A Study of Efficient RDFS Entailment in External Memory

MSc Thesis

W.J. Haffmans

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Databases & Hypermedia Group

MSc Thesis

A Study of Efficient RDFS Entailment in External Memory

Author:
W.J. (Wouter) Haffmans

Daily Supervisor:
dr. G.H.L. (George) Fletcher

Supervisors:
prof. dr. P.M.E. (Paul) de Bra
dr. C. (Christian) Stahl

Eindhoven, June 2011

Abstract

The World Wide Web never stands still. The Semantic Web (SW) is one of the core aspects that plays a big role in this development. The Semantic Web allows information to be tagged, related and most of all understood by computers. The W3C Linking Open Data community project takes this one step further by linking Semantic Web data from various sources. By connecting all these loose elements of data it becomes possible to find out more information from various sources. A standard has been developed by the World Wide Web Consortium (W3C), called Resource Description Framework (RDF). This loose structure allows for flexible ways to relate any sort of data and describe its semantics. Generally RDF data sets can be represented as a directed graph of objects (the nodes) and relations (the edges).

One basic extension of RDF is RDF Schema (RDFS). This layer describes some basic semantics of a set of resources, which includes transitivity for some relations. A common problem in various areas of computer science is computing the (transitive) closure or entailment of a graph. Similarly, it is not trivial to check if a triple exists in the closure of an RDFS graph, if there are complicated rules which infer new data.

This thesis investigates three physical indexing methods for RDF storage, comparing them using a specific entailment algorithm proposed in the literature. As such the indexing schemes were compared to find out which one can best be used to perform efficient updates on RDF graphs. Especially the amount of I/O page faults were taken into consideration. Disk I/O is slow and therefore expensive in terms of time. As RDF graphs are generally too large to fit in main memory, the I/O cost should be minimised to achieve better performance. Empirical experiments of this comparison show that an enhanced version of the state of the art physical representation yields the best results.

Keywords: Semantic Web, RDFS, Entailment, I/O Efficiency

Preface

This master's thesis is the result of my graduation project which concludes my Computer Science and Engineering master's degree at the University of Technology Eindhoven. I started in 2003 with a bachelor's degree in Computer Science, followed by the CSE degree. Now that the thesis is complete, my time of studying in Eindhoven has also come to an end. It has been a very interesting time in which I have learnt a lot which will certainly be put to good use in the future.

The project was performed internally in the Databases & Hypermedia department of the faculty Mathematics and Computer Science. The Semantic Web, which this thesis is mostly about, promises to combine exactly these two areas of research. Hypermedia was always about linking documents and data together, just like the Semantic Web does now, and database technologies are important to keep everything running smoothly. Personally I believe there is a bright future for these technologies, making information more accessible than ever before.

I would like to thank my graduation supervisor, George Fletcher, for providing regular feedback and sharing his expertise throughout the project. Furthermore thanks go out to Paul de Bra and Christian Stahl for their input in the project.

I also would not have been able to complete these studies without the support of my parents, brother and sisters – thanks for your continuing support. Last, but most definitely not least, thanks to my girlfriend Rosana and our son Seth, who make my home that very special place I love to go to at the end of every day. I could not have done it without you.

Wouter Haffmans
June 2011

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The World Wide Web never stands still. Huge investments are put in research and development of web services. Especially popular in recent years are social networks. These services allow members to find friends, friends of friends, and discover similar interests of others. Search engines also continue to improve themselves by finding out more about what the data they discover actually means. If an article on the web is shared on a social network by a friend, the search results will let you know if you look for a related term. These kind of relations are continuously discovered and used in various ways. Not only to serve the user in such a direct way, but also to gather information for marketing purposes.

The *Semantic Web*[1] (*SW*) is one of the core aspects that plays a big role in this development. The Semantic Web allows information to be tagged, related and most of all *understood* by computers. The *Linking Open Data* (*LOD*) community project by the *World Wide Web Consortium* (*W3C*) takes this one step further by linking Semantic Web data from various sources [10]. This data can come from any part of the web, including popular sites such as *Wikipedia*, *MusicBrainz* and *Last.FM*. Some governments also participate in this project by providing open data sets, such as the Dutch *rechtspraak.nl*, *US Census* information and UK post codes, research or educational data sets [5].

Consider the (hypothetical) example of a review for a movie; using the Semantic Web and linked data, the review can be linked to the movie's entry on *IMDb*. The author's information is linked to a *FOAF*-record (Friend of a Friend), but also reveals that the author is also active on *Twitter* and *Facebook*. Through *Facebook* we find out that the director of the movie is in fact a close friend...

By connecting all these loose elements of data it becomes possible to find out more information from various sources. A standard has been developed by the W3C, called *Resource Description Framework*[2] (*RDF*). This loose structure allows for flexible ways to relate any sort of data and describe its semantics.

RDF consists of *triples* which desribe relations between two data items (*resources* or *atoms*), using a third resource. Resources can be placed in any position of the triple; by connecting them an RDF *graph* (or Semantic Web Graph) is created.

One basic extension of RDF is *RDF Schema* (*RDFS*). This layer describes some basic semantics of a set of resources – the RDFS *vocabulary*. This description of behavior includes *transitivity* for some relations: if A and B are related with a transitive relation R, and resources B and C are related in the same way, then A and C are also related using R.

A common problem in various areas of computer science is computing the (transitive) *closure* or *entailment* of a graph. This is the base graph, expanded with all implicit relations that can be stated due to the use of transitive relations. While the naive ways to compute it do work, doing this efficiently is a different story. Similarly, it is not trivial to check if a triple exists in the closure of an RDFS graph, if there are complicated rules which infer new data.

[1] `http://www.w3.org/2001/sw/`
[2] `http://www.w3.org/RDF/`

1.1 Research Scope and Goal

The research of this thesis focuses on Semantic Web storage. It especially focuses on storing data in such a way, that it is possible to efficiently and quickly retrieve inferred information from the database, using a specific set of rules. In other words, research is focused on checking if specific information is in the closure or entailment of a Semantic Web graph.

Inferring data has many implications. As argued by Chirkova and Fletcher [2] it is required to check if a statement exists in the entailment of the graph after removal of a triple from the graph. Any statement which makes it possible to still derive the just-removed triple may trigger problems, as it may not be possible to deterministically remove a statement from the graph.

The basis in solving this is laid out by Muñoz et al. [13]. They described an algorithm which can verify if an RDF graph H is entailed by another RDF graph G. More specifically, the algorithm is able to check if a specific *RDF triple* exists in the closure of graph G.

Unfortunately the algorithm merely exists in theory. There is no reference implementation to work from. Additionally its efficiency is proven only based on algorithmic complexity. Practical limitations such as relatively slow disks are not taken into consideration.

The end goal of our research is to design, implement and test a framework to efficiently check if a triple exists in the closure of a graph, using this algorithm as a starting point. Because Semantic Web data is often much too large to fit in main memory, several optimizations will have to be made to make the algorithm more efficient for real-world usage.

There are several major tasks involved in the complete research.

1. Implement the algorithm and verify its correctness.
2. Adjust the algorithm for better I/O performance in general.
3. Create test data which models real-world data.
4. Develop and compare various physical representations using the algorithm on the test data.

The research goal can therefore be described as follows:

Research Goal: *Compare various physical representations of RDFS graphs for (I/O) efficiency, using models of real-world data and an algorithm to check if such an RDFS graph entails a triple.*

1.2 Research Method

The project as described in the previous section can be split into smaller pieces of work. The various points to tackle are as follows:

1. Study the proposed algorithm.
2. Modify algorithm for general I/O efficiency.
3. Create an implementation of the algorithm.
4. Verify the correctness of the algorithm using this implementation.
5. Correct the implementation if necessary.
6. Develop different physical representations of RDFS data.
7. Generate test data with a structure like real-world data.
8. Extract queries from this test data.

9. Compare the efficiency of entailment checking over the physical representations, using the implementation, test data and their queries.

The end result is an application which is able to manage RDFS data and contains an implementation of the mentioned algorithm, which can check for arbitrary triples if they exist in the entailment of the loaded graph. It is also able to report statistics of an execution of the algorithm, especially the amount of pages that need to be read to answer a query.

Additionally the results of the comparisons using this application will be produced. This results in a choice for the most practical of the tested physical representations, based on the executed experiments.

1.3 Contributions

This study contributes a few novel things. First, an implementation of the proposed entailment algorithm is made, with some extensions and corrections. Second, a new method of storing and indexing RDFS data is introduced, which builds on the current state of the art methods. This new method yields significantly better performance with the aforementioned algorithm, at a minimal cost in other areas.

Finally this study provides an in-depth emperical study comparing three storage methods based on the disk I/O performance, measured by the amount of cache misses of the database.

1.4 Thesis Outline

This thesis will discuss the various indexing methods for RDF storage, comparing them using the entailment algorithm. Especially the amount of cache misses, which potentially cause disk I/Os, will be taken into consideration. Disk I/O is slow and therefore expensive in terms of time. As SW graphs are generally too large to fit in main memory, the I/O cost should be minimized to achieve better performance.

The contents of this report is as follows.

Chapter 2 gives some background information regarding RDF, RDFS, the algorithm and used (internal) data structures.

Chapter 3 discusses the correctness of the tested algorithm, including some modifications made beforehand to achieve better performance.

Chapter 4 describes the experimental setup, including the tested physical representations, the data sets and queries that were used.

Chapter 5 has information regarding the framework that was built to run the experiments.

Chapter 6 contains the experimental results of the research, as well as an analysis of those results.

Chapter 7 provides conclusions based on the results, and discusses future work that can be performed to continue research in the area of RDF entailment.

1.5 Related Work

Many others have already studied the performance of different storage methods of RDF data in various different ways, for example [9, 18, 19, 21]. A lot of research has gone into indexing RDF data for fast query processing. In other words, it is focused mostly on specific queries to extract data from an RDF graph. These indexes are benchmarked, mostly using “wall time” to measure performance. Especially when the benchmarks use small data sets, the data may live entirely

in the underlying file system cache. Therefore such experiments can give unreliable results, as the measurements do not include actual disk I/O costs. Also, the queries are often limited in complexity or join depth.

Because large, real-world data sets generally do not fit in main memory, the results are not guaranteed to be very scalable solutions. The performance of for example compared indexing techniques may therefore be unrealistic when it comes to real-world usage.

Others have used various methods to compute the entailment of RDFS graphs. One way is to use a MapReduce technique. The problem is divided and solved in a distributed manner [15, 25, 29]. On the other hand, Datalog engines are used to work with the entailment of an RDFS graph in a manner related to logic programming [11].

This thesis is different in two ways. Its focus is based on an actual algorithm which is required to determine if a triple is entailed by a graph. It does not just run static queries, but tries to dynamically check if a triple is in the closure of a graph using specific rules about inferred data. The used algorithm uses the inference of RDFS which means the length of some sub-graphs can heavily influence the running time of the algorithm. As such, the queries are not the sole factor which define the running time of the algorithm, but the used data set is a major factor in this as well.

This thesis also counts the number of I/O cache misses as measure. As the cache size of the application is limited, this will measure the performance of the algorithm when the RDF graphs grow beyond the limits of RAM memory. This is supported by the fact that the test data contains graphs with millions of triples in them. The used data sets are modeled after real-world data. They are, however, much larger and structured such that the algorithm will in general perform worse using this test data, than when using it on real-world data.

An additional contribution introduced in this thesis is an extension of the Triple-T physical representation [9]. This extension stores a small part of the entailment of an RDFS graph, by inferring this data upon updates of the graph. This adds the benefit that looking up information is quicker, while not compromising much on the consumed storage space of the graph, or complexities of updates.

Chapter 2

Preliminaries

This chapter will discuss some preliminary concepts used throughout this thesis. These concepts are the foundation upon which this research continues. Discussing these topics in-depth is beyond the scope of this thesis; only the required basic information will be provided.

2.1 RDF

The Resource Description Framework (RDF) was devised by the W3C to provide a way of storing abstract data. As the name suggests, it is targeted at describing resources. More specifically it is used to model information contained in web resources. The first specification was published in 1999. The current set of specifications was published in 2004. Work has started into revising this version. This was kicked off by a W3C Workshop *RDF Next Steps* [1].

The data structure of RDF is simple. Every data set contains statements, or triples, only. Each triple is of the form $(subject, predicate, object)$. The resource ($subject$) is described using a relation ($predicate$) to another resource ($object$). *Resource* is a very liberal term, as it can in essence be anything. Each word can be used in any position of a triple, so relations and resources can be easily interchanged. Throughout this thesis the term *atom* will often be used as synonym of an RDF resource.

A data set, such as the one in figure 2.1, is an RDF graph. Each statement can be viewed as two nodes (subject and object) connected by an edge (predicate).

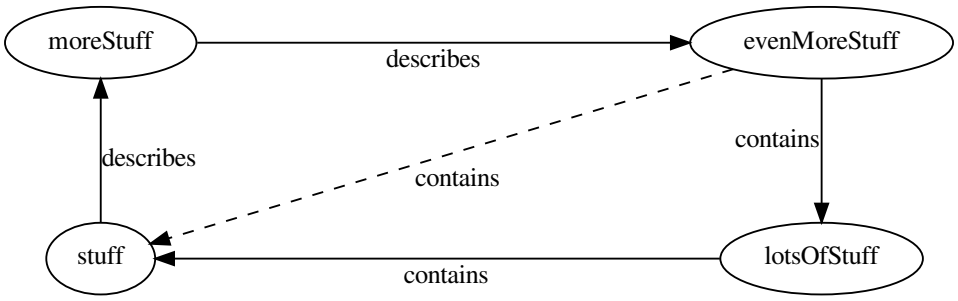


Figure 2.1: RDF Graph Example

Consider the graph in figure 2.1. Here the atoms $stuff$, $moreStuff$, $evenMoreStuff$, $lotsOfStuff$ are used as both subjects and objects, depending on the triple. For example, in $(stuff, describes, moreStuff)$ the $stuff$ atom is the subject. Atom $describes$ is the predicate, whereas $moreStuff$ is the object.

[1] W3C Workshop – RDF Next Steps `http://www.w3.org/2009/12/rdf-ws/`

RDF graphs are mostly used in the Semantic Web and to link data in the Linking Open Data project. There is no fixed way of serializing or storing RDF data, although there are some commonly used serialization formats. These include *RDF XML*[2], *Notation 3*[3] (*N3*) and *RDFa*[4].

Because of the loose structure of RDF and the sheer size of the graphs, often containing millions of statements, active research still involves finding ways to structure the data such that it can be reasoned over efficiently. Each year there is a Semantic Web challenge that challenges organisations to develop scalable RDF tools. One track of this challenge is the Billion Triples track. The goal of this track is "*to demonstrate the scalability of applications as well as the capability to deal with the specifics of data that has been crawled from the public web.*" [20]

RDF is used in many large projects. These include Linked Open Data, which connects data from various popular source, including some *Open Government Data*. The target of the latter is to make government data better searchable. Furthermore RDF can also be used to collect information about files or documents on a personal computer, such as in the *NEPOMUK*[5] project.

All these sets contain a lot of (meta)data which is stored in RDF graphs. Each document, person or entity can produce multiple RDF triples. With the possibility of having to keep metadata of thousands or even millions of documents (for example e-mails or web pages) or people in a graph (for example in social networks), it becomes evident that RDF graphs can easily grow to become huge data sets.

2.2 Ontologies

RDF graphs can roughly be split into two parts. The first part is the ontology. This consists of RDF triples which describes the data in the second part, the instance data. For example, the ontology describes the domains and ranges of predicates and the type of data – the metadata of the other data.

The other part instantiates actual data, making statements about items and defining relations. The instance data sets are generally much larger (up to millions or even billions of triples) than the ontologies. In general this data is already metadata (since it describes items – such as people – but does not contain the items themselves), making the ontology meta-metadata.

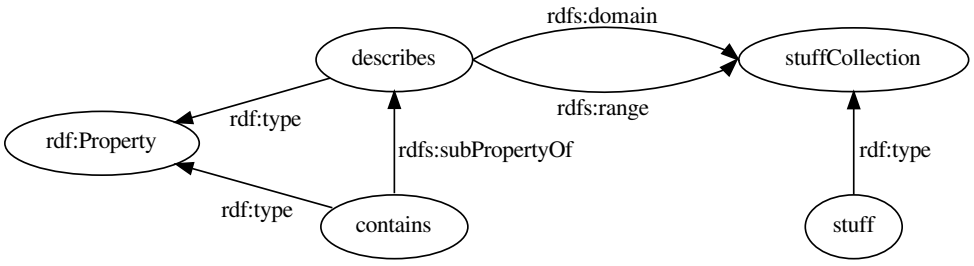


Figure 2.2: RDF Ontology Example

Figure 2.2 shows part of the ontology of the data in figure 2.1. Resources *contains* and *describes* are properties. These connect resources of type *stuffCollection*, as both domain and range. The *stuff* resource, but also *moreStuff*, *evenMoreStuff* and *lotsOfStuff*, are such resources of type *stuffCollection*.

Ontologies and instance data are syntactically equal. Both contain triples, which are of the same form: entities (the subject and object) are related using a predicate. Because the ontologies

[2]RDF XML: `http://www.w3.org/TR/rdf-syntax-grammar/`
[3]N3: `http://www.w3.org/TeamSubmission/n3/`
[4]RDFa: `http://www.w3.org/TR/rdfa-syntax/`
[5]NEPOMUK: `http://nepomuk.semanticdesktop.org`

and instance data use the same base structure – RDF – they are also regularly stored together in one graph. Note however that it is perfectly valid to keep the two stored separately. Also, a graph can simply merge and combine multiple ontologies if necessary.

A lot of current research is about ontology evolution. This involves research focused on changing and updating ontologies – in part automatically – to update the knowledge contained in the ontology. This involves updating the instance graph to reflect the changes in the ontology as well. This can be done either materializing the new data, or by finding implicit data on demand, which will be discussed in this thesis.

2.3 RDFS

RDF Schema (RDFS, [28]) is an extension of RDF. Whereas RDF provides just a basic vocabulary, RDFS adds inferencing rules on top of the vocabulary. For example, RDFS provides semantics for the `rdfs:subPropertyOf` predicate. If a graph states (A, `rdfs:subPropertyOf`, B) and (P, A, Q), then RDFS states that implicitly the graph also contains (P, B, Q).

Due to this inference, adding a triple to the graph may result in another set of implicit triples also being present in the graph. This may be a vast number, and materializing all this data may be a mere waste of space if it is never really used. Additionally, after removing a triple from the graph, a lot of (implicit) data may still exist if no special care is taken to remove it. Also, these updates become more costly if implicit data is materialized.

On the other hand, not materializing the data means the graph has to be scanned for implicit data whenever the data is searched. This means queries are more costly, as checking for a single triple in the graph may require several sub-queries to be performed. Essentially the finding of implicit data is deferred until it is actually required – but updates are as efficient as they can possibly be, because there is no need to do any work apart from inserting or removing the specific triples.

All in all, the choice of whether to materialize implicit data or not is a trade-off between the better performance of updates, or better performance of look-ups. It depends on how the data is used, and how often it is updated, to determine which is the best solution for a specific graph.

As already briefly mentioned before, this research is focused on locating inferred data on demand, without materializing the entailment of a graph. At most a small part of the entailment will be materialized, to see its effect in reduction of I/O costs, while keeping updates reasonably efficient as well.

The base set of rules describing the RDFS semantics can be found in appendix B.

2.4 Minimal RDFS

This study is limited to a subset of RDFS, which Muñoz et al call ρdf [13]. The selected RDFS vocabulary is the most essential part of RDFS, as well as self-contained: it does not rely on the RDFS vocabulary beyond the subset, nor does the rest of the RDFS vocabulary rely on this subset.

For readability, most of the terms will be abbreviated throughout this document. See table 2.1 for the vocabulary of ρdf and the corresponding abbreviations that are used. Additionally two base terms, `rdfs:Class` and `rdf:Property` (abbreviated: `class` and `prop` respectively) will also be used.

Muñoz et al. [13] also introduce Minimal RDFS (mrdfs) as follows:

> "*A minimal RDFS triple (mrdfs-triple for short) is a ground ρdf-triple having no ρdf vocabulary as subject or object. A mrdfs-graph is a set of mrdfs-triples.*"

In this definition, a *ground triple* is a triple without any blank nodes. A blank node is one node in a triple, which has no literal data. They are also called anonymous resources. Blank nodes can only be used as subject or predicate, according to the RDF standard [26].

Property	Abbreviation
rdfs:subClassOf	sc
rdf:type	type
rdfs:subPropertyOf	sp
rdfs:domain	dom
rdfs:range	range

Table 2.1: ρdf vocabulary

By definition, in minimal RDFS the ρdf terms can only appear as predicate. Therefore, a proper minimal RDFS graph will not expand RDFS as there will be no statements about any of the terms in ρdf – the semantics do not change. This research is therefore restricted to valid ground triples. This notion is also supported by the fact that using ρdf or RDFS keywords in the subject or object role, is considered "non-standard use of RDFS-vocabulary" [6].

The inference rules used in mrdfs are also given, and derived from the original RDFS semantics (see appendix B, [27]). For completeness, they can also be found in table 2.2. These rules define under which conditions triples in specific forms can be derived.

Rule	Precondition(s)	Inferred
1a	G, map $\mu : G' \to G$	G'
1b	G, $G' \subseteq G$	G'
2a	$(A, \texttt{sp}, B)$, $(B, \texttt{sp}, C)$	$(A, \texttt{sp}, C)$
2b	$(A, \texttt{sp}, B)$, (X, A, Y)	$(X, \texttt{type}, B)$
3a	$(A, \texttt{sc}, B)$, $(B, \texttt{sc}, C)$	$(A, \texttt{sc}, C)$
3b	$(A, \texttt{sc}, B)$, $(X, \texttt{type}, A)$	$(X, \texttt{type}, B)$
4a	$(A, \texttt{dom}, B)$, (X, A, Y)	$(X, \texttt{type}, B)$
4b	$(A, \texttt{range}, B)$, (X, A, Y)	$(Y, \texttt{type}, B)$
5a	$(A, \texttt{dom}, B)$, $(C, \texttt{sp}, A)$, (X, C, Y)	$(X, \texttt{type}, B)$
5b	$(A, \texttt{range}, B)$, $(C, \texttt{sp}, A)$, (X, C, Y)	$(Y, \texttt{type}, B)$
6a	(X, A, Y)	$(A, \texttt{sp}, A)$
6b	$(A, \texttt{sp}, B)$	$(A, \texttt{sp}, A)$, $(B, \texttt{sp}, B)$
6c	$P \in \rho$df	$(P, \texttt{sp}, P)$
6d	(A, P, X), $P \in \{\texttt{dom}, \texttt{range}\}$	$(A, \texttt{sp}, A)$
7a	$(A, \texttt{sc}, B)$	$(A, \texttt{sc}, A)$, $(B, \texttt{sc}, B)$
7b	(X, P, A), $P \in \{\texttt{dom}, \texttt{range}, \texttt{type}\}$	$(A, \texttt{sc}, A)$

Table 2.2: Deductive rules for the fragment ρdf

In this study, unless otherwise stated, we will use the term *graph* as synonym of mrdfs-graph.

2.5 Entailment Algorithm

The algorithm which is discussed in this thesis was introduced by Muñoz et al. [13]. This algorithm checks, in $O(n \; log \; n)$ complexity (where n is the amount of triples in the entire graph, including ontology) if a triple can be inferred from an RDFS graph G using the mrdfs rules. This is done without actually materializing any inferred data.

The algorithm uses 6 steps to cover all the rules, explicitly excluding reflexivity. Each of these steps is distinct. Depending on the triple that is checked, only one of the steps will be executed. This mostly depends on the predicate of the triple.

The last two steps are very similar and also the most complicated steps of the algorithm. These steps involve the implicit typing rules, which together use the entire ρdf fragment to get to an

answer. This therefore also includes traversing parts of both the `sp` and `sc` hierarchies in order to find the right answer.

The authors of the algorithm claim that the algorithm is "*correct and complete.*" The verbatim algorithm, as originally listed in [13], table 2, is shown in listing 2.1. The algorithm checks the existence of triple (a, p, b) in graph G.

In the algorithm and throughout the remainder of this thesis, G(x) means the sub-graph of triples of G with predicate x. $G_\emptyset$ is the sub-graph of triples with a predicate that is not part of ρdf.

1. **IF** $p \in \{$`dom`, `range`$\}$ **THEN** check if $(a, p, b) \in$ G(`sp`)
2. **IF** $p =$ `sp`, $a \neq b$, **THEN** check if there is a path from a to b in G(`sp`)
3. **IF** $p =$ `sc`, $a \neq b$, **THEN** check if there is a path from a to b in G(`sc`)
4. **IF** $p \notin \rho$df **THEN** check if $(a, p, b) \in G_\emptyset$: if it is not:
   LET G(`sp`)* be the graph G(`sp`) with the following marks:
   For each $(u, v, w) \in G_\emptyset$ then mark v with (u, w).
   IN Check in G(`sp`)* if there is a path from a vertex marked with (a, b) which reaches p.
5. **IF** $p =$ `type` **THEN**
   LET G(`sp`)$'$ be the graph G(`sp`) with the following marks:
   For each triple $(u,$ `dom`$, v) \in G$ mark the vertex u in G(`sp`) with $d(v)$.
   For each triple $(z, e, y) \in G_\emptyset$ mark the node e with $s(z)$.
   LET L_d be the ordered list of elements $d(v)$ such that there is a path from v to b in G(`sc`).
   LET L_s be the ordered list of elements $s(z)$ such that either:
   (1) in G(`sp`)$'$ there is a path from a node marked $s(z)$ to a node marked with an element in L_d, or
   (2) there is $(z,$ `type`$, v) \in G$ for $d(v) \in L_d$.
   IN Check if s(a) is in L_s.
6. Repeat point 5 symmetrically for range instead of `dom` (making the corresponding changes).

Listing 2.1: Original algorithm of checking (minimal RDFS entailment of triple (a, b, c) [13]

2.6 B+-Trees

One of the tested physical storage mechanisms – where physical means the way data is structured on a medium such as a hard disk – uses B+-Trees to index the data [4]. Such a tree is a structure which makes it easy to locate specific data, including range traversal. It is not to be confused with the term Binary Tree, although there are some similarities. A B+-Tree is a variant of a B-Tree. The differences between these two are beyond the scope of this thesis. More detailed background information can be found in [4].

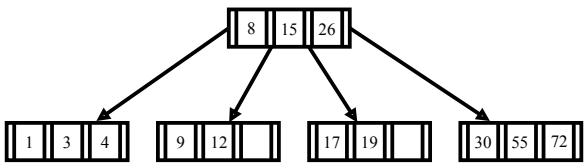


Figure 2.3: (Part of a) B+-Tree, node size 3

The structure of a B+-Tree is as follows. Each level has several nodes. Each node contains multiple keys which can be ordered – such as natural numbers. Each of these has two pointers to a node on a lower level: one to a node with items lower (in the ordering) than the key, the other

one to a node with items greater than the key. Two adjacent keys in the same node share the pointer in the middle, which points to a node with items in between the two keys.

The amount of keys in a node is the same for all, and generally depends on the amount of keys which are expected to fit on a certain number of disk pages. This is efficient, because blocks on disk are loaded per page. Additionally, reading sequential data from disk is faster than doing random look-ups.

A B^+-Tree is also balanced. All the leaf nodes, which point to actual data, are on the same level. During updates, the tree is rebalanced if required, such that each node is at least full to a certain threshold (in B^+-Trees this generally means nodes must be kept at least half full).

Traversing down these B^+-Trees to find an item is as easy as loading the root node, finding out which pointer to follow and loading the next node. As each node can contain many items, that also implies a lot of unnecessary items are also ignored on every step.

B^+-Trees can store a high number of items in each node. Each level also contains a high number of nodes (increasing exponentially). Therefore the cost to look up an item in the tree can be done in as few as 3 or 4 disk I/Os, depending on the used node size. Even with 10 million records and a node size of just 10 records, only 7 I/Os are required to locate an item [4].

The implementations made for this research internally use the Berkeley DB library [6]. This library implements B^+-Trees. As such, in the rest of this thesis, whenever we mention B-Trees – the basic structure of which B^+-Trees are a slightly more advanced variant – we actually mean B^+-Trees.

2.7 Hash Tables

The other major structure that is used to store RDF graphs in external memory involes hash tables. This makes use of a hashing function. Such a function takes a key to insert, and returns a value of a specific length (in bits). All data is stored in a table with buckets, such as in figure 2.4. The buckets contain the keys and their data. The table in fact generalizes an array in which it is possible to directly load specific data.

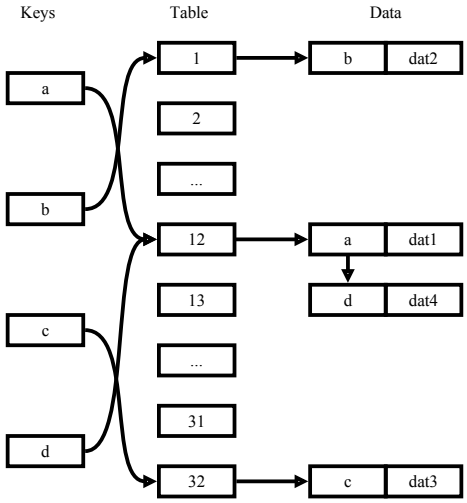


Figure 2.4: Hash Table Example

The outcome of the hash function determines which bucket to place a key in. Such a bucket can be quickly located and loaded, after which the actual data can also be found. If multiple keys have to be placed in the same bucket – a so-called collision – the data is chained. Essentially this

[6] Berkeley DB: `http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html`

is a form of a linked list, where a linear search within a bucket (and potentially overflow buckets, if the data in the bucket grows beyond its size) will make it possible to find the necessary item. The technique to store these items in such a way is called chaining.

The standard methods of implementing hash tables in external memory are extendible and linear hashing. In *extendible hashing* a prefix tree is used to locate items. This is a dynamical structure which grows and shrinks gracefully [8]. In *linear hashing* the hash table is expanded by a power of two every time as required [12].

Unlike B-Trees, the data is not sorted. Range traversal is therefore not possible. However, using the hash function and loading the required bucket is generally faster than using B-Trees. While the worst-case scenario is $O(n)$ for hash tables (which is $O(log\ n)$ for B-Trees) if all items are located in the same bucket, the amortized performance is constant ($O(1)$). This does require a good hash function, which prevents collisions as much as possible.

If the number of data items grows too large to fit in the table, it is possible to grow the table and create extra buckets, and use a more detailed hash function. To prevent many overflow buckets or growing the table too quickly, which both degrade the performance of hash tables, it is important that the hash function distributes keys as evenly as possible over the available buckets.

Chapter 3

Correctness of Entailment Algorithm

In this chapter we will first introduce some changes and extensions made to the entailment algorithm introduced in section 2.5. We will then validate the correctness of the algorithm and discuss necessary corrections.

3.1 Validation Setup

The first step of the research is to make sure the entailment algorithm is indeed correct, based on the ρdf rules. The algorithm was implemented as explained by Muñoz et al., although with slight practical modifications.

3.1.1 Initial Modifications

The used algorithm in general takes the approach of marking nodes in a certain way, following by checking if specific conditions are met using these markings. This is no problem as long as the graph can be loaded into main memory. If it does not, these markings will have to temporarily be materialized to disk to make it even remotely feasible to run the algorithm. It may be more efficient if the query *graph H* has multiple triples in it, rather than our use case which is limited to single triples. The markings only need to be computed once, to check whether the original graph completely entails the target graph.

However, this research is focused on target graphs which consist of a single triple. Also, we take the assumption that graph G is too large to fit in main memory. Therefore implementing the algorithm straight away can be very troublesome.

For these reasons, several changes were made to the base algorithm to make it feasible to execute the algorithm with large data sets. While the functionality and core process of the algorithm is kept the same, details were changed so the implementation is better capable of dealing with large data sets.

3.1.2 Reflexivity

The original authors explicitly omit `sp` and `sc` reflexivity from minimal RDFS. This is because these reflexivity rules (rules 6 and 7 from table 2.2, table 1 of [13]) are orthogonal to the rest of the semantics [13].

However, adding in reflexivity steps to the algorithm is fairly trivial, and does make the algorithm complete for the ρdf fragment. This expansion will also not influence the complexity of the entire algorithm, because the necessary steps can be taken in $O(log\ n)$ time assuming the graph is sorted.

Essentially, to check for reflexive triples, it is necessary to determine whether the subject (and object) is used as property or class respectively. This is done by checking whether specific triples exist.

For $(A, \texttt{sp}, A)$, these are the following triples and their corresponding rule numbers as given in table 2.2. A ? marks a blank: any resource can serve as substitution for this blank, and satisfy the query. These triples are directly and trivially derived from the rules.

- $(?, A, ?)$ (6a)
- $(A, \texttt{sp}, ?)$ (6b)
- $(?, \texttt{sp}, A)$ (6b)
- $(A, \texttt{dom}, ?)$ (6d)
- $(A, \texttt{range}, ?)$ (6d)

Additionally, for rule (6c) it suffices to check whether the subject A in the query is part of the ρdf fragment.

For $(A, \texttt{sc}, A)$ (step 3b) the corresponding triples to check for are:

- $(A, \texttt{sc}, ?)$ (7a)
- $(?, \texttt{sc}, A)$ (7a)
- $(?, \texttt{dom}, A)$ (7b)
- $(?, \texttt{range}, A)$ (7b)
- $(?, \texttt{type}, A)$ (7b)

Expanding the algorithm is simple, by adding two extra steps (named 2b and 3b respectively) which perform these checks. Listing 3.1 shows these steps of the algorithm as they would fit in the original algorithm.

2b. If $p = \texttt{sp}$ and $a = b$, **THEN**
Check if $a \in \rho$df; if it is not:
LET G^* be the graph G with the following marks:
For each $(u, v, w) \in G$, mark green v
For each $(u, \texttt{sp}, w) \in G$, mark green u and w
For each $(u, \texttt{dom}, w)$ or $(u, \texttt{range}, w) \in G$ mark green u
IN Check if vertex a is marked green.

3b. **IF** $p = \texttt{sc}$ and $a = b$, **THEN**
LET G^* be the graph G with the following marks:
For each $(u, \texttt{sc}, w) \in G$, mark green u and w
For each $(u, \texttt{dom}, w)$, $(u, \texttt{range}, w)$ or $(u, \texttt{type}, w) \in G$ mark green w
IN Check if vertex a is marked green.

Listing 3.1: Algorithm steps for reflexivity

Proposition 1. *Checking for reflexivity does not influence the complete algorithm's $O(n \log n)$ complexity.*

Proof. A simple way to mark each node is by traversing all nodes in the graph G linearly. This trivially takes $|G| = n$ steps. The next step can linearly search the list of marked nodes to find out if a is marked, resulting in a running time of these steps of at most $O(n)$, which clearly is better than $O(n \log n)$. □

The method in the proof is very naive; using smarter techniques it may very well be possible to perform these steps more efficiently. That is beyond the focus of this thesis; we merely want to show that the overall performance of the algorithm is not impacted by these changes.

The actual implementation that was made is slightly different from the steps presented previously. For efficiency reasons, rather than first marking green the appropriate vertices, it immediately checks for matching triples in G by executing a look-up on the storage back-end. This is more efficient, because the first positive hit can terminate the algorithm and return the answer. This means that the actual implementation, in pseudo-code, is more like in listing 3.2.

```
If p = sp and a = b, THEN
  IF a ∈ ρdf THEN RETURN true
  ELSE IF (?, a, ?) ∈ G THEN RETURN true
  ELSE IF (a, sp, ?) ∈ G THEN RETURN true
  ELSE IF (?, sp, a) ∈ G THEN RETURN true
  ELSE IF (a, dom, ?) ∈ G THEN RETURN true
  ELSE IF (a, range, ?) ∈ G THEN RETURN true
  ELSE RETURN false

If p = sc and a = b, THEN
  IF (a, sc, ?) ∈ G THEN RETURN true
  ELSE IF (?, sc, a) ∈ G THEN RETURN true
  ELSE IF (?, dom, a) ∈ G THEN RETURN true
  ELSE IF (?, range, a) ∈ G THEN RETURN true
  ELSE IF (?, type, a) ∈ G THEN RETURN true
  ELSE RETURN false
```

Listing 3.2: Pseudo-code for reflexivity

Proposition 2. *The implementation of steps 2b and 3b do not influence the complete algorithm's $O(n\ log\ n)$ complexity.*

Proof. In a very naive approach, an unsorted graph can be checked one triple at a time ($O(n)$ steps) to find a match for the required triples. The proposition is therefore trivially true. □

Of course a linear search is not the most optimal approach; depending on how the graph is stored much better performance can easily be achieved. Although various (physical) representations of the data will be discussed further in this thesis, the goal of this part is merely to show that the original complexity of the algorithm is not influenced by any changes introduced here.

Any other original refinements introduced by Muñoz are kept in place, especially the restriction that ρdf vocabulary is not used as subject or object in triples, to avoid redefinition of RDFS. This is considered "non-standard use of RDFS-vocabulary" [6]. As such, support for reflexivity is the only addition to the base algorithm. Blank nodes are also not supported.

3.1.3 Modifications of Step 4

The first actual modification is in step 4 of the algorithm. This step first marks many statements in the graph, before checking if there is a path from a specifically marked vertex to the required property p. The proposed change is to start at p and, using G(sp) find the subproperties of p. For each of these, as soon as they are found, the algorithm checks if (a, q, b) exists in the graph (with q being the subproperty).

The result is that rather than marking items in G(sp) based on statements in $G_\emptyset$ – which can be a very large sub-graph – the algorithm can be terminated as soon as a valid match is found. Rather than going over the complete sub-graph $G_\emptyset$, only the required part of G(sp) is traversed,

which is likely much smaller. This part is also traversed in the original algorithm, to check if there is a valid path.

Additionally to verify that the traversal terminates, it may be necessary to keep track of which properties were already processed earlier, such that they are not processed twice. This is to avoid problems in case G contains cycles.

In short, the pseudo-code for the new step 4 is as in listing 3.3.

```
Processed = ∅
Enqueue p into queue Q
WHILE Q is not empty DO
  Dequeue p' from Q
  IF p' ∈ Processed THEN Continue with next item
  Add p' to Processed
  IF (a, p', b) ∈ G THEN RETURN true
  ELSE FOR all (p'', sp, p') ∈ G
    DO Enqueue p'' into queue Q
RETURN false
```

Listing 3.3: Updated step 4

Proposition 3. *The worst case running time of $O(n\ log\ n)$ (with $n = |G|$) is maintained by the change made to step 4 of the algorithm.*

Proof. In the worst case scenario the algorithm traverses over all triples in graph G ($n = |G|$ steps). For each property, checking if the property was processed before can be done by binary search, in $O(log\ n)$ comparisons (assuming *Processed* is sorted).

Similarly, if G is sorted, checking for the existence of triple (a, p', b) in G can also be done in $O(log\ n)$ steps. Sorting the graph can be done in $O(n\ log\ n)$ if necessary, before running the algorithm. □

Recall that originally for this step, the theoretical limit was also $O(n\ log\ n)$ due to the construction of $G(\texttt{sp})^*$ in the original algorithm. All in all the changes are not expected to have a negative impact.

3.1.4 Modifications of Steps 5 and 6

For efficiency reasons, steps 5 and 6 were merged into one step. This allows for constructing L_d (step 5) and L_r (the counterpart of L_d in step 6) in one run. Rather than finding all domains and ranges for all classes, and marking them appropriately as suggested in the original algorithm, these lists are built by looking up the superclasses of b. For each superclass v, all matching u atoms in $(u, \texttt{dom}, v)$ are put in L_d. Similarly, the w in $(w, \texttt{range}, x)$ is placed in L_r. Both lists L_d and L_r are now constructed as specified in the original algorithm.

Constructing L_s explicitly is also omitted. This is originally constructed using triples in $G_\emptyset$ and checking if the predicate is a subproperty of an item in L_d or L_r, and then marking the subject and object appropriately.

Instead of this method, each property in L_d and L_r are used as base. From each such property p', a look-up is performed for $(a, p', ?)$ if p' is in L_d, or $(?, p', a)$ if p' is in L_r. Both look-ups are performed if a property occurs in both lists. From the base, the superproperties are also traversed using the same domain or range role, using similar look-ups.

To prevent traversing parts of the subproperty tree twice – once for L_d and once for L_r – two sets are kept which indicate which properties have been processed in which role (domain or range). This may not always help, since a property and all its subproperties may be processed in the domain role, before the initial property is discovered as being valid in range role as well, or vice versa. However, in practice this is not very likely to happen.

Whenever a look-up yields a positive result, a is indeed of the requested type b. The algorithm will then terminate early. In pseudo-code the entire procedure is as in listing 3.4

Enqueue b into queue Q
WHILE Q is not empty **DO**
  dequeue b' from Q
  FOR all triples $(d, \texttt{dom}, b') \in G$ **DO** put d in L_d
  FOR all triples $(r, \texttt{range}, b') \in G$ **DO** put r in L_r
  FOR all $(b'', \texttt{sc}, b') \in G$ **DO** enqueue b'' into queue Q

FOR all properties $p \in L_d \cup L_r$
  DO enqueue p' in R

$inLd = L_d$, $inLr = L_r$
$ProcessedLd = ProcessedLr = \emptyset$

WHILE R is not empty **DO**
  dequeue p' from R

  Verify that item p' has not yet been processed already:
  – $doLd = (p' \in inLd \wedge p' \notin ProcessedLd)$
  – $doLr = (p' \in inLr \wedge p' \notin ProcessedLr)$

  IF $\neg(doLd \vee doLr)$ **THEN** continue with next item

  IF $doLd$ **THEN**
  – **IF** $(a, p', ?) \in G$ **THEN RETURN** true
  – add p' to $ProcessedLd$
  IF $doLr$ **THEN**
  – **IF** $(?, p', a) \in G$ **THEN RETURN** true
  – add p' to $ProcessedLr$

  FOR all $(p'', \texttt{sp}, p') \in G$ **DO**
    IF $doLd$ **THEN** add p'' to $inLd$
    IF $doLr$ **THEN** add p'' to $inLr$
    enqueue p'' to R
RETURN false

Listing 3.4: Updated steps 5 and 6

Proposition 4. *The modifications to steps 5 and 6 of the algorithm do not influence the algorithm's worst case running time of $O(n \log n)$.*

Proof. Clearly the construction of L_d and L_r can be done in at most n steps. The second part may go over n triples (at most twice), each time checking if it was previously processed. Those checks can be performed in $O(log\ n)$ using binary search. Similarly, checking for all triples in G can be done in $O(log\ n)$. All in all the worst case theoretical complexity is $O(n\ log\ n)$. □

These last two changes do not affect the complexity of the algorithm, but do have several practical benefits which are expected to perform better than the original. This is mostly because only the required parts of the `sp` and `sc` sub-graphs are traversed, and the algorithm can terminate earlier if a match is found.

3.2 Validation

To verify the correctness of the algorithm initially a fairly simple back-end, using the relational database MySQL, was implemented. See section 5.1 for more details regarding the global architecture. This back-end focused purely on functionality; its performance does not matter. Because of available abstraction layers it was easy to correctly create this implementation.

Using this storage back-end, small testing graphs were created after which the algorithm was executed with different queries. Due to the small size of these graphs, it is possible to verify the correctness of the answers using manual inspection.

One erroor in the algorithm was discovered. While it could have been an implementation mistake, further investigation learned that this was not the case. The theoretical version of the algorithm is (subtly) incomplete.

The rule in question is rule 3b. Given a graph with (only) triples $(a, \texttt{type}, b)$ and $(b, \texttt{sc}, c)$, the triple $(a, \texttt{type}, c)$ was not found as being in the entailment. Rule 3b directly states this should be found, though.

It is true that rule 3b is used in the proof and the algorithm itself, but only indirectly if the resource a in the query is also actually used using a property with a domain (or range), which has a subclass relation with the target b. In other words: rule 3b is only applied in combination with rule 4.

All in all an extra addition needs to be made to step 5 of the algortihm, which answers these `type`-queries. First, a list is built using all $(x, \texttt{type}, w)$ triples in the graph. Each node x is marked $t(w)$ in G(`sc`).

The (ordered) list L_t can then be constructed by taking all elements $t(w)$ with a path from w to target b in G(`sc`). In other words: $(w, \texttt{sc}, b)$ holds in the graph. Finally, list L_s is expanded with those items $s(z)$ such that node z is marked with an element $t(w) \in L_t$. In short, this implies there is a $(z, \texttt{type}, w)$ triple, for which $(w, \texttt{sc}, b)$ holds. The (existing) check for the existance of an element $s(a)$ in this list L_s then implies the existence of the query $(a, \texttt{type}, b)$.

There is no need to update step 6, since the domain and range do not play a role in these additions. Alternatively a complete separate step can be added to the algorithm, which performs the required checks for rule 3b.

Step 5 with these additions will then become as in listing 3.5. The newly added lines are marked with a +.

```
  IF p = type THEN
    LET G(sp)' be the graph G(sp) with the following marks:
      For each triple (u, dom, v) ∈ G mark the vertex u in G(sp) with d(v).
+     For each triple (x, type, w) ∈ G mark the vertex x in G(sc) with t(w).
      For each triple (z, e, y) ∈ G_∅ mark the node e with s(z).
    LET L_d be the ordered list of elements d(v) such that there is a path from v to b in G(sc).
+   LET L_t be the ordered list of elements t(w) such that there is a path from w to b in G(sc).
    LET L_s be the ordered list of elements s(z) such that either:
    (1) in G(sp)' there is a path from a node marked s(z) to a node marked with an element in
        L_d, or
    (2) there is (z, type, v) ∈ G for d(v) ∈ L_d.
+   (3) there is (z, type, w) ∈ G for t(w) ∈ L_t.
    IN Check if s(a) is in L_s.
```

Listing 3.5: Corrected step 5

The correctness of the extra steps follows from inspection of the rules. In this particular case, rule 3b is used once, followed by zero or more applications of rule 3a, as explained. The first extra line in the algorithm marks $t(x)$ if there is a triple $(a, \texttt{type}, x) \in G$ (for any x). The second line filters out those $t(x)$ elements for which x is not a subclass of b. Finally, $s(a)$ is added to L_s if a is marked with any $t(x) \in L_t$.

It is also clear that the complexity is not affected, as the additions are of a very similar form as already existing lines. Obviously the same techniques can be used to construct and inspect L_t.

The actual implementation is somewhat different, as discussed previously. Together with the correction, the pseudo-code for step 5 is as in listing 3.6. This updated version returns true as soon as a valid $(a, \texttt{type}, b')$ triple is found, for which $(b', \texttt{sc}, b)$ holds. Checking for this triple is the only necessary addition in this version.

```
Enqueue b into queue Q
WHILE Q is not empty DO
  dequeue b' from Q
+ IF (a, type, b') ∈ G THEN RETURN true
  FOR all triples (d, dom, b') ∈ G DO put d in L_d
  FOR all triples (r, range, b') ∈ G DO put r in L_r
  FOR all (b'', sc, b') ∈ G DO enqueue b'' into queue Q

FOR all properties p ∈ L_d ∪ L_r
  DO enqueue p' in R

inLd = L_d, inLr = L_r
ProcessedLd = ProcessedLr = ∅

WHILE R is not empty DO
  dequeue p' from R

  Verify that item p' has not yet been processed already:
  − doLd = p' ∈ inLd ∧ p' ∉ ProcessedLd
  − doLr = p' ∈ inLr ∧ p' ∉ ProcessedLr

  IF ¬(doLd ∨ doLr) THEN continue with next item

  IF doLd THEN
  − IF (a, p', ?) ∈ G THEN RETURN true
  − add p' to ProcessedLd
  IF doLr THEN
  − IF (?, p', a) ∈ G THEN RETURN true
  − add p' to ProcessedLr

  FOR all (p'', sp, p') ∈ G DO
    IF doLd THEN add p'' to inLd
    IF doLr THEN add p'' to inLr
    enqueue p'' to R
RETURN false
```

Listing 3.6: Corrected step 5 and 6 pseudo code

The only extra step in this implementation is that there is an additional look-up for a specific triple, while traversing the sub-graph G(sc). This look-up can be performed on $O(log\ n)$ or better (using a sorted graph G). Therefore the change does not impact the total complexity of $O(n\ log\ n)$.

Chapter 4

Experimental Setup

In this chapter we will start by discussing three methods of storing RDFS graphs in external memory. After that the methods of generating test data and their corresponding queries for the experiments are described.

4.1 Indexes

The main purpose of the research was to investigate the performance of the algorithm over three basic physical representations of RDF graphs. The algorithm is deemed to be realistic enough for the experiments. Also the algorithm provides enough variety to deal with different types of queries. It is possible to use it for single look-ups, semi-complex queries which require a limited amount of traversal or inference, and complex queries which will need to do larger traversals to check if a triple exists in graph G. All of this is done without materializing the complete entailment of the graph.

Three methods of storing the RDFS data sets are compared. We will compare the currently most used standard, the state of the art method and an extension to this method. These three different indexes are:

- B-Tree (The current standard)
- Triple-T (State of the art)
- Triple-T with neighbor information (Our extension in this investigation)

4.1.1 B$^+$-Tree Index

The B$^+$-tree index actually consists of three sub-indexes. The method is widely used in common, commercial RDF storage systems such as *Virtuoso* [16] as well as academic projects such as *RDF-3X* [14]. Despite other proposed structures which research often shows performs better, this storage mechanism is currently the most commonly used.

The three sub-indexes make it possible to find triples with one or more blank atoms in the query. Each index contains the whole triple, with the atoms in a specific order. The first index, `SOP`, has triples encoded in the order of subject, object and finally predicate. `PSO` encodes the triples in predicate, subject, object order, and finally `OSP` uses the order, subject, predicate order. As we will discuss later, a variant uses the `OPS` order instead of `OSP` for performance reasons.

The rationale for these orderings is that in most graphs, the subject atom is expected to be the most selective. This means the expected number of triples in the graph with a specific atom in the subject role is lower than the number of triples with a specific atom in the predicate or object role. After the subject, the object role is considered to be the most selective, followed by the predicate.

The predicate often contains atoms from the (relatively small) ontology, which means predicates are often reused across triples. Similarly it is expected that the range of these predicates contains a limited set of values. As such the object role is less selective than the subject, which can be anything.

Although we assume this selectivity order, it may be wise to use a different selectivity order if a graph behaves differently, and adapt the indexes to match that.

To make each sub-index sortable properly, the three items of a triple are concatenated, each prefixed by their individual length in two bytes. The required sorting is preserved, while there is no risk of confusion that may occur if, for example, UTF encoding is used and the three items are separated by 0-bytes (which is another historically common string encoding method). Due the the prefix, sorting is primarily done on the byte length of the atom (generally the amount of characters), followed by a lexicographical sorting. As the bytes are compared directly and they are UTF encoded, uppercase letters are ordered in front of lowercase letters.

This sorting is fine because we do not actually need to do any range-traversal using the data. We only really need to make sure the items are grouped properly: first on the primary atom, then the secondary.

Each index is a B^+-tree on its own. Because all data is sorted, finding the appropriate position of a full triple can be found by traversing the tree top-down. How a lookup is executed exactly, depends on the amount of fixed atoms in the query.

Consider the sample graph G which contains the following triples, describing a fictional office: (Peter, worksFor, Initech), (Milton, worksFor, Initech), (Initech, boss, Bill) and (stapler, missingAt, Initech). This would produce data in each index as in table 4.1. Note that sorting is taken care of in the example.

SOP	**PSO**	**OSP**
(Peter, Initech, worksFor)	(boss, Initech, Bill)	(Bill, Initech, boss)
(Milton, Initech, worksFor)	(worksFor, Peter, Initech)	(Initech, Peter, worksFor)
(Initech, Bill, boss)	(worksFor, Milton, Initech)	(Initech, Milton, worksFor)
(stapler, Initech, missingAt)	(missingAt, stapler, Initech)	(Initech, stapler, missingAt)

Table 4.1: Sample graph as materialized in B-Tree

If all three atoms are fixed, there can be only one match. The `SOP` index will always be preferred. It does not really matter which index is used though. The B^+-tree is traversed down, to find the precise match of the triple. If the leaf bucket does not contain the triple, a negative answer will be returned.

If there is at least one free atom in the triple, the best index will be chosen depending on the available atoms. If the subject is unknown, the `OSP` index will be used, since the object is the most selective of the two available atoms. If the predicate is unkown, `SOP` is the best choice. If the object is unknown, `PSO` is used. Using that index saves having to check many items with the proper subject, as would be the case with the `SOP` index.

The B^+-Tree can in this case – except for the unknown subject – be used to find the first triple in the index, which has the two known items in the right place. Then all items next to it can be scanned and loaded, up until the point where the primary and secondary items no longer match. All these scanned triples are returned as matches of the query.

If the subject is the unknown atom, only the correct object can be found using the index. The first triple with this object is located, and all its triples are scanned linearly. All items with a matching predicate will be added to the result set, others are skipped. This scanning continues until the first triple with a non-matching object is located.

If `OSP` was replaced with an `OPS` index for this case, it is expected that better performance would be achieved as the need to traverse all subjects of triples with the object would be removed. This may be beneficial, but not necessarily. It is no unlikely that all the triples with a single object are still located on the same page in the index, and therefore no extra I/O costs are involved for

this scanning. As such, a downside of using `OPS` is that data locality is reduced which may in fact in some cases have a negative impact on performance.

For these reasons both variants (`OSP` and `OPS`) were tested in the experiments. The `SOP` and `PSO` indexes were kept the same, as following the selectivity reasoning they are already most likely to be the most efficient.

Then there is the case of having just one known atom. In this case it depends solely on the known triple which of the three indexes is used. The index finds the first matching triple and then scans all matching triples in a similar fashion as before.

4.1.2 Triple-T Index

The Triple-T index is designed by Fletcher and Beck [9]. One of the key points of this index is that data locality can make joins more efficient. Specific queries are therefore answered more efficiently.

Whereas the B-Tree index is statement-centric – complete triples are indexed – the Triple-T index is atom-centric. In the original design, the index contains only the atoms themselves. Each item points to a payload of three buckets. Similarly to the B-Tree index these are in `SOP`, `PSO` and `OSP` orderings, with an alternative version having `SOP`, `PSO` and `OPS` orderings. The first atom in each of these buckets is left out of the payload, as this is the role of the indexed atom.

As such, for a triple (A, P, B), each of the three atoms are indexed. The "subject" bucket of A contains the item (B,P). The "predicate" bucket of P has (A,B) and the object bucket of B has an item (A,P) in it. The buckets are sorted, to make finding matching items possible in $O(log\ n)$ using binary search.

Reusing the same sample graph as for B$^+$-Trees, the data would be materialized as in table 4.2. As we will discuss below, the primary atoms are not sorted in any way. Inside each bucket the items are sorted lexicographically.

Primary Atom	**Subject bucket**	**Predicate bucket**	**Object bucket**
Bill			(Initech, boss)
boss		(Initech, Bill)	
Initech			(Milton, worksFor) (Peter, worksFor) (stapler, missingAt)
Milton			
missingAt		(stapler, Initech)	
Peter	(Initech, worksFor)		
stapler	(Initech, missingAt)		
worksFor		(Milton, Initech) (Peter, Initech)	

Table 4.2: Sample graph as materialized in Triple-T

To find a triple, the used index depends solely on whether the subject, object or predicate (in that order) in the query is fixed. This order is explained by the selectivity of these three atoms. As the payload data will generally be loaded completely into memory, as discussed below, the internal sorting will only make a minor difference in performance. After all, there is no difference in the I/O costs involved in loading the data.

This difference means a different payload is selected for those triples where the subject and predicate are known, but the object is unkown. The B-Tree index would use the `PSO` order: this saves having to look through many objects (the secondary atom in the `SOP` order) to find those objects which are used in conjunction with the required predicate.

It is however best to load as little data into memory as possible. Because the subject is considered more selective it will generally have fewer triples in its payload. This means the I/O

costs for loading the subject data will be lower. Therefore, using the `SOP` index instead of `PSO` is expected to generally perform better for these kinds of queries.

As there is slightly less overhead in storing the triples, the space consumption of Triple-T is lower than that of the B-Tree index. Increased data locality also improves efficiency.

Hash Index

The implemented version of Triple-T has some slight modifications, mostly for practical reasons. First, the main index uses a hash-based index, rather than a B-Tree. As the used algorithm does not have ordering constraints in the sub-queries (such as "the object is lower than 5"), using a B-Tree which is sorted is not necessary. In general hash tables use fewer I/Os than B-Trees to do look-ups.

Using a hash table as base for the B-Tree approach is not possible. In many cases the look-ups that occur do not include all three atoms of a triple. Because the B-Tree index is built using complete triples, range traversal is required to prevent having to scan the complete index to find triples if not all parts are known. In Triple-T the index contains single atoms, which removes the need of these range checks.

A hash index is built up such that each key in the index is hashed using a special function. This function should construct a hash that evenly distributes the unique values in separate buckets. There is a limited bucket size. Whenever a bucket grows too large, the hash table size will be increased by splitting the bucket.

Split Index

Because Berkeley DB only supports storing key/value pairs, the payload of a key must be loaded in its entirety at any time. As such it is not possible to skip reading unnecessary buckets and scan only the required parts on disk. To save disk I/O costs, a few more changes were applied.

The main change this has caused is that rather than storing the atom in the index, with its three buckets as the value of the pair, the buckets are indexed separately. In the index, the key is prefixed with an indicator of which triple bucket the value has. In the current use case that does not hurt performance, as the look-ups never require multiple roles of one atom after each other. In the complex parts of the algorithm only chains of triples with transitive predicates are traversed. For each step in this traversal either the subject or object of the next triple is known; and this will change after every step.

Because of that, loading the two other buckets of the payload every time would be useless. As the size of multiple buckets obviously is more likely to exceed a number of disk pages, this could also lead to extra I/O costs. Therefore having each atom in the index up to three times – once for each bucket – is likely to be slightly more efficient.

Secondary B-Trees

In the tested data sets, some resources occur in the same role very frequently. Some data is explicitly stored in the graph, such as the types of all properties and classes (statements such as $(CLASSX, \texttt{type}, \texttt{class})$ and $(PROPY, \texttt{type}, \texttt{prop})$) as well as the many `sp` and `sc` triples.

Whenever a lookup on these frequent atoms is required, Berkeley DB would require to load the entire sub-graph in memory. Additionally, when updating such a value in the data store, the same data had to be loaded, updated and then stored again, which is highly inefficient if the data is too large.

Therefore a "secondary" B-Tree is spawned for each frequent triple whose data size grows too large. This B-Tree indexes only the secondary and tertiary atoms of the bucket. Each of these B-Trees contains the data of exactly one bucket. Because of the I/O overhead to search through this B-Tree (estimated at four disk pages which need to be read), the threshold to spawn this B-Tree is set to a value corresponding to four disk pages.

Using this threshold, it does not matter if either the minimum of five disk pages are loaded to retrieve all data at once, or to look up the items through the B-Tree. Since most frequent triples

in the used data are so frequent that they take up many more pages than these five, this solution is much more efficient.

Compression

The second real change made in comparison to the original proposal is to compress the payloads in the hash index. By using compression the chances of the data spanning multiple disk pages becomes smaller, which will save disk I/Os. To perform the compression, the common zlib compression[1] method is used.

Compression is applied regardless of the data size. This may, if the data is very small, actually result in a slightly larger payload than the original data size [17]. For this to occur the data has to be only a few bytes in size though; the added overhead for zlib will then still not cause the payload to grow beyond one disk page, which is typically 4096 bytes.

The main downside is that compressing and decompressing the data causes additional CPU overhead while loading or saving data. This may cost extra wall clock time. However the goal of this thesis is to reduce I/O costs as much as possible. Smarter strategies to determine whether compression should be applied or not can improve performance of the Triple-T implementation.

Neighbor Data

To make it possible to use one implementation for the base Triple-T index and the Triple-T index with distance – to be discussed next – extra distance information is added to the payloads. This adds overhead of one byte per triple. Obviously it is unlikely that exactly these few bytes cause additional pages to be required to store a bucket.

When secondary B-Trees are used this does imply that besides the key itself, a value is stored, but only if the distance is greater than 0. In the regular Triple-T version that means there is no difference in I/O costs for secondary B-Trees.

4.1.3 Triple-T Index with Neighbor Data

As already briefly mentioned in the Triple-T description, this enhanced Triple-T version ("*Triple-T with distance*" or *Triple-T+d*) adds extra information about the distance between subject and object. The basics are all the same as with the regular Triple-T information. The index mainly adds the option to materialize a small part of the closure when updating the index.

The index recognizes two atoms as being transitive: `sp` and `sc`. This follows from rules 2a and 3a in [13]. Additionally the maximum distance to materialize is configurable. By default this is set to 0, which implies not materializing anything extra. This default will behave the same as the regular Triple-T method. Therefore one generalized implementation was made, to minimize variance between the two implementations. During the experiments merely the materialization distance was configured to change the behavior.

This index adds a fourth element to each triple: the distance it covers. Direct relations, which are not inferred, have distance 0. A distance of 0 is not actually stored in the implementation; rather, if no data is found when looking up a triple, distance 0 is assumed.

For triples with one of the transitive atoms as predicate, the index maintains some inferred triples up to the configured distance. All triples are stored at most once, with the lowest discovered distance. The process to keep the index up-to-date is discussed below.

Consider a graph with non-inferred triples $(a, \texttt{sp}, b)$, $(a, \texttt{sp}, c)$, $(b, \texttt{sp}, c)$, $(b, \texttt{sp}, d)$, $(c, \texttt{sp}, e)$ and $(e, \texttt{sp}, f)$. Each of these have distance 0. When materializing triples up to distance 1 the triples that are stored are shown in table 4.3.

The triple $(a, \texttt{sp}, c)$ is stored with the lowest distance encountered: 0. However, removing the non-inferred $(a, \texttt{sp}, c)$ would result in a new inferred triple $(a, \texttt{sp}, c)$ with distance 1 (through $(b, \texttt{sp}, c)$). Also, the distance of $(a, \texttt{sp}, f)$ would have a distance of 2 and is therefore not materialized.

[1] zlib: `http://zlib.net`

Triple	Distance
$(a, \texttt{sp}, b)$	0
$(a, \texttt{sp}, c)$	0
$(b, \texttt{sp}, c)$	0
$(b, \texttt{sp}, d)$	0
$(c, \texttt{sp}, e)$	0
$(e, \texttt{sp}, f)$	0
$(a, \texttt{sp}, d)$	1
$(a, \texttt{sp}, e)$	1
$(c, \texttt{sp}, f)$	1

Table 4.3: Materialized triples in Triple-T with distance up to 1

Insertion

If this distance is set to 1 or higher, upon insertion of a triple the direct "neighbors" are inspected. For example, when inserting $(a, \texttt{sp}, b)$ the triples of the form $(?, \texttt{sp}, a)$ and $(b, \texttt{sp}, ?)$ are loaded from the index. The returned data contains a fourth element, the distance covered by the mentioned triple.

For each of the triples in this set, the appropriate extra triples are materialized if the new triple's distance doesn't exceed the maximal configured distance. The triples on the left and right of the newly inserted triple are also combined, such that new paths which jump over the newly inserted triple are also properly materialized. This keeps the index consistent. No recursion is required: the payloads of neighboring atoms will contain enough information to discover all inferred triples, since they will have their neighbors readily available in the index.

Removal

Removing a triple is slightly more complicated. This is mainly because there is no guarantee that triples which are possibly inferred by the removed triple are no longer inferrable (within the configuration limits) after removing the triple. In fact, the base triple which is removed might still be inferrable through other triples.

As such the removal of a triple takes place in two steps. First all possibly inferred triples are also removed. This includes only those materialized triples where the distance matches the inference distance. If the distance is different, the materialized version has a lower distance (keeping in mind that only the smallest distance is stored). This inference can then not occur through the just removed triple.

The next step is to check for each removed triple, if it is still inferrable within the configured bounds. If a triple (a, p, c) is still inferrable, there is a triple (a, p, b) with distance 0 and a triple (b, p, c) with any distance d, for which $d + 1$ is lower than the configured maximum distance.

It is possible to retrieve these triples by doing two look-ups in the index. The first is to find all bs using the index of a as subject. The second look-up is to use b as subject, and to check if the pair (c,p) occurs in the payload (recall the `SOP` ordering of the bucket).

Also, all possible matching bs have to be checked. There may be multiple matching paths, but it is not possible to determine beforehand if a specific b results in the shortest path inferring (a, p, c). The shortest distance of this triple needs to be reinserted.

This insertion is performed in the same way as any other triple insertion. The main difference is that the initially inserted triple has a higher distance than 0. This is a simple generalization in the insertion algorithm, which needs to take the base triple's distance into account when finding the inferred triples to materialize.

As there may be multiple triples which have been removed initially, one may at first not be reinserted because the required interim triple (b, p, c) was removed earlier, and was not yet materialized because it is later in the list to check for reinsertion. When that triple is inserted

later on, the insert algorithm will correctly update the index by inserting triples which were skipped earlier.

As shown, updates to graphs using this partial materialization is not trivial. While inserting new data can still happen in constant time by just checking the neighbors, removing a materialized triple can have a noticable impact on performance. Of course the level of this impact depends on the degree of the nodes, as well as the distance up to which triples are materialized.

There may also be extra page I/Os to do a single look-up, due to increased payload sizes if part of the entailment is materialized. While checking for a chain of triples with a transitive predicate, fewer look-ups have to be done in total. Because each look-up can easily use 3 or 4 page reads, the extra costs quickly pay off – especially as soon as two or more traversal steps in the algorithm can be skipped.

This extra data is materialized only for triples with transitive predicates. These are usually in the ontology. The instance data is left untouched. For example, the mrdfs rules state that $(A, \texttt{sp}, B)$ and (X, A, Y) infer (X, B, Y). However, when a graph already contains (X, A, Y) and the triple $(A, \texttt{sp}, B)$ is inserted, the inferred triple is *not* materialized.

4.1.4 Expected Results

It has already been shown that the use of Triple-T is likely to achieve better performance than the B-Tree index [9]. The same is expected to hold when using external memory to store all data. The impact of the applied changes to the original version of Triple-T are also expected to benefit Triple-T even more..

Running queries using transitive predicates – which make up the complex parts of the entailment algorithm – are expected to yield better results when part of the closure is materialized, as in Triple-T+d. This does require a small adjustment in the algorithm.

The normal procedure to check if a path from a to b exists using a predicate p, is to pick a starting point (a or b) and find all initial triples ($(a, p, ?)$ and $(?, p, b)$). For every result of this look-up, the same step is repeated until either there are no results anymore, or the requested path is found.

With the extra materialized triples these sub-queries will immediately return triples one or more steps further. Any returned triple with a distance lower than the maximum does not need to be traversed over; the data that would return, is either already in the result set, or it will be found through the traversal on those triples with the maximum distance.

While the B-Tree and base Triple-T indexes will not benefit from this extra logic, they are also not negatively impacted. When using Triple-T with distance having to do fewer look-ups in the data set will give the index an advantage, especially for those transitive queries.

Summarized, the expectations of the performance of the three indexes are (from worst to best):

Look-up with non-transitive predicate: B-Tree, then Triple-T and Triple-T+d (which will perform equally well).

Single look-up with transitive predicate: B-Tree, then Triple-T+d, then Triple-T. The difference between Triple-T+d and Triple-T is expected to be minimal.

Traversing transitive predicate: B-Tree, then Triple-T and finally Triple-T+d.

Data size: B-Tree is expected to be the largest, followed by Triple-T+d, with Triple-T being the smallest.

In conclusion, Triple-T+d is expected to perform the best in real-world scenarios, when doing look-ups. We will not be comparing update costs of these indexes, but at least updating will be more costly in Triple-T+d than in Triple-T, when the updated triple contains a transitive predicate.

Whether using Triple-T+d over Triple-T is beneficial in real-world scenarios depends mostly on how often the ontology is updated, the size of the ontology and if triples with transitive predicates such as `sp` and `sc` are regularly updated.

4.2 Test Data

The complexity of the tested algorithm in this thesis depends mostly on the ontology size. Larger and more complex subclass and subproperty graphs imply that the algorithm has to traverse more triples in these trees. Therefore, to properly test the algorithm to the fullest and compare the different indexing methods, synthetic data has to be generated. Preferrably this data follows a realistic structure, but it has to be sufficiently large to ensure that the algorithm is realistically capable of dealing with large data sets – up to a size larger than commonly found in Semantic Web data.

As shown in earlier research, many realistic structures follow a power-law distribution. This is not restricted to Semantic Web data alone [3, 23, 24]. In the sense of the RDFS graphs, this means the top level classes and properties in the `sc` and `sp` sub-graphs have a high degree. The degree per node decreases in each level. Therefore the top class has many subclasses, but each of those has fewer subclasses. Because of that, `sc` and `sp` graphs are generally shallow.

We will discuss two types of generated data sets. For each type, multiple graphs of different sizes were generated.

4.2.1 PoweRGen

The first attempt in this was made using *PoweRGen*, a tool from ICS-FORTH [23]. This was created by Yannis Theoharis as part of his Master's project, and is designed to create realistic Semantic Web graphs.

Unfortunately there were several practical limits in using this tool. The main problem is memory consumption. In order to generate all data, several large structures are kept in memory. A linear problem (simplex) solver processes the data to generate a structure. For sufficiently large graphs, containing more than several thousands of classes or properties, the tool consumed all available memory or was terminated because it exceeded the java (32 bit) limit of 4 Gb of memory.

Additionally, for unclear reasons, the tool could not create data sets with both subclass and subproperty graphs. As such two separate executions were made to create subclass graphs. In one of these, all classes were renamed to properties and the subclass relations were changed to subproperty relations, for the purpose of this thesis.

It also turned out that if the same parameters are used, the structure of the subclass and subproperty graphs is exactly the same. This means there is little variation in the sub-graphs. This is supported by the repetition in the average length of `sp` and `sc` chains, as shown in table 4.4. All in all, while the structure of the graphs is expected to be realistic, the size and complexity is fairly limited.

Because of the aforementioned problems, it was not possible to generate large graphs using this tool. Instead, to make the graphs sufficiently large for the experiments, they were combined. An extra class was added, to which up to a 100 graphs were connected through a triple of the form $(CLASS0, \texttt{sc}, NEWCLASS)$. Each data set also has all classes, properties and atoms prefixed with a unique prefix, to make them distinguishable. If more than 100 graphs were added, multiple extra classes were added, which were in turn also connected in the same manner using another extra class. This process was repeated for properties in the generated data.

PoweRGen also does not generate any instance data, which is required for the algorithm we focus on in this research. Therefore instance data was generated after merging several smaller graphs. PoweRGen already defined domains and ranges for the properties it had generated. For the extra properties generated to expand the graphs, domains and ranges were not defined. The queries generated for these graphs did not require this information, as we will discuss later.

The instance data was generated as follows: for each class in the graph, 10 instance atoms were defined. The list of properties using the class as either domain or range was collected. If the property list was empty, or with a 25% chance, the type was assigned directly to the class, by adding a triple of the form $(ATOMX, \texttt{type}, CLASSY)$. Otherwise, a random property is chosen from the collection of available properties.

The property has the class as domain, range, or both. If both, the atom is randomly assigned as domain or range with a 50% chance for each. A random atom of the other class is picked, and a triple ($ATOMX$, $PROPERTYX$, $ATOMB$) is added to the database.

Several characteristics, such as the `sp` and `sc` chain lengths were extracted from the data sets. These are presented in table 4.4. The n column represents the amount of smaller graphs combined into the data set. Each sub-graph contains 1,000 classes and properties each. The total degree (`totalDegreeVRExp`) is set to 0.7. The chances of a class having a degree of 0 (i.e. classes which do not have subclasses and are not a subclass themselves, `p0`) was set to 0.25. Another 25% of the classes are leafs (`percentageOfZeros`). The percentage of properties with equal domain and ranges (`selfLoops`) was set to 0.126, and the target depth (`depth`) was set to 10.

The contents of the used parameters.conf is in listing A.1 (appendix A). This excludes the path configuration. For each graph, PoweRGen was run twice: once to generate a `sc` graph, and once to create the taxonomy which defines the properties, `sp` relations and domains/properties of the properties. Due to problems with the tool, this could not be properly done in one run.

Set	n	**Total triples**
100	100	1,459,711
200	200	2,919,438
400	400	5,838,863
500	500	7,298,576
750	750	10,947,865
1000	1,000	14,597,136

(a) Basic statistics

Set	`sp` Triples	`sp` Chains	**Max** `sp` chain length	**Avg** `sp` chain length
100	30,100	29,800	4	1.326
200	60,202	59,600	5	1.510
400	120,404	119,200	5	1.510
500	150,505	149,000	5	1.510
750	225,758	223,500	5	1.510
1000	301,010	298,000	5	1.510

(b) `sp` Subgraph statistics

Set	`sc` Triples	`sc` Chains	**Max** `sc` chain length	**Avg** `sc` chain length
100	29,600	29,200	4	1.897
200	59,202	58,400	5	2.086
400	118,404	116,800	5	2.086
500	148,005	146,000	5	2.086
750	222,008	219,000	5	2.086
1000	296,010	292,000	5	2.086

(c) `sc` Subgraph statistics

Table 4.4: PoweRGen Sets Statistics

4.2.2 Boost Graph Data

To make it possible to have deeper graphs, another approach was also taken to generate synthetic data. The data sets needed to be sufficiently large, with the property and class sub-graphs sufficiently deep. Additionally to make the graphs resemble realistic ontologies, it is required that these sub-graphs follow the power law structure.

To achieve this, the Boost C++ libraries[2] were used – specifically Boost Graph. This library is able to generate graphs following a Power Law Out Degree algorithm. This algorithm generates

[2]Boost C++ Libraries: `http://www.boost.org/`

a graph from three parameters, n, α and β. Each vertex in the graph (n in total) is allocated a number of credits, using a power-law distribution. Edges are added and in each case a credit is deducted from the source vertex. The β and α parameters define the initial number of credits, $\beta \cdot x^{\alpha}$ with x a random value between 0 and $n-1$. The β parameter therefore influences the average degree of vertices, whereas α determines how steeply the power-law curve drops off. A large α value indicates a steep curve [7].

By running this algorithm twice, the required subproperty and subclass relations are created. This did result in many cycles in these graphs. Since a hierarchical structure is more realistic in ontologies (although not required by RDFS), and also has less complexities in computing statistics of the graphs, these cycles were removed. By keeping track of the processed properties or classes while doing traversal over these trees during the main algorithm, the existence of cycles can be checked and termination of the process can be guaranteed.

To combine the two sub-graphs each property was assigned a domain and range from the list of classes. Of 12.5% of the properties RDF literals were assigned as domain or range. The necessary instance data was generated in the same way as with the PoweRGen data.

Several of these data sets were generated. Each one was created using the n parameter of the algorithm as main varying factor. The values used are 500,000, 1 million, 2 million and 4 million respectively. The α parameter was set to 0.7 every time, the β parameter was adjusted to generate sufficiently deep graphs. The value for α was chosen to get a good combination of deep graphs, with the root nodes having a high degree. It may not be the most realistic scenario, but these data sets were designed to be generally worse cases than realistic data sets.

The total size of the graphs is somewhere in between 15 and 16 times the value of the n parameter. This can be split up as follows:

- 2 times n `type` statements of the classes and properties
- 2 times n `dom` and `range` statements respectively, for each property
- 10 instance atoms/statements per class, for n classes
- 1 to 2 times n atoms for the subproperty and subclass information.

The first three items in the list are by definition. The last item depends on the randomly generated graphs, and may vary. The exact statistics of the data sets can be found in table 4.5.

4.3 Queries

After creating the data sets, it became possible to extract queries from these sets. There are various distinct steps in the algorithm, of which at any time just one step is executed (taking into consideration that steps 5 and 6 are combined to one step).

To compare the physical representations with different use cases, four classes of queries were constructed. The first class (see figure 4.1a) is a simple look-up. It executes step 1 by checking for a triple of the form $(?, \texttt{dom}, ?)$ or $(?, \texttt{range}, ?)$, of which the subject is a known property and the object a known class.

The second class is a single-step traversal using `sp` (see figure 4.1b). First the disconnected `sp`-triples were found. These are triples $(A, \texttt{sp}, B)$ for which A and B are not used in any other `sp`-triples. For a random selection of these, a random triple of the form (P, A, Q) is looked up. Subject P and object Q in such triples are instance atoms. The query then becomes (P, B, Q). This is a direct application of rule 3b.

The third class is very similar to the second (see figure 4.1b), except there is no limitation to the length of the `sp`-chain. In fact, the longest chains were located first and used as base for the queries. An instance using the lowest subproperty in a chain was located as (P, A, Q). A query then becomes (P, B, Q) with B the highest superproperty in the chain (i.e. $(A, \texttt{sp}, B)$ exists in the graph or its entailment).

Set	n	α	β	**Total triples**
500k	500,000	0.7	5,000	8,028,454
1m	1,000,000	0.7	7,500	15,853,402
2m	2,000,000	0.7	13,000	32,109,285
4m	4,000,000	0.7	20,000	63,649,886

(a) Basic statistics

Set	`sp` Triples	`sp` Chains	**Max** `sp` chain length	**Avg** `sp` chain length
500k	518,505	3,714,433	40	8.734
1m	930,898	3,830,223	38	6.807
2m	2,059,727	29,942,857	68	14.906
4m	3,842,154	25,328,322	67	9.449

(b) `sp` Subgraph statistics

Set	`sc` Triples	`sc` Chains	**Max** `sc` chain length	**Avg** `sc` chain length
500k	513,877	5,893,165	49	11.462
1m	930,341	3,875,103	35	6.785
2m	2,065,237	602,270,600	181	7.131
4m	3,839,035	29,796,685	65	11.135

(c) `sc` Subgraph statistics

Table 4.5: Boost Data Sets Statistics

The fourth class of queries are those following implicit typing (see figure 4.1c). The longest chains of `sp`-triples are located in the graph, and the domain and range of the last property (the super-property of all others) are located. In the figure, this is the `sp`-relation between *prop1* and *propN*.

Of these classes identified by the domain and range, the longest available `sc`chains are located in a similar way. These chains start with a subclass, going up to a root superclass using `sc`-triples. In the graphical example, the domain is *class1* with the superclass being *classN*.

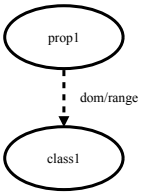
Next an instance of the property at the start of the `sp`-chain is located, and the final query is generated. This is of the form ($ATOM$, `type`, $CLASS$) which needs as many as possible instantiations of the given ρdf rules to get to a match. If the longest located `sc`-chain starts at the domain of the property, $ATOM$ will be the domain of the instance and $CLASS$ will be the last class in the `sc`-chain (*classN*).

The full procedure then was set up as follows. Each data set was loaded and 100 queries of each class were generated using the given data set. Half of these 100 are "positive" queries: the triples are known to exist in the data set. The other 50 are "negative" queries. These triples do not exist in the entailment of the data set. For such triples generally more data will have to be traversed when running the tested algorithm, which means the time it will cost to answer these queries, will be greater than for the positive queries.

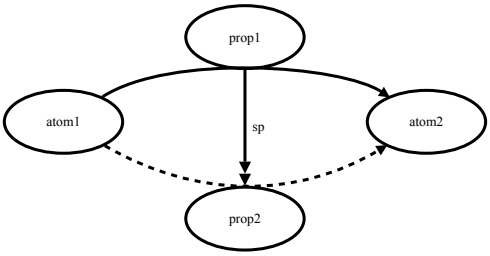
The negative queries were generally created by replacing the subject or object in the query with another (randomly chosen) resource of a similar type. As such the form of the query is the same. These replacements are chosen such that the complexity of answering the query is kept high, for example by using data of `sp` or `sc` chains with similar length.

In the actual experiments, the data was first loaded once for each physical representation. Next, each set of 100 queries was executed 100 times in a row. To make sure the internal cache in between queries would not affect the results, after each query the cache was flushed. The easiest way to do so in Berkeley DB is by disconnecting (closing) the database, and re-opening it. As such, a clean database cache was used for every query against the database.

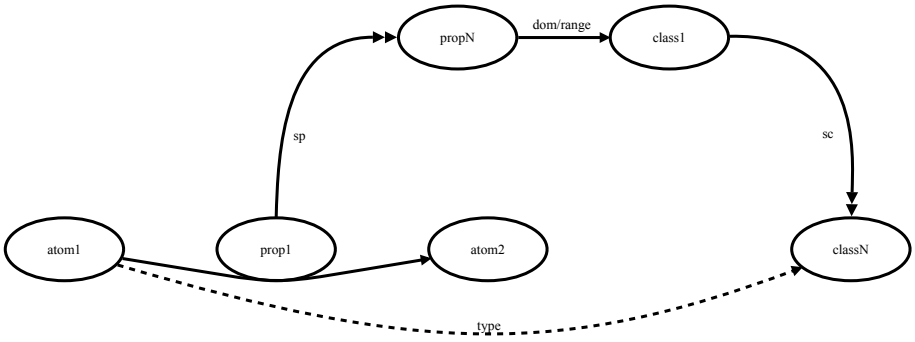
After each query several statistics are recorded. This includes statistics like the amount of pages read into the database, the time it took to execute the query and more. Merging these statistics gives the end results, discussed further in this thesis.



(a) Simple look-up



(b) Transitive traversal



(c) Complex "type" query

Figure 4.1: Forms of Generated Queries
(Double arrows indicate chains of multiple triples.)

Not all queries were generated and used for both types of data sets. Due to the limits in complexity of the PoweRGen graphs, it was not possible to generate complex enough queries – or proper queries at all – using the second and fourth classes. Because the first class will behave similarly for both data sets, as it is a simple look-up and nothing else, that too was used only for the Boost graph data. With the complexity of the fourth class already used to experiment with the Boost graph data sets, the third class was used only for the PoweRGen data. In summary, that means classes 1, 2 and 4 were used only for the Boost graph data sets, and class 3 was used only for the PoweRGen data sets.

Chapter 5

Implementations

This chapter discusses the framework that was developed and used to run the experiments with. The implementation of the entailment algorithm is highlighted.

5.1 Global Architecture

To run all experiments and to be able to compare all results, a C++ application was created. To provide abstraction from system interfaces and for its convenient API, the Qt toolkit[1] was chosen as base toolkit to develop the application with. The global architecture is shown in figure 5.1.

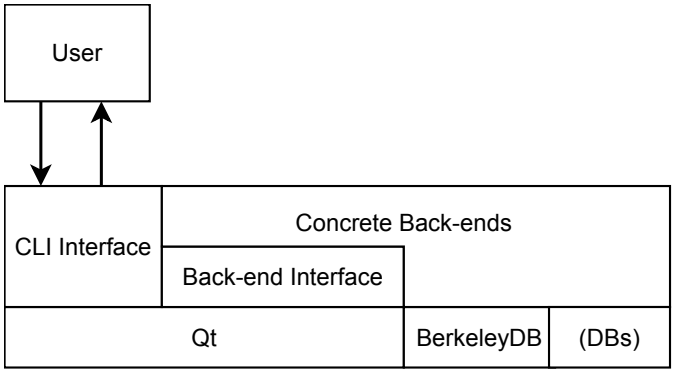


Figure 5.1: Global Implementation Architecture

The application implements the entailment algorithm using an abstract storage back-end interface. This algorithm can be executed, along with many other things, using a command line interface. Other commands included in the implementation in a similar fashion are those to generate data, do look-ups, insert and remove data and many others.

The storage back-end is defined by an abstract interface, which can be implemented in various ways without having to modify the actual algorithm. As such it can be guaranteed there are no differences in the implementation of the algorithm itself, which may influence the experiments.

The three discussed different back-ends were implemented. Each of these used Berkeley DB as base storage (the C library to be precise), in which the indexes are created. As such, the experiments are not influenced by the use of different base libraries. There may, however, be subtle differences caused by Berkeley DB, as in the first case a B-tree was used as main index, whereas the other two used a hash-index.

[1] Qt: `http://qt.nokia.com/`

Other back-ends that were implemented use a relational database (with support for MySQL[2] and SQLite[3]) and one which uses Virtuoso[4] through Soprano[5]. These were mostly used for the algorithm verification process, but not for any performance analysis as discussed in this thesis.

The architecture supports the use of multiple back-ends. The special `connect` command will spawn the correct concrete back-end which will be interfaced – using the back-end interface – by the other commands. As said, for this thesis all used back-ends use Berkeley DB as storage base, but a concrete back-end can use any storage mechanism as base. This means the application supports running the algorithm using any storage and indexing mechansism, including existing RDF engines, relational databases or plain file storage.

The user interacts with the application by entering commands. To automate processes helper scripts can be created which list the commands to execute. These can simply be fed to the application's standard input. Such scripts were created to automate processes included but not limited to the generating of test data and running the experiments.

Limited effort was put into making the implementation efficient. Using Valgrind[6], specifically the Memcheck tool, memory leaks were kept to a minimum. Its callgrind tool was used to optimize some code paths, while Massif was used to measure overall and peak memory usage.

However, the focus of this research is on comparing three different storage implementations with an equal interface, especially based on the amount of page reads from disk. It was more important to get comparable results, rather than having an highly optimized implementation. Therefore it is likely that the overall performance of the implementation can still be improved.

The application has extensive features to manage RDFS data, including commands to insert, remove, list, count and check for triples. For an overview of the available commands in the application, see appendix D.

5.2 Entailment Check Implementation

The entailment algorithm is also implemented as a specific command of the application. The implementation differs from the theoretical version in ways described in section 3.1. After validating the correctness of the algorithm, the required corrections were also implemented. These corrections will be discussed further in this thesis. Additionally there are a few generalizations in the implementation, making it more flexible.

The main reason is that, while the algorithm supports the required constructs for mrdfs, ontologies in general may use predicates which behave similarly to those in mrdfs. So rather than just sticking strictly to the algorithm, the application supports "*rule sets*." The default rule set is the one defined by the mrdfs rules.

The sets are loaded prior to running the algorithm, with the base mrdfs rules always loaded first. For each set the algorithm is repeated, until a valid match is found. For the experiments conducted in this thesis, no extra rule sets are used.

Custom rule sets can be a subset of similar rules. They consist of predicates which are mapped to the original mrdfs predicates such as `sp`and `sc`. Some parts of the algorithm may require that there is a mapping for several original predicates, because the rules are not limited to a single ρdf atom.

The generalization using these sets is straightforward. At any time a look-up is performed using an atom in ρdf, that atom is replaced with the atom that maps to the original ρdf item. If there is no mapping in the rule set, the query is simply skipped, as it can never be satisfied using the given rules. In some cases this implies a complete part of the algorithm can be skipped, depending on the rule set.

[2]MySQL: `http://www.mysql.com`
[3]SQLite: `http://sqlite.org`
[4]OpenLink Virtuoso: `http://virtuoso.openlinksw.com/`
[5]Soprano: `http://soprano.sourceforge.net/`
[6]Valgrind: `http://valgrind.org/`

The various steps in the algorithm are also split up into smaller parts, to make the code more comprehensible and maintainable. Each of these parts is executed by a wrapper. If a part is not valid for the query because the predicate P does not match the predicate required for the step, that part of the algorithm immediately returns false, so the next one is attempted. If a part returns true as answer to the query, then the remaining parts are not checked: the answer is immediately returned.

Besides that, a basic look-up of the query is always performed first, before any step of any rule set is executed. This is required for steps 1 (`dom` and `range` triples) and 4 (triples with the predicate not in ρdf) and even completely implements step 1 of the algorithm. As such there is no performance penalty for these steps.

For steps 2a and 3a (transitivity of `sp` and `sc`) the lookup result would, if existing, be returned by the first iteration of the traversal, so there is a slight performance penalty for those steps. The look-up is also unnecessary for steps 2b and 3b (reflexivity of `sp` and `sc`) as similar look-ups (with the atom or object blank) are performed during the algorithm itself.

Finally, for the `type` queries (steps 5 and 6) the implementation is also such that the explicit look-up is skipped in the first iteration of the `sc` traversal. Therefore there is no actual performance penalty.

All in all the extra check may incur a small performance penalty for some cases. However, in other cases it may improve performance as the algorithm will be able to terminate earlier and may not have to load any unnecessary data. Since the implemented traversals follow a depth-first search strategy, the improvement can in fact be fairly large, depending on the position of the necessary data in the first iteration. Additionally, when using multiple rule sets, the simple check makes it possible to skip running the algorithm for the first rule sets, until a matching rule set is used.

Recall that the algorithm is implemented once for all storage back-ends. As such, the penalty is at most one look-up per indexing technique. Since the same data and test queries were used in our experiments, the exact I/O cost solely depends on the used indexing technique. The results are therefore still comparable.

Chapter 6

Experiment Results and Analysis

This chapter covers the results of the empirical experiments. It starts with the experimental environment that was used, and the metrics that were measured. Next we will discuss the individual database sizes for each index, followed by the query statistics and comparisons of the three indexes.

6.1 Environment

All experiments were executed on the same machine. This is a moderate work station, equipped with the following hardware and software:

Brand Dell OptiPlex 980 MT

Chipset Intel® Q57 Express Chipset

CPU Intel Core i5-650 3.20 Ghz, 2 cores with VT, 4 Mb cache, 64-bit x86-64

Memory 4 Gb RAM (2x2048 Mb, 1333 MHz DDR3 Dual Channel)

Hard disk 320 Gb (7200 RPM), SATA II 3 Gb/s

OS Fedora release 14 (Laughlin)

Kernel Linux 2.6.35.11-83.fc14.x86_64

File system Ext4

Qt Qt 4.7.1

Berkeley DB Berkeley DB 4.8, C version

Compiler GCC 4.5.1 20100924 (Red Hat 4.5.1-4)

The framework is single-threaded. Therefore only one of the available cores was used. The experiments were run with as few other applications running as possible. Most importantly no graphical desktop enviroment was started. As such the experiments were executed with as little external interference as possible.

For the experiments, sets of queries were generated for each data set as discussed in section 4.3. The data sets were loaded using each representation as discussed in section 4.1. The Triple-T with distance information was configured to materialize 1 step of `sp` and `sc` triples.

The queries were then executed 100 times in a row on the corresponding data set, using each physical index. In between each query, the Berkeley DB cache was flushed by closing the database, removing the cache files and re-opening the database. The cache size of Berkeley DB was set to 1 Mb, the page size to 4 Kb, for all experiments. With cache sizes under 500 Mb, Berkeley DB increases it by 25% (to 1.25 Mb in this case) for cache overhead.

To ensure memory usage is limited, Valgrind's tool Massif was used for one separate run of each experiment. This records the peak memory usage of the application. The peak amount of memory used was consistently lower than 150 Mb. This includes the memory consumed by shared libraries, such as Qt, and memory mapped files used for Berkeley DB's cache. Most of this (69%) was definitely used by Qt, about 28% was occupied by libdl, and only a mere 1.14% was used directly by Berkeley DB for its memory mapping files.

As such it is obvious the framework cannot load the databases into main memory completely, validating the measures of disk I/O below.

Several statistics were recorded for each query that was executed. These were:

- Cache hits and misses (as returned by Berkeley DB's `DB->stat()` function).
- Pages read and written (as returned by Berkeley DB's `DB->stat()` function).
- Real ("Wall Clock"), User and System time (as returned by `getrusage()` on Linux).
- Minor and major page faults (page reclaims and page faults, as returned by `getrusage()` on Linux).
- Block input/output operations (as returned by `getrusage()` on Linux).

These measures are at any time limited to only the execution of the algorithm. Overhead of opening and/or closing the database is not included

The amount of pages read for Berkeley DB is equal to the amount of cache misses. Naturally, no pages are written during the running of an experiment, so those values remained 0 for all experiments.

The statistics for the minor page faults and block output operations show the same behavior as these cache misses. This makes sense, as each cache miss means a page has to be loaded from disk. This results in the same amount of page faults on the OS level: the 4k page size was chosen to be the same as the page size of the underlying file system. As Berkeley DB memory maps its cache to files, each page that is loaded into the cache is also written to the memory mapped file, resulting in several block output operations. Hence these statistics are omitted from this thesis.

Unfortunately due to the lack of root access on the used PC, it was not possible to clear the OS cache in between each execution of experiments or even each query. To clear this cache properly the PC had to be rebooted in between each runs. Therefore, the results in the Timings section (6.3.2), are the averages of 5 runs using a cold cache. Each of these runs were executed almost directly after starting the PC; the databases were not accessed after the reboot and prior to running the experiments.

6.2 Database Sizes

The sizes of all databases are shown in tables 6.1 (PoweRGen) and 6.2 (Boost Graph). For the B-Tree index the sizes of the `SOP`, `PSO` and `OSP` buckets are shown separately. For both Triple-T indexes, the sizes of the main index and the secondary B-Trees are shown.

The sizes of the `OPS` variants of the indexes were also inspected. There were no significant differences these sizes compared to the `OSP` variants. Therefore, tables 6.1 and 6.2 only show the sizes of the `OSP` variants.

The statistics also showed that the following secondary B-Trees were generated for all Triple-T based data sets:

Subject buckets of *None*

Predicate buckets of `dom`, `range`, `sc`, `sp`, `type`

Object buckets of `class`, `prop`

Set	B$^+$-Tree			Triple-T		Triple-T+d	
	SOP	PSO	OSP	Main	B$^+$-Trees	Main	B$^+$-Trees
100	69.7 MB	128.2 MB	106.2 MB	159.4 MB	55.2 MB	159.5 MB	55.7 MB
	304.14 MB			**214.68 MB**		**215.25 MB**	
200	140.8 MB	260.7 MB	214.9 MB	319.7 MB	109.8 MB	319.9 MB	111.4 MB
	616.57 MB			**429.48 MB**		**431.40 MB**	
400	282.9 MB	525.2 MB	431.2 MB	647.2 MB	218.0 MB	647.6 MB	220.4 MB
	1.21 GB			**865.25 MB**		**867.98 MB**	
500	353.9 MB	652.6 MB	539.5 MB	1.2 GB	270.5 MB	1.2 GB	274.5 MB
	1.51 GB			**1.50 GB**		**1.51 GB**	
750	530.8 MB	979.6 MB	808.6 MB	1.3 GB	402.9 MB	1.3 GB	410.8 MB
	2.26 GB			**1.65 GB**		**1.66 GB**	
1000	707.6 MB	1.3 GB	1.1 GB	2.5 GB	534.7 MB	2.5 GB	544.9 MB
	3.01 GB			**2.99 GB**		**3.00 GB**	

Table 6.1: PoweRGen Data Set Sizes

Set	B$^+$-Tree			Triple-T		Triple-T+d	
	SOP	PSO	OSP	Main	B$^+$-Trees	Main	B$^+$-Trees
500k	353.8 MB	642.0 MB	559.9 MB	1.2 GB	238.8 MB	1.3 GB	309.9 MB
	1.52 GB			**1.48 GB**		**1.55 GB**	
1m	705.2 MB	1.2 GB	1.1 GB	2.5 GB	472.2 MB	2.5 GB	592.6 MB
	3.03 GB			**2.95 GB**		**3.07 GB**	
2m	1.4 GB	2.6 GB	2.3 GB	5.0 GB	978.1 MB	5.0 GB	1.3 GB
	6.29 GB			**5.94 GB**		**6.26 GB**	
4m	2.9 GB	5.2 GB	4.5 GB	10.0 GB	1.9 GB	10.0 GB	2.4 GB
	12.62 GB			**11.88 GB**		**12.44 GB**	

Table 6.2: Boost Data Set Sizes

The biggest of these was consistently the predicate bucket of `type`. This contains all type atoms, which were also materialized for all properties and classes as part of the data set. For obvious reasons the size difference between base Triple-T and Triple-T with distance information was mostly visible in the predicate buckets of `sc` and `sp`. The other extra B-Trees did not differ in size significantly.

These B-Trees support the choice discussed in section 4.1.2 to use the `SOP` index if both the subject and predicate are fixed in a look-up (i.e. a look-up of the form $(S, P, ?)$). Using the `PSO` bucket would for the common cases – such as `sc` and `sp` predicates, which are used a lot in the queries – result in having to look inside the secondary B-Trees to find the necessary matches.

Except for the *500k* and *1m* data sets, the total sizes of both Triple-T versions are slightly smaller than or at worst as great as the B-Tree versions. For the *500k* and *1m* sets, the Triple-T with distance version is slightly bigger, while the base Triple-T size is always smaller than that of the B-Tree versions.

Especially in the PoweRGen data sets, the Triple-T versions are significantly smaller than the B-Tree versions. Because there are fewer `sc` and `sp` triples in these sets and because these trees are shallow compared to the Boost Graph data sets, the impact in size for materializing the necessary extra data is very minor.

The major exception to this is the *500* set, for which the Triple-T sizes are close to the B-Tree sizes. Especially the main index grows significantly in size compared to the *400* set, yet the *750* set is not much bigger. Most likely the *500* set reached a fill factor threshold which triggered an increase in the hash table size. This doubles the size of the main hash table, resulting in a jump in

consumed size. The same issue seems to appear for the *1000* set, although there is no larger set to compare with. A better growth strategy may reduce this problem, albeit with possible drawbacks of having to grow more gradually and therefore reducing update performance.

As such, especially with larger data sets, using Triple-T does save some space over using B-Trees, up to roughly 5% for the Boost Graph data sets and even 20 to 30% for the PoweRGen generated graphs.

Recall that the implemented version of Triple-T indexes each atom three times, once for each bucket. Space usage of Triple-T is probably lower if the three buckets are combined into one, as the hash table will grow less quickly. However, because the payloads will be larger, performance may be negatively impacted as it is more likely that a payload occupies multiple pages.

6.3 Statistics

This section will show various measurements of the experiments. The base charts will show the sum of all executions and all queries. For example, when showing the total amount of pages read for a specific data set, index type and query class, this number is the sum of all 100 executions of all queries in that class. Each execution runs 50 positive and 50 negative queries.

Furthermore, the graphs are split into three main query classes: positive queries, negative queries and all queries combined. Positive queries are those triples which do exist in the given data set, negative queries are those triples which are not, and the combination of the two will give the sum of all classes.

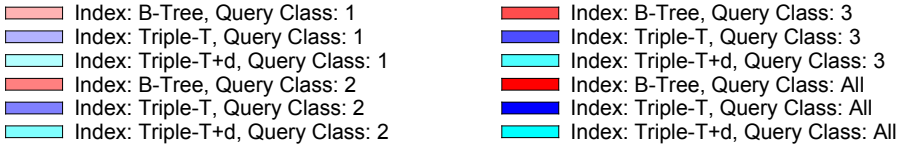
The PoweRGen data set graphs have only one query type. Therefore three bars are shown in each group, one for each physical representation. From left to right these are B-Tree, Triple-T and Triple-T with distance.

For the Boost Graph data sets, each of these groups contains of twelve bars each. These are divided in four smaller groups of three bars each. Each group represents a query class – the type of queries as discussed in section 4.3. The first group are the plain look-ups, the second group the single-step traversals and the third group the complex `type` queries. The fourth group is the sum of all these, or the overall average for charts showing averages per query. This fourth class encompasses 300 executions of 100 queries each: 100 of each query class.

Again, each bar of these four groups represent the physical representations that are used, in the same order as for PoweRGen.

Many charts use a logarithmic scale, as the values between the query classes or even the used indexes are vastly different.

(a) PoweRGen data legend



(b) Boost Graph data legend

Figure 6.1: Legend of graphs

The averages that are listed represent the arithmetic average per single query. The legends of all graphs are in figure 6.1.

6.3.1 Cache Misses

Whenever Berkeley DB needs to load a page (4 Kb) of the loaded database, it first checks its internal cache (of 1 Mb) if the page is available directly. Increasing the cache size will naturally improve performance, as the chances of a page being available in the cache become greater. However, a greater cache size also implies more memory consumption. As explained before, the cache size was limited to measure the performance under fairly bad scenarios.

Figures 6.2 and 6.3 show the amount of cache requests and cache misses Berkeley DB encountered during the experiments. The amount of requests is the sum of cache hits and cache misses. The cache misses correspond to loading of pages which were not available in the cache. Furthermore the fraction of cache misses over the total amount of cache requests is shown separately.

PoweRGen Experiments

As can be seen in figure 6.2a, running the experiments using the PoweRGen data sets requires the most pages to be found in the expected efficiency order: B-Tree loads the most pages, followed by basic Triple-T and then Triple-T with distance.

However, looking at the amount of cache misses in figure 6.2b, basic Triple-T stands out for negative queries. It seems the working set for B-Tree is more efficiently used than base Triple-T, which using fewer cache requests still has more cache misses.

The clear winner in all cases is Triple-T with distance, which needs to do less traversal. As the queries are simple traversals – using look-ups of triples with transitive predicate alone – the benefit of having extra information of these relations materialized is great, even though the transitive chains are relatively short.

The percentage of cache misses over the total amount of cache requests, shown in figure 6.2c, support the above conclusions. Especially for negative queries, which potentially require more branches of the transitive tree to be traversed show that B-Tree is able to re-use most pages in its cache – relatively more than both Triple-T versions. However, as the absolute number of cache misses is crucial in measuring the performance using external memory, the Triple-T with distance index performs better than anything else.

Boost Graph Experiments

The results of the experiments using the complex Boost Graph data make one thing clear: no matter how large the graph is, the B-Tree solution requires the most pages to be loaded. The amount of cache requests (figure 6.3a) is clearly higher than with the Triple-T based indexes. The differences are especially large for the less complex queries.

As expected, Triple-T and Triple-T with distance perform equally well on the two base query classes. The extra data that is materialized in the enhanced version is not used for these queries. For complex queries, it becomes clear that adding the extra data has its benefits. Whereas the basic Triple-T has about as many cache misses as the B-Tree solution, this amount is almost always lower when a small part of the entailment is materialized (figure 6.3b).

The only exception are the positive queries in the *2m* set, where base Triple-T performs slightly better than the version with distance. The exact cause is unknown, but this may be due to characteristics of the queries or the data set. It is possible that several queries could be answered without any traversal, in which case the enhanced version may require more data to be loaded.

Additionally, in the *500k* data set, while the total amount of cache requests for basic Triple-T is lower than the B-Tree set, the amount of cache misses is slightly higher. This can be caused by the fact that for each look-up, Triple-T has to load payload data – which may include data that is not immediately required. This causes the cache to fill quicker, which makes the working set smaller. As such, it may happen that some data needs to be re-loaded during the execution of a query. Increasing the cache size slightly may therefore reduce this difference, or even make Triple-T perform better than B-Tree. Triple-T with distance does perform batter than both other indexes in this case.

A final benefit of Triple-T is the fraction of cache misses over the total cache requests. In the charts the difference in cache requests between Triple-T and Triple-T with distance is fairly small, compared to the difference in cache misses. As shown in figure 6.3c, the percentage of cache misses over the total amount of cache requests is – for complex queries – the lowest when using B-Tree storage. This means the working set in the cache is better re-used while computing the results. However, the total amount of cache requests is so much higher that the number of cache misses is still significantly higher than with Triple-T.

On the other hand, the percentage of cache misses for Triple-T with distance is much lower than base Triple-T. Taken into consideration that the total amount of cache requests is also lower, this shows that Triple-T with distance indeed performs significantly better than the basic Triple-T version.

6.3.2 Timings

To compare the execution time of the queries, three times are measured. The first is *real* or *wall clock* time. This is the total time the process took in seconds. The *user* time is the time spent in user mode on the system, whereas the *sys* time is the time spent in kernel processes. This corresponds to the results of, for example, the `time` command on UNIX-like systems. As discussed earlier, the amount of runs making up these statistics is limited. Rather than having 100 runs, there are just 5. This may make the statistics less accurate.

PoweRGen Experiments

The results in the PoweRGen set are not quite as you may expect, considering the cache misses discussed in section 6.3.1. In all cases the B-Tree solution takes less time than both Triple-T versions, with no signifant differences between the latter two, as shown in figure 6.4a.

The wall clock time shows that for the smallest set, using either Triple-T version it takes about three times as long to answer a query, opposed to using the B-Tree version. The difference decreases as the data set grows. Figure 6.4b shows that the biggest differences are caused by the negative queries, which in all cases require a longer time to get a result than the positive queries.

Figure 6.4a also shows that Triple-T uses significantly more time in user and kernel mode compared to B-Tree. These do not include the waiting for disk pages to be loaded. This is likely to be due to the compression that is applied. The used Triple-T implementation adds overhead on the CPU level after each page load.

Another cause of relatively bad performance of Triple-T can be found in the way the data is structured and stored. Recall that the PoweRGen data set has queries which simply traverse a single `sp` or `sc` chain. The naming of the used properties can benefit the B-Tree solution. These names are not random, but `Class0`, `Class1`, ..., `ClassN`, as generated by PoweRGen. These names are prefixed with a sub-graph identifier. The chains do not spread across multiple sub-graphs, but are local within a single PoweRGen generated graph. The same goes for the properties.

This means there is more predictable data locality in the B-Tree. All look-ups performed by the algorithm are of the form $(A, \texttt{sp}, ?)$ or $(A, \texttt{sc}, ?)$, depending on the query. This means only the `PSO` index is used. Also, as the classes in a chain will be neatly grouped together, they are also more likely to be on disk pages closer to each other. This means the hard disk has to seek less to load the required pages: it may be able to just read-ahead the required pages from disk.

In general a hard disk has an average seek time of 6-10 milliseconds. That means 1 or 2 extra seek operations per query for Triple-T can already completely cause the difference in the results. This is not unlikely: in the Triple-T implementation, there is one big hash table for all buckets. The subject buckets of the corresponding classes will be requested by the algorithm. These may be scattered all over the index file, causing the disk to seek the correct pages more often, causing the delay.

Restoring some of the original Triple-T properties may help. Depending on the bucket sizes, a graph may perform more efficiently if all three buckets in Triple-T are joined together in one key. The hash table is less likely to fill up and may remain smaller. However, as originally motivated,

overflow buckets are also expensive if they require loading more data than required. This is a trade-off to be made depending on the RDFS graph and its usage.

The same goes for compression, which has a lesser impact but is still visible. Not compressing data in Triple-T until it becomes too large to fit on one page can reduce the CPU time required to load data.

With more random names for classes and properties, and with `sc` and `sp` chains, the data locality in the B-Tree solution will also be reduced. This is expected to degrade the performance of the B-Tree implementation for these kinds of queries.

Having said that, it is clear that performance on all accounts is still good. Even though the traversals are shallow, generally 4 to 5 steps, the framework can still answer 50 to 100 queries per second (or 3,000 to 6,000 per minute) on a moderate PC. Better hardware and a larger cache can vastly increase that number.

Boost Graph Experiments

The results for the Boost Graph data sets are in figure 6.5. A first glance shows that the results are much closer to the results shown by the cache misses discussed in section 6.3.1. There are a few cases were base Triple-T requires slightly longer to return a result than using B-Tree. This happens mostly in the complex queries. With the larger data sets, there is a modest benefit for Triple-T with distance, which performs slightly better than the other two physical representations.

Figure 6.5a also shows that Triple-T often requires a little more CPU time compared to B-Tree. As discussed with the PoweRGen timings, this may be due to having to decompress the data from memory.

Overall the differences are not very large. Even with a cold cache, each of the three indexes perform almost equally well. However, the Triple-T with distance index does perform at least as good as – but mostly better – in all cases when looking at the all queries combined in each data set.

Figure 6.5b also makes it clear that the data set size does not influence the time it takes to answer basic queries much. There is more variation in the complex queries, but this can also be an artifact of the characteristics of the different data sets. As shown in section 4.2.2, the chain lengths within the data sets do differ plenty to cause such variations.

For the simple queries – single look-ups and traversing one step – performance looks very good. Not much more than 0.02 seconds is required to answer a single query, on average. In fact, the most basic look-ups get a result back in under 0.01 second in most cases. That results in being able to answer 50 to 100 queries per second, or 3,000 to 6,000 per minute.

The complex types of queries are clearly at a disadvantage. With a cold cache it quickly takes more than 1, and regularly even 2 or 3 seconds to get a result. Despite the moderate hardware, that means performance can degrade to as much as being able to answer only 20 queries per minute. However, considering that the lengths of these queries are longer than in realistic scenarios, performance is still not too bad. Additionally, using a bigger cache and not running from a cold cache, can easily improve performance.

6.3.3 OSP vs OPS index

As discussed in section 4.1.1, there is a choice to be made whether to order the object-based bucket in each of the physical representations either in `OSP` (Object, Subject, Predicate) order, or in `OPS` (Object, Predicate, Subject) order. Using the `OPS` order in combination with the `SOP` and `PSO` ordered indexes makes it possible to do efficient look-ups for all forms of look-ups. It is not necessary to scan many secondary items to find matching tertiary items.

However, data locality is reduced and therefore `OPS` may in fact require more I/O operations when executing multiple look-ups in a row, or when performing join operations.

The previous analysis in this study were based on the results using the `OSP` index. We will now compare variations where the `OSP` order was replaced with an `OPS` index order, in all three storage back-ends. For clarity, only the overall results are used for the Boost Graph data sets.

That is: the result of all three used query classes is combined. In most cases, the complex query class (class 4) will influence this overall number the most.

In the charts, each index is repeated using two bars. The left of these pairs is the `OSP` variant, as discussed in the previous sections. The second bar represents data gathered using an implementation with the `OPS` index order.

PoweRGen Experiments

As the cache requests and misses show for the PoweRGen data sets, figure 6.6, the benefit of using `OPS` is almost non-existent. As expected, the Triple-T indexes are practically not impacted at all. For the B-Trees, the number of cache requests in fact lies a little bit higher for the negative queries, when using `OPS` (figure 6.6a).

As figure 6.6b shows, regarding the amount of cache misses the differences are slightly bigger when using B-Tree storage. Especially for the negative queries, using `OSP` as index order seems to be more beneficial. The amount of cache misses and even cache requests is higher when using `OPS`. This suggests that the working set in the limited cache is put to use more efficiently when using `OSP` order.

Regarding the execution times as shown in figure 6.7: these timings were recorded using a hot OS cache. Therefore, page faults on the high level generally could be solved without physically accessing the disk. Therefore, the timings are different from the cold cache results discussed previously.

These times, especially the wall clock time in figure 6.7b, show some interesting results for the Triple-T based indexes. While there was no visible impact concerning the amount of page reads when switching from `OSP` to `OPS`, Triple-T is impacted mostly in a negative way. Although the differences are minor, they are visible.

Recall that the object bucket in Triple-T is loaded whenever the object is known, but the subject of the look-up triple is unknown. As such, the subject is always a blank in look-ups where the object bucket is used. The expected result would be that `OPS` would therefore give slightly better results: rather than having to scan all subjects for possibly matching predicates, the right predicate can immediately be found and all corresponding subjects can be returned, without actively scanning the remained of the bucket. This should normally be more efficient.

This suggest that the problem lies with loading the data as it comes from the store into more useful memory structures. The exact cause is not easily determined, though.

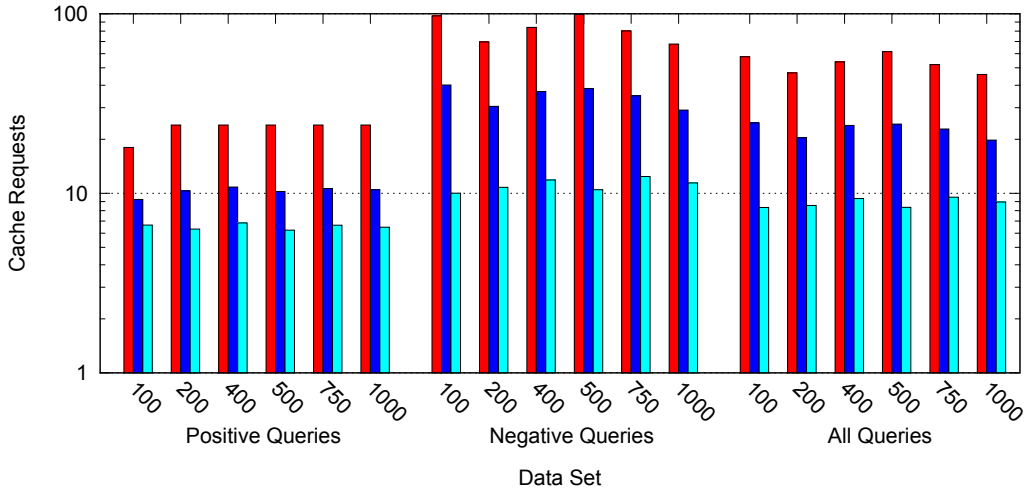
Boost Graph Experiments

Looking at the similar statistics for the Boost Graph data sets (figure 6.8), there again is no clear winner. Triple-T based indexes still perform pretty much the same for both cache requests and cache misses. In the B-Tree index the amount of cache requests (figure 6.8a) tends to be slightly lower for `OPS`, except for the negative queries over the *2m* data set. The amount of cache miseses is, however, pretty much equal (figure 6.8b). Therefore the fraction of cache misses (figure 6.8c) is slightly higher.

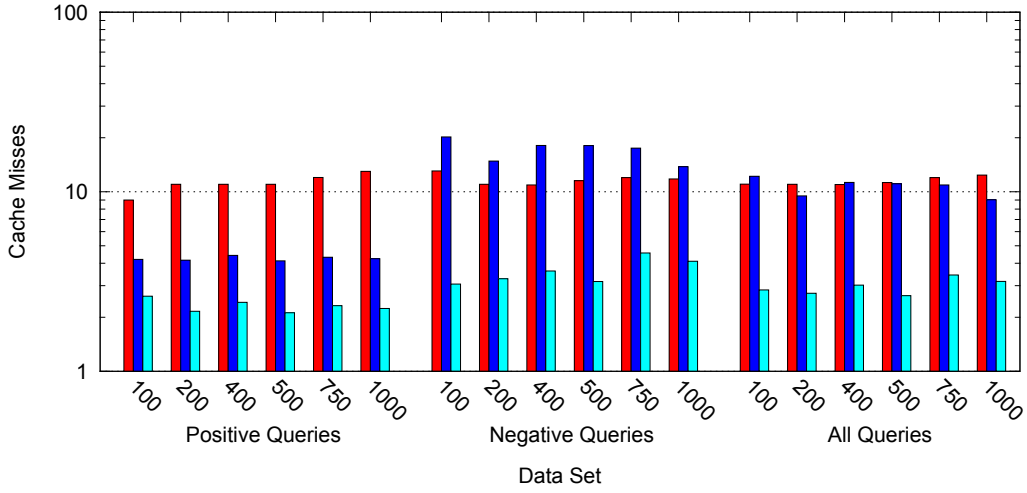
Because the absolute amount of cache misses is the same, there is no significant improvement of using `OPS` instead of `OSP` ordering. However, as the amount of cache requests is lower in `OPS` there may be a very minor benefit depending on the implementation: loading fewer pages also means that fewer pages have to be processed. While there is no gain on the I/O level, there might be a minimal profit in overall timings.

The timing results in figure 6.9 exhibit similar behavior for Triple-T as with the PoweRGen sets. `OPS` performs slightly worse, contrary to expectations. Again the differences are quite minor.

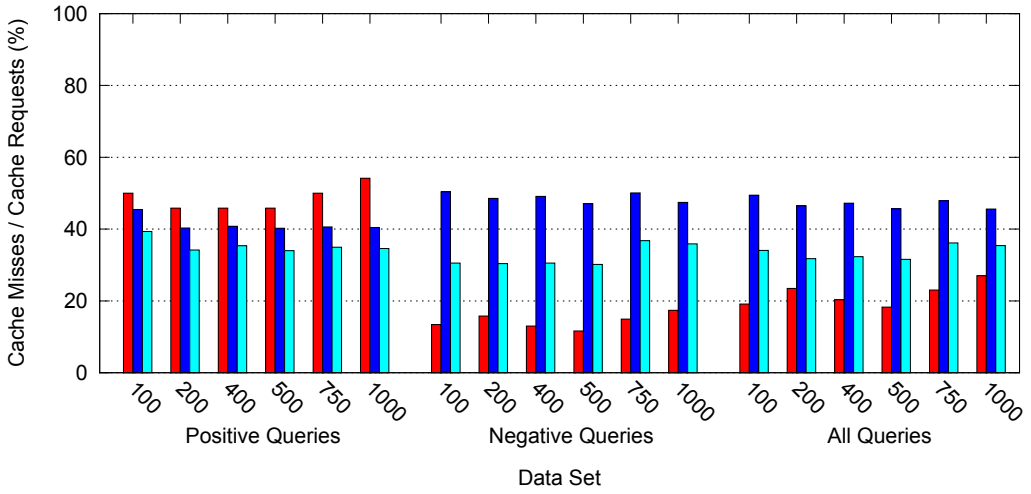
In conclusion, there is no significant difference between organising the object bucket of the indexes in subject, predicate order or predicate, subject order. The results are at least comparable and often even pretty much the same. Therefore the order is best selected by making an informed decision using the characteristics of the RDFS graph that is stored, and the queries that will be executed.



(a) Cache Requests Average per Query

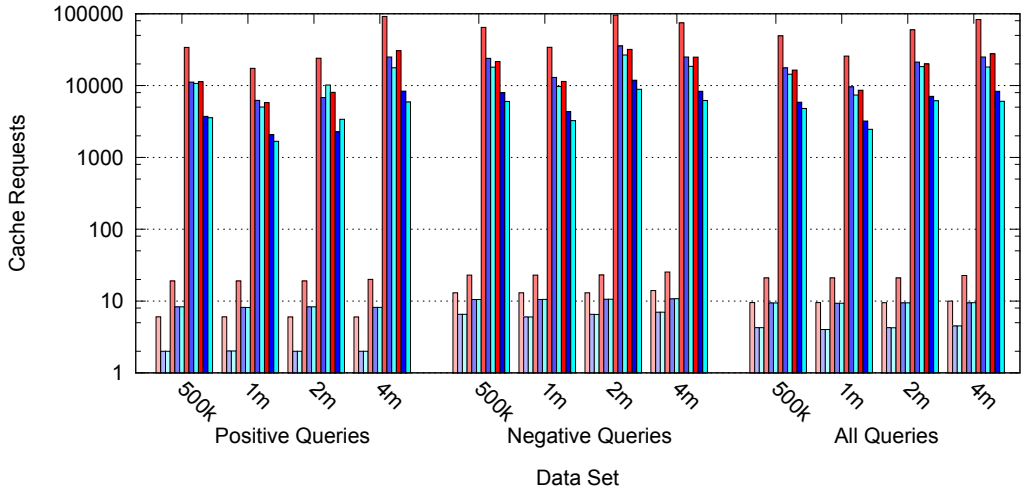


(b) Cache Misses Average per Query

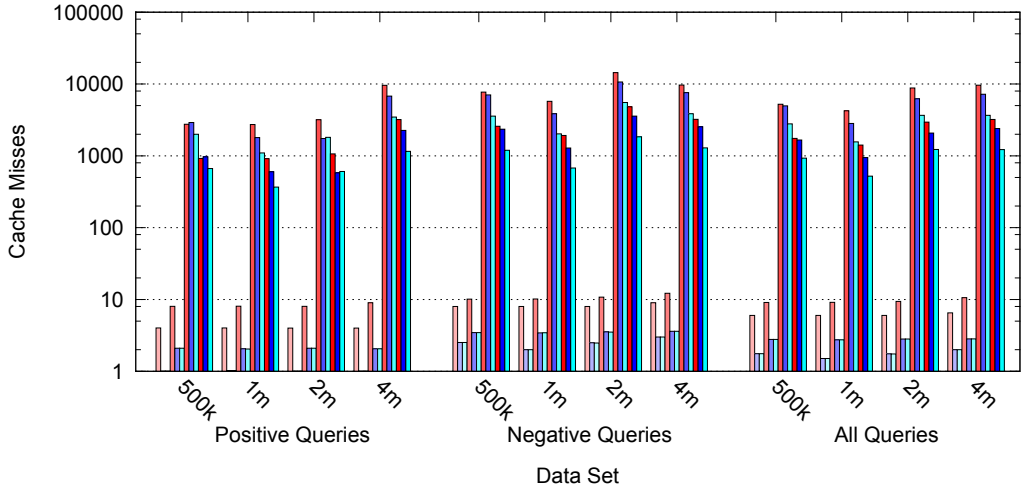


(c) Fraction of Cache Misses

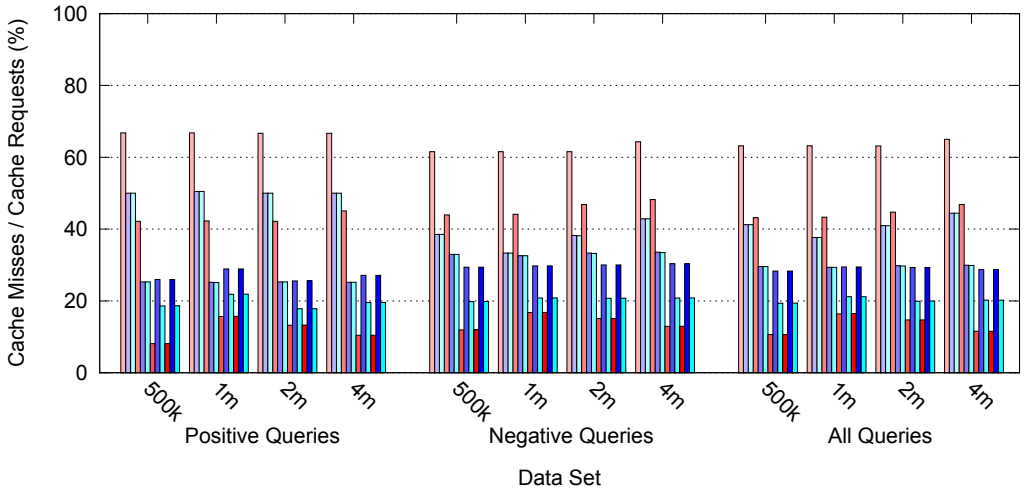
Figure 6.2: PoweRGen Graph Cache Statistics



(a) Cache Requests Average per Query

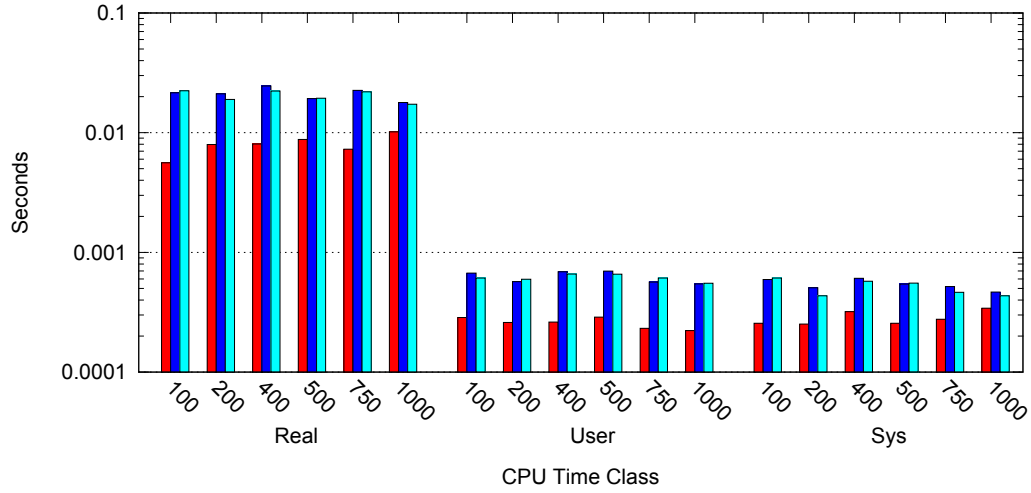


(b) Cache Misses Average per Query

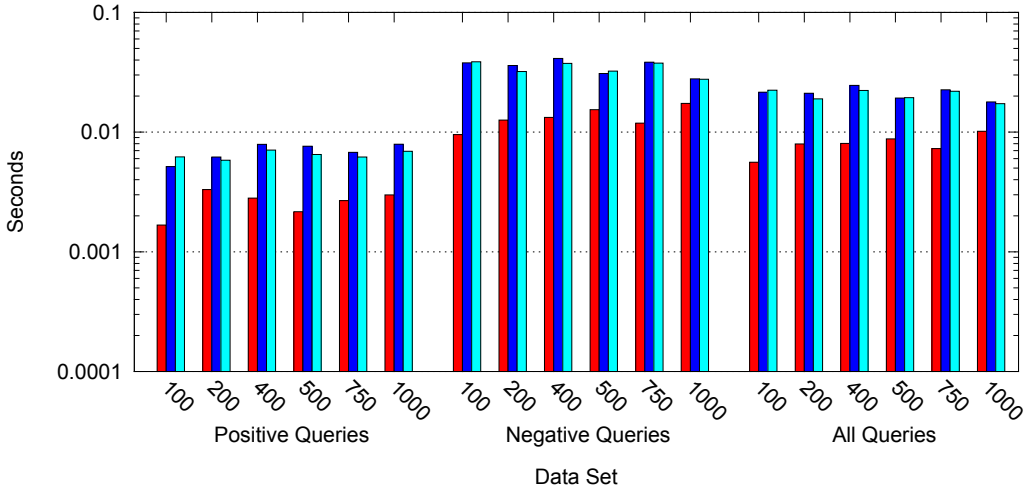


(c) Fraction of Cache Misses

Figure 6.3: Boost Graph Cache Statistics

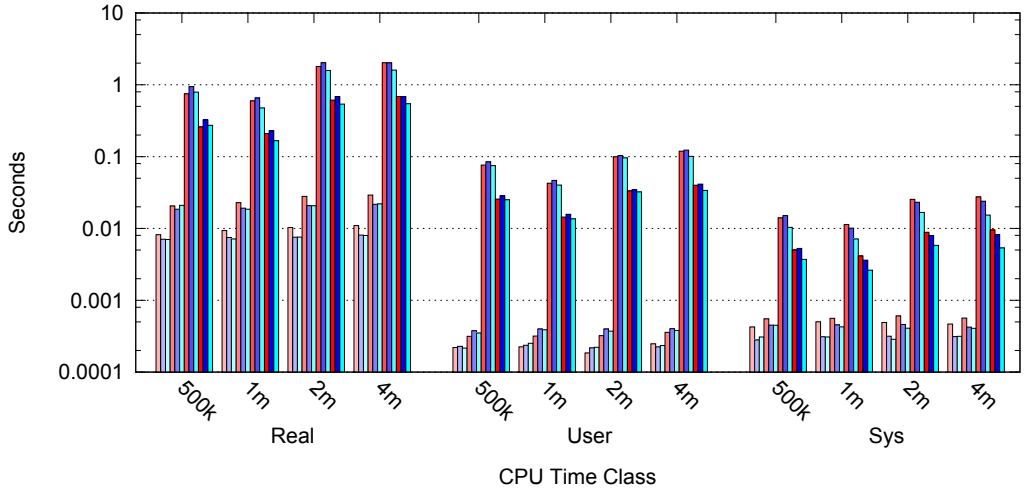


(a) Overall Execution Times

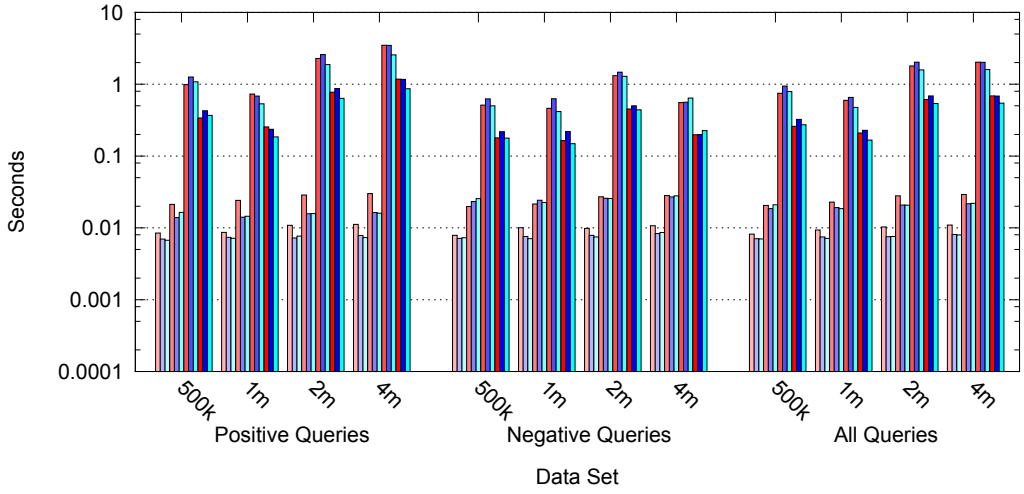


(b) Wall-clock Times per sort of Query

Figure 6.4: PoweRGen Graph Time Statistics

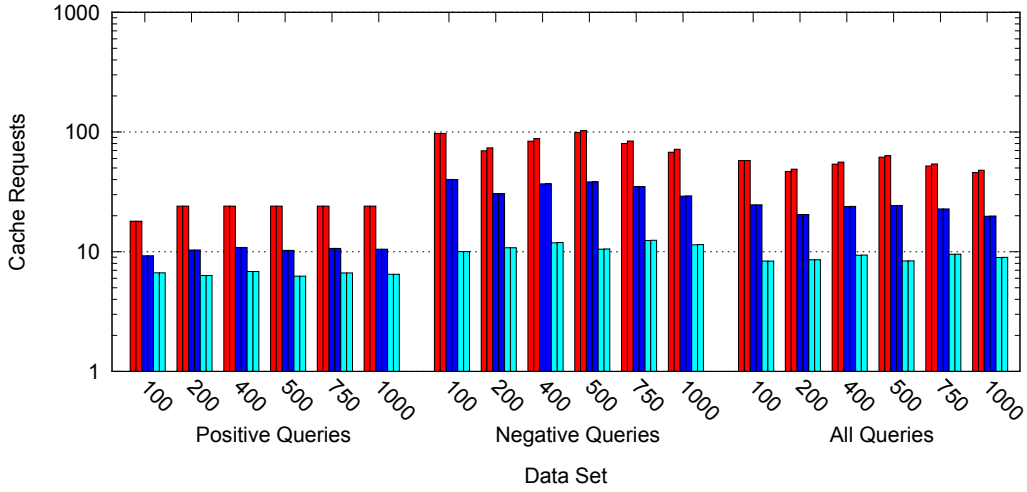


(a) Overall Execution Times

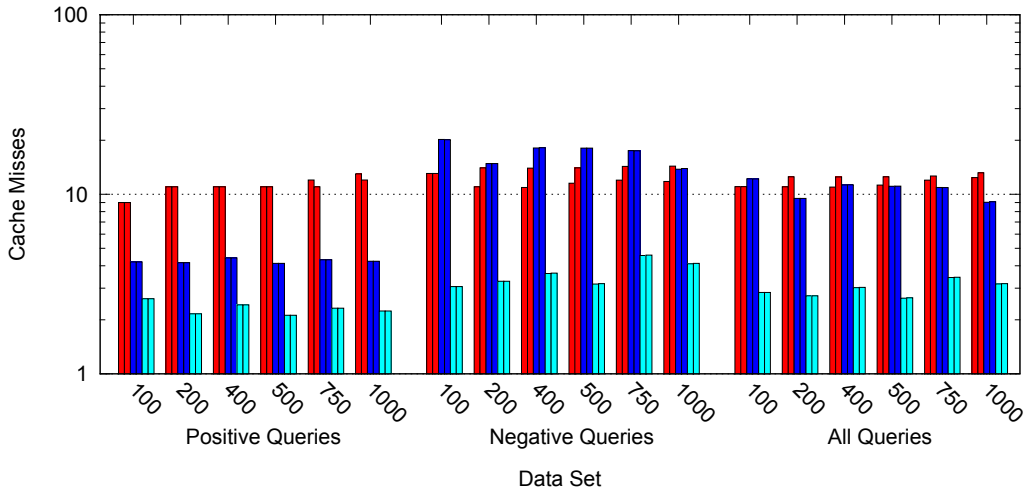


(b) Wall-clock Times per sort of Query

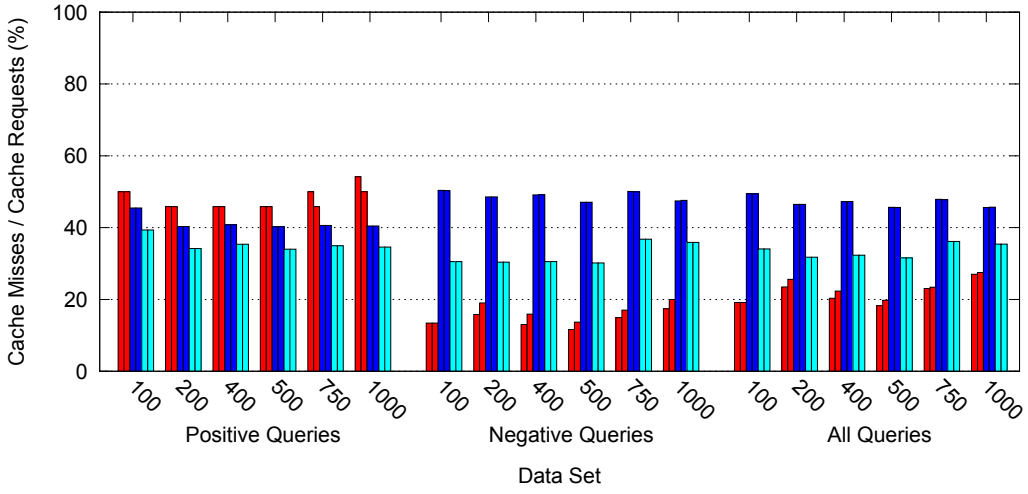
Figure 6.5: Boost Graph Time Statistics



(a) Cache Requests per Query

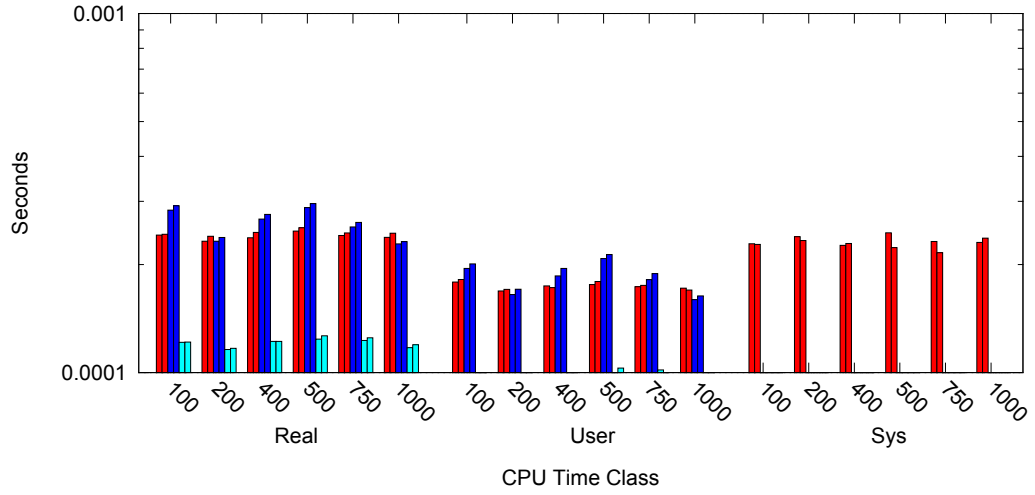


(b) Cache Misses per Query

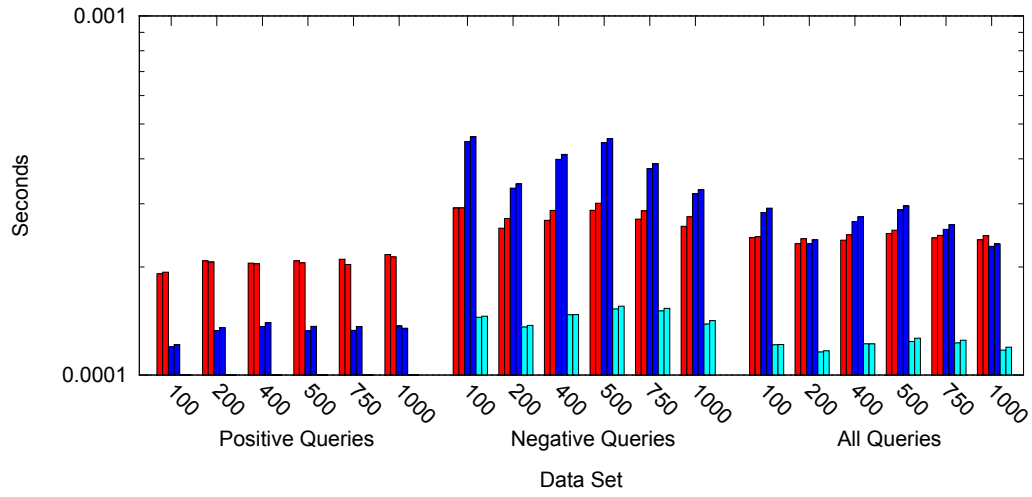


(c) Fraction of Cache Misses

Figure 6.6: OSP vs OSP Comparison: Cache (PoweRGen)

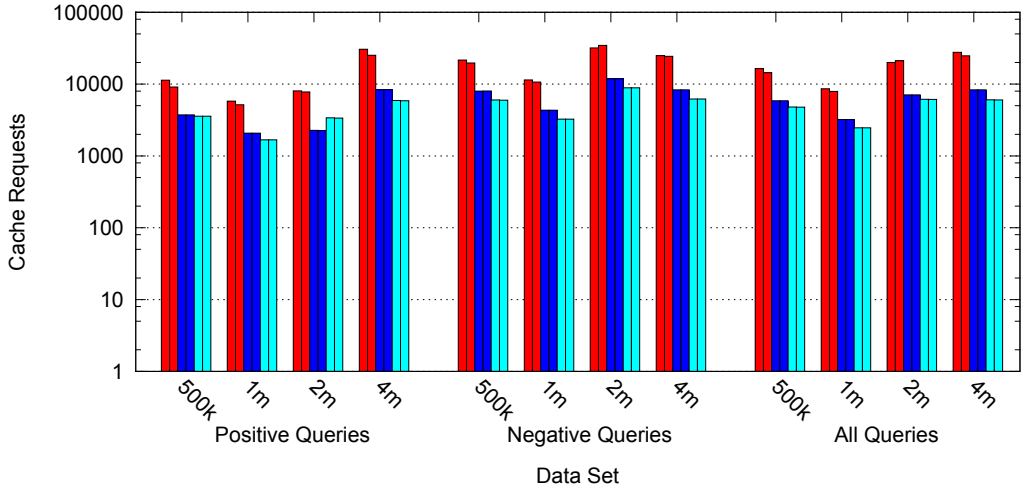


(a) Overall Execution Times

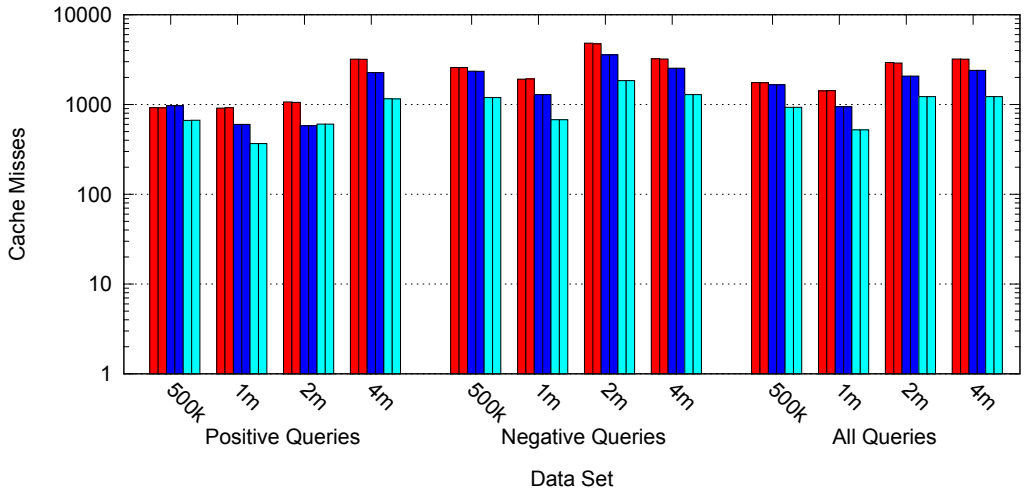


(b) Wall-clock Times per sort of Query

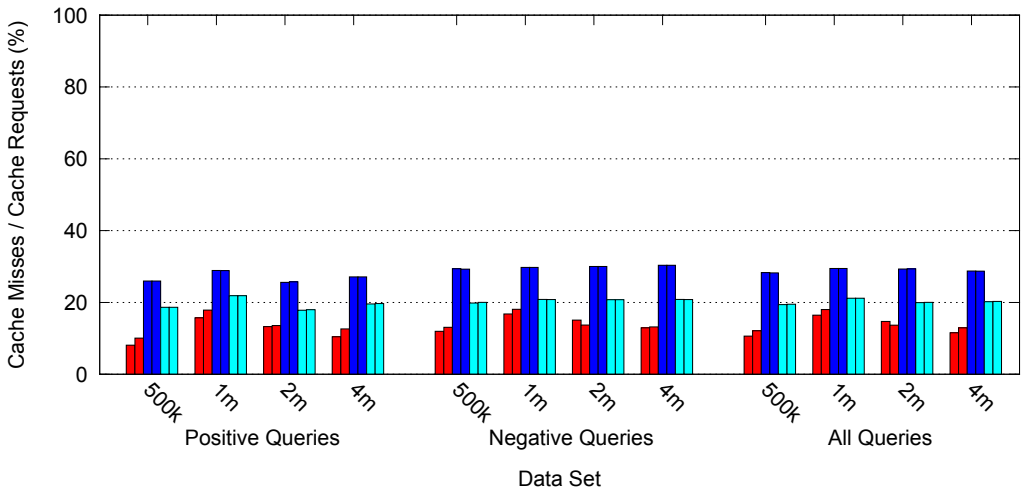
Figure 6.7: OSP vs OSP Comparison: Time (PoweRGen)



(a) Cache Requests per Query

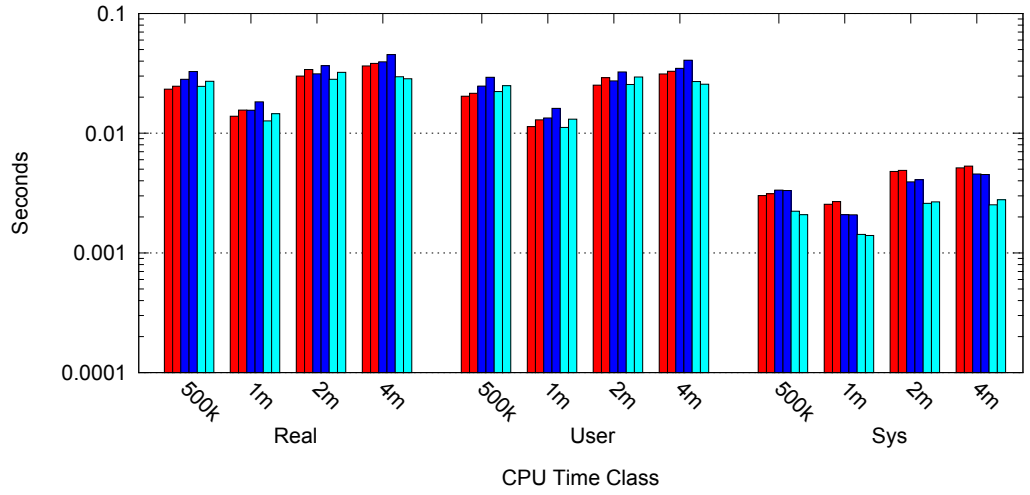


(b) Cache Misses per Query

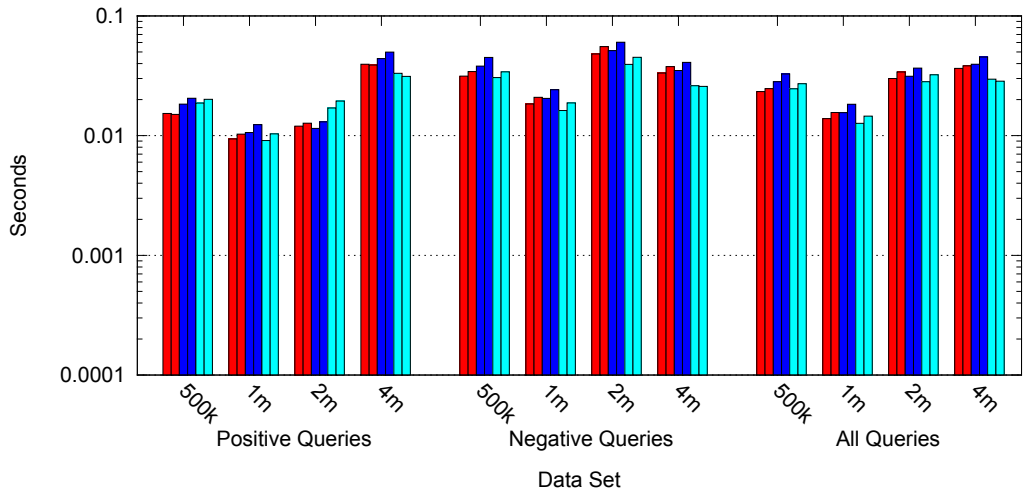


(c) Fraction of Cache Misses

Figure 6.8: OSP vs OSP Comparison: Cache (Boost Graph)



(a) Overall Execution Times



(b) Wall-clock Times per sort of Quer

Figure 6.9: OSP vs OSP Comparison: Time (Boost Graph)

Chapter 7

Conclusions

As shown throughout this thesis, it is not trivial to efficiently determine if a triple is entailed by an RDFS graph or not. A proposed algorithm was first modified and (slightly) extended for practical reasons. Its implementation was then tested. A subtle mistake was discovered and corrected. The algorithm was also adjusted for specific single-triple computations, allowing early termination in the more complex queries. Additionally, support reflexivity of `sp` and `sc` was added.

Next test data was generated, which follows the structure of real-world data. Graphs of various sizes were generated. To test the actual scalability of the algorithm, the ontology sizes in some of these graphs are much larger than what can be expected in most real-world data.Because the algorithm is mostly bound by the size of the ontology – instance data matters less – it was shown that even under quite extreme circumstances, the algorithm will perform well.

The test data was generated using two different libraries, PoweRGen and Boost Graph. The former, while tailored for RDFS data, had limitations making it hard to generate sufficiently large graphs with the desired properties. The Boost Graph data sets proved to be a bigger challenge for the algorithm.

The experiments were done using three different forms of physical representation of the data, of which two are closely related. The first, baseline B-Tree approach is using a common form of data storage which – in slightly different forms – is also used in many commercial systems.

The Triple-T index is newer and not yet widely used. However it promises to be a decent candidate for RDFS data storage, with fast look-ups and efficient join processing [9]. An enhanced version of Triple-T – Triple-T with distance – which materializes a small part of the graph's entailment was created to reduce the amount of look-ups required to complex queries.

Several changes to the original Triple-T index were discussed and implemented, to attempt to improve its performance. These modifications are present in both the base Triple-T and Triple-T with distance versions.

In the experiments we compared both the amount of cache requests and misses as indicator for I/O performance. Additionally several timings were measured and compared. Finally, to make certain triple look-ups more efficient, the internal order of the object sub-index was compared using both `OSP` and `OPS` ordering in each of the used indexes.

7.1 Results

In general, Triple-T performs at least as good as the more common B-Tree index, while using less space. Often it requires fewer pages to be read from disk, making it more efficient than the B-Tree index.

Triple-T with distance has proven to be much more efficient in queries of transitive predicates. The amount of cache misses measured by Berkeley DB is significantly lower. Also in terms of time required to answer a query, Triple-T with distance mostly performs better than the other two tested indexes. For simple queries the performance was equal to the base version of Triple-T.

The main drawback is that updating data using transitive predicates is less trivial and therefore slower. Further research will need to be done to compare and optimize updates between these methods of physical representation.

The Triple-T modifications such as compression and splitting the indexes may have had a negative impact. Further research will need to be performed to give more conclusive results regarding these changes.

The comparison of `OSP` versus `OPS` ordering of the object buckets of each index gave no conclusive results. Both forms performed roughly the same.

7.2 Future Work

The research discussed in this thesis is only a small part of a much bigger picture involving RDF and RDFS. There is a lot of research that can still be done following this research. In this section we will discuss extensions to the Muñoz algorithm, applications of the algorithm as well as possible extensions to indexing RDFS data efficiently.

7.2.1 Triple-T Enhancements

The implementation of Triple-T introduced several changes over the original version. On paper these changes would not impact I/O efficiency, or even improve it. However, as the experiments have shown, this may not be the case due to the way hardware behaves.

Therefore, to get the best out of the Triple-T solution, more experiments can be performed. This includes varying several properties:

- Using a hash table index versus a B-tree index
- Combining all three buckets of a resource into one payload or not
- In the case of the split index, putting all resources in one big hash table, or three smaller ones
- Always using compression, no compression at all, or compression only when it reduces the required amount of pages to store the payload

Furthermore, additional experiments can be performed to compare materializing one or more steps of the transitive closure in Triple-T with distance. The update complexity will remain the same, but in practice they may be slower as more data needs to be inserted. Additionally the space occupied by the data will grow, but query performance is expected to be improved.

7.2.2 RDFS Rules

Obviously the studied ρdf is only a subset of the RDFS vocabulary. The full set of rules following the specification of the RDF Semantics [27] add various extra rules, deliberately left out of ρdf. By expanding ρdf with terms `rdfs:Resource`, `rdfs:Class` and `rdf:Property` from RDFS, various extra rules involving these terms can easily be added. See table B.1 in appendix B for the full set of rules for the RDF semantics. Appendix C shows the set of relevant definitions for the ρdf vocabulary, from [28].

Rule `rdfs6` in [27] states that if a graph contains $(a, \texttt{type}, \texttt{rdf:Property})$, then a is a subproperty of itself (i.e. $(a, \texttt{sp}, a)$ exists). Consider the graph with only this precise triple. This is not yet covered by the mrdfs rules. The same holds for classes, using rule `rdfs10`.

The rules also lack the notion that all resources are of type `rdfs:Resource`, as defined by the RDF Semantics rules `rdfs4a` and `rdfs4b`. Additionally, every class is also a subclass of `rdfs:Resource`, as defined by rule `rdfs8` [27].

As the corrections below will show, all these problems can be solved easily by doing a series of additional look-ups for specifically constructed triples. First though, to prevent duplication or infinite loops, it will help to show that some queries are in fact equal.

Proposition 5. *Given triples* $A = (a, \texttt{sc}, \texttt{rdfs:Resource})$, $B = (a, \texttt{type}, \texttt{rdfs:Class})$ *and* $C = (a, \texttt{sc}, a)$, *and graph* G, *the following statement always holds:* $A \in G \Leftrightarrow B \in G \Leftrightarrow C \in G$.

Proof. The proof is straightforward, using the RDF Semantics rules [27] and the RDF Schema vocabulary [28]. The part $A \in G \Rightarrow B \in G$ follows directly from the definition of the domain of `rdfs:subClassOf` (`sc`). The other way around, $A \in G \Rightarrow B \in G$, follows directly from rule `rdfs8`. This solves $A \in G \Leftrightarrow B \in G$.

For $B \in G \Rightarrow C \in G$, the definition of rule `rdfs10` directly proves this. Finally, $C \in G \Rightarrow B \in G$ is again a direct conclusion of the definition of the domain and range of `rdfs:subClassOf`. This solves $B \in G \Leftrightarrow C \in C$.

Combining these two conclusions, the correctness of the proposition is shown. □

Proposition 6. *Given triples* $A = (a, \texttt{type}, \texttt{rdf:Property})$ *and* $B = (a, \texttt{sp}, a)$, *and graph* G, *the following statement always holds:* $A \in G \Leftrightarrow B \in G$

Proof. Again, inspecting the rules and RDFS definitions suffices to complete the proof both ways. The first part, $A \in G \Rightarrow B \in G$, follows immediately from the definition of `rdfs6`. For $B \in G \Rightarrow A \in G$, the definition of the domain (and range) of `rdf:Property` immediately proves this. □

Because these equalities hold, it is easy to add support for queries of the form $(a, \texttt{type}, \texttt{rdf:Property}) \in G$, $(a, \texttt{sc}, \texttt{rdfs:Resource}) \in G$ and $(a, \texttt{type}, \texttt{rdfs:Class}) \in G$. These can simply be rewritten to their $(a, \texttt{sp}, a)$ and $(a, \texttt{sc}, a)$ counterparts respectively, such that steps 2b and 3b will answer the queries.

As these are already implemented using plain look-ups on various triples, the steps need to be expanded slightly to make support for `rdfs:Resource`, `rdfs:Class` and `rdf:Property` complete. The detection of classes is complete, except for the exact triples $(a, \texttt{type}, \texttt{rdfs:Class})$ and $(a, \texttt{sc}, \texttt{rdfs:Resource})$. Similarly, the only look-up that has to be added for the properties is $(a, \texttt{type}, \texttt{rdf:Property})$.

To complete support for rules `rdfs:Resource`, using `rdfs4a` and `4b`, whenever a query of the form $(u, \texttt{type}, \texttt{rdfs:Resource})$ is requested, the existence of triples $(u, ?, ?)$ or $(?, ?, u)$ must be checked. From using the base RDFS graph, these triples also exist for any u in the vocabulary of RDFS, as all those terms appear as subject or object in their defining syntax triples.

Finally, rules `rdfs12` and `rdfs13` can quite trivially be implemented by running the necessary look-ups during the algorithm. As they both use `sp` and `sc` respectively in their goals, these look-ups will also cause expansion of the algorithm for steps 2-6 (except for the reflexive steps 2b and 3b).

7.2.3 Blank Nodes and Cycles

Muñoz et al. prove that the algorithm also works with the extension that the derived graph H has triples with no more than one blank node [13]. In that case, determining whether a graph G entails H based on the ρdf fragment, can be done in $O(|H||G|\ log\ |G|)$.

The algorithm as discussed in this thesis focuses on those cases in which H is a single triple (and it was also optimized for this use). In that case, the complexity is the same as the original algorithm, $O(|G|\ log\ |G|)$.

The main changes in the actual algorithm as implemented will involve changing the order in which the `sp`and `sc`triples are traversed. The current version uses a specific order, but starting from a blank node is not practical. In that case, the traversal should start from the other end. The process is only a little more involved for steps 5 and 6 in the algorithm, as also shown by [13].

7.2.4 Efficient Updates

Following the findings and the working implementation of this thesis, the next step is to put the entailment algorithm to actual use. A sample of this is an update algorithm which can (deterministically) remove extra triples, such that another is no longer entailed by the graph.

While inserts are fairly trivial to perform, removing a triple is not always that easy. This is because the removed triple may still be inferrable from the updated graph.

Such an update algorithm is proposed by Chirkova and Fletcher [2]. They take the approach of using tuple-generating dependencies (*tgds*) to describe the RDFS rules. Using an unchase algorithm with these tgds, a list of triples can be constructed which are (potentially) used to infer a specific triple in the graph.

Due to the lack of negation in RDF, it is not possible to just add a negated triple to the graph and leave it at that. Therefore at least one of the triples which derive the to-be-deleted statement must be removed, such that the inference is broken and the statement is effectively removed.

A more stricter form of this is when there is essentially just one (base) triple which causes the inference to occur. This deterministic form allows for automatic deletion of triples without having to make any choices (either by human intervention or automatically).

Near the end of the paper, algorithm 6.1 is a proposal of how to perform this deterministic deletion. The algorithm directly checks whether a graph still entails a triple, in step 3. That step can now be implemented directly using the algorithm discussed in this thesis; what remains is the other "key step" in the algorithm: computing the unchase of the triple in the graph. Especially for computing a deterministic unchase, this part of research intuitively overlaps that of logical programming. Expertise and knowledge in that area may help in constructing an overall efficient update algorithm.

7.2.5 Better Transitive Indexing

As shown by the extended Triple-T index, being able to skip over some queries can greatly reduce the amount of look-ups that have to be performed in an RDFS graph. This leads to lower I/O costs, which in turn implies better performance and scalability. This is mostly because the costly part of the algorithm involves finding paths between two atoms over a transitive predicate. More specifically, it is about checking if a path between two items exists; it is not required to know the exact path.

A new method of indexing transitive predicates may therefore give a significant performance boost. To achieve this, ordering the atoms in specific ways can introduce options to being able to more quickly determine whether a path exists or not. For example, consider an ordering $O_R(X)$ which defines the weight of atom X in transitive relation R. If for each triple (a, R, b) the property $O_R(a) < O_R(b)$ is maintained, simply checking the positions of a and b can already limit the amount of steps that need to be taken during the traversal. After all there is no need to traverse beyond triples (x, R, y) for which $O_R(y) >= O_R(b)$. However, maintaining such an index properly during updates is far from trivial.

Additionally sub-trees can be introduced, placing each atom in a sub-tree. By maintaining extra lists of relations between these sub-trees, an algorithm may be able to more quickly discard parts to search through. Consider an algorithm trying to check if a path using relation R exists between a and b (i.e. a query for triple (a, R, b)). Denote $T_R(a)$ as the set of sub-trees reachable from a through R. If this set contains the sub-tree in which b resides (denoted by $t(b)$), other trees in the set $T_R(a)$ can be discarded or delayed for further searching. A valid path from the entry points in $t(b)$ (atoms reachable from a) to b can then be found locally within the sub-tree. If this is found, (a, R, b) will also exist.

Although the entire path will still be searched, the sub-trees may hint in the right direction. As such, a valid path may be found sooner, although this is not guaranteed.

If $t(b) \notin T_R(a)$ then it is first necessary to check if there are paths from any $x \in T_R(a)$ to $t(b)$. As the amount of trees is smaller than the total amount of statements in the graph, this question is most likely quicker to answer.

Again, maintaining these subtrees and covering all subtleties involved in this approach is not trivial. Additionally, since there is no restriction of cycles of the `rdfs:subPropertyOf` and `rdfs:subClassOf` trees, there may be additional complexities to beware of. In fact, a restriction on cycles on `rdfs:subPropertyOf` was explicitly removed from the recommendation in 2001 (Action 2001-09-21#5 [22]).

Appendix A

PoweRGen Configuration

The base configuration to generate the smaller graphs of the PoweRGen data sets, is provided in a `parameters.conf` file. The contents of this file as it was used, excluding the library and output paths, is presented in listing A.1.

```
numOfSchemas = 1
format = XML
solver  = mosek
validate  = false
plot  = false
taxonomy = true
numOfClasses = 1000
numOfProps = 1000
totalDegreeVRExp = 0.7
p0 = 0.25
selfLoops  = 0.126
multipleArcs = 0.177
attributes  = 0.05
descendantsPDFExp = 1.8
percentageOfZeros = 0.25
depth = 10
integralityThreshold  = 0.5
depthPercentage = 0.6
tree  = false
ACCThreshold = 0.6
CRE = 0.6
```

Listing A.1: PoweRGen parameters.conf file

Appendix B

RDF Semantics

The rules in table B.1 are taken from the RDFS entailment rules in the RDF Semantics W3C Recommendation [27]. The triples have been rewritten to be more in line with the rest of this thesis. See the W3C recommendation document for the full set of rules. The complete recommendation also lists several other sets of semantic rules which may not always apply.

Rule Name	**If graph G contains**	**then add:**
`lg`	(u, a, l)	$(u, a, \_{:}n)$ Where $\_ : n$ identifies a blank node allocated to literal l by this rule.
`rdfs1`	(u, a, l) With l a plain literal	$(\_{:}n, \texttt{type}, \texttt{rdfs:Literal})$ Where $\_ : n$ identifies a blank node allocated to l by rule `lg`.
`rdfs2`	$(a, \texttt{dom}, tlitx)$, (u, a, y)	$(u, \texttt{type}, x)$
`rdfs3`	$(a, \texttt{range}, x)$, (u, a, v)	$(v, \texttt{type}, x)$
`rdfs4a`	(u, a, v)	$(u, \texttt{type}, \texttt{rdfs:Resource})$
`rdfs4a`	(u, a, v)	$(v, \texttt{type}, \texttt{rdfs:Resource})$
`rdfs5`	$(u, \texttt{sp}, v)$, $(v, \texttt{sp}, x)$	$(u, \texttt{sp}, x)$
`rdfs6`	$(u, \texttt{type}, \texttt{prop})$	$(u, \texttt{sp}, u)$
`rdfs7`	$(a, \texttt{sp}, b)$, (u, a, y)	(u, b, y)
`rdfs8`	$(u, \texttt{type}, \texttt{class})$	$(u, \texttt{sc}, \texttt{rdfs:Resource})$
`rdfs9`	$(u, \texttt{sc}, x)$, $(v, \texttt{type}, u)$	$(v, \texttt{type}, x)$
`rdfs10`	$(u, \texttt{type}, \texttt{class})$	$(u, \texttt{sc}, u)$
`rdfs11`	$(u, \texttt{sc}, v)$, $(v, \texttt{sc}, x)$	$(u, \texttt{sc}, x)$
`rdfs12`	$(u, \texttt{type}, \texttt{rdfs:ContainerMembershipProperty})$	$(u, \texttt{sp}, \texttt{rdfs:member})$
`rdfs13`	$(u, \texttt{type}, \texttt{rdfs:Datatype})$	$(u, \texttt{sc}, \texttt{rdfs:Literal})$

Table B.1: RDFS Entailment Rules

Appendix C

ρdf Vocabulary Triples

The following triples are the triples that define the ρdf vocabulary, extended with `rdfs:Resource`, `rdfs:Class`and `rdf:Property`. These are derived from the RDFS Vocabulary [28]. The labels, comments, `rdfs:isDefinedBy`, etc. elements are omitted here – this list only mentions the type of each term, for properties the domain and range, and for classes any class they are a subclass of. For a complete definition of RDFS see [28].

rdfs:Resource

- (`rdfs:Resource`, `rdf:type`, `rdfs:Class`)

rdf:type

- (`rdf:type`, `rdf:type`, `rdf:Property`)
- (`rdf:type`, `rdfs:domain`, `rdfs:Resource`)
- (`rdf:type`, `rdfs:range`, `rdfs:Class`)

rdfs:Class

- (`rdfs:Class`, `rdf:type`, `rdfs:Class`)
- (`rdfs:Class`, `rdfs:subClassOf`, `rdfs:Resource`)

rdfs:subClassOf

- (`rdfs:subClassOf`, `rdf:type`, `rdf:Property`)
- (`rdfs:subClassOf`, `rdfs:domain`, `rdfs:Class`)
- (`rdfs:subClassOf`, `rdfs:range`, `rdfs:Class`)

rdfs:subPropertyOf

- (`rdfs:subPropertyOf`, `rdf:type`, `rdf:Property`)
- (`rdfs:subPropertyOf`, `rdfs:domain`, `rdf:Property`)
- (`rdfs:subPropertyOf`, `rdfs:range`, `rdf:Property`)

rdf:Property

- (`rdf:Property`, `rdf:type`, `rdfs:Class`)
- (`rdf:Property`, `rdfs:subClassOf`, `rdfs:Resource`)

rdfs:domain

- (`rdfs:domain`, `rdf:type`, `rdf:Property`)
- (`rdfs:domain`, `rdfs:domain`, `rdf:Property`)
- (`rdfs:domain`, `rdfs:range`, `rdfs:Class`)

rdfs:range

- (`rdfs:range`, `rdf:type`, `rdf:Property`)
- (`rdfs:range`, `rdfs:domain`, `rdf:Property`)
- (`rdfs:range`, `rdfs:range`, `rdfs:Class`)

Appendix D

CLI Interface

The following is a list of commands supported by the developed application. The application supports two tables: `original` and `update`. The latter was used to store updated graphs in, but this is not actively used by default. The default table for all commands that support it is `original`.

help
: Usage: `help [COMMAND]`
Get a list of all available commands, or more detailed usage information of a specific command.

connect
: Usage: `connect URL`
Connect to a storage back-end. `URL` specifies which database to connect to. Its scheme is the storage type, followed by (optionally) a username, password, hostname and port number, and a path which represents the database name. Additional options can also be added in a familiar method. Some examples:

  - mysql://username:password@localhost:3306/database
  - sqlite:///path/to/file
  - virtuoso:///path/to/location
  - bdb:///path/to/directory/
  - bdbbt:///path/to/directory/?cache=1m&pagesize=4k
  - bdbtt:///path/to/directory/?cache=1m&pagesize=4k

  Supported back-ends at the time of writing include MySQL, SQLite, Virtuoso (through Soprano), a basic Berkeley DB back-end (bdb), Berkeley DB B-Tree and Berkeley DB Triple-T.

disconnect
: Usage: `disconnect`
Close the open back-end or back-end connection.

listOptions
: Usage: `listOptions`
List the keys of options available for the currently connected back-end.

setOption
: Usage: `setOption KEY VALUE`
Set an option for the currently connected back-end.
This is used to configure the Triple-T back-end distance, e.g. using `setOption distance 1`.

srand

Usage: `srand SEED`
Seed the random number generator. `SEED` is the seed to seed with (unsigned integer). If omitted, the current system time is used.

load

Usage: `load FILE [SEPARATOR [TABLE]]`
Load a formatted file with triples into the database. The file indicated by `FILE` must contain one triple per line, in which each resource is separated by `SEPARATOR` in subject, predicate, object order. `TABLE` is either `original` or `update`, which represents which table to work with.

export

Usage: `export FILE [SEPARATOR [TABLE]]`
Export the triples in a table to a file. The file will contain one triple per line, with resources separated by `SEPARATOR`. `TABLE` is either `original` or `update`, which represents which table to work with.

insert

Usage: `insert SUBJECT PREDICATE OBJECT [TABLE]`
Insert a triple into the database. The triple is defined by its subject, predicate and object. It is inserted in the table indicated with `TABLE` (`original` or `update`).

remove

Usage: `insert SUBJECT PREDICATE OBJECT [TABLE]`
Remove all matching triples from a table in the database. The triple is defined by its subject, predicate and object. Use _ to denote a blank. `TABLE` is either `original` or `update`, which represents which table to work with.

truncate

Usage: `truncate [TABLE]`
Truncate the table indicated by TABLE. Special table `rules` removes all custom rule sets.

exists

Usage: `exists SUBJECT PREDICATE OBJECT [TABLE]`
Check if a triple exists in a table. `SUBJECT`, `PREDICATE` and `OBJECT` represent the resources of the triple to check. A _ can be used to denote a blank. `TABLE` is either `original` or `update`, which represents which table to work with.

list

Usage: `list SUBJECT PREDICATE OBJECT [TABLE]`
List all triples matching the request in a table. `SUBJECT`, `PREDICATE` and `OBJECT` represent the resources of the triple to check. A _ can be used to denote a blank. `TABLE` is either `original` or `update`, which represents which table to work with.

chainstats

Usage: `list PREDICATE [TABLE]`
Get the statistics of a transitive predicate, like `sc` or `sp` in the current graph. `TABLE` is either `original` or `update`, which represents which table to work with.

listRules

Usage: `listRules`
List all defined deterministic tuple-generating dependencies. Use `addRule` and `removeRule` commands to modify these.

addRule

Usage: `addRule SUBJECT1 PREDICATE1 OBJECT1 SUBJECT2 PREDICATE2 OBJECT2`

Add a new deterministic tuple generating dependency used in `unchase`. All variables in the right-hand side triple must also occur on the left-hand side. A variable is denoted by using the syntax `@n` with `n` a posttive integer number. For example, a symmetrical predicate `R` rule can be defined as follows: `addRule @1 R @2 @2 R @1`. This implies that if a triple (a, R, b) is found, (b, R, a) will considered to exist as well (in `unchase` only).

removeRule

Usage: `removeRule SUBJECT1 PREDICATE1 OBJECT1 SUBJECT2 PREDICATE2 OBJECT2`
Remove a rule added by `addRule`.

closurecheck

Usage: `closurecheck SUBJECT PREDICATE OBJECT [TABLE]`
Check if a triple exists in the closure of the graph in a table. Implementation of algorithm by Muñoz et al. Custom rules are ignored. Use the "unchase" command to check if there is an unchase of the triple. If so, it is also in the closure.

unchase

Usage: `unchase SUBJECT PREDICATE OBJECT [TABLE]`
Get the set of triples in the unchase of a given triple. Each returned triple is checked to be in the entailment of the graph.

stats

Usage: `stats`
Get current cache statistics from the back-end.

dotexport

Usage: `dotexport PREDICATE FILE`
Export a predicate graph to a `dot` file. Only table `original` is supported.

graphgen

Usage: `graphgen PREDICATE PREFIX TYPE N [ALPHA [BETA]]`
Generate subclass or subproperty graphs using power-law properties. `PREDICATE` is the predicate to use as edge. `PREFIX` is the naming prefix for vertices. TYPE is the type to assign to the vertices. `N` is the amount of vertices to place in graph. `ALPHA` is the value controlling how steeply the curve drops off. A larger value indicates a steeper curve. `BETA` is the value controlling the average degree of vertices.

See `http://www.boost.org/doc/libs/1_41_0/libs/graph/doc/plod_generator.html` for information about the parameters.

killcycles

Usage: `killcycles PREDICATE [TABLE]`
Make sure a graph of a given predicate contains no cycles. Cycles will be broken by removing a statement. `TABLE` is either `original` or `update`, which represents which table to work with.

ontologylinks

Usage: `ontologylinks [PROPERTY [CLASSTYPE [ZERODEGREEPCT [SELFLOOPPCT]]]]`
Create links between properties and classes using domain/range triples. `PROPERTY` is The type of properties. All properties should have an existing $(name, \texttt{type}, \texttt{PROPERTY})$ statement in the database. `CLASSTYPE` is the type of classes. All classes should have an existing $(name, \texttt{type}, \texttt{CLASSTYPE})$ statement in the database `ZERODEGREEPCT` is the fraction of classes (0-1) which should not be assigned as domain or range of properties at all. `SELFLOOPPCT` is the fraction of properties that should construct a self-loop, i.e. an equal domain and range.

dotontolgy

Usage: `dotontolgy FILE [PROPTYPE [TABLE]]`
Export an ontology – links between classes using properties – to DOT format. `FILE` is the file

to write to, `PROPTYPE` indicates which type is used for properties. `TABLE` is either `original` or `update`, which represents which table to work with.

FindQueries

Usage: `FindQueries CLASS N VALID [FILE [TRUNCATE]]`
Find interesting queries of a specific type. `CLASS` is the type of queries to generate:

1 A simple lookup: (*PROPERTY*, `dom`, *CLASS*).

2 A single traversal step: (*ATOM1*, *PROPERTY*, *ATOM2*).

3 A long query: (*ATOM*, `type`, *CLASS*).

4 A long traversal (*ATOM1*, *PROPERTY*, *ATOM2*).

`N` indicates the total amount of queries to generate, of which `VALID` will be existing triples. Queries will be written to `FILE` (use `-` for standard output). If `TRUNCATE` is 1, the file will be truncated first.

callgrind

Usage: `callgrind`
Start or stop callgrind instrumentation.

quit

Usage: `quit`
Close the application.

Bibliography

[1] Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors. *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*, volume 5823 of *Lecture Notes in Computer Science*. Springer, 2009.

[2] Rada Chirkova and George Fletcher. Towards well-behaved schema evolution. In *WebDB*, 2009.

[3] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

[4] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11:121–137, June 1979.

[5] Richard Cyganiak and Anja Jentzsch. The linking open data cloud diagram. `http://richard.cyganiak.de/2007/10/lod/`, September 2010. [Online; accessed 26 May 2011].

[6] Jos de Bruijn and Stijn Heymans. Logical foundations of (e)rdf(s): Complexity and reasoning. In *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 86–99. Springer Berlin / Heidelberg, 2007.

[7] Andrew Lumsdaine Doug Gregor. Boost graph library, plod_iterator documentation. `http://www.boost.org/doc/libs/1_46_1/libs/graph/doc/plod_generator.html`, 2005. [Online; accessed 20 April 2011].

[8] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4:315–344, September 1979.

[9] George H. L. Fletcher and Peter W. Beck. Scalable indexing of rdf graphs for efficient join processing. In David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy J. Lin, editors, *CIKM*, pages 1513–1516. ACM, 2009.

[10] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 1st edition, 2011.

[11] Giovambattista Ianni, Thomas Krennwallner, Alessandra Martello, and Axel Polleres. Dynamic querying of mass-storage rdf data with rule-based entailment regimes. In Bernstein et al. [1], pages 310–327.

[12] Per-Ake Larson. Dynamic hash tables. *Commun. ACM*, 31:446–457, April 1988.

[13] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Simple and efficient minimal rdfs. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):220 – 234, 2009. The Web of Data.

[14] Thomas Neumann and Gerhard Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.

[15] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. Marvin: Distributed reasoning over large-scale semantic web data. *J. Web Sem.*, 7(4):305–316, 2009.

[16] Orri Erling (OpenLink Software, USA). Towards web scale rdf. In *4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008)*, October 2008.

[17] Greg Roelofs and Jean-loup Gailly. zlib technical details. `http://www.zlib.net/zlib_tech.html`, May 2006. [Online; accessed 13 April 2011].

[18] Sherif Sakr and Ghazi Al-Naymat. Relational processing of rdf queries: a survey. *SIGMOD Record*, 38(4):23–28, 2009.

[19] Michael Schmidt, Thomas Hornung, Norbert Küchlin, Georg Lausen, and Christoph Pinkel. An experimental comparison of rdf data management approaches in a sparql benchmark scenario. In Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan, editors, *International Semantic Web Conference*, volume 5318 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2008.

[20] SemanticWeb.org. Semantic web challenge. `http://challenge.semanticweb.org/`, 2010. [Online; Accessed 13 April 2011].

[21] Lefteris Sidirourgos, Romulo Goncalves, Martin L. Kersten, Niels Nes, and Stefan Manegold. Column-store support for rdf data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.

[22] Aaron Swarts. Rdfcore wg telecon 2001-09-21 minutes. `http://lists.w3.org/Archives/Public/w3c-rdfcore-wg/2001Sep/0326.html`, September 2001. [Online; accessed 5 May 2011.

[23] Y. Theoharis. On power laws and the semantic web. Master's thesis, University of Crete, February 2007.

[24] Yannis Theoharis, Yannis Tzitzikas, Dimitris Kotzinos, and Vassilis Christophides. On graph features of semantic web schemas. *IEEE Trans. Knowl. Data Eng.*, 20(5):692–702, 2008.

[25] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Scalable distributed reasoning using mapreduce. In Bernstein et al. [1], pages 634–649.

[26] W3C. Rdf primer. `http://www.w3.org/TR/rdf-primer/`, February 2004. [Online; Accessed 25 May 2011].

[27] W3C. Rdf semantics. `http://www.w3.org/TR/rdf-mt/`, February 2004. [Online; Accessed 2 May 2011].

[28] W3C. Rdf vocabulary description language 1.0: Rdf schema. `http://www.w3.org/TR/rdf-schema/`, February 2004. [Online; Accessed 24 January 2011].

[29] Gregory Todd Williams, Jesse Weaver, Medha Atre, and James A. Hendler. Scalable reduction of large datasets to interesting subsets. *J. Web Sem.*, 8(4):365–373, 2010.