIResponse { }
	element insertOrganization {
	element classURI { text },
	element property { text },
	element file { text },
	element format { text }
	}
	element insertOrganizarionResponse { }

element insertOrganizationURI {
element classURI { text },
element personURI { text }*,
element property { text },
}
element insertOrganizationURIResponse { }
	element insertPerson {
	element classURI { text },
	element personURI { text }*,
	element property { text },
	element file { text },
	element format { text }
	}
	element insertPersonResponse { }

```
element insertProperty {
    element className { text },
    element instanceURI { text },
    element propertyName { text },
    element propertValue { text },
    element rangeType { text }
}
element insertPropertyResponse { }
```

```
element insertSchema {
    element className { text },
    element correlateToUri { text },
    element instance { text }
}
element insertSchemaResponse { }
```

```
element insertInstanceURI {
    element className { text },
    element correlateToUri { text },
    element instance { text }
}
element insertInstanceURIResponse { }
```

The corresponding ones for updating information already stored in the Registry (including deletion of instances from the registry, update of the range properties with the constraint that the properties have literals as a range, etc.) are:

```
element removeInstance {
    element className { text },
    element instanceURI { text }
}
element removeInstanceResponse { }

element editInstanceURI {
    element className { text },
    element oldURI { text },
    element newURI { text }
}
element editInstanceURIResponse { }

element editProperty {
    element className { text },
    element instanceURI { text },
    element propertyName { text },
    element oldValue { text },
    element newValue { text },
    element rangeType { text }
}
element editPropertyResponse { }
```

Additionally, the Registry Service uses the Query Service in order to retrieve data from the registry by evaluating RQL queries. The user can directly pose RQL queries through the *query* method of the Query Service or use the method that is implemented by the Registry Service API, called:

```
element evaluatePredefinedQuery {
    element queryCategory { text },
    element queryId { text },
    element param { text }*,
    element format { text}
}
element evaluatePredefinedQueryResponse {
    text
}
```

that can be used when the application needs to use one of the predefined queries which are in turn dynamically specified by the service developer in an XML file.

4.3.8 Versioning Service

The Versioning Service is responsible for constructing a new persistent version of a name or graph space already stored in the repository, in effect allowing the creation of several versions of an ontology or knowledge base, while keeping the logical relationships between each of its versions, i.e., which version was created as an evolution of which pre-existing one etc. It should be noted that the use of the Versioning Service is an indispensable part of any persistent change upon a name or graph space, as whenever a change upon a name or graph space becomes persistent, the resulting (updated) name or graph space should not overwrite the existing one, but should result to the creation of a new version of the name or graph space under question.

The versioning service in SWKM comes in two flavors, *"Import Version"* and *"Create and Import Versions"* operations. *"Import Version"* operation can be thought as a coordinator over the Importer and the Registry services, that stores a document and updates the Registry with versioning information, transactionally. *"Create and Import Versions"* offers less generic functionality, but more efficient, as it is creates versions based on deltas, i.e. on the actual differences between subsequent versions (which typically are much smaller than the complete contents of the versions themselves).

4.3.8.1 Import Version Operation

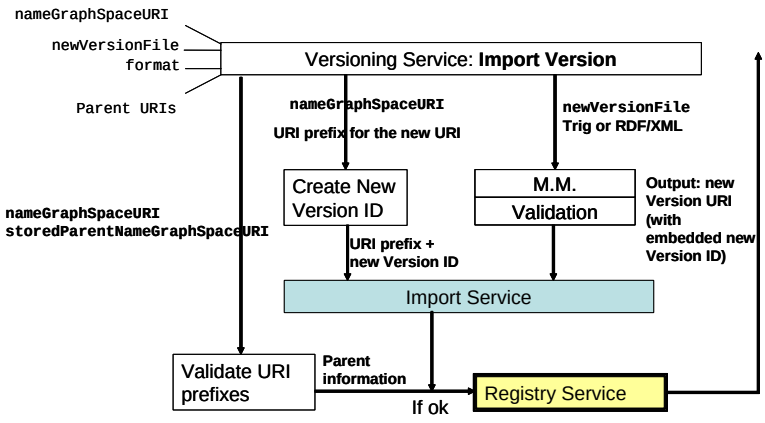


Figure 4.13: Import Version Operation

The *Import Version* operation offers versioning at the level of single name or graph

spaces. To this end, it takes as input the information regarding the version's URI, the parent versions' URI and the contents of the new version and creates a persistent version of the name or graph space in the given URI, with a new version ID. More specifically, the URI of a version is extended with a version tag that allows us the discrimination between various versions of a single name or graph space.

One of the requirements of the method is that the new version and its parents should have the same URI prefix, as they are assumed to be different versions of the same name or graph space. Therefore, the validity of the input URIs should be verified before making the new version persistent. The validity comprises not only checking the the URI prefixes match, but the parents are already registered in the Registry, possibly by a previous invocation of the versioning service.

Success of the validation (and the import) is a prerequisite for the new version to be recorded in the registry. If validation succeeds, the Registry Service is used to record the new version of the name or graph space. The final output of the service is a URI that includes the URI prefix and the version ID of the new version. A user may choose an arbitrary version ID (instead of allowing the automatic generation of one).

The interface to this operation is:

```
element importVersion {
    element document { rdfDocumentElement },
    element parent { text }*,
    element dbSettings { dbSettingsElement }?
}
element importVersionResponse {
    element versionUri { text }
}
element importVersionWithSpecificId {
    element document { rdfDocumentElement },
    element parent { text }*,
    element targetVersionId { text },
    element dbSettings { dbSettingsElement }?
}
element importVersionResponse {
    element versionUri { text }
}
element rdfDocumentElement {
    element URI { text },
    element content { text },
    element format { text }
}
```

The input consists of an RDF document parameter, which groups the RDF content itself, along with its base URI and its serialization syntax. The base URI is used to determine the URI prefix to be used in the new version's URI. The parent[] parameter is a list of strings, each containing the URI of one of the parent(s) of the current version. If there is no previous version of the given name or graph space (i.e., if the currently created version is the first one), then there are no parents, so the list is empty. Notice that the URI prefix could also be determined using the parents' prefixes, but this approach would fail for versions with no parents (i.e., for new name or graph spaces).

The RDF content is a string describing all the triples of the new version of the name

or graph space. These triples should be stored as the content of the new version. The format of the string in newVersionFile could be either TRIG or RDF/XML; the exact format is determined using the format parameter.

The output of the above method is a string containing the full URI, which includes both the URI prefix (i.e., the common URI prefix that is shared among all the versions of this name or graph space) and the version ID of the new version. This URI could be later used by the caller in order to get the contents of the new version, through a call to the Export Service.

4.3.8.2 Create and Import Versions Operation

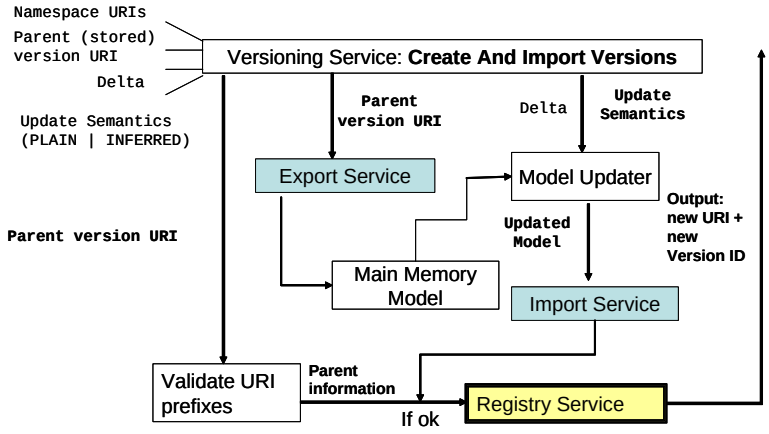


Figure 4.14: Create and Import Versions Operation

The *Create And Import Version* operation offers the functionality of specifying new versions not by their entire contents, but by the difference with an existing version, i.e. by specifying added and deleted triples (a delta). It also takes multiple URIs, unlike Import Version operation which only accepts one. Furthermore, these URIs can only be namespaces, not graphspaces, since it is yet unclear of the semantics of this operation in that case.

When a request is made, we first use the Exporter service to load the contents of all specified namespaces, and all their dependencies, transitively. We then create a main memory model by the results. Then, the question is where and how to apply the delta in order to create new namespace versions. The triples of the delta themselves are not

explicitly associated to a namespace. Instead, we use the convention to add the triples to the namespace of the subject of the triple; the same convention followed to assign triples to namespaces when creating main memory models with SWKM. So we define an implicit association of every triple to a namespace (the same would be problematic with graphspaces since there could be data instances triples with subject URIs that denote resources, and not schema defined in a namespace). This answers the "where" but not the "how". We offer two ways of applying triples to a namespace: either with a direct way, or by also considering (and adding) inferred triples (this option is controlled by the "updateSemantics" request parameter). Note that *existing namespaces are not affected in any way* by this operation; where needed, new namespaces are created by copying existing namespaces and applying the delta changes to the copies.

We do not allow triples to be added or deleted from (copies of) namespaces that do not appear in the arguments of the operation as this could create subtle side-effects that could go unnoticed there were made by mistake. Now, for each namespace that has at least one added or deleted triple, a new version of it is created, which is the result of the initially stored namespace and the addition and deletion of some triples (the initial namespace of course remains unchanged). The reason to have the operation defined with multiple namespaces instead of one (like Import Version) is because some interdependent namespaces could need to change in one step, so that their outcome is consistent. For example, if two namespaces that depend on each other change independently, then each one will now depend on the old version of the other one.

The interface of this operation is:

```
element createAndImportVersions {
    element existingVersionURI { text }*,
    element delta { deltaElement },
    element updateSemantics { text },
    element dbSettings { dbSettingsElement }?
}
element createAndImportVersionsResponse {
    element existingVersionURI { text }*,
    element newVersionURI { text }*
}
```

Some care for updating dependencies between specified namespaces is taken. All new versions will contain updated links (if there existed) to other new versions from the same operation. Old namespaces will continue to depend on the name or graphspaces they depended before this operation. For example, assume the user specifies namespaces $NS1, NS2, NS3$, and a delta that only affects $NS1$ and $NS2$. Also assume that $NS2$ and $NS3$ both depend on $NS1$ (i.e. $NS2 \rightarrow NS1, NS3 \rightarrow NS1$). After the operation, since delta affected only the two first namespaces, these namespaces will be created: $NS1', NS2'$. Also, $NS2'$ will now point to $NS1'$. But since $NS3$ was not affected by the delta, it will still point to the older $NS1$, even if it was declared in the operation and participated in the main memory model where this changes occurred. This is due to the model (which also takes cares of storing) ignoring changes in main memory representation, when there is an already stored namespace with the same URI - in this case, it silently ignores the changed namespace. So, whether after this operation, the latest version of each namespace will depend on the newer versions wherever they depended on the old, is based on whether the particular delta affects the dependent namespace: if it does, references will be updated, otherwise they will be left unchanged pointing to the old spaces.

We will explain the design forces behind this implementation, assess this service in Section 5.4.

In Figure 4.15 we see the interaction of the Change Impact service with Exporter and the Main Memory Model.

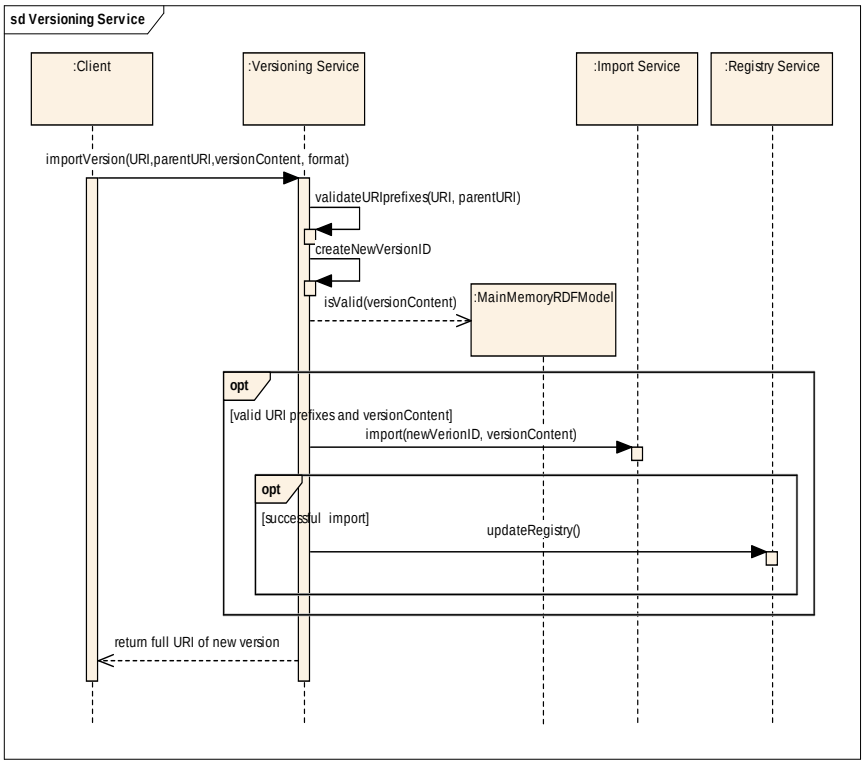


Figure 4.15: Versioning service interaction with Exporter, Main Memory Model and Importer

4.4 Usage of Services Example

In this section we will work through a simple example on working with the services. Assume we have a simple schema describing students and professors, shown below. We define a Student and a Professor class, and then a "hasSupervisor" property with domain "Student" and range "Professor".

```
<?xml version="1.0"?>
<rdf:RDF xml:lang="en"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xml:base="http://example.org/studentsAndProfessors.rdf#">

    <rdfs:Class rdf:ID="Student">
```

```
        <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/
    </rdfs:Class>

    <rdfs:Class rdf:ID="Professor">
        <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/
    </rdfs:Class>

    <rdfs:Property rdf:ID="hasSupervisor">
        <rdfs:domain rdf:resource="#Student"/>
        <rdfs:range rdf:resource="#Professor"/>
    </rdfs:Property>
</rdf:RDF>
```

We would store this schema as a namespace in SWKM repository using a call of this form:

```
String document = ...; //store in a variable the above document
importer.store("http://example.org/studentsAndProfessors.rdf#",
        document, "RDF/XML", false);
```

Now assume we also have a schema defining departments, which simply have a list of persons, which can be either students or professors.

```
<?xml version="1.0"?>
<rdf:RDF xml:lang="en"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xml:base="http://example.org/departments.rdf#">

    <rdfs:Class rdf:ID="Person">
        <rdfs:subClassOf
```

```
            rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
    </rdfs:Class>

    <rdf:Description
        rdf:about="http://example.org/studentsAndProfessors.rdf#Student">
        <rdfs:subClassOf rdf:resource="#Person"/>
    </rdf:Description>

    <rdf:Description
        rdf:about="http://example.org/studentsAndProfessors.rdf#Professor">
        <rdfs:subClassOf rdf:resource="#Person"/>
    </rdf:Description>

    <rdfs:Class rdf:ID="Department">
        <rdfs:subClassOf
            rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
    </rdfs:Class>

    <rdfs:Property rdf:ID="hasMember">
        <rdfs:domain rdf:resource="#Department"/>
        <rdfs:range rdf:resource="#Person"/>
    </rdfs:Property>
</rdf:RDF>
```

This schema effectively extends the previous one. We can store it like this:

```
String document = ...; //store in a variable the above document
importer.store("http://example.org/departments.rdf#", document, "RDF/XML", false);
```

If the repository didn't contain the first schema, then the second would fail to be stored, as it depends on the first.

We can review the stored classes by executing the simple "Class" RQL query:

```
String result = query.query("RDF/XML", "Class");
```

Printing the result would yield something like:

```
<?xml version="1.0"?>
<rdf:RDF xml:lang="en"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
<rdf:Bag rdf:about="#Bag_!">

<rdf:li rdf:type="http://www.w3.org/2000/01/rdf-schema#Class"
        rdf:resource="http://example.org#studentsAndProfessors.rdf#Student" />
<rdf:li rdf:type="http://www.w3.org/2000/01/rdf-schema#Class"
        rdf:resource="http://example.org#studentsAndProfessors.rdf#Professor" />

<rdf:li rdf:type="http://www.w3.org/2000/01/rdf-schema#Class"
        rdf:resource="http://example.org#departments.rdf#Person" />

</rdf:Bag>
</rdf:RDF>
```

Lets store some data instances as well, conforming to these schemata. We create this document:

```
<?xml version="1.0"?>
<rdf:RDF xml:lang="en"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:rdfsuite="http://139.91.183.30:9090/RDF/rdfsuite.rdfs"
    xmlns:std="http://example.org#studentsAndProfessors.rdf#"
    xmlns:dep="http://example.org#departments.rdf#"
```

```
    xml:base="http://example.org#data.rdf">


    <std:Student rdf:about="http://example.org#DimitrisAndreou">
        <std:hasSupervisor>
            <std:Professor
                rdf:about="http://example.org#VasilisChristophedes"/>
        </std:hasSupervisor>
    </std:Student>

    <dep:Department rdf:about="http://example.org#UnivOfCrete"/>

    <rdf:Description rdf:about="http://example.org#UnivOfCrete">
        <dep:hasMember
            rdf:resource="http://example.org#DimitrisAndreou"/>
        <dep:hasMember
            rdf:resource="http://example.org#VasilisChristophedes"/>
    </rdf:Description>

</rdf:RDF>
```

And we store it like this:

```
importer.store("http://example.org/data.rdf#", document,
        "RDF/XML", false);
```

We could again see the stored instances with RQL queries. Now, lets use the Exporter service instead.

```
List<String> uris =
        Arrays.asList("http://example.org#departments.rdf#");
exporter.fetch(uris, "RDF/XML", false, false);
```

This call would fetch the `departments.rdf` document we imported earlier, with probable syntactic differences, but with semantically equivalent content. Observing that this document depends on `studentsAndProfessors.rdf`, we can get both documents by specifying that we want all dependencies to be fetched too:

```
List<String> uris =
        Arrays.asList("http://example.org#departments.rdf#");
exporter.fetch(uris, "RDF/XML", true, false);
```

This would fetch both documents back. Another option is to also fetch the data instances that are described with the schemata requested, as following:

```
List<String> uris =
        Arrays.asList("http://example.org#departments.rdf#");
exporter.fetch(uris, "RDF/XML", true, true);
```

This would fetch both documents (namespaces) and a graphspace containing a student instance, a professor instance, and a property instance connecting the two.

Lets create a second version of the Departments document, one that incorporate a new type of Person, `"Staff"`. We will name it with a different URI, `"http://example.org/departments2.rdf#"` so that it will not interfere with the old one.

```
<?xml version="1.0"?>
<rdf:RDF xml:lang="en"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xml:base="http://example.org/departments2.rdf#">

    <rdfs:Class rdf:ID="Person">
        <rdfs:subClassOf
            rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
```

```
</rdfs:Class>

<rdfs:Class rdf:ID="Stuff">
    <rdfs:subClassOf
        rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdfs:Class>

<rdf:Description
    rdf:about="http://example.org/studentsAndProfessors.rdf#Student">
    <rdfs:subClassOf rdf:resource="#Person"/>
</rdf:Description>

<rdf:Description
    rdf:about="http://example.org/studentsAndProfessors.rdf#Professor">
    <rdfs:subClassOf rdf:resource="#Person"/>
</rdf:Description>

<rdf:Description rdf:about="#Staff">
    <rdfs:subClassOf rdf:resource="#Person"/>
</rdf:Description>

<rdfs:Class rdf:ID="Department">
    <rdfs:subClassOf
        rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdfs:Class>

<rdfs:Property rdf:ID="hasMember">
    <rdfs:domain rdf:resource="#Department"/>
    <rdfs:range rdf:resource="#Person"/>
</rdfs:Property>
```

```
</rdf:RDF>
```

In summary, we added a new `Staff` class, and an is-a relation of it to class `Person`. We can store this second `departments` document as we did the first. Then, we can compare it to the first one, by the Comparison service:

```
Delta delta = comparison.compare(
    new String[] { "http://example.org/departments.rdf#" },
    new String[] { "http://example.org/departments2.rdf#" }, "Delta_explicit");
```

The returned delta would contain a set of added triples (the new class instantiation and the is-a relation) and an empty set of deleted triples.

We could give this delta to the Change Impact service, like so:

```
Delta newDelta = changeImpact.changeImpact(
        new String[] { "http://example.org/departments.rdf#" }, delta);
```

And the service would yield a more complete Delta, one which would contain implicit triples that would occur if we applied the given delta.

Until now, we tried to create two versions of the `Departments` document in an ad-hoc fashion. This way we cannot ask later whether these versions are related, and how. Instead, we can use the Versioning service to store these versions. Assuming an empty repository, and `document1` containing the first version and `document2` containing the second, we would write:

```
String document1 = ...;
String document2 = ...;

String v1 = versionManager.importVersion("http://example.org/departments.rdf#",
        new String[] { }, document1);
String v2 = versionManager.importVersion("http://example.org/departments.rdf#",
        new String[] { v1 }, document2);
```

Note that the `importVersion` invocation yields a version URI assigned to the importer version. The second invocation uses the returned URI to declare that the second version is a subsequent version of the first. The first URI could be "http://example.org/departments.rdf _ 1#" while the second `"http://example.org/departments.rdf _ 2#"`.

Now, we can ask the subsequent ("kid") versions of the first namespace, an operation that will yield the second version, like this:

```
String result = registry.evaluatePredefinedQuery("Versioning", "findKids",
        new String[] { v1 }, "RDF/XML");
```

This would yield all subsequent versions of `v1`, in RDF/XML format.

4.5 Evaluation of the SWKM platform

Here we will explore the relative strengths and weaknesses of the SWKM middleware, in comparison with the rest of the middleware platforms discussed in chapter 2.

4.5.1 Strengths

SWKM is well integrated with powerful declarative query and update languages (RQL and RUL respectively), providing an easy way to developers to access and update data in the repository, without needing an array of less general services. For example, Sesame offers an explicit service that removes triples matching a simple (subject, predicate, object) triple pattern, which is an attempt to fill the gap of a full-fledged update language. Kowari's iTQL language offers comparable functionality, with declarative RDF inserts and deletes, but no updates (sometimes the latter can be emulated with a deletion followed by an insertion). But iTQL, unlike RQL/RUL, also offers administrating services, like creating and dropping models, taking backups of the database or restoring the database from a backup, etc.

SWKM makes it easy and efficient to export the contents of namespaces and graphspaces (through the Exporter service), as internally these are used to organize and group RDF

information, which would be impractical to do via querying. Graphspace support is still lacking or upcoming in other platforms.

SWKM uses SOAP via HTTP protocol to make its services as accessible as possible, since this protocol is currently highly supported in all main-stream languages (one of its main design goals was language neutrality/interoperability).

Last, but not least, the SWKM offers a comprehensive array of services that apart from basic imports and exports, queries and updates, offer facilities for evolution, finding differences, organizing ontologies and versioning, all in one integrated platform. SWKM attempts to be a one-stop solution for the most common requirements of Semantic Web applications, without requiring developers to install multiple add-ons or extensions to the server to support them.

4.5.2 Weaknesses

A major limitation of the SWKM platform is lack of support for distributed (or global) transactions. That is, an import request or an update cannot participate in an external, wider transaction, and this can be very problematic in case that such a transaction fails and is rolled back; then the SWKM repository would have to be manually restored, which can be a very complicated task. SWKM is technically limited by the fact that the underlying RQL/RUL engine runs on its own process, in a different platform (C++) than that of the rest of SWKM, and database connections cannot be shared simultaneously through Java and C++ without inordinate amount of work. Kowari tackles this problem by implementing the XA protocol, which enables global two-phase commits. Even without that support, Kowari exposes transaction operators in iTQL, so one could integrate manually Kowari into global transactions even without XA support. The problem that each basic service of SWKM operates strictly on its private transaction is apparent even inside SWKM services. For example, in Import Version operation of the Versioning Service, two main operations are executed: importing a namespace or graphspace into the repository, and notifying the Registry service of a new relationship, which in turn uses the Update service to record it. If the update fails for any reason, and we want to then the service will leave the repository in an inconsistent state; both a version will be stored

and not recorded in the Registry. [5]

Sesame and Jena both offer programmatic control for transactions in their respective libraries, though their middlewares cannot be directly integrated in wider transactions. Notably, a major design goal of Sesame 2 is better support for transactions.

Another limitation is that SWKM currently supports only RQL/RUL, but ignores standardization efforts which resulted in the SPARQL query language and protocol. This may limit its adoption, as clients will feel less secure using proprietary languages than a W3C standard.

[5] A work-around for this particular case would be to try atomically the Registry call first, then atomically the Import call, and if the latter fails, manually undo the changes of Registry happened in the first step.

Chapter 5

Related Work

This section examines the state of the art in benchmarking main memory representations and in versioning services and tools for the Semantic Web.

5.1 Related Work on Benchmarking RDF/S Main Memory Models

To the best of our knowledge, the benchmark presented here is the first extensive benchmarking of MMRMS. Instead, previous related work compare either RDF/S repositories implemented in secondary memory [43, 5, 37] or native and RDBMS-based RDF/S repositories with MMRMS [22, 36]. However, how to choose representative systems from each category is an open issue. As a matter of fact, our experimental study demonstrates quite divergent performance of the four MMRMS we evaluate, which is not actually reflected in the fourteen queries of the LUBM Benchmark [22]. Finally [29] compares the performance of MMRMS with respect to semantic association discovery queries. Unlike [29], we employ a complete query workload that allows for testing a wide broad of client programs information needs.

More precisely, in [43] we evaluated the performance of taxonomic queries (combining Q3 and Q18 of Table 3.1) for three popular database representations (i.e. schema-aware,

schema-oblivious and a hybrid one) of RDF/S schemata and instances employed by existing RDF/S repositories (i.e. RDFSuite[1], Jena-DB[2], Sesame-DB[3], DLDB[4], RStar[5], KAON[6], PARKA[7], 3Store[8]). The conclusion drawn from these experiments was that the most space and time efficient representation was the hybrid one supported by RDFSuite. In the same direction a recent work [5] reaffirmed the same performance advantages of the schema-aware approach over the schema-oblivious one (i.e. the triple-based view) supported by the majority of secondary memory RDF/S repositories. Furthermore, [37] compares four RDBMS-based RDF/S repositories (i.e. Jena-DB, Sesame-DB, SOR[9] and KAON) with three native ones (i.e. YARS[10], AllegroGraph[11], OWLIM[12]). According to this study SOR outperforms all other systems taking benefit from the performances of the DB2 query optimizer.

On the other hand, [22] compared two main-memory, (i.e. Sesame-MM, OWLJessKB[13]) with two secondary memory OWL systems (i.e. DLDB-OWL, Sesame-DB). The emphasis was to evaluate the OWL reasoning components in terms of the tradeoff between expressiveness and tractability in Description Logic reasoners. Moreover, [36] compared two MMRMS (i.e. Sesame-MM and Jena-MM), with two RDBMS-based (i.e. Sesame-DB and Jena-DB), and three native ones (i.e. Sesame-Native, Kowari[14], YARS). Specifically, the main conclusions drawn from [36] are that: *a*) native systems provide solid performance on large data applications but lack of sufficient inference capability, *b*) RDBMS-based systems need a further work on performance improvement, although providing nice support of inference, and *c*) MMRMS have serious scalability problems and can meet the needs only of small-scale SW applications.

[1] athena.ics.forth.gr:9090/RDF/
[2] jena.sourceforge.net/index.html
[3] www.openrdf.org/index.jsp
[4] swat.cse.lehigh.edu/downloads/dldb-owl.html
[5] www.alphaworks.ibm.com/tech/semanticstk
[6] kaon.semanticweb.org/
[7] www.cs.umd.edu/projects/plus/Parka/
[8] www.aktors.org/technologies/3store/
[9] http://www.alphaworks.ibm.com/tech/semanticstk
[10] sw.deri.org/2004/06/yars/
[11] agraph.franz.com/allegrograph/
[12] www.ontotext.com/owlim/
[13] edge.cs.drexel.edu/assemblies/software/owljesskb/
[14] www.kowari.org/

Finally, [29] studies four MMRMS, i.e. Jena-MM, Sesame-MM, Redland[15] and Brahms[16]. Brahms is a special purpose MMRMS, specifically developed to support efficiently semantic association discovery (i.e., finding a connecting path of semantic annotations between two resources). Here, we focused on general purpose MMRMS and we have included in our query workload semantic association discovery queries.

5.2 Related Work on Versioning

This section briefly discusses other systems and tools that offer versioning related services.

5.2.1 Ontoview

OntoView [33] is a web-based system[17] inspired by CVS [8] 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

[15] librdf.org/
[16] lsdis.cs.uga.edu/projects/semdis/brahms/
[17] A web demo is available here: http://test.ontoview.org/ but is currently not functional

the result of any of these types of change, more (human) input is needed to classify the change.

OntoView accepts changes and ontologies via several methods. Ontologies can be read in as a whole, either by providing a URL or by uploading them to the system. The user has to specify whether the provided ontology is new or that it should be considered as an update to an already known ontology. In the first case, the user also has to provide a "location" for the ontology in the hierarchical structure of the OntoView system.

Then, the user is guided through a short process in which he is asked to supply the meta-data of the version (as far as this can not be derived automatically, such as the date and user), to characterize the types of the changes, and to decide about the identifier of the ontology.

OntoView provides an explicit treatment for version identifiers. Typically, the XML Namespace mechanism is used to uniquely identify an ontology. This most of the times refers to a web location, i.e. the URL of the ontology file. OntoView distinguishes between this location identifier, and the logical identifier of the version, allowing them to vary independently. Each version will always have a unique location identifier, but not all versions will have a unique logical identifier; only versions which were created by conceptual changes do. So, creating a version by non-conceptual changes (for example, by adding comments to ontology constructs) do not cause the ontology and its elements to change identifiers.

OntoView also provides a web "diff" view for comparing two versions of an ontology (see Figure 5.1). The actual `diff` tool of CVS is used to implement that. This is a line-based tool, where the order of text is significant. So to produce a 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. In this web view, the user can characterize each difference, as explained previously.

The main advantage of storing the conceptual relations between versions of concepts and properties is the ability to use these relations for the re-interpretation of data and other ontologies that use the changed ontology. To facilitate this, OntoView can export differences between ontologies as separate mapping ontologies, which can be used as

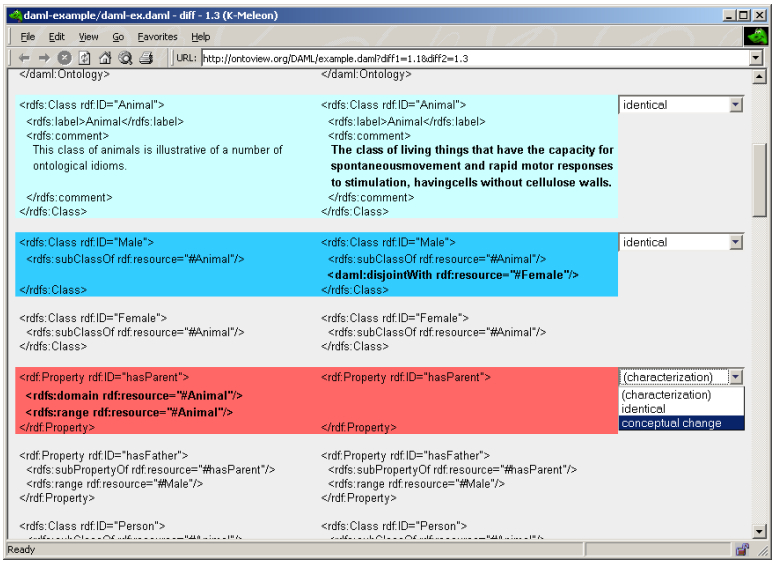


Figure 5.1: Comparing two ontologies in OntoView

adapters for data sources or other ontologies. These mapping ontologies thus impose a certain view or perspective on data source or other ontology.

In summary, the described system allows users to track the conceptual relations and transformations between concepts in different versions. Users are able to specify the conceptual implication of each difference between subsequent versions. Most important functions of OntoView (planned or implemented):

- read in ontologies, ontology transformations, and/or mappings
- view a specific version of an ontology
- differentiate ontologies (*diff*):
  - show changed formal definitions
  - show changed comments
  - show types of change: conceptualization or explication
- allow users to assign properties to differences
- export ontology transformations (i.e., description of the changes required to make an ontology identical to another)

- consistency checking of version combinations (i.e., 'is the combination of two specific ontology versions valid or produces contradictions?')

Currently it is based on CVS, and its underlying, syntactic-level, *diff* tool.

5.2.2 SemVersion

SemVersion [48] 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 [8]. 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. A main ("trunk") version is distinguished, also matching typical practices in CVS-like versioning systems.

Branching and merging are also supported. Merge is only offered with *union* semantics, as is the case with CVS. There is also some primitive support for reporting conflicts, but not resolving them programmatically.

The version storage of SemVersion is based on top of a *quad store* (also referred to as *named graphs*), which is repository of quadruples, instead of triples. This allows to create named sets of triples, which in turn, represent versions, while the "name" of the version (the forth dimension of the quad) is the version identifier (URI) itself. This identifier can be easily used to annotate a particular version with semantic data, such as provenance, comments, etc.

At the implementation level, persistence is handled by RDF2Go [18], which provides common storage interfaces over triple- and quad-stores (SemVersion uses the abstraction of the latter), such as Jena [19], Sesame [20], YARS [21], NG4J [22], etc. SemVersion stores each

[18] http://ontoware.org/projects/rdf2go/
[19] http://jena.sourceforge.net/
[20] http://www.openrdf.org/
[21] http://sw.deri.org/2004/06/yars/
[22] http://sites.wiwiss.fu-berlin.de/suhl/bizer/ng4j/

version of an RDF model as unique independent graph that contain the whole model.

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 a property to the blank nodes 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).

Two type of *diff*s are provided, *structural* and *semantic*. "Structural" means a set comparison of the triple sets, while "semantic" means including into the triple sets all inferred triples.

No declarative query language is supported for the contents of the stored versions, but supposedly the underlying storage providers may offer native querying capabilities, although usage of this effectively breaks the encapsulation of the implementation details of how versions are actually stored.

5.2.2.1 The MarcOnt Ontology Builder Case

SemVersion is applied in the case of the *MarcOnt initiative* [35]. MarcOnt initiative is concerned with developing an ontology for librarians through collaboration of multiple parties, and translators between the various existing library ontologies, like MARC21[23], BibTeX[24] and DublinCore[25]. A key tool in MarcOnt initiative, with the purpose of enabling the collaborative process, is the an ontology builder, integrated to the MarcOnt portal.[26]. SemVersion is used to implement the aforementioned builder.

Concurrent versions of the MarcOnt ontology are built out of suggestions proposed by

[23]http://www.loc.gov/marc/
[24]http://www.bibtex.org/
[25]http://dublincore.org/
[26]Currently, an online demo is offered here: http://portal.marcont.org/

community members (see Figure 5.2).

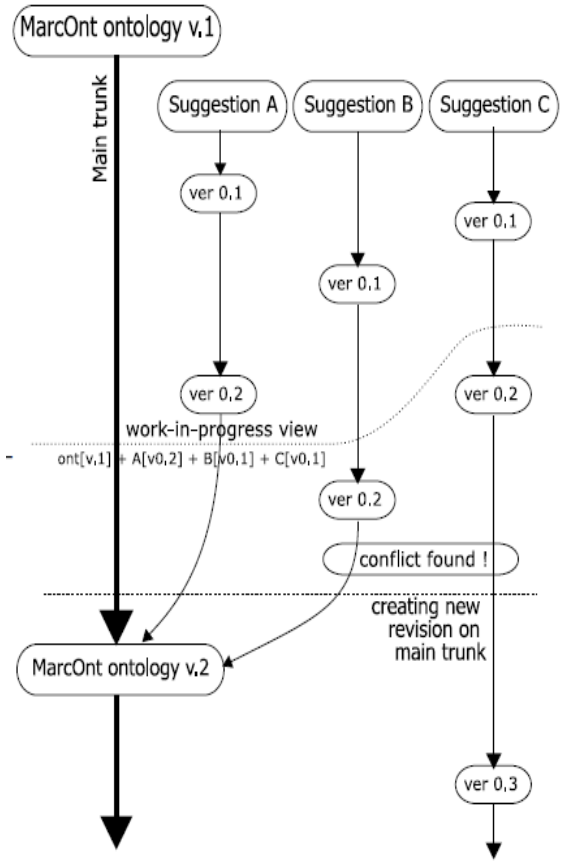


Figure 5.2: Multiple concurrent ontology branches

These suggestions themselves can evolve over time. After a voting process, some are applied and a new ontology version is created.

The MarcOnt ontology builder allows the user to create and manipulate ontologies online, viewing the difference between versions both syntactically and semantically (as defined previously), searching versions by using keywords, vote for versions created by peers and see the current ranking of the version, view some basic metadata about a particular ontology (who and when submitted it, and a comment).

Various viewers of the ontology are provided: a tree, with collapsing and expanding nodes (this could be somewhat problematic in multiple inheritance cases as nodes have to be replicated in order to be represented as a tree; although multiple inheritance is not used widely), a text editor with OWL, N3 and N-TRIPLE support, a table with triples, and a graph viewer using an applet (unfortunately, this currently is not working; probably

it would be the most interesting of the views). Finally, a web page is offered that can create mapping rules from one version to another, i.e. map a class X of a version to a class Y of another (in case where class X was renamed to Y in the second version).

5.2.3 Blackboard Collaboration Architecture of ConcepTool

Work of Compatangelo et al. [13] regards the management of version changes in a collaborative setting, using a blackboard architecture. In the described system, clients locally update ontologies, and the changes are transparently propagated to the central server, where a new version is created and is made available to the rest of the clients. Overall, several management services are described, which are summarized:

- Access and update rights and control
- Changes annotation/justification
- User profiling (assessing the contribution of each user, updating a user "reputation", which can later be reviewed by other peers, and taken into account when selecting alternative design paths)
- Reasoning about changes

The system offers access and update policies. JavaSpace [20] was leveraged to implement the blackboard architecture. [27]

5.2.4 MORE

Huang and Stuckenschmidt [27] present a novel approach in ontology versioning, which attempts to provide basic support for the problem of an evolved ontology's compatibility with application. The main use case is the following: the user imports a new version of some ontology into the system, and can then ask what facts of previous versions no longer hold in the last version, and what facts are new to the last version.

[27] Currently the software is unavailable (the download link at http://www.csd.abdn.ac.uk/research/IKM/projects/ConcepTool/ is not maintained), and cannot be properly evaluated.

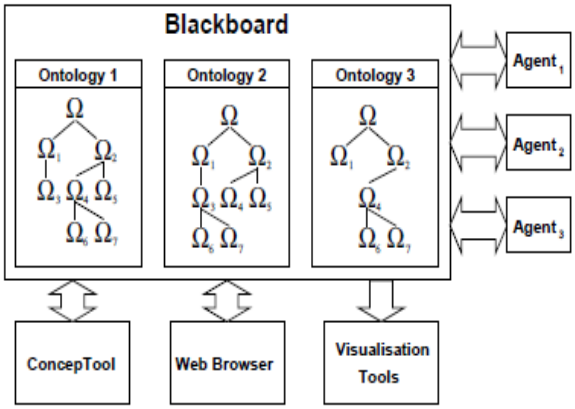


Figure 5.3: Blackboard Architecture

Since the main use case involves retrospective reasoning (from the point of view of the latest imported version), they define a set of logic past-oriented operators for reasoning about derivable statements in different versions. Namely:

- $PreviousVersion(f)$ operator states that a fact f holds just one state before the current state.
- $SomePriorVersion(f)$ operator states that f holds sometimes in the past with respect to the current state.
- $AllPriorVersions(f)$ operator states that f holds always in the past with respect to the current state.
- $AllVersions(f)$ operator which is as $AllPriorVersions$ but considers additionally the current version too.
- $Since(f, y)$ operator states that f holds since y holds too.

Then, they use linear temporal logic for reasoning about commonalities and differences between different versions, restricted to those operators. For the implementation of evaluating a query in this temporal logic, the standard model checking algorithm, which has been proven an efficient approach[12].

Supported queries can be divided in two categories: reasoning queries and retrieval queries. The former concerns with an answer either "yes" or "no", and the latter concerns

an answer with a particular value, like a set of individuals which satisfy the query formula. Namely, the evaluation of a reasoning query is a decision problem, whereas the evaluation of a retrieval query is a search problem. Both types are supported by the same underlying temporal logic.

Some examples of reasoning queries:

- *"Are all facts still derivable?"* $==> PreviousVersion(f) \wedge f$
- *"What facts are not derivable anymore?"* $==> PreviousVersion(f) \wedge \neg f$

Some examples of retrieval queries, assuming $child(c, c')$ is a condition that defines that concept c' is subsumed by concept c, and there are no other concepts between them, and S is the (linear) versioning space and o an ontology:

- *"newChildren(S,o,c)"* $==> \{c' | S, o \equiv child(c, c') \wedge \neg PreviousVersion(child(c, c'))\}$
- *"obsoleteChildren(S,o,c)"* $==> \{c' | S, o \equiv \neg child(c, c') \wedge PreviousVersion(child(c, c'))\}$
- *"invariantChildren(S,o,c)"* $==> \{c' | S, o \equiv child(c, c') \wedge PreviousVersion(child(c, c'))\}$

The same definitions can be extended into the cases like parent concepts, ancestor concepts, descendant concept and equivalent concepts.

Along these lines, a reasoner for multi-version ontologies called MORE has been implemented. The system is a mediator between an application and description logic reasoners (see Figure 5.4), and provides server-side XML-based services for uploading different versions of an ontology and posing queries to these versions. Requests to the server are analyzed by the main control component that also transforms queries into the underlying temporal logic queries if necessary. The main control element also interacts with the ontology repository and ensures that the reasoning components are provided with the necessary information and coordinates the information flow between the reasoning components. The actual reasoning is done by model checking components for testing temporal logic formulas that uses the results of an external description logic reasoner for answering queries about derivable acts in a certain version.

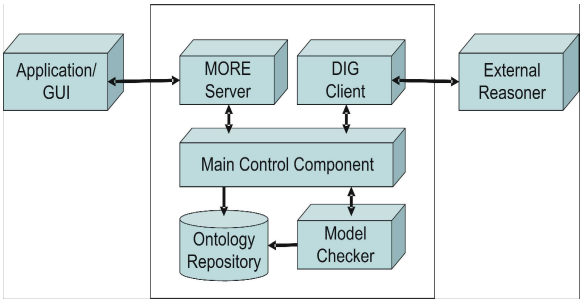


Figure 5.4: MORE System

The MORE prototype is implemented in Prolog and uses the XDIG interface[28], an extended DIG description logic interface for Prolog[28]. MORE is designed to be a simple API for a general reasoner with multi-version ontologies. It supports extended DIG requests from other ontology applications or other ontology and metadata management systems and supports multiple ontology languages, including OWL and DIG[29]. This means that MORE can be used as an interface to any description logic reasoner as it supports the functionality of the underlying reasoner by just passing requests on and provides reasoning functionalities across versions if needed. Therefore, the implementation of MORE is independent of those particular applications or systems.

Its main weakness is that linear temporal logic can only be used in linear version spaces, or in tree version spaces with past-based logic queries (a version's ancestors form a single line), and it is difficult to extend this work to function for future-based logic queries (that would imply tree version spaces, due to branches), or DAGs (due to version branches *and* merges). Scalability in non-linear logic queries evaluation could also become an issue.

5.2.5 DIP

In the context of project *DIP* [30] ("Data, Information, and Process Integration with Semantic Web Services"), funded by European Union, an ontology versioning tool[31] has been created.

[28]http://wasp.cs.vu.nl/sekt/dig
[29]http://dl.kr.org/dig/
[30]http://dip.semanticweb.org/
[31]http://www.omwg.org/tools/versioning/v1.0/FactSheet.html

Contrary to most other systems, DIP versioning is based on WSML ontology language[32], also developed in the context of the DIP project. The primary goal of WSML is the desire to formally model Web Services and capture their operational semantics, so as to enable . There are several variants of the language, namely:

- WSML-Core, which lies at the intersection of Description Logics and Logic Programming, and can thus function as the basic interoperability layer between both paradigms
- WSML-DL, a Description Logic language, with similar expressive power to OWL-DL [Patel-Schneider et al., 2004].
- WSML-Flight, with features oriented towards Logic Programming
- WSML-Rule, a full-blown Logic Programming language
- WSML-Full, a superset of WSML-DL and WSML-Rule, i.e. the merge of both paradigms,

DIP offers a suite of SW related tools, most packaged as Eclipse plug-ins [3]. Collectively, they handle editing and browsing ontologies, persistence, mapping concepts between pairs of them, versioning and reporting. We will concentrate on the versioning tool in particular.

Each version in DIP can be in one of two states: a committed (stable) one or a version in progress. A committed version is guaranteed to be immutable, so one can safely use it. It can only be 'modified' (that is, creating a modified copy, not altering the original version) by creating another version, possibly by 'branching' the former one. On the other hand, a 'version in progress' is modifiable and not meant to be published yet. (Note that this notion is not strictly necessary; a versioning system could deal only with committed versions, and let an engineer work on a local, private copy of an ontology before committing the finalized version of it). A user can add a comment at each commit, and also can choose an arbitrary version identifier for the committed ontology, as long as this is not already used. A commit dialog can be seen at Figure 5.5.

[32]Although formal mappings from WSML to RDF and OWL are provided

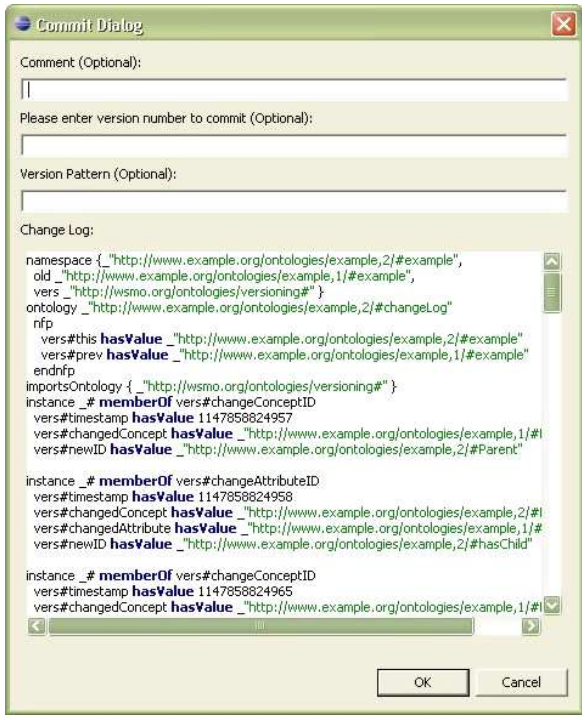


Figure 5.5: A commit dialog in DIP versioning tool.

The ontology editor tracks changes as they happen, and can propose a mapping from the initial ontology to the modified one. This only works for elements which did not change substantially. The user may also view the underlying change log, which is represented in the library as, rather surprisingly, a string, which is then parsed to create a mapping.

For version persistence, DIP versioning uses the Ontology Representation and Data Integration (ORDI [33]) framework, which provides a storage abstraction. There have been developed implementations that are based, respectively, on top of *Sesame*, *FOR* repository[34] and *YARS* repository[35].

The tool's abilities in summary include:

- Extending a version to produce a new one
- Logging every update into a change log, that can be parsed to produce an ontology
- Creating a mapping between concepts of subsequent versions of an ontology

[33]http://www.ontotext.com/ordi/v0.4/FactSheet.html

[34]The fact sheet of the closed-source repository is available here: http://sw.deri.org/2005/03/diprdf/UnicornRepositoryFactSheet.html

[35]http://sw.deri.org/2005/03/diprdf/FactSheet

- Parses the change log to create a textual description of the possible consequences and compatibility issues that can a rise from the changes described in that log
- Removing redundant entries from change logs (for example, replacing twice the same concept)

Regarding the storage of versions, no versioning-specific storage strategy has been developed; the underlying storage is simply manipulated to store each version as an independent copy.

5.2.6 GVS

GVS [1] is a recent tool by Hewlett Packard labs, based on the Jena framework[36]. The versioning dimensions offered by GVS are the following ones:

- Author (or Source) - *who* created a version
- Time - the physical moment that the version was added to the storage layer of GVS

The main operations are adding ('asserting') and deleting ('revoking') graphs, at particular times. The main query capability it to ask for a *GraphOverTime* object, by giving a set of sources, in the sense that the returned object represents the history of an RDF graph. A 'GraphOverTime' gives access to graph instances, as were asserted at certain time periods (i.e., at periods where they were asserted but not yet, if ever, revoked).

The main difference of this work to others is the choice of the level of granularity. The problem of granularity is well explained in [15]. 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 5.6. The first one shows an unnamed resource (blank node) with surname 'Ding' and first name 'Li'. The second graph is identical, while

[36] jena.sourceforge.net/

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 connected to them named resources or literals. Clearly, when blank nodes are involved, equality of triples can't reliably be used as identification of equal RDF content.

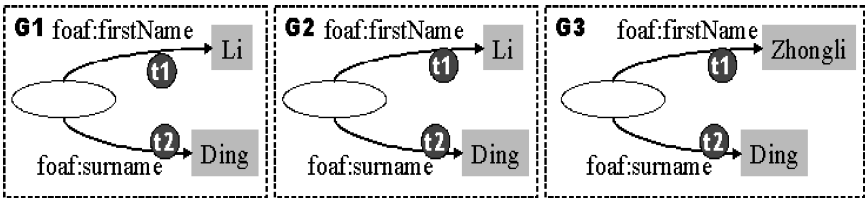


Figure 5.6: The three RDF graphs above show personal information from three sources. The first one asserts that a person who has first name 'Li' and surname 'Ding'.

In [15], 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 [45], one of the differences being that molecules also consider an arbitrary reasoning -the "background ontology"- while MSG deals only with RDF).

GVS uses RDF/S as the background ontology W, and according to this, it decomposes a given graph (which represents the version to be inserted) into its molecules. These can be simply merged to create the original graph. The contents of a version are uniquely identified by the set of the molecule identifiers that comprise the version. The molecules are stored once (there is no need to store identical duplicates), in the same manner that a triple-based versioning system would store all unique triples somewhere, and then store sets of references to the triples to express a version, so not having to store multiple times

the actual bytes of a triple.

GVS offers two storage kinds: a memory-based, and a filesystem-based, both backed with Jena's Graph abstraction. Files are stored in plain N-TRIPLES format.

For tracking history, GVS records and provides a sequence of time events, i.e. when changes occured, and by which source.

Performance-wise, GVS is rather primitive currently, as it does not use indexes (except for the main-memory-only case) for tracking the events of additions and deletions of graphs. An obvious optimization would be to leverage a relational database instead of files directly, so to be able to more easily create such indexes.

Querying capabilities are also thin. Apart from tracking the history of the graph assertions of a particular source (note that a source cannot add separate graphs - if a new graph is added without the previous one removed, then the graph returned will be the union of the two), GVS offers the ability to track the graphs of a *union* of sources. With this, valid (unrevoked) graph contents, asserted by at least one source at a specific past moment, are also reported. This presumably could be extended with other boolean operations, like intersection, which would mean "the contents asserted by *all* specified sources". Assertions can then be queried by triple-based patterns with Jena.

In terms of security, it offers optional write restrictions per user, but currently no reads can't be restricted. The security policy itself is described as an RDF graph[37].

Other features, not so much related to versioning, are a RESTful API (graphs can be obtained via HTTP GET, updated via HTTP PUT, asserted/revoked via HTTP POST).

5.2.7 Summary Comparison

Table 5.1 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.

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

[37]Possibly to demonstrate that an RDF data model is so convenient to use that it should be preferred to other formats

Features	Description
Structural Diff	The ability to compute the direct (set-)difference of two versions
Semantic Diff	Ability to compute the difference over the closure of the involved models/versions (i.e. inferred triples are also taken into account)
Change Log	The maintainance of the sequence of applied changes
Change-based Deltas	Can changes be expressed in terms of change operations?
Concept Mapping between versions	Automatic or semi-automatic association of concepts in one version to its next, in order to facilitate migration etc.
Storage approach	How is the problem of storing versions handled from a high level view?
Storage tools	What persistence tools are used?
Declarative Query Language	The ability to evaluate declarative queries over the contents of versions
Branch	The ability to spawn multiple versions from a single one
Merge	The ability to combine multiple versions to create a new one

Table 5.1: Features of the comparison

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.

By *Change-based Deltas* we mean the feature by which a user can create a version from a preexisting one by specifying declaratively the change that would create the new one. For example, this could be a update query in a language such as RUL, or simply sets of added and deleted triples produced by a diff algorithm. If the user cannot define a new version in terms of older ones and changes upon them, then the only remaining option is to import the entire contents of the new version to be created, which should be less efficient (and as the user typically would not create the entire contents from scratch, it is implied that the capability of applying changes to an RDF model is already available to him).

By *Concept Mapping between versions* we mean the association of concepts that can be derived from the Change Log or the application of Changed-based deltas (depending on the expressive power of the language used for the deltas), so the existence of at least one of these is a requirement for this feature.

The overview of available functionality is presented in Table 5.2.

We will briefly expand on how each toolkit fares for each distinguished feature we highlight, as synopsized in the above table.

Features	Systems						
	OntoView	**SemVersion**	**Blackboard / ConcepTool**	**DIP**	**MORE**	**GVS**	**SWKM**
Structural Diff	yes	yes	no	no	no	no	yes
Semantic Diff	no	yes	no	no	yes	no	yes
Change Log	no	no	no	yes	no	no	no
Change-based Deltas	yes	yes	yes	yes	no	no	yes
Concept Mapping between versions	no	no	no	yes	no	no	no
Storage tools	CVS	RDF2Go	?	ORDI (Sesame / YARS / FOR)	?	File system	RDFSuite (PostgreSQL)
Storage approach	Delta-based	State-based	State-based, also stores diffs when a version is created by applying one	State-based, also stores change-logs in main memory while working	State-based	Delta-based with molecule decomposition	State-based
Declarative Query Language	no	yes (as long as underlying store supports it)	no	yes (as long as underlying store supports it)	yes	no	yes
Branch	yes	yes	yes	yes	no	no	yes
Merge	no	yes	no	no	no	no	yes

Table 5.2: Features comparison of versioning systems and tools

5.2.7.1 Structural Diff

OntoView, SemVersion and SWKM, all offer basic structural diff between models. OntoView's support is the weakest, as it simply depends on the textual diff tool of CVS and on canonicalizing the RDF serialization (so, for example, indentation does not create erroneous differences). SemVersion and SWKM abstract completely from the serialization form used to represent SW data.

5.2.7.2 Semantic Diff

SemVersion, MORE, and SWKM offer a diff functionality that understands to some degree the semantics of RDF. This is closely related to inference and subsumption relations; the semantic diff algorithm can understand when implicit knowledge (triples) are added or removed.

5.2.7.3 Change Log

From the inspected tools, only DIP offer a change monitoring editor. That is, the DIP's editor keeps track of changes as the user makes them. This approach has, in principle, the potential to capture more fully the semantics of the change operations; instead of expressing the change with low level primitives (add triples, remove triples), high level user interface gestures can be leveraged, for example to more accurately describe the user's intention "move this hierarchy under this new ancestor" rather than "remove these is-a relations, add these".

5.2.7.4 Change-based Deltas

OntoView, SemVersion, ConcepTool, DIP and SWKM all have the ability to express a new version to be stored, as a function of a preexisting version and some change operations, instead of enumerating the whole contents of each new version. Of course, each system varies greatly in exactly how the changes are expressed. For example, OntoView represents changes as added and deleted text lines in the serialization of a model. SemVersion represent a change as a pair of added and deleted models, which consist of triples. SWKM uses pairs of added and deleted triples, in TriG serialization format.

5.2.7.5 Concept Mapping between versions

Only DIP offers rudimentary support for automatic mapping between concepts in subsequent version. To do so, it takes advantage of change logs which are generated by a visual editor during the work of an ontology engineer.

5.2.7.6 Storage Tools

Most tools use existing RDF repositories to store versions. Since an RDF repository already abstracts storage for RDF models, this is a good idea. SemVersion uses RDF2GO for storage, which abstracts several RDF repositories in a unified API. SWKM's storage relies on PostgreSQL (by mapping RDF schema and data to an RDBMS database). GVS uses a custom scheme based on the file system. DIP uses ORDI which, like RDF2GO, is

an abstraction over RDF repositories, and specifically over YARS, FOR, and Sesame.

5.2.7.7 Storage Approach

From a high level, these tools can be separated by the approach they take in order to store a version: store an independent copy of each version? Do something smarter to reduce storage requirements? But what happens with retrieval time?

We generally categorize SemVersion, ConcepTool, DIP, MORE and SWKM as stated-based, as they all create a full copy for each new version, even if the version shares much with a preexisting version. OntoView, naturally, stores only the (textual) deltas, as this is the way the underlying CVS works. GVS takes a very different route by storing only unique RDF "molecules" (see Section [**?**]); thus versions do not store redundant copies of their shared content. As of yet, it suffers from relatively bad performance for both storing and retrieval. Whether the cause is the work-in-progress state of GVS or the more fundamental reason of too fine granularity remains to be seen.

5.2.7.8 Declarative Query Language

Here we are concerned with whether the versions created with each tool can be declaratively queried. SemVersion delegates query evaluation to the underlying repositories (abstracted by RDF2GO), which offer various query engines (for instance, through Sesame: SPARQL, RDQL, RQL etc). Along the lines is also DIP, which delegates to its supported repositories (abstracted by ORDI) the evaluation of queries. SWKM is integrated with an RQL and RUL interpreter, and versions created by it can be queried with these languages (note that currently the queries do not pass through the versioning layer, so it is up to the user to preserve the existing versions without modifications by a RUL update query). MORE offers a logic based query language, but it is not general enough for application purposes; it is targeted at identifying the consequences of changes between versions, incompatibilities, etc.

5.2.7.9 Branch

Branching a version to spawn a new one is offered in OntoView, SemVersion, Concep-Tool, DIP and SWKM. MORE works only in a linear versioning space, and GVS does not have the notion of predefined versions that can be branched; versions are just facts asserted by someone at some time interval. So, a logical "version" is only defined given a specific time interval and users that assert facts.

5.2.7.10 Merge

Merging is supported in SemVersion, with an attempt at some primitive conflict resolution. In SWKM merging is not directly related to the versioning service. The merged version is to be created using the SWKM's main memory model, and then importing the created version, also defining two or more parents (which were merged). No specific facility is offered in SWKM for merging support, but only the general purpose main memory model and the generic service that stores new, arbitrary versions.

For an elaborate treatment of existing theoretical works related with ontology evolution and versioning, we refer you to a recent survey [19]. We will briefly mention here two important proposals that offer and understanding of the problems of versioning and possible approaches to a solution.

In [25] Heflin and Hendler discuss the problems associated with managing ontologies in distributed environments. They introduce SHOE, a web-based knowledge representation language that supports multiple versions of ontologies. SHOE includes annotations for making explicit the backward-compatibility relations between versions, allowing (but not enforcing) a computer agent to determine compatibility between versions.

Heflin and Pan [26] propose formal semantics of three types of inter-ontology links: commitment to an ontology by a resource, extension of one ontology by another, and backward-compatibility of an ontology with a prior version.

Klein and Fensel [32] compared ontology versioning to database schema versioning [40]. They described prospective and retrospective use (interpreting data with a newer or older version than the one they were initially created, respectively).

5.3 Open Issues

Issues that worth further research include:

- ability to manage versions of any user-defined granularity

- declarative version definition language

  For example ability to define a version as a the result of applying a number of operators over existing versions.

- provision of query services that involve more than one versions.

  For instance one might want to evaluate a query over the contents of a set of versions.

  Another aspect is the ability to query the metadata that are related to the various ontology versions (like time, author, compatibility, ancestry). For instance one might want to find all ontologies that are children of one particular ontology versions A and at the same time are compatible with another particular ontology versions B.

- efficient index structures for storing KBs aiming at reducing the storage space requirements

  Most of the existing works on versioning in the context of the Semantic Web propose high level services for manipulating versions but none of these have so far focused on the performance aspect of these services (they mainly overlook the storage space perspective). Two key performance aspects of a version management system is the storage space and the time needed for creating (resp. retrieving) a new (resp. existing) version. This is an interesting line for further research.

- provision of integrated graphical user interfaces for aiding the user-enacted version management tasks

5.4 Evaluation of the SWKM Versioning Service

We'll now sum up SWKM versioning offering and make an evaluation of it, compared to other systems. The basic traits of SWKM versioning can be epitomized in these points:

- The granularity of versions is set to that of namespaces and graphspaces, i.e. only namespaces and graphspaces may have versions, not models, nor triples (unless, of course, one was willing to assign a distinct graphspace per triple)
- Each version is stored as a complete, distinct copy in the repository (as is also the case with OntoView, Semversion, but notably not GVS)
  - In case of namespaces, the triples themselves are different, as the namespace URI changes (and for each triple contained in a namespace, at least the subject has the same namespace prefix which is changed from version to version, thus creating distinct triples) so they can't be shared
  - In case of graphspaces, where triples are treated more liberally in the sense that their namespace URIs don't have to much that of the container graphspace, the values of the triples are shared, but nevertheless each version stores an physically distinct set of *pointers* to these values
- Some automatic link (dependency) update between namespaces is provided for new namespace versions, as long as they are created by Create And Import Versions operation. Apart from that, the user must manually update the dependencies as he wishes.

We shall explain the technical circumstances and challenges that lead into these design decisions. First of all, keep in mind that, in SWKM, namespaces *define* schema triples; a schema triple cannot exist without being part of a namespace, and it is part of exactly one namespace (the one with the same namespace URI prefix as its subject, for example the triple "$ns1 : A \quad rdf : subClassOf \quad ns2 : B$" belongs to namespace $ns1$).

Each version needs to be identified by a URI that corresponds to a namespace or graphspace containing it (we will call *absolute URI* from now one the URI which contains versioning identification information, and *relative URI* a URI that does not). One

advantage of using absolute URIs is that the existing Exporter interface works regardless whether the name or graphspace is versioned or not (recall that Exporter takes as input only URIs, not version identifiers). If the URI of each version was relative, then existing code that relied on Exporter could become unspecified as soon as a used name or graphspace became versioned.

Another option would be to change the Exporter interface, so that it includes version identifiers per each URI to be exported, and possibly special identifiers to denote the "latest" (or "trunk") version or initial version, and a dummy version identifier that demanded an unversioned URI or raise an error.

Ideally, we would desire *not* to be forced to use absolute URIs if not necessary, and handle versioning as metadata on top of relative URIs. Absolute URIs create problems with maintaining anything external that was dependent on those URIs; with relative URIs there is nothing to maintain, but then there is less control over what a URI really means - it needs a resolution step via some versioning context. To give an example, consider the case of having query template strings. With relative URIs, the queries can be readily applied to any specific version, which would be specified via another path. With absolute URIs, the process is rendered quite problematic: the string templates themselves would have to change (practically, this means replacing the original namespace URI with the one that includes the new version identifier) to target each version, and this string-based work can be cumbersome with complicated query language syntaxes. One, though, could argue that there is not much value having unchanged queries that target multiple versions; for one reason, they would probably break in some versions due to changes, or worse, syntactically they would still work, but with different semantics (if concept meaning subtly changed from a version to the next).

Another consequence of having absolute version URIs, combined with granularity that is not coarse enough to capture multiple interdependent namespaces and graphspaces, is the link management between name and graphspaces that evolve. Suppose that there is an ontology defined inside a namespace, which evolves to a multitude of versions. Also suppose that there are multiple other namespaces and graphspaces which refer to (depend on) said namespace. If the new versions are to be imported with Import Version

operation, the user is responsible for creating new versions of the dependent name or graphspaces that would then depend on the newer versions instead of the old. As we explained earlier, manual updates may be in some cases required (i.e. not handled automatically), and currently SWKM platform does not offer a means to answer the question "what is depending on this version", which would allow custom measures to be taken for dependencies. Nor Registry could be used for this purpose as its usage is optional; not all dependencies are necessarily recorded there.

If instead relative URIs would be used, it would be possible that links would not have to be changed in-place; rather, when dependent spaces were aggregated to a coherent model, a versioning context would have to be provided (e.g. a mapping of relative URIs to unique version identifiers), to make validation possible. Or else, if the granularity was set coarser, that is, to the level of providing versions of entire models, then the user would have the option to nest all links completely inside a single version, and it would be possible to manage them completely and transparently.

It is to be noted that in practice, absolute versioned URIs are not uncommon at all. Some obvious examples are:

- `http://www.w3.org/2001/XMLSchema#`
- `http://www.w3.org/2000/01/rdf-schema#`
- `http://www.w3.org/1999/02/22-rdf-syntax-ns#`

All these encode versioning identification in the namespace URI. As we see, this simple technique is very valuable for wide publicity without ambiguity with regards to the version referred. So, we think it is necessary to allow the use of relative URIs (like SemVersion) and to have the ability to create absolute URIs to mark specific, important versions, so others can reliably depend on them. This is similar to *tagging* a release in a CVS or Subversion system - that is, giving a stable and readable name to a version.

As noted previously, coarsening the granularity level could allow even more useful cases, without sacrificing currently existing possibilities. But such an approach puts much more pressure on the persistence level and algorithms: for instance, assume versioning on models would be provided. Chances are that some parts change more often than others.

Storing a copy of the entire model for each version could prove too much of a burden, when some parts of it could be reused as-is because they were unchanged. In terms of storage, techniques to be able to partition version contents in such a way that much of it is reused (instead of copied) wherever possible is an area that needs attendance. Of course, such an effort would complicate the evaluation of queries, which would have to be evaluated on distributed parts of a version which have to be combined, instead of a single uniform space which could be queries directly.

Chapter 6

Conclusion

6.1 Synopsis and Key Contributions

In a nutshell, the contribution of this thesis lies in:

- benchmarking several different main memory representations, and
- designing and implementing versioning services for SW data

Specifically, several contemporary RDF management system/middleware platforms were analyzed in terms of architecture and supported services. The various abstractions of main memory models of each platform were compared, and conclusions and useful advice about which model is more preferable for typical use cases were given. In addition, this thesis provided a detailed description of the SWKM platform architecture, its available middleware services and implementation, and in particular its versioning services. Various design decisions behind the versioning services were discussed, by also comparing with other systems where relevant, and insights were provided on the related problems of implementing versioning facilities.

6.2 Directions for Further Research

Regarding main memory APIs an issue that is worth further research is the investigation of index structures that can offer efficient query evaluation for both RDF/S triple

and object views, and have low memory space requirements.

Regarding versioning, issues that are interesting for further investigation:

- ability to manage versions of any user-defined granularity
- declarative version definition language

  For example ability to define a version as a the result of applying a number of operators over existing versions.

- provision of query services that involve more than one versions.

  For instance one might want to evaluate a query over the contents of a *set* of versions. (The Registry service can already cope with one querying one version at the time).

  Another aspect is the ability to query the metadata that are related to the various ontology versions (like time, author, compatibility, ancestry). For instance one might want to find all ontology versions that are children of one particular ontology version A and at the same time are compatible with another particular ontology version B.

- efficient index structures for storing a large number of versions, aiming at reducing the storage space requirements

  Most of the existing works on versioning in the context of the Semantic Web propose high level services for manipulating versions but none of these have so far focused on the performance aspect of these services (they mainly overlook the storage space perspective). Two key performance aspects of a version management system is the storage space and the time needed for creating (resp. retrieving) a new (resp. existing) version. This is an interesting line for further research. Specifically for SWKM, it would be interesting to adopt graphspaces as the storage unit for version parts, which would open up possibilities for providing a multitude of options for the storage/efficiency trade-off. This way a more fine grained persistence for versions could be implemented, so that one could only need to store parts of versions that actually change, instead of creating entire copies of the versions in the storage.

Doing this in a way that it is still possible to transparently evaluate queries over the contents of a version is both a challenging and a promising possibility.

- provision of integrated graphical user interfaces for aiding the user-enacted version management tasks

Bibliography

[1] Graph versioning system. http://gvs.hpl.hp.com/.

[2] Ontology server research. http://www.starlab.vub.ac.be/research/dogma/OntologyServer.htm.

[3] Eclipse platform technical overview, 2003.

[4] SWAD-Europe Deliverable 7.1: RDF API requirements and comparison, 2003.

[5] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Procs of the VLDB'07*, Vienna, Austria.

[6] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Procs of the SIGMOD '89*.

[7] Sofia Alexaki. Storage of RDF metadata for Community Web Portals. Master's thesis, Computer Science Department - University of Crete, 2000. http://139.91.183.30:9090/RDF/publications/sofia.pdf.

[8] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[9] C. Bizer and D. Westphal. Developers Guide to Semantic Web Toolkits for different Programming Languages, 2007.

[10] Jeen Broekstra and Arjohn Kampman. SeRQL: A Second Generation RDF Query Language.

[11] Jeremy Carroll, Christian Bizer, Patrick Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *Procs of the WWW'05*, Chiba, Japan.

[12] Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. pages 1635–1790, 2001.

[13] Ernesto Compatangelo, Wamberto Vasconcelos, and Bruce Scharlau. The ontology versioning manifold at its genesis: a distributed blackboard architecture for reasoning with and about ontology versions. Technical report, 2004.

[14] Li Ding and Tim Finin. Characterizing the Semantic Web on the Web. In *Procs of the ISWC'06*, Athens, GA, USA.

[15] 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.

[16] Y. Ding and D. Fensel. Ontology library systems: The key to successful ontology reuse, 2001.

[17] John Domingue. Tadzebao and webonto: Discussing, browsing, editing ontologies on the web.

[18] A. Farquhar, R. Fikes, and J. Rice. The ontolingua server: A tool for collaborative ontology construction, 1996.

[19] Giorgos Flouris, Dimitris Manakanatas, Haridimos Kondylakis, Dimitris Plexousakis, and Grigoris Antoniou. Ontology change: classification and survey. *Knowledge Engineering Review*, 2007. (to appear).

[20] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, 1999.

[21] Y. Guo, J. Heflin, and Z. Pan. "Benchmarking DAML+OIL Repositories". In *Procs of the ISWC'03*, Florida, USA.

[22] Y. Guo, Z. Pan, and J. Heflin. An evaluation of knowledge base systems for large owl datasets. In *Procs of the ISWC'04*, Hiroshima, Japan.

[23] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A comparison of rdf query languages, 2004.

[24] J. HEFLIN and J. HENDLER. Dynamic ontologies on the web. 2000.

[25] Jeff Heflin and James A. Hendler. Dynamic ontologies on the web. In *AAAI/IAAI*, pages 443–449, 2000.

[26] Jeff Heflin and Zhengxiang Pan. A model theoretic semantics for ontology versioning. 2006.

[27] Z. Huang and H. Stuckenschmidt. Reasoning with multiversion ontologies: A temporal logic approach. In *Proceedings of the 2005 International Semantic Web Conference (ISWC05)*, 2005.

[28] Zhisheng Huang and Cees Visser. Extended dig description logic interface support for prolog. Deliverable D3.4.1.2, SEKT, 2004.

[29] Maciej Janik and Krys Kochut. Brahms: A workbench rdf store and high performance memory system for semantic association discovery. In *Procs of the ISWC'05*.

[30] G. Karvounarakis, V. Christophides, and D. Plexousakis. Querying Semistructured (Meta)data and Schemas on the Web: The case of RDF & RDFS. Technical Report 269, ICS-FORTH, 2000. Available at: http://www.ics.forth.gr/proj/isst/RDF/-rdfquerying.pdf.

[31] Siorpaes Katharina and Hepp Martin. Requirements and state-of-the-art document for a scalable ontology, instance data and mapping management infrastructure. myontology project deliverable, DERI, 2007.

[32] M. Klein and D. Fensel. Ontology versioning for the semantic web, 2001.

[33] M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. "Ontology versioning and change detection on the web". In *Procs of the 13th European Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, pages 197–212. Springer, 2002.

[34] George Konstantinidis, Giorgos Flouris, Grigoris Antoniou, and Vassilis Christophides. Ontology evolution: A framework and its application to rdf. 2007. In Proceedings of the Joint ODBIS and SWDB Workshop on Semantic Web, Ontologies, Databases (SWDB-ODBIS-07).

[35] Sebastian Ryszard Kruk, Marcin Synak, and Kerstin Zimmermann. Marcont initiative. bibliographic description and related tools utilising semantic web technologies, 2005.

[36] Baolin Liu and Bo Hu. An evaluation of rdf storage systems for large data applications. In *Procs of the SKG'05*.

[37] Robert Lu, Chen Wang, Li Ma, Yong Yu, and Yue Pan. Performance and Scalability Evaluation of Practical Ontology Systems. In *Procs of the Joint ODBIS & SWDB. Colocated with VLDB'07*.

[38] M. Magiridou, S. Sahtouris, V. Christophides, and M. Koubarakis. "RUL: A Declarative Update Language for RDF". In *Procs. 4th Intern. Conf. on the Semantic Web (ISWC-2005)*, Galway, Ireland, November 2005.

[39] Eyal Oren, Renaud Delbru, Sebastian Gerke, Armin Haller, and Stefan Decker. ActiveRDF: Object-Oriented Semantic Web Programming. In *Procs of the WWW'07*.

[40] John F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[41] Andy Seaborne. RDQL - A Query Language for RDF.

[42] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In *Procs. of the 4th International Semantic Web Conference (ISWC'05)*.

[43] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In *Procs of the ISWC'05*.

[44] Yannis Theoharis. On Power Laws and the Semantic Web. Master's thesis, Computer Science Department, University of Crete, February 2007.

[45] 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.

[46] Yannis Tzitzikas and Dimitris Kotzinos. "(Semantic Web) Evolution through Change Logs: Problems and Solutions". In *Proceedings of the Artificial Intelligence and Applications , AIA'2007*, Innsbruck, Austria, February 2007.

[47] Max Völkel. Writing the semantic web with java, 2005.

[48] Max Volkel, Wolf Winkler, York Sure, Sebastian Ryszard Kruk, and Marcin Synak. "SemVersion: A Versioning System for RDF and Ontologies". Heraklion, Crete, May 29 • June 1 2005. Procs. of the 2nd European Semantic Web Conference, ESWC'05.

[49] David Wood, Paul Gearon, and Tom Adams. Kowari: A platform for semantic web storage and analysis. 2005. XTech Conference.

[50] D. Zeginis, Y. Tzitzikas, and V. Christophides. On the foundations of computing deltas between rdf models. 2007. In Proceedings of the 6th International Semantic Web Conference (ISWC-07).

[51] A. Zhdanova, R. Krummenacher, J. Henke, and D. Fensel. Community-driven ontology management: Deri case study. Compiegne, France, 2005. IEEE Computer Society Press. Proc of the IEEE/WIC/ACM International Conference on Web Intelligence.