University of Crete

Computer Science Department

Compact Archiving of Multiple (possibly Versioned) RDF/S Triple Sets

Maria-Georgia Psaraki

Master's Thesis

Heraklion, March 2011

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Συμπαγής Αρχειοθέτηση Πολλαπλών Εκδόσεων από Σύνολα Τριπλετών RDF/S

Εργασία που υποβλήθηκε από τη

Μαρία-Γεωργία Ψαράκη

ως μερική εκπλήρωση των απαιτήσεωνν για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

____

Μαρία-Γεωργία Ψαράκη, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

____

Γιάννης Τζίτζικας, Επίκουρος Καθηγητής, Επόπτης

____

Δημήτρης Πλεξουσάκης, Καθηγητής, Μέλος

____

Ειρήνη Φουντουλάκη, Ερευνήτρια του Ινστιτούτου Πληροφορικής ΙΤΕ, Μέλος

Δεκτή:

____

Άγγελος Μπίλας, Αναπληρωτής Καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Μάρτιος 2011

Compact Archiving of Multiple (possibly Versioned) RDF/S Triple Sets

Maria-Georgia Psaraki
Master's Thesis
Computer Science Department, University of Crete

Abstract

The Semantic Web (SW) is an evolving extension of the World Wide Web, in which content can be expressed not only in natural language, but also in languages (e.g. RDF/S) that can be interpreted formally enabling the provision of more advanced searching, sharing and integration services. However since knowledge is not static, but evolves over time, there is a need for techniques for managing this evolution. One of them is that of archiving past versions. Archiving is useful for various reasons (interoperability, traceability, provenance). For instance, in e-science failure to keep the previous states of data (over which other experiments were based) jeopardizes scientific evidence and our ability to verify findings.

In this work we use the term RDF KB (for short KB) to refer to any set or RDF/S triples. POI (Partial Order Index) is a structure which has been proposed recently for storing multiple (either versioned or not) KBs. POI exploits the fact that RDF is based on a graph data model, and hence an RDF KB does not have a unique serialization (as it happens with texts). This characteristic justifies exploring directions that have not been elaborated by the classical versioning systems for texts (e.g. versioning systems for software). In brief, POI offers notable space saving in comparison to the differential storage of KBs (storing deltas), as well as efficiency in various cross version operations, especially in cases where the contents of the KBs are subset-related.

This thesis focuses on methods for further reducing the space requirements of POI. Specifically we introduce a variation of POI that we call CPOI (Compact POI), which relies on gapped triple identifiers with variable length encoding schemes for natural numbers. For this structure we identified conditions under which CPOI guarantees space gains (over POI and other storage options). Since these are sufficient (not necessary) conditions, we conducted an extensive experimental evaluation, also for measuring the compression ratio achieved, and for comparatively evaluating various identifier assignment policies.

The results showed significant storage savings, specifically, the total space required in large and realistic synthetic datasets is in average around 25 times less than the size of the original dataset and 3 times less than the size of a differential (delta-based) storage. The total size of CPOI is about 60%-80% of the size of plain POI, while the size of the compressed sets is 8% of the size of the uncompressed sets.

Supervisor: Yannis Tzitzikas
Assistant Professor

Συμπαγής Αρχειοθέτηση Πολλαπλών Εκδόσεων από Σύνολα Τριπλετών RDF/S

Μαρία-Γεωργία Ψαράκη
Μεταπτυχιακή Εργασία
Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Ο Σημασιολογικός Ιστός (ΣΙ) είναι μία εξελισσόμενη επέκταση του Παγκόσμιου Ιστού, στην οποία το περιεχόμενο μπορεί να εκφραστεί όχι μόνο σε φυσική γλώσσα αλλά και σε γλώσσες (π.χ. RDF/S) που επιδέχονται τυπικής ερμηνείας και καθιστούν εφικτή την παροχή προηγμένων υπηρεσιών αναζήτησης, διαμοιρασμού και ολοκλήρωσης πληροφορίας. Καθώς όμως η γνώση δεν είναι στατική, αλλά συνεχώς εξελίσσεται, απαιτούνται διάφορες τεχνικές για τη διαχείριση της εξέλιξής της. Μία τέτοια ανάγκη είναι αυτή της αρχειοθέτησης προηγούμενων εκδόσεων. Η αρχειοθέτηση εκδόσεων είναι χρήσιμη για διάφορους λόγους (διαλειτουργικότητα, ιχνηλασιμότητα, προέλευση). Για παράδειγμα στην Η-Επιστήμη (E-Science) η αποτυχία διατήρησης των προηγούμενων καταστάσεων των δεδομένων (επί των οποίων εκτελέστηκαν μεταγενέστερα πειράματα) θέτει σε κίνδυνο τη δυνατότητα επαλήθευσης των ερευνητικών αποτελεσμάτων.

Στην εργασία αυτή με τον όρο Βάση RDF (BR) θα αναφερόμαστε σε οποιοδήποτε σύνολο τριπλετών RDF/S. Το POI (Partial Order Index) είναι μία δομή που προτάθηκε πρόσφατα για την αποθήκευση πολλών (εκδόσεων) BRs, η οποία αξιοποιεί το ότι η RDF βασίζεται σε ένα μοντέλο δεδομένων γράφου, και άρα μία BR δεν έχει μία μοναδική σειριοποίηση (όπως έχουν τα κείμενα). Αυτή η ιδιαιτερότητα δικαιολογεί τη διερεύνηση κατευθύνσεων που δεν έχουν μελετηθεί στα πλαίσια των κλασσικών συστημάτων εκδόσεων για κείμενα (π.χ. στα συστήματα διαχείρισης εκδόσεων λογισμικού). Εν συντομία το POI προσφέρει σημαντική εξοικονόμηση αποθηκευτικού χώρου σε σχέση με τη διαφορική αποθήκευση (delta-based storage), ιδιαίτερα αν υπάρχουν BRs οι οποίες σχετίζονται με σχέση υποσυνόλου.

Η εργασία αυτή επικεντρώθηκε σε μεθόδους για περαιτέρω μείωση των αποθηκευτικών απαιτήσεων του POI. Συγκεκριμένα, προτείνουμε μια έκδοση του POI που ονομάζουμε CPOI (Compact POI) η οποία χρησιμοποιεί διαφορικά (gapped) αναγνωριστικά τριπλετών και

μεταβλητού μεγέθους κωδικοποιήσεις φυσικών αριθμών. Για τη δομή αυτή μελετήσαμε αναλυτικά τις συνθήκες υπό τις οποίες εγγυάται εξοικονόμηση χώρου σε σχέση με το απλό POI και άλλες επιλογές αποθήκευσης. Επειδή οι συνθήκες είναι ικανές (αλλά όχι και αναγκαίες), και προκειμένου να μετρήσουμε το λόγο συμπίεσης (compression ratio) που επιτυγχάνεται, αλλά και να αξιολογήσουμε συγκριτικά διάφορες πολιτικές ανάθεσης αναγνωριστικών, προχωρήσαμε και σε μια εκτενή πειραματική αξιολόγηση. Τα αποτελέσματα κατέδειξαν σημαντικά οφέλη χώρου, ήτοι ο απαιτούμενος χώρος (σε μεγάλα σύνολα συνθετικά παραγμένων δεδομένων) είναι κατά μέσο όρο περίπου 25 φορές μικρότερος από το χώρο των αρχικών δεδομένων και 3 φορές μικρότερος από τη διαφορική αποθήκευσή τους. Τέλος, ο χώρος που καταλαμβάνει το CPOI είναι περίπου το 60%-80% του χώρου που απαιτεί το απλό POI, ενώ η συμπιεσμένη αναπαράσταση των συνόλων του αντιστοιχεί στο 8% του χώρου που απαιτούν τα μη συμπιεσμένα σύνολα.

Επόπτης Καθηγητής: Γιάννης Τζίτζικας
Επίκουρος Καθηγητής

Στους δύο μου Ευτύχηδες

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω θερμά τον επόπτη καθηγητή μου κ. Γιάννη Τζίτζικα για την πολύ καλή συνεργασία μας και την πολύτιμη καθοδήγησή του για την ολοκλήρωση αυτής της εργασίας. Επίσης, θα ήθελα να ευχαριστήσω τους καθηγητές κ. Δημήτρη Πλεξουσάκη και κα. Ειρήνη Φουντουλάκη που με προθυμία δέχτηκαν να συμμετάσχουν στην εξεταστική επιτροπή της μεταπτυχιακής μου εργασίας, καθώς και για τις εύστοχες παρατηρήσεις και συμβουλές τους.

Ένα μεγάλο ευχαριστώ οφείλω στους φίλους μου, ιδιαίτερα στη Μαίρη Καμπουράκη για τη στήριξή της και όσα ζήσαμε μαζί τα τελευταία δύο χρόνια, καθώς και στο Νίκο Μανώλη και τον Παναγιώτη Παπαδάκο για τις συζητήσεις μας. Επίσης θα ήθελα να ευχαριστήσω και τα υπόλοιπα παιδιά των λευκών για όσα ζήσαμε και μοιραστήκαμε μαζί.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου, τον μπαμπά μου Ευτύχη και τον αδερφό μου Ευτύχη για τη συνεχή στήριξη και συμπαράσταση που μου έχουν προσφέρει και για όλα όσα έχουν κάνει για μένα όλα αυτά τα χρόνια. Επίσης, θα ήθελα να ευχαριστήσω τον θείο μου Ανδρέα για τη στήριξη και την αγάπη του.

Contents

Table of Contents iii

List of Tables v

List of Figures ix

1 Introduction 1

1.1 Introduction to the Semantic Web 1
1.2 Versioning Services 4
1.2.1 Versioning Services in the Software Development Area 4
1.2.2 Ontology Versioning 4
1.2.2.1 Versioning for RDF Models 5
1.3 Contribution of this Thesis 6
1.3.1 Organization of this thesis 6

2 Background and Related Work 9

2.1 RDF/S Knowledge Bases 9
2.2 Requirements and Key Aspects 9
2.3 RDF Triple Storage 11
2.3.1 Sesame 11
2.3.2 Jena2 12
2.3.3 3store 13
2.3.4 YARS (Yet Another RDF Store) 15
2.3.5 Discussion on Database Representations 15
2.4 Compact RDF Triple Representations 17

2.4.1 RDF-3X 17
2.4.2 HTD representation 20
2.4.3 Discussion 22
2.5 Versioning Approaches for Semantic Web 22
2.5.1 x-RDF-3X 22
2.5.2 Versioning Model for Biomedical Ontology Versions 23
2.5.3 OntoView 25
2.5.4 SemVersion 27
2.5.5 Ontology Middleware Module 29
2.5.6 Comparison 31
2.6 Compression Techniques in IR 32
2.6.1 Document identifier assignment according to the lexicographical ordering of the URLs 33
2.6.2 Document reordering using B&B algorithm 33
2.6.3 Document identifier reassignment using TSP algorithm 33
2.6.4 Assigning identifiers to documents on the fly 34
2.6.5 Document identifier reassignment using dimensionality reduction with SVD 34

3 The Partial Order Index (POI) 35
3.1 Introduction 35
3.2 Framework and Notations 36
3.3 The Partial Order Index (`POI`) in Detail 38
3.4 Analyzing Storage Space Requirements 40
3.5 Version Insertion Algorithms 42
3.5.1 *POI-Plain (Insert POI_p)* Insertion Algorithm 43
3.5.2 *POI-DoubleStack (Insert POI_{ds})* Insertion Algorithm 45
3.5.3 *POI-DoubleStackAndClean (Insert POI_{dsc})* Insertion Algorithm . . 46
3.5.4 History-based Version Insertion Speedup 48
3.6 Cross Version Operations 49
3.6.1 Containment Checking 49
3.6.2 Containment Queries 49

3.7 Extensions . . . 50
3.7.1 Semi-lattice approach . . . 50
3.8 Comparison with Change-Based Approach . . . 51

4 The Compact POI (CPOI) 53
4.1 Introduction . . . 53
4.2 Analytical Comparison of CPOI with plain POI . . . 55
4.2.1 Gaps and Storage Space . . . 59
4.2.1.1 Uniform Codes for Ids . . . 60
4.3 Assignment Policies . . . 63
4.3.1 Discussion . . . 68
4.4 CPOI and real world applications . . . 68

5 Experimental Evaluation 69
5.1 Synthetic Datasets . . . 69
5.2 Real Datasets . . . 72
5.3 Discussion over Datasets . . . 72
5.3.1 Metrics for Comparing Datasets . . . 73
5.4 Compared Options . . . 74
5.5 Results . . . 77
5.5.1 Summary of compression ratios . . . 83
5.5.2 Experimental Results vs Analytical Results . . . 84
5.6 Discussion . . . 84
5.6.1 Operations over a CPOI . . . 84

6 Conclusions and Future Work 87
6.1 Synopsis and Key Contributions . . . 87
6.2 Directions for Further Research . . . 88
6.2.1 Inverted File Structure . . . 89

List of Tables

2.1 RDF/S storage schemes 17

3.1 Version creation services 37
3.2 Notations for Storage Graphs 39

4.1 Upper and Lower Bounds for $Gaps(N)$ 59
4.2 Space (in bits) by the nodes of each compression method 63

5.1 Compression ratios of `CPOI` in comparison with the rest policies 83
5.2 Compression ratios of `CBD` in comparison with the rest policies 84

List of Figures

1.1 The Semantic Web architecture 2
1.2 RDF and RDFS layers. Blocks are properties, ellipses above the dashed line are classes, ellipses below the dashed line are instances. 3

2.1 The Sesame Architecture 12
2.2 The object-relational schema used with PostgreSQL 13
2.3 The relational schema used with MySQL 14
2.4 Jena2 Database Schema 15
2.5 Database schema and ER diagram showing table relationships 16
2.6 Schema-aware representation 17
2.7 Schema-oblivious representation 17
2.8 Schema-hybrid representation 18
2.9 Differential indexes in RDF-3x (left) and Triple compression algorithm (right) 19
2.10 Structure of a compressed triple 19
2.11 Triples compression through an example 19
2.12 Construction of the HDT format from a set of triples 21
2.13 Incremental representation of an RDF data set with HDT 21
2.14 The transaction inventory that keeps all transactions info 23
2.15 The relational repository schema for the versioning model 24
2.16 Comparing two ontologies in Ontoview 26
2.17 The Layered Architecture of SemVersion 29
2.18 The database schema used from the SQL92SAIL of Sesame 30
2.19 The database schema implemented of Ontology Middleware as extensions of Sesame 30

3.1 Storage Example when the partial order index is used (right) or not (left) . 38
3.2 Best (upper) and worst (bottom) case for POI 40
3.3 Example of adding a new version using Algorithm *Insert POIds* 42
3.4 Example of the semi-lattice approach 51

4.1 A compact representation of POI 55
4.2 Synopsis of analytical results 63
4.3 Storage graph of POI without and with a gapped representation 64
4.4 Assignment by Size 65
4.5 Assignment by Frequency 66
4.6 Assignment by BFS 67

5.1 Synthetic dataset generation method for Dat1 70
5.2 Synthetic dataset generation method for Dat2 71
5.3 The distribution (in logscale) of triples in versions (left) and in nodes (right) for Dat2 for d=0.7 (Y: frequency, X: triples) 71
5.4 SCD of Storage Graphs' for Dat1 and Dat2 74
5.5 AverageDepth (left) and MaxDepth (right) of Storage Graphs' for Dat1 and Dat2 74
5.6 Space for the nodes' contents for Dat1 (left) and Dat2 (right) of POI, CPOIu and CPOI (for Unary and Elias-γ encoding). 77
5.7 Nodes's space for Dat1 (left) and Dat2 (right) of CPOI using Elias-γ encoding for various assignment policies. 78
5.8 Nodes's space for Dat1 (left) and Dat2 (right) of CPOI using unary encoding for various assignment policies. 79
5.9 Ranking of reassignment policies wrt their compression ratio for Elias-γ encoding (left) and unary encoding (right). 79
5.10 Total space for Dat1 (left) and Dat2 (right) of IC, CB, POI, for various d values 80
5.11 Total space for Dat1 (left) and Dat2 (right) without IC, for various d values 80
5.12 Nodes' space in GO. 81
5.13 Total storage space in GO with IC (left) and without (right). 81

5.14 Nodes' space in `Dat3`. 82
5.15 Total storage space in `Dat3` with `IC` (left) and without (right). 82

6.1 Store data using secondary memory 89
6.2 Pointers and storage graph kept in main memory (left) and inverted file structure kept in secondary memory (right) 90

Chapter 1

Introduction

1.1 Introduction to the Semantic Web

The semantic web [4] 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 process it more intelligently. It derives from W3C director Sir Tim Berners-Lee's vision of the Web as a universal medium for data, information, and knowledge exchange. Instead of just creating standard terms for concepts as is done in XML, the Semantic Web also allows users to provide formal definitions for the standard terms they create. Machines can then use inference algorithms to reason about the terms and to perform translations between different sets of terms. It is envisioned that the Semantic Web will enable more intelligent search, electronic personal assistants, more efficient e-commerce, and coordination of heterogeneous embedded systems. The proposed, by World Wide Web Consortium (W3C)[1], overview of the Semantic Web can be seen in Figure 1.1.

Ontologies as Conceptual Models on the Web

Ontologies are often seen as basic building blocks for the Semantic Web, as they provide a reusable piece of knowledge about a specific domain. A commonly used definition of the term (based on the original use of the term in philosophy) is that an ontology *is a specification of a shared conceptualization of a domain* [16]. Ontologies are often developed

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

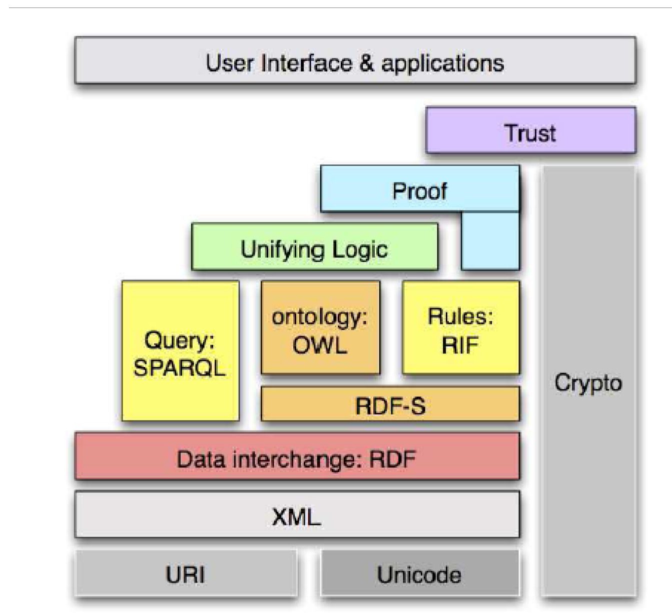


Figure 1.1: The Semantic Web architecture

by several persons and continue to evolve over time. Domain changes, adaptations to different tasks, or changes in the conceptualization require modifications of the ontology. This will likely cause incompatibilities in the applications and ontologies that refer to them, and will give wrong interpretations to data or make data inaccessible [26].

In order to form a real Semantic Web, it is necessary that the knowledge that is represented in the different versions of ontologies is interoperable. It is therefore important to create links between ontology versions that specify how the knowledge in the different versions of the ontologies is related. These links can be used to re-interpret data and knowledge under different versions of ontologies. Therefore, a versioning methodology is needed to handle revisions of ontologies and the impact on existing sources.

RDF Data Model and RDFS

Usually an ontology defines a hierarchy of classes of objects in the described domain and binary relations, called properties. The Resource Description Framework (RDF)[2] is a framework for representing such information in the Web. RDF has a simple data model that is easy for applications to process and manipulate. It is based upon the idea of

[2] http://www.w3.org/TR/PR-rdf-syntax/

making statements about resources in the form of subject-predicate-object expressions. These expressions are known as *triples* in RDF terminology. The subject denotes the resource (an RDF URI reference or a blank node[3]), and the predicate (an RDF URI reference) denotes traits or aspects of the resource and expresses a relationship between the subject and the object (an RDF URI reference, a literal or a blank node). A collection of such triples represents an RDF graph, that is independent of any specific serialization syntax.

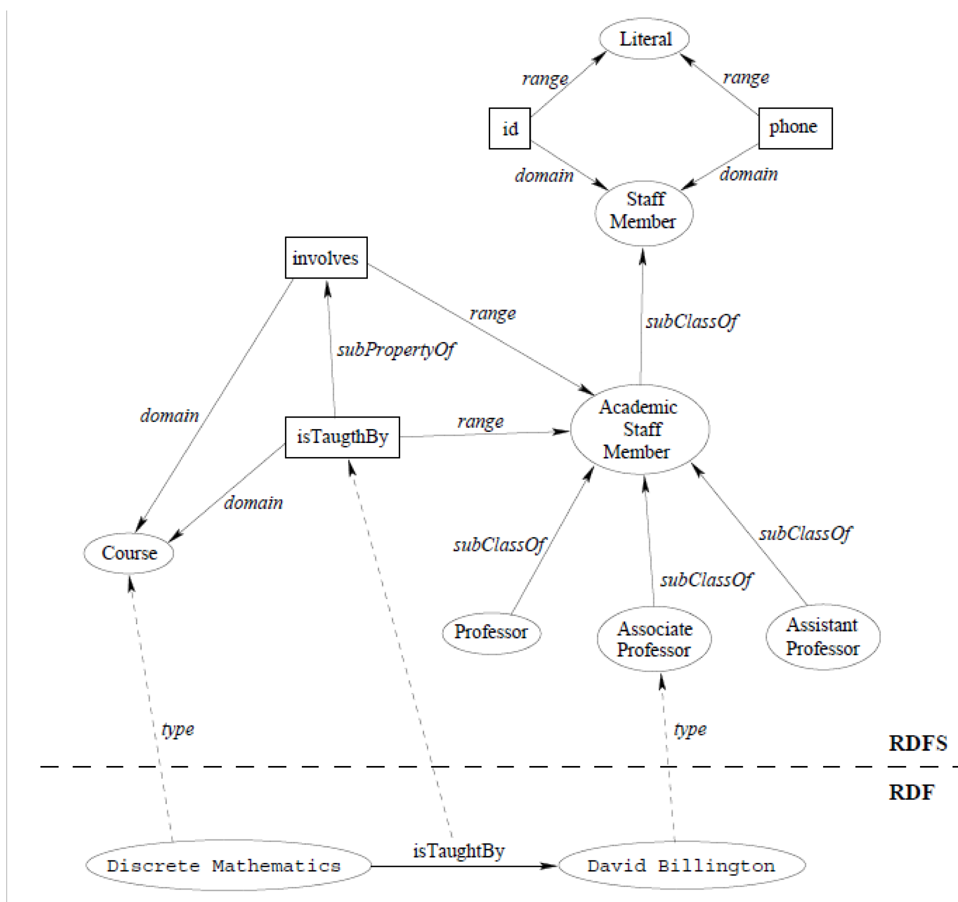


Figure 1.2: RDF and RDFS layers. Blocks are properties, ellipses above the dashed line are classes, ellipses below the dashed line are instances.

RDF Schema (RDFS)[4] is an extension to RDF. It is used to define application-specific classes and properties (a function that is not supported in RDF). Statements of RDFS make it possible to define hierarchies of classes, hierarchies of properties and to describe

[3]A non-universally identified object, whose URI could not be shared across different RDF descriptions
[4]http://www.w3.org/TR/rdf-schema/

domains and ranges of the properties. It also allows resources to be defined as instances of classes, and subclasses of classes. Figure 1.2 illustrates an example of RDF and RDFS layers (found in [2]).

1.2 Versioning Services

1.2.1 Versioning Services in the Software Development Area

Versioning is an extremely popular practice in the software development area. The term refers to *the creation and subsequent management of products that may be released a multiple number of times.* The main function of versioning is to track changes of a document or a product over time. The versioning system can keep information about who made a specific change and why, and allows removal of undesirable changes. Also, it helps users to know what the documents contents where at a specific point in time. Moreover, it can be used to achieve different versions of an important document.

Versioning is proved to be very helpful in software projects. It is extremely necessary for any collaborative and distributed development, because of its ability for tracking every change (allowing users to compare the current version of software with a previous one) and also for reversing the changes (as per the requirement). Another very important benefit of versioning is that, it helps in co-ordinating the efforts of contributing teams, which might be located far away from each other.

A number of alternative versioning systems have been developed, such as Concurrent Versions System (CVS) [3], in order to suite the needs of project teams. The technique of versioning has gradually become one of the most important tracking techniques in the field of software applications for all the above benefits that it offers.

1.2.2 Ontology Versioning

As ontology development becomes a more ubiquitous and collaborative process, developers face challenges in managing (and maintaining) multiple versions (of ontologies) akin to software development in large projects.

Ontology Versioning is defined as *the ability to handle changes in ontologies by creating*

and managing different variants of it. To achieve this ability, a methodology should provide methods to distinguish and recognize versions, procedures for updates and changes in ontologies, and an interpretation mechanism for effects of change. Focused on ontologies, we can say that a *versioning methodology* provides mechanism to disambiguate the interpretation of concepts for users of the ontology variants, and that it makes the effects of changes on different tasks explicit.

Traditional versioning systems in software development domain, like CVS [3], enable users to create and store versions, compare them, examine changes, and accept or reject changes, but they treat software code and text documents as *text files*. A versioning system for ontologies must compare the *structure* and *semantics* of the ontologies and not their textual serialization. Two ontologies can be exactly the same conceptually, but have very different text representations. For example, their storage syntax may be different (e.g. RDF/XML, N-Triples, Trix). Even in the same format, the order in which definitions appear in the text file may be different. We could canonicalize the ontology at the syntactic level (in order to use efficiently a traditional text-based versioning system). Moreover, a representation language may have several mechanisms for expressing the same semantic structure. Thus, text-file comparison is largely useless for ontologies.

1.2.2.1 Versioning for RDF Models

Several modern applications, e.g. in life sciences [20, 14], require the provision of versioning services over RDF datasets (either schema-free or schema-based) for various purposes:

- `Archiving`: Keeping records of the different states (versions) of an ontology (during ontology evolution process).

- `Preservation`: Both the long-term maintenance of the different versions and continued accessibility of their content (i.e. reconstruction capability of an arbitrary version at a later time).

- `Provenance`: The documentation of the creation and the derivation history of an ontology version.

For instance, failure to keep the previous states of scientific data (over which other experiments were based) jeopardizes scientific evidence and our ability to verify findings [8]. Furthermore, collaborative applications, e.g. in e-learning [40], require supporting several RDF datasets with parallel evolution tracks.

1.3 Contribution of this Thesis

As already mentioned in 1.2 one of the main functions of a version control system is the ability to store versions. At the simplest level, it could simply retain multiple copies of the different versions of the program, and label them appropriately. Another approach that a revision control system may follow is to keep tracks and account for ownership of changes to documents and code. The later seems to be very useful as it requires much less storage space and it also allows easily track of changes between versions.

However, both of the above approaches (even the later) can have high space requirements in order to store a vast amount of versions. We need an efficient method that guarantees significant space saving for storing large amount of data.

In this thesis our focus concentrates on providing efficient storage (and retrieval) services for Semantic Web Knowledge Bases (KBs). The contribution of this thesis lies in:

- Exploiting an indexing scheme for storing several versions (of RDF KBs).

- Proposing a compact representation for this indexing scheme based on gapped representation of triple identifiers and variable-length encodings. We consider each triple as a whole (i.e. we do not treat each triple's element separately).

- Evaluating analytically and experimentally the storage space requirements for this representation and comparing it with the storage space requirements of the main approaches that are followed in RDF domain.

1.3.1 Organization of this thesis

The rest of this thesis is organized as follows:

Chapter 2 describes the background and related work that have been done in ontology versioning and storage (focus on RDF models), as well as compression techniques that are popular in other domains and are adopted by our approach.

Chapter 3 describes an indexing scheme (called POI) for efficient storage and retrieval of several versions of ontologies.

Chapter 4 elaborates on the problem of further reducing the storage space of POI. The proposed techniques and the required storage space are evaluated analytically and experimentally.

Chapter 6 concludes and report possible ideas that are worth for further research.

Chapter 2

Background and Related Work

2.1 RDF/S Knowledge Bases

In general, a knowledge base[1] (for short KB) is a special kind of database for knowledge management, providing the means for the computerized collection, organization, and retrieval of knowledge.

We concentrate on RDF KBs, where the knowledge is represented in RDF. An RDF KB can be seen as a graph, whose nodes are entities (e.g., persons, companies, movies, locations) and whose edges are relationships (e.g., bornOnDate, livesIn , actedIn). In this graph are stored both RDF descriptions (instances) and RDF Schemata (class and property hierarchies), as they described in Section 1.1. In any case, a KB can be considered as a finite set of triples, that are either schema triples or instance triples.

2.2 Requirements and Key Aspects

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.

In general, the *functional* requirements for a versioning environment include:

- Create a new version by extending an existing one

[1] http://en.wikipedia.org/wiki/Knowledge_base

- 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
- Versions comparison, analysis of changes and description of their consequences (such as the exact compatibility problems that are introduced)

As far as the *performance* requirements, a version management system should provide high performance:

- in the *time* needed to create (resp. retrieve) a new (resp. existing) version
- in the *storage space* required for each version

Several non text-based tools that have been developed for ontology versioning focus on high level functions and they mainly overlook the performance requirements. In this work, we concentrate on the latter.

The related work presented in the following Sections pertains to four different domains of interest:

(i) `RDF Triple Storage`: Describes some popular RDF/S triple stores.

(ii) `Compact RDF Triple Representations`: Discusses systems and methods that aim at compressing the RDF triples of an RDF/S KB. These works are complementary to our work, as our approach aims to compress the total space needed for storing several versions but does not compress the set of distinct triple strings. So, one could adopt such techniques in order to compress the set of distinct triple strings, since they are managed independently of the versions' content and further reduce the total storage space.

(iii) `Versioning Approaches for Semantic Web`: Discusses systems that support versioning services for ontologies. Most of them store ontology versions using existing storage tools or RDF/S repositories and they mainly focus on high level functions.

(iv) `Compression Techniques in Information Retrieval`: Discusses compression techniques that are popular in the information retrieval area and they have shown to be very efficient. We examine the adoption of such techniques in our index.

2.3 RDF Triple Storage

This Section discusses some popular RDF/S triple stores.

2.3.1 Sesame

Sesame [7] is an open source[2] java framework for storing, querying and reasoning with RDF/S. It can be used as a database for RDF/S or as a java library for applications than need to work with RDF internally. It allows persistent storage of RDF data and schema information, and provides access methods to that information through export and query modules.

Sesame does not support versioning services. However, it can be used from other versioning tools at the bottom layer of their architecture as a storage container for RDF (e.g. we have seen that OMM follows such as approach).

Storage Space Perspective

An overview of Sesame's overall architecture is depicted in Figure 2.1.

Sesame's design and implementation are independent from any specific storage device. Thus, it can be deployed on top of a variety of storage devices, such as relational databases, triple stores or object-oriented databases, without having to change other functional modules.

In order to achieve this, The Storage And Inference Layer (SAIL API) is used, which is an internal API designed for storage and retrieval of RDF-based information. Its main design principles are that it should:

- define a basic interface for storing RDF/S and retrieving and deleting from (persistent) repositories

[2]http://sourceforge.net/projects/sesame/

- abstract from the storage format used (i.e. if the data are stored in a database, in memory or in files)
- be extendable to other RDF-based languages like DAML+OIL [21].

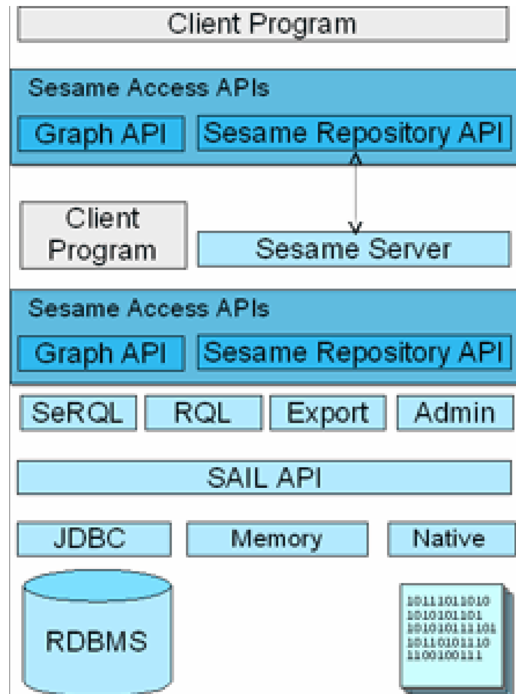


Figure 2.1: The Sesame Architecture

Each Sesame repository has its own SAIL object to represent it. The most important of these implementations of the SAIL API is the SQL92SAIL, which is a generic implementation for SQL92[3]. Recall that OMM implementation is based on this schema. It has been tested in use with several DBMSs, including PostgreSQL[4] and MySQL[5]. Figures 2.2, 2.3 depict the above database schemata respectively.

2.3.2 Jena2

Jena2 [44] is an open source[6] Semantic Web framework for Java. It provides an API to extract data from and write to RDF graphs. The graphs are represented as an

[3]http://www.andrew.cmu.edu/user/shadow/sql/sql1992.txt
[4]http://www.postgresql.org/
[5]http://www.mysql.com
[6]http://sourceforge.net/projects/jena/files/Jena/

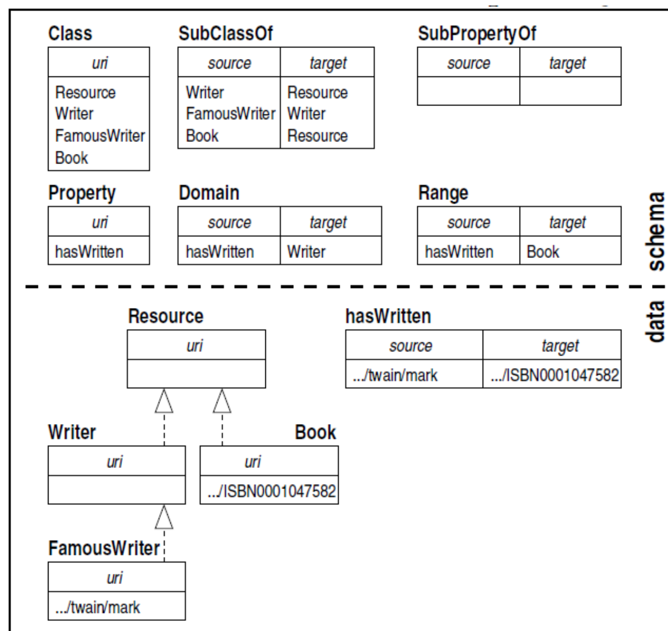


Figure 2.2: The object-relational schema used with PostgreSQL

abstract "model". A model can be sourced with data from files, databases, URLs or a combination of these. A Model can also be queried through SPARQL query engine. Jena also includes a rule-based inference engine. Moreover, it provides support for OWL through an OWL API. Jena supports serialisation of RDF graphs to RDF/XML, N3, N-Triples and relational database.

Storage Space Perspective

For persistent storage of RDF graphs a database engine is used (e.g. Postgresql, MySQL, Oraqle). The database schema of Jena2 implementation consists of a statement table, a literals table and a resources table as it is shown in Figure 2.4.

2.3.3 3store

3store[18] is an open source[7] RDF *triple store* developed within the Advanced Knowledge Technologies (AKT) project[8] of the University of Southampton. It offers efficient handling of large RDF knowledge bases (it has the ability to handle at least 20 million

[7]http://sourceforge.net/projects/threestore/
[8]http://www.aktors.org/technologies/3store/

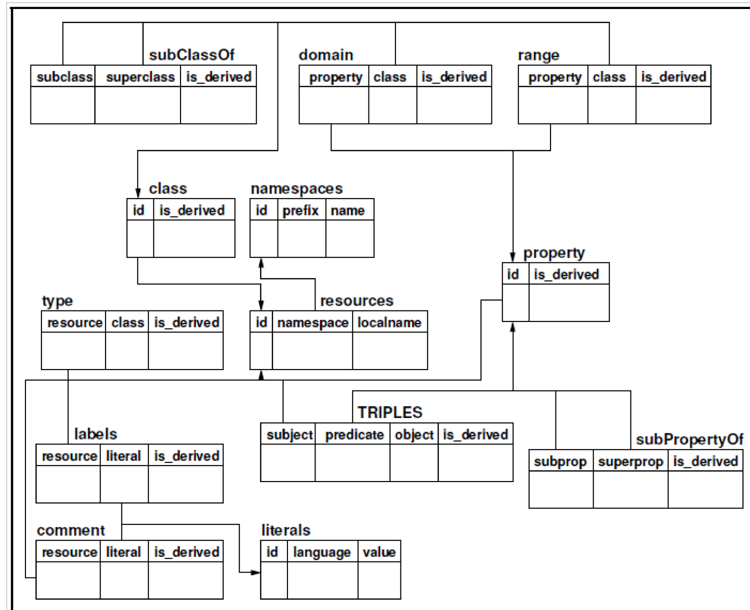


Figure 2.3: The relational schema used with MySQL

triples and 5000 classes and properties) as well as processing of queries and generation of RDF entailments. 3store is written in C and uses MySQL to store its raw RDF data and caches. It provides access to the RDF data via RDQL or SPARQL over HTTP, on the command line or via a C API.

Concisely, 3store main utilities include:

`import`: Takes RDF files, parses them and inserts the triples into the database.

`rebuild taxonomy`: Infers triples based on the schemas loaded via import.

`info`: Supplies summary information about the database, including numbers of files and triples present.

`RDQL`: Query engine.

Storage Space Perspective

3store uses MySQL database to store RDF data and information. The arrangement of tables in database schema is shown in Figure 2.5

Statement Table

Subject	Predicate	Object
mylib:doc1	dc:title	Jena2
mylib:doc1	dc:creator	HP Labs - Bristol
mylib:doc1	dc:creator	Hewlett-Packard
mylib:doc1	dc:description	101
201	dc:title	Jena2 Persistence
201	dc:publisher	com.hp/HPLaboratories

Literals Table

Id	Value
101	The description - a very long literal that might be stored as a blob.

Resources Table

Id	URI
201	hp:aResource-WithAnExtreme-lyLongURI

Figure 2.4: Jena2 Database Schema

2.3.4 YARS (Yet Another RDF Store)

YARS [19] is a system for persistent storage and retrieval of RDF data. It is a lightweight open-source Java implementation and provides an efficient RDF indexing structure. It defines and realizes a complete index structure including full-text indices for RDF triples with context in order to support fast storage and retrieval of large amount of RDF data (in the order of billions of triples). Persistency is supported through the use of B+ trees. Internally, each RDF triple is extended with context (i.e., the source of the RDF triple) and stored as quadruples (Subject, Predicate, Object, Context) encoded in N3[9] format.

Storage Space Perspective

YARS in order to implement the index organization uses JDBM[10], a lightweight open-source library that includes implementation of B+ trees for persistent storage of data on disc.

2.3.5 Discussion on Database Representations

Most of the above tools use a database approach in order to store their RDF/S data and information. There are several ways to represent these information in a database

[9] http://www.w3.org/DesignIssues/Notation3.html
[10] http://jdbm.sourceforge.net/

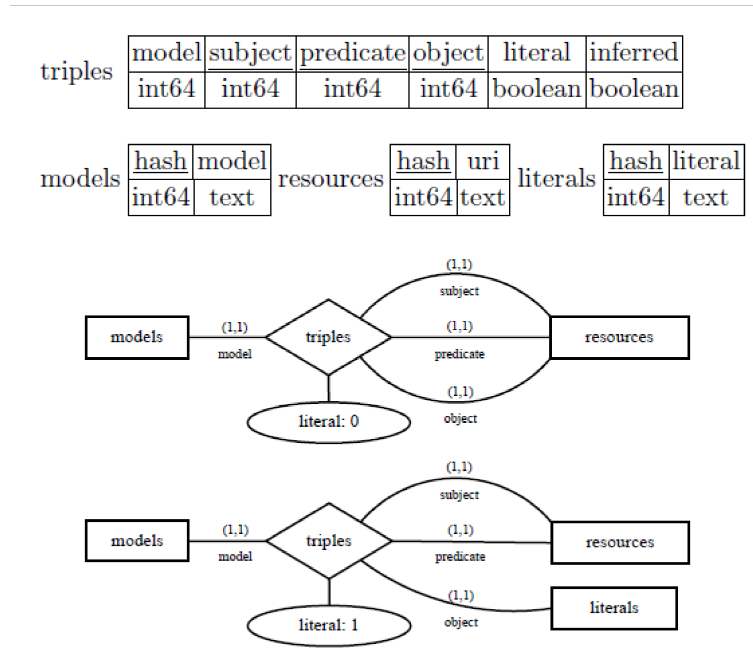


Figure 2.5: Database schema and ER diagram showing table relationships

schema. In [38] database represantations of RDF/S schemata and data are classified into three popular groups:

a. *a schema-aware*: one table per RDF/S schema property or class is used, with explicit or implicit storage of subsumption relationships (Figure 2.6).

b. *a schema-oblivious*: a single table with triples of the form ⟨subject-predicate-object⟩ is used, using integer identifiers (IDs) or not (using URIs themselves) to represent resources (Figure 2.7).

c. *a schema-hybrid*: one table per RDF/S meta-class is created, namely, for class and property instances with different range values (i.e., resource, string, integer, etc.). Figure 2.8 depicts Hybrid representation.

Table 2.3.5 summarizes the storage schemes implemented by existing RDF/S stores.

Comparing these database representations with our approach, we could correlate our approach with schema-oblivious representation, since we treat each triple as a whole.

It is very important to note, at this point, that all the above tools adopt a database implementation to store their data because they focus on executing queries fast and

Schema-aware	Schema-oblivious
Sesame	*Sesame*
	Jena2
	3store

Table 2.1: RDF/S storage schemes

efficiently over those data. They mainly overlook the storage space needed to store them, as they presume that there is plenty of secondary memory space.

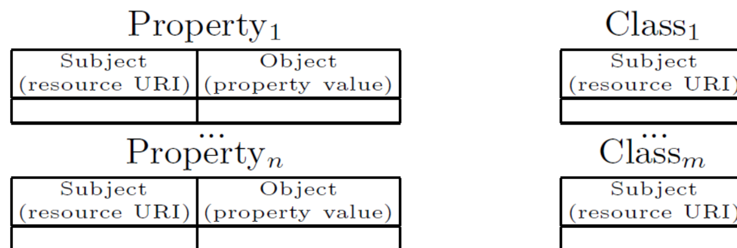


Figure 2.6: Schema-aware representation

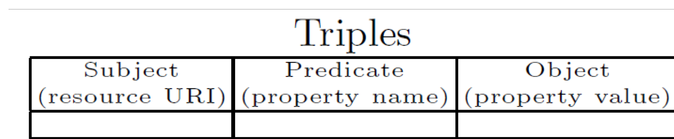


Figure 2.7: Schema-oblivious representation

2.4 Compact RDF Triple Representations

This Section discusses systems and methods that aim at compressing the RDF triples of an RDF/S Knowledge Base. They propose solutions for storing RDF triples for indexing and querying or publishing and exchange.

2.4.1 RDF-3X

RDF-3X [29] engine is an architecture for executing SPARQL queries over large repositories of RDF triples. This work aims at providing not only a powerful and fast query

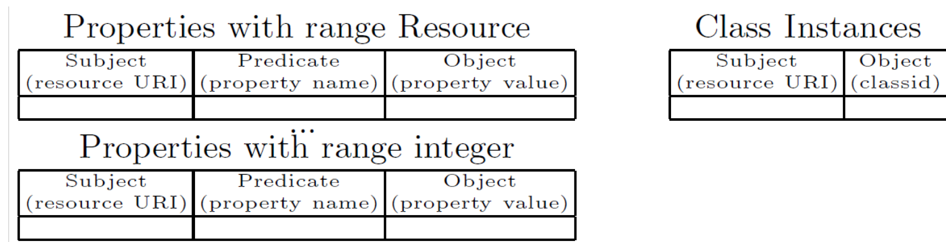


Figure 2.8: Schema-hybrid representation

processor, but also an efficient RDF triple storage.

In that architecture all triples are stored (sorted lexicographically) in a clustered B^+-tree, but instead of storing the values of each triple (S, P, O), all literals are replaced by ids using a *mapping dictionary*. Several indexes are built in order to cover all 6 permutations of the three dimensions of a triple and also all permutations of six binary and three unary projections. For the projection indexes, the missing component(s) (S, P, or O) are replaced by count aggregates, for fast statistical lookups. This *exhaustive-indexing* approach is illustrated in the left part of Figure 2.9[11]. Moreover, based on the observation that neighboring triples are very similar (as the triples are sorted lexicographically), a compression scheme for triple ids is adopted, storing only the changes between them (inspired by methods used in IR). Byte-wise compression is used as it seems to perform better than bit-wise compression for their dataset. Figure 2.10 depicts the structure of a compressed triple. The delta for each value is computed, and then the minimum number of bytes is used to encode just the delta. A header byte denotes the number of bytes used by the following values. Each value consumes between 0 bytes (unchanged) and 4 bytes (delta needs the full 4 bytes), which means that there are 5 possible sizes per value. For three values these are 5 * 5 * 5 = 125 different size combinations, which fits into the payload of the header byte. The remaining gap bit is used to indicate a small gap: When only value3 changes, and the delta is less than 128, it can be directly included in the payload of the header byte. This kind of small delta is very common, and can be encoded by a single byte. The details of the compression are shown in the right part of Figure 2.9. The algorithm computes the delta to the previous tuple. If it is small it is directly encoded in the header byte, otherwise it computes the δ_i values for each of the tree values

[11] All Figures quoted in this Section are taken from the respective paper of each work.

and calls encode. encode writes the header byte with the size information and then writes the non-zero tail of the δ_i (i.e., it writes δ_i byte-wise but skips leading zero bytes). This results in compressed tuples with varying sizes, but during decompression the sizes can be reconstructed easily from the header byte. As all operations are byte-wise, decompression involves only a few cheap operations and is very fast.

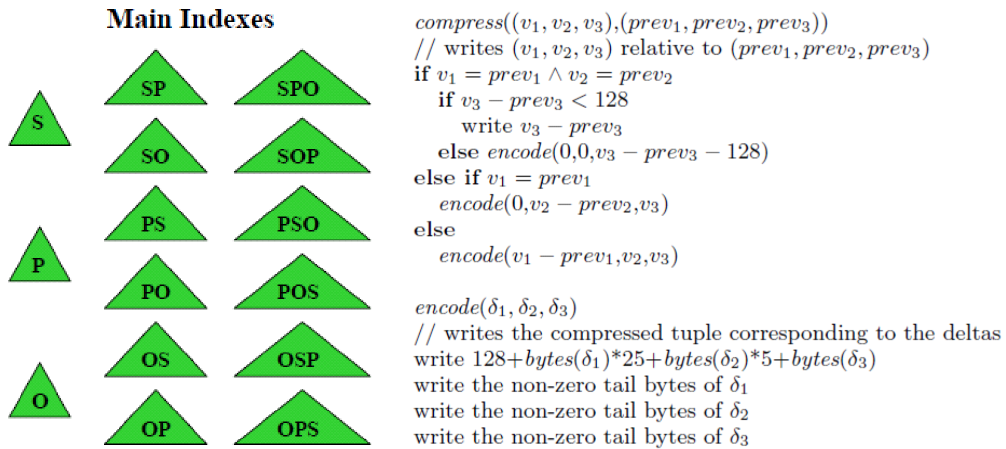


Figure 2.9: Differential indexes in RDF-3x (left) and Triple compression algorithm (right)

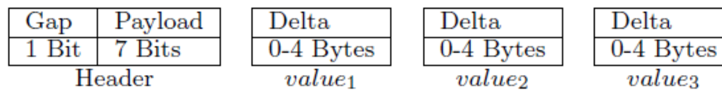


Figure 2.10: Structure of a compressed triple

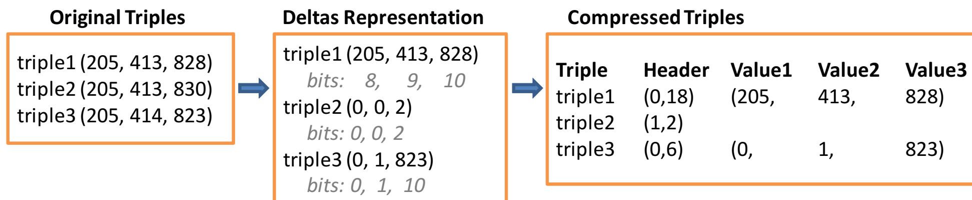


Figure 2.11: Triples compression through an example

Figure 2.11 illustrates an example of compressing three triples. All literals are replaced with their triple ids. The first triple remains the same, consuming 6 bytes (as many bytes

as needed to represent its values). The second triple requires only 1 byte, as only the third value changes and the delta can be included in the header byte. Lastly, the third triple requires 4 bytes for its representation.

On all experiments that were performed, the total storage space for all indexes together was less than the size of the original data (1.5 times less than the size of storing data in RDF (XML) format and 3 times less than the size of storing data in triple form).

2.4.2 HTD representation

In this work a compact representation format for RDF data is presented [13], aiming at compressing RDF data and providing indexed access to the RDF graph. This representation decomposes an RDF data source into three main parts: Header, Dictionary and Triples. It offers space savings and allows on-demand indexed access to the graph.

It is based on three main components:

- A *header*, including logical and physical metadata describing the RDF data set. This serves as an entrance point to the information on the data set. It is kept in plain form as it should always be available to any receiving agent for processing.
- A *dictionary*, organizing all the identifiers in the RDF graph. This provides a catalog of the information entities in the RDF graph with high levels of compression. *PPM*[10] is used to encode the dictionary.
- A set of *triples*, which comprises the pure structure of the underlying RDF graph while avoiding the noise produced by long labels and repetitions.

In order to obtain an HDT representation of an RDF graph, firstly the basic RDF features necessary to build the dictionary and the underlying graph are extracted, and then some practical decisions are covered in order to have the HDT concrete implementation for publication and exchange of RDF. This process is depicted in Figure 2.12.

The sets of subjects, predicates and objects in RDF are not disjoint. In order to assign shorter IDs, four sets are distinguished: S-O (common subject-objects), S (the non common subjects), O (the non common objects), P (the predicates). Figure 2.13 depicts an example of these four sets within a dictionary building process.

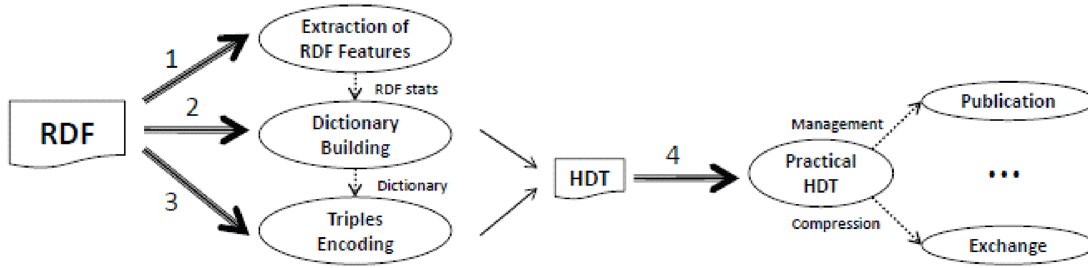


Figure 2.12: Construction of the HDT format from a set of triples

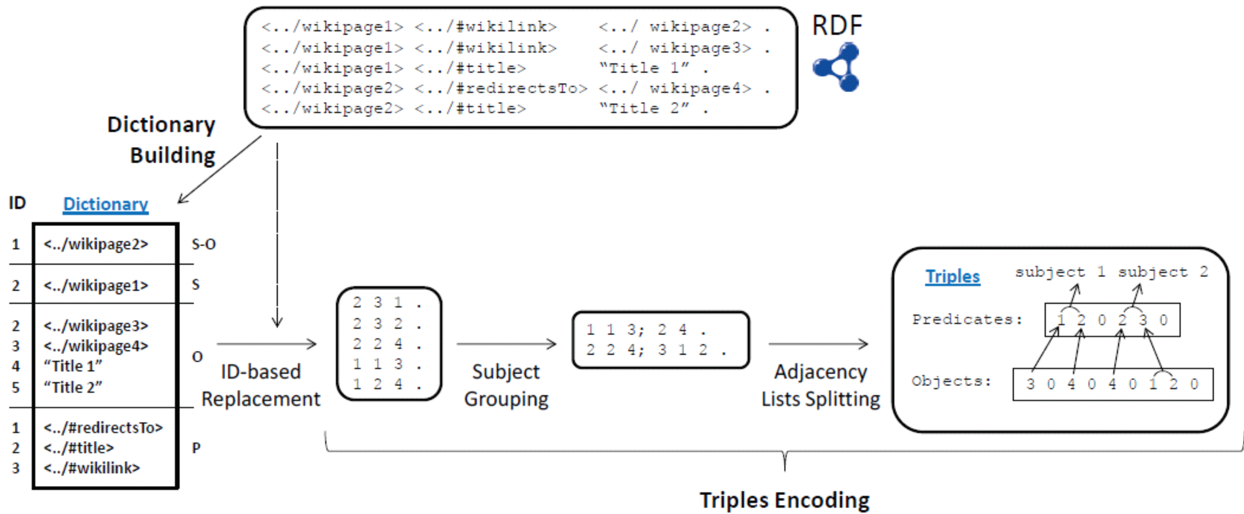


Figure 2.13: Incremental representation of an RDF data set with HDT

The Dictionary component assigns a unique ID to each element in the data set. An original RDF triple can now be expressed by three IDs, replacing each element in triples with the reference in the dictionary (ID-based replacement).

The Triples component performs a subject ordered grouping, that is, triples are reorganized in adjacency lists,[12] in sequential order of subject IDs. Due to this order, an immediate saving can be achieved by omitting the subject representation, as it is known the first list corresponds to the first subject, the second list to the following, and so on. In

[12] Adjacency List is a compact data structure that facilitates managing and searching. For example, the set of triples: $(s, p_1, o_{11}), ..., (s, p_1, o_{1n_1}), (s, p_2, o_{21}), ..., (s, p_2, o_{2n_2}), ..., (s, p_k, o_{kn_k})$ can be written as the adjacency list: $s \rightarrow [(p_1, (o_{11}, ..., o_{1n_1}), (p_2, (o_{21}, ..., o_{2n_2})), ..., (p_k, (o_{kn_k}))]$.

the notation above, all the data is represented by one stream, in which the list of objects associated with a subject (s) and a predicate (p) is represented just after the p. Instead, this representation is splited into two coordinated streams of Predicates and Objects. The first stream of Predicates corresponds to the lists of predicates associated with subjects, maintaining the implicit grouping order. The end of a list of predicates implies a change of subject, and must be marked with a separator, e.g. the non-assigned zero ID. The second stream (Objects) groups the lists of objects for each pair (s, p). These pairs are formed by the subjects (implicit and sequential), and coordinated predicates following the order of the first stream. In this case, the end of a list of objects (also marked in the stream) implies a change of (s, p) pair, moving forward in the first stream processing.

2.4.3 Discussion

Both of the above approaches ([29, 13]) aim at compressing the RDF triples by using ids for each (subject-predicate-object) element of the triples, and providing indexes that can be exploited for query processing. These works are complementary to our work, since we treat each triple as a whole, and we do not compress the set of distinct triple strings. Specifically, one could adopt such techniques in a versioning framework in order to compress the set of distinct triple strings, if they are managed independently of the versions' content, and to further reduce the overall occupied space.

2.5 Versioning Approaches for Semantic Web

This Section discusses other systems and tools that support versioning services for ontologies. Most of them store ontology versions using existing storage tools or RDF/S repositories. Some of the latter are described in the next Section as well.

2.5.1 x-RDF-3X

The work presented in [30] presents an extension of the RDF-3x system [29] that supports versioning and some other services as time-travel access and transactions on RDF databases. Versioning is achieved by maintaining versions of individual triples (updates are considered as pairs of insertions and deletions, so two timestamps fields are used, the

created and *deleted* to denote the life of each triple version). The [*created*, *deleted*] interval is the lifespan of the triple version, where *deleted* has a null value for versions that are presently alive. The database state of a given point in time t can be reconstructed by returning all triples for which t falls into the corresponding lifespan interval.

Ideally, timestamps reflect the commit order of transactions, but unfortunately the commit order is not known when inserting new data. In order to cope with this problem each transaction is assigned a write timestamp once it starts updating the differential index, and this timestamp is then used for all subsequent operations. Ideally the migration is performed at transaction commit only, which means that the timestamps perfectly reflect the commit order and need no further updates. Moreover, a *transaction inventory* is used, that tracks transaction ids, their begin and commit times (BOT and EOT), the version number used for each transaction, and the largest version number of all committed transactions (highCV #) at the commit time of a transaction (Figure 2.14). This inventory serves to efficiently decide if a transaction committed before another one. Also, it relates the relative time of transaction ordering and version numbering to wall clock times. This is needed for supporting time-travel queries and snapshot isolation.

transId	version #	BOT	EOT	highCV#
T_{101}	100	2009-03-20 16:51:12	2009-03-20 16:55:01	300
T_{102}	200	2009-03-20 16:53:25	2009-03-20 16:54:15	200
T_{103}	300	2009-03-20 16:54:01	2009-03-20 16:54:42	300
...	...	...	...	...

Figure 2.14: The transaction inventory that keeps all transactions info

Regarding the experimental evaluation synthetic workloads were constructed based on real data (from the LibraryThing book-tagging web site) and the x-RDF-3x system was compared with two other systems: PostgreSQL and Jena. The results show that the first system was 3 times more space efficient than the others systems.

2.5.2 Versioning Model for Biomedical Ontology Versions

In [23] a database approach is proposed for managing versions of biomedical ontologies. It aims at providing management of many versions of large biomedical ontologies and efficient storage of them. That approach maintains a life time (with two additional fields

as the x-RDF-3x system) for all ontology concepts, relationships and attribute values.

The version model has been realized within a generic database repository to store the versions of different ontologies and to support uniform version access. Figure 2.15 depicts the relational repository schema for the versioning model. It consists of different entities to consistently represent ontologies, ontology versions, concepts and the ontology structure (Ontology Relationships). The latter entity represents the set of relationships; each of them is associated with a relationship type (e.g. is-a, part-of). To flexibly store attribute values for each ontology concept the entity-attribute-value concept has been applied. In this way, the entity Attributes holds the set of attributes describing the semantics of the associated attribute values (entity Element Attribute Values) of an ontology concept. Each ontology concept, relationship, and attribute value is associated with a life time represented by the fields *date-from* and *date-to*. Using these time periods, any ontology version can be completely reconstructed by taking the *version date* attribute (entity Ontology Versions) into account, such that it holds $date - from \leq version\, date \leq date - to$. Actually, one can determine the snapshot of an ontology for any point in time after the creation date.

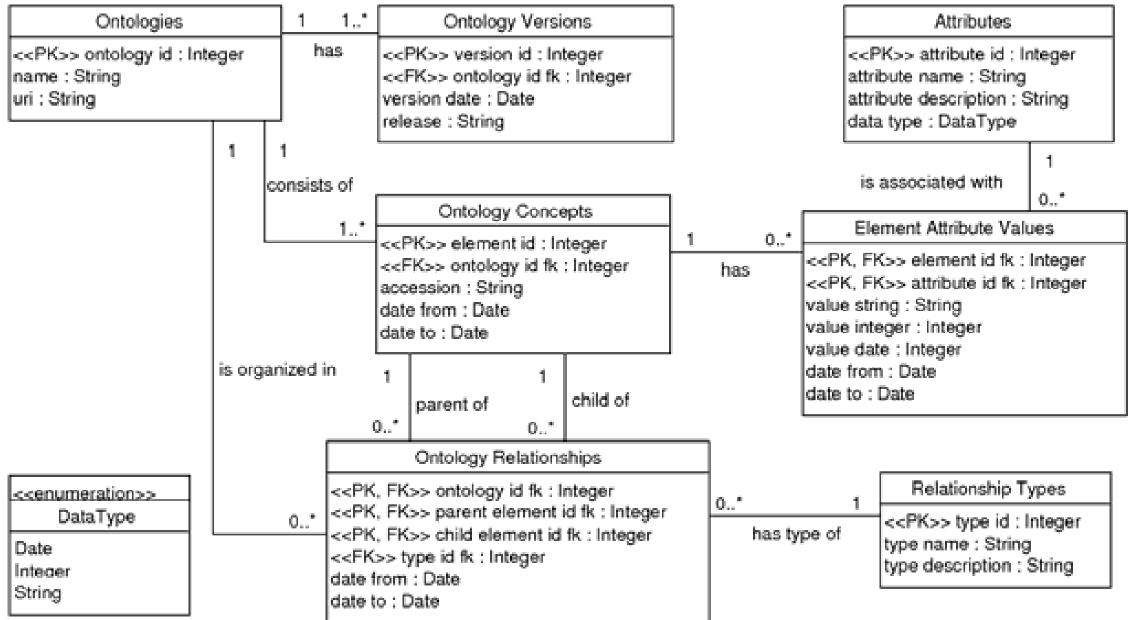


Figure 2.15: The relational repository schema for the versioning model

Regarding the experimental evaluation 62 versions of the Gene Ontology has been tested and compared the storage requirements with the `IC` method. However no measured comparative results are reported (only estimations regarding the compression achieved).

2.5.3 OntoView

OntoView [27] is a web-based system that helps users to manage changes in RDF-based ontologies. It provides a transparent interface to different versions of ontologies, by maintaining not only the transformations between them, but also the conceptual relations between concepts in different versions. It allows user to specify the conceptual implication of each difference between subsequent versions and to export the differences as *adaptations* or *transformations*.

One of the main features of OntoView is the ability to compare ontologies. Two terms are used to distinguish changes:

`Conceptual change`: a change in the definition of a concept or property that affects its formal semantics.

`Explication change`: a change in the definition of a concept, without changing the concept itself. For example, attaching a slot "fuel-type" to a Class "Car", is a change that describes the Class more extensively, does not affect the conceptualization. OntoView compares versions of ontologies at a *structural* level, showing which definitions of ontological concepts or properties are changed. An example of such a graphical comparison of two versions of a DAML+OIL ontology is depicted in Figure 2.16. In this web view, the user can characterize each difference, as described previously. The comparison function of OntoView uses the actual *diff* tool of CVS. This is a line-based tool, that highlight the lines that textually differ in two versions. So, in order to achieve *structural* comparison, the ontology is canonicalized at the syntactic level before being given to the *diff* tool.

OntoView provides some basic support for the analysis of the effects of changes. First, on request it can also highlight the places in the ontology where conceptually changed concepts or properties are used. For example, if a property "hasChild" is changed, it will highlight the definition of the class "Mother", which uses the property "hasChild".

Moreover, OntoView reads in changes and ontologies via several methods. Specifically, ontologies can be read in as a whole, either by providing a URL or by uploading them

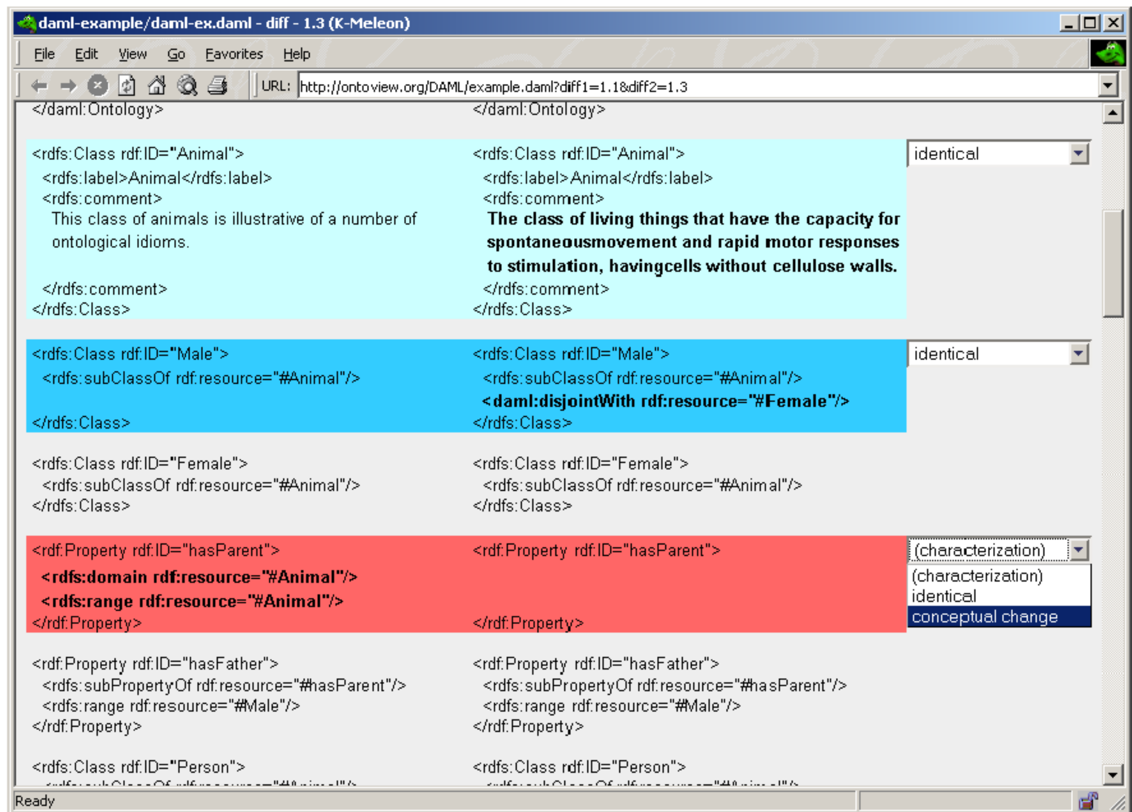


Figure 2.16: Comparing two ontologies in Ontoview

to the system. The user has to specify if 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.

Identification of versions of ontologies is a very important issue. OntoView supports persistent and unique identification of web-based ontologies. It uses the namespace mechanism with URIs for ontology identification, separated from the location of the ontology file. Depending on the compatibility effects of the type of change, it assigns a new identifier or it keeps the previous one (if the new version has non-conceptual changes).

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 adapters for data sources or other ontologies.

Storage Space Perspective

OntoView supports ontologies' storage and provides a transparent interface to arbitrary versions of ontologies. It stores the contents of the versions and also keeps track of the meta-data (i.e. author, date and intention of the update), the conceptual relations between constructs in the ontologies (e.g. specified by equivalence relations or subsumption relations) and the transformations between them (e.g. change of a restriction on a property, addition of a class, removal of a property). It is based on CVS, and its underlying, syntactic-level, *diff* tool. CVS is widely used in software development to allow collaborative development of source code. It uses *delta compression*[13] for efficient storage of different versions of the same file.

2.5.4 SemVersion

SemVersion [43] is a Java library for providing versioning facilities for RDF models and RDF-based ontology languages. It is inspired by CVS.

SemVersion provides an easy to use API. The user can commit a new version either by providing the complete contents of the version or a *diff*, that is, the *change* that is to be applied on a preexisting version to create the new one. Every version is decorated with metadata (like parents, has a branch, a label and a provenance URI).

One of its main functions is the computation of *diffs*. It provides both structural (set-theoretic difference of two RDF triple sets) and *semantic* diff (including into the triple sets all inferred triples).

Identification of versions is also provided, by generating globally unique URIs for every version.

SemVersion also handles the problems created in computing the structural diff caused by blank nodes. Blank nodes cannot be globally identified, as they lack a URI, so there is

[13] *Delta compression* is a way of storing (or transmitting) data in the form of differences between sequential data rather than complete files. The differences are recorded in discrete files called "deltas" or "diffs".

no knowledge about the relation between two blank nodes from different models. The RDF semantics dictates to treat them as different. As a result, the diff between a model and itself is not empty, if it contains blank nodes. To overcome this, SemVersion introduces the concept of *blank node enrichment*. This technique adds a property to the blank nodes leading to a URI, which is globally unique. This URI makes blank nodes globally addressable, while they remain formally blank nodes in the RDF model. So, the content of every version is blank node enriched before it is stored in the RDF storage layer.

Moreover SemVersion supports branch and merge operations. Also, there is some primitive support for reporting conflicts, but not resolving them programmatically.

Storage Space Perspective

The high-level architecture of the implementation consists of several layers (Figure 2.17). Each layer depends on the layer below. To users only the SemVersion API and the RDF2GO API are exposed. The layers are:

- `SemVersion API` - the versioning API to be used by users;
- `RDFREACTOR` - a framework for domain specific object-oriented RDF access in Java;
- `RDF2GO` - an abstraction over triple (and quad) stores, such as Jena[44], Sesame[7], YARS[19], etc.
- `YARS` - a scalable quad store.

SemVersion stores each version of an RDF model as unique independent graph than contain the whole model. Even if the user provides a *diff* in order to create a new version, then these changes are applied to the model (all the identified triples are added or removed) and the new version is stored independently. The architecture of SemVersion is based on RDF2GO and RDFREACTOR, as we have already mentioned. Both choices helped in keeping the programming flexible and fast, so the triple store can be easily switched from one to another. All these storage alternative choices of Jena, Sesame and YARS are described in the next Section.

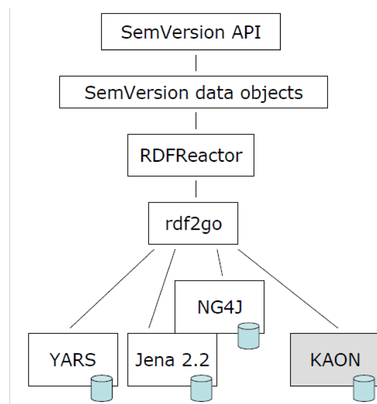


Figure 2.17: The Layered Architecture of SemVersion

2.5.5 Ontology Middleware Module

Ontology Middleware Module (OMM) [24] is an open source[14], knowledge management system. It allows knowledge representation and management in RDF/S and languages structurally compatible with it, such as, DAML+OIL and OWL. OMM is an extension of Sesame [7] repository enriching it with support of versioning, meta-information, access control (security) and Description Logic (DL) reasoning.

As far as versioning services, OMM provides tracking of the changes in the knowledge (on structural level). The history of changes in the repository could be defined as sequence of states, as well as a sequence of updates, because there is always an update that turned repository from one state to the next one. Some of these states could be pointed out as versions (that is a user's or application's decision). So, versions are labelled states of the repository.

OMM supports meta-information for resources, statements and versions. Also, in order to resolve problems that caused by the inference of triples - for example, an addition of a statement could lead to the appearance of more statements, but what should be done if later we delete the first statement? Deletion of the inferred statements too is not always desirable because of dependencies that may have been created in the meantime with others new statements - OMM uses an approach called Truth Maintenance System (TMS). In essence, for each statement (or at least for the inferred ones) information is being kept

[14]http://www.ontotext.com/omm/downloads.html

about the statements or expressions that "support" it, i.e. such that lead to its inference.

Storage Space Perspective

The implementation of OMM is based on Sesame's database schema (shown in Figure 2.18), extending it appropriately to support the tracking of the changes and it is depicted in Figure 2.19. In table *TRIPLES_HIST*, the columns *BornAt* and *DiedAt* indicate the states of the repository (versions). Specifically, these columns contain a *UID*, that is an update identifier whose value determines the "lifetime" of the statement. In table *UPDATES* all the relevant information to each update is kept (i.e. the time when it happened and the user who performed the update).

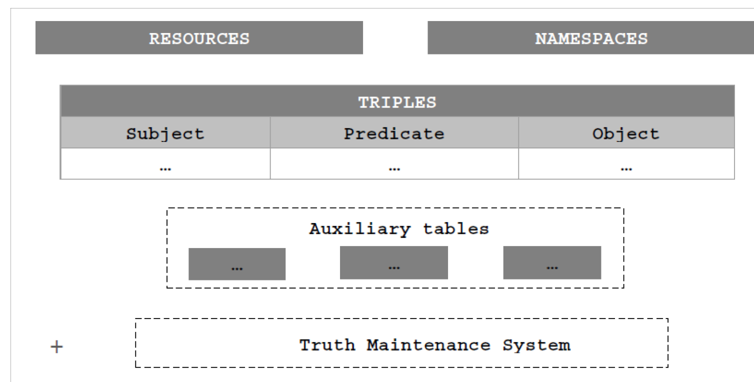


Figure 2.18: The database schema used from the SQL92SAIL of Sesame

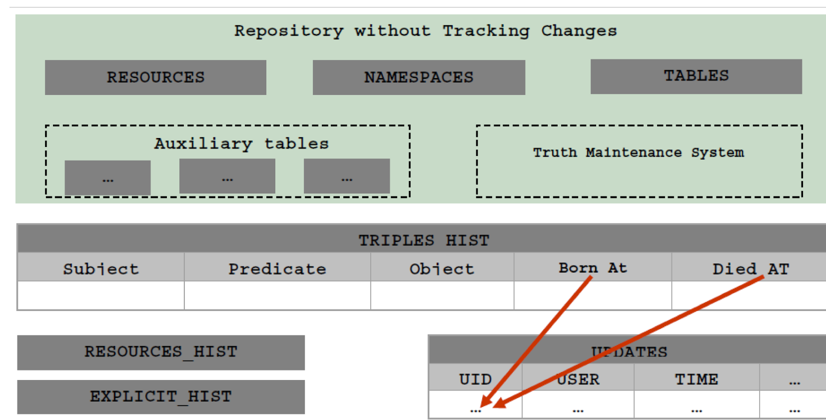


Figure 2.19: The database schema implemented of Ontology Middleware as extensions of Sesame

For example, suppose that we add a triple $t1$ in version $v1$ (UID=1), and then we delete it in version $v2$ (UID=2), and then we readd it in version $v3$ (UID=3) and last we delete it in version $v4$ (UID=4). The $TRIPLES_HIST$ table will be as follows (for brevity's sake we have used $TripleId$ column instead of $Subject$, $Predicate$, $Object$ columns):

$TripleId$	$BornAt$	$DiedAt$
$t1$	1	2
$t1$	3	4

The main idea of this implementation is that the original tables (of the database schema of Sesame) remain unchanged and additional tables are used to preserve the tracking information. When a user needs to be able to work with a previous version, the appropriate state is branched into a separate auxiliary repository which is read-only and has no support for tracking changes. This schema has the following advantages and disadvantages:

+ working with the current version is as fast as without tracking, the implementation is much simpler

+ working with old version is as fast as working with the current one - the same optimizations work

- some information is duplicated

- to start work with old version a dummy branch is created, which takes both considerable time and space

2.5.6 Comparison

Discussion on Storage Space Requirement

Most of the tools 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). In general, we could identify the following approaches:

- `Independent Copies (IC)`: Every new version is stored independently. This approach is followed by SemVersion and OntoView. It is also adopted in [31]. It

guarantees that the retrieval of an arbitrary version is prompt, but the obvious drawback of this approach is the excessive storage space requirements.

- `Change Based (CB)`: Only deltas between two subsequent versions are stored. This approach adopted in [3, 39]. To construct the contents of a particular version one has to execute a (potentially long) sequence of deltas (which could be computationally expensive). Related work for the Semantic Web includes [46, 41]. [30, 23, 24] falls into this category as it keeps tracks of the changes. The only difference with this implementation is that it is faster to retrieve the contents of a particular version (one query need to be executed).

2.6 Compression Techniques in IR

This section discusses compression techniques that are popular in the information retrieval (IR) area. In IR, the adoption of gapped document identifiers and identifier encoding schemes saves space in an inverted index because there are words that occur in many documents (in general because the distribution of words in documents tends to follow an inverse power-law [47]). Specifically, if the documents that share a lot of common words get close identifiers, and we adopt a *gapped representation* for their identifiers using an encoding scheme that represents small integers with a small number of bits, then the postings lists of these common words will have small bit representation. To ensure that documents with a lot of common words get close identifiers, the documents of the corpus are clustered, so that documents that share a lot of common words are placed in the same cluster. Subsequently documents are given identifiers on a cluster basis.

Experiments in the IR domain (specifically in the TREC web data [33]) have shown that the *compression ratio* of the compressed index is around 30%-45%. In more detail, there are two main issues concerning this problem: (a) the assignment technique for documents ids and (b) the encoding that will be used. Regarding the former, many approaches have been proposed for the reassignment of documents ids [35, 36, 34, 6, 5], aiming at improving the performance of compression by achieving as small gaps as possible.

Most of these works propose algorithms for the reassignment problem, based on the

fact that an efficient ordering can be derived from *clustering*. So, these algorithms first cluster the collection of documents and then assign the new ids considering the way that documents are grouped within the clusters.

2.6.1 Document identifier assignment according to the lexicographical ordering of the URLs

In [35] a very simple and effective ordering is proposed: assigning identifiers to documents according to the lexicographical ordering of the URLs. This is based on the hypothesis that "documents containing correlated terms are very likely to be hosted by the same site thus will also share a large prefix of their URLs".

The algorithm just consists of sorting the list of URLs, yet very efficient, since it does not require any set intersection operations. The results have shown that this technique is very efficient and fast. It achieves better compression ratios than the clustering techniques and the time needed to compute the assignment is two orders of magnitude smaller than the time needed by clustering. Furthermore, the URL-ordering method is more scalable to collections of even billion of URLs.

2.6.2 Document reordering using B&B algorithm

In [6] approach a weighted similarity graph is constructed, in which a vertex represents a document and an edge between two vertices represents their similarity. After that, a reordering of the documents ids is performed according to the *B&B* algorithm recursively splits the graph into smaller subgraphs, representing smaller subsets of the collection, until all subgraphs become a singleton. Thereafter, the docIds are reassigned according to a *depth-first* traversal of the resulting tree.

2.6.3 Document identifier reassignment using TSP algorithm

In [34] a weighted similarity graph is constructed as before. Thereafter, the *Traveling Salesman Problem* (TSP) heuristic algorithm is used to find a cycle in the similarity graph having maximal weight and traversing each vertex exactly once. The suboptimal cycle found is finally broken at some point and the docIds are reassigned to the documents according to the ordering established. The rationale is that since the cycle preferably

traverses edges connecting documents sharing a lot of terms, if we assign close ids to these documents, we should expect a reduction in the average value of d-gaps and thus in the size of the compressed inverted file index. Unfortunately, the above approaches are too expensive in both time and space, and that makes them unfeasible for large collections.

2.6.4 Assigning identifiers to documents on the fly

The work presented in [36] proposes four assignment techniques, based on clustering, which assign ids to document *on the fly*, i.e. during the inversion of the document collection. To compute efficiently an effective assignment, they propose the *Transactional Model*, which is based on the popular *bag-of-words* model and allows the assignment algorithm to be placed into the typical spidering-indexing life cycle of a WSE. The proposed algorithms can be used to assign docIds while the spidered collection is being processed by the Indexer. They follow two opposite strategies: in the first approach the algorithms recursively split the collection in a way that minimizes the distance of lexicographically closed documents (top-down), while in the second approach (bottom-up), the algorithms compute an effective reordering using variants of the popular *k-means* clustering algorithm. When the index will be actually committed on the disk, the new docId assignment will be already computed. Instead, the other methods proposed so far require that the inverted file has already been computed in advance.

2.6.5 Document identifier reassignment using dimensionality reduction with SVD

Finally, [5] proposes a reassignment technique that arranges the input data into a lower dimensionality space, which reflects the main association relationships between terms and documents. The *Singular Value Decomposition* (SVD) technique is used and then the Greedy-NN *TSP* algorithm is applied. However SVD transformation is not very scalable, and the block partitioning approach proposed for scalability reduces the effectiveness of the method.

Chapter 3

The Partial Order Index (POI)

3.1 Introduction

In this Chapter an indexing scheme is described, called `POI` [42], which is based on partial orders and focus on performance aspects such as the storage space and the time needed to create/retrieve a version, as they described in Chapter 1. This Section provides an overview of `POI`, while the subsequent Sections describe it in detail.

`POI` aims at exploiting the fact that it is expected to have several versions (not necessarily consecutive) whose contents overlap, as well as the fact that RDF graphs have not a unique serialization, as it happens with text. The latter allows the exploration of directions that have not been elaborated by the classical versioning systems for texts (e.g. [39, 3]), in order to reduce the storage space and to provide efficient version insertion and retrieval.

Specifically, `POI` approach views an RDF KB as a *set* of triples. It is important to note that the structure (and occupied space) of `POI` is independent of the version history. Knowledge of version history can be exploited just for speeding up some operations (specifically the insertion of versions that are defined by combining existing versions). It follows that the benefits from adopting `POI` are not limited to versioning. It could also be exploited for building repositories appropriate for collaborative applications, e.g. for applications that require keeping personal and shared spaces (KBs) as it is the case of modern e-learning applications (e.g. see trialogical e-learning [40]). In such cases, a set of KBs are needed to be stored that are not historically connected and are expected to

overlap.

In comparison to the change-based approaches (proposed in the context of the SW [46] or not [9, 11]) we could say that `POI` stores explicitly only the versions with the minimal (with respect to set containment) contents. All the rest versions are stored in a positively incremental way, specifically positive deltas are organizing as a partially ordered set (that is history-independent) aiming at minimizing the total storage space. It follows that `POI` occupies less space than the change-based approach in cases where there are several versions (or KBs in general) not necessarily consecutive (they could be even in parallel evolution tracks) whose contents are related by set inclusion ($\subseteq$).

Regarding version retrieval time, the cost of retrieving the contents of a version in the change-based approach is analogous to the distance from the closest stored snapshot (either first or last version, according to the delta policy adopted [11]). Having a `POI` the cost is independent of any kind of history, but it depends on the contents of the particular version, specifically on the depth of the corresponding node in the `POI` graph.

In general, `POI` is an advantageous approach for archiving set-based data, especially good for inclusion-related and "oscillating"[1] data.

3.2 Framework and Notations

First of all, some notations are introduced as they are described in [42].

Def. 1 If $\mathcal{T}$ is the set of all possible RDF triples (or quadruples if we consider graph spaces [17]), then :

- a *knowledge base* (for short KB) is a (finite) subset S of $\mathcal{T}$,
- a *named knowledge base* (for short NKB) is a pair (S, i) where i is an identifier and S is a KB,
- a *multi knowledge base* (for short MKB) is a set of NKBs each having a distinct identifier, and
- a *versioned knowledge base* (for short VKB) is a MKB plus an acyclic *subsequent* relation over the identifiers of the NKBs that participate to MKB.

[1] Suppose, for instance, a movie database describing movies, theaters that showtimes. Unlike movie descriptions, which rarely change, theaters and showtimes change daily.

	Service	Pre-condition	Post-Condition
1	$insert(S,i)$	$i \notin id(V)$	$V' = V \cup \{(S,i)\}$
2	$merge(i,j,h,\odot)$	$\{i,j\} \subseteq id(V)$, $h \notin id(V)$	$V' = V \cup (D(i) \odot D(j), h)$, $next'(i) = next(i) \cup \{h\}$, $next'(j) = next(j) \cup \{h\}$

Table 3.1: Version creation services

Let Id be the set of all possible identifiers (e.g. the set of natural numbers). If $v = (S,i)$ is a NKB (i.e. $S \subseteq \mathcal{T}$, $i \in Id$) then we will say that i is the identifier of v, and S is the content of v (we shall write $id(v) = i$ and $D(i) = S$ respectively).

Let $V = \{v_1, \ldots, v_k\}$ be a MKB. We shall use $id(V)$ to denote the identifiers of v_i, i.e.

$$id(V) = \{ \ id(v) \mid v \in V\}$$

and $D(V)$ to denote their KBs, i.e.

$$D(V) = \{ \ D(i) \mid i \in id(V)\}$$

We shall use T_V denote the set of all distinct triples of V, i.e.

$$T_V = \cup_{i \in id(V)} D(i)$$

So $D(V)$ is a family of subsets of T_V.

A *subsequent* (version) relation over a MKB V is any function of the form $next : id(V) \rightarrow \mathcal{P}(id(V))$, where $\mathcal{P}(\cdot)$ denotes powerset. For instance, suppose that $next(i) = \{j,k\}$. In this case we will say that i is a direct previous version of j and k, and that j and k are direct next versions of i. If $next(i) = \emptyset$, then we will call version i *leaf* version of V. We call a version id i the *root* version of V, if there does not exist any version id j, such that $i \in next(j)$. A pair $(V, next)$ is a VKB if the graph $(id(V), next)$ is acyclic.

Table 3.1 presents some version management services and their semantics in the form of conditions that should hold before their call and after their run. In particular, *insert* adds a new version with id i and version content S. Additionally, *merge* differs from *insert* service in that the content of the new version (with identifier h) is the result of the application of a set operator, denoted by $\odot$ (i.e. $\cup, \cap, \setminus$), on the contents of two (or more) existing versions (having identifiers i and j). In this way we could model operators like those proposed in [22].

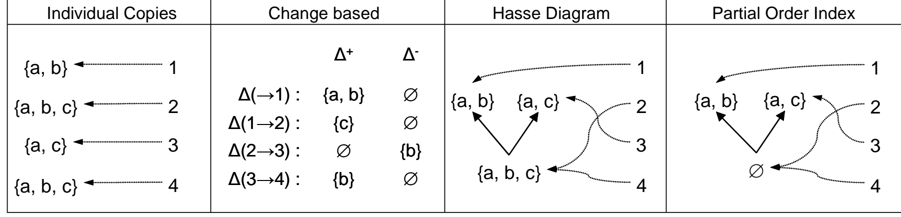


Figure 3.1: Storage Example when the partial order index is used (right) or not (left)

3.3 The Partial Order Index (`POI`) in Detail

In this Section `POI` is described through a small intuitive example. Consider a versioned knowledge base $(V, next)$. Specifically consider a V comprising four versions with: $D(1) = \{a, b\}, D(2) = \{a, b, c\}, D(3) = \{a, c\}$ and $D(4) = \{a, b, c\}$, where a, b, c denote triples. This means that $V = \{(\{a, b\}, 1), (\{a, b, c\}, 2),$
$(\{a, c\}, 3), (\{a, b, c\}, 4)\}$. Below methods for storing V are presented. To aid understanding, Figure 3.1 sketches the storage policies that we will investigate in the sequel. One trivial approach, which is adopted by current SW versioning tools [43, 31, 25], is to store each individual NKB independently and entirely. Another approach [46, 9, 11] is to store the initial NKB and the deltas of every other version with respect to its previous version. Since versions usually contain overlapping triples, we propose the use of an index in order to reduce the number of triple copies. For instance, in the example of Figure 3.1 four copies of triple a are stored in the trivial case (see the left part of Figure 3.1). However, with the use of the `POI`, only two a copies are stored (see the right part of Figure 3.1). To describe this index, firstly some preliminary background material and definitions are introduced.

Given two subsets S, S' of $\mathcal{T}$, we shall say that S is narrower than S', denoted by $S \leq S'$, if $S \supseteq S'$. So, $\emptyset$ is the top element of $\leq$, and the infinite set $\mathcal{T}$ is the bottom element. Clearly, $(\mathcal{P}(\mathcal{T}), \leq)$ is a partially ordered set (poset).

We can define the *partial order of a versioned KB* by restricting $\leq$, on the elements of $D(V)$. For brevity, we shall use the symbol $\sqsubseteq$ to denote $\leq_{|D(V)}$.

In our running example, we have $D(V) = \{\{a, b\}, \{a, b, c\}, \{a, c\}, \{a, b, c\}\}$ and the third diagram of Figure 3.1 shows the Hasse diagram of the partially ordered set $(D(V), \sqsubseteq$). This diagram actually illustrates the structure of the so-called *storage graph* that is introduced below.

Notation	Definition	Equiv. Notation
R	a binary relation over the set of nodes N	$\rightarrow$
R^t	the transitive closure of the relation R	$\rightarrow^t$
R^r	the reflexive and transitive reduction of the relation R	$\rightarrow^r$
$Up(n)$	$= \{n' \mid (n,n') \in R\} = \{n' \mid n \rightarrow n'\}$	
$Down(n)$	$= \{n' \mid (n',n) \in R\} = \{n' \mid n' \rightarrow n\}$	
$Up^t(n)$	$= \{n' \mid (n,n') \in R^t\} = \{n' \mid n \rightarrow^t n'\}$	
$Down^t(n)$	$= \{n' \mid (n',n) \in R^t\} = \{n' \mid n' \rightarrow^t n\}$	
$content(n)$	$= \cup\{\ stored(n') \mid n' \in Up^t(n)\}$	

Table 3.2: Notations for Storage Graphs

A *storage graph* Υ is any pair $\langle \Gamma, stored \rangle$ where $\Gamma = (N, R)$ is a directed acyclic graph and *stored* is a function from the set of nodes N to $\mathcal{P}(\mathcal{T})$. If $(a, b) \in R$, i.e. it is an edge, we will also write $a \rightarrow b$.

For a node $n \in N$, $stored(n)$ is actually a set of triples, so it is the storage space associated with node n. Table 3.2 shows all notations (relating to storage graphs) that will be used.

For each node n of a storage graph we can define its *content*, denoted by $content(n)$, by exploiting the structure of the graph and the function *stored*. Specifically we define:

$$content(n) = \cup\{\ stored(n') \mid n \rightarrow^t n'\} \tag{3.1}$$

so it is the union of the sets of triples that are stored in all nodes from n to the top elements of Γ. We should stress that $content(n)$ is not stored, instead it is computed whenever it is necessary.

The *Partial-Order Index*, for short `POI`, is a storage graph whose structure is that of $(D(V), \sqsubseteq^r)$. Note that $\sqsubseteq^r$ denotes the reflexive and transitive reduction of $\sqsubseteq$. Consider a NKB $(D(i), i)$. Each version id i is associated with a node n_i whose storage space is defined as:

$$\begin{aligned} stored(n_i) &= D(i) \setminus \{\ D(j) \mid D(j) \sqsubseteq D(i)\ \} \\ &= D(i) \setminus \{\ stored(n_j) \mid n_i \rightarrow^t n_j\ \} \\ &= D(i) \setminus \{\ stored(n_j) \mid n_j \in Up^t(n_i)\ \} \end{aligned}$$

It follows easily that if $n_i \rightarrow^t n_j$ then it holds $stored(n_i) \cap stored(n_j) = \emptyset$. The fourth diagram of Figure 3.1 illustrates the storage graph of this policy for our running example. For each node n_i the elements of $stored(n_i)$ are shown at the internal part of that node.

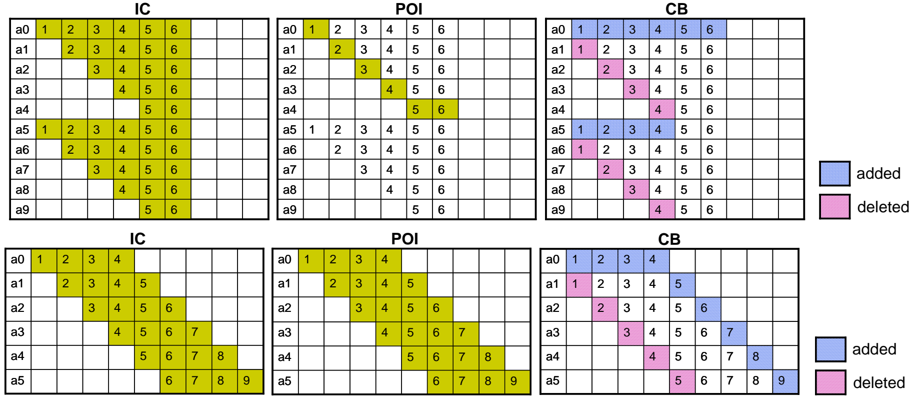


Figure 3.2: Best (upper) and worst (bottom) case for `POI`

Although we have 4 versions, Γ contains only 3 nodes. As an example, $D(4) = \{a, b\} \cup \{a, c\} = \{a, b, c\}$.

3.4 Analyzing Storage Space Requirements

In this Section, `IC` , `CB` and `POI` approaches are compared with respect to *storage space*. To show the best and worst cases for `POI` we consider two examples. In the first example, we consider 10 knowledge bases (KBs) a_i, where $i \in [0, 9]$ each being a set of triples where each triple is assigned a unique identifier. Specifically assume that the contents of the KBs are:

$a_0 = a_5 = \{1, 2, 3, 4, 5, 6\}$,

$a_1 = a_6 = \{2, 3, 4, 5, 6\}$,

$a_2 = a_7 = \{3, 4, 5, 6\}$,

$a_3 = a_8 = \{4, 5, 6\}$, and

$a_4 = a_9 = \{5, 6\}$,

where 1...6 are triple identifiers. We assume that they form a single evolution track, i.e. a_{i+1} is the next version of a_i for all $i \in [0, 9]$. The upper part of Figure 3.2 illustrates what is stored in `IC`, `POI` and `CB` approach. The stored triples are drawn in a colored background. Especially for `CB` we draw with different background the triples that should be added with those that should be deleted (both are stored).

In the second example, we consider 6 knowledge bases (KBs) a_i, where $i \in [0, 5]$, again forming a single evolution track. Specifically assume that the contents of the KBs are:

$a_0 = \{1, 2, 3, 4\}$,

$a_1 = \{2, 3, 4, 5\}$,

$a_2 = \{3, 4, 5, 6\}$,

$a_3 = \{4, 5, 6, 7\}$,

$a_4 = \{5, 6, 7, 8\}$, and

$a_5 = \{6, 7, 8, 9\}$,

where 1...9 are triple identifiers. The bottom part of Figure 3.2 illustrates what is stored in `IC`, `POI` and `CB` approach.

Let us now describe formally the best and worst case of each policy. If Z denotes a policy, we shall use $space_t(Z)$ to denote the number of triples that are stored according to policy Z. It is not hard to see that

$$|T_V| \leq space_t(\mathtt{POI}) \;\leq\; space_t(\mathtt{IC}) = \sum_{i \in id(V)} |D(i)|$$

$$|T_V| \leq space_t(\mathtt{CB}) \;\leq\; 2 \sum_{i \in id(V)} |D(i)|$$

Regarding the first formula, strict inequality holds, i.e. $space_t(\mathtt{POI}) < space_t(\mathtt{IC})$ if there is a version whose content is a subset of the content of another version. The worst case for `POI` is when all nodes of the storage graph are leaves (except for the root). This is the case drawn in the bottom part of Figure 3.2. That case leads to space requirements equal to those of `IC`. On the other hand, the best case for `POI`, is when the content of every version is a subset of the content of every version with greater (or equal) content cardinality. In that case every triple is stored only once in the storage graph. This is the case drawn in the upper part of Figure 3.2. Regarding `CB`, the worst case is to have a track where the same set of triples is once stored and once deleted in an alternative fashion. In that case, `CB` stores $2 \times space_t(\mathtt{IC})$ triples. The best case for `CB` is when the contents of the KBs form a chain with respect to $\subseteq$ (also holding in the best case for `POI`) and they are consecutive in the version history (which is not required for the best case of `POI`). In that case `CB` stores every triple only once and thus coincides with the best case of `POI`.

Regarding graph size, for `IC` and `CB` no index structure has to be kept as the entire of every version is stored explicitly. Concerning `POI`, the number of nodes of the storage graph is $|D(V)|$. Notice that $|D(V)| \leq |id(V)|$, i.e. less than or equal to the number of

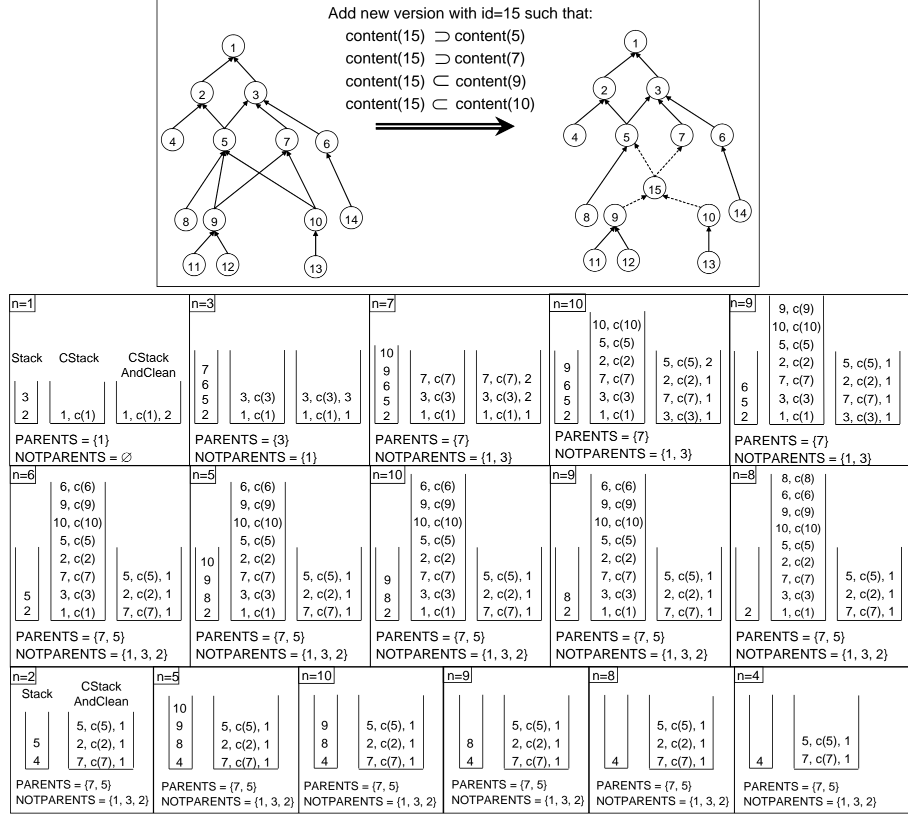


Figure 3.3: Example of adding a new version using Algorithm *Insert POIds*

versions[2]. The number of edges coincides with the size of the relation $\sqsubseteq^r$. This relation can have at most $\frac{N^2}{4}$ relationships. This value is obtained when Γ is a bipartite graph, whose $\frac{N}{2}$ nodes are connected with all other $\frac{N}{2}$ nodes. More on the overall comparison of these policies are described below.

3.5 Version Insertion Algorithms

The insertion algorithms exploit the structure and the semantics of the storage graph. Intuitively, we have to check whether the new version is subset or superset of one of the existing versions. To this end, we start from the root(s) of the storage graph and we descend. To be more specific, let $\mathcal{B}$ be a family of subsets of $\mathcal{T}$, let $<$ be the cover relation over these and let $<^r$ be its transitive reduction. Suppose that we want to insert a new subset A (i.e. $A \notin \mathcal{B}$) and update accordingly the relation $<^r$. We define:

$$
\begin{aligned}
Parents(A) &= \min_{<}\{\ B \mid A < B\} = \{\ B \mid A <^r B\} \\
Children(A) &= \max_{<}\{\ B \mid B < A\} = \{\ B \mid B <^r A\}
\end{aligned}
$$

[2] In the example of Figure 3.1, $|D(V)| = 3$ while $|id(V)| = 4$

3.5.1 *POI-Plain* (`Insert POI`$_p$) Insertion Algorithm

Algorithm *InsertInPoset*

Input: A: a set of triples.

Output: updated cover relation of the poset so that to contain a node corresponding to A (if there is no such node already).

1. /* FIND PARENTS */
2. Stack = new STACK();
3. PARENTS = new Set();
4. NOTPARENTS = new Set();
5. **while** not(isEmpty(Stack))
6.   **do** n = pop(Stack);
7.   **if** $n \in$ NOTPARENTS
8.   **then** push(Stack, $\{x \in Down(n) \mid x \notin PARENTS)\}$);
9.   **else** **if** n.contents $= A$ /* a node with contents A already exists */
10.   **then** break;
11.   **else** **if** n.contents $\subset A$ /* all upper nodes of n are certainly not parents */
12.   **then** PARENTS = (PARENTS $\cup\{n\}) \setminus Up^t(n)$;
13.   NOTPARENTS = NOTPARENTS $\cup Up^t(n)$;
14.   push(Stack, $Down(n)$);
15. /* FIND CHILDREN */
16. Stack = new STACK(PARENTS); /* a new stack with initial contents the set PARENTS */
17. CHILDREN = new Set();
18. **while** not(isEmpty(Stack))
19.   **do** n = pop(Stack);
20.   **if** (n.contents $\supset A$)
21.   **then** CHILDREN = CHILDREN $\cup\{n\}$;
22.   **else** push(Stack, $Down(n)$);
23. /* CONNECT A */

24. nA = new node(A);
25. **for** each $c \in CHILDREN$
26. **do** Add($c \rightarrow nA$); /* i.e. Add($c <^r A$) */
27. **for** each $p \in PARENTS$
28. **do** Add($nA \rightarrow p$); /* i.e. Add($A <^r p$) */
29. /* ELIMINATE REDUNDANCIES */
30. **for** each $c \in CHILDREN$
31. **for** each $p \in PARENTS$
32. **do if** $c \rightarrow p$
33. **then** Delete($c \rightarrow p$); /* i.e. if $c <^r p$ then Delete($c <^r p$) */

To update $<^r$ we have to add the relationships $A <^r p$ for every $p \in Parents(A)$, and $c <^r A$ for every $c \in Children(A)$. In addition we have to eliminate redundant relationships (that may exist between $Parent(A)$ and $Children(A)$, specifically we have to eliminate all $c <^r p$ relationships where $c \in Children(A)$ and $p \in Parents(A)$). Let now see how we could find the children and the parents of a set A and update appropriately the relation $<^r$. Returning to the problem at hand, this scenario corresponds to the case where each node n of a storage graph had explicitly stored *contents*(n). Algorithm *InsertInPoset* sketches the crux of the algorithm in pseudocode. It is based on a Stack and two sets called PARENTS and NOTPARENTS. The root of the storage is denoted by $root(\Gamma)$ and every storage graph has a single root corresponding to a dummy version with an empty content. Note that the relation $\rightarrow$ of the storage graph corresponds to the relation $<^r$, i.e. $a \rightarrow b \Rightarrow content(b) \subset content(a)$. An indicative example of version addition is illustrated in Figure 3.3. The stack contents are shown in every step.

To implement version insertion we could use Alg. *InsertInPoset*. The only difference is that in a storage graph *n.contents* (where n is a node) is not explicitly stored (instead only *n.stored* is stored). One naive approach would be to compute *n.contents* by taking the union of the stored triples of its (direct and indirect) broader nodes, i.e. to use the formula (3.1). Each such computation would require $\mathcal{O}(d(n))$ set union operations where $d(n)$ is the depth of the node n multiplied to the average number of parents. If the storage graph is a tree, then all set union operations would actually be concatenations (and thus faster). In case of DAG, we have to perform set union operations only for nodes that have

more than one father. We will hereafter call this algorithm ***POI-Plain*** insertion (for short `Insert` $\mathtt{POI_p}$) algorithm.

3.5.2 *POI-DoubleStack (Insert POI_{ds})* Insertion Algorithm

To reduce the number of set union operations that are issued by the `Insert` $\mathtt{POI_p}$ algorithm for computing *content*(n) for a node n, here we present a more time efficient algorithm which employs a second stack (actually it is a cache) for keeping stored (and thus reusing) results of operations that have already been computed. The second stack, called $CStack$ (where $'C'$ comes from contents), stores elements comprising of two components: a version id and its content (i.e. a set of triples). The extension of Alg. *InsertInPoset* with the second stack is Algorithm *Insert-*POI_{dsc} .

Algorithm *TSContent*

Input: n: a node id.

Output: the set of triples comprising the content of node n (the stack CStack is updated, if necessary).

1. **if** e = **lookup**(CStack, n)
2. **then return** e.content; /* returns the contents of element e */
3. **else**
4. Res = stored(n);
5. **for** each $n' \in Up(n)$
6. **do** Res = Res $\cup$ **TSContent**(n'); /* this is a concatenation if $|Up(n) = 1|$*/
7. **push**(CStack, (n, Res)); /* pushes a pair (key, content) to the stack */
8. **return** Res;

To compute the contents of a node n it uses the function **TSContent** (Algorithm *TSContent*) which accesses the second stack. If the storage graph were a tree then we would be sure that all broader nodes of a node are in the stack (in both $Stack$ and $CStack$ stacks). However the storage graph is a DAG in the general case, so this is not always true. That's why TSContent uses a lookup and if the sought element is not in $CStack$ it creates and stores it to $CStack$. We will hereafter call this algorithm *Insert POI-DoubleStack*

(for short `Insert POI`$_{ds}$) insertion algorithm.

3.5.3 *POI-DoubleStackAndClean (`Insert POI`$_{dsc}$)* Insertion Algorithm

To reduce the total space needed by *CStack*, Alg. *Insert-POI$_{dsc}$* actually uses a different implementation of Alg. *TSContent* called **TSContentAndClean** (Alg. *TSContentAndClean*) that frees the contents of those versions that are not needed any more. Specifically, an element of *CStack* should be removed if one of the following two conditions holds:
(a) its content is not a subset of A, so the traversal of the storage graph will not continue to its descendants and therefore its contents are not needed any more,
(b) it is not a (definite) parent of the node to be inserted and all its children are already in *CStack*. Specifically, if all its children are already in *CStack* the version content is not needed because it is a subset of the content of every child of it.

To this end we extend the structure of each *CStack* element with a third component, denoted by Y, which is actually a variable initialized to the number of children (of the corresponding node) that is decreased by one whenever lookup finds and fetches that element. When it reaches 0, the element (if not a parent) should be removed because that means that the contents of all its children are already stored in *CStack*. We will hereafter call this algorithm *Insert POI-DoubleStack(Clean)* (for short Insert `Insert POI`$_{dsc}$).

Algorithm ***Insert-***POI_{dsc}

Input: A: a set of triples.

Output: updated storage graph and CStack.

1. /* FIND PARENTS */
2. Stack = new STACK();
3. PARENTS = new Set();
4. NOTPARENTS = new Set();
5. push (Stack, {root(Γ)});
6. **while** not(isEmpty(Stack))
7.   **do** n = pop(Stack);
8.   **if** n ∈ NOTPARENTS
9.   **then** push(Stack, $\{x \in Down(n) \mid x \notin PARENTS\}$);

10. **else**
11. **if** n.contents = A /* a node with contents A already exists */
12. **then** break;
13. **else**
14. **if** **TSContentAndClean**(n, PARENTS, n) $\subset A$
15. **then** PARENTS = (PARENTS $\cup\{n\}) \setminus Up^t(n)$;
16. NOTPARENTS = NOTPARENTS $\cup Up^t(n)$;
17. **delElems**(CStack, $Up^t(n)$);
18. push(Stack, $Down(n)$);
19. **else** **delElem**(CStack, n);
20. /* FIND CHILDREN */
21. Stack = new STACK(PARENTS);
22. CHILDREN = new Set();
23. **while** not(isEmpty(Stack))
24. **do** n = pop(Stack);
25. **if** **TSContentAndClean**(n, PARENTS, n) $\supset A$
26. **then** CHILDREN = CHILDREN $\cup\{n\}$;
27. **else** push(Stack, $Down(n)$);
28. /* CONNECT A */
29. nA = new node(A);
30. nA.stored = A $\setminus \cup$ {**TSContentAndClean**(n') $\mid n' \in$ Parents};
31. **for** $c \in CHILDREN$
32. **do** Add($c \rightarrow nA$); /* i.e. Add($c <^r A$) */
33. **for** $p \in PARENTS$
34. **do** Add($nA \rightarrow p$); /* i.e. Add($A <^r p$) */
35. /* ELIMINATE REDUNDANCIES */
36. ... as in Alg. *InsertInPoset*
37. /* UPDATE THE STORED CONTENTS OF THE CHILDREN NODES */
38. **for** $c \in CHILDREN$
39. **do** c.stored = c.stored - A;

Algorithm *TSContentAndClean*

Input: n: a node id

1. $PARENTS$: the ids of n's parents
2. $initialN$: a node id (in the root call of the routine n = initialN).

Output: the sets of triples comprising the content of node n. The stack is updated (addition, deletion) if necessary.

3. **if** e = **lookup**(CStack, n)
4. **then** Res = getElemContent(CStack, n);
5. **if** (e.Y == 1) & (n $\notin$ PARENTS)
6. **then** **delElem**(CStack, n);
7. **else** **editElem**(CStack, (n, Res, Y), (n, Res, Y-1)); /* decreases the 3rd component of the stack element */
8. **else**
9. Res = stored(n);
10. **for** each $n' \in Up(n)$
11. **do** Res = Res ∪ **TSContentAndClean**(n', PARENTS, initialN); /* concatenation */
12. **push**(CStack, (n, Res, $|Down(n)|$)); /* pushes a triple (key, content, number of children) to the stack */
13. **return** Res;

Figure 3.3 shows these stacks in our running example. Specifically, the left stack is the *Stack*, the center stack is the *CStack* as employed by *TSContent* algorithm, while the right one is the *CStack* as employed by *TSContentAndClean* algorithm. After the PARENTS have been computed, *CSTack* remains the same and therefore we show only *Stack*. Notice how shorter *CStack* is according to *TSContentAndClean*.

3.5.4 History-based Version Insertion Speedup

We have already described a general algorithm for inserting versions and recall that the storage space requirements of `POI` are independent of the evolution history. However the knowledge of the evolution history can speed up version insertion, especially in the case we insert versions that are defined through set operations over existed versions. Below we discuss some cases.

- Creation by Union

  For instance, consider the case we want to insert a new version v_3 such that $D(v_3) = D(v_1) \cup D(v_2)$, where v_1, v_2 are existing versions. In that case, the search for parents begins with the `POI` nodes n_1, n_2 that correspond to v_1 and v_2 respectively, instead of the root, because $D(v_1) \subset D(v_3)$ and $D(v_2) \subset D(v_3)$. Nodes n_1 and n_2 will be the parents of the `POI` node that corresponds to v_3, unless there exists a descendant n_4 (corresponding to an existing version v_4) of n_1 or n_2, such that $D(v_4) \subset D(v_3)$.

- Creation by Intersection

  Additionally, the case where $D(v_3) = D(v_1) \cap D(v_2)$ is in a sense dual to the previous one, as the children of n_3 will be n_1, n_2 or some nodes above them.

3.6 Cross Version Operations

In general, cross-version operations are expensive in both `IC` and `CB` approach. The availability of a `POI` allows performing them more efficiently.

3.6.1 Containment Checking

For instance, containment checking can clearly benefit from a `POI`. To decide whether $D(i) \subseteq D(j)$ one could pose a reachability query on the storage graph (no need to access the contents of the versions). Moreover, by adopting a labeling scheme [1] for the storage graph Γ we could decide inclusion in $\mathcal{O}(1)$.

3.6.2 Containment Queries

Additionally, let S be a set of triples. Suppose we want to find all versions i such that $S \subseteq D(i)$ (or $D(i) \subseteq S$). Specifically consider the following kind of queries:

- subset queries, i.e. $\{\ i \mid D(i) \subset S\}$

- equality queries, i.e. $\{\ i \mid D(i) = S\}$

- superset queries, i.e. $\{\ i \mid D(i) \supset S\}$

Such queries would be very expensive in the `IC` or in the `CB` approach. By employing a `POI` we can use the insertion algorithm to insert S to the storage graph. Let n be the inserted node. The sought versions are those that point to nodes of $Nr^t(n)$ (resp. $Br^t(n)$). Specifically, if A denotes the nodes of the storage graph what we should find in order to compute the answer, then:

- subset queries: $A = Nr^t(n)$
- equality queries: $A = \{n\}$
- superset queries: $A = Br^t(n)$

The answer of such queries, i.e. the sought versions identifiers, are those identifiers that point to the nodes in the set A.

Notice that for subset/supset queries the answer can be computed (and returned) gradually and in an order that reflects the distance (in the Hasse Diagram) from the node that corresponds to the set S. So in a superset query the versions whose contents are closer to S are returned first. It follows that this technique is appropriate for big datasets where gradual query evaluation and results ranking are desirable properties.

Containment queries can be straightforwardly extended with negation.

- negative subset queries, i.e. $\{\ i \mid D(i) \not\subset S\}$

  Here $A = N \setminus Nr^t(n)$

- inequality queries, i.e. $\{\ i \mid D(i) \neq S\}$

  Here $A = N \setminus \{n\}$

- negative superset queries, i.e. $\{\ i \mid D(i) \not\supset S\}$

  Here $A = N \setminus Br^t(n)$

3.7 Extensions

3.7.1 Semi-lattice approach

Let X be the minimum in size family of subsets of $\mathcal{T}$ which includes the family of sets $D(V)$ and is closed with respect to intersection. Obviously, $(X, \sqsubseteq)$ is a meet-semilattice.

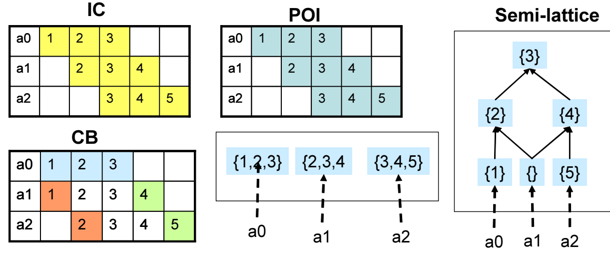


Figure 3.4: Example of the semi-lattice approach

We can define a storage graph whose structure is that of $(X, \sqsubseteq^r)$. Again, each version id i is associated with a node n_i whose storage space is defined exactly as in POI.

It follows easily that the storage graph of this policy satisfies the following properties:

- $i \neq j \Rightarrow stored(n_i) \cap stored(n_j) = \emptyset$
- for each $i, j \in id(V)$, the storage graph contains a node x such that $content(x) = S_i \cap S_j$.

One can easily see that this index guarantees that each $t \in T_V$ is stored exactly once (something that is not true for POI).

Figure 3.4 shows an example of a worst case for POI and how the semi-lattice approach would be in that case. Notice that each triple is stored only once.

The storage gains obtained are compensated by the space required to keep the excessive number of nodes and edges of the storage graph.

3.8 Comparison with Change-Based Approach

As far as the storage space requirements, POI stores explicitly only the versions with the minimal (with respect to set containment) contents. All the rest versions are stored in a positively incremental way. Specifically positive deltas are organized as a partially ordered set (that is history-independent) aiming at minimizing the total storage space. It follows that POI occupies less space than the change-based approach in cases where there are several versions (or KBs in general) not necessarily consecutive (they could be even in parallel evolution tracks) whose contents are related by set inclusion ($\subseteq$).

In respect of the cost to retrieve the contents of a version, in the CB approach is analogous to the distance from the closest stored snapshot (either first or last version,

according to the delta policy adopted [11]). On the contrary, in POI it is independent of any kind of history, but it depends on the contents of the particular version, specifically on the depth of the corresponding node in the POI graph.

Chapter 4

The Compact POI (CPOI)

4.1 Introduction

In the previous Chapters we discussed the most popular approaches that are followed for the storage of several versions. As the necessity of keeping many versions of a KB (or more) in many applications brings on the need of storing a vast amount of data, the exigency of an efficient storage approach with low storage space requirements becomes essential. In the previous Chapter, `POI` was described and discussed. It was explained that `POI` structure offers efficient performance in aspects like the storage space and the time needed for handling versions of RDF/S data and information. However, there are cases that a `POI` cannot guarantee space savings (i.e. versions having none subset or superset relations). We need an approach that not only offers significant space savings over a large amount of data, but also guarantees space savings in all cases.

This Chapter elaborates on the problem of further reducing the storage space of `POI`. In particular, it investigates whether techniques which have been used with success in the area of IR (Information Retrieval) systems and WSE (Web Search Engines) can be exploited for RDF data. In IR, the adoption of encoding schemes in an inverted index saves space because there are words that occur in many documents. Specifically, if the documents that share a lot of common words get close identifiers, and a *gapped representation* for their identifiers is adopted using an encoding scheme that represents small integers with a small number of bits, then the postings lists of these common words will have small bit representation. To ensure that documents with a lot of common words get

close identifiers, the documents of the corpus are clustered, so that documents that share a lot of common words are placed in the same cluster. Subsequently documents are given identifiers on a cluster basis. Experiments in the IR domain (specifically in the TREC web data [33]) have shown that the *compression ratio*, defined as the size of the compressed file as a percentage of the uncompressed file (i.e. $Compression\ ratio = \frac{Compressed\ size}{Uncompressed\ size} 100\%$), is around 30%-45%. In our case, since each triple has a unique identifier, and hence each node of POI stores a set of identifiers, we investigate whether special identifier assignments and integer encodings, such as Elias-γ [12], can reduce (and under what conditions) the occupied space.[1] Although POI by construction offers significant storage gains, the motivation for investigating this approach is to tackle the cases where POI is slightly beneficial; cases where there are several overlapping versions which are not related by inclusion. In such cases, POI behaves like the IC approach. An appropriate identifier encoding promises space gains in such cases.

Hereafter the term CPOI (Compact POI) will be used to refer to a compact version of POI, that relies on gapped numeric identifiers and special encodings. To grasp the idea Figure 4.1 illustrates an example of using the gapped representation of the nodes's contents and their encoding using Elias-γ. We consider 6 knowledge bases (KBs), whose contents are:
a0 = {1, 2, 3, 4, 5},
a1 = {1, 2, 3, 4, 5, 6},
a2 = {1, 2, 3, 4, 7, 8},
b0 = {1, 2, 3},
b1 = {1, 2, 3, 4, 5, 6}, and
b2 = {1, 2, 7},
where 1...8 are triple identifiers.

The left part of Figure 4.1 shows the structure of the original storage graph (plain POI). The middle part of the same Figure exhibits the gapped representation with integers. In this representation, as we will explain in detail later, we consider the elements of a node as a list of ids in ascending order and we keep the first id unchanged while each one of the rest ids is replaced by its difference from the previous id. The right part of Figure 4.1

[1] In general schemes that encode small values with shorter codes are ideal for applications where the probability of the integer n follows a power law [32].

shows the final form of a CPOI. Specifically, the first value of every node is represented as a normal integer, while the rest values are represented using a special encoding for integers (Elias-γ here).

Regarding evaluation we consider that an identifier assignment/encoding scheme is successful if the size of CPOI is: less than the size of the original (non-encoded) POI, and less than the size obtained using a random assignment of identifiers. In addition, the cost for the reassignment should be fast enough (certainly polynomial), and it should be efficient to enrich the index with new versions.

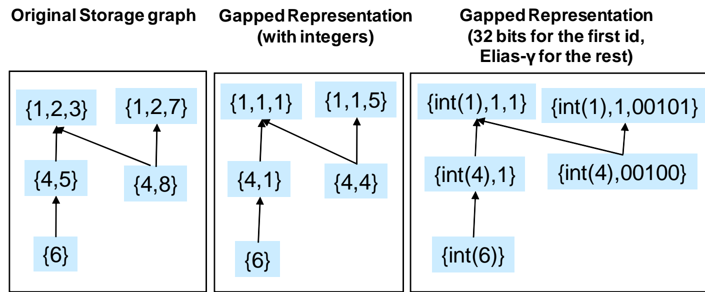


Figure 4.1: A compact representation of POI

Comparing CPOI with IC and CB approach

In Chapter 3 the structure of POI was described. Apart from the description of the way that this structure stores versions, it was shown that it uses triples' identifiers instead of triples themselves, keeping all triples' strings in a table, in contrast to IC and CB that store triples' strings.

The main differentiation of a CPOI is that it uses a gapped representation and special encodings to represent triples' identifiers in the storage graph.

4.2 Analytical Comparison of CPOI with plain POI

We shall use T to denote the set of all distinct triples and $|T|$ the cardinality of this set. The *storage graph* Υ of a POI is a pair $\langle \Gamma, stored \rangle$ where $\Gamma = (N, R)$ is a directed acyclic graph and *stored* is a function from the set of nodes N to the powerset of T.

Each node n of the storage graph of POI holds a set of numeric triple identifiers. Let use the same symbol n to denote this set (i.e. $n.stored$) and $|n|$ to refer to its cardinality. We can keep this set as a list, and suppose this list is sorted in ascending order. This list

can be considered as a sequence of gaps between triples identifiers, e.g. for the sequence [32011, 32013, 32014, 32017], the sequence of gaps would be [32011, 2, 1, 3] (of course the original identifiers can be recomputed through sums over the gaps). This *d-gapped* representation as it is commonly known, is very popular in the area of IR [45]. The list of gapped identifiers can then be compressed using a suitable compression scheme. Compression is obtained by encoding small values with shorter codes. Special encodings, such as Elias-γ, δ [12], Golomb-Rice [15], Binary Interpolative Coding [28] and Variable-byte encoding, are efficient for small integers, as the smaller the integer is the less space is needed. However, when integers become large, the storage space also becomes large. For example, and using Elias-γ, the list [32011, 2, 1, 3] can be represented as [32011, 010, 1, 011]). The second, third and forth element of that list are bit sequences. However notice that the first element of the list is an integer in decimal which could be digitally encoded with a commonly used 4 byte representation, because we expect to have small numbers only in gaps, so the first number of the list could be any number. This means that it would not be beneficial to adopt a special encoding for the first number. So [32011, 32013, 32014, 32017] is actually represented by the following list of bit sequences $[00000000000000000111110100001011, 010, 1, 011]$, and this is what we call `CPOI`. It follows that the more id numbers are close to each other in every node, the more space saving can be achieved.

Note that in our approach each id represents an entire triple, in contrast to the related work of compact RDF triple representations (e.g. x-RDF-3X, HTD representation) where an id represents a triple's element (subject, predicate or object).

The Gaps of a Node

For each node n the space required for its representation depends on both $|n|$ and the way we represent the ids of the triples in n. Regarding the latter let us define the sum of gaps between consecutive ids, as this determines (approximates) the total size of ids (and it is independent of any particular encoding scheme). Indeed, such sum of gaps coincides with the bits required if the *unary* representation[2] is used. Specifically, if $n =$

[2]Each positive integer n is represented by n bits, specifically by n - 1 ones followed by one zero (or n - 1 zeros followed by an one).

$\{tr_1, ..., tr_{|n|}\}$, we define:

$$gaps(n) = \sum_{i=1}^{|n|-1} (tr_{i+1} - tr_i) \tag{4.1}$$

For example, if $n = \{1, 4, 8\}$, then $gaps(n) = 3{+}4 = 7$. The smaller the value of $gaps(n)$ is, the better representation (for n) can be achieved. It is not hard to see that: $|n| - 1 \leq gaps(n) \leq |T| - 1$. The minimum value and therefore the best case for $gaps(n)$ is achieved when the ids are consecutive numbers (e.g. $n = \{8, 9, 10\}$). On the other hand, the worst case for $gaps(n)$ is when n contains ids that cover the entire range of values (hence from 1 to $|T|$). In that case, $gaps(n)$ equals to $|T| - 1$, as it actually expresses the transition from 1 to $|T|$ and each id covers one step of that transition. For example, if $T = \{1, ..., 100\}$, the worst representation for a node n such that $|n|$=3, leads to $gaps(n) = 99$ (e.g. both $n = \{1, 5, 100\}$ and $n = \{1, 80, 100\}$ lead to the same value for $gaps(n)$). It follows from this observation that if we know the id of the first and the last element of n, then we can compute $gaps(n)$ without having to use formula (4.1), since it holds:

$$gaps(n) = \sum_{i=1}^{|n|-1} (tr_{i+1} - tr_i) = tr_{|n|} - tr_1 \tag{4.2}$$

assuming that $n = \{tr_1, ..., tr_{|n|}\}$.

Furthermore, we could canonicalize the value of $gaps(n)$ in order to be able to estimate how efficient a representation for a node is. So, for measurements and comparisons we could use $\frac{gaps(n)}{|n|}$ or $\frac{gaps(n)}{|T|}$, in order to obtain values that are independent of the number of elements of any particular node.

The Gaps of all Nodes

We can define the total gaps of all nodes of the storage graph $\Gamma = (N, R)$ as:

$$Gaps(N) = \sum_{n \in N} gaps(n) \tag{4.3}$$

In essence, $N = \{n_1, ..., n_{|N|}\}$ is a set of subsets of T.

If all sets of N are *pairwise disjoint* (i.e. $n_i \cap n_j = \emptyset$, $\forall i \neq j$, $i, j \in [1, |n|]$), then we can achieve the best assignment of identifiers for every node by assigning consecutive ids to the triples of every distinct node. Hence:

$$Gaps(N) \geq \sum_{n_i \in N} (|n_i| - 1) = \sum_{n_i \in N} |n_i| - \sum_{n_i \in N} 1 = |T| - |N|$$

On the other hand, the worst assignment in the above case is the one obtained when every node contains triples (ids) that cover the greatest possible range of values. Specifically, the first node will contain the triples with ids 1 and $|T|$ (these will be the min and max ids of that node), and hence, $gaps(n_1) = |T| - 1$ (according to the worst case of a node). Respectively, the second node will include the triples with ids 2 and $|T| - 1$ (since every triple occurs only once), and hence, $gaps(n_2) = (|T| - 1) - 2 = |T| - 3$. We can proceed analogously for the rest nodes, so the k-th node will contain the triples with ids k and $|T| - (k - 1)$, and thus $gaps(n_k) = (|T| - (k - 1)) - k = |T| - (2k - 1)$. Therefore:

$$\begin{aligned}
Gaps(N) &\leq (|T| - 1) + (|T| - 3) + ... + (|T| - (2|N| - 1)) \\
&= \sum_{i=1}^{|N|}(|T| - (2i - 1)) = \sum_{i=1}^{|N|}(|T|) - \sum_{i=1}^{|N|}(2i - 1) \\
&= |N||T| - 2\sum_{i=1}^{|N|} i + |N| \\
&= |N||T| - 2\frac{|N|(|N| + 1)}{2} + |N| \Leftrightarrow \\
Gaps(N) &\leq |N|(|T| - |N|)
\end{aligned}$$

Note that if all sets of N are pairwise disjoint then it always holds that $|T| - |N| \geq 0$, i.e. $|T| \geq |N|$, since the maximum number of sets in a partition of a set s, is equal to $|s|$ (that partition consists of singletons). In the extreme case where $|T| = |N|$ (and thus all nodes are singletons) the gaps are indeed 0.

Prop. 1 If a storage graph has $|N|$ sets and they are *pairwise disjoint*, then: $|T| - |N| \leq Gaps(N) \leq |N|(|T| - |N|)$.$\diamond$

Let us now discuss the case that leads to the worst reassignment for every node. That case comes up when N is not a *partition* of T, and thus there are overlaps between the ids of several nodes. The worst reassignment occurs when every node contains triples (ids) that cover the whole range of values. Specifically, for each node n_i we have $gaps(n_i) = |T| - 1$. Note that this case can occur only if the intersection of all nodes $n \in N$ is greater or equal than 2. Consequently: $Gaps(N) \leq |N|(|T| - 1) = |N||T| - |N|$. On the other hand, the case that could lead to the best reassignment if overlaps exist, occurs when the intersection between two nodes consists of triples with consecutive ids. Indeed in the first

node those ids should be at the beginning of the list, while in the second they should be at the end. For instance, consider four nodes: $n_1 = \{1, 2, 3\}$, $n_2 = \{2, 3, 4\}$, $n_3 = \{3, 4, 5, 6\}$, $n_4 = \{5, 6, 7\}$. In that case, we can achieve consecutive ids that differ by one, hence:

$$Gaps(N) \geq \sum_{i=1}^{|N|}(|n_i| - 1) = \sum_{i=1}^{|N|} |n_i| - |N|$$

We conclude that in the case that there are overlaps among the nodes' contents then:

Prop. 2 If a storage graph has $|N|$ sets and there are *overlaps* then: $\sum_{i=1}^{|N|} |n_i| - |N| \leq Gaps(N) \leq |N||T| - |N|$. $\diamond$

Table 4.1 summarizes the above bounds for $Gaps(N)$. Note that the above are the lower/upper bounds of all *possible* N that can emerge, having $|T|$ distinct triples and $|N|$ nodes. However, for a particular graph N this lower bound may not be the *greatest* lower bound, and the upper bound may not be the *least* upper bound.

	$Gaps(N)$	
Case	**Lower Bound**	**Upper Bound**
N is a *partition* of T	$\lvert T\rvert - \lvert N\rvert$	$\lvert N\rvert(\lvert T\rvert - \lvert N\rvert)$
N is not a *partition* of T	$\sum_{i=1}^{\lvert N\rvert} \lvert n_i\rvert - \lvert N\rvert$	$\lvert N\rvert\lvert T\rvert - \lvert N\rvert$

Table 4.1: Upper and Lower Bounds for $Gaps(N)$

4.2.1 Gaps and Storage Space

If we consider that the first id of every node is represented as a normal integer of four bytes and the successive (gapped) ids are coded using *unary* codes, then the storage space required measured in bits, and denoted by $Space_{CPOI}(N)$, is:

$$Space_{CPOI}(N) = 32 * |N| + Gaps(N) \tag{4.4}$$

Without a gapped and unary encoded representation, the required space by a 32-bit representation is:

$$Space_{POI}(N) = 32 * \sum_{i=1}^{|N|} |n_i| \tag{4.5}$$

We can write equation (4.5) also as:

$$\begin{aligned} Space_{POI}(N) &= 32 * (|N| + \sum_{i=1}^{|N|}(|n_i| - 1)) \\ &= 32 * |N| + 32 * \sum_{i=1}^{|N|}(|n_i| - 1) \end{aligned}$$

We can now express the condition under which CPOI requires less space than POI as: $Space_{CPOI}(N) < Space_{POI}(N) \Leftrightarrow$

$$32 * |N| + Gaps(N) \quad < \quad 32 * |N| + 32 * \sum_{i=1}^{|N|}(|n_i| - 1) \Leftrightarrow$$

$$Gaps(N) < 32 * \sum_{i=1}^{|N|}(|n_i| - 1) \tag{4.6}$$

Prop. 3 If $Gaps(N)$ is at most 32 times more than the lower bound of $Gaps(N)$, i.e. if $Gaps(N) < 32 * \sum_{i=1}^{|N|}(|n_i| - 1)$, then CPOI saves space. $\diamond$

By combining Prop. 2 and Prop. 3, it follows that we gain with a CPOI if the upper bound of $Gaps(N)$ according to Prop. 2 is less than the right hand of the condition of Prop. 3 (which guarantees gain), i.e. if : right of Prop. 3 > right of Prop. 2 $\Leftrightarrow$

$$32 * \sum_{i=1}^{|N|}(|n_i| - 1) > |N||T| - |N| \Leftrightarrow$$

$$32 * \sum_{i=1}^{|N|}|n_i| - 32 * |N| - |N||T| + |N| > 0 \Leftrightarrow$$

$$32 * \sum_{i=1}^{|N|}|n_i| - |N| * (|T| + 31) > 0$$

By expressing $\sum_{i=1}^{|N|}|n_i|$ as $|N| * avg(|n_i|)$, where $avg(|n_i|)$ is the average number of elements of a node, we can write: $32*|N|*avg(|n_i|) - |N|*(|T|+31) > 0 \Leftrightarrow 32*avg(|n_i|) > |T| + 31$. The above says that we can always save space with a CPOI when the number of distinct triples is not greater than 32 times the average number of elements of a node.

Prop. 4 If the number of distinct triples is not greater than 32 times the average number of elements of a node, then CPOI saves space. $\diamond$

Note that since Prop. 4 is based on the extreme case where we have the worst case for $Gaps(N)$, it specifies sufficient (not necessary) condition. This means that we can still gain with a CPOI even if the condition of Prop. 4 does not hold.

4.2.1.1 Uniform Codes for Ids

Instead of using four bytes per integer, or adopting a gapped and specially encoded representation, we could use a *uniform representation* of $\lceil \log_2 |T| \rceil$ bits for each id.

Obviously this leads to space savings. For example, if $|T| = 10^6$ then instead of 32 bits, we could use $\log_2 10^6 = 20$ bits. Hence, we can achieve compression ratio of $\frac{20*\sum_{i=1}^{|N|}|n_i|}{32*\sum_{i=1}^{|N|}|n_i|}100\% = \frac{20}{32}100\% \simeq 60\%$, i.e. the compressed space is around 60% of the original space. Hereafter, we will denote the space required by the above representation as $Space_{CPOI_U}(N)$, where the 'U' comes from *uniform*. It follows that:

$$Space_{CPOI_U}(N) = \sum_{i=1}^{|N|} |n_i| * \lceil \log_2 |T| \rceil \tag{4.7}$$

`CPOI` requires less space than `CPOIu` when:

$$32 * |N| + Gaps(N) \leq \sum_{i=1}^{|N|} |n_i| * \lceil \log_2 |T| \rceil \Leftrightarrow$$

$$Gaps(N) \leq \sum_{i=1}^{|N|} |n_i| * \lceil \log_2 |T| \rceil - 32 * |N|$$

Prop. 5 `CPOI` requires less apace than $\texttt{CPOI}_U$, iff $Gaps(N) \leq \sum_{i=1}^{|N|} |n_i| * \lceil \log_2 |T| \rceil - 32 * |N|$. $\diamond$

Note that in uniform representation the number of required bits is definite for a specific T, while in unary representation that depends on the assignment of triples' ids, and on the value of $Gaps(N)$. However we should note that the uniform representation is not very practical (for the problem at hand) due to the limitation of the number of integers that it can encode. The insertion of new versions with brand-new triples would require changing all triple identifiers and consequently the contents of all nodes. Instead, 32-bits or special encodings can be used indisputably when coding integers whose upper-bound cannot be determined beforehand (assuming $|T|$ is less than 4,3 billions, i.e. $|T| < 2^{32}$).

We can compare `CPOI` and `CPOIu` also wrt the worst/best case of `CPOI` (as expressed in Prop. 2 and equations (4.4) and (4.7)), i.e. in a way that does not require knowledge of $Gaps(N)$. In particular, the worst case of unary is better than uniform encoding, when:

$$32 * |N| + |N| * |T| - |N| \leq \sum_{i=1}^{|N|} |n_i| * \lceil \log_2 |T| \rceil \Leftrightarrow$$

$$\sum_{i=1}^{|N|} |n_i| * \lceil \log_2 |T| \rceil \geq |N| * (|T| + 31) \Leftrightarrow$$

$$avg(|n_i|) * \lceil \log_2 |T| \rceil \geq |T| + 31 \qquad (4.8)$$

It follows that unary representation is better than uniform when we have high average node size and small $|T|$, i.e. large overlapping.

The other way around, the best case of unary is worse than uniform encoding, and therefore uniform is certainly better than unary representation, when:

$$32 * |N| + \sum_{i=1}^{|N|} |n_i| - |N| \geq \sum_{i=1}^{|N|} |n_i| * \lceil \log_2 |T| \rceil \Leftrightarrow$$
$$31 * |N| \geq \sum_{i=1}^{|N|} |n_i| * (\lceil \log_2 |T| \rceil - 1) \Leftrightarrow$$

$$avg(|n_i|) * (\lceil \log_2 |T| \rceil - 1) \leq 31 \qquad (4.9)$$

Version Insertion Algorithms

The insertion algorithms are the same used in plain `POI`. The only alteration to the initial steps is the use of encoded lists instead of sets for the contents of the nodes. Specifically, when the contents of a node are retrieved (i.e. step 9 of ***Insert*** $\mathtt{POI}_p$ Insertion Algorithm), firstly the list is decompressed and then we obtain the set of triples ids of that node. Correspondingly, when the content of a node is set (i.e. step 24 of ***Insert*** $\mathtt{POI}_p$), firstly that content (set of triples' ids) is converted to a compressed list and then it is stored to the node.

Synopsis

The analytical comparison presented in this section identified conditions that guarantee space savings. Table 4.2 summarizes the occupied space for each compression method. Figure 4.2 illustrates the conditions under which one choice is certainly better (requires less space) than another. Notice that we can check whether some conditions hold or not (specifically those in Prop. 4 and equations (4.8) and (4.9)), even if we have a plain `POI`, i.e. without the need of a gapped representation. On the other hand the conditions that refer to $Gaps(N)$ require having a gapped representation and the value of $Gaps(N)$ depends on the way identifiers have been assigned. However we can have space savings even if the above conditions are not met, since the conditions rely on lower and upper

bounds. This motivates the experimental evaluation of Section ??, where we attempt to compare assignment approaches (aiming at approaching the lower bound of $Gaps(N)$), and investigate the amount of space saving that we can achieve.

Encoding	Storage Space for Nodes
POI	$32 * \sum_{i=1}^{\|N\|} \|n_i\|$
CPOI	$32 * \|N\| + Gaps(N)$
CPOIu	$\sum_{i=1}^{\|N\|} \|n_i\| * \lceil \log_2 \|T\| \rceil$

Table 4.2: Space (in bits) by the nodes of each compression method

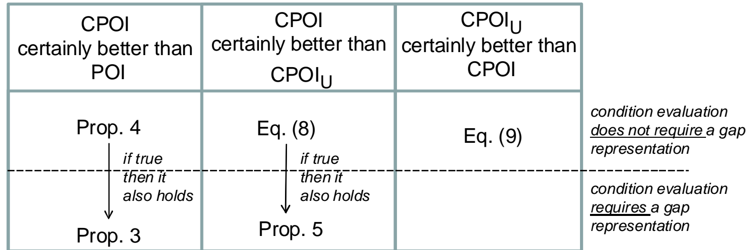


Figure 4.2: Synopsis of analytical results

4.3 Assignment Policies

The previous section provided lower and upper bounds, for $Gaps(N)$, and used them for identifying sufficient conditions that lead to compression, assuming the unary encoding. Our objective here is to find an assignment of ids that gives a small value for $Gaps(N)$. We need a method that takes as input the storage graph of POI and reassign the ids of triples so that each node to contain ids close to each other. Moreover, that method should be also efficient in time.

Below we discuss a number of approaches:

• **DEFAULT (id assigned at the first appearance of triple)**. According to this policy, each triple is assigned an id, the first time that it appears in one version, and it gets the smallest available integer. Obviously, this assignment does not depend on the structure of POI.

• **Node Size.** We order the nodes of the storage graph by their size (i.e. $|n_i|$) and then we assign ids to their triples starting either from the larger nodes, or from the smaller nodes. If we start from the larger nodes then these nodes are favored, so we can expect gains for them. An alternative approach is to start assigning identifiers starting from the smaller nodes. The motivation is that a small node can "waste" more bits per id, than a large node

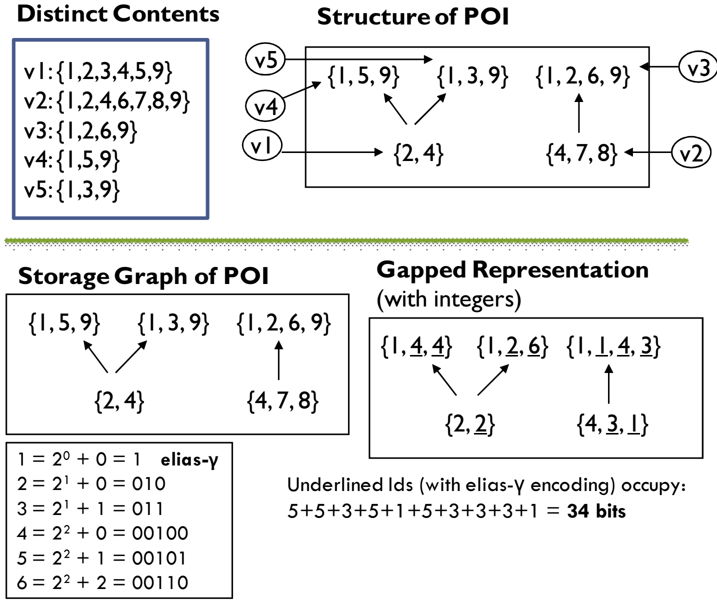


Figure 4.3: Storage graph of POI without and with a gapped representation

in a "bad" assignment. Specifically, consider two nodes n_i and n_j such that $|n_i| < |n_j|$ We can define the "bits per id of a node n" as follows: $bpi(n) = \frac{|gaps(n)|}{|n|}$. The worst (i.e. space consuming) case is $bpi(n) = \frac{|T|-1|}{|n|}$. It follows that $worst(bpi(n_j)) < worst(bpi(n_i))$ and this is the motivation for starting from small nodes. Obviously, nodes with size equal to 1, do not benefit from a gapped representation, therefore we assign identifiers to their triples at the end. In conclusion, each policy (from large or small nodes) has its own pros and cons.

Figure 4.3 illustrates an example that we will use to explain each one of the assignment approaches. Specifically, it shows the distinct contents of 5 versions and the corresponding graph of POI before and after the adoption of a gapped representation. Figure 4.4 depicts the procedure of reassignment by Size.

• **Triple Frequency.** For each triple in T we count the number of nodes that contain it and then we order the triples according to this number, getting a list of the form: $\langle$ appearing k times $\rangle$, $\langle$ appearing $k-1$ times $\rangle$, ... , $\langle$ appearing 1 time $\rangle$. Then we assign ids starting from the most frequently occurring triples, aiming at achieving consecutive (or close) ids in several nodes. If we want to reduce the maximum gap between an id and the rest ids, then it is beneficial to assign to that id the value $|T|/2$, since the max gap in that case is $|T|/2$. The ids that can give the maximum gaps (with the rest), are those

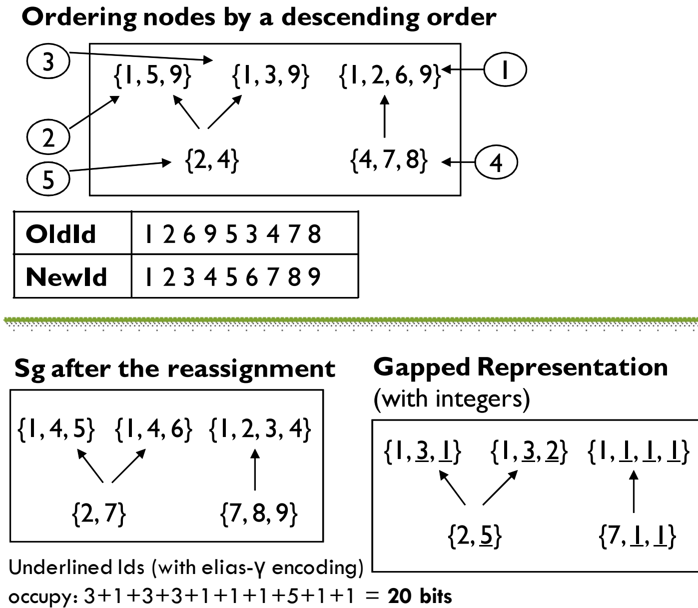


Figure 4.4: Assignment by Size

in the ends of the interval, i.e. 1 and $|T|$. So one reasonable approach would be to start giving ids to the frequent triples starting by $|T|/2$ and then continue using ids based on their absolute distance from $|T|/2$ (e.g. if $|T|/2 = 50$, then consume ids in the following order 50,51,49,52,48, and so on). It follows that the least occurring triples will get ids close to 1 or $|T|$. Moreover, note that a triple with f=1 does not affect other nodes than the one it appears in, so we do not have to be concerned about the ids of triples with $f = 1$ (i.e. no overlapping issues as they appear only in one node). Furthermore, we can exploit the fact that the triples in each frequency list are grouped so that those of the same node are adjacent, in order to achieve consecutive ids for the unique triples of almost every node (this method guarantees consecutive ids for the unique triples of every node).

Finally, if we consider that the distribution of the triples in nodes follows a power-law, then the expected size of the list with f=1 is much bigger than the sizes of the other lists.

Considering all the above, we start the assignment from the list of triples with f=1 and $id = 1$. When we have consumed the half of that list, we continue the assignment with the most frequently occurring triples. The rest lists follow (according to their frequency) and at the end we assign the remaining half of triples with f=1.

Figure 4.5 shows the procedure of reassignment by Frequency. We first create the frequency lists and then we start assignment from the list with f=1 until the half of that list (in our example we will consume three triples). We continue assignment with the most freuquent triples (i.e. triples appearing in three nodes, in two nodes and finally the remaining two triples of the first list). Thereafter, we update our storage graph with the new ids. The resulting gapped representation needs less space for its storage than the initial graph.

This reassignment is more expensive than the previous ones since it requires computing the frequency of all triples.

Assignment by Frequency

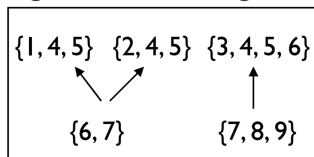
Frequency Lists

in3: 1,9
in2: 2,4
in1: 5,3,6,7,8

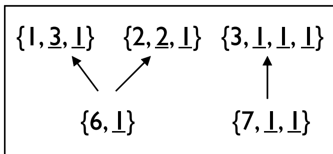
Start assigning until: $\left\lceil \frac{5}{2} \right\rceil = 3$

OldId	5 3 6 1 9 2 4 7 8
NewId	1 2 3 4 5 6 7 8 9

Sg after the reassignment



Gapped Representation (with integers)



Underlined Ids (with elias-γ encoding) occupy: 3+1+3+1+1+1+1+1+1+1 = **14 bits**

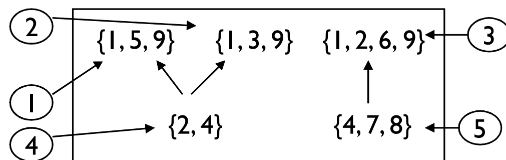
Figure 4.5: Assignment by Frequency

• **Storage Graph.** Another approach is to exploit the structure of the storage graph. Specifically, we can traverse the storage graph and assign ids by the order we encounter nodes. Let's first make some remarks regarding the storage graph: (a) If two nodes are connected (i.e. one is a direct or indirect child of the other) then they certainly have disjoint content. This means that overlaps can occur between nodes which are not connected (and clearly nodes of the same level fall into this category). (b) Each node of the storage graph is pointed to by at least one version. Since the contents of the versions that a node represents is equal to the triples stored at that node plus in all father nodes

of that node, the maximal nodes of the storage graph, i.e. those which have no father(s), store all the triples of the versions they represent. It follows, that if the variance of the version contents' sizes is small, then the maximal nodes are expected to store more triples than the nodes which are not maximal. Based on the above observations, one approach is to traverse the storage graph in a `Breadth-First Search (BFS)` manner, i.e. to start from high level nodes and then to descend. In this way it is expected that we will encounter larger nodes at the beginning and such nodes can (is not impossible to) overlap. Alternatively, and with the same motivation as Node size small, we could adopt a *reverse* BFS policy. In comparison to *Node Size* policy, the storage graph-based policies have the following benefit: successive (during the assignment) nodes have higher probability to has overlaps. Figure 4.5 depicts the procedure of reassignment by BFS.

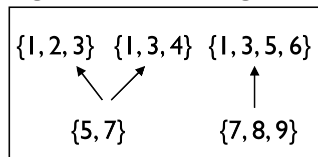
Assignment by Position (BFS)

Ordering nodes by a breadth-first traversal

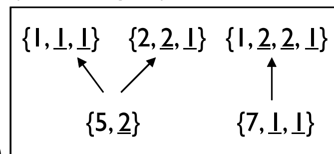


OldId	1 5 9 3 2 6 4 7 8
NewId	1 2 3 4 5 6 7 8 9

Sg after the reassignment



Gapped Representation (with integers)



Underlined Ids (with elias-γ encoding) occupy: 1+1+2+1+2+2+1+2+1+1 = **14 bits**

Figure 4.6: Assignment by BFS

With respect to computational cost, apart from *Frequency*, the rest reassignment policies do not require any preprocessing, and are very fast (roughly each method requires traversing the storage graph once).

4.3.1 Discussion

The assignment problem can be considered as an optimization problem. This case is approached by clustering. In essence, a clustering algorithm, such as k-means algorithm, aims to minimize the distances between relative objects.

4.4 CPOI and real world applications

CPOI can be exploited by several modern applications that require versioning services. Specifically, in those that have large storage space requirements and changes emerge frequently. For instance, in GO (that it is been updated daily) we could adopt a CPOI (for secondary memory storage) to store all versions. In that case, when a user requests for a specific version, we retrieve and return the whole content of that version. Also, in e-learning systems, where several users wish to be able to collaborate using parallel tracks in their tasks, a CPOI for main memory storage could be also adopted. In general, CPOI is an index aiming at reducing the overall space needed to store a large amount of data, and it is even more beneficial in cases that there are versions with subset-related content.

Chapter 5

Experimental Evaluation

5.1 Synthetic Datasets

To evaluate the above policies in large datasets we generated and used *synthetic* datasets. We have to note that related works mainly report results over synthetic datasets or over real datasets which are not versioned. The only exception is [23] which reports some results over 62 versions of GO (Gene Ontology). Moreover real versioned datasets usually form a *chain* of versions (not parallel tracks), and chains cannot be used for evaluating the potential of `POI` for space saving over multiple independent evolution tracks. For these reasons we decided to use synthetic datasets. We used three kinds of *synthetic* datasets.

`Dat1`. Using the synthetic KB generator described in [42], we created a dataset (`Dat1`) consisting of 1000 versions, each having 10,000 triples on average, where the size of each triple is 100 bytes (a typical triple size). The version generation method is illustrated in Figure 5.1. As in real case scenarios, a new version is commonly produced by modifying an existing version. In order to generate the content of a new version, we first choose at random a parent version and then we either *add* or *delete* triples from the parent contents. The difference in triples with respect to the parent content is 10%, i.e. 1000 triples. We have an additional parameter d that defines the probability to choose triple additions (so with probability $1-d$ we subtract triples). In this respect, we create versions whose contents are either supersets or subsets of the contents of existing versions. We experimented with d in the range of [0.5, 0.9] (we ignored values smaller than 0.5 as

subtractions usually do not exceed additions). For additions, we assumed that the 25% of the additional triples are triples which already exist in the KB (in the content of a different than the parent version), while the rest 75% are brand new triples. This is motivated by the fact that in a versioning system it is more rare to re-add a triple which exists in an old version and was removed in one of the subsequent versions, than to add new triples. Notice that as d increases, more new triples are created and less are deleted (so the total number of distinct triples increases).

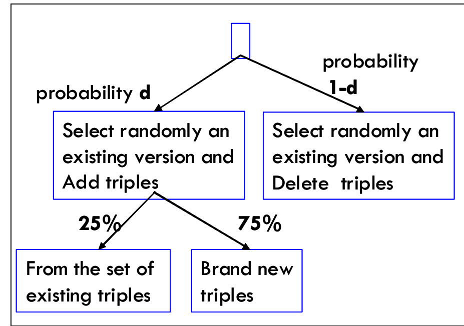


Figure 5.1: Synthetic dataset generation method for `Dat1`

`Dat2`. The `Dat1` consists of versions whose contents are proper subset or superset of the contents of the parent versions. To obtain a more realistic dataset, containing less $\subset$ relationships over contents, we created another dataset (`Dat2`) with a different version generation method which is illustrated at Figure 5.2. At first, we choose at random an existing version as a father and then we either only add/delete triples from the parent content or we make *both* triple additions and deletions. In the first case, we create versions whose contents are either subsets or supersets of the contents of existing versions, like `Dat1`. We have an additional parameter a that defines the probability to choose the first case (and $1 - a$ the second case). We experimented with a=0.3 (so the probability a version to be proper subset or superset of the parent version is 0.3). In the second case, we add and delete triples from the parent content. We use parameter x to specify the proportion of added triples (and $1 - x$ for the deleted triples). The parameter x ranges [0.5, 0.9], so $1 - x$ ranges (0.1, 0.5). We used $x = d$ for the production of the dataset used in the experiments. The difference in triples with respect to the parent content is 10% at the left branch, and the same holds for the the right branch. This proportion is represented by the parameter y. For additions, in both cases (i.e. in both a and $1 - a$ branches) we assumed that the 25% of the additional triples are triples which already

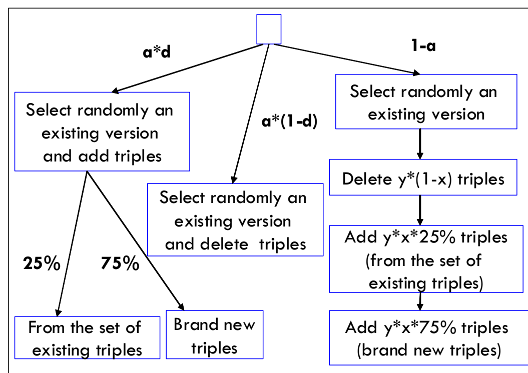


Figure 5.2: Synthetic dataset generation method for Dat2

exist in the KB, while the rest 75% are brand new triples. Dat2 consists of 1000 versions, each having 10,000 triples on average as well.

The frequency of triples in versions approximates a power law distribution. This hold for both the frequency of triples in versions, and the frequency of triples in the nodes of POI. Just indicatively, Figure 5.3 show in log scale the distribution of triples in versions (left), and in nodes (right) for Dat2 with $d = 0.7$ where $|T| = 531,500$. Notice that if we exclude the first 10,000 triples the rest 521,500 triples follow a power-law distribution as their plot in the log scale approximates a straight line.

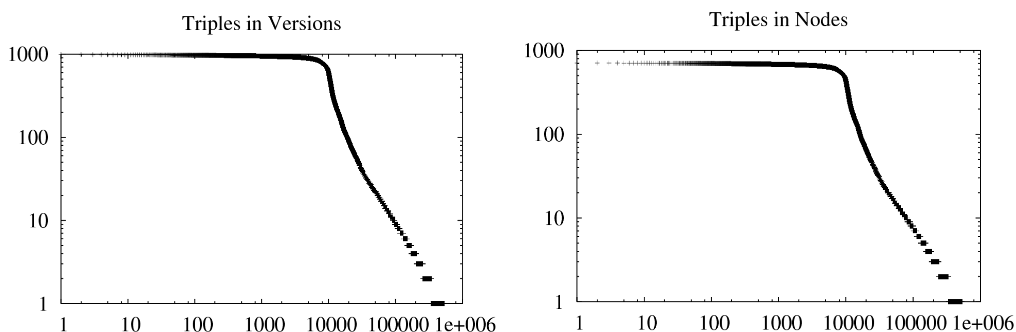


Figure 5.3: The distribution (in logscale) of triples in versions (left) and in nodes (right) for Dat2 for d=0.7 (Y: frequency, X: triples)

After feeding these datasets to POI we observed that the storage graph of POI for Dat1 has a large number of edges, while the storage graph for Dat2 has a more flat morphology with less connections and depth. Specifically, the average depth of the graph for Dat1 (resp. Dat2) in our experiments (and assuming all d values) is 3 - 5.5 (resp. 1.4), while the max depth for Dat1 (resp. Dat2) is 13 - 14 (resp. 5).

Dat3. Lastly, we created a third synthetic dataset (Dat3) where no version is subset of

another, versions share a lot of common triples and the triple frequency follows a power-law (for being close to real datasets). The above characteristics imply that `POI` cannot be effective, as it has the same space requirements as the `IC` approach. We expect that `CPOI` would be much better than `IC` and we want to estimate how `CPOI` performs in comparison with `CB` approach. `Dat3` consists of 1000 versions, each having 10,000 triples on average, and $|T|$=400,000. It was produced as follows: at first we compute the frequency of each distinct triple[1], creating a list of the form $< tripleId, \#versions >$. Then we start filling each version's content consuming the ids of every list consecutively. Specifically, once we have inserted one triple in #versions (according to the list), we continue with the next triple starting insertions from the subsequent version of the one that we added the previous tripleId last.

5.2 Real Datasets

`GO versions`. Moreover, we conducted experiments over only a few versions of GO. We used the RDF/S dumps from the Gene Ontology (GO) project[2]. This dataset contains 27,640 classes, and 1,359 property instances and uses 126 properties to describe genes. We used only a few versions, specifically 6 versions (v.16-2-2008, v.25-11-2008, v.24-3-2009, v.5-5-2009, v.26-5-2009, v.22-9-2009) which *are not successive*, so a lot of changes exist between them, and indeed none of them is subset of another. We have to note that if we had used a higher number of versions, then that would be an advantage for `POI`.

5.3 Discussion over Datasets

Summarizing the main characteristics of the above datasets we could mention:

`Dat1` : Consists of versions whose contents are proper subset or superset of the contents of the parent versions.

`Dat2` : It is the more realistic dataset, as it contains versions that are subset/superset-related with other versions and versions that are not subset/superset-related with

[1] Using the formula $Freq(tripleId) = (\frac{100.230}{tripleId})^{0,6} + 25$, thus each triple appears at least to 25 versions.
[2] www.geneontology.org/

any other version as well. However, in the later case, they have common triples with other versions (overlapping among their content).

`Dat3` : Consists of versions that share a lot of common triples and no version is subset of another. This dataset captures the extreme case where `POI` (and `CPOI`) graph has a flat structure, which is the worst case for `POI`.

`GO` : Like `Dat3`, no version is subset/superset of any other version. Also, the versions that have been used are not successive, so a lot of changes exist between them.

5.3.1 Metrics for Comparing Datasets

Subset Content Density

In order to measure how efficient `POI` can be in comparison with other strategies with more correctness, we are taking into account one more parameter that reflects the storage graph's form as far as the subset content relationships among versions. We introduce *SubsetContentDensity* (`SCD`) norm, which express the fraction of versions whose content is subset of other versions' content. Specifically, we define:

$$SubsetContentDensity(V) = \frac{\{\ (v, v') \in V^2 \mid v.cont \subseteq v'.cont, v \neq v'\}}{|V|(|V|-1)} \tag{5.1}$$

Its values range in [0, 1]. We get 0 when there is no version's content that constitutes subset of any other version's content, while we get 1 when all versions have the same content, so each version's content constitute subset of every other version's content. The intermediate values are the most interesting as we are expecting that as `SCD` is increasing so efficient `POI` turns.

Futhermore, we measured the average depth of the versions and the max depth of the storage graph's nodes. The first one is defined as:

$$AverageDepth = \frac{\sum_{nodes}(number\ of\ versions\ point\ to\ this\ node) * (node\ depth))}{number\ of\ versions} \tag{5.2}$$

We use the above measures to compare the graph form between the datasets (`Dat1` and `Dat2`). The results are shown in Figures 5.4, 5.5. As we can observe, `Dat1` generates a more coherent graph's structure than `Dat2`. The graph produced by `Dat1` spreads in

depth in contrast with the graph produced by Dat2, that spreads in depth. This is a presumable result, as in Dat1 the content of every version is super or subset related with the content of another version, while in Dat2 that is not the case. Obviously, the above metrics are referred only to Dat1 and Dat2, as they are meaningless for Dat3 and GO.

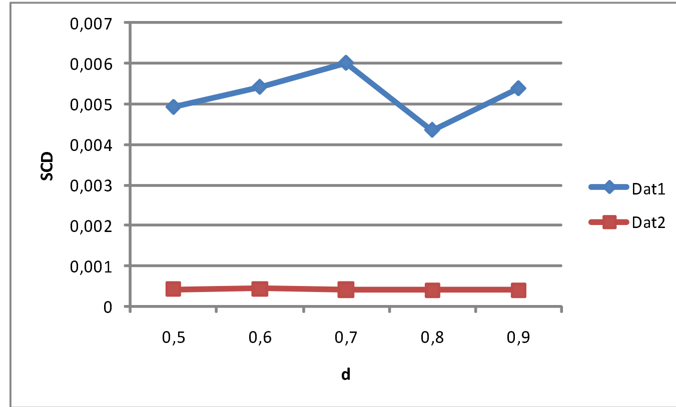


Figure 5.4: SCD of Storage Graphs' for Dat1 and Dat2

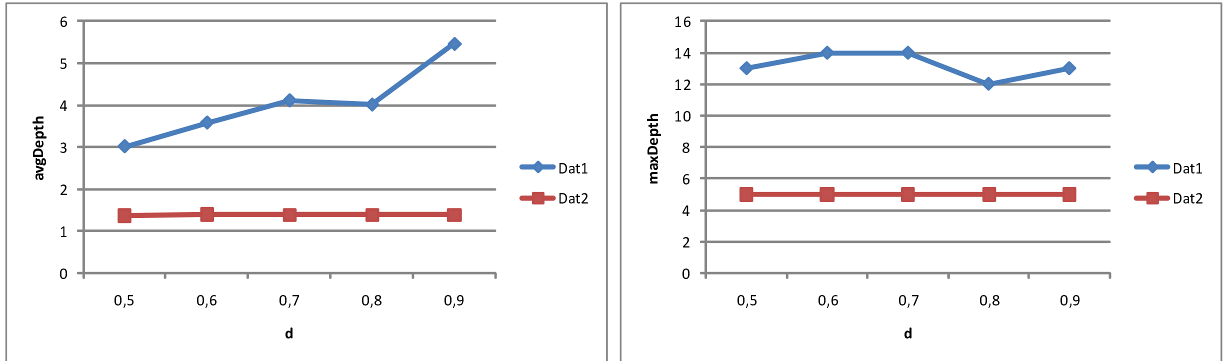


Figure 5.5: AverageDepth (left) and MaxDepth (right) of Storage Graphs' for Dat1 and Dat2

5.4 Compared Options

For each dataset we compared the following options:

a) plain POI (i.e. no reassignment, 32-bit integer encoding),

b) CPOI DEFAULT (i.e. no reassignment),

c) CPOI after ordering triples randomly (*Random*),

d) `CPOI` after ordering triples wrt node size (*Size* in descending order and *Size_Reverse* in ascending order),

e) `CPOI` after ordering triples wrt their frequency in nodes (*Frequency*),

f) `CPOI` after ordering triples wrt their position in the storage graph (*BFS* and *BFS_Reverse* for reverse traversal),

g) `CPOIu` using uniform encoding.

We compared the above options with respect to the following aspects: storage space for the node contents, total space, and time to assign the identifiers. For `CPOI` we tested two encodings: Unary (as it is very close to the analytical results), and Elias-γ. The latter is better than unary because for an integer n, Elias-γ requires $2\lfloor \log_2 n \rfloor + 1$ bits, while unary requires n bits. Here we explain briefly how these techniques work.

Unary encoding

A positive integer n is represented by n bits, specifically by *n-1* ones followed by one zero (or *n-1* zeros followed by an one). As an example, lets see the encoding for numbers 1-17:

Number	Encoding
1 =	0
2 =	10
3 =	110
4 =	1110
5 =	11110
6 =	111110
7 =	1111110
8 =	11111110
9 =	111111110
10 =	1111111110
11 =	11111111110
12 =	111111111110

13 = 1111111111110

14 = 11111111111110

15 = 111111111111110

16 = 1111111111111110

17 = 11111111111111110

Elias-γ encoding

A number x is represented by a concatenation of two parts:

a. a unary code for $1 + \lfloor logx \rfloor$ and

b. a code of $\lfloor logx \rfloor$ bits that represents the values of $x - 2^{\lfloor logx \rfloor}$ in binary

The encoding for numbers 1-17 are:

`Number`	`Encoding`
$1 = 2^0 + 0$	1
$2 = 2^1 + 0$	010
$3 = 2^1 + 1$	011
$4 = 2^2 + 0$	00100
$5 = 2^2 + 1$	00101
$6 = 2^2 + 2$	00110
$7 = 2^2 + 3$	00111
$8 = 2^3 + 0$	0001000
$9 = 2^3 + 1$	0001001
$10 = 2^3 + 2$	0001010
$11 = 2^3 + 3$	0001011
$12 = 2^3 + 4$	0001100
$13 = 2^3 + 5$	0001101
$14 = 2^3 + 6$	0001110
$15 = 2^3 + 7$	0001111
$16 = 2^4 + 0$	000010000
$17 = 2^4 + 1$	000010001

5.5 Results

We compared the storage space for the nodes' content of POI, CPOIu, and CPOI with Unary and Elias-γ encoding. Firstly, we present the results for Dat1 and Dat2 and thereafter for GO and Dat3, as they demonstrate extreme cases for POI.

Results for Dat1 and Dat2

The results for Dat1 (resp. Dat2) are shown in the left (resp. right) part of Figure 5.6. It is evident that in both datasets CPOI with Elias-γ is by far the best compression method, achieving a 7.5%-8.5% compression ratio. CPOIu comes second achieving only 59%-63% compression ratio. Third comes CPOI with Unary encoding. However, in Dat1 and for the case where d=0.9 (i.e. when we have almost only additions, and thus a lot of $\subseteq$-relationships) it is fourth, i.e. worse than plain POI. This happens because as d increases, $|T|$ increases as well, while $\sum_{i=1}^{|N|} |n_i|$ decreases (because of the subset/superset relations). Hence, the required space for POI decreases. That does not hold for CPOI, as it depends only on $Gaps(N)$.

Regarding the d value, since in Dat2 we have less subset/superset relations wrt Dat1, as d increases, $|T|$ and $\sum_{i=1}^{|N|} |n_i|$ increases as well, consequently, the required space for POI increases. Regarding CPOI, its space in Dat2 increases as in Dat1 since it depends on $Gaps(N)$ and for larger d we have larger $|T|$.

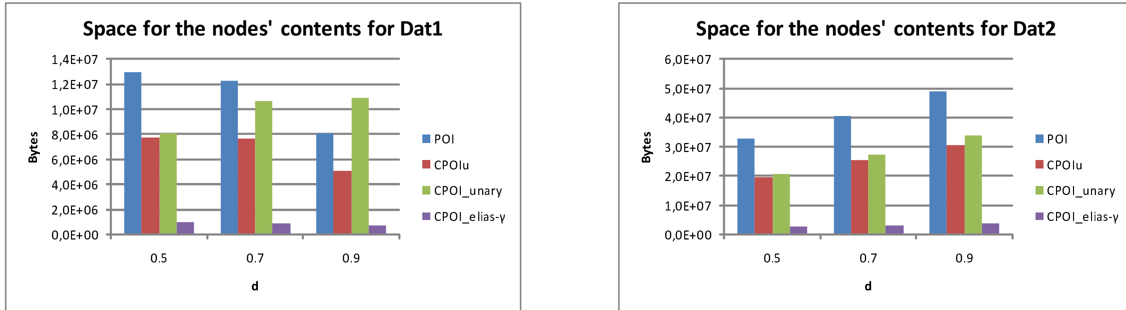


Figure 5.6: Space for the nodes' contents for Dat1 (left) and Dat2 (right) of POI, CPOIu and CPOI (for Unary and Elias-γ encoding).

Regarding the reassignment policies, Figure 5.7 shows comparative results for Dat1 and Dat2 using Elias-γ. In both datasets *BFS* is the best reassignement (with compression ratio 7.5%-8%), outperforming CPOI DEFAULT. All the others reassignment policies are outperformed by CPOI but none of them is worse than POI or CPOIu. The left part of

Figure 5.9 groups and ranks reassignment policies according to the compression ratio they achieve for each dataset.

Regarding nodes' contents, another interesting observation is that a gapped representation with Elias-γ gives around 20% compression ratio (even with a random id assignment). A "good" (i.e. at least not random) assignment (with Elias-γ encoding) gives around 8% compression ratio (i.e. three times less space) in comparison to the random assignment.

The time required for the reassignment is short for all policies ranging from 4 to 11 secs for `Dat1`, and from 14 to 46 secs for `Dat2`[3]. The fastest policy is *BFS*, while the slowest is *Frequency*.

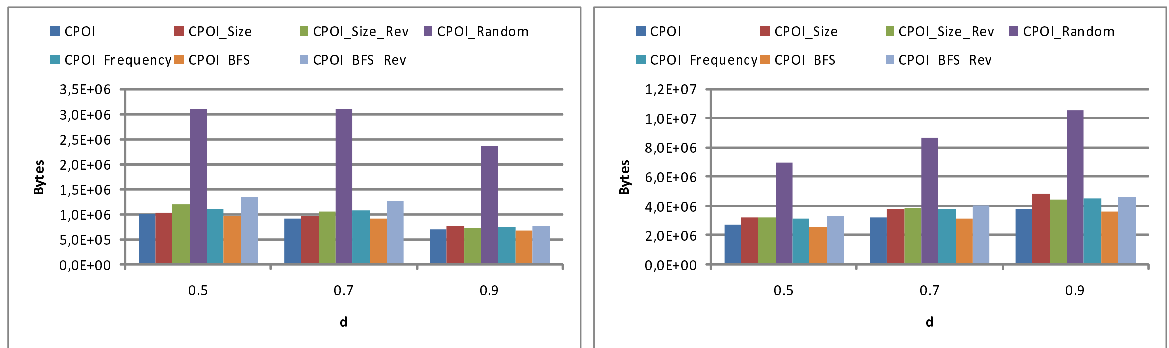


Figure 5.7: Nodes's space for `Dat1` (left) and `Dat2` (right) of `CPOI` using Elias-γ encoding for various assignment policies.

Unary Although the Unary is not as good as Elias-γ (as we have seen earlier), we comparatively evaluated the reassignment policies also for this encoding since the space requirements of that encoding is what is captured by the analytical results of Section 4.2. Figure 5.8 show comparative results for `Dat1` and `Dat2`. We observe larger variations in comparison to Figure 5.7, since Unary encoding better reflects $Gap(N)$ than Elias-γ. We can see that `CPOI` DEFAULT outperforms all reassignment policies in these experiments. Regarding the other policies, in `Dat2` *BFS* is the best, followed by *Size_Reverse* and *Frequency*, while in `Dat1` the above policies are interchanged with each other for different values of d. In both datasets the worst is *Random*, followed by *Size* and *BFS_Reverse*. The right part of Figure 5.9 groups and ranks the reassignment policies according to the

[3]The implementation is in Java and all experiments were carried out in main memory of a PC with Pentium(R) IV 3.40 Ghz, 1,49 GB Ram, and Windows XP.

compression ratio they achieve for each dataset. We can see that `POI` outperforms all re-assignments policies in `Dat1`, while in `Dat2` *BFS*, *Size_Reverse* and *Frequency* outperform `POI`.

The time for the reassignment for all policies is again slow ranging from 4 to 11 secs for `Dat1` and from 14 to 40 secs for `Dat2`. The fastest policy is *BFS*, while the slowest is *Frequency*.

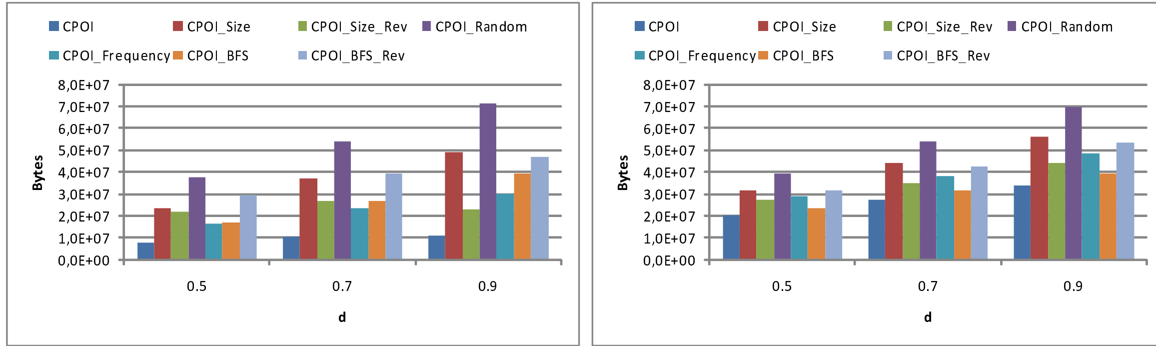


Figure 5.8: Nodes's space for `Dat1` (left) and `Dat2` (right) of `CPOI` using unary encoding for various assignment policies.

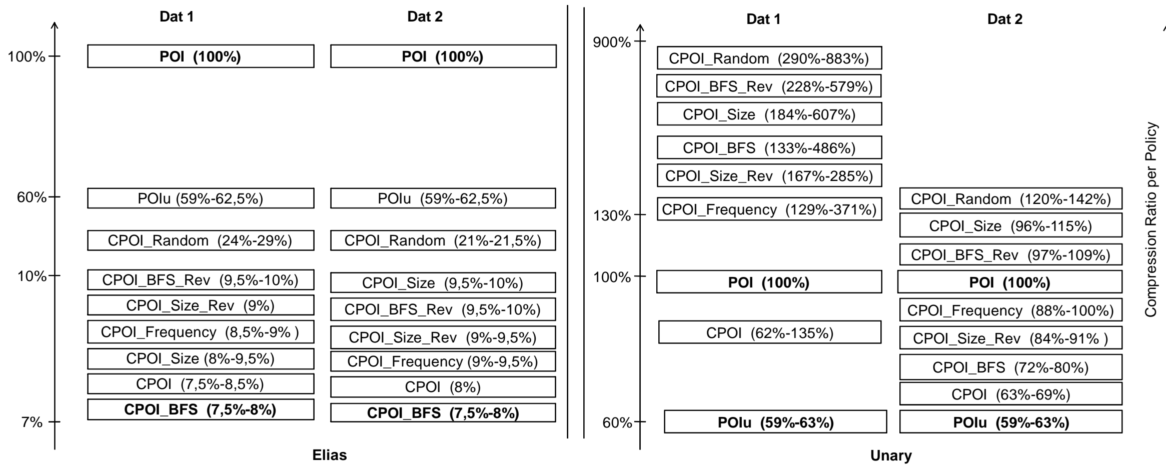


Figure 5.9: Ranking of reassignment policies wrt their compression ratio for Elias-γ encoding (left) and unary encoding (right).

Total Space

So far we have compared the space required by the sets of triple ids. To quantify the overall benefit of using `CPOI`, we compared the total storage space requirements of `IC`, `CB`, `POI`, `CPOI` (Elias-γ and DEFAULT id assignment). We also conducted experiments over a CB Dictionary (CBD) approach, that in comparison with CB, does not store the

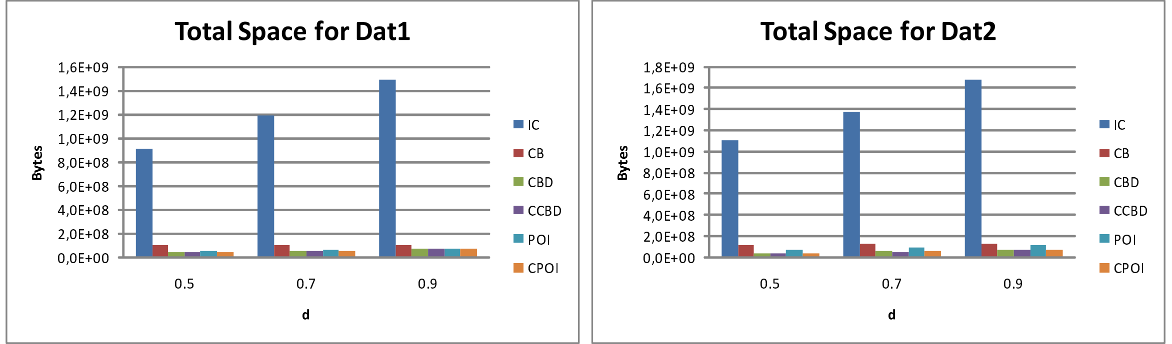


Figure 5.10: Total space for `Dat1` (left) and `Dat2` (right) of `IC`, `CB`, `POI`, for various d values

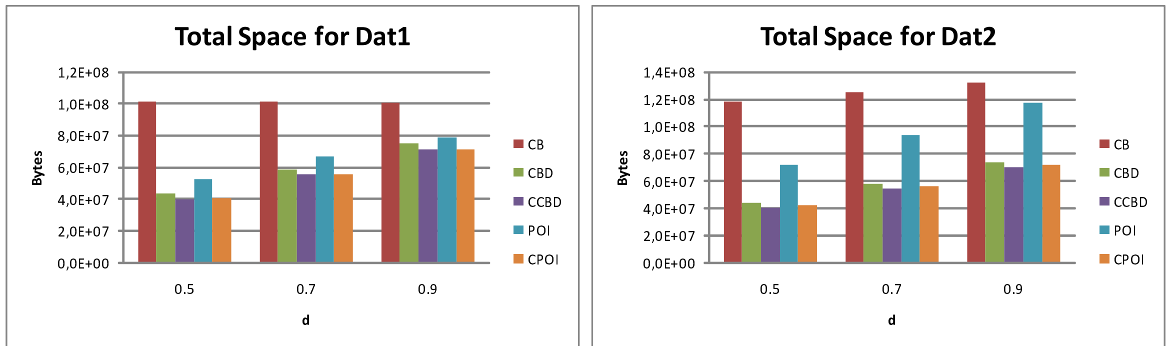


Figure 5.11: Total space for `Dat1` (left) and `Dat2` (right) without `IC`, for various d values

entire triples of each delta, but it keeps a mapping of triples with identifiers and stores only the identifiers (integers) in each delta node. This approach is certainly better than `CB` for those triples that occur at least in two deltas. We extended this approach with experiments over gapped identifiers and variable length encoding. We shall use the term CCBD (Compact CBD) to refer to it.

The left (resp. right) part of Figure 5.10 shows the space requirements for `Dat1` (resp. `Dat2`), for various values of d (0.5 - 0.9). Figure 5.11 depicts the same space requirements without `IC` approach. We can see that `CPOI` and `CCBD` are always better than the other policies. `CCBD` slightly outperforms `CPOI` in both datasets.

The main results can be summarized as follows. Using a `CPOI` with *BFS* reassignment and Elias-γ we can achieve compression ratio of 8% of the size of nodes' content and about 60% compression ratio over the total size of the graph, while with the adoption of a `CCBD` we can achieve 56% - 60% compression ratio (over the total size of the `POI` graph).

Comparing the total size of `CPOI` with the other two methods (`IC` and `CB`), the former requires only about 4.3% (4.4%-4.8% for `Dat1` and 3.8%-4.3% for `Dat2`) of the space

needed for `IC` and 35.4%-70.9% (40.1%-70.9% for `Dat1` and 35.4%-54.8% for `Dat2`) of the space needed for `CB` approach.

Regarding the total size of `CCBD` with the other two methods (`IC` and `CB`), the former requires only about 4.3% (4.4%-4.8% for `Dat1` and 3.7%-4.2% for `Dat2`) of the space needed for `IC` and 34.1%-70.7% (39.8%-70.7% for `Dat1` and 34.1%-52.9% for `Dat2`) of the space needed for `CB` approach. Comparing the total size of `CCBD` with `CPOI`, the former requires about 96.3% - 99.8% (99.1% - 99.8% for `Dat1` and 96.3% - 96.5% for `Dat2`).

Results for `GO` and `Dat3`

The results for `GO` regarding the storage space of nodes are shown in Figure 5.12 (left). We observe that unary and Elias-γ encodings offer significant gains. Regarding the reassignment policies Figure 5.12 (right) shows comparative results using Elias-γ and encodings. Now Figure 5.13 shows the total storage space. The first observation is that `POI` is better than `IC`. This is due to the storage gains achieved by keeping triple ids (in `POI`) instead of triple strings (in `IC`). `POI` is better than `CB` even if the graph is flat. So does `CBD`. `CPOI` and `CCBD` are better approaches (`CCBD` slightly outperforms `CPOI`).

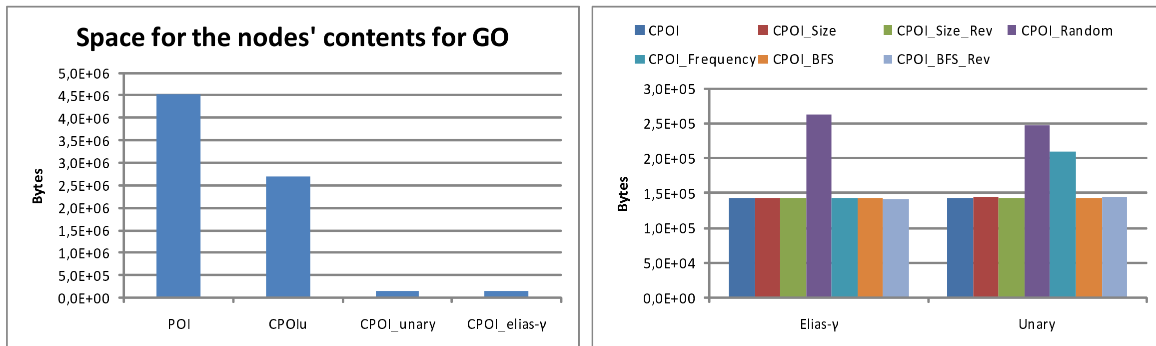


Figure 5.12: Nodes' space in GO.

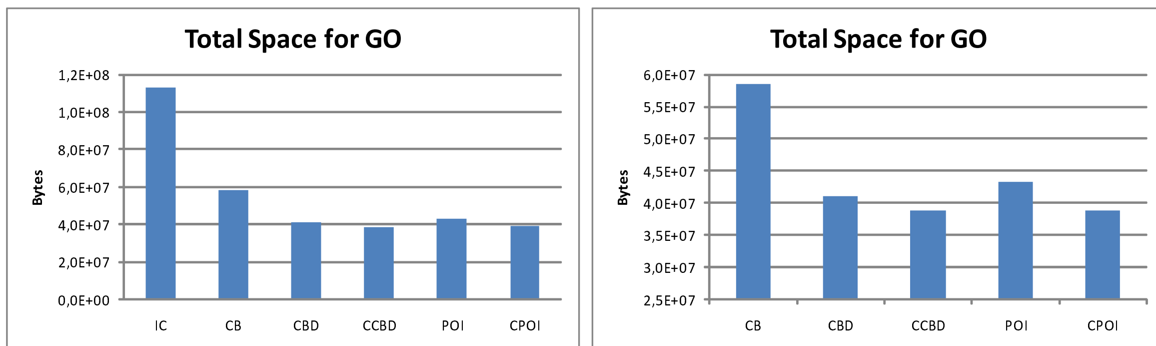


Figure 5.13: Total storage space in GO with `IC` (left) and without (right).

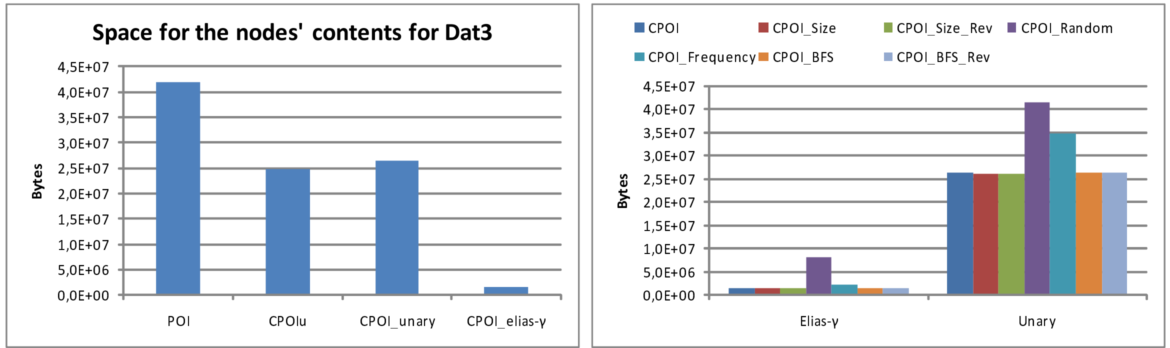


Figure 5.14: Nodes' space in `Dat3`.

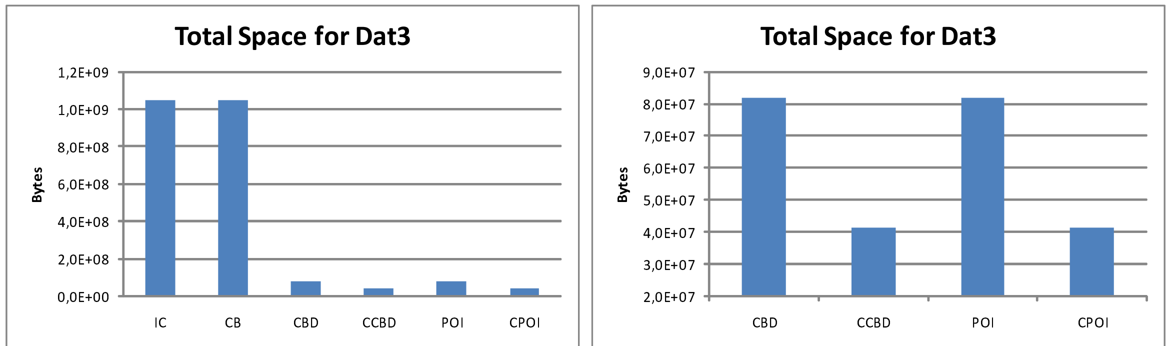


Figure 5.15: Total storage space in `Dat3` with `IC` (left) and without (right).

The results for `Dat3` for the nodes' space are shown in the left part of Figure 5.14, while comparative results using Elias-γ and unary encodings are shown in the right part. `CPOI` outperforms `POI` as we expected. `CPOI` with Elias-γ gives the best compression ratio, i.e. 3.5%. `CPOIu` comes second achieving 59.4% compression ratio, while `CPOI` with Unary encoding achieves 63% compression ratio. Regarding the reassignment policies, the best reassignement is *Size_Rev* for unary and *Size* for Elias-γ (with compression ratio 62% and 3.3% respectively). *Size*, *Size_Rev* and *BFS_Rev* outperform `CPOI` DEFAULT. The other reassignment policies are outperformed by `CPOI` DEFAULT, but none of them is worse than `POI` or `CPOIu`.

Regarding the total storage space the results are shown in Figure 5.15. `POI` and `CBD` are much better than both `IC` and `CB` (as they store each distinct triple once), while `CB` is slightly worse than `IC`. `CPOI` and `CCBD` are even better approaches with `CCBD` to slightly outperform `CPOI`. We conclude that even if no version is subset of another, and we have a significant number of versions, then `CPOI` and `CCBD` are significantly better than the `CB` approach.

Regarding the comparison of POI (and CPOI) with CBD (and CCBD), in CBD the cost to reconstruct the contents of a version is history dependent (and may more expensive than POI), and POI offers faster subset checking. On the other hand the addition of a version in CBD can be faster than POI since there is no need for checks to determine its placement in the storage graph. In general we can say that the price to pay, in comparison to IC and CB, is slower additions of new versions and the adoption of gapped and encoded identifiers makes them slower. Although the insertion algorithm of POI exploits the structure and semantics of the storage graph, its plain version sometimes requires 100 seconds for the addition of a version. However, the variation with cache which has been proposed (in [42]), reuses results of set union operations, and leads to an average insertion time less than one second. Specifically, the average insertion time for Dat1 (resp. Dat2) was 0.3-0.8 sec (resp. 0.8-1 sec). The gapped and encoded identifiers incur only a small overhead, specifically in our experiments for Dat1 (resp. Dat2) the insertion time was 0.8-2.4 sec (resp. 3.2-4.4 sec).

5.5.1 Summary of compression ratios

Table 5.1 (resp. Table 5.2) summarizes the compression ratios of CPOI[4] (resp. CBD) for each dataset in comparison with the rest policies.

Compression Ratio	$\frac{CPOI}{IC}$	$\frac{CPOI}{CB}$	$\frac{CPOI}{CBD}$	$\frac{CPOI}{CCBD}$	$\frac{CPOI}{POI}$	$\frac{CPOI_R}{CPOI}$
Realistic						
Dat1	4.4% - 4.8%	40.1% - 70.9%	93.1% - 95.5%	100.2% - 100.9%	77.2% - 90.6%	99.9% - 100%
Dat2	3.8% - 4.3%	35.4% - 54.8%	95.4% - 98%	103.6% - 103.8%	58.1% - 61.5%	99.7% - 99.8%
Anti-correlated						
6 GO versions	34.4%	66.5%	94.7%	100.2%	89.9%	100%
Dat3	4%	4%	50.7%	100%	50.7%	99.8%

Table 5.1: Compression ratios of CPOI in comparison with the rest policies

[4] CPOI_R refers to the best reassignment for each dataset.

Compression Ratio	$\frac{CCBD}{IC}$	$\frac{CCBD}{CB}$	$\frac{CCBD}{CBD}$	$\frac{CCBD}{POI}$	$\frac{CCBD}{CPOI}$	$\frac{CCBD}{CPOI_R}$
Realistic						
`Dat1`	4.4% - 4.8%	39.8% - 70.7%	92.2% - 95.3%	76.5% - 90.4%	99.1% - 99.8%	99.2% - 99.8%
`Dat2`	3.7% - 4.2%	34.1% - 52.9%	91.9% - 94.6%	56% - 59.4%	96.3% - 96.5%	96.6% - 96.7%
Anti-correlated						
`6 GO versions`	34.4%	66.4%	94.5%	89.7%	99.8%	99.8%
`Dat3`	4%	4%	50.7%	50.7%	100%	100.2%

Table 5.2: Compression ratios of `CBD` in comparison with the rest policies

5.5.2 Experimental Results vs Analytical Results

Regarding `Dat1` and `Dat2` and with respect to the analytical comparison presented in Section 4.2, the conditions that do not require computing $Gaps(N)$, i.e. those in Prop. 4 and equations (4.8) and (4.9), are not satisfied, hence by looking at the features of our datasets ($|n_i|, |T|, avg|n_i|$) we cannot conclude which approach guarantees space benefits.

Referring to the conditions that require a gap representation (i.e. those in Prop. 3 and Prop. 5), by considering the DEFAULT reassignment, Prop. 3 holds for `Dat1` for all $d \in [0.5 - 0.7]$, while for `Dat2` for $d \in [0.5 - 0.9]$. Prop. 5 does not hold for any value of d neither for `Dat1` nor for `Dat2`. Regarding the other reassignment policies Prop. 3 holds for `Dat2` for BFS, Frequency and Size_Rev, while for Size and BFS_Rev it holds only for $d = 0.5$. These results agree with the experimental results.

Regarding `GO` the conditions that do not require computing $Gaps(N)$, i.e. those in Prop. 4 and equation (4.8) hold, so `CPOI` always guarantee space savings over both `POI` and `CPOIu`. Consequently, the conditions that require the $Gaps(N)$ value, i.e. Prop. 3 and 5, hold too.

As far as `Dat3` the conditions that do not require computing $Gaps(N)$, are not satisfied.

5.6 Discussion

5.6.1 Operations over a CPOI

In a versioning system, the main scenario contains additions of versions. Updates or deletions of existing versions are rare. The addition of a new version can change the

structure of the storage graph and thus the contents of some of its nodes. It follows that the addition of a new version can affect the gains from the assignment/encoding. Let's analyze this in more detail. Let n be the set of triples of a new version to be added. If $n \cap T = \emptyset$ then the storage graph is not affected and the encoding of the new triples can be optimal. Let now consider the case where $n \cap T \neq \emptyset$, and let n_{ex} denote the existing triples and n_{new} the brand new ones, i.e. $n_{ex} = n \cap T$ and $n_{new} = n \setminus T$. If n_{ex} can be expressed as union of nodes that already exist in the graph, and n is not subset of any of the existing versions, then for n_{ex} we do not have to represent any triple, while for n_{new} we can use an optimal encoding for them (since they are brand new triples). However, if n_{ex} cannot be expressed as union of nodes that already exist in the graph, then those which cannot be expressed as union of existing nodes will be represented explicitly using their current ids. In the general case where n is placed in the internal part of the graph, i.e. it has fathers and children (or only children), consequently $n \subseteq T$, then only the contents of the direct children of n in the graph will be changed. It follows that the addition of new versions does not significantly affect $Gaps(N)$. This is probably the reason why `CPOI` DEFAULT performs well in the experiments, i.e. its performance is very close to the best (BFS) policy. Note that the DEFAULT assignment, actually simulates a scenario with additions of versions.

However, and to further reduce space, we can periodically re-assign/encode identifiers (e.g. using the BFS method that gave the best results in our experiments). This is analogous to the policy that is usually adopted in the inverted index of an IRS (or WSE); since the addition of new documents affects the posting lists of the words that occur in that document, periodically the documents are clustered for assigning close identifiers. Since clustering is an expensive operation, it is done periodically, or other heuristics (like the one presented in [35]) are employed to reduce the clustering cost. In our case, the problem is easier since BFS is not computationally expensive. However, if we consider `CPOI` DEFAULT as baseline, then BFS yields a compression ratio in 95%-97% for nodes contents and in 99.7%-99.9% for the total space. Since the improvement is not significant, we can conclude that the reassignment is not necessary.

Chapter 6

Conclusions and Future Work

6.1 Synopsis and Key Contributions

The provision of versioning services over RDF datasets is of significant interest for several modern applications for achieving, preservation and provenance of the data. In this thesis we concentrate on this problem by providing versioning services and particularly archiving of RDF data.

We proposed a compact representation for `POI` based on gapped representation of triple identifiers and variable-length identifier encodings. We analyzed the space requirements of this representation and identified sufficient conditions that guarantee compression comparing to plain `POI`. Subsequently, we conducted a large number of experiments over various synthetic datasets, and using several methods for assigning identifiers.

We have also seen that even in anti-correlated for `POI` datasets, `CPOI` is still better than the `CB` approach. Regarding reassignment policies, we noticed that identifier assignment in a *first-in-first-served* basis is almost as good as reassignment policies, and from this we can conclude that identifier reassignment is not necessary.

Apart from RDF, it is not hard to see that `CPOI` is beneficial for applications that require keeping several sets (there are several applications of set-valued attributes [37]) and with a significant probability they are $\subseteq$-related. For instance, social networking systems have to keep information about large numbers of users. `CPOI` could be exploited for versioning groups. Since groups usually grow, rather than shrink, a significant number of versions are expected to be $\subseteq$-related.

The price to pay is slower insertion times for new versions (in comparison with `IC` and `CB`). However, a version is added only once, therefore `CPOI` is a beneficial choice for the intended application scenario.

Since we do not deal with the compression of the table that keeps the distinct triple strings and their ids, techniques like those proposed in [29] [13], are complementary to `CPOI` and if used *together* will further reduce the overall space.

6.2 Directions for Further Research

Issues that are worth further research include: to elaborate on the problem of exploiting that structure in order to store a bulk amount of data (in secondary memory), and to elaborate on reassignment techniques which are based on clustering although such methods are expected to be significantly computationally more expensive.

Regarding the former, if the storage graph $\langle \Gamma, stored \rangle$ of `POI` does not fit in main memory, then only Γ could be kept in main memory, while the function *stored* (i.e. the stored triples at each node) could be kept in secondary memory.

Figure 6.1 illustrates an example of storing data using secondary memory. We consider 6 knowledge bases (KBs), whose contents are:
a0 = {1, 2, 3, 4, 5},
a1 = {1, 2, 3, 4, 5, 6},
a2 = {1, 2, 3, 4, 7, 8},
b0 = {1, 2, 3},
b1 = {1, 2, 3, 4, 5, 6}, and
b2 = {1, 2, 7},
where 1...8 are triple identifiers.

For instance, a relational DBMS could be used with schema `Stored(vid,tid)` where `t`*id* is the triple identifier. To retrieve the contents of a version i, $Br^t(i)$ is needed to be computed using Γ and then a disjunctive query will be send to the db (with all ids in $Br^t(i)$).

The employment of ORBMSs (Object-Relational DBMSs) is beneficial (with respect to

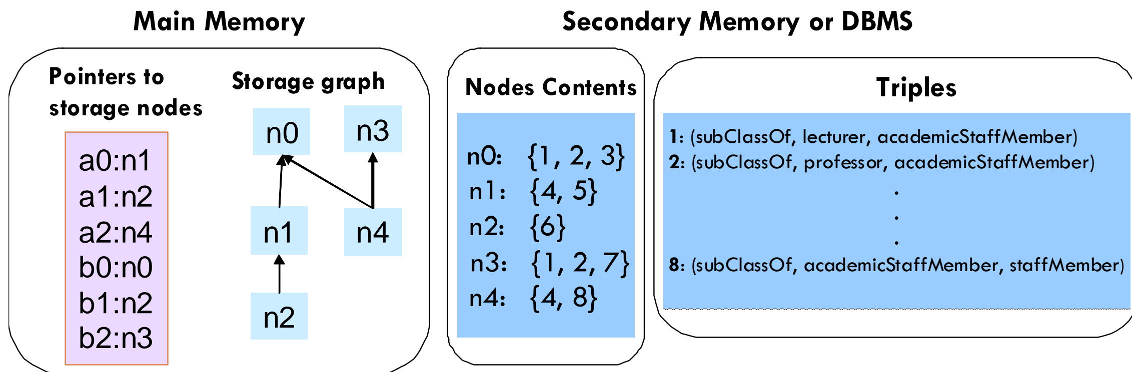


Figure 6.1: Store data using secondary memory

storage space) as they support set-valued attributes (for instance PostgreSQL supports array-valued attributes). In that case, *stored* could be stored in a table with schema `Stored(vid,SetOf(tid))`, and this representation is less space consuming. To retrieve the contents of a version i, again $Br^t(i)$ is needed to be computed using Γ and then a disjunctive query will be send to the DBMS (with all ids in $Br^t(i)$) and the union of the sets obtained will be taken.

Another approach that could be adopted is to keep the all needed information in an inverted file. In order to retrieve the contents of a version i, the same procedure needs to be followed as before. This approach is described briefly in the next Section.

6.2.1 Inverted File Structure

The inverted file is the most popular indexing technique. It is mainly used in Web indexing[1], because it offers fast access to documents based on their content. In general, this scheme allows users to retrieve documents using words occurring in the documents, sequences of adjacent words, and statistical ranking techniques. Also, it usually uses compression methods that ensure that the storage requirements are small and that dynamic update is straightforward. The only assumption that need to be made is that sufficient main memory is available to support an in-memory vocabulary.

For our indexing scheme we assume that *nodes* correspond to *words*, and *triples* to *documents*. For our running example, Figure 6.2 depicts the structure of the inverted file.

[1] http://en.wikipedia.org/wiki/Web_indexing

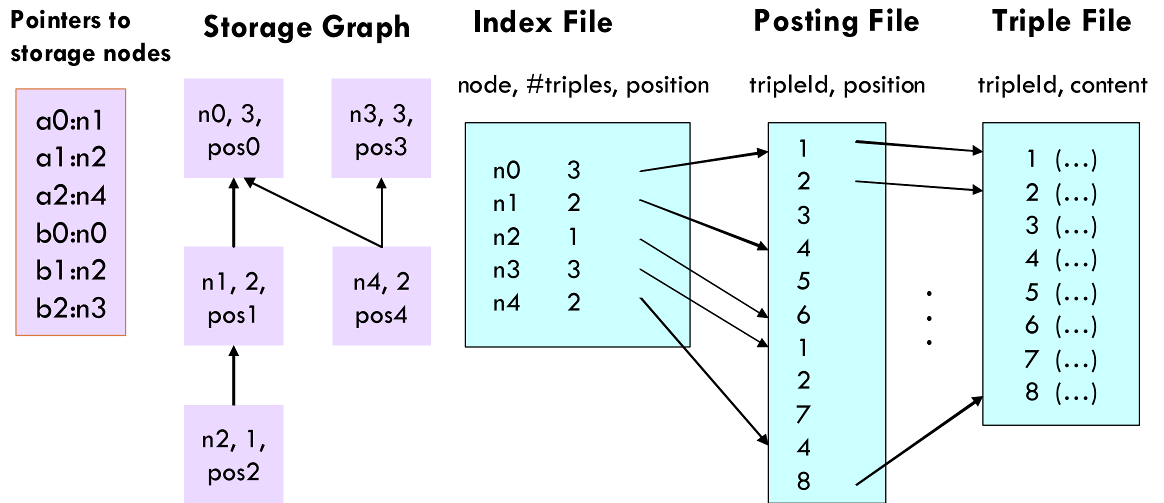


Figure 6.2: Pointers and storage graph kept in main memory (left) and inverted file structure kept in secondary memory (right)

We only need to store in main memory Γ and pointers indicating which node corresponds to each version. Γ consists of the nodes with their relationships and some extra information for each node, that is the *size* of the node (number of triples) and the *position* of the appropriate record to the posting file.

As far as the files kept in secondary memory, the inverted file structure requires three files:

- `Index File`: In essence, it keeps all the information stored in Γ, our *vocabulary*.
- `Posting File`: It contains records of the type $\langle tripleId - position \rangle$, where tripleId is a triples' identifier and position is the relevant position to the Triple File. For each node, all the records in the Posting File are stored consecutive and they are commonly known as *Inverted List* (aka *Posting List*).
- `Triple File`: It contains records of the type $\langle tripleId - content \rangle$, where tripleId is a triple's identifier and content is the triple itself. These records are considered to be sorted in ascending order (w.r.t. tripleIds numbers).

In order to retrieve the contents of a version i, we need to compute $Br^t(i)$ using Γ and then a disjunctive query containing all ids in $Br^t(i)$ needs to be answered. In particular, we will firstly retrieve all the tripleIds from the Posting File, and thereafter we will retrieve the real triples from the Triples File.

Bibliography

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. "Efficient Management of Transitive Relationships in Large Data and Knowledge Bases". *SIGMOD Records*, 18(2):253–262, 1989.

[2] Grigoris Antoniou and Frank vanHarmelen. *A Semantic Web Primer.* MIT Press, Cambridge, MA, USA, 2004.

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

[4] Tim Berners-Lee, Jim Hendler, and Ora Lassila. The Semantic Web: A new Form of Web Content that is Meaningful to Computers will unleash a Revolution of new Posibilities. *Scientific American*, 5(1), 2001.

[5] Roi Blanco and A'lvaro Barreiro. Document identifier reassignment through dimensionality reduction. In *In ECIR*, pages 375–387, 2005.

[6] D. Blandford and G. Blelloch. Index compression through document reordering. In *DCC'02: Procs of the Data Compression Conference*, page 342, Washington, DC, USA, 2002.

[7] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International Semantic Web Conference*, pages 54–68, 2002.

[8] P. Buneman, S. Khanna, K. Tajima, and W.C. Tan. Archiving Scientific Data. *ACM Transactions on Database Systems*, 29(1):2–42, 2004.

[9] S. Chien, V. J. Tsotras, and C. Zaniolo. "Version Management of XML Documents". In *Selected papers from the Third Intern. Workshop WebDB 2000 on The World Wide Web and Databases*, pages 184–200, London, UK, 2001. Springer-Verlag.

[10] John G. Cleary, Ian, and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32:396–402, 1984.

[11] P. Dadam, V. Y. Lum, and H. D. Werner. "Integration of Time Versions into a Relational Database System". In *VLDB '84: Procs of the 10th Intern. Conf. on Very Large Data Bases*, pages 509–522, San Francisco, CA, USA, 1984.

[12] P. Elias. Universal codeword sets and the representation of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.

[13] J. D. Fernández, M. A. Martínez-Prieto, and C. Gutierrez. Compact representation of large rdf data sets for publishing and exchange. In *Procs of the 9th Intern. Semantic Web Conf., ISWC'10*, 2010.

[14] James Geller, Yehoshua Perl, Michael Halper, and Ronald Cornet. Special issue on auditing of terminologies. *Journal of Biomedical Informatics*, 42(3):407–411, 2009.

[15] S.W. Golomb. Run Length Encodings. *IEEE Transactions on Information Theory*, IT-12:399–401, 1966.

[16] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5:199–220, 1993.

[17] C. Gutierrez, C. Hurtado, and A. Mendelzon. "Foundations of Semantic Web Databases". In *Procs of the 23th ACM Symp. on Principles of Database Systems (PODS)*, 2004.

[18] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk rdf storage. In *PSSS*, 2003.

[19] Andreas Harth and Stefan Decker. Yet another rdf store: Perfect index structures for storing semantic web data with contexts. In *DERI Technical Report*, 2004.

[20] Michael Hartung, Toralf Kirsten, and Erhard Rahm. Analyzing the evolution of life science ontologies and mappings. In *DILS*, pages 11–27, 2008.

[21] Ian Horrocks, Frank van Harmelen, Peter Patel-Schneider, Tim Berners-Lee, Dan Brickley, Dan Connoly, Mike Dean, Stefan Decker, Dieter Fensel, Pat Hayes, Jeff Heflin, Jim Hendler, Ora Lassila, Deborah McGuinness, and Lynn Andrea Stein. Daml+oil. March 2001.

[22] Z. Kaoudi, T. Dalamagas, and T. Sellis. "RDFSculpt: Managing RDF Schemas Under Set-Like Semantics". In *Procs of the 2nd European Semantic Web Conf. 2005 (ESWC05)*, Crete, Greece, 2005.

[23] Toralf Kirsten, Michael Hartung, Anika Gross, and Erhard Rahm. Efficient management of biomedical ontology versions. In *OTM Workshops*, pages 574–583, 2009.

[24] A. Kiryakov, K. Simov, and D. Ognyanov. Ontology Middleware: Analysis and Design. Deliverable 38, On-To-Knowledge project. March 2002.

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

[26] Michel Klein and Dieter Fensel. Ontology versioning on the semantic web. In *Stanford University*, pages 75–91, 2001.

[27] Michel C. A. Klein, Dieter Fensel, Atanas Kiryakov, and Damyan Ognyanov. Ontology versioning and change detection on the web. In *EKAW*, pages 197–212, 2002.

[28] Alistair Moffat and Lang Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.

[29] Thomas Neumann and Gerhard Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1):647–659, 2008.

[30] Thomas Neumann and Gerhard Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases. *PVLDB*, 3(1):256–263, 2010.

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

[32] David Salomon. *A Concise Introduction to Data Compression*. Undergraduate topics in computer science. 2008.

[33] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR*, pages 222–229, 2002.

[34] Wann-Yun Shieh, Tien-Fu Chen, Jean Jyh-Jiun Shann, and Chung-Ping Chung. Inverted file compression through document identifier reassignment. *Inf. Process. Manage.*, 39(1):117–131, 2003.

[35] F. Silvestri. Sorting out the document identifier assignment problem. In *Procs of ECIR'07*, pages 101–112, Berlin, Heidelberg, 2007.

[36] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *In Proc. of the 27th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 305–312, 2004.

[37] M. Terrovitis, S. Passas, P. Vassiliadis, and T. Sellis. A Combination of Trie-trees and Inverted Files for the Indexing of Set-valued Attributes. *Procs of CIKM'06*, pages 728–737, 2006.

[38] Yannis Theoharis, Vassilis Christophides, and Gregory Karvounarakis. Benchmarking database representations of rdf/s stores. In *International Semantic Web Conference*, pages 685–701, 2005.

[39] W. F. Tichy. RCS-•a system for version control. *Software Practice & Experience*, 15(7):637–654, July 1985.

[40] Y. Tzitzikas, V. Christophides, G. Flouris, D. Kotzinos, Hannu Markkanen, Dimitris Plexousakis, and N. Spyratos. "Emergent Knowledge Artifacts for Supporting Trialogical E-Learning". *Intern. Journal of Web-based Learning and Teaching Technologies (IJWLTT)*, 2(3):16–38.

[41] Y. Tzitzikas and D. Kotzinos. “(Semantic Web) Evolution through Change Logs: Problems and Solutions”. In *Procs of the Artificial Intelligence and Applications, AIA'2007*, Innsbruck, Austria, February 2007.

[42] Yannis Tzitzikas, Yannis Theoharis, and Dimitris Andreou. On storage policies for semantic web repositories that support versioning. In *ESWC*, pages 705–719, 2008.

[43] M. Volkel, W. Winkler, Y. Sure, S. R. Kruk, and M. Synak. ”SemVersion: A Versioning System for RDF and Ontologies”. In *Procs. of the 2nd European Semantic Web Conf., ESWC'05.*, Heraklion, Crete, May 29 - June 1 2005.

[44] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient rdf storage and retrieval in jena2. In *SWDB*, pages 131–150, 2003.

[45] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images.* Morgan Kaufmann, San Francisco, CA, 2. edition, 1999.

[46] D. Zeginis, Y. Tzitzikas, and V. Christophides. “On the Foundations of Computing Deltas Between RDF Models”. In *Procs of the 6th Intern. Semantic Web Conf., ISWC/ASWC'07*, pages 637–651, Busan, Korea, November 2007.

[47] George K. Zipf. *Human Behavior and the Principle of Least Effort.* 1949.