UNIVERSITY OF SOUTHAMPTON

Using Low Latency Storage to Improve RDF Store Performance

by

Alisdair Owens

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

April 2011

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Alisdair Owens

Resource Description Framework (RDF) is a flexible, increasingly popular data model that allows for simple representation of arbitrarily structured information. This flexibility allows it to act as an effective underlying data model for the growing Semantic Web. Unfortunately, it remains a challenge to store and query RDF data in a performant manner, with existing stores struggling to meet the needs of demanding applications: particularly low latency, human-interactive systems. This is a result of fundamental properties of RDF data: RDF's small statement size tends to engender large joins with a lot of random I/O, and its limited structure impedes the generation of compact, relevant statistics for query optimisation.

This thesis posits that the problem of performant RDF storage can be effectively mitigated using in-memory storage, thanks to RAM's extremely high throughput and rapid random I/O relative to disk. RAM is rapidly reducing in cost, and is finally reaching the stage where it is becoming a practical medium for the storage of substantial databases, particularly given the relatively small size at which RDF datasets become challenging for disk-backed systems.

In-memory storage brings with it its own challenges. The relatively high cost of RAM necessitates a very compact representation, and the changing relationship between memory and CPU (particularly increasing RAM access latency) benefits designs that are aware of that relationship. This thesis presents an investigation into creating CPU-friendly data structures, along with a deep study of the common characteristics of popular RDF datasets. Together, these are used to inform the creation of a new data structure called the Adaptive Hierarchical RDF Index (AHRI), an in-memory, RDF-specific structure that outperforms traditional storage mechanisms in nearly every respect.

AHRI is validated with a comprehensive evaluation against other commonly used in-memory data structures, along with a real world test against a memory-backed store, and a fast disk-based store allowed to cache its data in RAM. The results show that AHRI outperforms these systems with regards to both space consumption and read-/write behaviour. The document subsequently describes future work that should provide substantial further improvements, making the use of RAM for RDF storage even more compelling.

Contents

List of Figures

List of Tables

Listings

Acknowledgements

With many thanks to my advisors, m.c. schraefel and Nick Gibbins for giving me the benefit of their experience, and so readily sharing their knowledge. I am indebted also to Andy Seaborne for consistently offering his help and advice, and to Talis for kindly providing the server used to perform the larger-scale tests. Thanks to my family for all of their support, and especially to my wife Clare, whose capacity to provide help and encouragement never cease to amaze me.

Chapter 1

Introduction

Resource Description Framework (RDF) (Lassila et al., 1999) is a means for expressing knowledge in a generic manner, without requirement for adherence to a tightly specified schema. It is designed to provide a flexible means to support simple data aggregation, discovery, and interchange, and has already found use as an underlying data format in such fields as e-science (Taylor et al., 2005, 2006) and faceted browsing (Smith et al., 2007; schraefel et al., 2004). The goal of researchers in the area is that as technologies mature, the Semantic Web will be built upon linked RDF data (Berners-Lee et al., 2001).

Making real use out of information encoded in the RDF format requires the ability to process that information: most particularly to be able to ask questions, or queries, about the data. Indeed, the ability to efficiently ask questions about RDF data is a fundamental requirement of the Semantic Web. Small scraps of information can be stored in flat files, but as datasets scale up, they generally require a structured storage system that allows them to be queried against in a performant manner. Systems that manage RDF data n this manner are known as RDF or Triple stores.

There is already a considerable body of work dedicated to information storage and retrieval: the Database Management System (DBMS) community has been working in this area for many years, and a great deal of progress has been made - an overview of which can be found in Date (1990). High performance RDF storage depends to a significant extent on correct application of existing DBMS research, and so these areas of research run in parallel: indeed, many RDF stores are built as layers that rely on existing relational DBMSs (RDBMSs) to do much of the work. Unfortunately, the unpredictable structure of RDF graphs has proved challenging for traditional DBMSs, and they have not been able to scale effectively to large quantities of RDF data. The reasons for this are explored in the following section.

1.1 Problem Statement

The performance of Relational DBMSs (RDBMSs) has resulted in users becoming accustomed to being able to perform complex, meaningful queries over very large amounts of information. Unfortunately, RDF datasets have proven challenging to work with in comparison to traditional relational ones, resulting in RDF stores being both slow and lacking in scalability by the standards of modern DBMSs[1]. The most powerful single machine RDF stores are capable of storing around two billion RDF statements, or in the order of tens or a few hundred gigabytes of data (Erling and Mikhailov, 2009), and pattern-match queries performed over much smaller datasets can produce unacceptable performance (Smith et al., 2007). Commercial RDBMS products, on the other hand, are capable of storing terabytes of data whilst preserving real time query performance.

The performance issues of RDF stores can be categorised as scale and query latency related. Some progress has been made on scaling, with improved single machine systems and the emergence of distributed stores (Harth et al., 2007; Erling and Mikhailov, 2008), but relatively little work has examined the creation of stores that offer low query latencies. This places limitations upon software that is designed for human interaction, in particular the new wave of rich web applications that rely on regular contact with a backing data store (Smith et al., 2007).

1.2 Motivation, Aims, and Approach

While RDF stores have been in development since shortly after the RDF data model was created, none have fully satisfied the challenges posed by the RDF graph's limited structure. Poor RDF store performance is a problem for a variety of reasons. Most importantly, it's a limiting issue for the wider Semantic Web: without the ability to manipulate and extract information from large data stores in a performant manner, it will be impossible to develop the sort of killer features that popularised the Web. In particular, it will be difficult to develop high quality human-interactive applications: user perception of application performance has a strong impact on satisfaction (Palmer, 2003).

Further, much of the current web is based on providing information for free (or supported by advertising) - and this is a fundamental driver of its popularity. This is only possible because the cost of providing information is very low. In order for the Semantic Web to grow, it must be cheap and feasible to work with Semantic Web information, or individuals and organisations simply won't provide their data in Semantic Web formats

[1] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/

Finally, the performance of RDF stores is a key issue for those who wish to work with data of limited or free schema, even outside the context of a Semantic Web. Many e-science projects make use of RDF stores as a result of their flexibility. Indeed, these projects have produced many of the largest existing RDF datasets currently in existence (Belleau et al., 2008; Jain et al., 2009).

The wider area of DBMSs has been researched for decades, and is one of the most important and active fields under the umbrella of Computer Science. RDF exhibits some features that make it difficult to model using traditional database systems: the structure of an RDF document is highly unpredictable, and does not lend itself to storage in any but the most generic of schemas. This unpredictability is also evident in query patterns: unlike more conventional relational systems, support for performant arbitrary queries is expected on RDF stores. Finally, RDF's datums (triples) are very small. It thus exhibits an unusually large number of individual data points compared to the amount of information encoded, meaning that when processing a comparable amount of information, each operation generally has more datums to process.

Typically, each of these issues inhibits efficient storage and query optimisation, making even advanced stores relatively slow - and it's a difficult problem to fix. Some partial solutions exist:

- **Extreme read-orientation**. Many stores take this approach (Harth et al., 2007; Weiss and Bernstein, 2009; Neumann and Weikum, 2008), which certainly improves the performance of queries. Unfortunately, some systems are completely non-updateable, while others are so slow to update that using them in an environment with writes would be impractical. This kind of optimisation, while effective, neuters one of RDF's key advantages: that it is a format for describing data where one can add information in a free manner.
- **Distribution**. This tends to allow scaling to larger datasets, as well as larger numbers of users (Harth et al., 2007; Harris et al.; Erling and Mikhailov, 2008). It has not, however, been shown to substantially improve the performance of individual queries.
- **Storing in memory**. Low latency storage methods such as RAM are a promising avenue for RDF storage, particularly as they become more affordable. Current main-memory systems are not heavily developed, and can consume excessive amounts of RAM, as will be shown in Chapter 4.

1.3 Hypothesis

This document considers a variety of options for improving the performance of RDF stores, but focusses on the application of low latency storage systems to improve performance. While storing data in-memory provides an obvious immediate improvement when compared to legacy media such as hard disks, carefully tuning data structures for such storage mediums can yield substantial additional benefits.

This thesis posits that RDF-specific memory-based structures can substantially outperform alternative methods, and that existing in-memory structures can be improved upon substantially. The veracity of this assertion is determined through the creation of an alternative physical storage schema for RDF data, based on a detailed analysis of both RDF datasets, and the properties of modern CPU and RAM subsystems. This structure (called the Adaptive Hierarchical RDF Index, or AHRI) uses much less memory than existing solutions while:

- Offering excellent insert performance.
- Remaining open at all times to fast updates.
- Offering excellent query performance.
- Offering detailed statistics to improve RDF query.

This thesis shows that existing work would benefit from a stronger understanding of the structure of common RDF datasets, since prior to this document little work had been performed on producing human-readable statistics over RDF. Further, thanks to the limited penetration of memory-backed stores, the behaviour of modern computers beyond the overwhelming latency imposed by disk access has not be considered in sufficient detail in existing literature. This work has important implications for RDF storage as a whole, as RDF can experience an unusual degree of benefit from analysing the behaviour of modern CPUs and memory subsystems.

1.4 Contributions

The body of work described in this thesis has yielded a variety of contributions, produced to the end of creating AHRI. These six key contributions are detailed below.

1. An investigation into the behaviour of modern computer architectures, and how the features of these architectures can be utilised to improve the performance of RDF data stores.

2. A deep review of the structure of RDF datasets, including a tool to produce detailed statistics over RDF data. These statistics provide insight into common factors in RDF documents that can be used to inform the development of future stores.

3. An application of the knowledge gained in (1) and (2): a high performance, compact new data structure for storing RDF information. AHRI is suitable for general purpose RDF storage, and particularly for systems that rely on low latency storage such as UIs or caches.

4. A detailed validation of AHRI against a variety of existing data structures that are used for RDF data storage. This evaluation confirms that the performance of AHRI makes a substantial difference even in the context of all the other latencies associated with working with RDF data. The evaluation also provides a framework for testing alternative index types in a standardised manner.

5. AJP: a prototype query answering system for AHRI, in the form of a Jena plugin. While currently in a relatively basic state, this plugin can be extended to provide a fully featured RDF store.

6. A review of the suitability of Java (and the JVM in particular) for creating DBMSs, concluding that the language is suitable for such work, with the caveat that attention must be paid to the memory consumption of small objects.

1.5 Overview of Thesis

The document is structured as follows:

- Chapter 2 presents a basic background description of the core technologies of the Semantic Web, with particular reference to RDF, defining terms and situating the importance of developing RDF storage systems. It goes on to compare RDF to models used in existing database systems, with particular reference to relational systems, and describes why RDF is significantly different.

- Chapter 3 goes on to provide detailed related work, describing five key areas: the characteristics of modern computer architecture; how data is physically represented in storage in modern DBMSs; the importance of indexing to modern RDF storage; the importance of operator implementation specifics (for example, join type); and the use of distribution to scale RDF stores.

- Chapter 4 investigates the properties of modern Java Virtual Machines (JVMs), and what impact these properties have upon RDF storage. This work is performed because the most commonly-used, mature Semantic Web frameworks use Java

technologies and JVM behaviour can have a substantial impact upon many modern RDF stores.

- Chapter 5 performs a detailed study of RDF datasets. While RDF imposes very little structure on documents, this chapter uncovers the structural features that are featured in most common datasets, and uses these as a means to inform the design of structures for storing RDF data.
- Chapter 6 presents AHRI, the RDF data structure that resulted from the investigations in previous chapters. It is designed to perform well on modern computer architectures, and execute effectively on virtual machines such as that used by Java.
- Chapter 7 describes in-depth tests upon AHRI. These tests show that AHRI uses less space than alternative structures, while providing superior insert, update, and query performance. It presents an implementation of AHRI into a real triple store, showing that AHRI provides a significant speedup, particularly on difficult queries.
- Finally, Chapter 8 offers conclusions and future work.

Additionally, this thesis has four appendices:

- Appendix A provides small code samples from some of the work described in Section 4.1.
- Appendix B describes the test machine used for many of the experiments performed in this thesis.
- Appendix C presents statistics on additional RDF documents to support the work performed in Chapter 5.
- Appendix D provides some raw evaluation data from the experiments performed in Chapter 7.

1.6 Declaration

I, Alisdair Owens, declare that this thesis and the work presented within are both my own, and have been generated by me as the result of my own original research. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.

- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.
- None of this work has been published before submission.

Signed:
Date:

Chapter 2

Background and Research Motivation

This chapter describes the Semantic Web and several of its core technologies. It presents the case for supporting the development of RDF stores in the context of the Semantic Web's requirement for high performance data storage and retrieval.

2.1 The Importance of the Semantic Web

The Semantic Web (Berners-Lee et al., 2001) is a large-scale effort to bring machine-processable data to the World Wide Web. This is intended to allow machines to be able to understand and easily navigate a web of data. Mechanisms for shared understanding enable machines to communicate with each other, even in situations where they were not expressly designed to do so. The advantages that can be found in this endeavour are extraordinary: in particular, the long-awaited potential of software agents could be realised (Hendler, 2001). Consider the following example:

Having decided to become healthier, Alisdair is undertaking a new fitness regime at the gym. As well as regular exercise, Alisdair's trainer has recommended a more healthy diet plan to him. As a member of the gym, he has complimentary access to a large selection of recipes. Since he feels like trying something new, he asks his agent (accessed through a smartphone) to pick one for him. The agent, knowing the foods that he particularly likes and dislikes, works on finding him a recipe. It can do this because metadata on the recipes is held in an RDF store. This allows the agent to query for recipes that use ingredients or cooking methods that he might particularly enjoy. It then presents the best option to him for confirmation, along with a note that he will need to buy more ingredients to be able to cook it. It sounds good, so he accepts, and asks the agent to tell him where he can get the items he needs from. The agent, knowing that the weather is

good and that Alisdair likes to walk, looks for shops in the immediate area, and suggests two in close proximity that between them should stock everything that he needs.

This example shows a variety of benefits, in the elimination of a great deal of drudgery from Alisdair's life. Of course, if he wants to perform any tasks, such as picking the recipe himself, he can, but if he chooses he can have large parts of his life automated for him. This example is enabled by the intersection of two concepts: Intelligent Agents and the Semantic Web. The agent learns about Alisdair's preferences, and understands certain concepts such as food, recipe, shop, and weather. Other services on the internet also understand some of these concepts: the gym's agent understands recipes, while the BBC's agent might understand weather (as well as the date and time that Alisdair wants to know the weather for). The shops' agents understand various kinds of food and whether something is in stock. Alisdair's agent is able to communicate through these shared understandings to bring about the scenario described above.

Of course, the agents are the things that understand the concepts. However, the process of sharing a vocabulary such that agents can communicate about concepts they understand, and the mechanism for publishing that data, are brought about through the Semantic Web. The Semantic Web has innumerable other uses: researchers on the Semantic Grid (Taylor et al., 2005) are using it to advertise the availability of computing resources. E-Science researchers (Taylor et al., 2006) are using Semantic Web languages to exchange and aggregate data. There are Semantic Web browsers such as Tabulator (Berners-Lee et al., 2006) that offer individuals the ability to browse Semantic Web data for themselves. Faceted browsers like mSpace (schraefel et al., 2005) use Semantic Web data to provide a rich browsing experience, releasing information that would have had to be painstakingly manually collated previously. These are just a subset of the current uses of the Semantic Web, and the potential uses of the future are limited only by the imagination - and the capability of the backing technologies to support them.

The development of Semantic Web languages is proceeding apace: of the Semantic Web layer cake, as seen in Figure 2.1, RDF (Lassila et al., 1999), RDFS (Brickley et al., 2004), OWL (Patel-Schneider et al., 2004), and SPARQL (SPARQL Protocol and RDF Query Language) (Prud'hommeaux and Seaborne, 2006) have reached a stable state. A simplistic explanation of these is that RDF provides the ability to express data, SPARQL provides a mechanism for querying this data, while RDFS and OWL add to the ability to share concepts (for example, providing mappings from one concept to another), as well as infer new data from that already present.

2.2 Data Representation

RDF is, as previously mentioned, the underpinning language for data expression in the Semantic Web (Lassila et al., 1999). It is expressed in the simple manner of a triple,

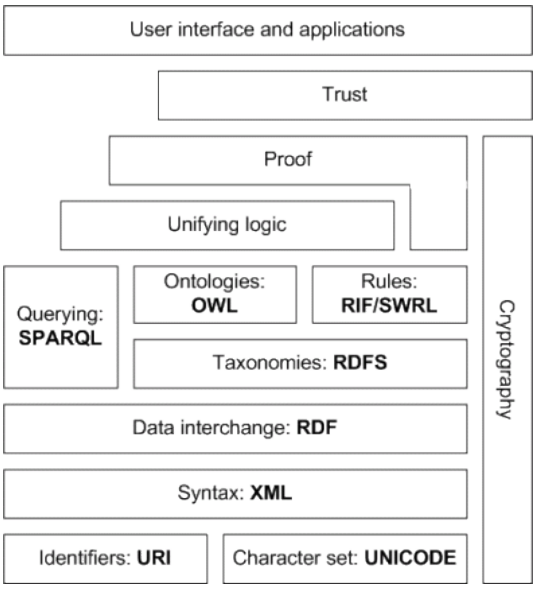


FIGURE 2.1: The Semantic Web layer cake (May 2008).

composed of subject, predicate, and object (S,P, and O respectively). This is roughly analogous to the subject verb and object of a simple sentence (Berners-Lee et al., 2001): for example:

Subject: Alisdair
Predicate: Has Gender
Object: Male

This is expressed visually in Figure 2.2.

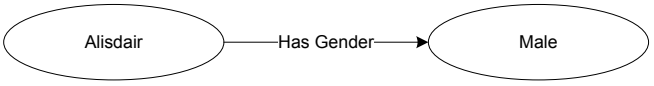


FIGURE 2.2: Triple concept

RDF triples are built out of Uniform Resource Identifiers (URIs) (Berners-Lee et al., 1998) and literals. A URI is a unique identifier that denotes a concept: for example, the URI for a dog might be *http://www.example.com/animals/dog*. A literal is simply a string, such as 'Alisdair Owens', with optional additions denoting language (such as English or French) and datatype (any supported by XML, such as int and datetime). Ideally, a URI is unique (no other concepts have the same URI), and each concept only has one URI to describe it. However, while uniqueness is relatively simple to ensure through naming conventions, it is very likely that any concept will have more than one URI associated with it, through the creators of the URI being unaware of the existence of others. Along with URIs and literals, a third type called a Blank Node (or B-Node) exists. This is used in situations where one wishes to refer to an implicit concept without giving it an explicit name. URIs, literals and B-Nodes are collectively referred to as 'nodes' in the rest of this document.

The use of URIs in RDF makes it easy to find documents that relate to information that an individual or agent is interested in and understands. For example, if someone (or their piece of software) is looking for information about dogs, and they know the URI *http://www.example.com/animals/dog* refers to the concept of a dog, they know that a triple containing that URI is certainly relevant to them.

In an RDF triple, the subject and predicate are guaranteed to be URIs, as they must refer to concepts (if I wish to talk about myself, it makes no sense to assert facts about the string 'Alisdair Owens', whereas it does make sense to do so about my URI). The object can be either a URI or a literal. URIs are related to each other through their expression in triples. This is shown in Figure 2.3.

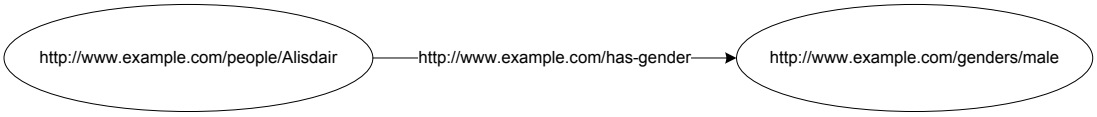


FIGURE 2.3: RDF triple

An RDF document is simply a set of RDF triples. As these triples refer to URIs, relationships between concepts are described, and a directed graph of information is created. This is a natural way to describe most information (Berners-Lee et al., 2001). This is illustrated in Figure 2.4, where for simplicity the prefix 'ex:' is used to replace 'http://www.example.com/'. There is no limit to the structure of this graph, beyond the need to express the data in triples format.

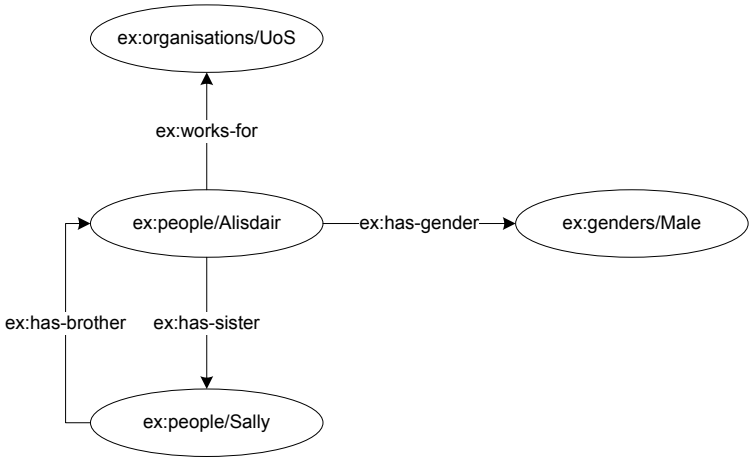


FIGURE 2.4: RDF graph

RDF, then, has a great deal of power and flexibility. It offers the ability to specify concepts and link them together into an unlimited larger graph of data. As a storage language, this affords several advantages:

- RDF supports simple data aggregation: linking data sources together can simply be a matter of adding a few additional triples specifying relationships between

the concepts. This is much easier than the complicated schema realignment that might have to occur in a standard data repository such as an RDBMS.

- The use of URIs offers the opportunity to discover new data, as the same URI is (conceptually) used to refer to a concept across every document in which that concept is contained. While this ideal will usually not be the case, any degree of URI reuse is of benefit.
- Since the data graph is unlimited, with no requirements for data to be or not be present, RDF offers a great deal of flexibility. There are no requirements for tightly defined data schemas as seen in environments such as RDBMSs, which is a significant benefit when the structure of the data is not well known in advance (Taylor et al., 2006).
- RDF offers a single language for representing virtually any knowledge. This is useful in terms of allowing reuse of parsing and knowledge extraction engines.

RDF offers a very useful data model, but as with any information, the topic of how to manage that data is important. Clearly, in the case of small datasets, it may be sufficient to simply statically store an RDF file, and allow individuals to process it as they wish. However, in many cases this approach will be inadequate: as the dataset grows, or concurrent users wish to access or modify it, it becomes necessary to have a system for managing it. This is the preserve of DBMSs, and the DBMSs of the Semantic Web world are known as RDF (or Triple) Stores. RDF stores allow a repository of RDF data to be queried in place, using a language such as SPARQL (described in Section 2.3).

2.2.1 RDFS and OWL

Semantic Web technologies offer the ability to infer new facts from the base facts found in an RDF datast. While not the focus of this document, it is worthwhile to give a brief summary of the languages that enable this feature. RDF Schema is an extension to RDF that adds some basic constructs (Lassila et al., 1999). Most importantly, this includes classes and subclasses, which allows statements about something's type. This means one could make statements such as 'Greg has a type of "Human" ', and, with an additional statement that a 'Human' is a subclass of the type 'Animal', infer that Greg is an Animal. Further additions include property domains and ranges, allowing the ability to make statements about the class of objects that act as the subject and object of particular properties.

OWL (and its recent successor, OWL 2 (Motik et al., 2009)) adds much more wide ranging capabilities, aimed at providing computers with the ability to share not just information, but vocabulary (Patel-Schneider et al., 2004). This means that potentially,

even if computers do not share the same understood ontologies, they might be able to communicate by expressing concepts and relations that they do understand. OWL adds extensive reasoning capabilities, varying within three sublanguages:

- OWL Lite, which offers minimal reasoning capabilities designed to support classification hierarchies. This enables reasoners to work with OWL Lite ontologies and produce relatively fast results.
- OWL DL, which offers a great deal of expressiveness, along with guarantees that all reasoning will be both complete and computable.
- OWL Full, which offers maximum expressiveness, with no guarantees that reasoning can be concluded in finite time.

Reasoning over RDFS and OWL ontologies is complex. Most RDF stores pre-compute much of the entailment of RDFS data (forward chaining). This effectively determines all the new facts that might be inferred and asserts them, leading to a relatively minimal impact upon query performance beyond the requirement to store more triples.

The pre-computation of the full entailment of even OWL Lite data is complex and likely to result in an explosion of the number of triples that need to be stored. Reasoning at the point of the query (backward chaining) is often too expensive to support interactive-time query satisfaction. This problem is largely outside the scope of this document, as it focuses on the issue of storing and querying the RDF graph, rather than performing efficient reasoning.

2.3 Data Extraction

Given a standard set of data representation languages, it is of clear use to have a standard mechanism for extracting subsets of information from documents expressed in them. A variety of query specifications have been created to accomplish this, with the SPARQL standard being the W3C's recommendation (Prud'hommeaux and Seaborne, 2006). SPARQL, like other languages of its kind (Seaborne, 2004; Karvounarakis et al., 2002; Broekstra and Kampman, 2003b), works by allowing users to specify a graph pattern containing variables, which is then matched against a given data source, with all matching datasets returned. Figure 2.5 gives an example.

```
SELECT ?x WHERE {
?x <http://www.example.com/has-gender> <http://www.example.com/male> .
}
```

FIGURE 2.5: SPARQL query

The query shown in Figure 2.5 would select all unique values ?x, where there is a triple that matches any subject ?x, and the specified predicate and object (in this case, anything with a gender of male). The data is returned in a standard XML-based format.

Queries can be built up into a pattern longer than one triple in length. In Figure 2.6, there are two constraints, which ought to return any URIs representing a human male:

```
SELECT ?x WHERE {
    ?x <http://www.example.com/has-gender> <http://www.example.com/male> .
    ?x <http://www.example.com/has-species> <http://www.example.com/human> .
}
```

FIGURE 2.6: SPARQL triple pattern

These query patterns are the fundamental operation in SPARQL. While SPARQL remains a simple language when compared to Structured Query Language (SQL), the equivalent in the world of RDBMSs, it has a variety of additional features: the ability to sort output, retrieve subsets of the output, or ensure that each output row is distinct, for example.

Two more fundamental operations are the ability to specify that some parts of the pattern are optional, and to specify that a given variable binding must match a certain value or pattern. An example with these two features, using the prefix 'ex:' to replace 'http://www.example.com/' is shown in Figure 2.7. This query finds all male humans who have fewer than 20 books, and if there is any record of the food they like, retrieves that too. Note that if there is no record of the food they like, their information is still retrieved due to the pattern being marked as optional.

```
PREFIX ex: <http://www.example.com/>

SELECT ?x ?foods WHERE {
    ?x ex:has-gender ex:male .
    ?x ex:has-species ex:human .
    ?x ex:has-book-count ?bookcount .
    FILTER (?bookcount < 20)

    OPTIONAL {
        ?x ex:likes-food ?foods .
    }
}
```

FIGURE 2.7: SPARQL query using FILTER and OPTIONAL

The benefit to be gained through the use of a standard query language is clear: potentially, a human or computer could connect to any open data repository, make a very specific request for information, and retrieve machine-processable data. This is in stark contrast to the web of today, which machines have a great deal of difficulty traversing

in a meaningful manner, and which even humans can have difficulty in finding relevant information.

2.4 RDF in Relation to Other Database Models

As described in Section 2.2, effectively working with large datasets of any kind implies a system to manage that data: Database Management Systems. In any database system, data is accessed according to some model: that is, there is some logical concept of how data is laid out within the system. This section describes data models in common use today, with a particular focus on the pre-eminent relational model. It relates this knowledge back to the RDF data model as described in Section 2.2, and asks the question: is the RDF data model fundamentally different? The answer to this question dictates the extent to which the approaches used in traditional DBMSs can be applied to RDF stores.

2.4.1 Early Database Models

A database management system is a computerised record keeping system. This document distinguishes between the database, which is the body of data, and the database management system which manages that data.

The storage and processing of databases is one of the earliest uses of computer systems. Database systems were created to enable such enormous tasks as tracking inventory data related to the Apollo project. Early systems were designed for sequential access via tape drive, and were later adapted for magnetic hard drive storage. Data was stored in a strict hierarchical or network-oriented manner (Tsichritzis and Lochovsky, 1976; Taylor and Frank, 1976; Date, 1990).

What was notable about these database systems was that the manner in which they logically stored data reflected the way in which in which it was physically stored on the hard disk. Changes to the way data was physically represented (to improve performance, for example) necessitated changes to both the dataset itself, to match the new database structure, and to the applications sitting on top of the database such that they could physically traverse the data. These applications accessed the data in a procedural manner, navigating from node to node to find the information that they needed. This mechanism was optimised for the retrieval of individual pieces of data, rather than whole datasets matching particular criteria.

Clearly, this mechanism for data storage and management has significant disadvantages. Changes to the DBMS could result in a lot of work modifying existing databases to fit, and a requirement to modify existing applications to take into account the new data

traversal paths they would have to take. Further, writing queries was something that only a highly skilled professional would do, and while there was scope for the fine tuning of queries to maximise performance, it relied on the programmer working out the optimal manner in which to retrieve data. The modern database market has evolved massively from this starting point, thanks in large part to the relational data model, derivatives of which are pre-eminent in the DBMS market today.

2.4.2 The Relational Data Model

A radical diversion from early approaches was proposed by E. F. Codd in Codd (1970). In his approach, a mathematically complete data model based on set theory and predicate logic is used to define the logical storage of data, and the interactions that can be performed on it. This is known as the relational model. In particular, it emphasises the separation of this data model from the way the information is physically stored: that is, the DBMS may choose to lay the data down in storage in any manner, but the way in which the data appears to the user remains consistent.

The relational model defines data in terms of relations, consisting of any number of tuples and attributes. Relations are broadly analogous to tables, consisting of rows and columns. These terms are used interchangeably in the rest of this document. These relations are (conceptually) unordered. Each tuple is unique (since it makes little sense to assert the same fact twice). Data retrieval in the relational data model differs significantly to the way it was performed in prior systems, primarily in that queries are specified in a declarative language, which allows users to state what data they want to retrieve, without forcing them to specify how to retrieve it. Generally, in relational systems it is the responsibility of the DBMS to work out how to make the query run as fast as possible (Stonebraker et al., 1976). The component that performs this work is usually known as the query optimiser. This removes the burden of optimisation from the application programmer, and allows the database system to be queried with a much smaller level of expertise (Stonebraker, 1980).

The relational model is designed to support broad operations that return a large number of results: queries like ‘retrieve all mechanics who have worked on a car containing part x’. This was a relatively complex operation in previous data models, where each node would have to be separately navigated to through hierarchies that may not have been designed for this kind of query.

Relations can have a variety of operations performed upon them, each of which produces a relation as an output. This ‘closure principle’ means that query commands can be chained. These include, in particular, select, project, and join. These are explained below, and illustrated in Figure 2.8.

Select: A selection (or restriction) is a simple unary operation that returns all tuples in a relation that satisfy a particular condition. For example, one might select all tuples in a relation describing people, where the value of the 'Surname' attribute is 'Owens':

Project: A projection is a unary operation applied to a relation by restricting it to certain attributes. Non-unique results are filtered out of the resulting relation.

Join: A join is a binary operation used to combine information in relations based on common values in a common attribute.

ID	Surname	First Name
1	Owens	Alisdair
2	Owens	Sally
3	Smith	Daniel
4	Livingstone	Ken

Table 1: Table describing individuals

ID	Has-visited
1	Boston
1	London
1	Lyon
2	Boston
2	Edinburgh
2	London
2	New York
3	London
3	Portsmouth

Table 2: Table mapping individuals' IDs to places they have visited

ID	Surname	First Name
1	Owens	Alisdair
2	Owens	Sally

Result of selecting over the Surname 'Owens' on table 1.

Surname
Owens
Smith
Livingstone

Result of projecting over Surname on table 1.

ID	Surname	First Name	Has-visited
1	Owens	Alisdair	Boston
1	Owens	Alisdair	London
1	Owens	Alisdair	Lyon
2	Owens	Sally	Boston
2	Owens	Sally	Edinburgh
2	Owens	Sally	London
2	Owens	Sally	New York
3	Smith	Daniel	London
3	Smith	Daniel	Portsmouth

Result of joining table 1 and table 2 on the 'ID' column

FIGURE 2.8: Illustration of common database operations

2.4.3 Other Data Models

Since the relational data model gained dominance in the 1980's, other models have also been created. Perhaps the most heavily publicised challenger is the Object data model described in Atkinson et al. (1989). This is based on the familiar principles found in

object-oriented programming, and indeed these DBMSs are often used as a persistence mechanism for application objects.

In the object model, a database designer creates 'classes', which are templates describing objects that can be created. Each object stores certain data, and has 'methods' (defined by the class) that can modify or retrieve that data. Object-based DBMS have amassed a body of criticism (Date, 1990) for their perceived slowness and inflexibility: due to their very nature, it is difficult to perform arbitrary queries across these databases, as each object is designed to support specific operations. While the object model is very much appropriate for applications, which use the objects for pre-defined, specific purposes, a DBMS is much more likely to require more ad-hoc use. Some of the useful features of ODBMSs have been incorporated into many commercial databases, in a hybrid model called the Object Relational Model. This document does not consider Object Relational Models to a great extent: there is little need for the complexity of objects in a system that models tiny discrete data items such as triples.

There are many other models in existence. Increasingly common are Data Warehouses (DWs) and Data Marts. These are often, as in many RDF stores, built as a layer on top of SQL databases: indeed, SQL now provides explicit support for them. DWs are built for specialised applications such as business decision support, which often require complex, unpredictable queries over massive quantities of batch-updated data (Chaudhuri and Dayal, 1997). Warehouses may be constructed as an aggregate of many smaller operational databases, and are a very large task to construct: it is very important to define a data schema that effectively models business processes and captures the right information. Query performance is much more important than ability to process writes, and a lot of data (such as aggregate figures) is precalculated to save work.

Finally, a common model used by applications for data persistence is simple key/value pair storage, as evidenced in Berkeley DB (Olson et al., 1999). This DBMS allows arbitrary data assertion and retrieval, assuming the structure conforms to its simple model.

In general, most models work on a presumption that data will be asserted in a well-understood manner. Table 2.1 offers a brief overview of the differences between current models.

2.4.4 Representing RDF

While the purpose of RDF stores is similar to that of conventional database systems such as the dominant RDBMSs, Object-Relational DBMSs (ORDBMS) and Object-Oriented DBMSs (OODBMS), RDF graph storage and querying bears notable differences in terms of the structure of the data that is stored. Whereas existing database systems largely require that the data structures that can be asserted into them (the schema of the data)

are defined prior to assertion of actual data (Date, 1990), RDF stores allow arbitrary assertion of knowledge in the form of triples (or quads if provenance information is also desired). While the very concept of a triple is a data schema in and of itself, it is extremely loose compared to that expected to be defined within most previous database systems.

There are important reasons why it is necessary to explicitly define schema in existing database systems:

- It defines what data is expected to be asserted into the system. Since most current databases act as knowledge stores for a fixed set of applications, this is usually both reasonable and useful: it prevents the assertion of data of an incorrect structure for those applications to use, and preserves data integrity (Date, 1990).
- It offers cheap, detailed information to the DBMS on how the data is structured: how it might best be laid down in storage, how queries can be optimised using knowledge of indexes, row lengths, and so on. (Date, 1990; Stonebraker et al., 1976)

2.4.4.1 Storing unstructured data

While the requirement for strict schema definition is usually helpful in traditional database environments, the situation regarding RDF storage is rather different, as RDF has been designed to be as unconstrained as possible. As previously noted, this has advantages in terms of accessing arbitrary data sources on the Semantic Web, interoperation between heterogeneous data sources, and situations where the data is of unknown or constantly changing structure (Taylor et al., 2005). However, lack of defined structure generates difficulties in optimising stores such that they are capable of storing large numbers of triples, and querying them in an efficient period of time (Carroll et al., 2004; Smith et al., 2007). Current RDF stores are restricted to storing orders of magnitude less data than relational systems (Lee, 2004; Schmidt et al., 2010; Bizer and Schultz, 2009).

As noted, an individual installation of a traditional DBMS product is likely to have a known set of applications running upon it. Thus, the access patterns can be anticipated, and the database can be optimised for those patterns through the use of indexes and other tactics. While arbitrary access is supported, this can be dramatically slower than doing so through the predicted routes (Date, 1990). In contrast, an open data node (a store that is publicly accessible) on the Semantic Web might be used in a variety of manners. It could be accessed in a completely arbitrary manner, as different users request different information, or it might have a certain set of applications that perform the majority of data requests. It might have to adapt to new applications suddenly

adding a lot of load with a new style of query that it had not previously had to satisfy often (Erling and Mikhailov, 2008).

As mentioned in Section 2.4.3, constructing a basic schema for RDF storage is straightforward: indeed, it is possible to represent RDF using the relational model and translate SPARQL queries into SQL (Harris, 2005). Many RDF stores are built into or on top of existing relational DBMS engines, and even non-relational RDF stores usually use the concepts of select, project, and join to answer queries. Conceptually, RDF can be modelled as simply a long list of triples, and this can be represented using a single relation. If one wishes to normalise, one can use more tables to store a list of unique URIs and literals, with the triple table itself storing integer keys into those tables.

Unfortunately, RDF's flexibility (in both the manner in which data is represented and the manner in which it is queried) presents a barrier to creating more complex, expressive representations. The ease with which the structure of an RDF document can change makes the creation of anything but the most simplistic of fixed storage schemas very challenging. This can be considered the major factor that differentiates the RDF model from the other common representations. These differences can be seen at a glance in Table 2.1.

	Intended Use	**Expected Data Structure**	**Queries**
RDF	Arbitrary knowledge representation	Triples, potentially no greater repeating structure	Unknown level of query predictability
Relational	Application support, knowledge base	Tables, predefined structure	Mostly predictable queries, but includes arbitrary query support
Object	Application support	Objects, predefined structure	Mostly predictable queries, may include some arbitrary query support
Data Warehousing (various)	Decision support, statistics, knowledge base.	Tables, predefined structure	Limited query predictability
Key/Value	Application support	Key/value pairs	Unknown level of query predictability, relatively simplistic query support

Table 2.1: Database model comparison

An attempt to implement a more descriptive schema that adapted to the structure of the data is described in Wilkinson (2006), but this approach has its own issues. While it was

shown to confer some performance advantages, and attempts were made at managing the evolution of the schema automatically (Ding et al., 2003), it generally proved a complex, largely manual task (Abadi et al., 2007). As will be seen in the following chapters, the difficulty of creating anything but the most general of schemas for RDF in the relational model is mirrored by a difficulty in creating a physical storage schema that provides adequate performance: indeed, it has been suggested that RDF stores produce incomplete results in an attempt to limit query costs (Erling, 2009).

2.4.4.2 Further information

There are several more features of the RDF data model that are of interest when constructing a DBMS implementation:

- There is no requirement for partial text searching over URIs: that is, while URIs are strings, there is no requirement to match over a portion of that string, because URIs are discrete concepts.
- Sorting has no inherent meaning for RDF URIs, since they are simply labels for a concept rather than data in and of themselves.
- There is likely to be a requirement for partial text searching over literals.
- Typically, most SPARQL queries specify a predicate, and are searching for either subjects or objects. It is less common to search for the predicate that connects two concepts (Seaborne, 2008).
- RDF typically has a large number of data points (triples) relative to the physical size of the data.

2.4.4.3 Summary

RDF storage poses a variety of challenges for traditional datastores, largely due to the unpredictable nature of information encoded in RDF, and the unpredictable manner in which such data may be queried. It is important to solve these challenges, most particularly because the problem of RDF storage is important to the success of the Semantic Web. If individuals or organisations are to be expected to host data and allow users to query it, particularly in a free environment, it has to be feasible to support low latency, concurrent queries over large quantities of such information. Interfaces on to RDF data for human users must maintain the interactive performance to which they have become accustomed.

The following chapter goes on to explore the issues of importance when creating a DBMS, and the techniques used to approach those issues. These are evaluated for their relevance

to RDF storage, with the purpose of informing the creation of future, high performance RDF DBMSs.

Chapter 3

Related Work

RDF storage and query is a challenging problem, thanks to the nature of the RDF data model: data structure and query load are both highly unpredictable, and each data point in an RDF document is very small, implying a large number of data points to encode a meaningful amount of information. Managing and working with such a large quantity of points in a performant manner is difficult.

This chapter considers mechanisms for improving the performance of RDF stores, drawing on knowledge from the wider world of relational DBMSs, and existing experiences of RDF store creation. This knowledge is analysed, and opportunities for improvement are derived. Several important factors in the creation of a high performance RDF store are considered:

- Section 3.1 provides background on the architecture of modern computer systems. This is of critical importance when designing a DBMS, and highlights common misunderstandings with regard to the manner in which hardware components behave.
- Section 3.2 examines the problem of translating the RDF data model into a representation suited for storage and retrieval on a computer, using the knowledge gathered in Section 3.1 to examine the techniques used in current RDF stores.
- Section 3.3 tightens the focus to indexing data structures. Since RDF stores typically have to extract small amounts of data from a vast corpus, while experiencing unpredictable queries, the indexing technique used is extremely important. This section reviews the most promising indexing technques in the DBMS world, analysing their suitability for RDF storage.
- Section 3.4 describes the importance of the join operation in RDF storage, and how the amount of time spent joining can be minimised through careful query optimisation and precalculated joins.

- Section 3.5 describes the primary method for scaling RDF stores to extremely large quantities of data: clustering information across multiple machines. This section includes a description of work performed by the author aimed at overcoming the issues that RDF presents with regard to distribution.
- Section 3.6 briefly describes how the techniques described in the preceding sections are implemented in a variety of existing RDF stores.
- Finally, Section 3.7 distils the preceding sections into an analysis of the most promising opportunities for future work.

3.1 Characteristics of Modern Hardware

In order to understand how to create a performant RDF store, it is obviously important to understand how the hardware on which a store is to be run behaves. This section offers a brief overview of the components of modern computers that are particularly relevant to DBMS performance, with a focus on the commonly used x86 architecture.

3.1.1 Disk

The majority of modern DBMSs make use of disk-based storage. It is plentiful and cheap, with consumer-level disks offering up to two terabytes of space.

Unfortunately, while the speed of CPUs has continued to rise dramatically, the performance of hard disks has not kept pace (Stonebraker et al., 2007). The speed of sustained reads and writes on the disk is quite slow, in the order of 100MB/s. Even more critically, there is an average seek time associated with travelling from one block of data to another non-sequential block in the order of 10ms. The specific value of this seek time is dependent upon how far apart the data is located (Abadi et al., 2006).

This storage medium, in particular its seek time, is a major limiting factor in both read and write performance in any disk-based DBMS. To put this in perspective, using a modern 3.0GHz processor that can execute up to four simple instructions per cycle, 120 million instructions might be executed in the time it takes to perform a *single* disk seek (Casazza, 2009). It has thus become necessary to attach a substantial number of disks to a typical database server, giving it the ability to perform several seeks in parallel. Upcoming solid state disk designs are less capacious, but by comparison feature much smaller seek times for reads. This is particularly relevant for RDF stores, which are generally required to process a great many very small data points: in this situation, assuming the processing cannot be kept fully sequential, access time is extremely important. It can be expected that as solid state disks mature they will become a popular choice for RDF storage.

3.1.2 Main Memory

On the face of it, storing data in RAM is a relatively simple matter: RAM itself has a constant access time, and its performance is vastly better than that of hard drives: modern paired consumer ram modules offer a throughput of over 10GB/s. This means that the requirement for pieces of logically contiguous data to be placed next to each other is looser, making RAM easy to work with. Since RAM is a resource that is consistently reducing in cost, it has the potential to become the main storage medium for applications that require very low latency.

Unfortunately, this view of RAM has been rendered overly simplistic, thanks to the failure of modern RAM technologies to keep up with the performance of CPUs (Boncz et al., 1999). While the bandwidth of RAM is very high, latency can be in excess of 200 processor cycles (Boncz et al., 2005). This makes it impractical for modern processors to wait for memory every time they need access to a piece of data (Drepper, 2007). As a result, data going to and from RAM is held in caches on the processor. These are explored in more depth in Section 3.1.3, but the practical upshot is that contiguity of data access remains important even when working with a main-memory system.

Other difficulties in working with RAM are that it is limited in size and not persistent. Thanks to its increasing availability, however, main-memory stores are becoming more practical, leading some observers (Stonebraker et al., 2007) to call for certain classes of DBMS to become main-memory based. RAM's lack of persistence complicates this somewhat, as it must be possible to reconstruct the database into RAM from a persistent store (usually a hard disk) in case of failure.

3.1.3 CPU

Making efficient use of the CPU has become an increasingly challenging task, thanks largely to the fact that the rate of improvement in processor performance has outpaced that of supporting technologies. In particular, both disk and memory latencies for random access are now vastly greater relative to processor performance than in previous years (Hua and Lee, 1990; Keeton et al., 1998). This rapid growth in processor performance has been supported by increases in clock frequency, combined with changes such as the introduction of pipelined, superscalar architectures and the addition of multiple processor cores per CPU (Harizopoulos et al., 2006). A single core of a modern CPU is now capable of executing up to four instructions per cycle on certain workloads (Casazza, 2009) - or in the order of 12 billion instructions in a single second at a 3GHz clock rate.

3.1.3.1 Superscalar and Pipelined Architectures

In a nutshell, pipelining is the process of breaking down the work required to perform an instruction into its component parts, and executing them sequentially. If the pipeline is kept full (i.e. once stage 1 of the pipeline has finished executing part 1 of instruction 1, it immediately begins executing part 1 of instruction 2), the processor can execute one instruction per cycle, despite the fact that any given instruction will take several cycles to complete (Anderson et al., 1967). This process has the benefit of allowing the CPU to maximise the utilisation of its functional units, and hide the fact that there are latencies involved in the processing of an instruction that make it impossible to compute in a single cycle. Pipeline lengths can vary dramatically between processor designs: the Intel Itanium 2 has a short pipeline length of seven stages, as opposed to 31 for the Intel Pentium 4 (Boncz et al., 2005).

Superscalar architectures involve a processor being able to fetch and complete more than one instruction simultaneously. This is performed not with the simple duplication of all functional units within the processor, but by the inspection of the instruction stream to find suitable instructions available for execution given the currently available unused functional units (Boncz et al., 2005).

Both of these architectural improvements have the benefit of increasing CPU throughput without the requirement for increases in clock frequency. Unfortunately, neither is foolproof. Both require data-independent instructions if they are to operate with full effectiveness: that is, if one instruction depends on the output of another, it cannot enter the pipeline (or be processed simultaneously) until the first instruction has completed, and the processor may have to insert stalls, or wasted clock cycles into the pipeline (Riseman and Foster, 1972). Fortunately, modern processors have the ability to process instructions out of order, allowing instructions that do not depend on actions performed in the pipeline to 'jump the queue'. This is usually highly effective, except in situations such as a tight loop that operates repeatedly on a small number of pieces of data, resulting in a lot of dependencies within the instruction stream (Zukowski et al., 2006). In this case, a lot of processor cycles can be wasted.

Instruction pipelines also benefit from a predictable instruction stream: that is, if the instructions involve a conditional branch to another code area, the processor has to guess which branch will be taken and fill the pipeline with those instructions (Drepper, 2007). If the guess is wrong, the pipeline has to be cleared, resulting in the loss of all ongoing work within it. Modern CPUs include branch prediction units that attempt to decide which branch will be taken in advance based on past behaviour: these are effective for branches that exhibit predictability (Drepper, 2007), and on suitable workloads can achieve correct predictions in over 97% of cases (Yeh and Patt, 1991).

3.1.3.2 Caching

In order to hide the performance inadequacies of main memory, a complex set of caches has been created. Of particular import are the data and instruction caches, and the Translation Lookaside Buffer (TLB).

When the CPU is looking for information in memory, it will check its caches first. If one of the caches has the information, the CPU can access it at the cost of cache latency rather than main memory latency. If not, the information is transferred from main memory into the cache, and other information is evicted on a Least Recently Used (LRU) basis. Typically, an entire 'cache line' (usually 64 bytes on modern systems) will be transferred from memory at once, making subsequent accesses to adjacent data especially fast (Harizopoulos et al., 2006). Accesses to main memory are pipelined: that is, the system does not have to wait for one request to finish being retrieved from memory before sending off another (Badawy et al., 2004). Thus, if the system always knows what information it will require well in advance, the latency of RAM is not an issue. In practice, however, this is rarely the case.

Traditional CPUs have Level 1 (L1) and Level 2 (L2) caches. The L1 cache is small (on the order of 16-32KB each for data and instructions), and extremely fast. Data can usually be retrieved from this level in around three processor cycles. The L2 cache is larger (at two or more megabytes in total), and somewhat slower, requiring around 14 cycles to access: this is still an order of magnitude faster than main memory, however (Drepper, 2007). Multi-core processors may differ from this standard approach: the Intel Core i7 has a three level cache system, with each core having an L1 and small (256KB) L2 cache of its own, with a large shared L3 cache (Casazza, 2009).

As long as data and instruction flow is sufficiently predictable, or occurs over a sufficiently small set of data, information can be held in and retrieved from cache, allowing the exceptional throughput of modern processors to be utilised to full effect. A simplified hierarchy of data storage is shown in Figure 3.1.

Assuming a working set of information larger than these small caches, predictability is once again key to maintaining overall performance. In the case of loading the instructions to run a program, if the processor knows what instructions will be accessed, they can be prefetched into cache. Conditional branch instructions again cause issues, this time with the caching of instructions: if the processor does not accurately predict which branch will be taken, it may end up having to clear the pipeline(s) and wait on main memory to retrieve instructions. This kind of stall is especially severe since the processor cannot perform any out of order execution in an attempt to cover the error (Ailamaki et al., 1999).

In certain situations, the CPU can also perform data prefetching into cache. Modern processors can detect sequential access in situations such as iteration over an array,

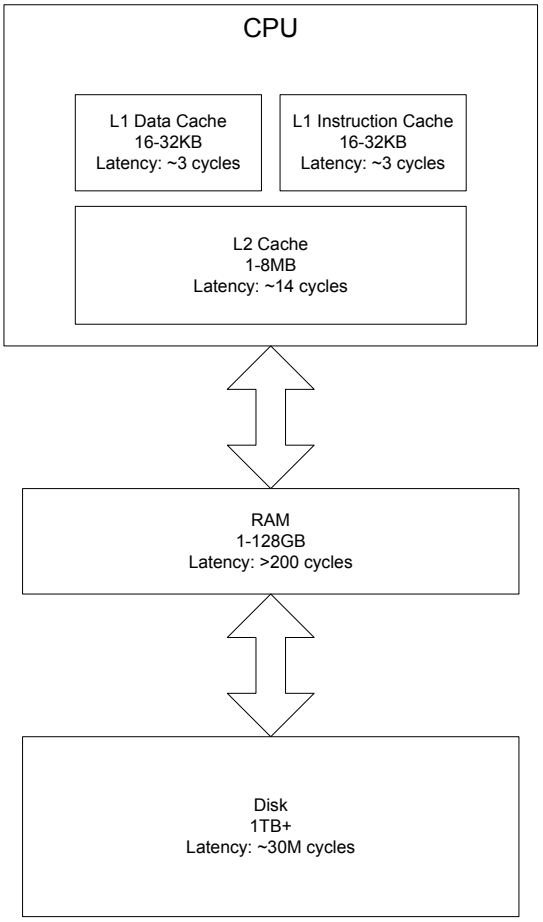


FIGURE 3.1: Data storage hierarchy.

and behave appropriately to fetch information into cache ahead of time (Drepper, 2007; Harizopoulos et al., 2006). Thanks to the high bandwidth of memory, extremely high performance can be maintained in this scenario. Other common operations such as tree traversal, linked list iteration, or binary chop over an array do not benefit from this optimisation, however, resulting in poor processor utilisation.

In modern operating systems, each process is given access to an area of memory that appears sequential, unused by any other process. This area is known as a virtual address space. Virtual addresses within this space are then mapped by the OS onto physical memory addresses. Since the process of translating virtual addresses to physical ones can be quite expensive, even in a system that performs much of the work in hardware, modern processors have a TLB. The TLB is a cache that stores commonly used virtual to physical address mappings (Ailamaki et al., 1999). The more memory pages an application uses, the more entries are required in the TLB, increasing the likelihood of overflowing its capacity and requiring expensive manual translations for memory accesses (Drepper, 2007). In practice, for applications that use a lot of memory, the TLB encourages accessing only a few pages at any one time: staying within these pages means that the

application will have all or most of its virtual to physical address mappings cached, improving performance.

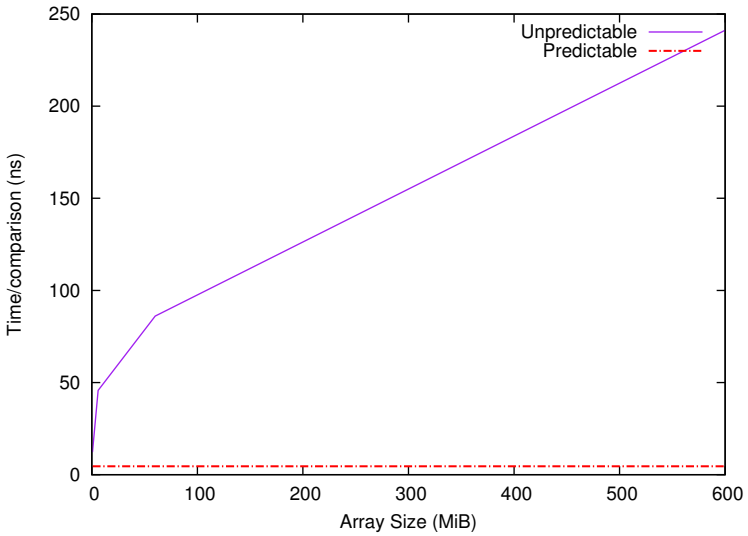


FIGURE 3.2: Binary chop cost per comparison as dataset increases in size

Array Size (MB)	**Comparisons Required**	**Unpredictable (ms)**	**Predictable (ms)**
0.6	18	2200	820
6	21	9610	960
60	24	20660	1090
600	28	67540	1270

TABLE 3.1: Cost of binary chop as dataset increases in size

In general, as the working set of information moves outside of the capabilities of these caches, overall performance degrades significantly thanks to the relatively high latency of main memory. This is illustrated in Table 3.1 and Figure 3.2, where the author created a simple application to perform repeated binary chops with predictable (i.e repeating) and unpredictable search terms, over a given quantity of data. It can be seen that with unpredictable search terms the time required increases out of proportion with the number of comparisons required, whereas with predictable terms the scaling is more linear. This is because the predictable terms are consistently accessing the same, already cached values, while the unpredictable terms need to wait regularly on memory. The disparity is small for a dataset that fits in cache, because the entire dataset can be cached, but becomes huge as the dataset scales up. The code for this test can be found in Appendix A.

Given this information, it can be seen that it is important for applications which require extremely high performance to ensure that data is compact, and that related data is

located contiguously where possible, maximising cache utilisation. DBMSs have historically performed poorly at this task (Ailamaki et al., 1999; Keeton et al., 1998; Knighten, 1999).

3.1.3.3 Multiple Cores

Attempts to increase clock frequency have come up against hard limits. As clock frequency increases, power consumption (and hence heat production) increases out of proportion. The traditional offset for this, reduction of the scale at which processors are manufactured, has become insufficient (Agarwal et al., 2000; Sutter, 2005), and a new approach to improving CPU performance was required beyond simply ramping up the frequency. The result of this is multi-core CPUs, essentially multiple processors on the same die, with certain shared components (for example, one level of cache may be shared).

Programming for multiple cores has its complications: thread synchronisation across processors is complex, and keeping caches current when a single location in memory may be altered by multiple cores can cause serious performance degradation (Drepper, 2007). Typically, however, multi-user DBMSs are well placed to take advantage of multi-core CPUs: these systems are inherently multi-threaded, working on several nearly independent problems at the same time. Many of the techniques used in distributed DBMSs (explored further in Section 3.5.3) can be applied to the problem of concurrent, single machine DBMSs.

3.1.4 Network

The behaviour of computer networks is important when discussing distributed stores. Typically, a round trip over gigabit ethernet with no other traffic has a latency in the order of 0.2ms (Erling and Mikhailov, 2008), and the maximum bandwidth for an individual Network Interface Card (NIC) is 1Gbit/s.

Practically, two factors have a significant impact upon these stated figures. Firstly, the effective bandwidth of the system reduces with an increasing number of messages: there is a significant overhead associated with sending a communication and the necessary acknowledgement. This means that effective bandwidth increases as the size of messages goes up (Erling and Mikhailov, 2008). Secondly, the structure of the network makes a big difference to the overall bandwidth between two machines. Two machines that are communicating across several network switches are much more likely to require access to a contended network line than two machines located on the same switch. Switches also tend to impose a substantial latency in the order of tens of microseconds per operation (Ousterhout et al., 2010), unless they are of a specialised high performance variety. It

is thus desirable to keep communication limited to machines on the same hub as far as possible.

3.1.5 Summary

This section provided an overview of the components of a modern computer system with special relevance to the creation of a DBMS. A recurring theme in modern computers is the issue of latency. Both disk and RAM have a very high latency compared to their maximum throughput, and the CPU experiences latencies in the processing of instructions: it has mechanisms to disguise them, but they only work if the workload is sufficiently predictable.

In order to achieve the highest possible performance, these components require predictable, contiguous access. This presents a significant challenge for DBMSs, since their job is usually to work with and extract relatively small amounts of information out of an extremely large corpus, an activity that inherently involves a certain amount of non-sequential access. The challenge, then, is to limit unpredictable access as far as possible without processing too much irrelevant data, or causing storage footprints to balloon overmuch. The balance between these factors depends on the components in question.

In addition to favouring sequential access, both RAM and disk provide a block of information in the course of an access: in the case of a disk, a page in the order of 4-16KB is retrieved. In the case of RAM, the equivalent is a 64 byte cache line. The difference in cost between doing work on only one datum in this block and doing work on the entire block is relatively small: in both cases, the cost of retrieving another nonsequential block is usually high compared to the cost of actually doing the work. The practical upshot of this is that data structures should attempt to make all of the data within a block at least somewhat related, as this extra information can be processed cheaply.

3.2 Physical Representation: Translating a Data Model into a Performant Storage Layer

As noted in Section 2.4, modern DBMSs have a logical view onto data that is not required to match the manner in which data is physically stored and manipulated on the system. The topic, then, of translating a logical representation into a performant physical one is clearly of great importance. This section considers the host of factors and challenges involved in creating a performant physical representation for any DBMS (Date, 1990; Stonebraker, 1980; Hawthorn and Stonebraker, 1986), including:

- What is the optimal manner in which to store the data for a given storage medium? Is the goal to optimise for small database footprint or performance? If the answer is performance, is read or write performance the most important?
- How can the most efficient use of the various components of the system be made, in particular the CPU, memory, and disk?

3.2.1 Physical Representations in DBMSs

The physical representation of a database has a large impact on read performance, write performance and space utilisation, and is thus a topic of clear importance. There is often a requirement for trading off between these considerations, and the focus is chosen depending on the expected usage profile of the DBMS. The choice of physical representation is also heavily influenced by the chosen storage mechanism (such as RAM, hard drive, or even flash memory).

In general, the most common (O)RDBMSs have physical representations that are remarkably similar to the logical layout of the relational model. Data is written to the disk row by row, kept loosely sorted, or 'clustered', on a given column or set of columns (Rowe and Stonebraker, 1986). Typically, the table will be accompanied by one or more indexes that allows, for a specified key, the location of rows containing that key to be located promptly: this is necessary since as the table grows it quickly becomes impractical to scan through all entries. Since indexes are of particular importance to RDF stores, due to the exceptionally long tables that they can require, the topic of indexing is explored in more detail in Section 3.3.

Row oriented representations can be considered optimised for write performance, in that adding a row to a table usually only requires a single write operation to the backing storage. This is appropriate for the most common DBMS tasks, such as a backing store for a web site, or storing employee payroll information, since data may change at any time and there is little requirement for performing extremely complex queries: most read operations will involve retrieving a single record (Shao et al., 2004).

Optimising for writes in this fashion can have a significant impact on read performance, however, which is of great importance for other applications such as data warehousing and decision support. Row-orientation means that in performing a select based on a single column, it is necessary to read the entirety of each row into memory. This results in greater data transfer, more memory use, less efficient use of CPU and disk caches, and is particularly damaging on wide tables. Finally, the fact that data is not maintained in correctly sorted order means that additional disk seeks can be required when retrieving data, and the cost of join operations increases (Stonebraker et al., 2005).

If database use is expected to be heavily read-biased, one might choose to optimise for reads. Characteristically, a read-optimised DBMS will maintain strict sorted order, and

may store its data in columns: that is, each column of data will be stored contiguously in disk or memory. This benefits read performance significantly when working with specified columns over a larger table, as irrelevant columns can simply be ignored (Stonebraker et al., 2005). In addition to a reduction in wasted memory and disk transfer time, this lack of wasted space has a beneficial effect upon CPU cache performance, as related data is more likely to be colocated within cache lines (improving access times, and resulting in less wasted cache). Schemes to improve the cache utilisation of row oriented DBMSs also exist, an example of which is PAX (Ailamaki et al., 2001). PAX stores information row-wise overall, but column-wise within a disk block, resulting in improved cache utilisation without significantly increasing time spent writing to disk.

In general, when designing the physical layer of a DBMS, the following rules of thumb should be considered:

- *When attempting to optimise data assertion performance, it is important to minimise the amount of data written to storage.* This includes reordering of data: for example, if data is kept in sorted order on disk or in memory, it is expensive to perform an insertion.
- *When attempting to optimise data retrieval performance, it is important to minimise the amount of data that is read from storage.* This does not necessarily mean that the data footprint should be small: if the data is stored in several representations, it is necessary only to read from the one that will allow the retrieval of the data in the quickest time. It is often useful to maintain data in sorted order, contiguously on the storage medium.
- For both cases, it is important to read or write the data as contiguously as possible to reduce the impact of memory and/or disk latency.

3.2.1.1 Compression

Thanks to the increasing disparity between disk and CPU performance, data compression has become a topic of increasing importance in the DBMS field. Where compression was originally utilised purely for the benefit of saving storage space (Stonebraker et al., 1976), it has now reached a point where in a disk-based environment the saving in the time taken to retrieve a chunk of data can actually result in improved overall query performance. This is thanks to the obvious improvement in effective transfer rate, combined with a reduction in average seek time due to the reduced distance between data points (Abadi et al., 2006).

Both read and write oriented stores may use compression. Most DBMSs that make use of compression inflate data either as it is streamed off disk, or in the process of

working on it. This necessitates extremely high performance algorithms of the kind described in Zukowski et al. (2006). As a result of this, DBMS compression techniques are usually very lightweight. Examples of these algorithms are simple dictionary compression, common prefix elimination, frame of reference (subtraction of a common maximum number and storage of the small delta), and run length encoding. These are commonly encoded at a block level (Poess and Potapov, 2003; Zukowski et al., 2006): that is, a given dictionary or common prefix will apply to a single (or small number of) disk block, reducing the cost of data changes when compared to maintaining a dictionary over the entire database. Some DBMSs also make use of more heavyweight algorithms such as Lempel-Ziv compression (Abadi et al., 2006; Ziv and Lempel, 1978), which can generally compress data reliably regardless of its format. This comes at the cost of greater compression/decompression time, and the loss of the ability to retrieve individual values: instead, a block must be decompressed en masse. To mitigate the cost of compression, some specialised DBMSs use dedicated hardware to decompress data as it is streamed off disk (Mueller and Teubner, 2009).

For memory-backed DBMSs, the use of even these simple forms of compression is a trading off of space for time, except in the most extreme cases. For data that is used rarely, this may be a desirable approach, but it does not offer the same clear-cut gain that compression in disk-backed systems does. However, in Abadi et al. (2006), the authors note that a better way in which to make use of compression is to integrate it into the query optimiser itself, such that the query optimiser can use aspects of the compression to its own benefit. For example, a join over two sorted, run length encoded columns is extremely simple compared to the equivalent join over uncompressed data. This adds significant complexity to the query optimiser, and is less simple to integrate into existing DBMS engines than simple pre-execution decompression, but represents the opportunity to create large performance improvements, even on in-memory systems.

3.2.2 Physical Representation in RDF Stores

While the creation of a simple logical representation for RDF is not difficult, it is challenging to create a performant physical representation. This section describes in detail the concerns with regards to implementation in RDF stores. This document does not offer any great detail on systems designed to put an RDF interface on an existing fixed relational schema, as described in Bizer and Cyganiak (2006): the focus in this document is on stores designed for unpredictable access patterns and unpredictable data changes.

Perhaps the standard model for an RDF triple store is that of a triple table storing identifiers representing URIs and literals, combined with mapping tables to translate these identifiers back into their lexical form. This approach is exemplified by 3Store (Harris, 2005), a system of moderate performance that runs on top of the MySQL relational engine. 3Store uses a single table in which to store the graph shape (as quads,

since it adds another field to denote provenance, or 'model'), as shown in Figure 3.3. Since MySQL is a simple row oriented store, the physical representation of this schema largely mirrors its logical structure.

Triples

Model	Subject	Predicate	Object	Literal	Inferred
64 bit hash	64 bit hash	64 bit hash	64 bit hash	boolean	boolean

Symbols

Hash	Lexical	Integer	Floating	Datetime	Datatype	Language
64 bit int	text	64 bit int	real	datetime	32 bit int	32 bit int

FIGURE 3.3: 3store data schema.

Each subject, predicate, and object field in the triples table contains a hash value, the actual text of which is discovered by joining to the symbols table, keyed on the hash value. This table contains information such as the lexical representation of the data, as well as integer, floating point and datetime representations stored for the purposes of performing comparisons between literals.

The answering of SPARQL queries is a relatively simple matter in this model: the SPARQL is translated into an SQL query that the underlying RDBMS can answer. For example, if one wished to answer the SPARQL query in Figure 2.5, 3Store might perform the SQL in Figure 3.4 upon the triples table.

```
SELECT subject
FROM triples
WHERE predicate=[hash of <http://www.example.com/has-gender>]
AND object=[hash of <http://www.example.com/male>]
AND model=0
```

FIGURE 3.4: SQL produced by 3Store

Clearly, additional SQL is required to determine the lexical representation of the hash values that would be returned by this query, but the mechanism is adequately illustrated. In the case of additional constraints in the SPARQL query, 3Store simply performs joins back onto the triples table. 3Store relies on the MySQL query optimiser to optimise the SQL it produces.

This schema offers a significant degree of flexibility, by virtue of the fact that any representation of triples is stored in a generic fashion, without requirement for schema or index customisation. There is no limitation upon the structure of the graph, except for the amount of data that MySQL can efficiently process.

The approach of a long triple table stored in a relational database is common in the world of RDF stores: popular systems such as Jena (Wilkinson et al., 2003), Sesame (Broekstra et al., 2003), and Redland (Beckett, 2002) all have popular backends that

utilise this kind of structure. However, while it is relatively simple to implement, and provides full support for RDF storage and query, it should be noted that the nature of the simple RDF schema described above is such that it is somewhat intractable for real RDBMSs: the triple tables are exceptionally long, with very little information per row. This has several effects:

- Very long, thin tables are a nonstandard optimisation case, making it challenging for DBMSs to produce relevant statistics to aid the automatic resolution of queries.
- An increasing quantity of rows usually increases the difficulty in finding any given piece of information.
- Typical queries become very expensive. Since a small amount of information is encoded per row, a useful amount of information typically requires a lot of rows to encode. Unfortunately, to answer queries, the triple table has to be joined to itself, and queries that involve lots of joins become rapidly more costly as the number of rows in the working set increases (Date, 1990).
- (O)RDBMSs usually have a per-row overhead due to tuple headers that provide information about the row. While these headers are useful for ensuring optimal behaviour with larger rows, in the case of RDF stores they can overwhelm the size of the actual data being stored (Abadi et al., 2007).
- The row-oriented versus column oriented debate is relatively academic. RDF rows are so small in a normalised environment that the benefits provided by column orientation are reduced somewhat, particularly since RDF query matching often requires that the whole triple be retrieved anyway. Most stores thus stick to a row-oriented approach, although it is, of course, still beneficial to consider ways to reduce the size of the data that is being worked with.

An alternative structure for RDF data, called Property Tables, was described in Abadi et al. (2007). This approach simply assigned a separate table to every property, storing each unique SO combination associated with that property in the table. While initial results showed that this ordering gave substantial advantages, subsequent investigations showed that using a different sort order for the triple table approach substantially reduced the performance improvement (Schmidt et al., 2010).

As noted in Section 2.4.4, in a relational database there is usually an expectation that a fixed set of applications will be running, with a largely predictable query load. When performing queries that are unexpected, and thus do not have appropriate indexes to aid the retrieval of data, query performance can quickly become extremely poor (Date, 1990). Since the knowledge of what queries will be performed is typically very limited in an RDF store environment, RDF stores often employ a highly comprehensive indexing

scheme. This, however, has associated costs in build time, maintenance, and storage space, making indexing a topic of particular importance in RDF stores. Section 3.2.2.1 considers comprehensive indexing approaches in more detail.

3.2.2.1 Indexing Strategies

Since RDF stores typically operate on one or a few very large tables, it is impractical to simply iterate through them to find the data in which one is interested. Indexes make it possible to specify fixed values for one or several attributes, and jump straight to results with these values. Most modern RDF stores employ a heavy indexing strategy to overcome their large table handicap: indeed, many stores have no need for a table, because all possible access patterns are covered by their indexes (Weiss et al., 2008).

Assuming the use of an index type that is ordered (that is, given an SPO index, one cannot restrict by P alone: S must be specified), $N!$ indexes, where N is the number of attributes, are required if one wishes to maintain a truly comprehensive set. Some newer RDF stores do implement all six of the possible index orderings over RDF data (the merit of which is discussed further in Section 3.4.3), but it is more common to use just three indexes: Subject-Predicate-Object (SPO), Predicate-Object-Subject (POS), and Object-Subject-Predicate (OSP). It can be seen that for any given combination of subject, predicate, or object, a corresponding index can be found in this set that is suitable to retrieve related data. A second point of interest in this design is that the table becomes unnecessary: the indexes contain all the data, so all the work can be done within them.

RDF stores implement a variety of different indexing strategies. These are examined in detail in Section 3.3.

3.2.2.2 Normalising

As previously noted, many RDF stores normalise URIs and literals into unique integer IDs. This offers several advantages: much less space is used to store each triple, reducing storage requirements and time required to transfer information to and from backing storage, improving cache efficiency, and making comparisons (for the purposes of joins) vastly quicker. In addition, working sets require much less space in memory, and the complication and inefficiency of working with large, variable length data is eliminated.

The major disadvantage of this approach is that at some point the IDs must be transformed back into their real lexical values again. Retrieving each uncached ID to lexical value mapping may require seeks on the disk, so this process can be extremely expensive. In general, if the output set of a query is similar in size to the total of all the data that entered the working set, this normalisation scheme will significantly reduce performance.

Fortunately, however, the output set of most queries is much smaller than this, and in general complex queries will benefit significantly from this approach.

Where possible, it is clearly worthwhile to eliminate the ID to lexical value conversion. This is possible in some situations: with 64 bit IDs it is possible to encode integers, dates, floats, and even small strings directly in the ID. This process is known as *inlining* (Owens et al., 2008). Some overhead is required to distinguish between genuine IDs and inline values, as well as the type of the inlined data, but it is generally possible to inline large ranges of several data types. Any data outside those ranges can be assigned an ID and treated as normal.

The mechanism for creating an ID also deserves attention. As noted in Section 3.2.2, many stores take a hash of the lexical value and use that as the ID (others, such as Kowari (Wood et al., 2005), generate IDs iteratively). This approach has the advantage that conversion of the lexical values of URIs and literals in a SPARQL query into IDs can be performed by simply taking a hash. As a result, no lexical form to ID index is required, saving both time and space.

Hash generation of IDs is attractive on the surface. Unfortunately, it provides no guarantees that prevent the generation of duplicate IDs. A collision cannot be cheaply detected, and so in the event of such a collision incorrect results will be retrieved from queries. Stores typically use a large 64 bit ID space to minimise the likelihood of this, but the probability of collision is unintuitively high: assuming a hash function with perfect distribution, and a 64 bit ID space, a 200 million ID dataset has a probability of experiencing a collision of around 0.1%, while a billion ID dataset is nearly 3%. A 72 bit ID space allows for 3 billion IDs while maintaining a collision probability of 0.1%, while for 80 bit IDs this rises to nearly 50 billion. This behaviour is defined by the mathematical problem known as the Birthday Problem, illustrated in Figure 3.5. Note that in practice, the likelihood of a collision is somewhat higher, since hash functions do not generally provide a perfect distribution.

The alternative to hash generation, incrementing IDs, is safer but slower. It requires a smaller ID space, and so can save space in this regard, but also requires an index to allow conversion from lexical form to ID. This index needs to be consulted for every RDF statement written into the store, and so can have a significant impact upon insert performance. In practice, many RDF stores use hash-based IDs, but this decision would require review in mission-critical systems.

3.2.2.3 Updates and Deletion

Current RDF stores, particularly those that scale to very large numbers of triples, tend towards read optimisation. While the initial bulk assert can be extremely fast, subsequent assertions can exhibit much poorer performance.

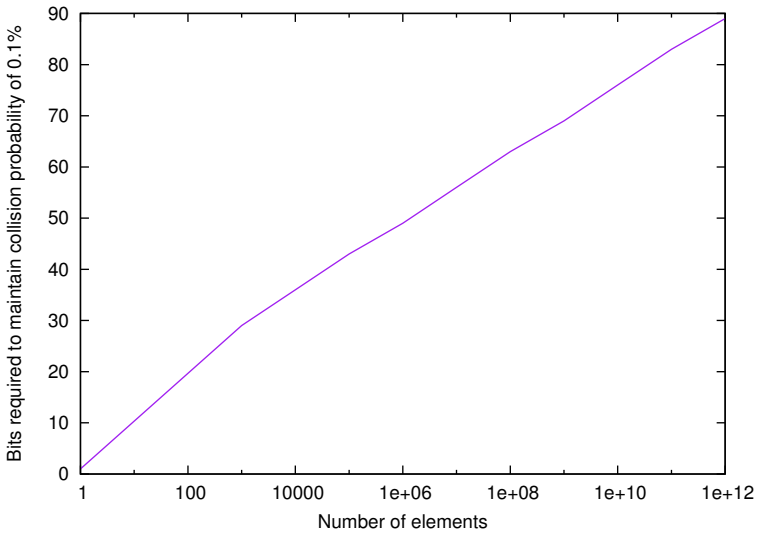


FIGURE 3.5: Graph of the the ID width required to maintain a 0.1% collision rate when using hash IDs

Deletions offer their own difficulties. In a simple RDF-only store there is little computational difficulty in eliminating a statement from the system, unless a complex compression scheme, heavy read-optimisation, or an unusual index structure is used. Recovering the resources the statement used is also difficult: assuming a normalised ID-based system, it is relatively time or space consuming to keep track of when IDs are no longer in use, and there needs to be a mechanism for ID recovery and reuse - whether it be an ongoing process or via bulk operation (which requires a sufficiently large ID and storage space). This is a relatively small problem in stores that do not experience significant deletions, but is important for systems that experience loads with regular updates. Current stores tend to be optimised for read operations, and do not perform ID deletion.

There is even greater complexity in deletions when it comes to systems that support inference (usually RDFS and/or OWL). Most RDF stores that offer inference do so making some use of forward chaining, or calculating entailment in advance. While this increases the amount of stored data, it usually dramatically reduces the cost of queries. Unfortunately, such systems do not usually keep track of how statements were inferred, meaning that when a statement is deleted, it is difficult to work out which inferred statements to remove. Keeping track of how statements were inferred (keeping in mind that this can happen more than once for any statement) is extremely expensive: an implementation was attempted in Broekstra and Kampman (2003a) for Sesame, but resulted in significant performance issues as data sizes scaled up.

As it stands, then, RDF stores today are largely found in read-mostly environments, which does not make use of RDF's flexibility. Work on incremental update and delete would provide a significant benefit.

3.2.3 Summary

Efficient physical representation of RDF is a significant challenge. RDF's highly variable structure does not lend itself to anything but the simplest of fixed schemas, and poses a challenge for adaptive systems. Unmodified RDBMSs are generally not suitable for the task of storing RDF: they are usually designed for wider, shorter tables, and issues like tuple header sizes and correct statistic generation inhibit performance. The Virtuoso ORDBMS (Erling and Mikhailov, 2009) is an example of a relational system that has RDF-specific modifications, and performs well.

Normalisation is considered to offer a substantial performance improvement over storing a triple table in lexical form. With this approach, most of the work in a query is performed on small, fixed size integers rather than large variable length strings, offering a less complex workload, smaller footprint, and a vast improvement in cache efficiency, as well as reduced I/O time in many cases. Correct implementation of normalisation still presents something of a challenge, with the most performant implementations suffering from the risk of data corruption, and most implementations never deleting mappings from hash to lexical form.

DBMS researchers are finding that compression can provide a significant performance benefit in disk based systems. Disk I/O is now so much slower than the rest of the system that it is cheaper to perform decompression than it is to transfer the uncompressed data. In memory based systems this benefit is less obvious, but the goal of reducing data size is certainly important: reduced data size generally improves the chance of cache line colocation as well as the total amount of information that can be held in cache, increasing overall performance. In addition, memory is a much more limited resource than disk, and so the goal of fitting more information into a given space is particularly important.

3.3 Indexing: A Key to High Performance RDF Stores

Storing data in an optimal manner for writing or later retrieval is all very well, but queries will still perform slowly if there is a requirement to scan through every row to find relevant pieces of information. To mitigate this problem, databases are indexed on columns of data (Date, 1990). This process creates a structure that, for a column or set of columns, quickly returns the location of specified data within those columns.

The topic of indexes has special relevance to RDF stores, because these systems are typically heavily reliant on them: the storage required for the indexes will often exceed the storage required for the data itself. This makes it especially important that indexes for RDF data are compact, fast, and easy to build and update.

There are a wide variety of indexing structures, each appropriate for different tasks. This section discusses the most popular and relevant of these, along with their performance characteristics, what applications they are suited to, and particularly their usefulness with regard to RDF storage and query.

3.3.1 Binary Search Trees

Binary search trees (BSTs) are tree structures in which each node is comprised of one given value, along with 'left' and 'right' pointers to subtrees that respectively contain only items less and greater than the node value. In general, for a balanced (that is, the height of any one leaf node in the tree is no more than one greater than any other leaf) tree, as depicted in Figure 3.6, a match can be found in $\log_2 N$ comparisons, where N is the number of items in the tree. Likewise, an insertion or deletion can be performed in $O(\log N)$ time.

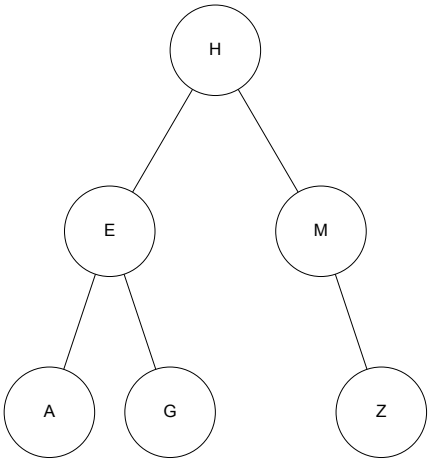


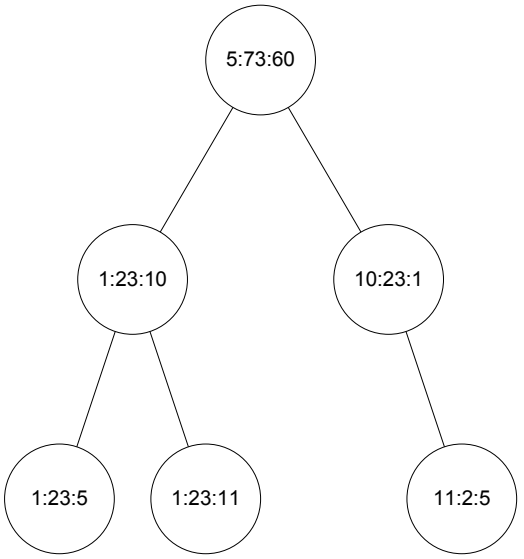
FIGURE 3.6: Balanced Binary Search Tree

Since a naive tree implementation will quickly go out of balance (and thus have a potentially worst case retrieval time of $O(N)$), there are a variety of different algorithms for trees that balance themselves automatically, and even (in the case of the Splay tree) for automatically optimising for quick retrieval of regularly accessed members. These algorithms include the Red-Black (Bayer, 1972), AVL (Adelson-Velskii and Landis, 1962), Treap (Aragon and Seidel, 1989) and Splay trees (Sleator and Tarjan, 1985). This document does not enter into great detail on each of these algorithms, but rather focuses on the broader characteristics of BSTs in general.

Since each traversal of a node will require a seek to a different location, BST-based indexes are fundamentally unsuited to storage on a hard disk. A BST indexing one billion items will have a height of 30, meaning 30 seeks are required to retrieve one datum. Data structures such as the B-tree (described in Section 3.3.2) are more commonly used for this purpose.

BSTs are often used in main memory-oriented systems. In this situation, this indexing mechanism offers reasonable retrieval times, fast in-order traversal, and potentially good space efficiency. The qualities of BSTs depend to a degree on node size, however: if the node size is small, then the storage overhead of the left and right pointers (in addition to any further information that a balancing tree will need to store) becomes significant. Node size also has an impact on cache efficiency: if a node is sufficiently small that more than one could fit into a cache line, a BST's poor contiguity of data access will often waste the opportunity.

BSTs, like all tree structures, also exhibit branch prediction issues: generally, unless the nodes that are being searched for are exceptionally repetitive, the branch that will be taken is unpredictable, with a corresponding impact on CPU pipeline performance.



ID	URI
1	ex:Alisdair
2	ex:likes-food
5	ex:beef
10	ex:Clare
11	ex:Dan
23	ex:has-friend
60	ex:meat
73	ex:kind-of

FIGURE 3.7: RDF stored using a BST

Figure 3.7 shows an implementation of a BST for an RDF store. This diagram shows a *composite* index in subject-predicate-object order: that is, the index is created over all three columns of a triple store. Composite indexes in trees are ordered: that is, it is impossible (or at least extremely inefficient) to determine who it is that 'likes-food' 'beef' using the tree in this example.

Since RDF stores typically have very small node sizes, BSTs are unlikely to be an effective index type. For 32 bit IDs, encoding a subject, predicate, and object will require just 12 bytes, relative to a minimum overhead of 8 bytes for the left and right pointers, or 16 on a 64-bit system. This is a space efficiency of just 60% at best, without even considering additional overheads: AVL trees, for example, require that a node store its height in the tree. In addition, since RDF stores are not subject to restricted range queries, in-order traversal is a higher level guarantee than is strictly required, although sorted output may be helpful for maintaining high performance joins.

Modified BSTs do see use in one notable RDF store: Kowari (and its derivative Mulgara) extend the node size by storing a range of values within the node (Wood et al., 2005). The left pointer is taken for values smaller than the minimum value in the node, and the right for those that are greater than the maximum value. Any search between the minimum and maximum results in a binary search for the search term within the node. This approach results in near 100% space efficiency for large node sizes, and is intended to maximise the utilisation of memory before being forced onto disk. It will also, however, usually result in a much greater tree height and thus more total seeks being required than in a comparable wide-node approach such as a B-tree.

3.3.2 B-trees

B-trees (Comer, 1979) are self-balancing tree structures in which each node can have multiple children, with each node apart from the root being required to be at least half full. This has the effect of offering a control over the height of a tree: the height of the tree is proportionate to $\log_n$, where n is the minimum number of items in each node, or the fanout. While the height of the tree decreases as the fanout gets larger, the number of in-node comparisons required to determine which child node to access increases. Overall, the number of comparisons required stays static.

This format is historically useful for block-based storage such as hard disks: keeping the nodes sized to a block (typically around 4-8KB) makes good use of the disk's characteristics: relatively few seeks are required due to the low height of the tree, and while each node is quite large, the cost of retrieving the whole disk block as opposed to a partial block is the same. Assuming the file system makes some effort to keep logically contiguous blocks physically contiguous, it is often worth expanding the node size to more than one block, and increasing the fanout further, since this additional data can be read very cheaply. By contrast, binary trees store a very small amount of data per node, drawing attention to the latency issues that hard disks experience.

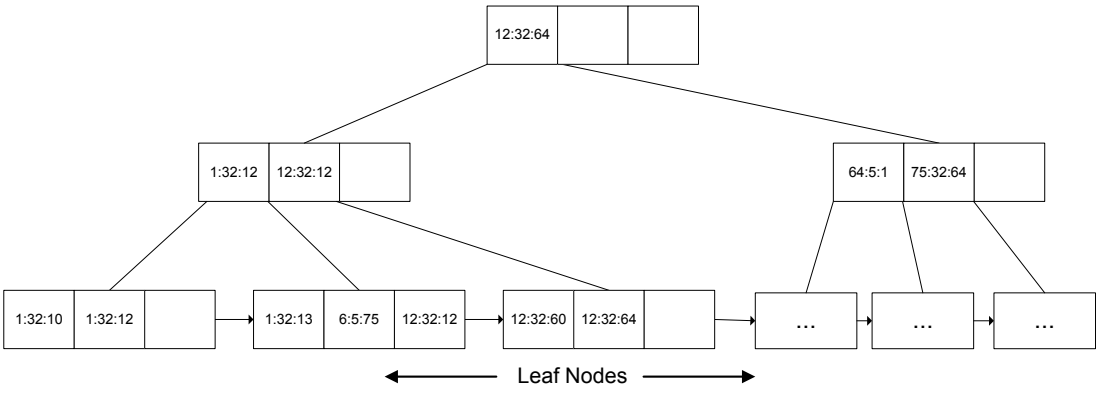


FIGURE 3.8: RDF IDs stored using a B^+tree

The B^+ variant of this tree (depicted in Figure 3.8) is particularly common, and modifies the structure of the B-tree such that all pointers to actual data are stored in the leaf nodes of the tree. This offers several significant advantages:

- Fanout can be increased somewhat without having to increase the size of the node, as data pointers are eliminated from non-leaf nodes.
- Leaf nodes do not require child pointers, saving space and improving locality.
- Leaf nodes can be easily linked, allowing high performance sequential traversal.

For the purposes of string comparisons, B-trees will typically store a sufficient prefix of the string to perform a comparison. Since RDF stores are usually performing comparisons of integer triple IDs, the entirety of each triple can be stored in the index. This means that for stores that offer comprehensive indexing, there is no need for a separate data table at all, and thus no need for pointers from the leaf nodes! Alternatively, the B-tree can simply store pointers to shared triple objects, but this incurs penalties for non-sequential access.

Another artifact of indexing over small datums is that the space used by child pointers is especially relevant, because they are a significant fraction of the space used in a node. Attempts have been made (Rao and Ross, 2000) to reduce the cost of these pointers, generally resulting in improved read performance due to increased opportunity for fanout and better cache locality, but at the cost of an increase in update costs.

As a result of their flexibility and reliably good performance, B-trees, in particular the B^+tree variant, remain perhaps the most common data structure for implementing disk-based indexes (Comer, 1979). Most triple stores backed by existing RDBMSs will make exclusive use of this index, and other dedicated systems such as Jena TDB (Owens et al., 2008) and RDF-3X(Neumann and Weikum, 2008) implement their own versions.

When considered for the purposes of main-memory DBMSs, the advantages of the B-tree and its variants are less clear cut. In particular, maximising fanout is no longer especially beneficial: the only analogy to ‘blocks’ in main memory are relatively small cache lines, and binary chop across large nodes does not make efficient use of cache prefetching. Further, since nodes are kept sorted, the larger the fanout, the more work required on insert into a node. This is generally trivial next to the cost of a disk seek, but, in an environment without such huge latencies involved, becomes significant.

It should be noted, however, that contrary to conventional wisdom B-trees can offer better performance than binary trees for in-memory indexing. If each datum held in a B-tree node is sufficiently small, a B-tree node may hold several, including pointers, in a single cache line. Doing extra processing on data already in the cache is extremely

cheap, so a correctly sized B-tree can mean fewer waits for main memory than with a binary tree (Rao and Ross, 1999).

Smaller fanouts tend to be more data cache friendly: a node size exactly the same as the length of a cache line guarantees a minimum of waits on the data cache. Initial studies on cache-friendly B+Trees thus used a node size of a single cache line (Rao and Ross, 2000, 1999). In practice, however, small node sizes result in skipping between a lot of different memory pages, causing the TLB's hit rate to drop. A small multiple of the size of a cache line offers superior performance (Hankins and Patel, 2003). Cache sensitive B+Trees have been shown to reduce find times by as much as 26% on data held in memory, and further gains can be made in an environment that supports programatic cache prefetching (Chen et al., 2001).

The B-tree and its variants are also guilty of wasting space, making their suitability for in-memory indexing somewhat questionable (Wood et al., 2005). They do not usually fill up each node with data (an average of 25% being wasted in a standard implementation), each node of size n contains $n + 1$ pointers, and when used as composite indexes the lower levels of the tree tend to contain a lot of repetition of data in the prefixes. This latter issue can be mitigated through the use of compression techniques such as those described in Section 3.2.1.1 and Lomet (2001), at the cost of increased update time and complexity.

An example of a store that uses compression is RDF-3X. This store implements compressed B+Trees, using a variable-length delta compression scheme to reduce their size (Neumann and Weikum, 2008). This scheme (using 32-bit IDs) performs a delta on each of S, P, and O in a node, comparing them to their counterparts in the preceding triple, and uses only the number of bytes required to encode that delta: for example, if $S_2 - S_1 = 502$, S_2 requires only 2 bytes to encode. Each triple has an overhead of one byte, which encodes the length in bytes of each of S, P, and O. This scheme saves a substantial amount of space, but does have a cost in terms of performance: firstly, in the operations required to perform the decompression, and secondly, due to the fact that the triples are of variable width. It is not feasible to perform a binary chop upon a tree node with variable width contents, so a linear scan must be performed, limiting performance somewhat. This tradeoff is likely to be beneficial for disk-backed stores like RDF-3X, but costly for in-memory systems.

3.3.3 T-Trees

T-Trees (Lehman and Carey, 1986) are a class of tree designed for in-memory DBMSs. T-Trees can be thought of as AVL trees with wide nodes, similar to the layout used in Kowari (described in Section 3.3.1). In this design, multiple logically adjacent values are held within each node. When traversing the tree to find a value, comparisons are

made against the smallest and largest values in the node. If the search term is outside the bounds of these values, the left or right child pointers are taken respectively. If the search term is between the smallest and largest values in the node, the node is searched internally.

The difference between T-Trees and Kowari's approach is that T-Trees store only pointers to the data they are indexing over, rather than copies of the data. To perform comparisons between a search term and values in the index, pointers to those values must be dereferenced. In many situations, this can save a substantial amount of space: when indexing over strings, for example, the string data need only be stored once no matter how many indexes point to it.

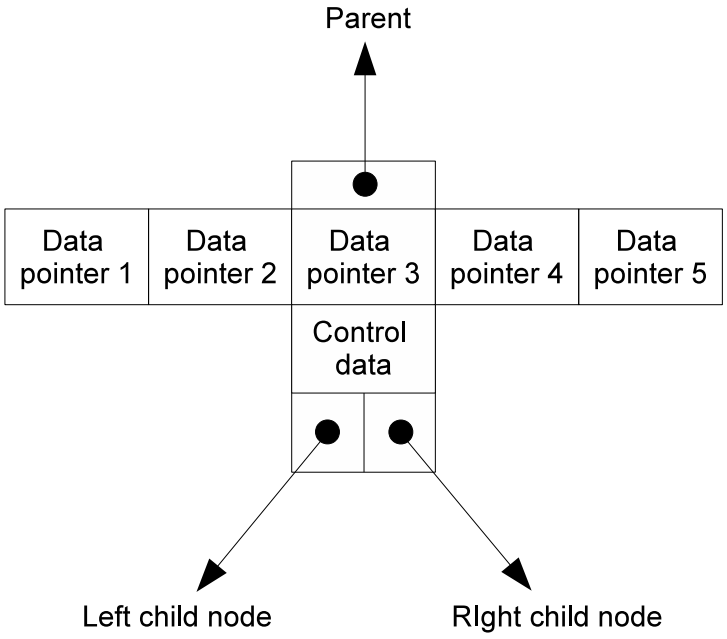


FIGURE 3.9: T-Tree node

The disadvantage of T-Trees is the requirement to dereference pointers every time a comparison has to be made. This disadvantage has become more acute with the passing of time, as the cost of the cache misses caused by pointer dereferences has increased, and as a result, T-Trees are now a relatively slow mechanism for indexing (Rao and Ross, 1999). For some applications, where each datum being indexed over is large relative to the size of a pointer, this relative slowness may be tolerable for the space savings offered. Since the size of an RDF triple is very small, however, the space savings caused by the normalisation of index data would be substantially offset by the cost of storing the pointers to that data. There is thus little reason to believe that T-Trees represent a worthwhile avenue of investigation for RDF storage.

3.3.4 Bitmaps

Bitmap indexes are popular for applications which require high read performance on arbitrary queries, particularly where it is infeasible to perform comprehensive indexing. They are commonly used in environments with wide tables such as data warehouses (Chaudhuri and Dayal, 1997). They are traditionally used for low-cardinality attributes such as 'Gender', or 'Country', but have been shown to be applicable even to columns with a high degree of unique values (Date, 1990).

A bitmap index simply creates a bitmap for each unique value that a column might take. Conceptually, each bitmap contains a bit for every item in the column, showing whether the field contains that value or not. Practically, a bitmap will usually encode a start and stop point, assuming all values outside that range are 0. This reduces the storage footprint, and also mitigates locking issues when updating these indexes.

The particularly useful feature of bitmap indexes is the manner in which the number of required indexes grows with the number of columns in a table. Consider the example of B-trees: if one wishes to implement a truly comprehensive index over a quad table, there are 4!, or 24 different indexes that can be created. Analysis of typical queries allows the removal of several relatively useless indexes, but as the number of columns grows (with the addition of, for example, temporal data) it quickly becomes impossible to maintain comprehensive indexing. With bitmap indexes, no such problem exists: it is only necessary to create one index per column. To perform a query such as that described in Figure 2.5, it is necessary only to retrieve the bitmaps for <http://www.example.com/has-gender> and <http://www.example.com/male>, AND them together, and examine the table at all positions in which there is a 1 in the resultant bitmap. Reading the bitmaps and ANDing them has excellent disk and cache performance: all work is done via sequential reads. Bitmap indexing and the mechanism for performing selects across columns are depicted in the context of an RDBMS query in Figure 3.10.

In order to make bitmap indexes space efficient for high cardinality columns, compression is necessary. Run Length Encoding (RLE) based algorithms such as Word Aligned Hybrid are usually very simple and worthwhile (Wu et al., 2006), and result in high space efficiency. Each bitmap in the index, once compressed, is quite small, tractable to load into memory, and can be joined to other columns using fast modified ANDing algorithms.

While bitmap indexes generally offer good read performance, they are more computationally complex to create and maintain than B-tree or hash based indexes, and demonstrate poor characteristics in terms of locking granularity: when performing an update, all bitmaps that encode values in the range of the update must be locked.

It should be emphasised that since bitmap indexes maintain only one sort order, accesses to the associated triple table will often not be contiguous. This is in contrast to the

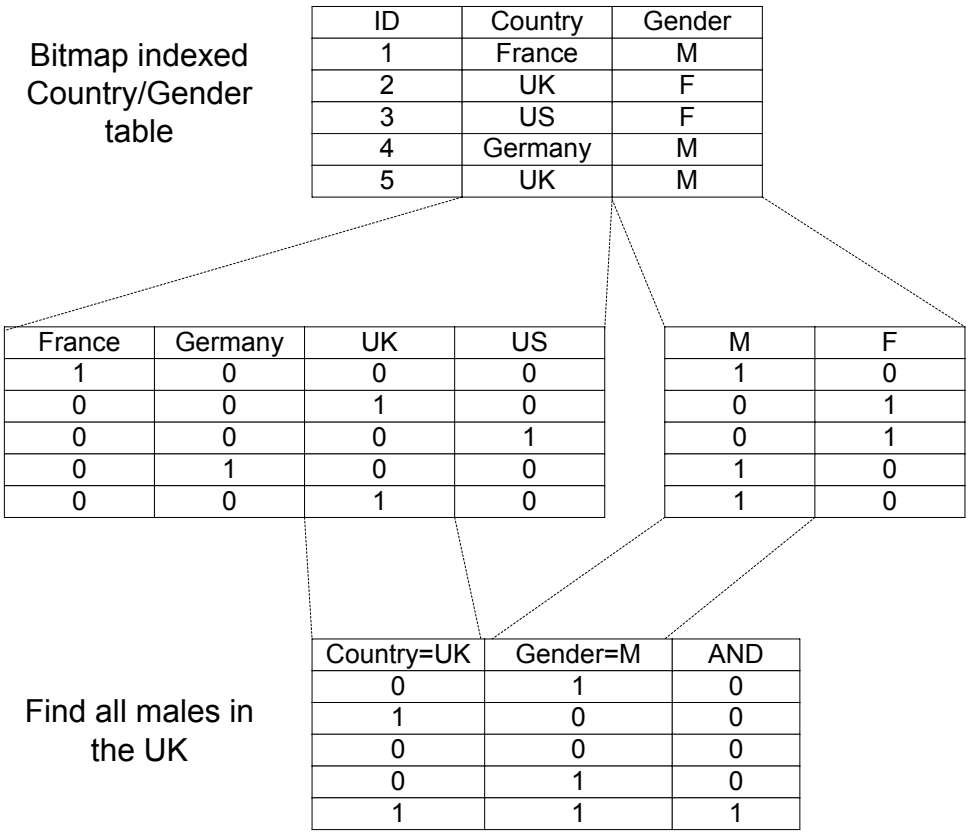


FIGURE 3.10: Querying using a bitmap index on a relational system

comprehensive composite tree indexes used in systems such as Jena TDB (Owens et al., 2008) that can encode all data within the index, and may significantly impact overall performance due to the cost of disk seeks and/or cpu cache misses.

From the point of view of disk-based RDF stores, Virtuoso (Erling and Mikhailov, 2009) has demonstrated that bitmap indexes can produce excellent results (Erling, 2006). Virtuoso uses a custom bitmap that acts very differently to the traditional approach shown in this section, however. In Virtuoso, bitmap indexes are composite and order-dependent: that is, they behave much like a tree based index in that, given a POS index, it is not possible to easily search by just O. Using this approach means that it is not necessary to AND bitmaps together: this has essentially been done in advance. It does, however, lose the property that the number of indexes required to maintain coverage scales linearly with the number of attributes.

3.3.5 Hash Tables

A commonly used index for RAM-based storage is the hash table (or hash map) (Date, 1990). Using a hash map, one might take the hash of a piece of data, and then store in a memory position corresponding to that hash a pointer to the location of that piece of

data in the database. This is an $O(1)$ operation, and since hash indexes usually require only one or two comparisons to be performed, the problem of unpredictable branches is effectively eliminated. It is, of course, necessary to utilise a suitable hashing algorithm to ensure that there are not too many hash collisions, and that the process as a whole offers good performance.

Unfortunately, hashes do exhibit a variety of less desirable characteristics. Hash indexes do not, of course, guarantee that there is any proximity on disk of logically ordered data (for example, sorted order). This means that if one were to perform a query that acts on a range of values, a disk seek would likely be required for each different value, creating massive efficiency issues. For this reason, hash indexes are usually used only in situations where queries are operating on discrete specified values, not over a range.

Unlike tree and bitmap indexes, hash indexes do not inherently provide support for composite indexing: it is thus necessary to create a sub-index below the primary level if one wishes to index over more than one column of data. Without careful design, this can be both slow and space inefficient. Taking an example from the RDF world, consider a Predicate-Object-Subject ordered index, for which a comprehensive index is required. If there are 100 predicates, and each predicate has an average of 1000 objects associated with it, and each of those objects has 10 subjects associated with it, a total of 101,001 index structures would be required to index only 10^6 triples. The per-index overhead would overwhelm the costs of actually storing the triples.

Consider, however, the alternative: with no sub-indexes (and thus allowing restriction only by predicate), only one index structure is required. However, if one wishes to check if the index contains a particular POS binding, one has to cycle through up to 10,000 values, comparing against each of them. This is a clearly undesirable alternative.

From the point of view of RDF/SPARQL, which rarely utilise limited range searches, hash maps can be an appropriate solution for both disk and particularly memory storage. Indeed, the most popular in-memory RDF stores such as Jena (Seaborne, 2009), Sesame, and SwiftOwlim (Ognyanoff et al., 2007) all use hash maps to store data. The problem of lack of support for composite indexes will become more significant as data sizes scale up, and might be resolved using subindex-based techniques.

3.3.6 Space Filling Curves

Space filling curve (SFC) (Sagan and Holbrook, 1994) can be used to produce multi-dimensional indexes (Bayer and Markl, 1998). SFCs are essentially a continuous curve that fills up any given square or cube (or even a hypercube of any dimension), assuming that object is constructed of discrete units. SFCs are usually repeating patterns that are constructed iteratively. Well-known examples of these are Z-order and Hilbert curves (Lawder and King, 2000), the latter of which is illustrated in Figure 3.11.

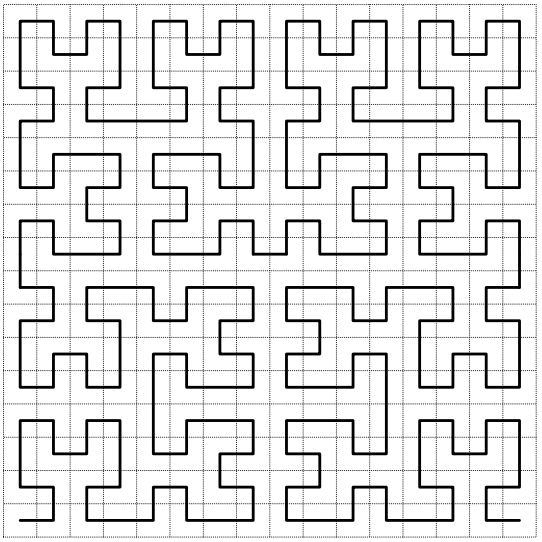


FIGURE 3.11: The two dimensional Hilbert curve

SFCs can be applied to RDF storage and indexing: the TriStarp[1] project has already utilised SFCs to store and index data in a non-RDF triple store. Taking RDF as a three dimensional storage problem (ignoring, for now, provenance), it can be imagined as a cube, with each dimension being one of subject, predicate, and object. An RDF triple is a point within the cube. The fact that RDF has more than one dimension is a problem when attempting to store it contiguously - in a one dimensional manner. SFCs can be applied to this problem: the curve passes through every point in the cube (or every triple, in this case), so the triples can be stored on disk in the order in which they are traversed by the curve. The result is a one dimensional representation of a three dimensional structure.

In SFC's such as the Hilbert or Z-order curves, indexing of this one-dimensional representation can be performed through a tree-based system (Lawder and King, 2000) (for example UB-trees (Bayer and Markl, 1998)). The repeating structure is evidenced at every level of construction of the SFC, and this repetition can be used to form a tree-based index into the curve.

Indexing via SFCs has the important property that no one dimension is dominant, as is the case with some more common techniques such as B-trees (Lawder and King, 2000). It is possible to retrieve data by any combination of dimensions (for example, fixing subject and property and searching for all related objects, or fixing object and searching for all related subject and properties). The particular dimensions that are supplied make no theoretical difference to query time (although if two dimensions are supplied, this will clearly be quicker under normal circumstances than if only one is). This property means that a single index can be used for all lookups, and could make SFC-based indexes

[1] http://www.dcs.bbk.ac.uk/TriStarp/

substantially more space efficient for RDF storage than the more common practice of using several conventional indexes.

SFC-based indexes are most often used in situations that require range selections over more than one dimension. Traditional DBMSs perform poorly at this task, since it is necessary to scan all datums that satisfy one of the ranges, and then restrict the resultant output by the other specified ranges. In the case where several broad ranges are required, or data is of low cardinality, this is extremely inefficient. Using SFC-based techniques, a volume is designated for retrieval, the points at which the curve intersects that volume computed, and these matching points retrieved (Ramsak et al., 2000). This property is of little relevance, however, to RDF stores, the queries for which usually come down to a fixed term (or set of terms), or the entire range of a dimension.

There appears to be little evaluation of the performance of SFC-based techniques as applied to triple graphs in the TriStarp system. It is possible to draw some inferences, however. While a good curve will keep spatially related information somewhat close together on disk, it is clearly impossible to maintain perfect locality, particularly as the amount of information expands. This is not a large problem for queries over a small range, but becomes a greater issue in RDF where, as noted above, queries of restricted range are not a common commodity. This property means that SFC-based indexes will inevitably involve a much higher proportion of non-contiguous accesses than indexes with a single dimension: this is a major issue for both disk and in-memory storage, where the costs of such a quantity of seeks are crippling. Combined with potentially high costs for calculating the location of a datum, this makes SFC-based indexes a space efficient but slow solution.

3.3.7 Summary

Indexing is of critical importance to RDF stores: indexes offer vast benefits when attempting to retrieve a few values from a very long table, which is a common situation in RDF storage. Indeed, some RDF stores exhibit such comprehensive indexing that there is no longer a need for the original data table. Maintenance of such a strategy is sustainable for triple stores, but becomes more challenging as more attributes, such as provenance or temporal information, are required.

Traditional B-tree indexes perform well for disk based storage. They are simple to implement, and require a small number of seeks compared to other tree-based methods to find any given item. They do waste a certain amount of space through their partially-filled nature and the repetition of prefixes, but this latter can be mitigated through correct application of compression.

For memory-based RDF systems, trees in general are a capable but limited solution. While they provide strong guarantees regarding sorted order, this is more than is required

for RDF stores that do not rely on merge sort, and thus have no use for sorted order. Trees generally waste a substantial amount of space in pointers and/or empty space, and offer poor characteristics with regards to contiguity of access during find operations. Block based tree indexes like B-Trees do, however, offer very good contiguity of access when iterating over leaf nodes.

Hash indexes are generally appropriate for in-memory RDF storage, as they offer amortised $O(1)$ retrieval and update with a low constant factor. Hash indexes bring with them their own issues, however. Care must be taken to ensure efficient use of space when creating hash indexes, and it should be noted that hash indexes do not inherently support indexing over more than one attribute. To scale to large RDF datasets, a space efficient solution to providing multi-attribute indexing is required.

3.4 Operator Implementation: The Importance of the Join in RDF Query

As noted in Section 2.4, the relational model implements several operators: most notably select, project, and join. Typically, with the aid of suitable indexes, performing a selection is quite cheap (Date, 1990). If a relevant index is available, it is possible to simply navigate directly to an item, and retrieve all subsequent tuples containing that data value. In this case, select scales linearly with the number of items that have been selected, and at worst logarithmically with overall table size, depending on what sort of index is used. Retrieval is complicated if the data is not clustered on the index: in this case, if no index is available, the operation scales linearly with overall data size. This can quickly become prohibitively expensive on large tables.

Projection is generally a brute-force algorithm, restricting a table to certain columns, and removing all duplicate values. Clearly, as the size of the data being projected over increases, the cost of projection increases in linear fashion. If data is sorted over the attribute(s) being projected over, little memory is required to perform the operation otherwise, it is necessary to remember previously seen values.

The operation of special relevance to RDF is the join: answering a SPARQL query over a traditional triple table schema implies joining the table onto itself repeatedly, once for each triple in the query. This can quickly become very expensive if the working set of information is allowed to grow too large. There are thus two areas of particular importance when attempting to reduce time spent in joins: a high-performance join algorithm, and minimising the set of data to be joined in the first place.

This section provides a brief overview of query optimisation to illustrate the importance of the order and manner in which operations are performed. The various mechanisms

for joining are then explored further in Section 3.4.2, followed by a brief exploration of precalculation as a method for reducing time spent in joins.

3.4.1 Query Optimisation

In the leap from procedural database systems to RDBMS, a switch was made to declarative query languages: that is, the agent making the query merely specifies what data is desired, not how to retrieve it. Working out how to retrieve the data is the job of the query optimiser and is, as Youssefi and Wong (1979) notes, of critical importance: while the same overall result will be obtained whatever order operations are performed in, a bad query execution plan can potentially cause data retrieval to be many orders of magnitude slower than it ought to be.

Automatic query satisfaction is not a trivial task. However, while a programmer may intuitively know the most efficient manner in which to process a query, this is by no means guaranteed, and requires significant insight and expertise. An automatic query optimiser can evaluate many different plans before settling on one with a low cost, and can do so without the input of a knowledgeable human. As noted in Date (1990), there are four steps to query optimisation:

1. Cast the query into internal form.
2. Convert to canonical form.
3. Choose candidate low-level procedures.
4. Generate query plans and choose the cheapest.

The first two stages essentially transform the query from a textual representation such as SQL or SPARQL into an internal form that is easier for a machine to process, performing trivial optimisations such as eliminating irrelevant statement ordering on the way. Step 3 is more complex, and involves working out low-level operations that can satisfy parts of the query. This attempts to produce worthwhile operations by considering such information as physical data structure on disk, availability of indexes to speed the operation, and so on. Each potential operation will have an associated cost calculated for it, at the minimum specifying number of disk accesses required, but possibly also including information such as memory and CPU usage. This data may be estimated where hard figures are not available or easily calculated. Depending on whether the operation has prerequisites for other operations to be performed first, it may well be possible to perform them simultaneously across multiple processor cores, processors, and disks to enhance performance.

Finally, step 4 involves the creation of a set of potential plans from the procedures generated in step 3. Clearly, there could be overwhelmingly many plans produced if there were a significant set of candidate procedures generated, so a heuristic to create only plausible plans is of great use in this situation. The order in which operations are performed has a huge impact on query performance: if the correct operations are performed early in the query, the working set can be cut down to the point that later, more challenging operations only have to work on a small amount of data.

While this overview gives a broad explanation of query processing, the implementation of these steps is quite difficult. SQL, the standard for most modern RDBMS, is extremely complex, and the creation of a high-quality optimiser for most cases is a difficult task, accomplished in a wide variety of manners. The cost of operations is usually calculated from statistics stored for each table, and the columns within them. Examples of this include cardinality of the table as a whole and the number of pages it occupies, as well as the number of distinct items in each column, and average values for each column. These statistics are quite simple, but can make a significant difference to the creation of an optimal strategy. Since they are so small, they can be stored in memory and accessed with great ease.

Satisfaction of SPARQL queries does not differ in concept, but has some differences in terms of implementation. RDF stores typically have a very few extremely long tables. This means that the statistics on each of those tables need to be very much more in depth than is normal for an RDBMS to in order to provide adequate results: otherwise the information available may be insufficient to provide good cost estimates. Virtuoso (Erling and Mikhailov, 2008) goes as far as performing real time sampling of the data rather than expending large amounts of storage on the necessary statistics.

In practice, it is reasonable to make some assumptions about the nature of RDF data: typically, there are many fewer properties than subjects or objects. This means that property-oriented subqueries should be pushed late into the query plan. It is also generally practical to store information about the cardinality of every property. This helps avert the worst-case situations that are of special importance when answering a query. These assumptions are explored further in Chapter 5.

3.4.2 Types of Join

Joining can be an expensive operation, involving as it does comparisons between two different tables, with the potential for nonsequential reads. There are a variety of algorithms, depending on the state of the data as regards sorting. This ranges from the very basic brute force algorithm, with a scaling factor of $O(n^2)$ with the size of the data being examined, to more useful techniques, such as indexed nested loops, merge, sort/merge, and hash joins (Date, 1990). These are described below.

3.4.2.1 Nested Loop

At its simplest level, the nested loop joins is the $O(n^2)$ algorithm mentioned above. It takes a pair of join inputs (tables, or outputs from another operator), and designates one the outer, and one the inner input. The inner input is then scanned for matches once for each item in the outer. This approach guarantees that all matches are found, and is very practical for small datasets. It is also highly pipelineable: methods like sort/merge or hash joins are 'blocking', in that they have to perform a large proportion of their work before they can start to output results. Nested loops, on the other hand, can output results in an iterating manner, without a large upfront cost. This same property means that nested loops joins require very little memory, as they do not have to build up complete result sets prior to producing output. Despite these advantages, however, nested loops joins scale very poorly to the larger joins that one might expect to perform with RDF stores.

A substantial improvement upon the naive nested loops algorithm exists, in the index nested loops (INL) join. This join relies on an index being available on the inner join input, and consults it for matches against each row of the outer join input. This alternative is substantially faster: for each row of the outer join input, only a single operation has to be performed if there is no match, rather than iterating through the entire inner input.

INL joins require very little memory, and are especially effective for some situations: if the left join input is small and is being matched against a very large right join input, the selectivity of the index is brought to bear, returning only relevant results and thus reducing computation. They are particularly relevant to RDF storage, since RDF stores often have comprehensive indexing strategies.

Nested loop joins are faster than or comparably effective to hash joins in the general case, as long as an appropriate index is available. If one is not, hash joins are substantially more effective (DeWitt and Gerber, 1985), and are thus used more often in ad-hoc queries. Nested loops queries also parallelise highly effectively (Sheu and Thai, 1991). It should be noted, however, that in disk-based stores the looped accesses to the index will have a cost in terms of repeated disk seeks, compared to hash or sort/merge joins, which are less inherently selective, but allow more linear disk accesses. Seek time is a smaller issue in memory-backed stores, and index nested loops joins can be highly effective in this environment, benefitting especially from their low memory consumption.

3.4.2.2 Merge and Sort/Merge

Merge joins assume that both inputs are sorted in order on the columns that are being joined on. With this being the case, a simple scan of both inputs can perform a join

in linear time with the amount of data being joined, if the join is one to many, or near linear if it is many-many. Merge join is always faster than sort/merge or hash joins if data is sorted correctly. For this reason, query optimisers in an RDBMS will usually keep track of the sort order of the current working set of data, and will order joins to allow as much use of merge joining as possible.

Sort/Merge joins simply sort the inputs as required, and then perform a merge join on the resulting data. This approach is largely constrained by the performance of the sort.

3.4.2.3 Hash

A hash join performs a single scan over each input. It creates a hash table on the first input, with a pointer to the corresponding tuple. When scanning the second input, it compares against that hash table to produce the joined output. This technique scales in linear fashion with the amount of data scanned, and does not require inputs to be sorted to work efficiently (although, of course, sorting will ensure better contiguity of access and cache utilisation). It is, however, likely to be slower than merge join, since operations such as hashing require a degree of computational expense. Further, it is less tractable to hold all the intermediate data on disk if no memory is available.

3.4.3 Join Mechanisms in RDF Stores

As described in Section 3.2.2.1, the standard physical storage schema in RDF stores is a three index layout using SPO, POS, and OSP indexes, effectively covering all access paths into the system. Some dedicated stores, such as Jena TDB, use an exclusively index nested loops approach to joining data, while systems based on top of RDBMSs will use whatever join mechanism the system chooses at query time.

Some newer stores, specifically Hexastore (Weiss et al., 2008) and RDF-3X (Neumann and Weikum, 2008), have chosen a different approach. They create all six possible index permutations, allowing them to make the greatest possible use of merge joins, and where merging is not possible employ sort-merge or hash join strategies.

Each of the different join strategies can be illustrated using the query specified in Figure 3.12, a query that one might run over an RDF database of students studying in the UK.

To answer this query using INL, one might use the process outlined below:

1. Look up pattern 1 in the POS-ordered index, receving results for *?x* back. Bind next value of *?x*. If bindings are exhausted, query is complete.

```
PREFIX ex: <http://www.example.com/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?mealname WHERE {
    1 ?x ex:attends-university ex:Southampton .
    2 ?x ex:nationality ex:bangladesh .
    3 ?x ex:has-gender ex:male .
    4 ?x ex:likes-meal ?meal .
    5 ?meal rdfs:label ?mealname .
}
```

FIGURE 3.12: SPARQL query to determine the meals enjoyed by Bangladeshi students at the University of Southampton. Triple patterns are numbered for reference purposes.

2. All values for pattern 2 are now fixed. Look up in any index. Receive at most one result back. If no results are returned, return to pattern 1 and bind next value of *?x*.

3. All values for pattern 3 are now fixed. Look up in any index. Receive at most one result back. If no results are returned, return to pattern 1 and bind next value of *?x*.

4. Look up, in an SPO-ordered index, which meals the currently bound value of *?x* enjoys, receiving results for *?meal* back. If no results are returned, or results are exhausted, return to pattern 1 and bind next value of *?x*, else bind next value of *?meal*

5. All values for pattern 5 are now fixed. Look up in any index. Receive at most one result back. If no results are returned, bind the next value of pattern 4. Otherwise, output a result.

An alternative merge and sort-merge strategy might take a variety of approaches, depending on how the query was optimised. If the store attempts to minimise the number of sorts performed, it might merge join patterns 1 and 3. It would then merge join 2 and 4. These two outputs would then be merge joined together, and sorted on *?meal*. Finally, the working set would be joined against the output of query pattern 5.

The characteristics of each approach vary substantially. Consider the following (fabricated) figures:

1. There are 10,000 students at the University of Southampton.

2. There are 15,000 Bangladeshi students in the UK, of which Southampton has 500.

3. There are 1,000,000 male students in the UK.

4. Each student (out of 2 million in the UK) likes around 4 different meals.

5. There are 20,000,000 different labelled 'things' in the database.

If the query were run using INL joins, triple pattern 1 returns 10,000 results. Pattern 2 returns a result 1/20th of the time, while pattern 3 will return a result 1/2 of the time. Pattern 4 returns four results on average, and pattern 5 then returns one result at most. This query, then, will run pattern 1 once, pattern 2 10,000 times, pattern 3 500 times, pattern 4 250 times, and pattern 5 1000 times, giving a total of 11,751 lookups.

If this query were performed using merge and sort/merge joins, the join might involve the following steps (although one might find an alternative ordering to cut down result sets further):

1. Merge join patterns 1 and 3: iterate over 10,000 items on the left side, and up to 1,000,000 on the right, producing an output of 5000 elements. This output is already sorted on ?x, and does not need sorting again.
2. Merge join patterns 2 and 4: iterate over 15,000 elements on the left side, and 2,000,000 on the right, producing an output of 60,000 elements. This output is already sorted on ?x, and does not need sorting again.
3. Merge join the output of steps 1 and 2, outputting around 250 results. Sort this output on ?meal.
4. Merge join the output of step 3 against the 20,000,000 items produced by triple pattern 5.
5. Output results.

The advantage of the many-index approach used by Hexastore and RDF-3X is that they avoid having to do a lot of expensive sorts, as if the data has a sort order, they can find an index to make use of that sort order. The fact that the query described in Figure 3.12 has very few variables is helpful in this regard: most of the time, the working set remains sorted on *?x*. Overall, this approach has been conclusively shown to provide significantly better performance than using fewer indexes with a merge/sort-merge strategy (Weiss et al., 2008).

No comparisons, however, have been made between the merge join strategy and the use of index nested loops. It's possible to see from this example that INL does a better job of eliminating irrelevant data, thanks to its superior selectivity: the merge join example has to iterate over many millions of items, which INL ignores by virtue of substituting additional bindings into index retrievals. The factor working against INL is that while it touches much less data, it performs a lot more random access. If that random access is cheap, INL will be much faster. If the random access is very expensive, merge and sort/merge will become faster.

Overall, then, memory-based or heavily cached disk-based systems are likely to benefit from an INL approach (particularly given its low memory usage), while systems without such benefits are likely to do better with merge/sort-merge or merge/hash join strategies. In future, for disk-based stores using INL joins, it may be worth investigating data structures like compact, in-memory Bloom filters (Bloom, 1970) that would inform whether a disk query would return any results or not, eliminating the large majority of disk seeks that would return no results. In this example, assuming a 1% error rate and no caching of information, the number of lookups for the INL approach would drop from 11,750 to 1,976. In practice, that number of seeks would usually be further reduced by the influence of cached data.

3.4.4 Join Minimisation

As previously noted, reducing time spent in joins is an excellent method for improving overall RDF store performance. One method for achieving this is to perform the work in advance. Abadi et al. (2007) describes the concept of 'materialised path expressions', in essence the process of pre-calculating joins such that they do not have to be performed at run time. The authors note that this can afford an orders of magnitude level improvement in performance on suitable queries.

Join precalculation is generally very attractive for read optimised disk-based systems. If a given join is performed regularly, a great deal of time can be saved by storing the completed join on disk. There are, however, a variety of complications to this approach. The precalculated data needs to be updated every time a related piece of information is added or removed, which can be expensive. In addition, it is necessary to determine what precalculated information would actually offer a significant benefit, which can be a complex process. Doing this work manually would be difficult, so it becomes necessary to maintain accurate usage statistics (or batch-processable logs) to allow the determination of what joins should be precalculated. Finally, precalculated joins are clearly not suitable for systems where storage space is very limited.

3.4.5 Summary

RDF offers a somewhat unusual problem with regards to operator implementation: its large triple tables (or, indeed, property tables) and the high likelihood of appropriate indexes being available means that the choice between INL and merge join systems is not completely clear for disk-backed systems. For memory-backed systems, it is likely that INL will have a significant advantage due to the lower cost of seeking.

Thanks to the sheer quantity of data points in a typical RDF store, RDF does require a special emphasis on minimising the time spent in joins. This can be achieved by

methods such as intelligent query optimisation, and join precalculation. The former is important for ordering queries appropriately, such that the working set stays as small as possible. This is a challenging problem, relying on high quality statistics to estimate the size of each data retrieval, and by extension the effect on the working set. Some systems, like RDF-3X (Neumann and Weikum, 2008), generate exhaustive statistics over their datasets, while others, such as Virtuoso, rely on estimation to save space (Erling, 2009). Since it is expensive to generate high quality statistics about RDF data thanks to its large quantity of data points, there is room for research in this area.

Join precalculation is clearly attractive for the large corpus, read-mostly use case. There is a clear need to be able to determine what precalculation is necessary, which again offers an opening for new work in the area.

3.5 Scaling to Extremely Large Systems Through Distribution

The most powerful single machine RDF stores are currently capable of storing up to around two billion triples[2]. Clearly, it is possible to use more expensive, more powerful machines to improve scalability and response times. Unfortunately, buying ever-faster machines yields diminishing returns as one escapes the commodity market. To realise practical, large scale improvements it is necessary to allow RDF stores to make use of the power of multiple machines. Traditional DBMSs underwent a similar evolution, as ever-increasing dataset sizes required the development of DBMSs with better scaling characteristics, and this research is of interest in the creation of a highly scalable RDF store.

When considering the distribution of RDF stores, it is important to draw the distinction between 'federated' and 'clustered' stores. A clustered store is, to all outside appearances, a single system: there is only one point of query, and no guarantee that any single system within the cluster will hold meaningful data. By contrast, a federated store is a system that amalgamates several existing stores, each one of which can be individually and meaningfully queried. One might desire this approach for (for example), providing the ability to query all museums in the UK about what artefacts they hold from a particular period of time. In this situation, each museum will have its own store, and will want to control its own data, but may be willing to share it such that it can be accessed from a federated system. A compromise between these two approaches is found in many peer to peer stores, which are able to ask other systems to store data, without necessarily requiring it (Battre et al., 2006; Heine et al., 2005; Nejdl et al., 2003; Cai and Frank, 2004).

[2] http://esw.w3.org/topic/LargeTripleStores

The differences between these the federated and clustered paradigms is significant from the point of view of performance. In a clustered system, the DBMS has the freedom to place data wherever it wishes, making it possible to distribute data based on a known function. This makes it possible to know trivially where a given datum will be located. In a federated system, it is necessary to either record where information is located, or have some kind of discovery mechanism. In either case, this has a serious impact on performance: there is no way to control data placement such that it is optimally located, and finding information has an additional cost in space and/or time. Federated systems are not considered in this section, as it is focused on using multiple machines to the end of improving the performance of an individual store.

The desired performance improvements in distributed DBMSs can be categorised as follows (Boral et al., 1990; DeWitt and Gray, 1992):

- **Scaleup**: An increase in the number of machines leads to the ability to store more data.
- **Speedup**: An increase in the number of machines leads to a reduction in the amount of time taken to serve an individual query, all other factors being equal.
- **Throughput Scaleup**: An increase in the number of machines leads to the ability to perform more transactions in a given time frame.

While ideally both speedup and scaleup will be linear with the amount of processing power available, this is a practical impossibility in any database system: some operations (such as sort) do not scale in linear time. There are other significant barriers to such a perfect level of system scalability (DeWitt and Gray, 1992):

- **Startup**: the time needed to start a parallel operation - if a small operation results in lots of processes being started across a lot of nodes, the cost of startup can overwhelm any advantages gained through increased parallelism.
- **Interference**: The slowdown each new process creates when accessing shared resources.
- **Skew**: The effect where one part of a parallelised operation takes much longer to complete than the others: since the job is limited by the slowest process, this can seriously affect performance.

A variety of hardware architectures have been utilised to create parallel database systems. These can be broadly grouped into three categories: shared memory (SM), shared disk (SD) and shared nothing (SN) (Stonebraker, 1986). In SM systems all processors

share a common central memory, in SD they have a private memory but a common collection of disks, and in SN they share only the ability to communicate with each other via messages over a network.

Generally speaking, shared nothing systems are favoured today for their excellent characteristics with regards to resource contention: the only shared resource is network access, and there is no need for the complex resource locking methods seen in SM and SD systems. This means that scaling up SN clusters has historically been easier than the alternatives (DeWitt and Gray, 1992; Stonebraker, 1986). Further, SN clusters can be built out of commodity parts, as used by companies like Google (Brin and Page, 1998), offering an excellent price/performance profile. It should be noted, however, that today's multi-processor/multi-core designs effectively create a SM system on each machine in a cluster, meaning that the complexities of shared memory systems are still relevant to the design of today's database systems.

The disadvantage of the SN approach is that there is greater complexity in deciding where data is placed: it is important to place data such that each machine undergoes a similar load profile to enable efficient scaling, and does not require excessive use of network resources. Ongoing maintenance (whether manual or automatic) to the distribution of data is necessary to prevent 'hot spots', or points at which data or query skew has caused a machine to have too high a workload. When these hot spots occur, they can usually be eliminated by redistribution of data on the machine.

3.5.1 Enabling Parallelism

Parallel execution can be enabled through a variety of strategies. Most obviously, it is possible to partition (or decluster) information across more than one machine, such that the time required to retrieve a large block of data is reduced, and the number of processes that can retrieve data at any one time (assuming they are not both trying to access the same data) is also increased (DeWitt and Gray, 1992; Mehta and DeWitt, 1997). It should be noted that when reading or writing small amounts of data, it is desirable to perform the work on as few machines as possible. This is because the setup costs will dwarf any advantages gained from partitioning (Khan et al., 1999). Section 3.5.2 considers the problem of how to decluster data in more detail.

Another means of parallelising database systems is to cluster the execution of relational operations, so that for a given operation (such as a join) each machine processes a defined range of data values out of an overall dataset. This prevents one machine from doing all the processing work and becoming a bottleneck (Boral et al., 1990).

Pipelining of operations can also provide a performance boost: many relational operators do not need to complete before they start emitting results. In this sense they can be viewed as a stream. The output of this stream can be directed to other operations,

which can start processing them in parallel with the first operation. The benefits of this approach are somewhat limited, however: pipelines are terminated by the presence of an operation (such as a sort) that cannot emit results until it is complete, rendering most pipelines relatively short (DeWitt and Gray, 1992). Further, some operations take much longer than others (an example of skew), thus causing some machines to have to undertake much more work than others.

Finally, parallelism is supported by simply allowing multiple users to access a system, and allowing the subqueries that form an individual query to run in parallel. This is enabled by the likelihood that different users and subqueries will be accessing different pieces of information, so hardware resources can be shared between them and the queries run in parallel. Multi-user systems can, however, exhibit greatly increased complexity with regards to transactional behaviour and resource locking, depending on the behavioural guarantees that are required.

These mechanisms for enabling parallelism can be characterised as occurring at three levels (Khan et al., 1999)

- **Inter-query**: The ability to run more than one query simultaneously.
- **Intra-query**: The ability to run different subqueries in parallel and pipeline operations.
- **Intra-operation**: Distributing single operations over more than one node for concurrent execution.

3.5.2 Data Partitioning

A standard approach to partitioning data in an RDBMS is horizontally partitioning (or declustering) each relation in the system. In these systems, tuples of each relation in the database are partitioned across the storage of each processing node on the network, allowing multiple machines to scan a relation in parallel. It also addresses hotspot issues, as the contents of regularly accessed relations are spread across multiple machines, and more can be added as necessary.

DeWitt and Gray (1992) describes methods for horizontal partitioning of data, dividing them into three common techniques:

- Round Robin: simply distributing the tuples in a round robin fashion to each server. This approach works well for sequential scans, but is inefficient if there is a desire to access tuples based on attribute values, since the location of a given tuple is unknown.

- Hash Partitioning: distribution of tuples by applying a hash function to an attribute value. The function emits a number which specifies a machine (and possibly location on that machine) on which to store the information. This approach is effective if tuples are accessed based on a fully specified attribute, but is much less effective for range queries: hashing does not do a good job of clustering related data. Further, hash partitioning suffers from difficulties with the addition of new machines to a cluster, and addressing hot spots: in a naive implementation it is not possible to repartition data.
- Range Partitioning: distribution of tuples by selecting a range over one attribute. For example, all tuples with a value of 'surname' between A-C go on one partition, D-E on another, and so on. This approach clusters data effectively. The major issue with this is that it risks both data and execution skew: one part of the range may have a disproportionately large quantity of the actual data, and one part of the cluster may get accessed much more frequently than others (this being particularly likely if it has to store more of the data).

Partitioning improves the response time of sequential scans, because more processors and I/O resources (disks or memory) are used to perform the scan. It aids associative scans (scanning based on an attribute value) because the number of tuples stored at each node is reduced, and hence index sizes are reduced. In the case of RDF, scans are usually associative thanks to the comprehensive indexing strategies employed by most stores.

It is important to decluster data in a manner appropriate to both the dataset itself, and the manner in which it will be accessed. In particular, the following factors have a significant influence:

- Degree of declustering: it is important to decluster to an appropriate extent. If a very small relation is partitioned over a very large number of machines, startup costs and overheads (such as disk seeks) will overwhelm any advantages gained from parallelism, as well as making poor use of resources. In practice, parallel systems such as Bubba (Boral, 1988; Boral et al., 1990) have found that full declustering is often inappropriate.
- Skew: it is important to ensure that each machine undergoes a comparable workload. A simple implementation will balance the quantity of information stored on each server, but it is also important to take into account the possibility that certain data ranges will be accessed much more regularly than others, creating an excessive load on some servers. This type of skew (execution skew) can be countered by balancing data distribution not by the volume stored on each machine in the cluster, but by the frequency with which each machine has to access data, particularly that which is uncached (Boral et al., 1990).

- Declustering attribute: it is necessary to partition on an appropriate attribute: the location of tuples is only known, if it is known at all, based on a function of that attribute. Queries that reference a relation based on a different attribute have to be flooded to all machines that store a portion of the relevant relation (Hua and Lee, 1990). This presents no barriers in a store with comprehensive indexing, since each index can be distributed based on its primary attribute, but is of interest when considering other strategies.

3.5.3 Distributing RDF Stores

RDF stores offer a few elements of special case behaviour with regards to distribution. Conveniently, the tendency of single system RDF stores to utilise quite a complete level of indexing is advantageous: each one of the SPO, POS, and OSP indexes can be distributed based on their subject, predicate, and object respectively, eliminating the issue of choosing a declustering attribute (Hua and Lee, 1990), and meaning that triples can be easily discovered whatever portion of the triple is supplied. Bitmap and SFC indexes also distribute effectively: since they index into a single attribute (the line number for bitmap indexes) there only needs to be one data ordering. These indexes do not guarantee that logically related data will be located on the same machine, however.

Perhaps the most significant issue when considering the clustering of RDF storage is data distribution. Generally speaking, there is usually a relatively even distribution of subjects, each subject being used a relatively small number of times. This makes it advisable to keep all data on a subject in an SPO index on a single machine, as startup costs will remove any gains from increased parallelisation. Predicates (and some objects), on the other hand, tend to be of much higher cardinality, potentially resulting in individual machines having to do excessive amounts of work and becoming hot spots. This is exhibited to an extreme extent in predicates such as rdfs:label, which is often used extremely regularly, and can result in certain machines storing very large portions of the POS index. YARS2, one of the few existing clustered triple stores, works around this problem by distributing predicate-ordered entries to a random server, and flooding all predicate-oriented queries to every server in the system (Harth et al., 2007). This approach is overly simplistic, removing contiguity of access and making querying against lower cardinality predicates (and particularly predicate-object pairings) unnecessarily expensive.

In Owens et al. (2008) the author proposed an alternative mechanism for dealing with these hot spots: an 'exception list' that stores exceptions to the usual rules, distributed to every machine in the cluster. Since the number of outliers are by definition relatively small, this list requires only a small amount of memory. This allows low-medium cardinality properties and objects to be stored as normal on a single machine, high cardinality ones to be stored over a subset of the cluster, and extreme cases such as rdfs:label to

be stored over the entire cluster. The latter two cases could have their distribution performed based on two attributes (such as P and O), so that queries that supply two attribute values can still hit only one machine, while single attribute requests gain the benefit of parallelisation. This approach should logically provide improved results.

Aside from these issues, RDF stores usually distribute effectively using traditional techniques. Since range searches are rendered moot in RDF query due to the standard normalised model, distribution based on a hash function is ideal. The main issue with hash-based distribution, that it does not provide room for the addition and removal of machines from the cluster, is easily solved (Erling and Mikhailov, 2008). If one pretends that a cluster has several thousand machines, one can assign several of these virtual machines to each physical server. Each server in the cluster holds a small amount of information describing where the virtual machines are located, and store and retrieve commands are subsequently performed on the virtual machines. Virtual machines can then be moved between servers at will. This process can create some issues with maintaining locality when required, but this can be overcome, as described in Owens et al. (2008).

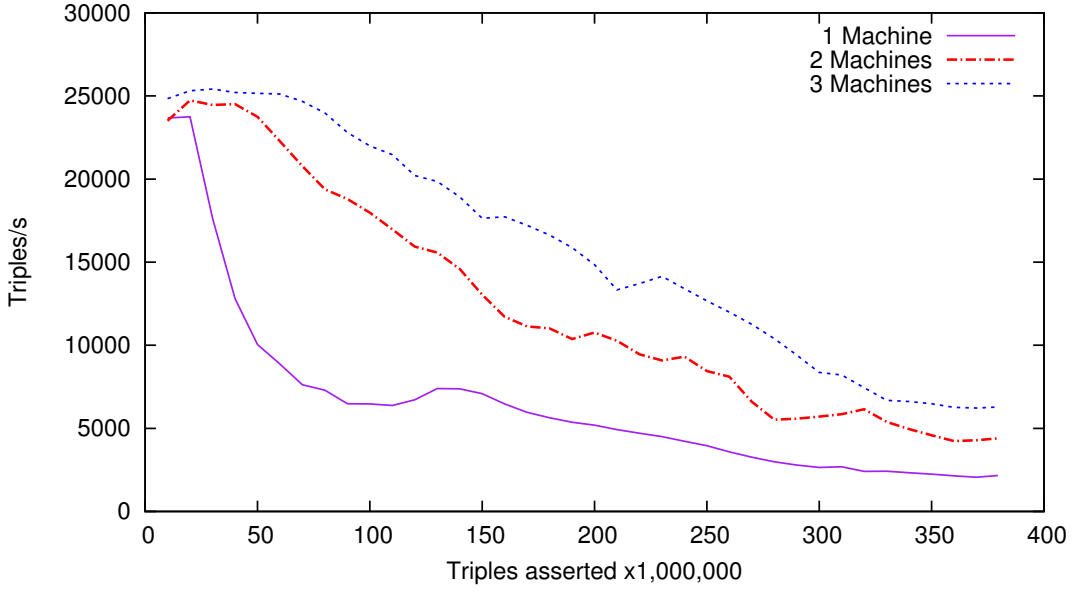


FIGURE 3.13: Rates of assertion during a Clustered TDB load (Owens et al., 2008)

Figure 3.13 shows the scaling with regards to assertion rates for 1, 2, and 3 machine clusters for the work performed by the author in Owens et al. (2008). As this figure indicates, assertion time for large RDF files scales excellently using a hash partitioning approach. Query performance depends on the types of queries being performed: some, such as those that involve large index nested loops joins provide excellent opportunity for parallelisation, while others offer more limited benefit. In systems such as this that normalise URIs and literals into unique IDs, the process of converting IDs to URIs also affords excellent parallelisation opportunities.

3.5.3.1 Distributing Memory Stores

There are no memory-only clustered RDF stores currently in existence. Distribution in this scenario has quite different requirements to a disk based environment: in the latter case, the latency of the network is usually much lower than that of a disk, so the latency is often hidden quite effectively. While it is generally beneficial to perform as little network transfer as possible, it is not excessively expensive to do so. By contrast, in a memory-only or heavily memory-based scenario, the cost of network access is substantial compared to an access to RAM (Erling and Mikhailov, 2008). Given this fact, it becomes more important to reduce network accesses wherever possible. This may require compromises such as globally cached data, which impedes scalability, or avoidance of parallelisation-enabling techniques such as index nested loops joins in favour of techniques which require fewer round trips. Alternative strategies such as modifying index nested loops joins to buffer multiple operations at a time might also prove fruitful.

In practise, Ousterhout et al. (2010) argues that it is possible to substantially reduce the latencies experienced by current networks, from several hundred microseconds to between 5 and 10, making them more practical for memory (or SSD)-oriented designs. This could be accomplished through a combination of improved practices:

- Lower latency network switches. Standard switches introduce substantial latency: around 100μs for a round trip across a typical network. Newer designs can be used to cut this by a factor of ten or more. Alternative network architectures such as Myrinet (Boden et al., 1995) and Infiniband (InfiniBand Trade Association, 2001) can also offer substantially reduced delays: Infiniband, for example, claims to offer microsecond-level latency.
- Reductions in the overheads imposed by general-purpose operating systems in socket communication. Current systems rely on passing through O/S layers to communicate information to an application. Ousterhout et al. (2010) argues that it will be possible to substantially reduce latency by dedicating a processor core to polling for data and performing basic packet processing, or allowing network cards to map areas in application memory, and directly pass in data.
- Alterations to the standard TCP protocol for intra-datacentre communication, such as reductions in retransmission windows.
- Increased bandwidth at the more contended parts of the network, in order to reduce the need for retransmission.

3.5.4 Summary

This section presented a brief summary of clustering in RDF stores and other DBMSs, including the author's own work in the area. While mid-sized datasets can be reasonably stored on a single machine, realisation of extremely large improvements in scalability will inevitably require a move towards clustered stores, the background of which was described in this section. This trend is evident in the relatively recent release of 4store (Harris et al.), Virtuoso Cluster (Erling and Mikhailov, 2008), YARS2 (Harth et al., 2007), and the author's own work described in this chapter.

In general, RDF distributes fairly efficiently using existing techniques, and the issues that do exist can be overcome: Section 3.5.3 describes the author's work in this area. Interesting research opportunities arise in the event of low latency storage such as main memory or SSDs becoming popular (Lee et al., 2009). Currently, the latency cost of accessing data over the network is not excessive in comparison to the cost of disk I/O, but this will change with low latency storage. Even when measures such as the installation of low-latency network switches are taken, it will become more critical to locally cache regularly accessed data, increasing update complexity and compromising linear scalability in the aid of better absolute performance.

3.6 Summary of Existing RDF Stores

As a whole, this thesis focusses on improving the various subsystems that combine to form an RDF store, particularly data storage structures. As a result, this literature review has considered existing RDF stores in a piecemeal fashion, rather than comparing them as monolithic entities. In order to provide an overview of how the techniques described in this review are used in real world stores, this section contains a brief summary of the properties of several popular existing systems.

Store	**Backing Storage**	**Join Mechanism**	**Triple Data Structures**
3Store (Harris, 2005)	RDBMS (Disk)	Mixed (DBMS-specified)	Standard RDBMS quad table with user-specifiable indexes (usually B+Trees).
4Store (Harris et al.)	RAM	INL	Distributed. Quads indexed using radix tries with a 4-bit radix.

continued on next page

Store	**Backing Storage**	**Join Mechanism**	**Triple Data Structures**
C-Store (Vertical Partitioning) (Abadi et al., 2007)	RDBMS (Disk)	Mixed (DBMS-specified)	Fully sorted columnar RDBMS property tables with B+Tree indexes over SO, and, optionally, OS (batch-updateable).
Hexastore (Weiss et al., 2008)	Memory, Disk	Merge, sort-merge	Six custom triple indexes. In an SPO-ordered index, for example, a sorted vector of Subjects links to associated sorted vectors of Predicates, which link to associated Object vectors (read only).
Jena Tuple Database (TDB)	Disk	INL	B+Trees over SPO, POS, and OSP.
Jena Memory Model (JMM) (Carroll et al., 2004)	RAM	INL	Custom single-attribute hash-based indexes (See Section 4.4).
Kowari (Wood et al., 2005)	Disk	INL	Wide node AVL trees (see Section 3.3.1).
RDF-3X (Neumann and Weikum, 2008)	Disk	Merge, hash	Delta-compressed B+Trees (read only).
Sesame 2 Native	Disk	Mixed	B+Trees over SPOG, POSG, additional indexes configurable.
SwiftOWLIM (Ognyanoff et al., 2007)	RAM	Unknown	Hash indexes.
Virtuoso (Erling, 2006)	Disk	Mixed	Triple table with user-configurable indexes (B+Tree and custom bitmap. See Section 3.3.4).

continued on next page

Store	Backing Storage	Join Mechanism	Triple Data Structures
YARS2 (Harth et al., 2007)	Disk	INL	Six custom, hash distributed 'sparse' indexes. Triples stored in fully sorted, huffman compressed lists divided into blocks. Each block is represented by an entry in memory that is used to determine the block in which a triple is located (read only).

TABLE 3.2: Summary of current RDF stores

3.7 Opportunities

There are a variety of opportunities for research in the areas described in this chapter. Valuable contributions can be made by minimising the time spent in joins through improved query optimisation, or work on precalculated joins, and there is certainly scope for improving the deletion performance of stores that perform inference.

Other opportunities largely center around the upcoming availability of low latency storage, a growing trend in the computing industry. RAM is becoming significantly cheaper, with 32-64GB machines now fairly commonplace. In addition, solid state disks (SSDs) are becoming increasingly common and practical.

Low latency storage can have a very significant impact on overall performance. In a disk based environment, the cost of disk seeks is by far the largest cause of waiting under many circumstances. A simple example illustrates this: in an uncached system based on B+trees, a billion triple index might have a tree height of 5. If an index nested loops join is performed that joins over just 10 items in the outer input, and a disk seek takes 10ms, the minimum I/O latency for that one operation is 500ms. Now, clearly a realistic system will cache most of the upper levels of the tree, but even if all but the final level is cached, the minimum I/O latency for the join operation is 100ms.

A typical SSD might have a random read latency of less than 0.1ms. The same join on that hardware would have an I/O latency of just 1ms: a very substantial improvement. Main memory is, of course, much quicker again. Since SPARQL queries are very join-heavy, this is excellent news from the point of view of performance.

Lower I/O latency changes the focus of research. In a disk bound environment, poor cpu cache utilisation or poor branching performance is likely to be overlapped to some extent by disk latency, and in any case can generally be considered much less significant by comparison. In an environment with low I/O latency, less overlapping is likely,

and more efficient use of CPU and memory will produce a relatively much larger gain. With their small datum size, RDF stores have a particular opportunity to benefit from improved cache locality.

Systems with relatively low I/O latency enable a greater variety of strategies for physical representation. Indexes based on SFCs, for example, become more practical in an environment with lower seek times. RDF-specific index structures become an interesting research area in a memory based environment that does not mandate the use of large blocks. Such index structures become particularly attractive since SSDs and RAM are typically significantly more expensive per gigabyte, placing an emphasis on space consumption.

In general, the author sees the increasing practicality of low latency storage as being perhaps the most important recent development in RDF storage, yielding a variety of new research opportunities, particularly in the area of physical storage schemas. To that end, this thesis focuses on the creation of memory-based index structures for RDF storage.

Chapter 4

Java as a DBMS language

Existing Semantic Web toolkits (for example, Jena (Carroll et al., 2004) and Sesame (Broekstra et al., 2002)) largely make use of Java and related technologies. Despite this, Java is not traditionally seen as a language suitable for the development of high performance database systems. This chapter contends that the prevailing attitude towards the suitability of Java is out of date, and discusses the implications of its use in the development of RDF storage systems.

The Java programming language's reputation for poor performance exists thanks largely to the immature technology behind early Java Virtual Machines (JVMs). In reality, however, modern JVMs have seen a host of advances such as Just In Time (JIT) compilation (Adl-Tabatabai et al., 1998), generational garbage collection (Sun Microsystems, 2006), aggressive inlining, escape analysis (Kotzmann et al., 2008), synchronisation performance enhancements (Russell and Detlefs, 2006), array bounds check elimination (Würthinger et al., 2007) and many more. These improvements have allowed JVMs to approach the time performance of compiled C or C++ across recent benchmarks within a factor of less than 2 (Fulgham and Gouy), and improve upon them in some tests. Java has thus become increasingly useful for performance-dependent applications.

A complete discussion of the JVM's compiler optimisations is beyond the scope of this chapter, but can be found in Kotzmann et al. (2008) and Sun Microsystems (2008). The observations made in this chapter are based on the Sun reference JVM running on an x86 architecture, but the information given is valid for most other systems as well.

The maturity of Java and its backing technologies has resulted in its being used in more efficiency-dependent systems. Hadoop (White, 2009; Borthakur, 2007), for example, is a Map-Reduce (Dean and Ghemawat, 2004) implementation designed for processing extremely large datasets. The two most popular Semantic Web frameworks, Jena and Sesame, are also implemented in Java. Noting the extent to which Java is used in Semantic Web technologies, this chapter performs a further investigation into this technology,

in order to determine whether it is suitable language choice for an implementation of the prototype data structure described in Chapter 6, and if it has unusual features that may help or hinder RDF storage.

4.1 Time Performance

Java is a virtual machine-based language, which can result in different behaviour compared to a compiled application running on the bare hardware. While Java has the overhead of dynamic code compilation and garbage collection, there are associated performance benefits, such as the extremely cheap memory allocation that garbage collection provides (Blackburn et al., 2004), and the ability to dynamically recompile code when it might be beneficial.

An example of this can be seen in some brief experimentation performed by the author as a test of branch prediction and cache performance. When performing a binary chop of varying predictability over an extremely large array, as illustrated in Table 4.1 and Table 4.1, Java exhibited much better worst case performance and significantly worse best case performance than a similar C implementation. Note that these results were experienced on Sun's reference implementation under Linux, and different implementations may yield different results. The code for these implementations, along with the compilation flags, can be found in Appendix A. The machine upon which the tests were performed is described in Appendix B.

Array Size (ints)	**Java (ms)**	**C (ms)**
150000000	44106	67540
15000000	17963	20660
1500000	9293	9610
150000	2514	2200

Table 4.1: Comparison of Java and C on an unpredictable large scale binary chop

Array Size (ints)	**Java (ms)**	**C (ms)**
150000000	1723	1270
15000000	1481	1090
1500000	1308	960
150000	1123	820

Table 4.2: Comparison of Java and C on a predictable large scale binary chop

4.2 Memory Efficiency

While Java's time performance is often broadly comparable to more traditionally performant languages, its space efficiency is often significantly worse: an issue that is important for data-centric programs like DBMSs. Benchmarks (Fulgham and Gouy) indicate that it is not unusual for Java applications to use 3 or more times the amount of memory of a traditional C or C++ program, even on large datasets where the proportion required for the JVM application is relatively small. There are a variety of causes for this:

- Baseline cost of the JVM application.
- Overhead of storing instrumentation data to allow the JVM to perform runtime optimisation.
- Overhead associated with objects.
- Overhead of garbage collection and memory model.

The first two issues are important for small applications, but are respectively constant and related to code size, and so are relatively insignificant for data-centric applications like DBMSs. The latter two issues scale up with the size of the dataset, and are thus relevant to this discussion.

The optimisation of small objects is an issue of great significance in JVM design. There is a cost associated with managing each object with respect to both space and time efficiency, and a large number of small objects exacerbates this overhead. These costs are considered in detail in this section, as well as in Section 4.3.

Each RDF triple encodes only a very small amount of information: a single relationship. As a result of this, an RDF document that expresses a meaningful amount of data typically requires a large number of triples. To represent this data, a natural object model will create an object for each RDF triple, with each URI or literal within the triple also represented by objects. The practical upshot of this is that current memory based RDF stores (as described in Section 4.4) produce a large number of objects, each of which contains very little data. While this is a natural fit to the RDF model, there is a minimum space overhead associated with each object created: in most JVMs, this amounts to two words (Bacon et al., 2002). This space is required to maintain class information and data related to synchronisation and garbage collection. In addition, on x86 architectures objects are usually aligned to word boundaries, according to standard practice on those systems.

On a 64-bit system, then, the per-object overhead is between 16 and 23 bytes, which in a small object-heavy system can lead to a large amount of wasted space. Further, poor

space utilisation can have performance implications: inefficient use of space can lead to poor utilisation of CPU caches, which as noted in Section 3.1.3.2 is an increasingly important issue in modern computers (Appel and Palsberg, 2002). Figure 4.1 gives an illustration of the memory use of Java objects.

A particular matter of interest for modern JVM implementations is the expense of small strings, of which most RDF DBMSs store many. There are two sources of potential overhead in Java's string implementation: firstly, string objects in Java store a variety of additional information, such as a cached hash code. Secondly, Java uses a fixed-width, 2 byte character type, using UCS-2 encoding. This encoding supports a wide range of Unicode characters, but for documents largely restricted to an 8 bit range, as is the case for a large proportion of documents outside of Asia, the extra space is wasted, and a variable length encoding would be more compact.

The TIJmp[1] Java heap profiling tool was used to perform a simple measurement of the space required by 1.5 million 20 character String objects. This test indicated that each String had a total overhead of 64 bytes on 40 bytes worth of data: enough space to store 32 more UCS-2 characters. This space could store 104 8-bit characters. The per-string overhead grows proportionately less as the size of the string increases.

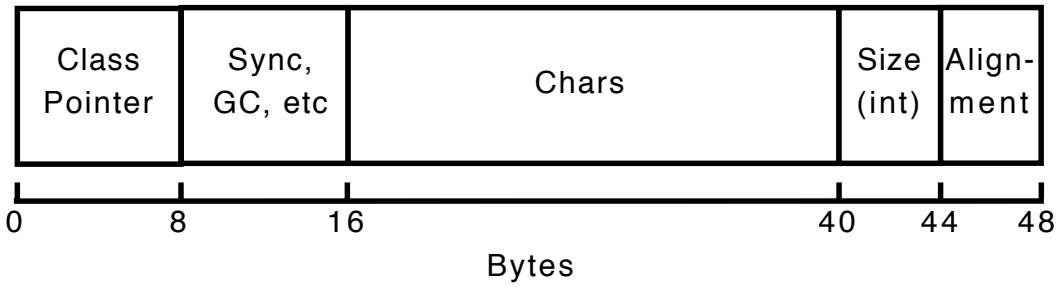


FIGURE 4.1: Memory usage of a Java object. This diagram illustrates an array of 12 characters. Note that characters are two bytes wide in Java in order to support Unicode.

4.3 Garbage Collection

Early garbage collectors had difficulty managing large number of objects (Ungar, 1984). This issue has, however, been effectively mitigated in more recent JVMs. Generational garbage collectors (Lieberman and Hewitt, 1983), as shown in Figure 4.2 split the heap into age-based generations. In the Sun reference JVM, allocations are performed sequentially in the 'eden' of the 'young' generation, and are promoted to the 'survivor' space of the young generation upon surviving a garbage collection (Sun Microsystems, 2006). This promotion works by copying the live objects in the eden to the end of the survivor space. There are effectively two survivor spaces allocated: each time garbage

[1] http://www.khelekore.org/jmp/tijmp/

collection is performed, all the live objects in the young generation are copied to the other space (Blackburn et al., 2004). This process effectively compacts information, eliminating fragmentation. After a given number of collections, objects are promoted to the 'tenured' generation, where they will stay until they are deallocated. Finally, there is a 'permanent' generation for information that will be needed for the lifetime of an application, such as class definitions.

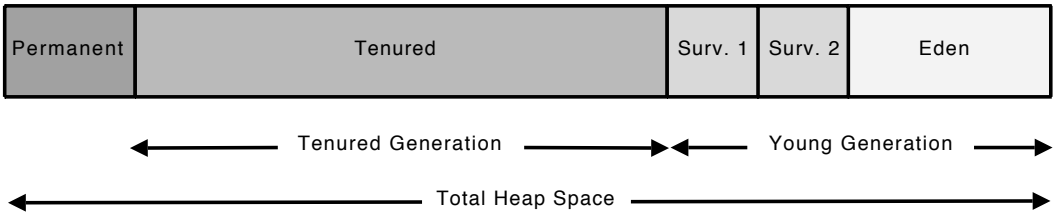


FIGURE 4.2: Generational memory layout in the Sun JVM. Older generations are indicated by darker shades.

Generational collectors are based on the observation that most objects created in a typical system are very short lived, while some live for an extremely long time (Lieberman and Hewitt, 1983). Collections work by determining what objects are still alive, and so there is effectively no cost for object deallocation; instead, there is a cost for each object that remains alive. They are performed much more regularly on the young generation, and since this is typically small, with a large proportion of deallocated objects, take relatively little time (Sun Microsystems, 2006). Collections on the tenured generation are relatively rare, but take much more time as a result of working on a much larger set of objects, and having to perform in-place compaction.

A consequence of this memory allocation system is that allocation is extremely cheap: in contrast to a typical 'malloc' system, all allocations are performed sequentially at a known location. The cost of garbage collection is variable, depending on how many live objects remain in the system. This means that while large object counts are less costly when compared to earlier technologies, there is still some time overhead associated with them. In the Jena Memory Model, for example, a full collection on a 1.43 million triple dataset, using the 'throughput' collector on a dual 1.8 GHz Opteron system requires as much as 15 seconds, during which the system cannot respond to queries. Fortunately, such a collection is very rare unless there are a large number of modifications being made to the dataset.

In a scenario where a very high number of alterations are not expected, Java's garbage collection is a good fit for in-memory DBMSs. The space overhead is not excessive: the young generation (which wastes half of its allocated space at any one time) need only be relatively small, as most data in a DBMSs is permanently stored. For a mode of use that results in large numbers of updates, alternative garbage collectors can be considered that aim to reduce latency, at the cost of higher memory requirements (Sun Microsystems, 2006).

4.4 Profiling an In-Memory, Java-based RDF Store

This section discusses the performance of the Jena Memory Model, an in-memory store written in Java, the design of which is discussed in Section 4.4.1. It contributes an evaluation featuring profiling on issues such as memory use, CPU time, and cache miss rates, evaluated in the context of the requirements that modern processors, memory subsystems, and Java itself place upon the design of RDF stores.

This section uses a Berlin SPARQL Benchmark (BSBM) dataset of 1.43 million triples, generated using a scale factor of 4000. The pieces of software used for measuring performance were OProfile[2] for CPU and cache profiling, and Netbeans[3] and TIJmp[4] for memory analysis. The hardware upon which the tests were performed is described in Appendix B.

4.4.1 The design of the Jena Memory Model

The Jena Memory Model (JMM) is designed to be an efficient fit to the RDF data model. It stores each RDF statement in three separate hash indexes, based respectively on Subject, Predicate and Object. This allows rapid lookup of triples based on partial match criteria, a necessity for efficient SPARQL query processing: given, for example, the Subject 'Alisdair', it becomes easy to find all triples with that subject.

Each index is comprised of a hash table that maps from nodes (URIs, B-Nodes, or literals) to Bunches (a set of RDF statements corresponding to that node). A Bunch is either an array of size 5 or 9 statements, or an open addressed hash table for instances where more statements are present. The manner in which Jena internally represents a small dataset is illustrated in Figure 4.3.

At the top level, Jena's hash-based indexes make sense for RDF storage. Their property of O(1) addition and lookup is highly desirable in a system that is expected to scale to large quantities of data. Their traditional weakness, the lack of sorted order, is less of an issue when storing RDF than it is with most other forms of data: URIs are discrete concepts to which sorting has little meaning, and so searches over an alphabetic range of URIs are less likely.

Jena's adaptive Bunch structure allows the index to grow in a manner that promotes high performance: the arrays used in smaller Bunches allow for fast linear traversal and compact representation, while the hash sets enable cheap duplicate detection as the Bunch grows, promoting fast assertion. Effectively, in the context of a POS-ordered index, the JMM is very fast for restricting over P, and for restricting over POS: the

[2] http://oprofile.sourceforge.net/
[3] http://www.netbeans.org/
[4] http://www.khelekore.org/jmp/tijmp/

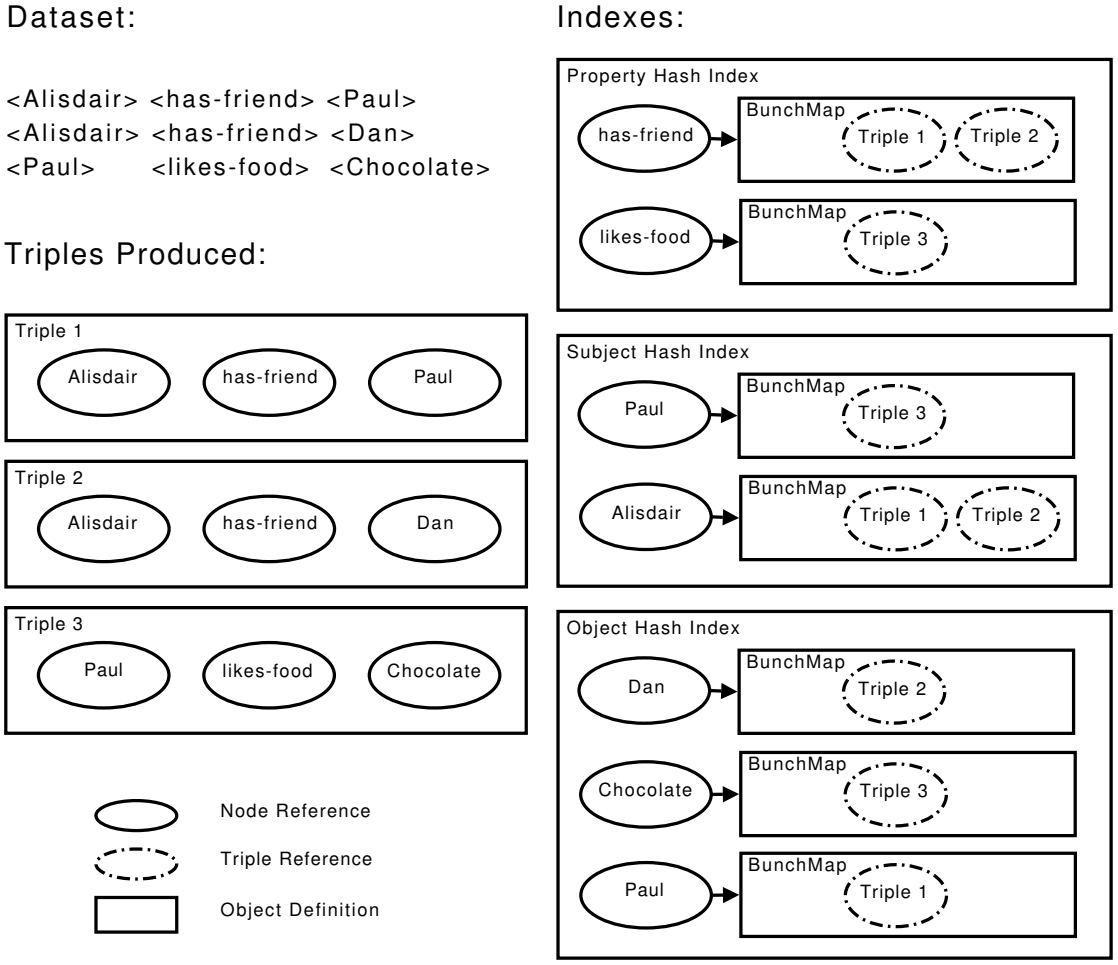


FIGURE 4.3: The Jena Memory Model: representing a small dataset

Hash Bunches provide a fast 'contains' check, while the Array Bunches are small enough to simply iterate over. Unfortunately, it has no indexing mechanism for restricting over PO, which can cause significant issues with performance.

While conceptually this design largely fits the RDF data model, its space efficiency is in question due to specific implementation details in the most common Java Virtual Machines (JVMs), as it creates a lot of small objects in certain circumstances. The reasons behind this are considered in more depth in Chapters 5 and 6.

4.4.2 CPU Profiling

This section considers the CPU efficiency of the Jena Memory Model, with particular reference to the process of answering queries rather than data assertion. The profiling tools use sampling to attribute time spent to methods: figures may thus not be perfectly accurate. To enable correct attribution of profiling data to methods, method inlining was turned off using the -XX:-Inline JVM option.

4.4.2.1 Indexes and Cache Efficiency

Jena uses a hash-based index scheme that does not support composite indexing. As a result, it spent a large proportion of its time iterating over indexes: iteration operations (next() and hasNext()) consumed 26% of CPU runtime. Jena's indexes offer excellent performance when matching against a single node (subject, predicate, or object): the underlying hash maps offer O(1) lookup, allowing the Bunch associated with a node to be retrieved quickly. However, if one wishes to restrict by a second node (for example, a predicate as well as a subject), it is necessary to iterate over all the elements in the retrieved Bunch to find the matches. This scales poorly as the Bunch expands.

Jena dedicated six times more storage to Hash Bunches than Array Bunches, with the Hash Bunches allocating space for an average of 1088 triples each, with some larger than 300,000. A consequence of the amount of data stored in large Hash Bunches is that Jena spends a lot of time iterating over the arrays backing those structures to retrieve matches. One result of this is that Jena's data cache efficiency is quite high: its access patterns are very predictable. Profiling indicated that over the course of a BSBM query session the proportion of data cache accesses that missed both L1 and L2 was just 0.45%, much lower than typical DBMSs (Ailamaki et al., 1999). It is likely that this percentage would increase with a better, more selective mode of access.

4.4.2.2 Node Comparisons

In any query, particularly those that deal with large working sets of data, a lot of node comparisons are necessary: joins must be performed, and triples must be matched within Bunches. Jena's design, however, does not preclude multiple node objects being created to represent the same actual node: it simply uses a node cache to try to reduce the number of duplicates that get created. This approach means that Jena does not have to maintain a separate explicit index to find nodes, but has a variety of disadvantages.

Since there may be more than one instance of logically equivalent nodes, it is not possible to use a simple referential comparison to determine node equality, and a String equality test is required. This is not a large performance hit when comparing strings of different lengths, since the inequality can be trivially discovered, but requires a computationally expensive character-by-character comparison in the case of equal length Strings.

This issue is exposed by the BSBM dataset: BSBM's automatically generated URIs have relatively little variation in length, and as a result Jena spent as much as 13.5% of its time performing String comparisons.

4.4.2.3 Garbage Collection

Tests indicated that Jena spent an insignificant amount of time (less than 0.05%) in garbage collection. This is due to the fact that the dataset is, in this case, static. The only garbage generated is short-lived objects related to handling queries, which are removed efficiently during collections of the young generation. In this case, generational garbage collection is an ideal match. It should be noted, however, that manually triggering a garbage collection caused a 15 second pause, indicating that in the event of a collection, the overhead is substantial.

4.4.3 Memory Profiling

The Jena Memory Model (JMM) was profiled after assertion of the dataset, with a full garbage collection triggered to remove any unused objects. The JMM used a total of 639.27MB after assertion, spread across 6,253,509 objects. This is an average of 466.4 bytes per triple. At nearly twice the size of the original data file, despite the normalising of repeated nodes, this footprint is clearly undesirable. It should be noted that this footprint is also dependent upon the dataset being in sorted order on the subject field, so that node cache utilisation can be maximised: using the same dataset in shuffled order increased storage requirements to 1.18GB. The space used by the sorted dataset broke down as follows: 377.71MB of the space was dedicated to node storage: nodes, String objects, their underlying character arrays, and so on. 257.86MB was dedicated to indexes: Triple objects, Bunches, and their underlying arrays. The remainder was used by instances of rarely used classes in the system.

There are several culprits for wasted space in the memory model. The most interesting is overhead associated with small objects: assuming even the bare minimum per-object overhead of 16 bytes, this amounts to 95.4MB. It can amount to substantially more when considering extra space required for alignment.

Secondly, the fact that nodes are not guaranteed to be normalised, combined with a relatively small, fixed node cache size, means that a lot of space is used storing duplicate nodes. Figure 4.4 indicates the number of nodes generated compared to those that exist in the dataset. The amount of memory used for storing nodes, even in this cached environment, validates the fully-normalised model.

Finally, there is the issue of empty space in the arrays that back both Hash and Array Bunches. Clearly, the fixed-size arrays of 4 or 9 elements will regularly be only partially full, and the Hash Bunches use at most 50% of their underlying array's capacity. This overhead exists for a reason: over-filling hash maps or resizing arrays each time they are added to is costly.

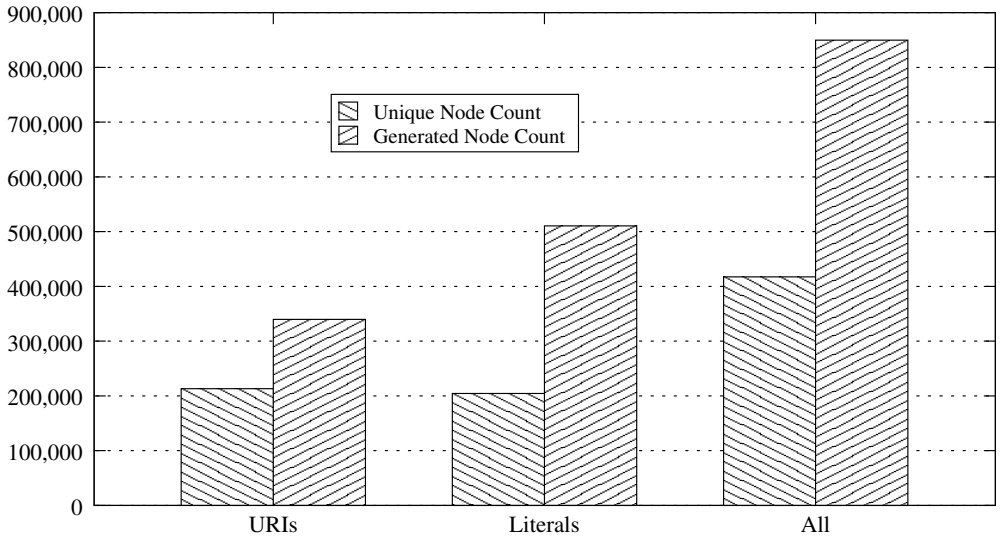


FIGURE 4.4: Nodes generated in the Memory Model versus unique nodes that exist in the dataset

4.5 Summary

This chapter has showed that modern JVMs are entirely suitable hosts for high-performance DBMSs:they offer excellent performance with respect to CPU time, and have the potential to be compact. Within this statement, however, there are caveats: programs that are likely to generate large quantities of small objects, in particular, are poor candidates for JVMs. This is due to the cost in terms of memory space, and the strain placed on garbage collection, as shown in Section 4.4.

If attention is paid to the weaknesses described in this chapter, however, there are few barriers to the creation of a performant Java-based RDF store. This analysis informs the use of Java as the implementation language for the prototypes described in Chapter 6.

Chapter 5

Examination of RDF Datasets

As previously emphasised in this document, the structure (or lack thereof) of RDF data remains a particular problem for efficient storage and retrieval. The commonalities that can be found in existing RDF datasets are not well understood, and it follows that understanding them in more depth would provide a substantial benefit to the development of high quality RDF storage systems. In order to inform such development, it was decided to create a tool to produce statistics on RDF documents.

This chapter describes the design and development of ExamineRDF, a tool created to produce detailed statistics over arbitrarily-sized RDF files. It is designed to require relatively little memory, scales linearly with the amount of data being processed, performs fast, append-only writes to disk, and reads from disk in large, contiguous chunks. Its only requirement is sufficient disk space to store its results during processing.

The chapter goes on to provide an explanation of the output of the ExamineRDF tool, and the use to which this output can be put. Finally, new statistics on a variety of popular RDF datasets are presented and analysed. This information offers insights into the compressibility of both triple data and the string sets found in RDF datasets, and provides much of the basis for the development of the new RDF data structure described in the following chapters.

5.1 ExamineRDF Design

ExamineRDF was created out of a desire to analyse popular RDF datasets such as DBpedia (Auer et al., 2007) and UniProt (Apweiler et al., 2004). DBpedia amounts to over 200 million triples, while UniProt is over three billion. Simply loading these datasets into an RDF store and extracting statistics using SPARQL queries would be impractical: it was found that just loading a 200 million triple set would take several hours on modern stores, and analytics would take much longer. Scaling this to UniProt, or even larger

datasets, would not be practical. This is the approach taken by RDFStats (Langegger and Wöß, 2009), the only alternative RDF statistics generation system that the authors are aware of. While it produces very detailed information, RDFStats does not effectively scale to very large datasets, and does not have support for human visualisation of results. As a result, the decision was made to build a custom system, the design of which is related in this section.

5.1.1 Parsing and Loading

The free Redland Raptor library (Beckett, 2002) offers a reliable, fast mechanism to parse RDF files of all common formats. As a result, the decision was made to use this library to provide parsing for ExamineRDF. Only one non-standard parse option was used: the feature RAPTOR_FEATURE_CHECK_RDF_ID was turned off. If this feature is left on, Raptor performs checks to eliminate duplicate triples. Tracking of duplicates requires a significant amount of memory that grows with the size of the dataset, and is thus impractical for a system designed to scale to arbitrary datasets.

After parsing an RDF triple, Raptor calls back to the main ExamineRDF program. The subject, predicate, and object are then hashed (using MurmurHash[1]), and the string data thrown away. The hashes are then used to uniquely identify the node. For each unique node, data is stored about its length (in terms of bytes, excluding any data type information), the amount it is reused, and the number of times it appears as the subject, predicate, or object of a triple. Further information is stored about how many times SP, PO, and OS pairings appear.

Storage of all of this information is handled by two large, constant size hash tables: one to hold the per-node information, and one to hold the pairing data. Since large RDF files would quickly exhaust all available memory, the program regularly dumps the hash tables to temporary files on the disk, and clears the hash tables. Once all the data is processed, the files can then be joined together.

5.1.2 Joining and Statistics Generation

Under normal circumstances, joining a large number of files that do not fit together in memory would be a painful process, involving a great deal of random I/O. ExamineRDF avoids this issue by virtue of its use of fixed size hash tables. The fact that the size of the tables never changes means that a node will always hash to the same slot: the program knows that for a given node or pairing, any data that requires joining will always be contained in the same slot, which occupies a limited, and generally very small, amount of memory. Since slots are always laid down on disk in ascending order, locating a slot

[1] http://sites.google.com/site/murmurhash

is trivial. All that is required, then, is to read sequential chunks from each temporary file in turn. Buffers default to 1MB per file. The process of joining slots from a buffer is illustrated in Figure 5.1.

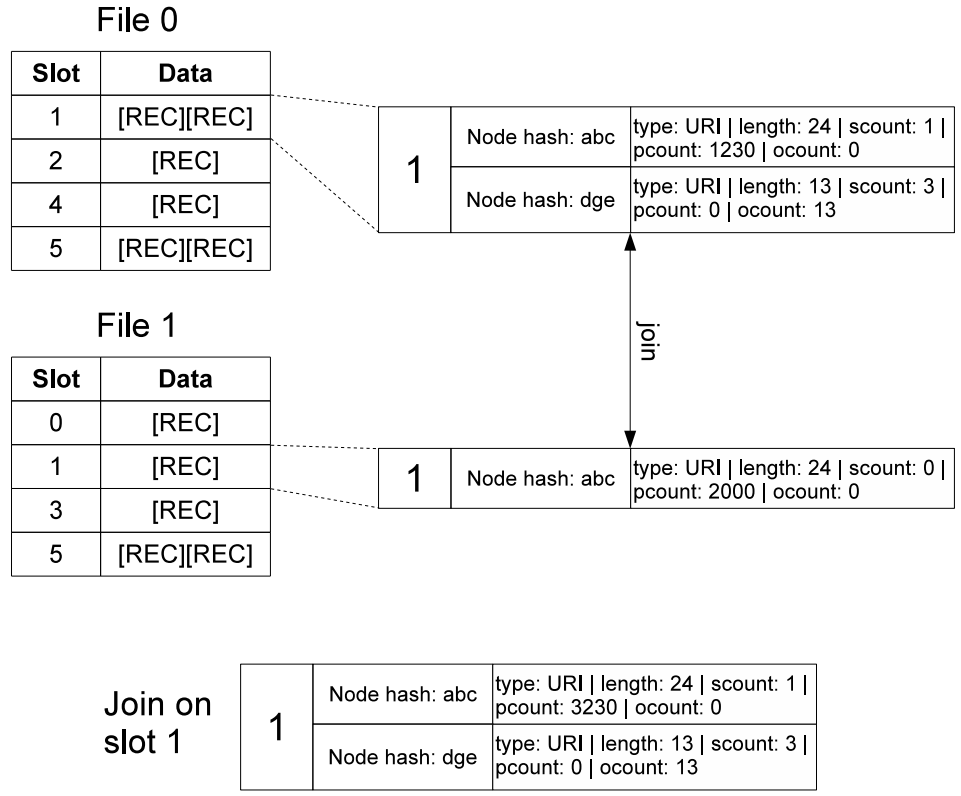


FIGURE 5.1: Joining file buffers in ExamineRDF

ExamineRDF's memory use can be configured: larger hash tables will result in lower disk space usage (as more repeating nodes are caught) and faster statistics generation (as fewer files need to be joined), but greater memory use.

ExamineRDF produces aggregate statistics on the fly during the join process, throwing away the joined data once it has been processed. This approach keeps the requirement for memory during the join phase low.

5.1.3 Design Discussion

ExamineRDF's core design plays to the strengths of modern disks: fast sequential reads and writes. As a result, it is able to fully process the DBpedia dataset in around 20 minutes on the machine described in Appendix B. UniProt takes significantly longer, at 10 hours. This is due mostly to the increased size of the dataset, but also to the fact that UniProt is encoded in XML/RDF rather than NTriples, and parsing XML is substantially slower. Figure 5.2 shows the time ExamineRDF required to process subsets of the DBpedia dataset, demonstrating linear scalability as the dataset grows.

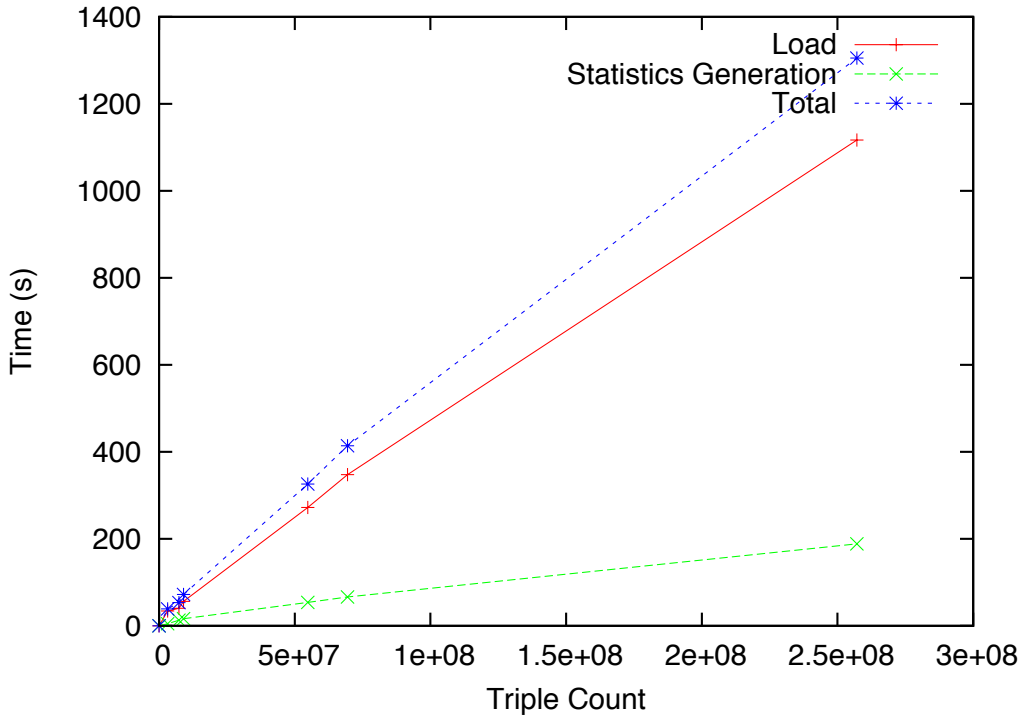


FIGURE 5.2: Time taken by ExamineRDF to process subsets of the DBpedia dataset

ExamineRDF does have one limitation upon scaling in its current implementation: it uses 64 bit hashes to 'uniquely' identify nodes. The program will not detect a hash collision, and so there is a risk that the statistics will be compromised. Assuming a hash function with perfect distribution, a dataset with 200 million nodes has a probability of experiencing a collision of around 0.1%, while a billion node dataset is nearly 3%. For perspective, UniProt has approximately 450 million nodes, while DBpedia has about 66 million. Generally speaking, this risk of collision is unimportant: to make a noticeable impact on the statistics, one heavily reused node would have to collide with another heavily reused node, and the likelihood of this is marginal. If ExamineRDF is used for datasets growing to hundreds of billions of triples or more, however, it may be worth moving to 128 bit hash values.

Initial tests showed that a more important skew on results were duplicate values: the statistics on the UniProt data, for example, suggested that a particular OS pairing was repeated over 200,000 times. This is a practical impossibility, since the UniProt dataset features only 127 distinct predicate values. As a result, duplicate detection was added to ExamineRDF.

Duplicate detection works in a similar manner to the rest of the program: each triple is hashed, and stored with a count in a fixed size hash table that dumps to a file when it is full. Prior to the main join process taking place, this data is read back in and joined. Triples with a count higher than 1 are duplicate values. As a result of the duplicate join process, dump files are created that contain negative counts for the relevant node and

pairing data. The dump files are then read in as normal during the main join process, and correct the results.

5.2 Output

ExamineRDF outputs basic human-readable text output, as well as detailed files for machine processing. It provides a top-level summary with the following information:

- Triple count.
- Unique URI and literal counts.
- Average URI and literal lengths (in bytes), including standard deviation.
- Average times each URI and literal are reused.
- Which URIs have appeared in which position in triples: how many have only appeared as S, or as S and P, and so on.

```
Triple Count: 231661194
URI Count: 30218224
Average URI length: 52.93, Standard Deviation: 20.45
Average URI reuse: 20.97
Appeared as (ignoring literals):
  S only: 1735317
  P only: 1101
  O only: 11794631
  S and P: 38559
  O and S: 16648616
  P and O: 0
  S, P and O: 0
  O including literals: 48039661
Literal Count: 36245030
Average literal length: 76.72, Standard Deviation: 282.03
Average literal reuse: 1.69
Blank Node Count: 0
Average Blank Node reuse: 0.00
```

FIGURE 5.3: Summary data output by ExamineRDF for DBpedia

For example, the DBpedia dataset's summary information appears as shown in Figure 5.3. In addition, the human-readable output contains tables with detailed data on the following information:

- The number of times nodes appear as S, P, or O: for example, UniProt has 39,749,143 subjects that appear only once.

- The number of times SP, PO, or OS pairings occur.
- URI and literal rate of reuse.
- URI and literal lengths.

These statistics give a wealth of information to work with, allowing the discovery of useful commonalities between RDF datasets. The basic summary data alone allows the determination of the ratio of unique nodes to triples, rates of reuse, the amount of unique string data it is necessary to store, and whether that is due to URIs or literals.

The detailed information gives even more useful feedback. Using the cardinalities of nodes and pairings, it is possible to garner detailed information on what proportion of a given RDF index would be comprised of repeating elements. This informs issues like the effectiveness of compression techniques, or whether a hash-based system using sub-indexes (as described in Section 3.3.5) would be practical. The lengths of URIs and literals informs issues like whether a few extremely large literals are a cause of the majority of string storage, whether URIs cause much of the storage overhead, and what kinds of compression might have a substantial impact on the size of a database.

Table 5.1 shows a subset of the node length information gathered over the UniProt dataset. The 'URI' and 'Literal' columns show the number of nodes with the size specified in the 'Node Length' column. For the sake of readability, ExamineRDF groups the size ranges as they increase: lengths between 10 and 19 are grouped, as are those between 100 and 199, and so on.

5.2.1 Visualisation

While the information output by the ExamineRDF tool is useful, the tabulated summary data does not give a good visualisation of the results. As a result, Perl scripts were written to process the machine-readable data files, and turn them into human readable HTML or LaTeX files with graph visualisations.

Producing useful graph visualisations is a challenge in and of itself, due to the large quantity of data, and the fact that it is very 'spiky': intermittent 0-values tend to make many graph visualisations unreadable. This section considers various potential types of graph, using the DBpedia dataset as an example.

5.2.1.1 Node and Pairing Data

Figure 5.4 shows a graph of the cardinality of predicates in the DBpedia dataset, plotting predicate cardinality on the x-axis against the number of predicates with that cardinality.

Node Length (bytes)	URI	Literal
Total	391273031	54206101
1-1	0	64
2-2	0	2190
3-3	0	31265
4-4	0	191842
5-5	0	375910
6-6	0	596624
7-7	0	1031449
8-8	0	1193425
9-9	0	2101651
10-19	190439573	20780878
20-29	124	1672825
30-39	63918056	1560708
40-49	131036014	1207753
50-59	5695675	750303
60-69	161304	701346
70-79	7430	762338
80-89	8280	710313
90-99	2293	684261

TABLE 5.1: Subset of node length information from the UniProt dataset

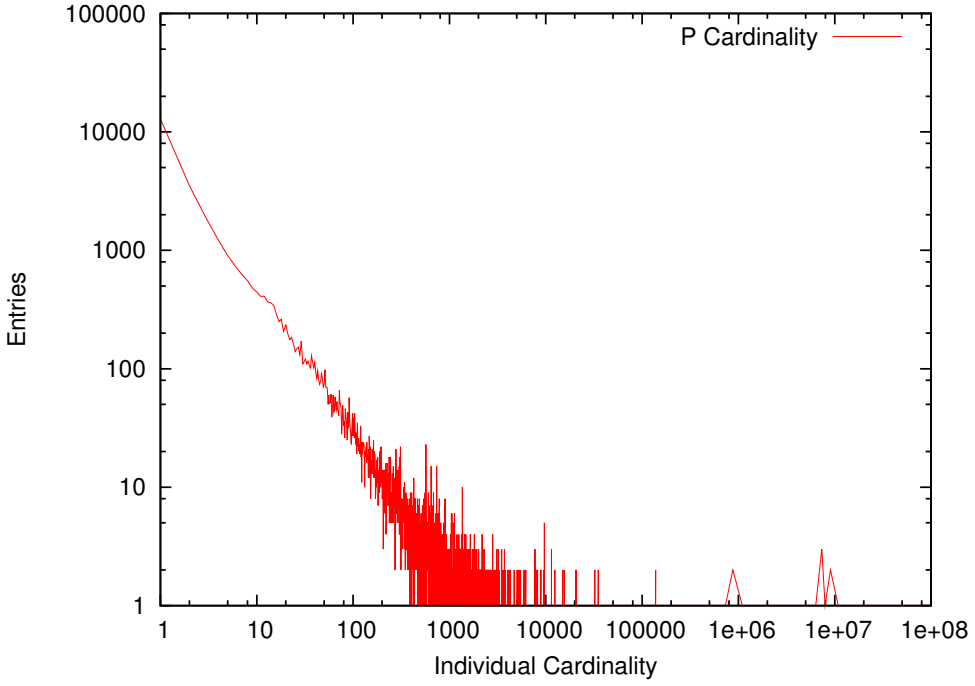


FIGURE 5.4: Cardinality of predicates in the DBpedia dataset

This figure provides some information: it is possible to tell that there are around 10,000 predicates that are used only once in the DBpedia dataset, and that there are just a few predicates that are reused a great many times. Unfortunately, however, the visualisation provides little additional information, due to the spiky nature of the data. It also

downplays the importance of the predicates that are repeated many times: while the majority of the dataset will be comprised of triples featuring one out of a small selection of predicates, this graph makes them appear almost insignificant. The problems of this form of visualisation are exacerbated when adding the data for S, O, and the SP, PO, and OS pairings, as shown in Figure 5.5.

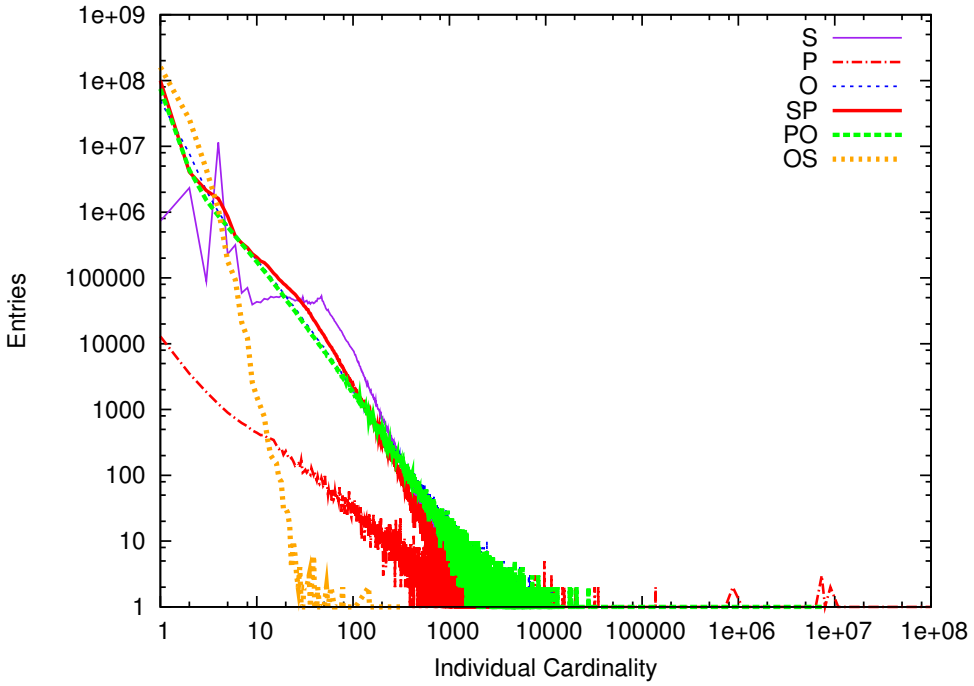


FIGURE 5.5: Naive (unclear) visualisation for node and pairing data in DBpedia

Clearly, Figure 5.5 provides virtually no useful information. An alternative visualisation is to base the y-axis upon how many triples the nodes or pairings have appeared in. This approach gives a much better idea of what proportion of the dataset is made up of triples containing subjects (for example) that repeat only once, or those that repeat many times. Figure 5.6 provides an example, again visualising predicates in the DBpedia dataset.

Figure 5.6 offers some improvement: It is clear from the graph that high-cardinality predicates are used in the bulk of triples in the DBpedia dataset. The issue of the data's spikiness is exacerbated, however. The problem is that as the graph goes to higher cardinalities, the likelihood of there being no predicates with that particular cardinality becomes higher. One might consider a scatter plot, but then the number of points, particularly near the beginning of the graph, makes the chart unreadable. A better solution is to make the graph cumulative, as shown in Figure 5.7.

Figure 5.7 gives a much more powerful, smoother visualisation. If one considers all predicates with a cardinality between m and n, the y-delta between those points describes the number of triples containing predicates with a cardinality between m and n. This is perhaps harder to initially comprehend than the previous, simpler graphs, but lends

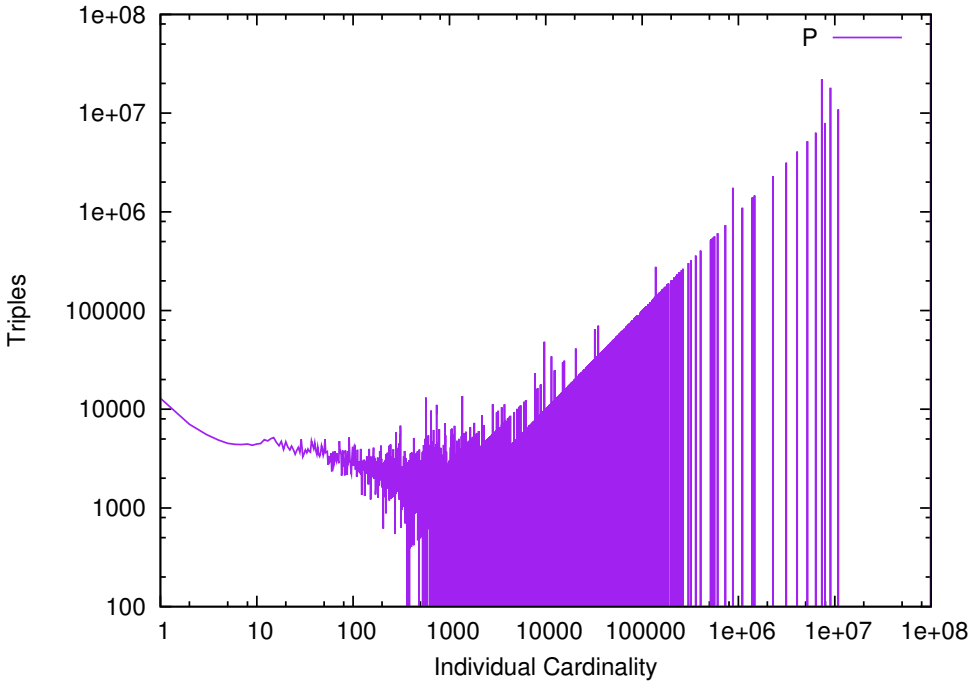


FIGURE 5.6: Improved visualisation for cardinality of predicates

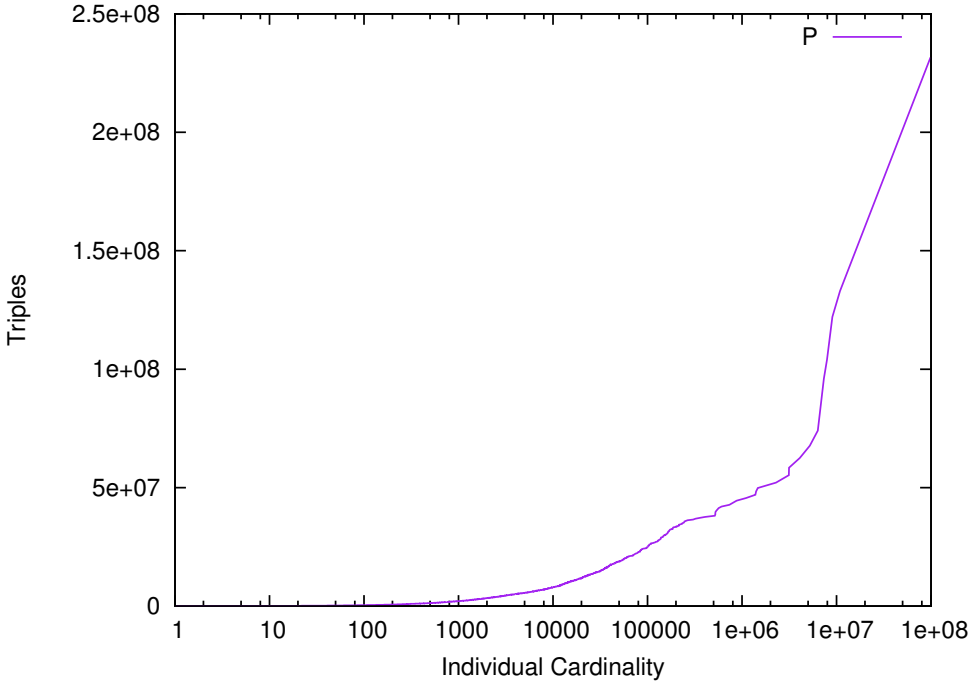


FIGURE 5.7: Cumulative visualisation for cardinality of predicates

itself well to determining what proportion of the dataset is made up of repeating or non-repeating values. This visualisation also lends itself well to including the data for S, O, and the SP, PO, and OS pairings all in one graph, as shown in Figure 5.8.

Using Figure 5.8, it is possible to discern a wide variety of information about the DBpedia

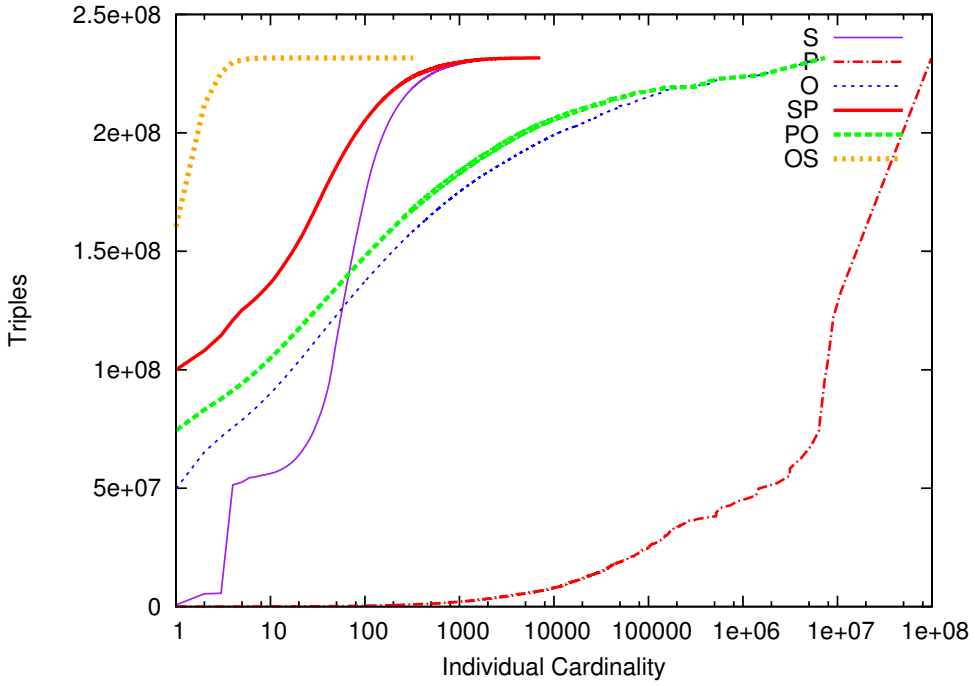


FIGURE 5.8: Cumulative visualisation of S, P, O, SP, PO, and OS cardinalities for the DBpedia dataset

dataset. Subjects are usually repeated between 1 and 200 times, with most of the dataset being comprised of subjects that repeat 50 times or more. Predicates take the opposite extreme: while there are a fair variety of different predicates, most of the dataset is comprised of triples featuring predicates that are reused tens of millions of times. Objects are something of a mixture: while most objects are repeated only a few times, there are enough objects of larger cardinality that triples featuring these objects make up a substantial proportion of the dataset.

The pairing data provides interesting information, too. Subject-Predicate pairings are of generally low cardinality: half the dataset is made up out of triples with pairings that never repeat. Predicate-Object pairings repeat at almost the same rate as Objects, suggesting that most objects are only ever referred to by one predicate. Finally, Object-Subject pairings are repeated extremely rarely. The impact of these insights is discussed further in 5.4.

5.2.1.2 Aggregate Node Reuse

ExamineRDF also produces detailed statistics over aggregate node reuse, covering the number of times nodes are reused in any position in a triple. These are provided in two forms, the first of which, shown in Figure 5.9, simply plots the number of times a node is reused (its cardinality) on the x-axis, against the count of nodes that are reused that number of times. The second, shown in Figure 5.10, is another cumulative graph,

similar to that shown in Figure 5.8. This plots node cardinality on the x-axis against cumulative (count of nodes with that cardinality) * (node cardinality). This latter graph helps visualise whether the dataset is made up of regularly repeating URIs and literals or not.

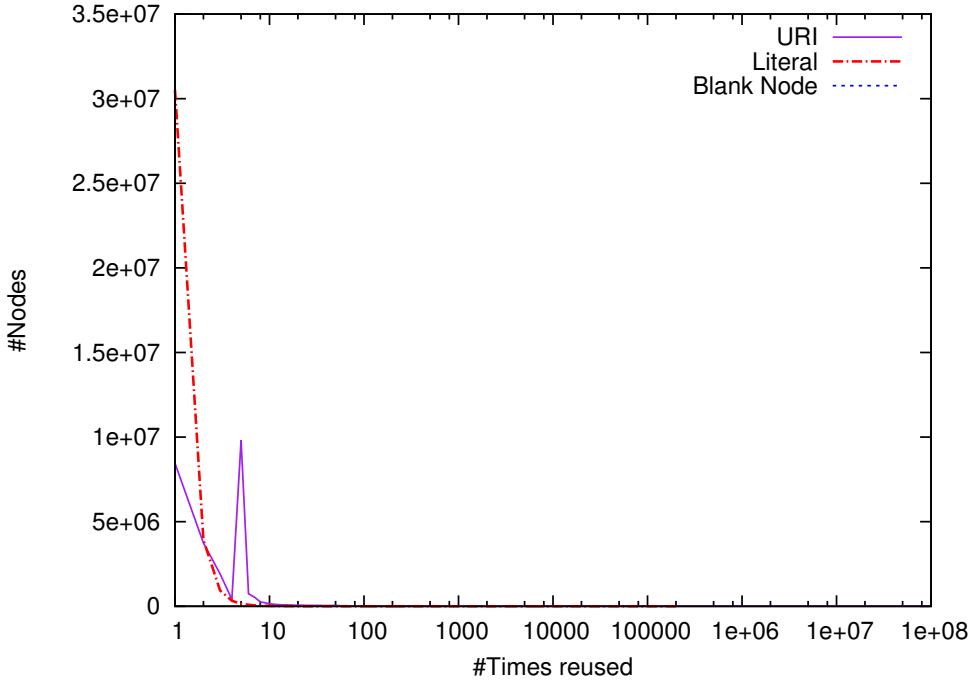


FIGURE 5.9: Node reuse data for DBpedia

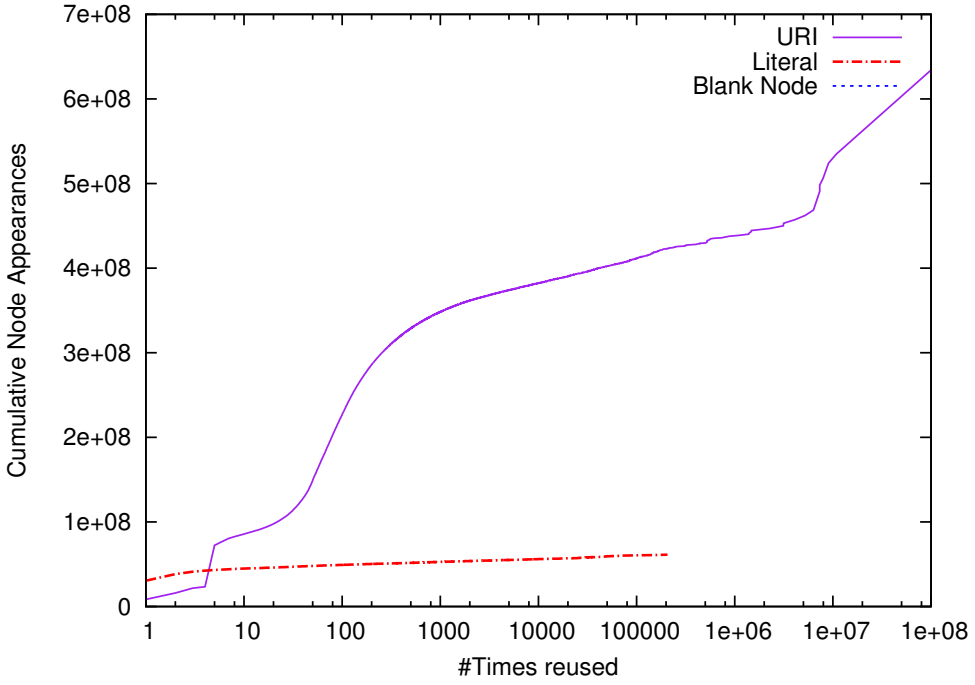


FIGURE 5.10: Cumulative node reuse data for DBpedia

Figure 5.10 provides more useful results, as Figure 5.9 hides information about the few nodes that repeat very regularly. Figure 5.9 might be improved by the addition of a log scale on the y-axis, but then the issue of the data's spikiness as it moves further along the x-axis becomes substantially more apparent. Several inferences about the DBpedia dataset can be drawn from these visualisations:

- URIs appear substantially more often than literals, much more so than is accounted for the fact that literals cannot appear as S or P.
- The DBpedia dataset contains no blank nodes.
- URIs experience a great deal of reuse, and the dataset is largely comprised of URIs that repeat more than 20 times.
- Literals generally repeat very little, although there are a few literals that repeat quite regularly. These are likely to be small integer values.

In fact, there are more unique literals in the DBpedia dataset than there are URIs. The fact that they're repeated so rarely means that URIs are seen much more often, however. This distinction can be made with the aid of the summary data, and the tabulated detailed information. Full tabulated data for all the datasets in this chapter can be found in Appendix C.

5.2.1.3 String Lengths

Finally, ExamineRDF also produces detailed statistics over string lengths. This data fits reasonably well into two different kinds of visualisation. The first, shown in Figure 5.11, simply plots node length on the x-axis against the number of nodes with that length on the y-axis. The second, shown in Figure 5.12, plots node length against cumulative bytes consumed.

Again, the non-cumulative form hides a lot of information. This can be mitigated by the addition of a log scale, but then the issue of data spikiness returns. There are several interesting facts about the DBpedia dataset exposed by these graphs:

- Literals occupy almost twice as much space as URIs.
- Most literals are quite small: around 10 characters long.
- The bulk of the space required by literal data is used in nodes larger than 100 bytes.
- Virtually all URIs are in a small size range, between 30 and 90 bytes. This is expected: much of the length of most URIs is a repeating prefix.

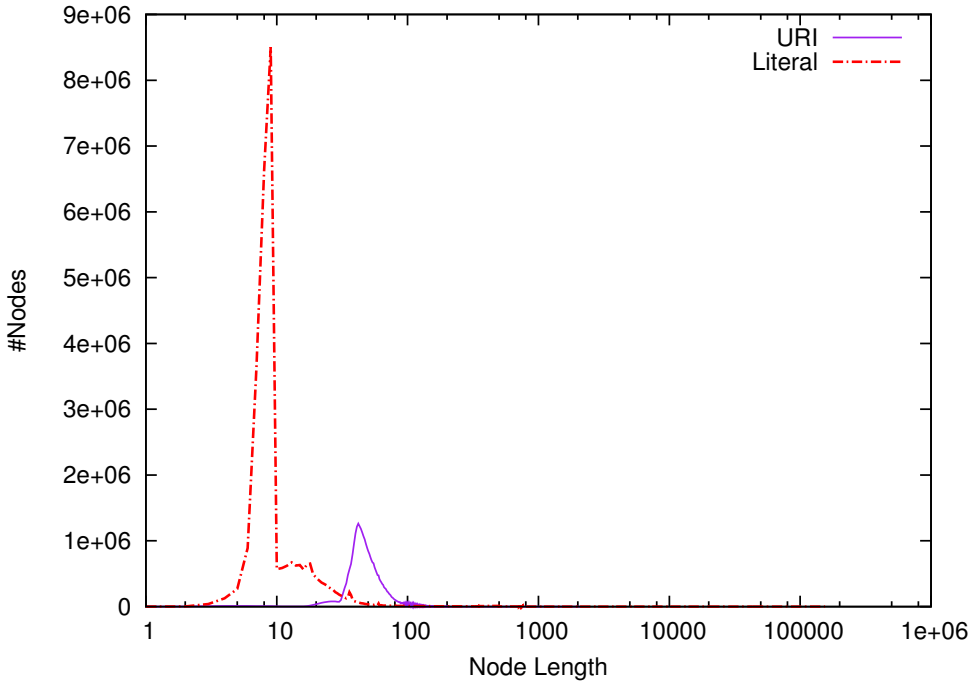


FIGURE 5.11: Node length data for DBpedia

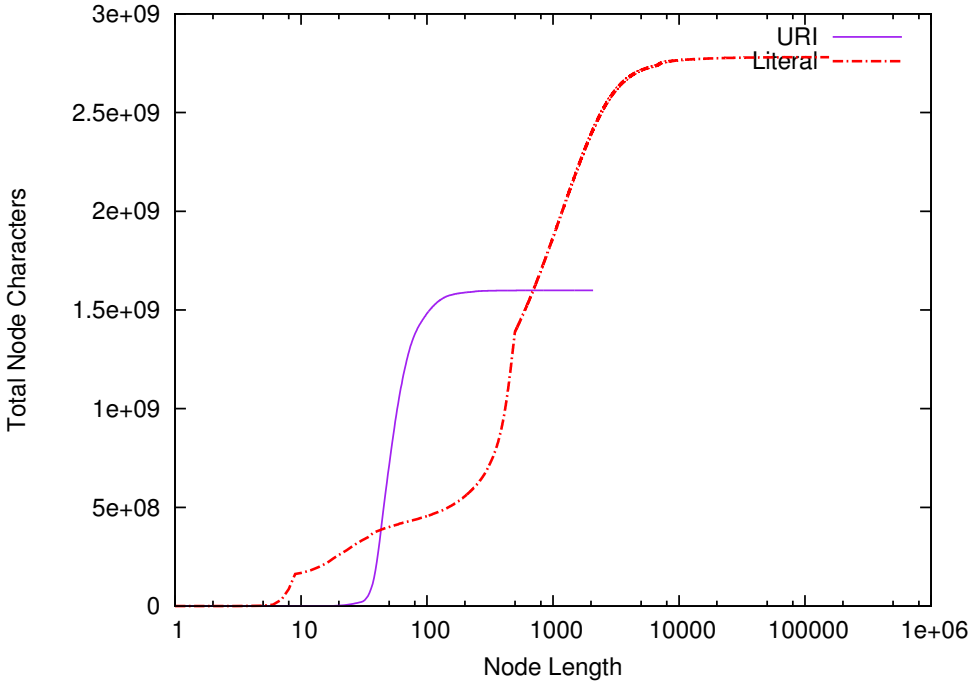


FIGURE 5.12: Cumulative node length data for DBpedia

5.3 Other Datasets

This section discusses two other datasets: UniProt, and a synthetically generated file from the Berlin SPARQL Benchmark (Bizer and Schultz, 2009), built using the standard

BSBM data generator, and options '-fc -pc 284826'. These datasets are used to demonstrate common trends in RDF storage, as well as provide a brief examination of BSBM's similarity to real-world RDF data. Appendix C contains the statistics for several more datasets that were used to inform the conclusions derived in this chapter.

5.3.1 Summary Information

```
Triple Count: 100000112
URI Count: 14739372
Average URI length: 88.57
  Standard Deviation: 2.40
Average URI reuse: 16.84
Appeared as (ignoring literals):
  S only: 8544915
  P only: 40
  S and P: 0
  O only: 5705305
  O and S: 489112
  P and O: 0
  S, P and O: 0
  O including literals: 14467148
Literal Count: 8761843
Average literal length: 477.77
  Standard Deviation: 601.05
Average literal reuse: 5.90
Blank Node Count: 0
Average Blank Node reuse: 0.00
```

(a) BSBM

```
Triple Count: 2809173894
URI Count: 391273031
Average URI length: 29.08
  Standard Deviation: 13.19
Average URI reuse: 18.98
Appeared as (ignoring literals):
  S only: 101791597
  P only: 104
  S and P: 0
  O only: 39637426
  O and S: 249843881
  P and O: 23
  S, P and O: 0
  O including literals: 93843528
Literal Count: 54206102
Average literal length: 158.32
  Standard Deviation: 301.96
Average literal reuse: 18.45
Blank Node Count: 0
Average Blank Node reuse: 0.00
```

(b) UniProt

FIGURE 5.13: Summary data

The summary data for UniProt and BSBM is shown in Figure 5.13. It's immediately clear that both UniProt and BSBM operate over much more tightly specified domains than DBpedia (shown in Figure 5.3): each have relatively few unique predicates, at 127 for UniProt and only 40 for BSBM, compared to nearly 40,000 for DBpedia. It might be argued, then, that DBpedia represents a more compelling use case for RDF: UniProt and BSBM might be represented using a relational schema, while it is not clear that the same is true for the DBpedia dataset.

Another notable issue is that BSBM has, on average, much larger strings than either UniProt or DBpedia: literals are three times longer than UniProt, and 6 times longer than DBpedia. Further, URIs are substantially longer in BSBM. Table 5.2 shows the space required per triple for string data for each of these three datasets, assuming the string data is normalised such that it does not have to be repeated. It's clear from these statistics that the synthetic BSBM dataset has atypically large storage requirements for string data.

Dataset	Literal	URI	Total
BSBM	41.86	13.05	54.91
DBpedia	12.0	6.90	18.90
UniProt	3.05	4.05	7.1

TABLE 5.2: Space required for string data for BSBM, DBpedia, and UniProt in terms of bytes per triple

5.3.2 Node and Pairing Data

Node and pairing cardinalities for BSBM and UniProt are shown in Figure 5.14 and Figure 5.15 respectively. These bear marked similarities to the DBpedia data shown in Figure 5.8, but do exhibit certain differences:

- Subjects have an even lower cardinality.
- SP pairings almost always have a cardinality of 1.
- OS pairings almost always have a cardinality of 1.
- Predicates are even higher cardinality: low cardinality predicates make no real impact on the dataset.

5.3.3 Aggregate Node Reuse

Aggregate node reuse data for BSBM and UniProt is shown in Figure 5.16 and Figure 5.17 respectively. In this case, UniProt shares some similarities with the DBpedia dataset, shown in Figure 5.10: Both feature dramatically greater overall instances of URIs than literals. BSBM, on the other hand, features a great many more literals.

In contrast to DBpedia, UniProt and BSBM feature many more literals that are repeated regularly. This is likely to be a result of integer values: the literal '0', for example, will be repeated very often.

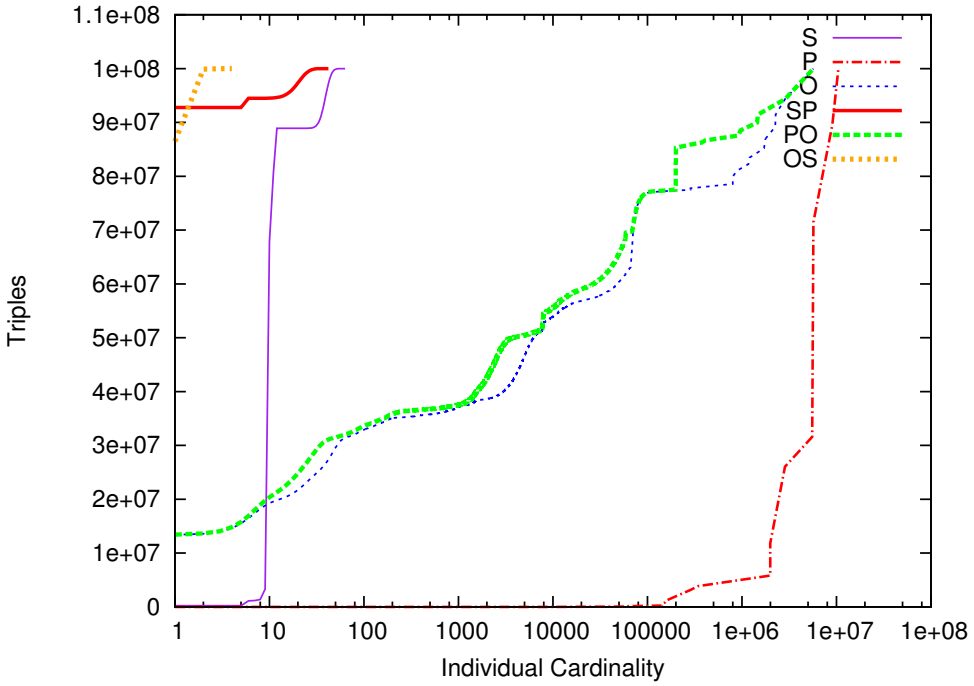


FIGURE 5.14: Cumulative node and pairing data for BSBM-100m

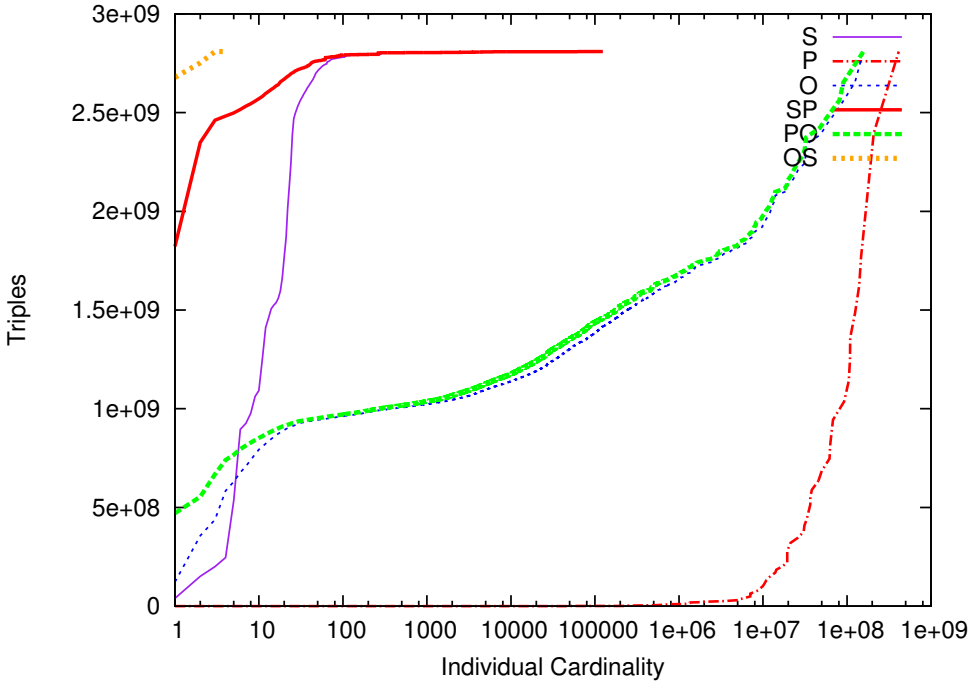


FIGURE 5.15: Cumulative node and pairing data for UniProt

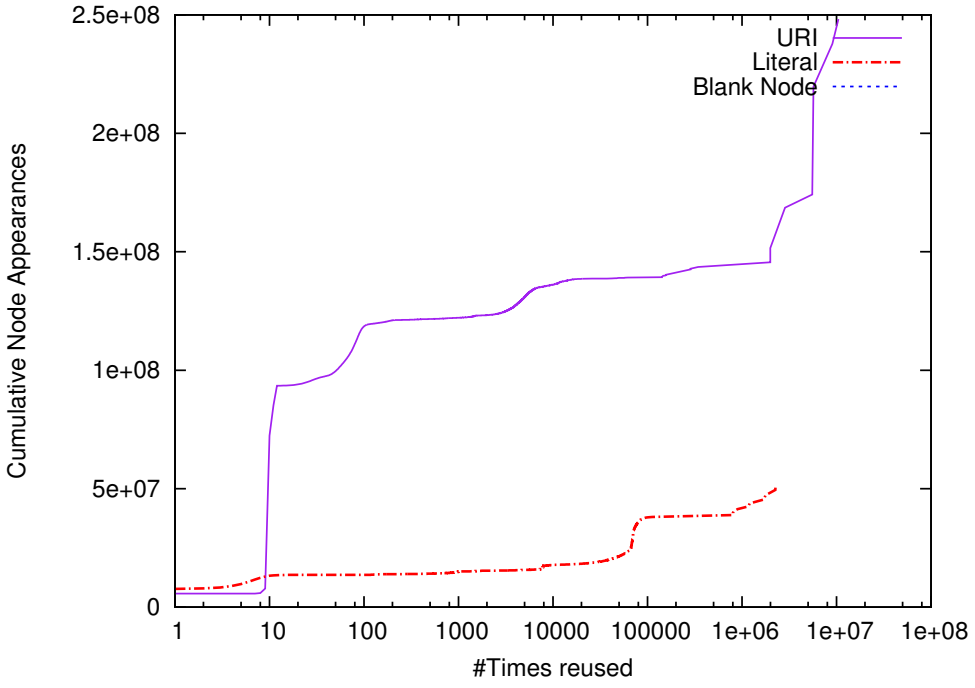


FIGURE 5.16: Cumulative node reuse data for BSBM-100m

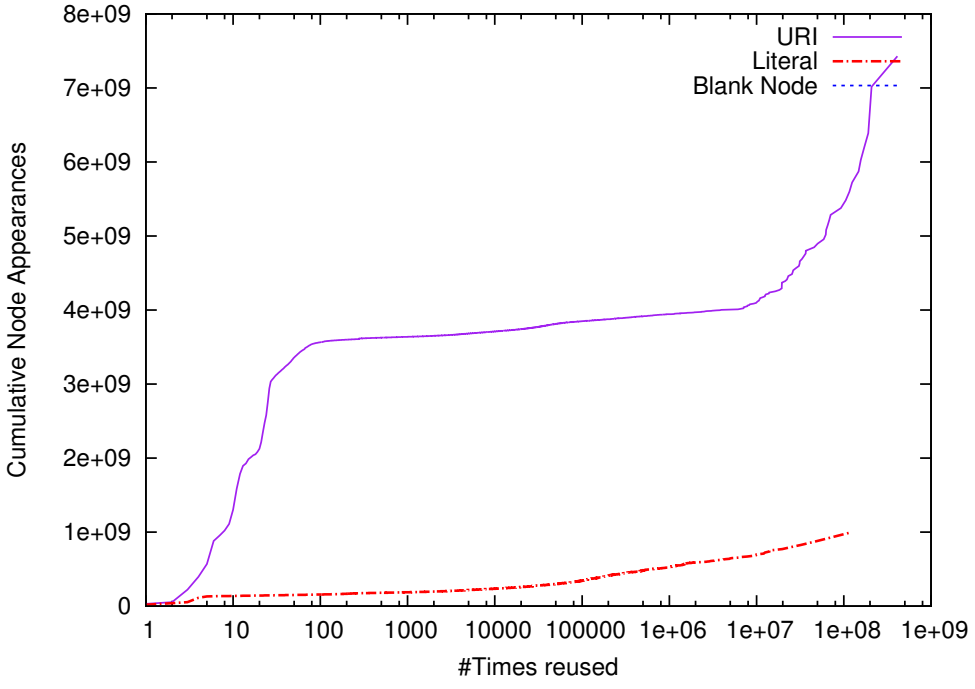


FIGURE 5.17: Cumulative node reuse data for UniProt

5.3.4 String Lengths

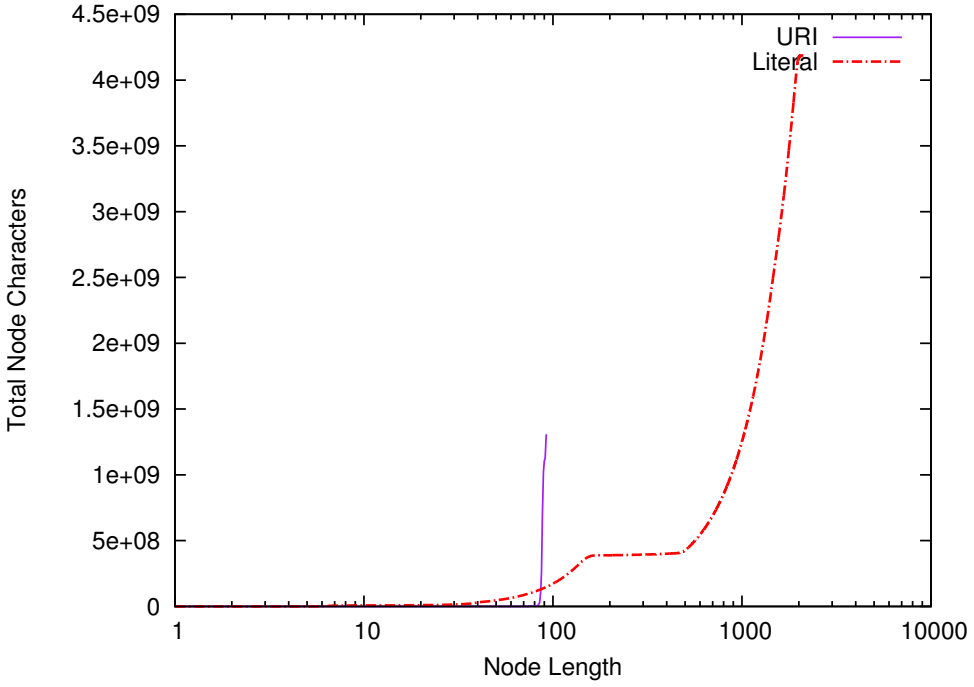


FIGURE 5.18: Cumulative node length data for BSBM-100m

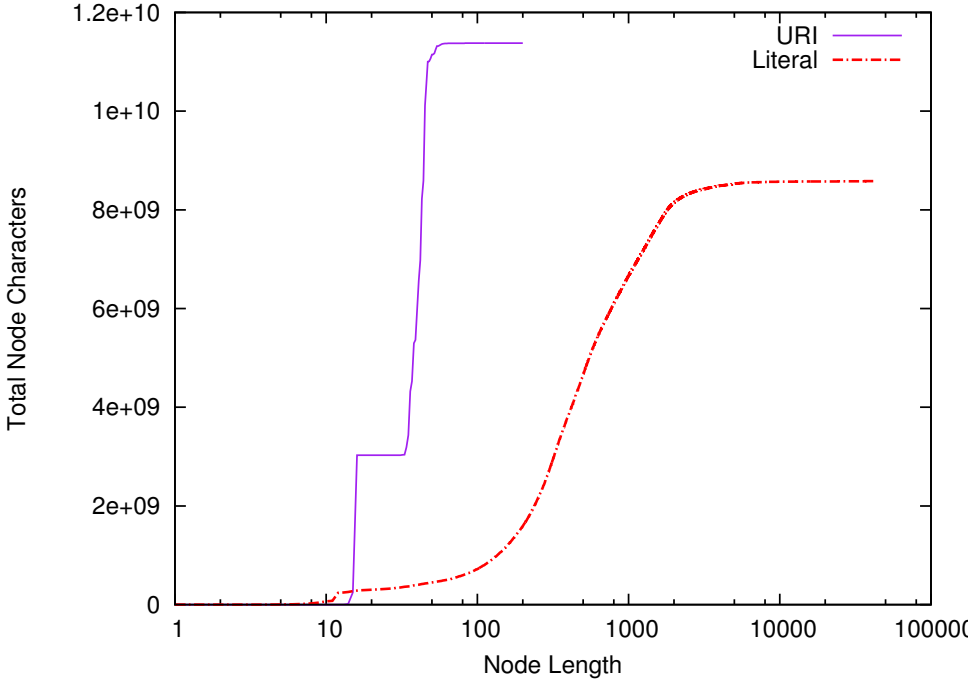


FIGURE 5.19: Cumulative node length data for UniProt

Node length data for BSBM and UniProt is shown in Figure 5.18 and Figure 5.19 respectively. BSBM shows a striking difference to UniProt and DBpedia (shown in Figure 5.12) here: along with string data requiring substantially more space per triple

than in the other datasets, BSBM has a much larger proportion of its string data in literals, particularly very large ones of over 500 bytes.

5.4 Discussion

The statistics produced by ExamineRDF provide a great deal of information, making it easier to design RDF storage systems in an evidence-based manner, and informing future research in the area. This section discusses the most important insights gained by the examination of the datasets examined in this section, as well as those covered by Appendix C.

5.4.1 RDF Index Design

The most obvious outcome of ExamineRDF's output is that datasets are largely comprised out of nodes that repeat several times, and encoding the node once for every triple it features in is a mistake. A clear case is thus made for the normalised model of triple storage, where nodes are uniquely identified by integer IDs or references.

Examination of the results in this chapter reveals that the different orderings of RDF data require very different behaviour from their data structures to extract high performance:

- The low cardinality of Subjects and SO/SP pairings means that lookup time is always dominant for retrievals from SPO or SOP-ordered indexes.
- When restricting by Predicate alone in a PSO or POS-ordered index, speed of iteration is paramount due to the very high cardinality of most Predicates.
- Since PS pairings are always low cardinality, restricting by these two attributes on PSO indexes is dominated by lookup time. By contrast, PO pairings are a mixture of high and low cardinalities. Depending on the use case, lookup time or iteration performance may be more important when restricting by two attributes over POS indexes.
- Objects also exhibit a mixture of cardinalities, so restrictions by one attribute in an OSP or OPS-ordered index depend on the use case.
- OS pairings are always low cardinality, and so restrictions by two attributes on an OSP-ordered index are always dominated by lookup time. By contrast, OP pairings are a mixture of cardinalities, so restrictions by two attributes on OPS-ordered data depend on the use case.

In order to complete this information, it will be necessary to perform studies of the kinds of RDF queries that are performed in the wild, to determine which of these factors are most important.

The qualities of POS-ordered data give an indication of the difficulties involved in creating scalable RDF stores. Any query that requires iterating over all the data associated with a high cardinality P or PO pairing will inevitably have to work with a huge amount of data. This issue is compounded by the fact that high cardinality predicates such as <rdf:type> or <rdfs:label> tend to scale up in linear fashion with the size of the dataset.

In practice, most triple patterns in a query include a fixed predicate (Seaborne, 2008). The result of this is that for OSP and SOP indexes, limited size is a greater priority than high performance.

In addition to retrieval performance, the size of RDF data structures is very important, particularly in contexts where a substantial amount of information is cached in memory. One approach to reducing space requirements is to eliminate repetition of IDs, as seen in systems like Hexastore (Weiss et al., 2008) and multi-level hash indexes. Many index types require that each triple be stored in its entirety: for example, a B-tree might store four triples in a leaf in the form SPO-SPO-SPO-SPO. If S and P are the same for each triple, it is wasteful to repeatedly encode them. Ideally, one would wish to store it in the form SPO-O-O-O instead. This can be accomplished in B-Trees with prefix compression schemes, while systems like Hexastore, or multi-level hash indexes, work by essentially creating sub-structures for each node and node pairing. For the purposes of this discussion, the Hexastore-style approach will be termed *prefix elimination*, as opposed to *prefix compression* for the techniques used on trees.

Unfortunately, the variance in the level of repetition between (and even within) different index orderings highlights the challenge in compressing triple indexes. The effectiveness of prefix elimination depends on regular ID repetition in order to make up for the overhead of storing pointers to the sub-structures. A similar issue exists for prefix-compressed tree indexes: the prefix must repeat often enough that the overhead of the compression scheme is eliminated.

In an environment where prefixes are reused very often, as in a POS-ordered index, the cost of the pointers in a prefix-eliminated index is virtually negligible. By contrast, in an SPO-ordered index, with a relatively low level of prefix reuse, the pointer cost is very significant. These observations are backed up by the calculations shown in Table 5.3. This table shows the size of a prefix-eliminated index like that used in Hexastore, normalised against the size of an equivalent uncompressed B+Tree, at varying levels of per-prefix overhead. Orderings are grouped by the two minimal sets of orderings that provide full index coverage, showing that there is no theoretical advantage to choosing either grouping. Note that this comparison ignores additional overheads like partial filling of data structures: it assumes that both structures experience 100% utilisation.

TABLE 5.3: Size of prefix-eliminated indexes over BSBM, DBpedia, and UniProt data, normalised against a B+Tree with 100 triple wide nodes. Both structures use 32-bit wide IDs.

Dataset	**Overhead (B)**	**SPO**	**POS**	**OSP**	**Avg**	**SOP**	**PSO**	**OPS**	**Avg**	**Overall Avg**
BSBM	8	1.28	0.46	1.33	1.02	1.28	1.19	0.60	1.02	1.02
	4	0.98	0.42	1.01	0.80	0.98	0.92	0.51	0.80	0.80
	0	0.66	0.38	0.68	0.57	0.66	0.63	0.43	0.57	0.57
DBpedia	8	0.85	0.66	1.36	0.96	1.17	0.77	0.92	0.96	0.96
	4	0.68	0.56	1.03	0.76	0.90	0.63	0.73	0.76	0.76
	0	0.51	0.45	0.69	0.55	0.62	0.49	0.54	0.55	0.55
UniProt	8	1.16	0.51	1.35	1.01	1.35	1.04	0.63	1.01	1.01
	4	0.89	0.46	1.02	0.79	1.02	0.81	0.53	0.79	0.79
	0	0.62	0.40	0.68	0.57	0.69	0.58	0.44	0.57	0.57

In the ideal case of no per-prefix overhead, prefix elimination saves a lot of space. In a more realistic situation of 4 or 8 bytes per-prefix overhead (typical pointer sizes), however, the situation is less clear cut. Additional per-prefix overheads are also likely to be incurred: for example, storing the count of values at each level of sub-structure.

The upshot of this is that for some orderings of the triple data, an indiscriminate prefix elimination scheme may get overwhelmed by overheads. In order to achieve very good compression ratios, they must be able to adaptively apply prefix elimination only where it is appropriate. For example, if prefix elimination were only applied over the 'S' portion of the SPO-ordered index on the BSBM dataset, the data structure would be only 70% of the size of an equivalent B+Tree, even with an 8 byte per-prefix overhead.

It is worth noting that Hexastore mitigates its overheads by sharing some information between different data orderings, resulting in a 20% overall saving of space when compared to the naive approach considered in this section. In general, however, this work indicates that prefix elimination schemes benefit from a more adaptive approach.

5.4.2 String Storage

The amount of space required for string storage in RDF datasets can vary substantially: UniProt requires only a third as much per triple as DBpedia, for example. Regardless, however, it is likely to occupy a substantial proportion of the space required by any dataset. This is an important issue for any store that operates in memory, or wishes to cache nodes effectively.

Reducing the space required for string storage must be split into the separate issues of URIs and literals, since these behave very differently. URIs are a relatively simple case for performant compression: they typically share long, regularly reused prefixes with a

short unique portion as a suffix. Prefix compression is a cheap, fast and effective way to compress data, and should allow much of the cost of URIs to be eliminated.

Literals are a more complex case, but generally require as much space as URIs, and are thus equally important. Long literals, which make up the majority of literal string data, are likely to be unpredictable text, for which simple methods like prefix compression are ineffective. General purpose compression algorithms such as Lempel-Ziv (Ziv and Lempel, 1978) are effective at compressing this sort of information, but the cost of decompression is substantially higher. For a high performance in-memory store, this cost may be unacceptable. An alternative approach, although unlikely to be faster, might be to offload large literals onto fast secondary storage such as an SSD.

The problem of large literals requiring time to access, whether it be by decompression or I/O latency, may not be as large as it initially appears. While the majority of space required for literals is for those that are large (over 100 characters), smaller literals make up the majority of the dataset: for example, 86.4% of literals in DBpedia are smaller than 50 characters. Assuming a linear access profile, and only large literals undergoing a compression or secondary storage strategy, most literal accesses would still cause no slowdown. It is likely that large literals will be reused less (since values like small integers are likely to be reused many times), meaning that even fewer accesses would cause a slowdown. Depending on the use case, these approaches may be a worthwhile compromise in order to save space.

Many types of small literal are amenable to compression, too. In an integer ID-based system, integers, and types that can be represented using integers such as dates, can be inlined directly into the ID, rather than being stored as separate node objects. Figure 5.20 shows an example of a 32-bit ID scheme that supports inlining. The first bit is used to determine whether the integer represents an ID or an inlined literal. If it is an ID, the remaining 31 bits are dedicated to the ID. If it is an inlined literal, the following 7 bits represent the type of the literal (integer, date, and so on), and the remaining bits are the actual literal data. If the literal is too wide to fit into that space, it will not be inlined.

Inlining offers a variety of benefits. Firstly, it saves space. The statistics show that the space required for small literals is almost insignificant, but these do not include overheads: space required to store the node in the node/ID map, and small object overheads being the most obvious examples. Secondly, inlining can improve performance. It does this by eliminating the cost (and likely cache misses) of performing the node/ID mapping. This is especially useful when performing filter operations, which often require a large quantity of ID to node conversions.

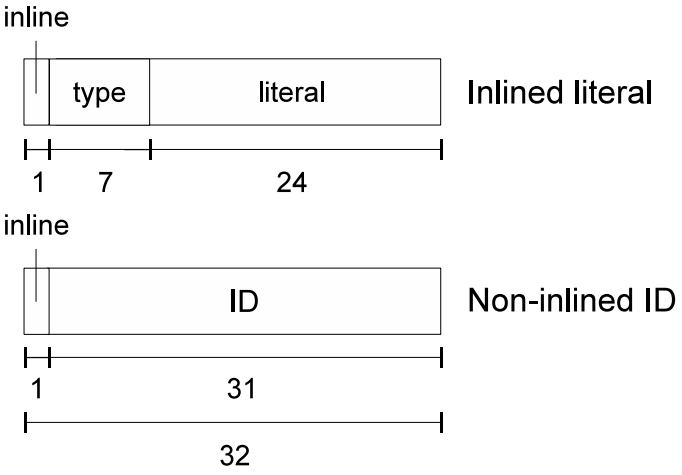


FIGURE 5.20: Inlining data into a 32-bit ID. All figures are bit-widths.

5.4.3 Synthetic Datasets

This section briefly considers the challenges facing synthetic data generation, particularly in an environment where queries must be automatically generated over the synthetic data.

The most obvious difference between the DBpedia and UniProt datasets is the number of distinct properties they contain. UniProt is a more obviously homogeneous, managed set of information, featuring just 127 distinct properties, as opposed to tens of thousands for DBpedia. BSBM attempts to mimic the characteristics of managed datasets like UniProt, generating triples in a repetitive pattern based on a constant scale factor.

Generating synthetic data via a repeating pattern is an understandable choice, as it is substantially easier to automatically generate queries over a predictable dataset. It is questionable, however, whether this approach represents the most compelling use case for RDF: that of a world where the data is unmanaged and unpredictable, where new properties may be added regularly and at will, and where multiple datasets are aggregated together. This environment is one in which the traditional relational model suffers for its expectation of predictable data, and RDF shines. By contrast, BSBM datasets translate well to a relational model, and the benchmark shows substantially better performance on such systems (Bizer and Schultz, 2009).

While BSBM effectively mimics the characteristics of managed datasets like UniProt, it does have some associated problems. One significant issue is the average string length. Both literals and URIs have a substantially larger average length than that of common real world RDF datasets. BSBM thus puts a greater emphasis on string compression than is strictly warranted. An additional issue is that BSBM's URIs display a very noticeable lack of variation in length when compared to real-world datasets.

To be effective tests of real world RDF store performance, benchmarks using synthetic datasets must carefully examine and simulate both real world datasets and the queries likely to be performed upon them. ExamineRDF can be used as a tool to improve the quality of synthetic datasets, a fundamental aid to research in the area: in the long run, the information provided by ExamineRDF should have the effect of improving the RDF stores that test against these synthetic testbeds.

5.4.4 The Future

ExamineRDF can already be used to inform the design of physical storage schemas and query optimisers for RDF data, but there is a variety of work that could be done to yield even more useful results. The data that ExamineRDF already collects could be used to produce even more statistics: for example, it would be interesting to know how the length of a node string affects its likelihood of reuse. It would also be useful to be able to set parameters such as a minimum overhead per node, in order to get a more complete understanding of the amount of space small nodes consume in a real-world implementation. Finally, there is certainly room for providing improved graph visualisations, and even making them interactive, to allow the user to experiment with what visualisation suits them best.

In terms of the core architecture, while ExamineRDF does scale linearly with the size of the dataset, it currently only uses a single core of a CPU. This lack of distribution could cause issues when scaling to datasets in the hundreds of billions of triples. Fortunately, it is exceptionally well suited to parallelisation. During the load phase, files could be split into portions, each of which is attended by one thread, writing to a separate file. During the join and statistics aggregation phase, the temporary files could also be chunked, thanks to the property that related data is always found within the same hash slot.

Overall, ExamineRDF provides a practical, scalable means to understand how the structure of RDF datasets affect how we need to store them. The information provided by this tool is an invaluable aid to the design of effective RDF storage structures, as will be described in Chapter 6.

Chapter 6

AHRI, a Highly Performant In-Memory RDF Index

In Stonebraker et al. (2007), Stonebraker, a lead creator of the Ingres (Stonebraker et al., 1976) and Postgres (Stonebraker and Kemnitz, 1991) DBMSs argues that for many datasets, main memory DBMSs represent a desirable alternative to disk backed systems. The amount of memory available in even mid-range server systems is in the order of tens of gigabytes, and there is no reason to believe that this amount will not continue to grow rapidly. Stonebraker suggests that datasets of less than 1 terabyte can be considered appropriate for main-memory systems in the near future.

The storage and query of RDF data generally fits this requirement: files of over a billion triples are considered truly large, and queries upon these datasets stress the capabilities of existing systems (Rohloff et al., 2007); yet a billion triples does not constitute a vast quantity of information. Indeed, datasets smaller than 100M triples (around 20GB of asserted data) can stretch the capabilities of current disk-backed systems, particularly in applications that require low-latency queries: for example, human-interactive applications, where users are unwilling to wait long periods of time between responses (Erling, 2009; schraefel et al., 2005).

Thanks to the consistently reducing cost of RAM, in-memory storage is becoming increasingly compelling for classes of data, like RDF, that require little space but a large amount of processing. A modern processor can perform tens of millions of operations in the time it takes to perform a single disk seek, and since such operations are common in RDF processing, this can add up to a vast amount of time lost.

It is clear that a direct in-memory solution has substantial performance benefits over a disk-backed system operating with enough RAM to cache all its data: disk-backed systems are constrained by the need to use data structures tuned for disk storage, and by the need to check that relevant data is stored in buffer pools. In-memory storage

does, however, bring its own set of challenges: Firstly, there is a much greater premium on space, rendering it impractical to store every possible sort order in a separate index, and secondly, latencies that are relatively well hidden by the overwhelming cost of disk access come to the fore.

Given the problems involved in creating high-performance RDF stores, moving to low latency in-memory storage is clearly an avenue worthy of future research. The focus chosen within this area was a data structure designed for indexing RDF data in-memory. Current RDF storage structures are flawed when used for in-memory storage: trees typically require a lot of space, and exhibit large find times, while hash-based structures either provide no support for composite indexing, or result in an excessive proliferation of indexing structures, and resultant space overheads. The primary contribution of this document is an investigation into a new structure for RDF data called the Adaptive Hierarchical RDF Index (AHRI).

6.1 Requirements

For the purposes of creating an in-memory index for RDF data, the following requirements were considered.

1. The index must support both fast find and fast iteration. Both of these features are necessary: SPO-ordered indexes, for example, generally have low cardinality contents and are thus find dominated, while POS-ordered indexes have high cardinality contents, and are thus dominated by the time it takes to iterate over their contents.
2. The index must support fast insertion and deletion. While many current uses of RDF are read-mostly (Weiss and Bernstein, 2009), this is partly a function of the fact that performant RDF stores usually have relatively poor update support. Much as has occurred on the Web, some workloads can be expected to become more write-heavy as RDF is exposed to a greater variety of applications.
3. The index must be compact: while RAM is constantly becoming cheaper, it is still a more expensive resource than disk, so it is important to use it efficiently.
4. The index must perform effectively when written using newer languages. Languages like Java or Python have substantial overheads for small objects, which must be considered in the design phase.

From these base requirements were derived several further issues of importance:

- Support for composite indexes is vital. As the indexes scale, the inability to filter by more than one attribute causes an overwhelming slowdown.

- Given the performance requirements, the index must be designed to miss cache infrequently, and cause very few branch mispredictions.
- The index must not create a large quantity of small objects. These will cause excessive space overhead, as discussed in Chapter 4.
- Any compression scheme used must exhibit exceptionally high performance, and not significantly impact the ability to add or remove triples from the store.

6.2 Design Decisions

Given the set of requirements described in Section 6.1, this section considers the broad architectural decisions that were made to support the creation of an in-memory index for RDF data.

6.2.1 Normalisation Strategy

Given the requirement for space efficiency, any in-memory index structure must feature a fully normalised strategy; that is, any given node must be represented in its full lexical form only once in the system. There are two common models for normalisation in RDF stores. The first (referred to as *Disk*) is seen most often in disk-backed stores. It is quite simple: one structure maintains a list of unique nodes, and provides mapping between nodes and an integer ID that is used to represent the node elsewhere. A second structure is a set of indexes, each of which represents the triples in the dataset in a given attribute order (SPO, POS, etc). These indexes encode node IDs inline to represent the subject, predicate, and object of each triple.

An alternative approach (referred to as *Mem*) is seen in in-memory stores: these typically have a model where each triple is represented by a triple object, each of which contains references to node objects. Indexes over this data then use references to point to related triple objects. This approach does not require a separate structure to convert between IDs and their equivalent nodes: the IDs in this system are references to triple and node objects, which implicitly encode where the nodes are located.

From the point of view of in-memory storage, *Mem* has a variety of advantages: space is saved by the lack of a node mapping structure, and by the fact that triples are normalised as well as nodes: *Disk* encodes the triple inline in each of its triple indexes, which means that $(Index\_Count * 3 * ID\_Size)$ bytes are used representing each triple. By contrast, *Mem* requires $((Index\_Count * Reference\_Size) + Triple\_Size)$ bytes per triple, where Triple_Size usually equates to $(3 * Reference\_Size)$. Further, retrieving the lexical text of a node is faster: one simply has to follow the reference rather than undertake the

mapping procedure in *Disk*. The space required by *Mem* and *Disk* for systems using 32 and 64-bit references or IDs is summarised in Table 6.1, assuming a 3 index system.

Strategy	32-bit (bytes)	64-bit (bytes)
Disk	36	72
Mem	24	48

TABLE 6.1: Size per triple given 32 and 64-bit IDs and references

Mem's advantage in terms of space is not all it appears, however. A 32-bit reference-based application can usually address no more than 4GB of RAM. This is not enough to store a very large quantity of information. On the other hand, a 32-bit *Disk*-like system running on a 64-bit computer can address a practically unlimited quantity of RAM: it is limited only by the number of IDs it can generate. A 32-bit ID space supports over 4 billion IDs, which, assuming the level of reuse seen in the UniProt dataset, is enough for 26 billion triples: enough for the foreseeable future. In practice, then, a 32-bit ID space is sufficient for *Disk*, where it is not for *Mem*.

A further issue is worth considering: having a separate object for each triple generates a lot of small objects. In most JVMs, objects have a minimum overhead of $(2 * Reference\_Size)$. For a Java-based implementation, then, the relative size per triple for *Mem* grows, as shown in Table 6.2.

Strategy	32-bit (bytes)	64-bit (bytes)
Disk	36	72
Mem	32	64

TABLE 6.2: Size per triple given 32 and 64-bit IDs and references on a Java-based implementation

Finally, there is a potential for *Disk* to be reduced in size by omitting repeating values (Weiss and Bernstein, 2009). For example, in a POS-ordered index, high cardinality Predicates will be repeated very often. If it were possible to eliminate such repeating values, the size of the *Disk* indexes could be shrunk dramatically. Table 6.3 shows the effect of this, using the UniProt dataset to estimate how much repeating data could be eliminated across the three index set. This figure (labelled *Disk (min)*) is idealised, since it assumes no overhead is required to enable the omission of repeating values.

Strategy	32-bit (bytes)	64-bit (bytes)
Disk	36	72
Disk (min)	20.8	41.6
Mem	24	48
Mem (Java)	32	64

TABLE 6.3: Size per triple including an ideal value for *Disk*

For in-memory stores, then, the *Disk* strategy is a clear winner for datasets smaller than 26 billion triples, particularly for any implementation that includes a small object overhead. Any overhead caused by the requirement to implement a node/ID mapping will not come close to overwhelming *Disk*'s advantage. When datasets larger than 26 billion triples come into existence, the *Mem* strategy will likely have some advantage in space consumption on systems with no object penalties.

A further concern with the *Mem* strategy is cache performance. In order to perform a search in an index, it is necessary to compare the content of different triples. This inherently requires that the triples be dereferenced, which will inevitably involve a large quantity of cache misses. It is likely, then, that the performance of *Mem* will be considerably worse than that of *Disk*. This observation parallels the discoveries in Rao and Ross (1999), where it was found that T-Trees, which attempt to save space by storing references to data rather than holding it inline, performed substantially worse than B-Trees with inline data.

A final point in favour of the *Disk* architecture is that it supports distribution, where *Mem* does not: a reference-based system inherently assumes that an object can be found somewhere in memory, unlike an ID-based one. Even if a shared memory system were implemented across the cluster of machines, the architecture has poor characteristics for distribution: since triple objects would be distributed across the cluster, 'dereferencing' them would prove prohibitively expensive.

Given these observations, it was decided to adopt the *Disk*-style model of data normalisation, using a monotonically increasing, 32-bit integer ID to identify unique nodes. The advantages and disadvantages of integer and hash IDs are discussed in detail in Section 3.2.2.2, but in this case the hashing approach is infeasible: a 32-bit ID space gives an unacceptably high risk of ID collision with only tens of thousands of nodes.

6.2.2 Overall Structure

In order to satisfy the requirement of a fast find time, it was clear that tree indexes were not a suitable solution. Hashes are attractive for their excellent find performance, but experience overwhelming overhead when acting as a composite, multi-level index: Figure 6.1 shows a simple example of a multi-level index, and it's easy to see how the number of index structures could blow up as the dataset increases in size. Given the wildly varying characteristics of the different orderings of RDF data, an adaptive solution was required: one that provided composite indexing where it would be beneficial, while ignoring it where the cost would be excessive. This approach allows for fast find times and low overheads.

The proposed structure features multiple levels of indexing: that is, given an SPO-ordered index, a level 1 (L1) index covers every Subject in the dataset, while a level

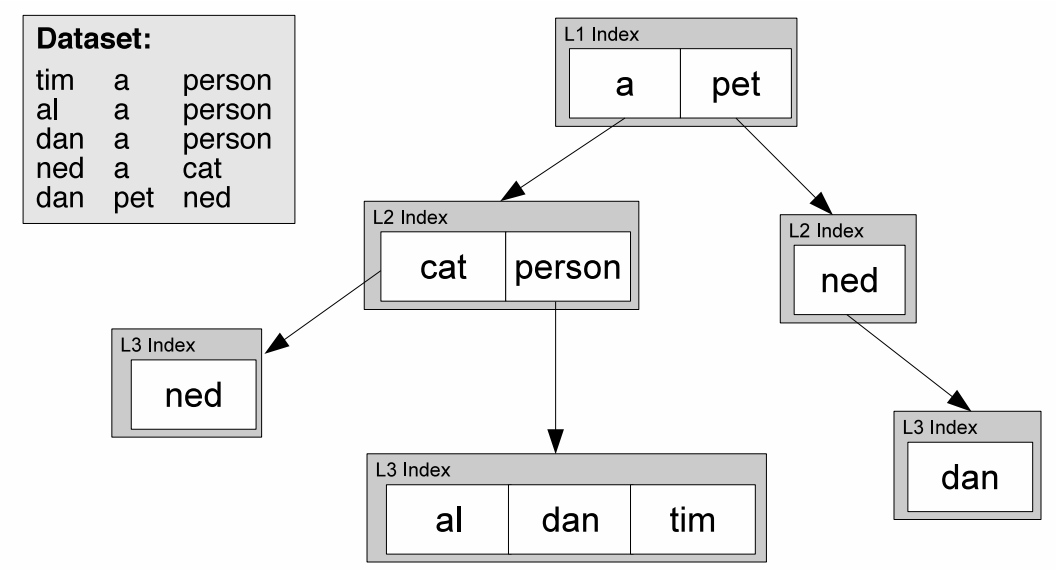


FIGURE 6.1: A full multi-level index structure, in POS ordering, over a simple RDF dataset

2 (L2) index covers every Predicate related to a given Subject. Finally, a level 3 (L3) index covers every Object related to a given Subject-Predicate combination. How many of these levels are used depends on the way the data behaves. Consider Figure 6.1, for example: for this data, one might decide that maintaining a separate L2 and L3 index for the triple 'dan pet ned' is wasteful. As a result, the L3 and L2 indexes are merged into one, as shown in Figure 6.2. This approach substantially curtails overheads, at the cost of forcing the creation of a greater variety of data structures, or ones that are flexible enough to support merging.

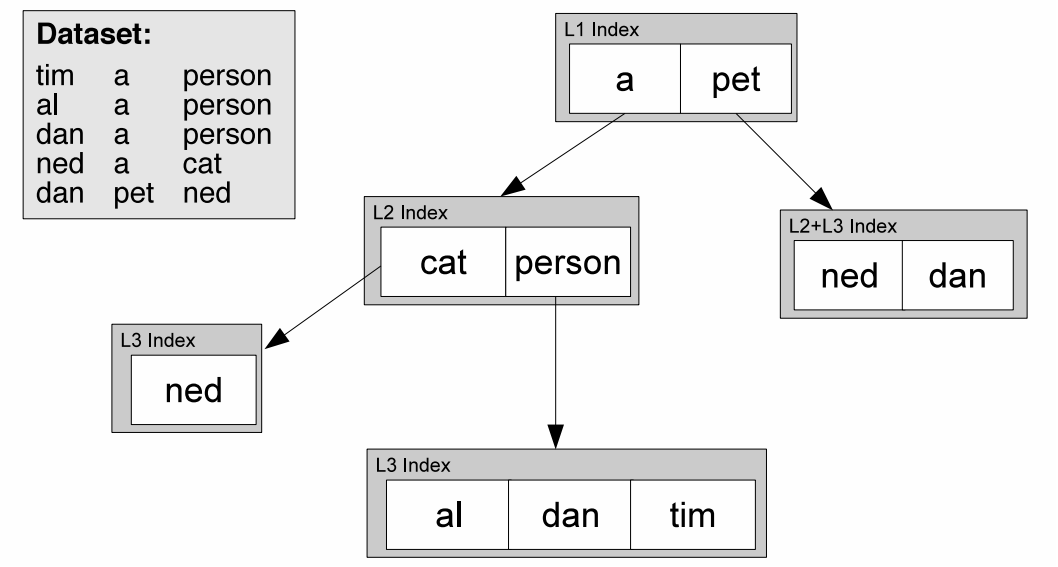


FIGURE 6.2: An adaptive multi-level index structure, in POS ordering, over a simple RDF dataset

In this model, assuming a three index store (SPO, POS, and OSP-ordered indexes), one would expect the following characteristics, based on the results found in Chapter 5:

- All three would feature an L1 index.
- SPO would require little or no L2 or L3 indexing, as Subjects are generally of low cardinality: it is practical to simply iterate or binary chop through the data associated with a given Subject.
- POS would feature a large quantity of L2 and L3 indexes: Predicates are of almost universally high cardinality, while PO pairings are also often of high cardinality.
- OSP would have a substantial requirement for L2 indexing, but almost none for L3: Objects are often of mid-high cardinality, while OS pairings are almost always extremely low cardinality.

An alternative approach would be to simply use distinct index structures for each of the three orderings, given their very different characteristics. An adaptive strategy, however, has the advantage that it can manage datasets that do not conform to the expected norms.

A multi-level design also offers the opportunity to eliminate the cost of repeating values, as described in Section 6.2.1. Since each L2 index is uniquely associated with an L1 value, it isn't necessary for the L1 value to be repeated inside the L2 index. Given this fact, as long as the L2 and L3 indexes have a very low overhead, a multi-level indexing strategy can save a substantial amount of space, providing a kind of implicit, free compression.

Finally, the multi-level design offers another substantial advantage: the freedom to choose, at each level, what kind of indexing structure to use. One might decide that a hash is appropriate for the L1 data, but want a B+Tree for L2 data, depending on the structure of the chosen dataset. Further, providing the ability to switch between such structures automatically would allow the system to tune itself without manual intervention. The decisions for what data structures to use are described in Section 6.3.

6.3 Per-Level Index Choices

This section considers the merits of various indexing strategies for each different level of the data structure. The choice of index can dramatically affect overall performance and size, and so picking a good strategy is extremely important. It is not required that each level have only one, fixed type of index, and the best choice will depend partly on the use case.

6.3.1 Level 1

The requirement of the L1 index structure is simply to provide a pointer to an L2 index, or, if no further indexing is required, a data structure that is able to hold a flat list of the second and third attribute values.

The most immediately obvious choice for an L1 index is a hash table. They offer fast, amortised O(1) insertion and retrieval, with a minimum of cache misses and branch mispredictions. They do, however, have a significant overhead associated with them: depending on type, hash tables can only be filled between 50-75% of their array size if they wish to maintain performance. Further, and more importantly, the table must store both the key (node ID) used to index into the table, and the pointer to the L2 index. Assuming a hash table with a maximum utilisation of 70%, with a growth factor of 2, using 32-bit pointers, a hash table will be 285-571% of the size of the raw data it stores. For reference, a B+Tree, for reference, would experience slightly lower overhead, at an average of just over 300%.

A less obvious, but simpler, alternative to the hash table is direct mapping. Direct mapping means simply using the ID being searched for as the index into an array of pointers. If the array uses a simple growth algorithm (such as increasing by a factor of 1.5 every time growth is required), direct mapping offers O(1) insertion and retrieval with an exceptionally low constant factor. Since the ID is the index, there is no need to store the ID keys in the array. Further, growing the array is much cheaper than growing a hash-backed array: hash table growth requires a re-hashing of every inserted element, while a direct mapping array can be increased in size by simply allocating a new, larger array and performing a simple memory copy from one to the other.

The overhead of direct mapping is dependent on the extent to which nodes are reused across attributes. The maximum overhead occurs when no Subject nodes are ever used as Objects or Predicates, no Predicates are ever used as Subjects or Objects, and so on. The minimum overhead occurs when every Subject appears as a Predicate and Object in other triples, and so on. Assuming a three index system, the average per-index overhead factor can be calculated as:

$$overhead = 1-((S\_Node\_Count+P\_Node\_Count+O\_Node\_Count))/(Total\_Nodes*3))$$

Taking the UniProt dataset as an example, this overhead factor is 0.52: the array to support the direct mapping will on average be 52% larger than the amount of real data in the array. Including a 25% average overhead to allow the array to grow in amortised O(1) time, a direct mapping array will be, on average, 185% of the size of the real data it stores.

Direct mapping is a poor fit for a predicate-ordered index: for example, UniProt has only 127 unique properties, and so a direct mapping approach is vastly inefficient in terms of

space. The average case, however, is substantially better than alternative approaches. In general, then, direct mapping requires around half of the overhead of a hash table or B-tree, and is dramatically quicker for both read and write operations, making it the index structure of choice for the L1 index.

6.3.2 Pointers and FixedBuckets

For the subsets of the dataset that do not require indexing beyond L1, it is necessary to store their additional attribute values somewhere: on an SPO-ordered index, for example, the Predicate and Object data needs to be stored. The most obvious approach is to place that information in an independent array, and have the L1 index store a pointer to that array. Find operations are performed by simply iterating through or binary chopping the array (if it is sorted). This simple approach is illustrated in the context of an SPO-ordered index, with no L2 or L3 indexes, in Figure 6.3. Note that for the remainder of this section, examples work in the context of an SPO-ordering.

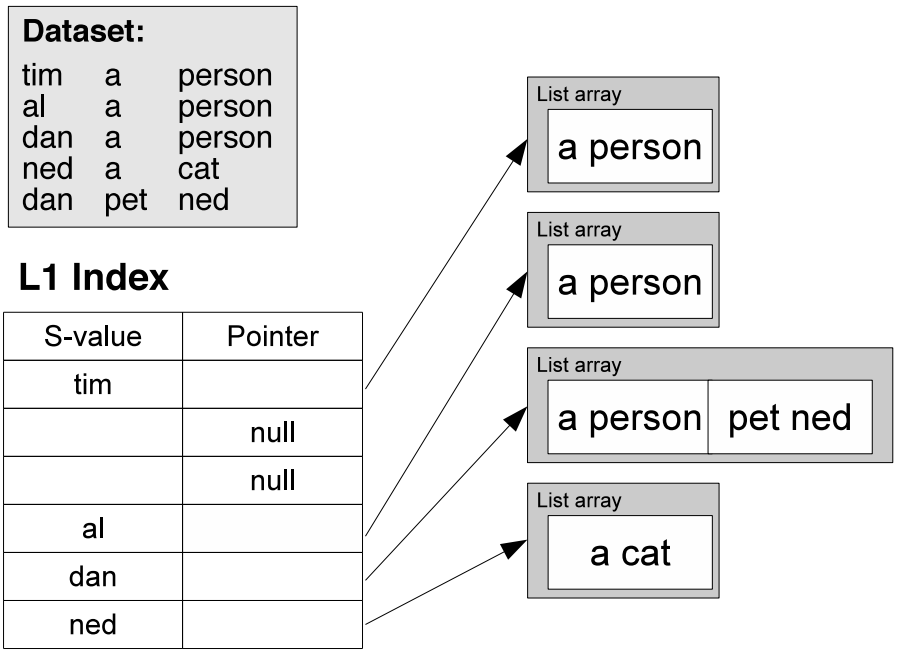


FIGURE 6.3: An SPO-ordered index with no L2 or L3 indexes

This independent array approach has a variety of flaws. Firstly, it has a substantial amount of overhead. For each Subject, it is necessary to store an array length counter, a pointer to the array, and any object overhead. When the array is storing 10 items or less, this adds up to a substantial amount: 8 bytes for the pointer, 2-4 bytes for the array length, and 16 bytes if there is per-object overhead similar to that seen in Java: 10-28 bytes on 8-80 bytes of data. Secondly, the data is non-contiguous: if one wishes to iterate over all the data, a vast number of cache accesses will be triggered. Further, this

approach has the potential to spread small amounts of data all over the virtual memory space, causing poor usage of the TLB.

These problems can be mitigated by a flat array approach. Instead of storing the data related to each Subject without an L2 index in a separate array, the data for all Subjects gets placed in one central array, as shown in Figure 6.4.

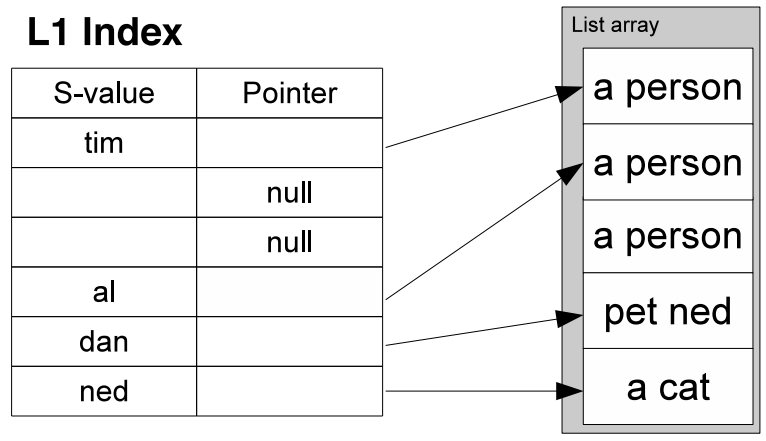


FIGURE 6.4: An SPO-ordered index with single array

The single array approach solves many of the multiple-array issues. Iterating over the data is now extremely fast, and the usage of the TLB is much improved. Overhead is much lower: the object penalty is now insignificant, and (if there are fewer than 2^{32} triples being stored in the array), a 4-byte integer can be used to index into the array, rather than an 8-byte pointer. The overhead on 8-80 bytes of data is now merely 6-8 bytes.

Unfortunately, this approach has some substantial disadvantages. Firstly, it isn't possible to use the data structure for more than 2^{32} triples without increasing the size of the pointer index. More importantly, this array is subject to fragmentation: as data is updated or deleted, holes will appear in the array that must be tracked. This approach is thus difficult to use effectively for dynamic stores.

A modification of this approach can work effectively, however. If a few arrays are used, each of them containing fixed size buckets of data, the problem is solved. The fixed-size buckets stored in each of these arrays are called FixedBuckets.

In this approach, any Subject that has only one PO pairing associated with it will have its data stored in one array. Any Subject which has two PO pairings will have its data stored in another array, and so on. The approach used for pointing into a FixedBucket array is quite simple. Using a 32-bit integer (called a *location*), the first 4 bits indicate the array, while the remaining 28 bits indicate the position of the bucket in that array. This means that each FixedBucket array can store up to 2^{28} buckets prior to overflowing. For reference, the largest FixedBucket array produced by the 2.8 billion triple UniProt

dataset is less than 1/4 of this size. In ideal circumstances, the first fifteen FixedBucket arrays could store over 34 billion triples. The FixedBucket approach is illustrated in Figure 6.5.

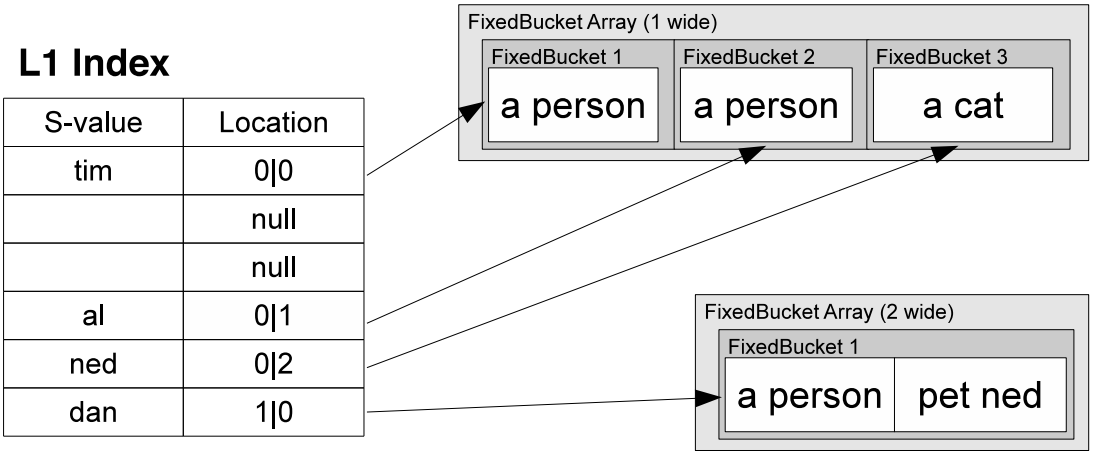


FIGURE 6.5: An SPO-ordered index with a FixedBucket array approach

If another PO pairing is attached to a given Subject, the data associated with that Subject is 'promoted' to the array which stores the next bucket size up, while the opposite occurs for deletion. Performing a promotion is simple:

- Copy FixedBucket (*FB1*) from current array (*array1*) to the tail of higher array (array2), inserting new element as required.
- Update *location* pointer to reflect *FB1*'s new position.
- Copy FixedBucket (*FB2*) from the tail of *array1* to *FB1*'s old position.
- Update *location* pointer to reflect *FB2*'s new position.

Note that in order to practically perform this operation, each FixedBucket must also store the Subject to which it is attached: moving *FB2* requires updating the pointer that points to it, which cannot be easily discovered without the Subject. The effect of an update is illustrated in Figure 6.6. In this update, 'al' buys a new fish, which he calls 'bob'. His data is thus promoted to the higher FixedBucket array.

The FixedBucket approach has excellent characteristics. They are very fast to iterate over, group related data effectively, can store a very large number of triples, and do not suffer from fragmentation. Their overhead is small: just 4 bytes for the *location* pointer and 4 bytes to store the Subject ID on 8-80 bytes of data. Note that in a bulk-built, non-updateable version, the Subject data would not be required. Further, the size of the FixedBucket is implicitly encoded in the location pointer, making it very fast (and space overhead-free) to retrieve statistics on data stored in FixedBuckets.

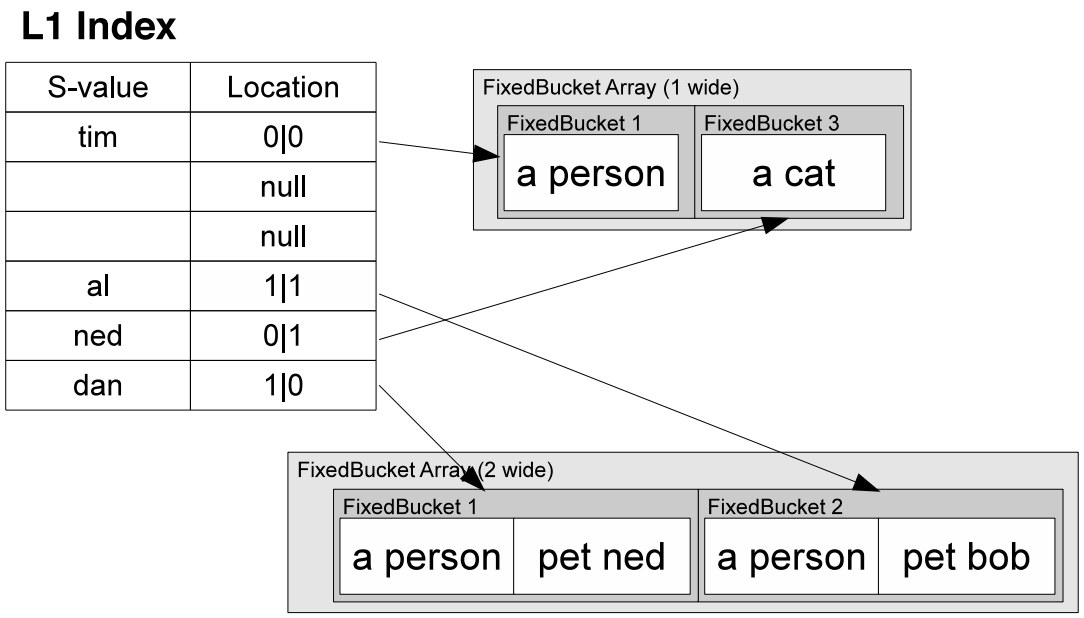


FIGURE 6.6: An SPO-ordered index with a FixedBucket array approach: update illustration

6.3.2.1 Physical Layout

In practice, the physical layout of a FixedBucket can differ from the conceptual layout depicted above. There are two candidate physical structures for FixedBucket arrays (depicted in Figure 6.7), depending on the use case. *In-order* stores the S value at the start of each bucket, followed by sorted PO pairings. This layout works well for element-at-a-time retrieval: all accesses inside the bucket are performed sequentially, and the layout is simple. Buckets can be easily rearranged.

The alternative structure, *vector* is designed for block-at-a-time retrieval. All the Subject values are stored in a block at the start of the array. Each FixedBucket then stores all its Predicate values followed by all its Object values (again in pairwise sorted order). This approach is designed for vector-based query engines, which will want to retrieve groups of values at a time. Using this layout, one can use fast block memory copy operations to copy all the Predicate values into one vector, and all the related Object values into another. This is not possible with *in-order*. *Vector* is more complex to update and grow than *in-order*, but this cost is relatively minor. *Vector* has further special-case performance advantages when compared to *in-order*: these will be described in more detail in Section 6.3.3.

6.3.2.2 Limitations

One obvious limitation of the FixedBucket approach is that due to the width of the 4-bit 'array' portion of the pointer, the size of a FixedBucket cannot exceed 16. In practice,

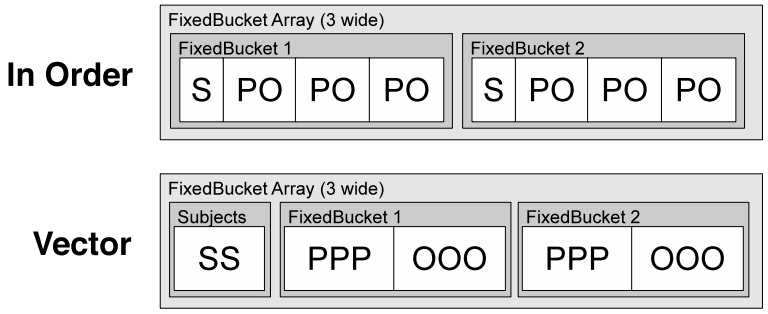


FIGURE 6.7: FixedBucket Layouts

the overhead of storing larger buckets in a separate object is quite small, so that is the approach taken. The highest value of array (15) is reserved, and is used to assert that the data related to that Subject is stored in an independent object. That object may be a flat list, or an L2 index, depending on how much there is of it. The 'position' portion of the pointer indexes into an array of references to these objects. Figure 6.8 shows how this would work when the Subject 'dan' has a lot more pairings attached to it.

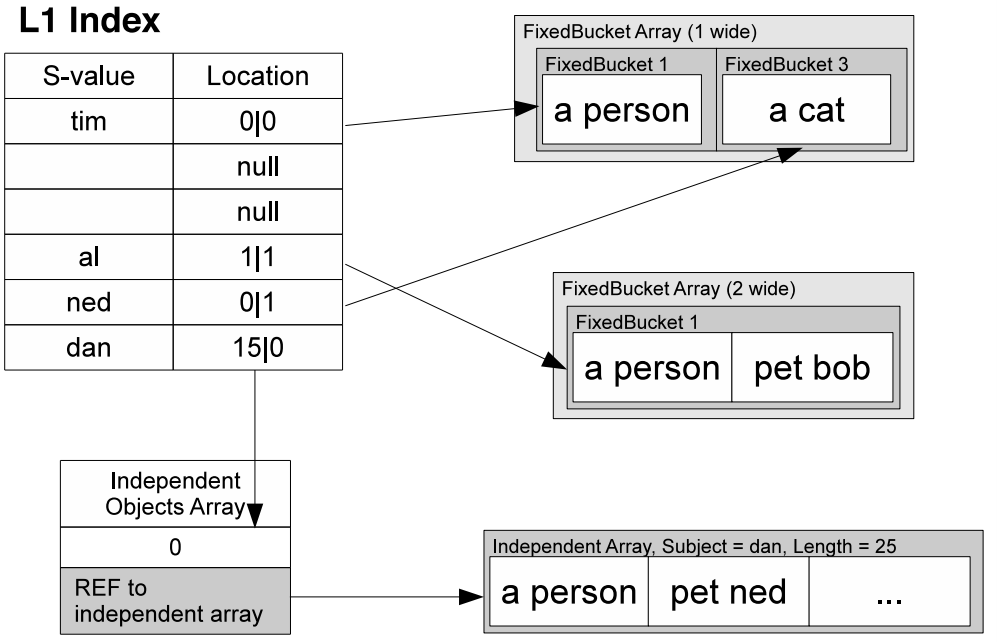


FIGURE 6.8: An SPO-ordered index with one independent array object

Overall, then, the direct mapping + FixedBucket solution offers excellent performance with low overheads. It provides a way to distinguish between data that is indexed further and that which is not, and a way to link to L2 indexes, the structure for which is discussed in Section 6.3.3.

6.3.2.3 Alternative FixedBucket Array Structure

The current FixedBucket Array structure is very simple: a flat array of FixedBuckets. To increase the size of the array, a new array is allocated, and all the existing FixedBuckets copied to it. To maintain amortised O(1) insertion performance, the new array is 1.5 times bigger than the old one, giving an average of 16.7% space wasted in a FixedBucket Array. This space can be substantially reduced using an alternative FixedBucket Array and location pointer structure, shown in Figure 6.9.

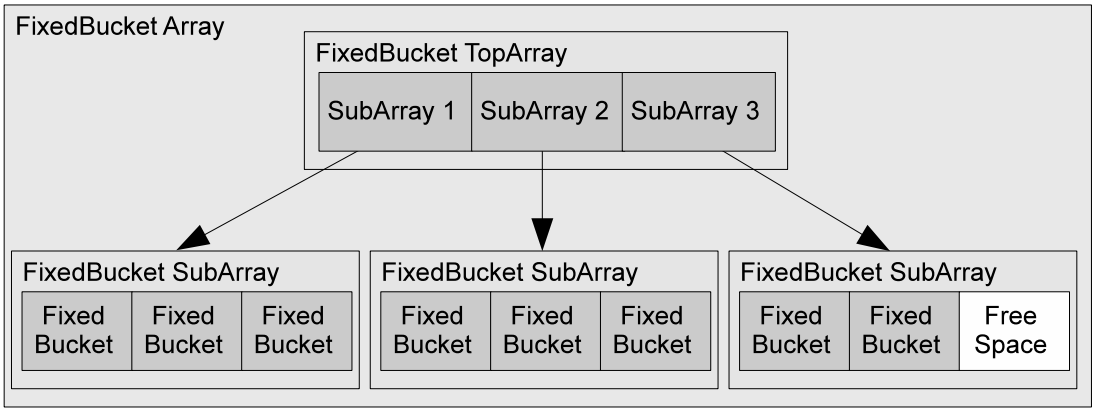


FIGURE 6.9: Alternative FixedBucket Array Structure

In this new structure, a FixedBucket Array essentially morphs into a two dimensional array. The first dimension (or TopArray) simply holds references to SubArrays. In this approach, a SubArray is allocated, and grows up to a maximum size as data is added. Once this size is reached, a second SubArray is allocated, and so on. This approach means that when growing, only the current, unfilled SubArray needs to be reallocated, and the maximum wasted space is only $0.5 * SubArraySize$.

In order to implement this approach, the location pointers need to be reworked to store SubArray information too. This approach can save a substantial amount of space, and substantially reduces overheads when growing the array, but also introduces the risk of an additional cache miss. As a result, AHRI does not currently implement this approach, although it is worthy of future consideration.

6.3.3 Level 2

The direct mapping approach used in the L1 index does not work for level 2 structures. An L2 index will contain substantially fewer entries than an L1 index, and yet the potential range of IDs is the same: the overhead is thus dramatically greater.

As a result, AHRI falls back to a hash-based structure for level 2. Aside from the use of a hash to locate data related to a given ID, the structure of this L2 index, known as

an *L2Hash*, is remarkably similar to that of the L1 index. Each ID is associated with a 32-bit *location* that points to a structure containing related data (in the context of an SPO-ordered index, the Objects related to an SP pairing). That *location* is again comprised of 4 bits *array* and 28 bits *position*, pointing to either a FixedBucket or an L3 index.

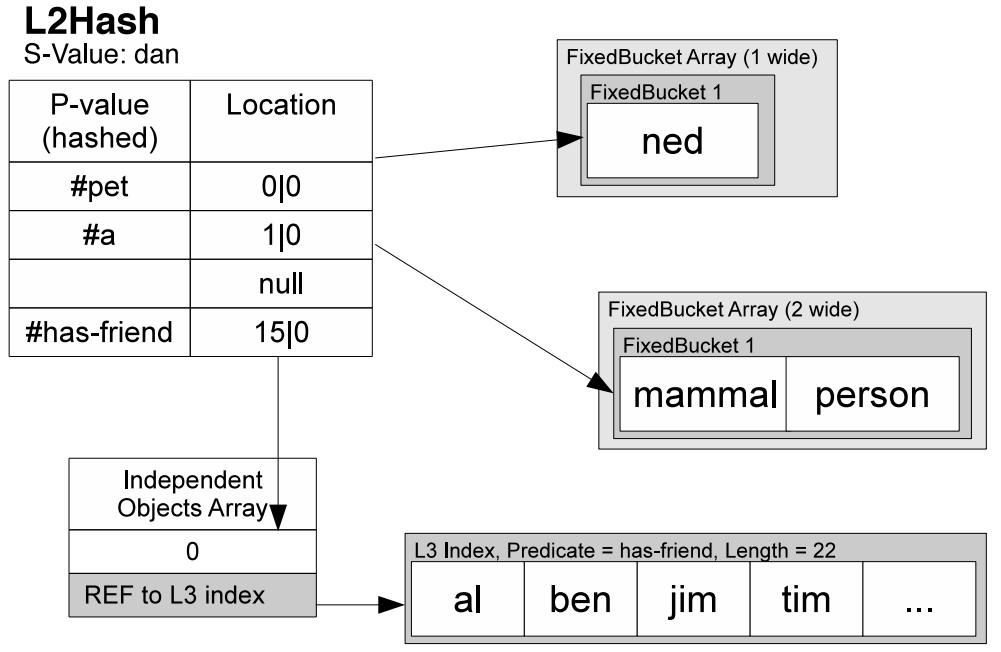


FIGURE 6.10: Level 2 hash index

For the purposes of compactness and cache-friendly linear access, the hash table used in this design is a linear addressing structure. The structure of the FixedBuckets used in L2Hashes is similar to that described in Section 6.3.2.1, but with a width of only one attribute value. The physical structure of the one-wide FixedBuckets is shown in Figure 6.11.

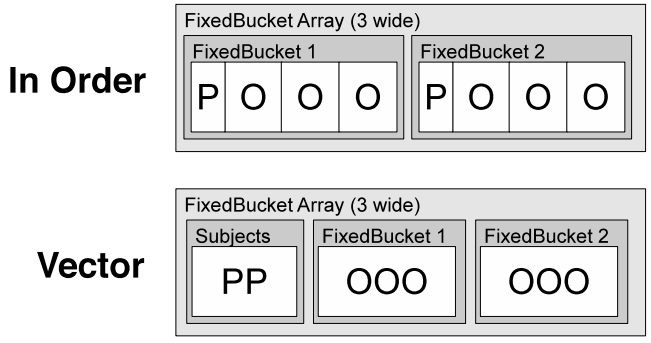


FIGURE 6.11: FixedBuckets with a width of one attribute value

As well as wanting to find individual elements, there is also a requirement to be able to return all the data within the L2Hash, to satisfy queries that fix only one value. Initial versions of the L2Hash iterated over all of the elements in the hash table, retrieving the

FixedBuckets or L3 indexes that they pointerd to in turn. This approach was extremely slow (about 1/4 the speed of a B+tree), as it offered poor locality of access as well as requiring extra calculation. An alternative approach is substantially faster: each FixedBucket array is iterated through in turn, with each FixedBucket within that array being accessed sequentially. After this, each L3 index is iterated through in turn.

The *vector* FixedBucket strategy has a useful optimisation when performing retrieval of all values in the L2Hash. For SP pairings (in an SPO index) that have a cardinality of one, the number of Predicate values is equal to the number of Object values. This being the case, when copying data out into vectors for a query processor, one can simply perform a straight memory copy of these Predicate and Object values into their respective vectors, across multiple FixedBuckets. This optimisation provides a substantial performance boost, particularly on OSP-ordered indexes, where most OS pairings have a cardinality of one.

The L2Hash design is highly efficient for POS-ordered indexes. The FixedBuckets offer generally low overheads, while the large majority of data in POS-ordered indexes will be held in L3 indexes: as long as these remain compact, then the index as a whole will perform well.

OSP-ordered indexes are something of a different matter, however. While many Objects are of fairly high cardinality, most OS pairings are of cardinality 1. This is the worst case for the multi-level index paradigm, which has fixed per-element overheads, relying on some of those elements to be high cardinality to produce good overall results. Since the fixed overhead is relatively high, a bucket width of one produces poor space efficiency. As a result, for OSP-ordered data AHRI produces high performance, but space efficiency only slightly better than a B+Tree. This is a less than ideal result: the OSP index is used by far the least out of the SPO, POS, OSP set, so the focus should be on compactness rather than performance. Future work will include experimentation with different, OSP-specific forms of L2 index. A delta-compressed B-tree would perhaps provide more appropriate characteristics.

6.3.4 Level 3

The level 3 index is used to provide support for a fast *contains?* operation, particularly important when asserting data. It must also allow very fast element-at-a-time or vector retrieval, while sorted order may be important for some use cases.

The most basic implementation is a flat list, which does provide excellent bulk retrieval performance. Unfortunately, as the index scales up, *contains?* performance becomes unacceptable. For very high *contains?* performance, a hash set is ideal: it is a simple O(1) operation to retrieve data, with low space overhead. Unfortunately, this approach

yields poor bulk retrieval performance: since any given slot may not be occupied, an unpredictable if statement is required over every element.

An improvement on these approaches is a combination: a list and hash set, with all data added to both. This approach provides excellent *contains?* performance and fast bulk retrieval, but at the cost of a substantially increased size overhead. Further, deleting elements from the index is still slow: it must be found in and removed from the list. However, in situations where a performant delete operation is not required, this index is very fast.

Finally, the B+Tree is worthy of consideration. At only one element wide it is possible to increase the fanout and lower overheads without hurting cache performance, and it also supports very fast iteration: as long as a reference is kept to the leftmost leaf node, there is no need to traverse the tree except during a *contains?* operation. Further, the B+Tree produces sorted output, which may be of benefit for some query optimisers.

It is worth mentioning that only one index out of the SPO, POS, OSP set needs to support a fast *contains?* operation: since all three attribute values are specified, any of the three indexes can be used. Particularly in situations where deletion performance can be slightly degraded, it is worth considering alternative data structures. One example is the wide-node tree structure used by Kowari (described in Section 3.3.1), which has relatively poor *contains?* performance, but provides extremely low overheads and good traversal performance. Such structures will be considered further in future work.

6.4 Complexity

Analysing AHRI's complexity is a slightly more involved topic than for many other data structures. This is due to its adaptive structure, where different components have different complexities. As a result, each of the different index structures is independently analysed below. Note that retrievals assume that a find operation for a single element is being performed: if multiple elements are being retrieved, a complexity of O(k) with the number of elements being retrieved must be added.

- The direct-mapped L1 indexes have a worst-case retrieval complexity of O(1). Insertion complexity is amortised O(1): that is, some insertions will trigger a growth of the table (O(n) with the size of the table), but this is amortised across all insertions for O(1) overall complexity.
- Retrieval from FixedBuckets has O(n) complexity with the size of the bucket: since FixedBuckets are always smaller than 16 elements, a linear search is used to retrieve elements. Insertion is amortised O(1): some insertions will trigger a growth of the FixedBucket Array, but this is O(1) amortised over all insertions.

- Retrieval from and independent array structures is O(log n) with respect to the size of the array: for these larger arrays, a binary chop is used to find data. Insertion is O(n) with respect to the size of the array, as these structures are kept sorted. The growth of these structures is limited to a maximum of *subn* elements, a value usually set at 100.
- Retrieval from L2Hash indexes, simple hash structures, is O(1). Insertion is amortised O(1). Again, resizing the table will trigger an O(n) table resize, but this is amortised across all other insertions.
- Retrieval from and insertion into L3 indexes depends on the structure chosen. For a hash set structure, retrieval is O(1), and insertion is amortised O(1). For the B+tree, insertion and retrieval are both O(log n).

These facts can be used to determine complexities for the structure as a whole. Since the growth of FixedBuckets and independent arrays are limited by small constants, their complexities in the context of the whole structure are effectively O(1). Since all structures other than the L3 indexes are O(1), the L3 index determines the complexity of the structure as a whole. Thus, if a hash set is used, AHRI's exhibits O(1) complexity for retrievals, and amortised O(1) complexity for assertions. If a B+Tree is used, AHRI's worst case complexity is O(log n) with the size of the largest L3 index for both retrievals and insertions.

Practically, using a B+Tree for an L3 index does not affect the performance of SPO and OSP-ordered indexes, since these structure make little or no use of such structures. In this case, AHRI will exhibit average-case complexity of O(1).

Overall, AHRI's complexity compares favourably with tree structures, which generally feature O(log n) scaling for insertions and retrievals with the total size of the dataset. AHRI largely features O(1) retrievals and insertions, and even where this does not hold, the scaling factor is the size of an individual L3 index, rather than the dataset as a whole.

6.5 Overall Design

Figure 6.12 shows the overall structural design of AHRI. Note that *subn* is the point at which AHRI decides to convert a flat list into an L2 index. Based on experimental observation, *subn* is usually set at a value of 100, but can be adjusted.

AHRI's design results in low overheads on typical RDF datasets, with the ability to adapt effectively to unusual situations. A particularly useful feature of AHRI's multi-level design is that the cost of find operations are largely predicated upon the number

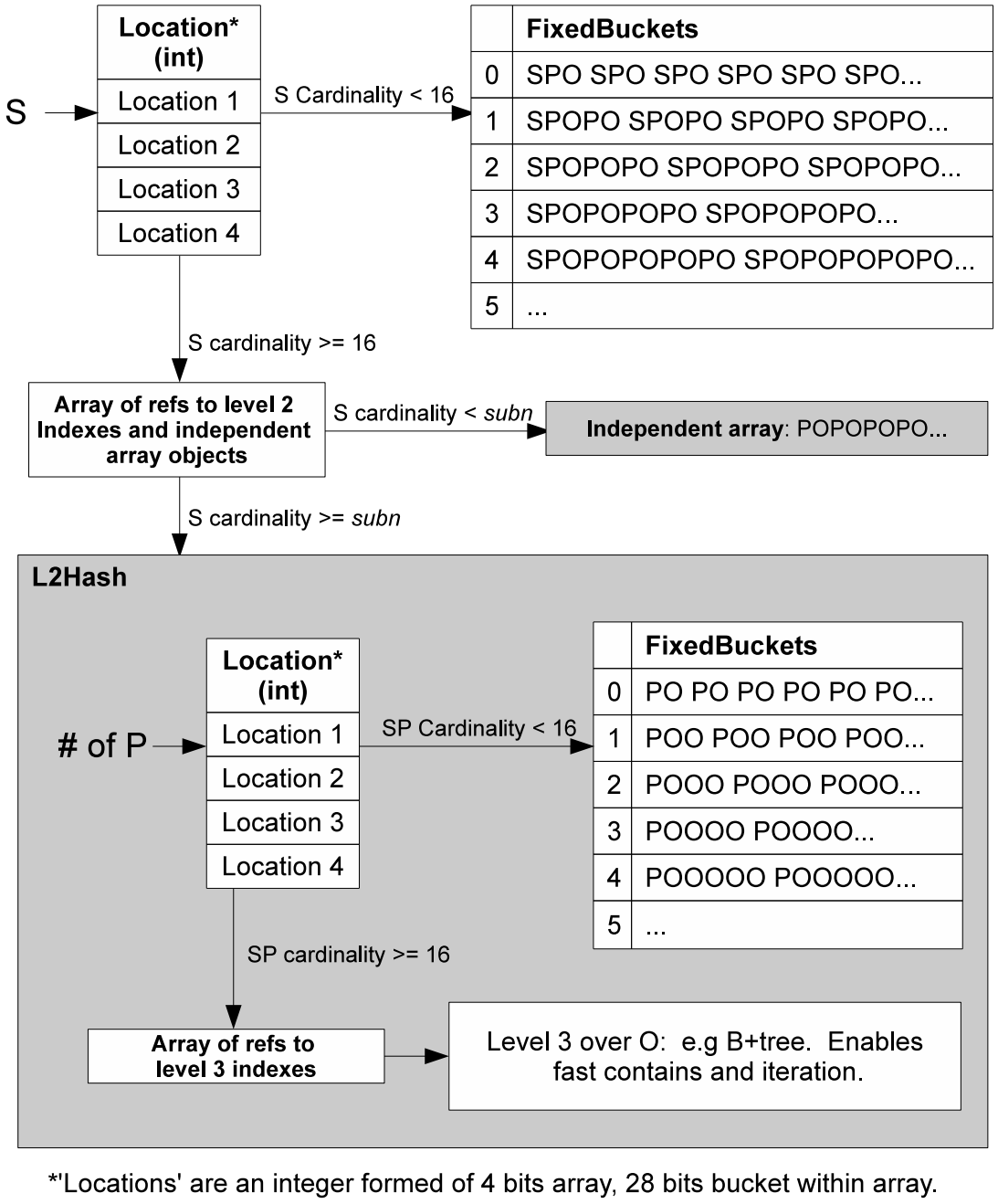


FIGURE 6.12: AHRI structural overview

of attributes being indexed over, rather than the size of the dataset: each level in the index can offer O(1) lookup times and insertion, with low constant factors. This is ideal for RDF, which features just three attributes. Thanks to the design of its sub-indexes, AHRI also features very fast iteration, meaning that whether a given retrieval matches many elements or few, AHRI can offer good performance.

Unlike many existing RDF data structures, AHRI also features the ability to add and delete data highly efficiently: FixedBuckets maintain zero fragmentation, while the other structures (hashes and B+trees) are also resist fragmentations effectively. The only

potential source of difficulty is the L1 index, where deletion of whole IDs can lead to fragmentation. This can be effectively mitigated, however, through the use of a free ID list.

6.5.1 Per-Index Suitability

AHRI's adaptive nature is important for controlling overheads. Take the example of SPO-ordered indexes: on the UniProt dataset, 85% of Subjects have a cardinality of 15 or less, and 75% have a cardinality of 10 or less. In such a situation, controlling per-Subject overheads is of paramount importance. AHRI accomplishes this by not performing unnecessary indexing, and through the use of very compact FixedBuckets. The combination of these factors results in very low overheads for datasets that are SPO-ordered.

AHRI also works well with POS-ordered data. Most data stored in a POS-ordered structure is in L3 indexes, meaning that AHRI cuts out a huge amount of repetition of Predicate and Object IDs: a kind of implicit compression. Due to the low quantity of predicates in the average dataset, the direct mapping L1 index is in theory very inefficient for Predicate-ordered data, but in practice this is rarely a problem: since Predicates are frequently mentioned, usually near the start of the dataset, they generally all have very low IDs, and the direct mapping table never has to grow. If this does become a problem, it might be worth either replacing the L1 index with a hash table in POS-ordered indexes, or taking steps to ensure that Predicates are always inserted into a reserved, low section of the ID space.

AHRI suitability for OSP-ordered data is slightly less clear cut. It is highly performant for this data ordering, but its advantage in terms of space is much more limited than for the other two index types. Since keeping the index size down is a major focus for the OSP index, it would be useful to consider alternative L2 index structures for OSP-ordered data, such as B+Trees or wide-node AVL trees.

Overall, however, AHRI fulfils the requirements listed in Section 6.1 admirably. It provides fast fin and iteration, fast insertion and deletion, is compact, and does not produce very large quantities of small objects. It also provides some ancillary benefits, which are explored in the following sections.

6.5.2 Statistics

As noted in Section 3.4.1, good quality statistics are essential to the effectiveness of a SPARQL query optimiser. One upshot of the AHRI design is that it provides a large quantity of useful statistics at very low cost. This is a result of the fact that AHRI implicitly groups a lot of data into FixedBuckets, each of which has a known

size. Since the number of independent data structures is generally very small compared to the number of FixedBuckets, it costs very little to store explicit statistics for those structures.

AHRI provides exact cardinality data for queries restricting over only one attribute. For queries restricting over two attributes, the situation is more complex: if the data to satisfy this query is stored in an L2Hash index, the statistics will be accurate. If not, they are estimated based on the first attribute's cardinality. Since data not being stored in an L2Hash is by definition of low cardinality, it is generally easy to make a good guess. The practical upshot of this is that when querying over two attributes, if the result set will be larger than the L2 index threshold *subn*, the statistics will be accurate. Otherwise, a good estimate is produced.

6.5.3 Value Skipping

In Neumann and Weikum (2008), the authors describe a mechanism for passing useful information 'sideways' through the query tree. This information can then be used to skip past irrelevant data and prevent it progressing further into the query plan, saving time. AHRI's L2Hash indexes support this better than tree-based structures that have no explicit index levels. Consider a POS index in which one needs to skip past a given O: using AHRI it is possible to skip up a level in the index hierarchy and move immediately to the next O. In a tree-based index index, it is necessary to either perform another find or scan until a new O is found. Considering that these Objects are often of high cardinality, and thus expensive to scan through, this is a useful capability.

AHRI's FixedBuckets and array indexes do not support such fast value skipping, but since they are used when the quantity of related data is small (less than *subn* elements), this isn't a significant loss.

6.5.4 Sorted Order

The sacrifice that AHRI makes in comparison to tree-based methods is that it does not guarantee that data is fully sorted in all cases. When retrieving all data from an L2Hash index, for example, it will read through all of its FixedBucket arrays in turn, followed by any level 3 indexes. Visiting buckets in sort order harms cache performance and results in a dramatic reduction in performance, as data is read in nonsequential fashion.

Assuming sorted level 3 indexes are used where available, AHRI displays the following properties:

- Queries that fix two attributes return sorted results.

- Queries that fix one attribute return sorted results if the result count is smaller than *subn*. Otherwise, grouped results are returned: from an SPO index, all Objects related to a Predicate will be returned together (and, indeed, sorted), but the order in which Predicates are returned is not guaranteed.
- Queries that fix no attributes (i.e. retrieve everything) return grouped results. In all cases, it is possible to cheaply determine in advance whether results will be returned in sorted order or not.

For most situations SPO-ordered indexes almost always return sorted results, while POS-ordered ones return sorted results over queries with two fixed attributes only, and OSP-ordered indexes are a mixture of the two. In practice, the importance of this behaviour depends on the join strategy being used. For merge joins, it is ideal to always retrieve sorted results, while for an index nested loops or hash join strategy sort order is less relevant.

It is possible to implement a fully sorted version of AHRI. Using a B+Tree or other sorted structure as an L2 index would provide the ability to traverse the index in-order. In the current implementation, this would lead to an unacceptably large slowdown, but this could be mitigated by a variety of strategies. Most importantly, the structure would have to be created in a language (such as C or C++) that supports software prefetching of data into cache: the fact that data was not being traversed sequentially would then matter less. Further, making an attempt to improve the sort order of FixedBuckets within their arrays would reduce the rate of nonsequential access. Fundamentally, however, AHRI is more naturally implemented as a partially-sorted structure.

6.6 Integration into the Jena RDF Toolkit

In order to perform tests that gave a good overview of AHRI's real-world performance, it was decided to integrate it into a real world Semantic Web query engine. Jena was chosen for this task, being a mature, popular toolkit with which the author already had some familiarity. This section describes the choices made in creating a prototype store that could use AHRI to answer queries.

6.6.1 Structure

The structure of the AHRI Jena Plugin (AJP) is fairly simple. It hooks into the Jena/ARQ[1] framework for parsing of RDF datasets and handling of SPARQL queries. AJP overrides methods in ARQ to perform query optimisation and answer segments

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

of queries called Basic Graph Patterns (BGP) (Prud'hommeaux and Seaborne, 2006). ARQ also handles formatting of results and allows queries to be answered over HTTP in concert with the Joseki[2] server.

AJP features three AHRI indexes, using the SPO, POS, and OSP orderings, and a string dictionary that maps between nodes and their respective IDs. This string dictionary is effectively two indexes. The first maps IDs to nodes using direct mapping: simply an array of references to nodes, where the node ID is the index of the node in the array. This allows simple, fast conversion between nodes and IDs. Nodes themselves are stored as Node objects in the Jena framework.

Converting from node to ID is only a slightly more complex task. AHRI uses a linear addressing hash table to convert from ID to node. The node is hashed, and the position in the table corresponding to the hash value is retrieved. That position stores an ID, which is then looked up in the ID to node conversion table. If the strings match, then the node is found. If not, the hash table moves to the next address, and retrieves the next ID.

This approach (illustrated in Figure 6.13) is very compact, but somewhat cache-inefficient, particularly in the case of re-probing on hash collision: it causes a potential cache miss every time it performs a lookup in the ID to node index, since these lookups are nonsequential.

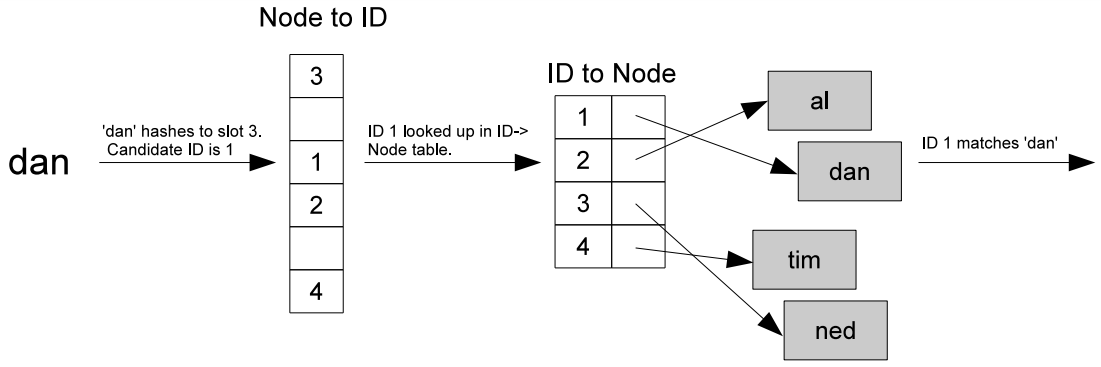


FIGURE 6.13: Node to ID index: compact but slow implementation

An alternative approach (depicted in Figure 6.14) is to store references to nodes directly in the node to ID index. This results in a minimum of eight extra bytes per entry overhead for the node reference. Overall, since node to ID conversions should be a relatively infrequent event, the slower, more compact approach was chosen.

[2]http://joseki.sourceforge.net/

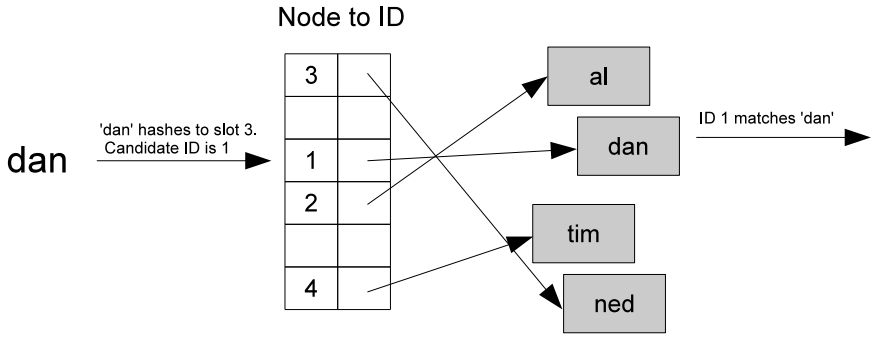


FIGURE 6.14: Node to ID index: larger, faster implementation

6.6.2 Optimisation

AJP's current implementation performs only basic static optimisation, based on the observation that in order to keep the working set of data small (and thus reduce costs), it is usually best to process triple patterns that return fewer matches sooner.

The query optimiser iteratively scores query triple patterns based on the elements they specify. OS is preferred, followed by SP, S, PO, O, P, and then no elements. This follows the typical cardinalities of each of these query patterns: a pattern that specifies Subject and Predicate, for example, usually retrieves substantially fewer elements than one that specifies Predicate alone. After a triple pattern is chosen to be next, any variables that the triple binds are marked as bound in the remaining triple patterns, each is re-scored, and another chosen, until none remain. In the future this approach will benefit from use of AHRI's statistics, rather than being purely static.

6.6.3 Join Strategy

AJP uses index nested loops as its only join strategy. For an in-memory environment, index nested loops are a compelling solution: they allow queries to be performed in small, pipelined chunks that require extremely small quantities of RAM. Due to the substantially lower cost of nonsequential access in RAM, the fact that INL processes a substantially fewer datums is a very substantial advantage. Further, INL parallelises highly effectively.

6.6.4 Answering Queries

AJP implements two query answering (QA) strategies. The first retrieves data from indexes in an element-at-a-time iterative approach, while the second retrieves vectors of information at a time. Much of the mechanics behind these two approaches are very similar, however.

The QA needs, for each triple in the query pattern, to perform a call to the correct index to serve that pattern, specifying what data is bound and what is not, and then to store the returned data in the correct order. Since this operation is performed extremely often, it is important to optimise it as far as possible. There are two obvious approaches to the problem. The first is to store, for each triple pattern, a value indicating what index to use. Then, when the pattern is reached during the QA process, use an *if* or *switch* statement to determine which index to call, and how to format the output data.

Unfortunately, this simple approach introduces a large quantity of unpredictable branch instructions, and is thus quite slow. An alternative is to define a class for each index ordering, inheriting from a common base type. For each triple pattern, an object is stored, created from one of these classes, that knows what index to call and what to do with attributes values passing to and from the index.

This approach is more elegant, but also not especially fast. It has two issues: firstly, calls to and from the object will be virtual, by virtue of the fact that the index that is used is not predictable. Secondly, such virtual function calls cannot be inlined. In an iterated environment, calls to these objects will be made every time a value is retrieved from an index, so these costs are extremely high.

The AJP QA (AQA) is built around a principle of minimising branch mispredictions and virtual function calls. It does this by exchanging what would be a virtual function call or unpredictable branch instruction for a data dependency on information that will almost certainly be cached, a lesser cost (Zukowski et al., 2006).

For each triple pattern, AQA pre-calculates which index it will be using, and stores a reference to the index, allowing it to call each index using the same code path. This is not sufficient of itself, however: since the same code path is being used to store each index, it is necessary to order the attribute values sent to and from the index appropriately: For example, when querying a POS-ordered index, it must be sent data in the order POS rather than the canonical SPO.

To accomplish this, AQA stores a value in an array for each unique node and node variable in a query. That value represents the current state of the node (which, in the case of fixed nodes, will never change). This structure is called the *values* array. There are two other arrays, called the *read* and *write* arrays, each of which index into the *values* array. The *read* array stores, for each triple, where to read the values with which to query the index. The *write* array stores where to put the values that come back. An illustration of a simple query using this method is shown in Figure 6.15.

Note that the values array has two special values at its head: JUNK and ANY. JUNK is the write destination for results that are of no interest: for example, in Figure 6.15, the Subject values retrieved are irrelevant, as the value is fixed. Retrieved Subjects thus get placed in JUNK. ANY, on the other hand, is a special value used when performing

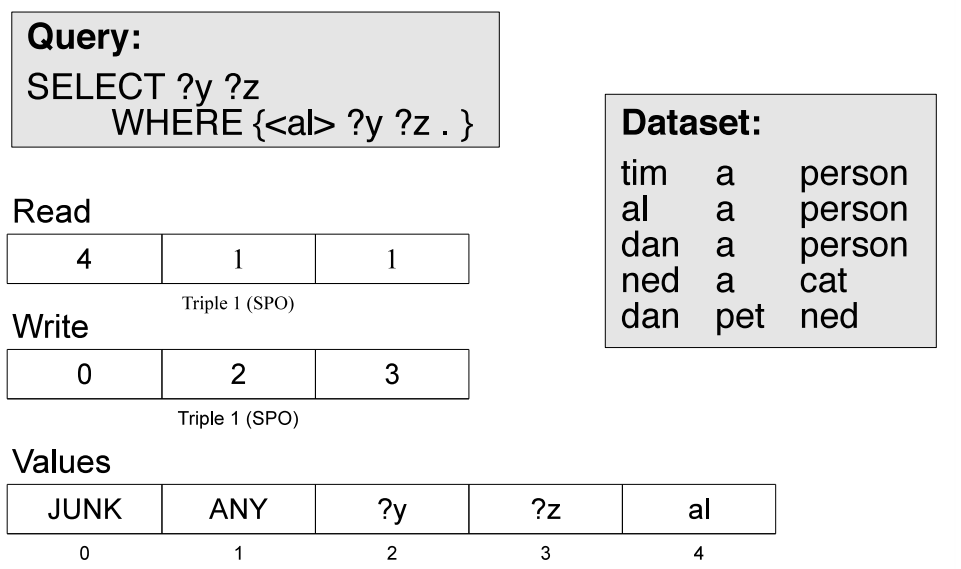


FIGURE 6.15: Query answering in AJP

```
MyTripleIterator = RetrieveFromIndex(Index_Reference, values[read[0]],
                                     values[read[1]], values[read[2]]);

while(MyTripleIterator.hasNext()) {
        values[write[0]] = MyTripleIterator.getVal1();
        values[write[1]] = MyTripleIterator.getVal2();
        values[write[2]] = MyTripleIterator.getVal3();
}
```

LISTING 6.1: Iterating over results from an index retrieval

a read from an index. A value of ANY indicates a variable. Listing 6.1 shows sample code to perform a request to the index and return all the results.

Using this approach, no virtual method calls or branch statements are required. The only unusual cost is in the data dependency required to find out which slots of the values array to read from and write to. Figure 6.16 provides a slightly more complex example. In this example, there are two triple patterns to answer, one using the POS-ordered index, and one using the SPO-ordered index. It is important to note that, as shown in the diagram, the values in the read and write arrays are ordered by the order of the index their triple pattern accesses: in this case, triple 1 is ordered by POS, and triple two by SPO.

Answering this query using INL is quite simple. The QA retrieves the first result for Triple 1, and stores the relevant data in the values array. As a result of this retrieval, there are now bindings for the ?x variable. The subsequent find operation for Triple 2 thus uses the bound values for ?x, rather than treating it as a variable. If the retrieval from triple 2 returns any results, they are iterated over, and result sets (or *binding sets*) are output as it goes. Once the results from Triple 2 are exhausted, it moves back to

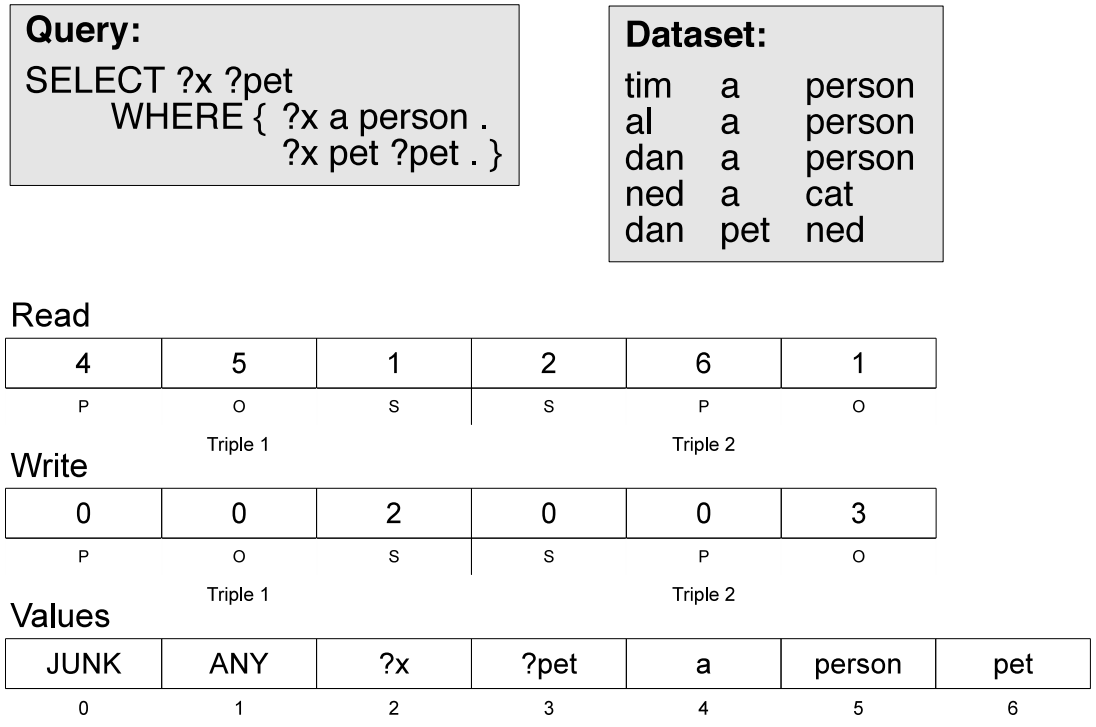


FIGURE 6.16: Query answering in AJP: a more complex query

```
Loop while CurrentTriple >= FirstTriple

        If (CurrentTriple.HasMoreResults)
                Get Next Result
                If (CurrentTriple is the last triple pattern)
                        Output Binding
                Else
                        Perform initial find operation for next triple
                        Move forward to next triple
                End If
        Else
                Return to previous triple
        End If

End Loop
```

LISTING 6.2: INL pseudocode

Triple 1, iterates to the next result (and the next value of ?x), and repeats. The generic pseudocode for this is shown in Listing 6.2.

As can be seen, the storage space required for this approach is minimal: simply enough space to store the values, read, and write arrays. This contrasts with a hash or sort-merge approach, which potentially requires a large amount of space.

6.6.4.1 Binding Sets

Outputting binding sets is generally a fairly simple matter. AJP implements a binding set that takes an array of node IDs (a copy of the variables portion of the *values* array),

and lazily converts them to Nodes.

One slight complication of this approach is that Jena references variables by their name rather than by an integer index into an array. To avoid unnecessary string comparisons to determine node equality, AJP remembers the order in which variables are asked for, and by default outputs variables in that order. It is designed to cope with changes in the order, although this results in suboptimal performance.

6.6.4.2 Optimisation

The standard INL algorithm described in Listing 6.2 has one notable flaw. This is illustrated by the query described in Figure 6.17, designed to match colleagues at a company who own the same pieces of sporting equipment: a crude way of determining that they might like to play games with each other.

```
PREFIX ex: <http://www.example.com/>

SELECT ?x ?colleague WHERE {
    ?x ex:works-at ex:some-company .
    ?x ex:has-colleague ?colleague .
    ?x ex:owns-sports-equipment ?equipment .
    ?colleague ex:owns-sports-equipment ?equipment .
}
```

FIGURE 6.17: SPARQL query that performs poorly using simple INL

Consider the effect if *?x* owns no pieces of sporting equipment, meaning that the query will produce no results with this binding of *?x*. Rather than going back to the first triple pattern and immediately binding a new value of *?x*, the standard algorithm simply goes back one triple pattern, and cycles through all of *?x*'s colleagues. Since this is clearly suboptimal, the algorithm was modified such that it would immediately retreat back to the first triple pattern, as shown in Listing 6.3.

6.6.4.3 Vector AQA

The standard iterated model of AQA is performant for most index structures. Unfortunately, AHRI has some special requirements that cause it to perform suboptimally in this environment.

Most structures, like trees, have one block of code that defines how to iterate over found results. In a B+Tree, for example, one simply sequentially iterates through all values in the leaf nodes that match the specified triple pattern. In a simple iterated environment, the function containing this code will be repeatedly called in order to retrieve the next result.

```
Loop while CurrentTriple >= FirstTriple

        If (CurrentTriple.HasMoreResults)
                Get Next Result
                If (CurrentTriple is the last triple pattern)
                        Output Binding
                Else
                        Perform initial find operation for next triple
                        If (NextTriple.FoundZeroResults)
                                Retreat to last triple with binding in NextTriple
                        Else
                                Move forward to next triple
                        End If
                End If
        Else
                Return to previous triple
        End If

End Loop
```

LISTING 6.3: Optimised INL pseudocode

Calling a function takes a certain amount of time: not a noticeable cost in most cases, but a significant issue when it is called very regularly. As a result, compilers make an effort to *inline* functions where it would be appropriate to do so: that is, the function call is directly replaced with a copy of the block of code in the function. Doing this eliminates the cost of the function call, at the cost of inflating code size (Ayers et al., 1997).

AHRI's iterators, unfortunately, do not lend themselves well to inlining. This is because AHRI makes extensive use of polymorphism: it has a variety of different internal index types, each of which has different iterators that all conform to the same external interface. This means that at compile time it is not possible to know what block of code will be called when retrieving the next result from an iterator, and it is thus not possible to inline the call. The result of this is a method call being performed for every result retrieved from an AHRI index. This causes a substantial slowdown for POS-ordered indexes, which are dependent upon fast iteration performance. SPO-ordered indexes are less badly affected, as they are generally dominated by find time.

Vector AQA (VAQA) was implemented to combat this issue. Instead of retrieving an element at a time from an index, it copies chunks (vectors) of data out at a time, into AQA-local structures. These local structures have a uniform access method, so calls to them can be efficiently inlined. VAQA provides improved overall performance when using indexes that don't inline effectively.

VAQA has an additional benefit: its ability to handle non-inlined index calls allows the use of a completely different data structure for each index. One might, for example, consider using a compressed B+Tree or wide-node AVL tree instead of AHRI for OSP-ordered data. This approach would not be efficient with the simple iterated model.

6.6.5 Limitations

The existing AJP implementation has a few limitations, largely as a result of the fact that it is designed as a prototype only. The most important issues are that the current support for OPTIONAL and FILTER keywords is very limited: they rely on Jena fallback mechanisms that require all node IDs to be converted back into nodes, then back into IDs again, as well as incurring other substantial overheads. This is a slow process, and means that the current implementation processes queries containing those keywords suboptimally. In particular, FILTER support could be substantially improved by adding support for the inlined IDs described in Section 5.4.2), which would reduce space requirements and dramatically speed up integer comparisons.

A second substantial issue is that the current implementation performs only a very little, static optimisation of queries. High quality query optimisation is a topic beyond the scope of this work, but the statistics generated by AHRI have the potential to be of great use to a high quality query optimiser, and could dramatically improve overall performance.

Finally, in the current implementation no attempt is made to curb the size of the string dictionary. As discussed in Section 5.4.2, there are a variety of strategies that can be used to reduce the size of the string dictionary, like common prefix elimination for URIs and general purpose compression algorithms for especially large literals.

6.7 Summary

This chapter has described AHRI, an adaptive in-memory index for RDF data. It is designed with an awareness of the structure of common RDF datasets, and the architectural features of modern processors in mind. As well as being designed to keep constant-factor costs (such as memory retrievals and branch mispredictions) low, AHRI provides attractive assertion and retrieval complexity, comparable to that of multi-level hash indexes, without the crippling inefficiency that these structures exhibit with regard to space consumption.

AHRI provides a variety of other advantages. It approaches size reduction in a pragmatic manner, informed by the analysis in Section 5.4 that showed that indiscriminate prefix elimation is costly. It also offers detailed statistics on the data it stores, providing excellent opportunities for query optimisers to improve their plans. The following chapter builds on this design work by testing AHRI, and showing that as well as performing well in theory, it offers excellent real-world characteristics.

Chapter 7

Evaluating AHRI

Chapter 6 described AHRI, a novel data structure for the storage and retrieval of RDF data. This chapter performs a detailed evaluation of AHRI, considering the extent to which it fulfills the requirements listed in Section 6.1. AHRI was evaluated against a variety of different index types, with the expectation that, thanks to its adaptive architecture, AHRI would improve upon the alternatives on insertion and retrieval performance, as well as occupying less space.

Section 7.3 describes initial tests, performed against a variety of different data structures. The machine these tests were performed upon offers the ability to inspect the rates of data cache and TLB misses, as well as branch mispredictions. This information helps to build a detailed impression of how AHRI performs the way it does.

Section 7.4 evaluates AHRI's ability to scale, using a machine with a large amount of memory to compare it against the fastest of the other candidate data structures. Section 7.5 goes on to test AJP, the Jena plugin that uses AHRI as an index type, verifying the assumption that improved index performance will have a substantial impact upon the overall performance of a SPARQL query. The plugin is evaluated using the standard Berlin SPARQL Benchmark, as well as with challenging custom queries over BSBM data, and a standard benchmark over DBPedia data.

Finally, Section 7.6 distills the information from these results into a discussion of AHRI's abilities in comparison to the other candidate structures.

7.1 Candidate Data Structures

The following candidate structures were indicated by the literature review described in Section 3.3:

- B+Tree with inlined IDs. (*B+Tree*)

- B-Tree with inlined IDs. (*B-Tree*)
- B-Tree with references to shared triple objects (*B-TreeRef*)
- Optimal BST with inlined IDs (*BST*)
- Bitmap index using Word Aligned Hybrid compression, with references to central triple table. (*Bitmap*)
- Jena Memory Model-like lightweight single-level, linear chaining hash with references to triple buckets, either arrays or hash sets (*Hash*)
- AHRI using inlined IDs, with B+Trees used as level 3 indexes (*AHRI*)

In general, these structures were implemented as standard. Data was asserted in SPO ordering, as is normal in the BSBM dataset, as well as most others. This has an impact on load time, as one can expect ordered data to load faster in indexes that are themselves ordered.

BST was implemented as a non-updateable structure, constructed from a sorted array of data. This approach meant that when iterating over data stored in a BST, there was a good chance that related data would be colocated in cache.

B-Tree and B-TreeRef were included in order to examine the difference in size and performance between structures that share triples, versus those that encode data inline. Their code bases are kept as similar as possible. None of the B-Tree variants have pointers in their leaf nodes: this is unnecessary, as all relevant data is encoded within the structure itself.

In every case, inlined IDs were 32 bits wide. For indexes that used shared triple objects, the triple object consisted of three 32-bit integers to represent S, P, and O. B-Tree variants used a node width of 100 triples, which provided a good balance between read and update performance.

7.2 Test Framework

The main evaluation followed a simple, standard procedure for testing data structures: the structures were placed in a test harness that directly measures the time taken to perform large quantities of operations, such as insertion or retrieval. This approach is common in the evaluation of new data structures, as can be seen in Rao and Ross (1999), Aguilera et al. (2008), and Hankins and Patel (2003).

The test harness reads triple data from files and asserts it into an index of a given ordering (for example, SPO or POS). In order to ensure that only the index assertion

performance was being tested, data files were converted into ID form and loaded into memory in advance, and loaded into the structures from there. The result of this is that retrieving a triple to load into an index only requires reading three sequential integers from an array. This eliminates factors such as disk I/O and parse time from the evaluation.

After loading data, but prior to performing queries or find operations, the test harness shuffles all the triples in the array describing the dataset. These triples then have 0-2 attributes changed to a value indicating a variable. The array is then used as the set of queries to perform upon the asserted data. This approach allows the testing of retrievals that do not, for example, specify an object.

In order to simplify the process of benchmarking a variety of indexes, the test harness allows a large amount of configuration. It features a noninteractive mode for the purposes of scripting, and an interactive mode that pauses after data load and after each test, in order to allow measurements of factors like index size to be performed.

For each index, over SPO, POS and OSP orderings, the following tests were performed:

- Load rate
- Size in memory (tested using TIJmp[1] memory profiler)
- Warmup (mixed query set, untimed)
- Queries with restriction over mixed attribute count
- Queries with restriction over 1 attribute
- Queries with restriction over 2 attributes
- Queries with restriction over 3 attributes (find operations)
- Find operations that will fail, restricting over 1, 2, and 3 attributes

The mixed query sets used 10% restriction over 1 attribute, 50% restriction over 2 attributes, and 40% restriction over 3 attributes. This is not intended to be an accurate representation of how many queries of each type will appear in a real-world query set, but rather to ensure that the JVM is not performing optimisations that are specific to 1, 2, or 3 attribute query sets.

Most of these tests are both standard and self-explanatory, but the last requires further clarification. Multi-level index types like AHRI can determine that a match will be not found quite early during a query: consider a find operation being performed on an SPO-ordered index, for example. If no matching Subject is found, the find operation

[1] http://www.khelekore.org/jmp/tijmp/

will fail immediately on the first level index. By contrast, a B+Tree still has to perform a full search. This factor is tested because it is especially important in a query engine using INL joins, as these will often perform searches for non-existent data.

When answering a query, each index is allowed to return data in three different manners. The first is an array: the test harness passes a large, empty array to the index, and the index fills the array with results from the query. The second is a simple, element-at-a-time iterator. Finally, the third is a vector-based iterator, designed specifically for AHRI to allow it to amortise the cost of its non-inlinable iterator calls, as described in Section 6.6.4.3. The tests in Section 7.3 use the array method of retrieval, as it was simple to implement in a performant manner for the wide variety of index types tested in that section.

As the field is narrowed substantially in Section 7.4, the simple and vector iterator models are used. These methods are most commonly used inside real DBMSs, and so provide further, more accurate insight into the real-world performance of the tested indexes.

7.3 Initial Tests

Initial tests were performed using a system of the following specifications:

- Dual 1.8GHz AMD Opteron, 1MB L2 cache per core
- 8GB RAM
- Linux kernel 2.6.32 (64 bit)
- 64 bit Sun reference JVM (version 1.6.13), run using options '-server -Xms7000M Xmx7000M'
- OProfile 0.9.4

Each index loaded 5 million triples generated by the Berlin SPARQL Version 2.0 benchmark tools (Bizer and Schultz, 2009), using the options '-fc -pc 14087'. Results measure only the time taken to insert data into the indexes: nodes were converted into IDs and loaded into memory in advance. Indexes were then tested in SPO, POS, and OSP order. Note that AHRI was configured using a B+Tree L3 index, and used the standard in-order FixedBucket layout.

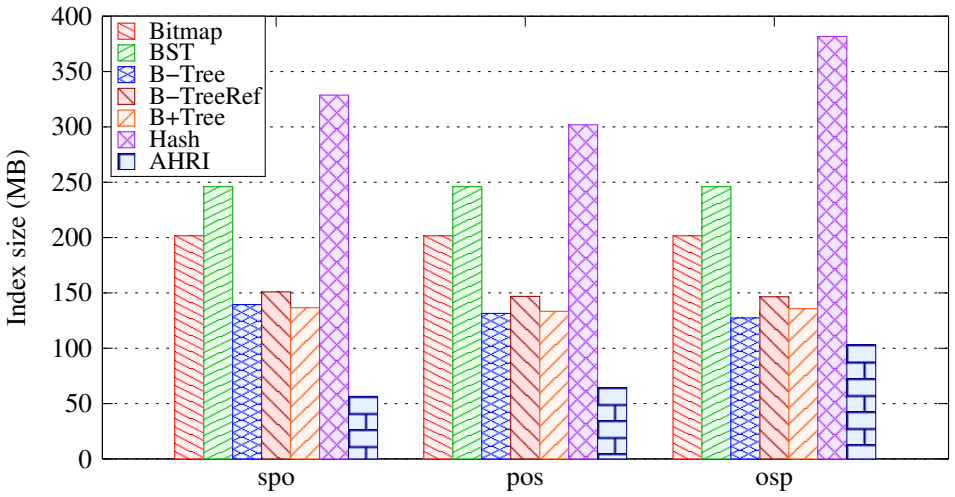


FIGURE 7.1: Index sizes for 5 million triples of BSBM data (lower is better)

7.3.1 Size

Since memory is a relatively expensive resource, it is important to consider the amount of space used by each index structure, as shown in Figure 7.1. Bitmap can use the same structures to provide SPO, POS, and OSP support, and so for the purposes of comparison, the total size was divided by three, and assigned to each of SPO, POS and SPO. Similarly, for structures that share common triple objects, the space used by those objects is divided by three, and assigned to each of the index orders.

Hash's lack of attention to overheads is clear in these results: it generates separate structures to store data related to each S, P, and O. This has disastrous results due to the quantity of Subjects and Objects that are of very low cardinality. AHRI shows similar degradation on the OSP index: OS pairings are typically of extremely low cardinality, and so AHRI generates a large quantity of data structures. Overall, however, AHRI requires substantially less space than the other index types, thanks to its policy of not storing repeating values, and producing very few objects.

The tree-based methods perform as expected: BST requires substantial additional space due to its pointer overheads, while B+Tree and B-Tree are broadly similar. B-TreeRef requires a little more space, but in an environment without object penalties for its shared triple objects would likely achieve parity. B-Tree and B+Tree benefit from their wide-node approach: they generate relatively few objects, all of a relatively large size.

Bitmap requires a substantial amount of space as a result of generating a large quantity of small objects: one for each distinct Subject, Predicate, and Object, as well as each triple object.

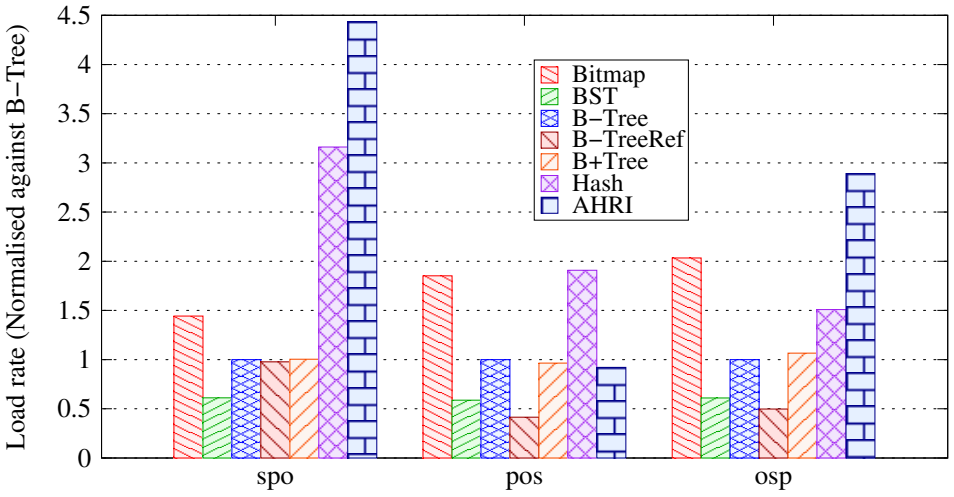


FIGURE 7.2: Load rate for 5 million triples of BSBM data (higher is better)

7.3.2 Load

Figure 7.2 shows load times. In order to improve graph readability, results other than index size are reported by normalising against the results achieved by B-Tree. Note that Bitmap and BST are built using bulk load methods, and do not represent the (much higher) cost of incremental update. All of the other indexes are built using incremental insertions, and so these results can be considered indicative of the cost of updating the structures, as well as the cost of bulk load.

It can be seen from these figures that AHRI provides particularly fast insertion. AHRI has a very fast search, and requires only relatively small movements of data to insert new values. AHRI's POS results are relatively poor because almost all insertion is performed into level 3 indexes: B+Trees. The other bucketed index, Hash, is also substantially faster than tree-based methods.

Of the tree-based methods, B+Tree and B-Tree produce broadly similar results, with BST falling behind. The results for B-TreeRef are of interest: it is competitive for the SPO index, but falls badly behind on POS and OSP. This is because the dataset is inserted in SPO order, and so triples are sequentially allocated in SPO order. Traversal of the SPO index is thus much more likely to yield contiguous access to triple objects.

7.3.3 Query

This section describes the query (read) performance for each index type. For each ordering, retrievals were performed restricting by one, two and three attributes, or a mixture (val1, val2, val3, mixed respectively). For the purposes of this test, each index returned results into an array that was passed into it. In order to improve readability

of the graph, results are normalised. Tables including the raw figures for each of these tests can be found in Appendix D.

Figure 7.3, Figure 7.4 and Figure 7.5 show the relative performance of each of the storage structures. AHRI dominates over the other indexing strategies, proving substantially faster in all but one case. Hash's lack of sub-indexing proves a substantial weakness: while it provides fast 'contains' checks, it is unable to index further into the large buckets produced by the POS and OSP orderings, and is forced to iterate over the entire bucket.

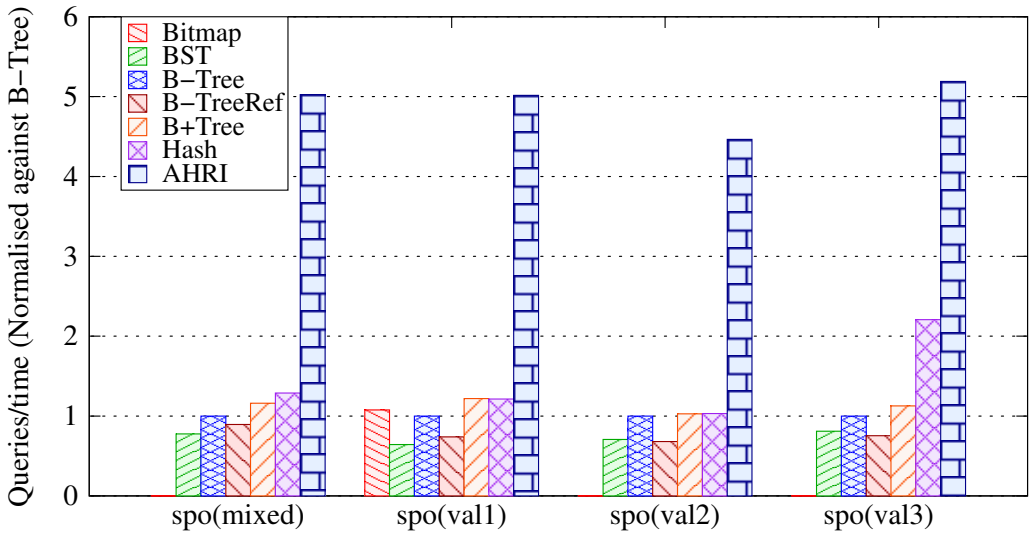


FIGURE 7.3: SPO query performance for 5 million triples of BSBM data (higher is better)

Figure 7.3 shows the results for each of the structures over SPO-ordered data. As expected, AHRI and Hash do well here. Due to the fact that Subjects are of almost universally low cardinality, this particular case plays to their major strengths: fast find operations.

By contrast, the tree based methods do poorly. This is not unexpected: performing finds is relatively slow in trees, and the find portion of the operation dominates the process of filling the array with the rest of the matching data.

BST is slower than most of the other options, due to poor cache performance. This same factor explains the poor results for BTreeRef: since it has to perform dereferencing to access a central triple table, it is more likely to miss cache. B+Tree provides the expected moderate performance gain over the next best tree implementation, B-Tree.

Finally, Bitmap's performance is especially bad. In order to return just a few results, Bitmap has to perform three hash lookups (to find the relevant bitmaps), AND the bitmaps together, and then iterate through the resulting bitmap and look up triples in the triple table. Bitmap never seriously challenges the alternatives, but does better as the number of elements to iterate over decreases, and the iteration time dominates.

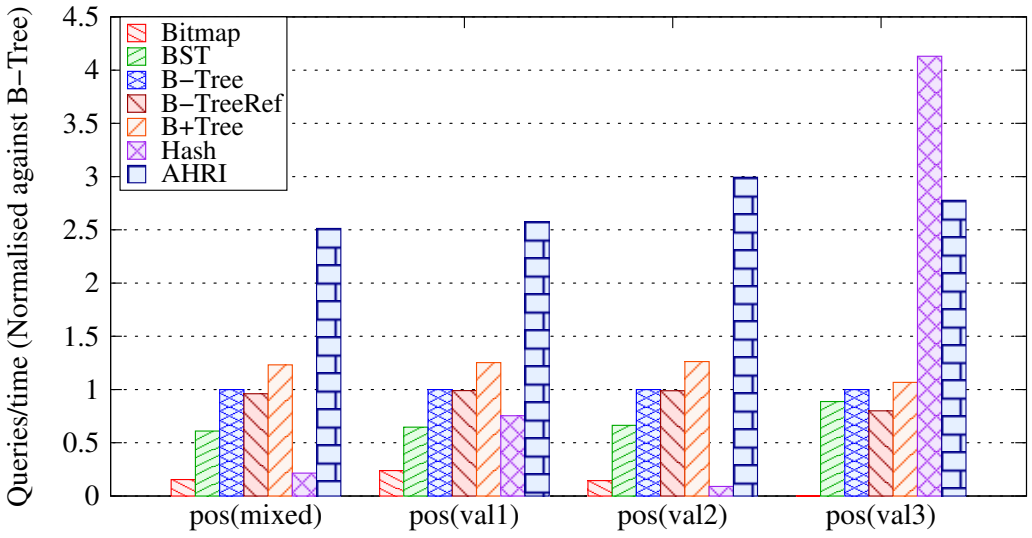


FIGURE 7.4: POS query performance for 5 million triples of BSBM data (higher is better)

AHRI offers slightly less improvement for POS-ordered data when compared to tree-based structures: tree structures, particularly those with wide nodes, are relatively performant for retrieving large amounts of sequential information. AHRI does gain, however, by the fact that it works with a substantially smaller quantity of data: in its L3 B+Tree indexes, it only has to read S values rather than the full triple. Further, AHRI does not have to perform tests during the iteration to determine if triples being read match the required attributes: it knows that when matching over a PO pairing, every element in the associated L3 index is required.

Hash fares very poorly in this test. It is fast when restricting over all three attributes, and capable enough when restricting by only the first attribute: slightly slower than most of the tree structures, because iterating over its hash sets requires an unpredictable branch statement. Unfortunately, its performance is extremely poor when restricting by two attributes. This is because it features no L2 index, and is forced, when trying to find a particular PO pairing, to iterate over all the elements attached to a given P.

Of the tree methods, B+Tree performs the best, as expected, again improving moderately over B-Tree. B-TreeRef improves over BST for this ordering. This is to be expected, as while retrieving the next triple from the B-TreeRef node incurs a potential cache miss, the same is true for moving to the next BST node. Bitmap improves slightly over its results for the SPO index, but is still uncompetitive.

Finally, the results for OSP show a similar trend to that in the other two orderings. AHRI is substantially quicker than the other candidate indexes, particularly in situations where the find part of the operation dominates any other work, as in *val2* and *val3* in this case.

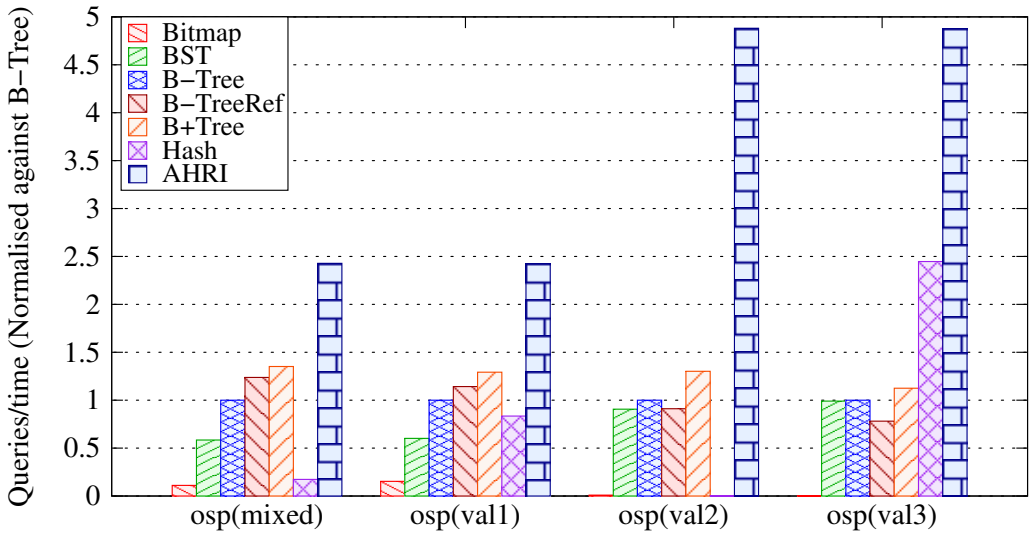


FIGURE 7.5: OSP query performance for 5 million triples of BSBM data (higher is better)

It should be noted that the results for this index are of lesser importance than for the first two, as it is generally used less. Future development will focus on reducing the size of OSP-ordered data rather than improving retrieval performance.

7.3.4 Failed Finds

The performance of failed finds is a significant issue in systems that use index nested loops joins, particularly in situations where a difficult query or poor query optimisation occurs. Consider the query in Figure 7.6, and assume that it is executed in-order. This query is designed to find all people at a company who manage to maintain a friendship despite differing opinions on Marmite. In answering this query, a number of bindings will be generated each for ?x and ?z, and the final triple pattern matches them up. This triple pattern will be called a great deal, and may fail a lot, so it is important that it performs well.

```
PREFIX ex: <http://www.example.com/>

SELECT ?x ?z WHERE {
    ?x ex:works-at ex:some-company .
    ?z ex:works-at ex:some-company .
    ?x ex:loves-food ex:marmite .
    ?z ex:hates-food ex:marmite .
    ?x ex:friend-of ?z .
}
```

FIGURE 7.6: Query to find friends at some-company with differing opinions on Marmite

The factor that distinguishes failed finds from normal find operations is the potential to catch them early on. For this example, consider two individuals, Dan and Bob. They both work at the company, but do not know each other, and have no other relationship. If one were matching using an OSP-ordered index, it would be possible to tell that the binding would fail upon searching for the subject and finding it not present.

Failed finds were simulated on the test harness by searching for non-existent IDs. To prevent the unwanted cache effects that would occur from using a single non-existent ID, non-existent IDs were simply the set of odd numbers ($0 < x <= (largest\_real\_ID + 1), x \mod 2 == 1$, while real IDs were even. Note that figures are not included for Bitmap, as this approach gives inaccurate results for this index type: the fact that the ID does not actually exist can be determined during the bitmap finding stage, without the need to actually compare bitmaps. The result of this is Bitmap appearing to perform unrealistically fast, and so this index type has been ommitted.

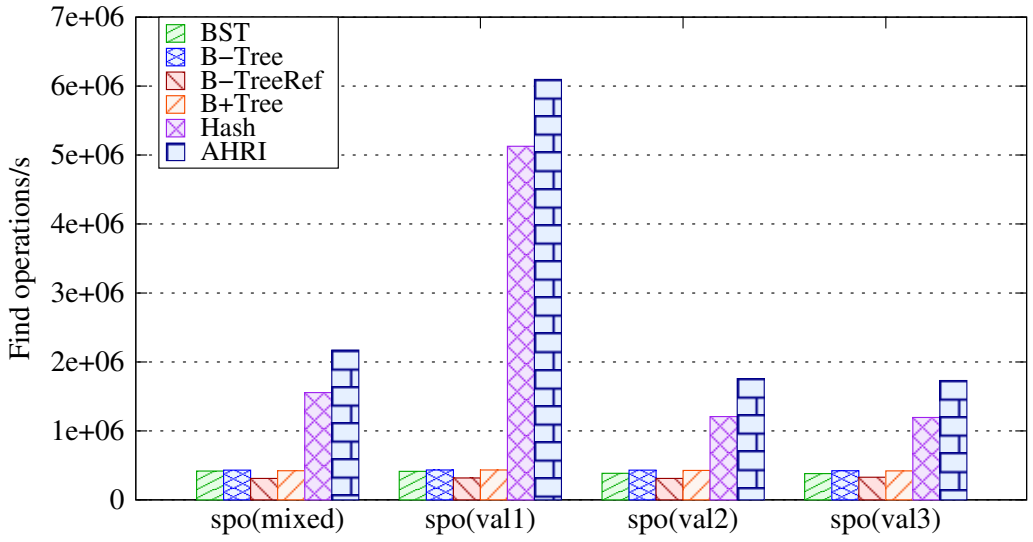


FIGURE 7.7: Failed find performance over SPO-ordered, 5 million triple BSBM dataset (higher is better)

Figure 7.7 shows failed find performance for SPO-ordered data. This graph shows the weakness of tree structures for find operations: they require a lot of processing to determine if a triple is contained within the graph. AHRI and Hash, on the other hand, perform very well.

For AHRI, if the subject simply doesn't exist, the find operation is extremely quick: it simply misses against the L1 index. If the predicate or object doesn't exist, the operation is somewhat slower: it is necessary to search through attached FixedBuckets.

Hash behaves similarly. It matches against an L1 index, and if a match is found either iterates through a small array to find a match, or hashes the triple object to find a match in a hash set. This approach results in excellent find performance.

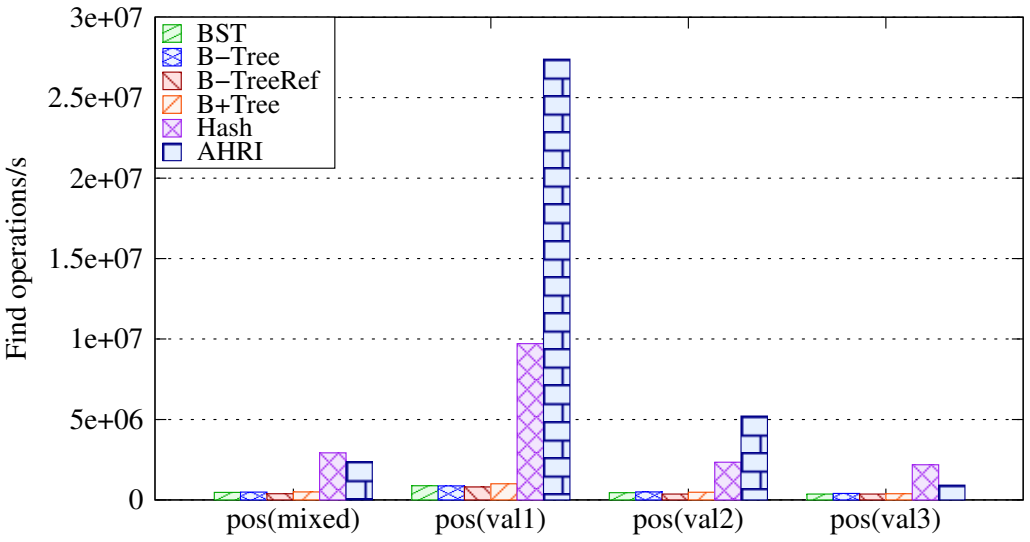


FIGURE 7.8: Failed find performance over POS-ordered, 5 million triple BSBM dataset (higher is better)

On POS-ordered indexes, AHRI is particularly fast on L1 failures: since there are fewer than 100 predicates in the L1 index, cache performance is extremely good. Its performance over L3 indexes is poor, however: an artifact of using a B+Tree to store the L3 data. Hash again does well here, showing its consistent performance with respect to find operations.

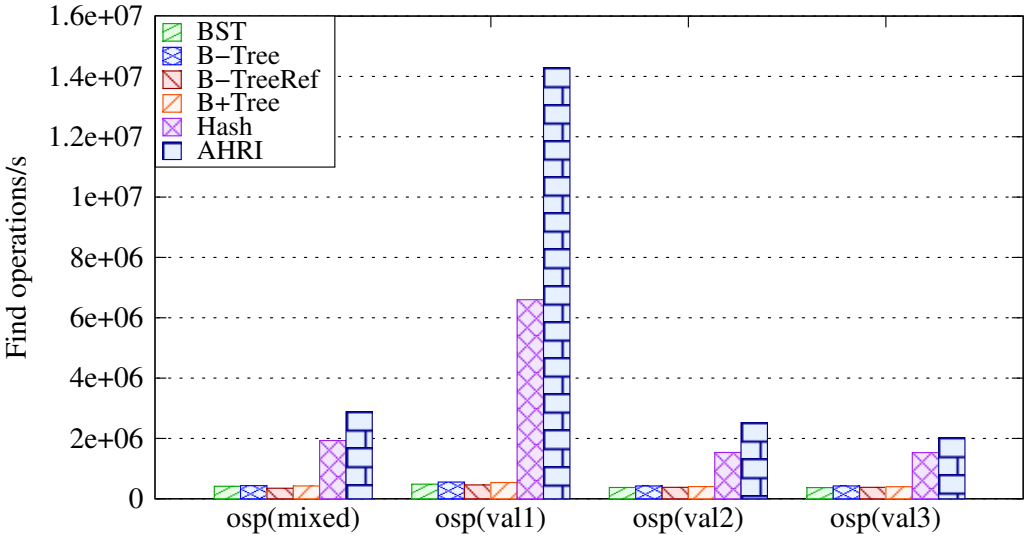


FIGURE 7.9: Failed find performance over OSP-ordered, 5 million triple BSBM dataset (higher is better)

AHRI is fast for OSP-ordered data: its L2Hash indexes can quickly determine whether a find operation has failed on the second attribute. Hash again does well, outshining the tree-based index types.

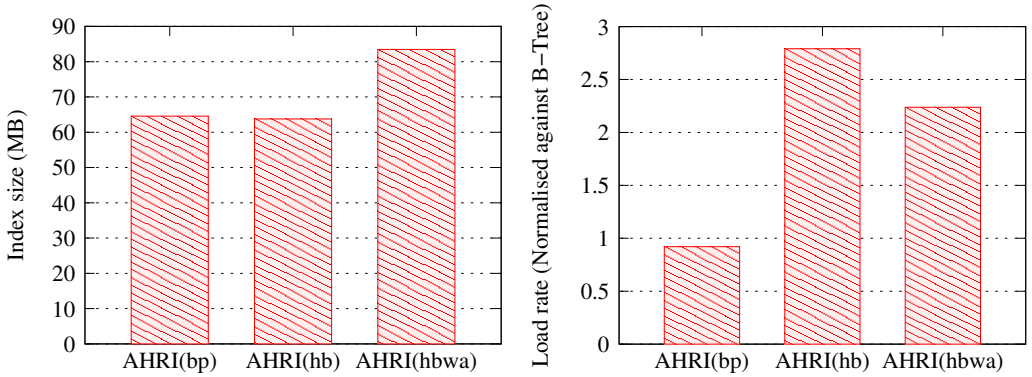
Overall, AHRI's performance for both failed and successful finds substantially outshines the other index types, particularly tree structures. When performing a find operation,

any of the SPO, POS, and OSP indexes can be used, since all three attribute values are specified. From these results, the SPO and OSP-ordered structures appear to do best for AHRI. OSP has an edge, but in a real-world system is used relatively little, and so is less likely to be cached. As a result, the SPO index was chosen as the means by which to perform find operations for AJP.

7.3.5 Alternative L3 indexes

As described in Section 6.3, AHRI can make use of a variety of different L3 indexes. The evaluation of AHRI concentrates on the B+tree L3 index (*AHRI-bp*), since this provides the most sorted output, and is hence useful in the widest variety of situations, but this section considers the alternatives: a hash set (*AHRI-hb*) and a hash set with an attached array (*AHRI-hbwa*), designed to improve the performance of range retrievals.

Both AHRI-hb and AHRI-hbwa use a linear addressing hash set with a load factor of 0.7, and a growth factor of 2. This gives an average utilisation of 52.5%. AHRI-bp uses a very wide node size of 1000 elements, optimising for size and read performance.



(a) Size (b) Load rate

FIGURE 7.10: Size (a) and load rate (b) on the POS index for alternative L3 index types, over 5 million triples of BSBM data.

Figure 7.10(a) and Figure 7.10(b) show the size and load rates for each L3 index type. As expected, AHRI-bp is substantially slower than the other strategies: inserting into a B+Tree is much more work than insertion into a hash set. AHRI-hbwa shows the expected small slowdown when compared to AHRI-hb, along with requiring significantly more space.

Figure 7.11 shows the read performance of each of the L3 index types. The results for POS-ordering are shown here, since POS-ordered data require heavy use of L3 indexes. AHRI-hbwa shows excellent all-round performance, using its array to perform retrievals when the third attribute is not specified, and its hash set to perform retrievals when it is. There are two caveats on this, however: firstly, AHRI-hbwa requires significantly more space than the other index types, and secondly, it cannot perform fast deletions

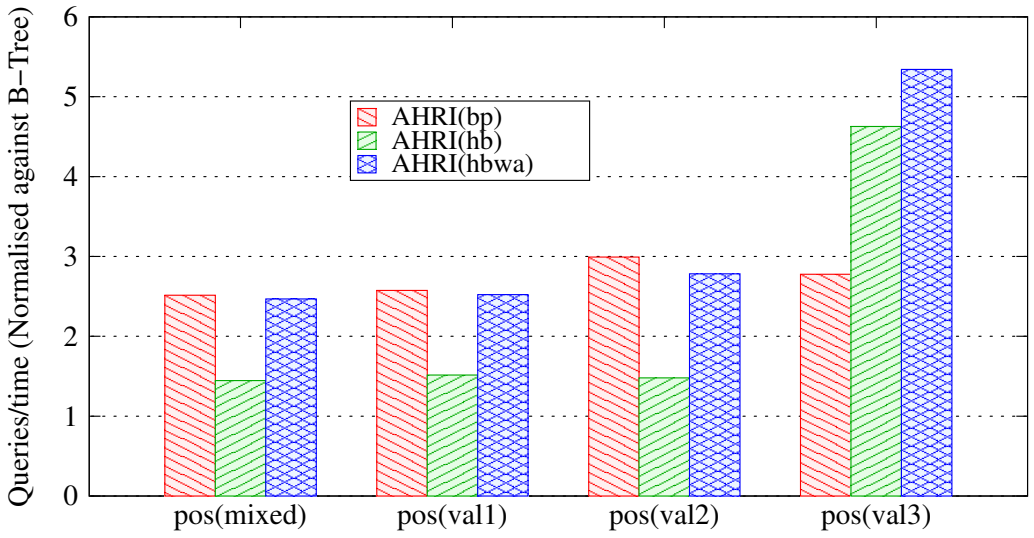


FIGURE 7.11: POS query performance for alternative L3 index types over 5 million triples of BSBM data (higher is better)

due to its array structure. It would be possible to modify AHRI-hbwa to perform fast deletions: the hash set could index into the array, rather than simply storing the value. This would slow down find operations somewhat.

Of the other types, AHRI-bp substantially outperforms AHRI-hb when iterating over the whole index, but is much slower when performing find operations. This is expected: finds are slow on a B+Tree due to having to traverse the tree, while iterating over a hash set is slow because each element requires performing an unpredictable comparison to determine whether the bucket is filled or not. AHRI-hb uses a smaller but comparable amount of space, so the preferred index type depends on the application to which it is being put.

7.3.6 Cache Performance

As described in Section 6.1, cache miss rate can have a substantial impact upon the performance of data structures being used on modern computers. Missing cache infrequently is thus a basic requirement of a performant index. This section analyses the data cache miss rate for each of the candidate data structures.

The statistics in this section were gathered using the OProfile[2] measuring tool. OProfile works by monitoring performance counters in the CPU. These counters increment every time a particular event happens: for example, an L2 data cache miss. Since it is obviously impractical to interrupt program execution for every such event, OProfile works by sampling: it sets an interrupt to occur every n times the event occurs. When the interrupt is triggered, OProfile records the currently running application, and attributes n counts of the event to that application.

[2]http://oprofile.sourceforge.net/

This sampling method obviously has the potential to introduce errors: a different application might be responsible for the majority of the cache misses. This is rendered unlikely by the choice of a high sampling rate (interrupting every 10,000 events), and the fact that the test harness is the only application putting a significant load on the machine.

For testing data cache misses, the following events were recorded:

- DATA_CACHE_ACCESSES (L1 data cache reads)
- DATA_CACHE_MISSES (L1 data cache misses)
- L2_CACHE_MISS, restricted to data miss only (L2 data cache misses)

This section considers miss rates relative to B-Tree, in order to make charts more readable. Raw data can be found in Appendix D.

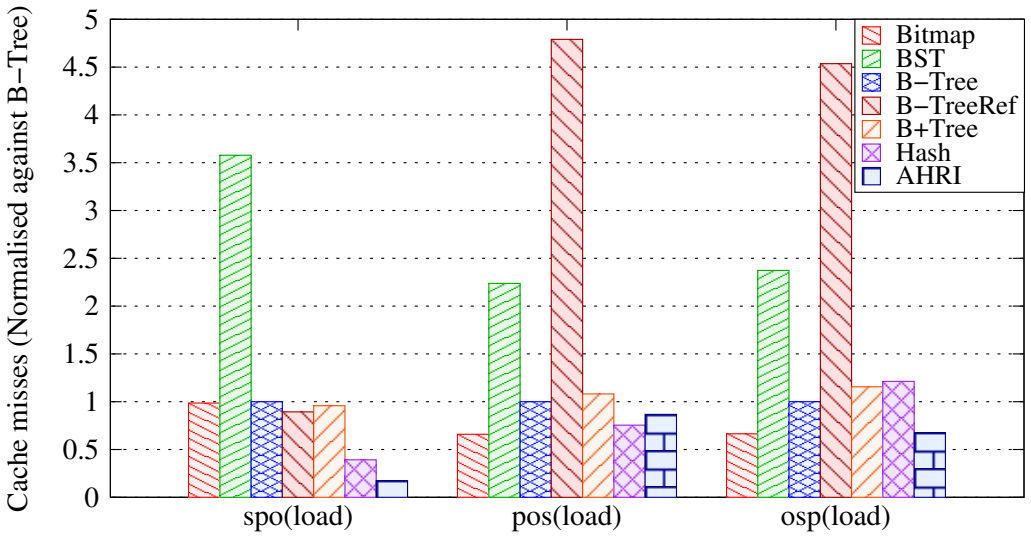


FIGURE 7.12: Cache misses when loading 5 million triples of BSBM data (lower is better)

Figure 7.12 shows the number of cache misses incurred by each of the different index types when loading 5 million triples of BSBM data. There are several interesting pieces of interesting information revealed by this chart. Firstly, AHRI performs very well in general. Particularly, on the SPO index, where data is inserted in-order, it misses very rarely indeed. As expected, it does less well on the POS index, where data is being inserted into a B-tree.

B-Tree and B+Tree perform comparably, with B-TreeRef being a substantial outlier. B-TreeRef does very well on SPO-ordered data, where, since the BSBM data is in SPO order, information is inserted in-order into the tree. This means that only a few nodes of the B-Tree are in active use, and each of the triples being compared against is likely to

be found in cache. The story is very different for the POS and OSP indexes, where data is inserted out of order. In this scenario, the point at which a triple is inserted into the tree is less predictable, meaning that the triples that are compared against in the tree are also less predictable. This results in a relatively low likelihood of these triples being found in cache. This poor behaviour explains the poor load rates found for B-TreeRef in Section 7.3.2.

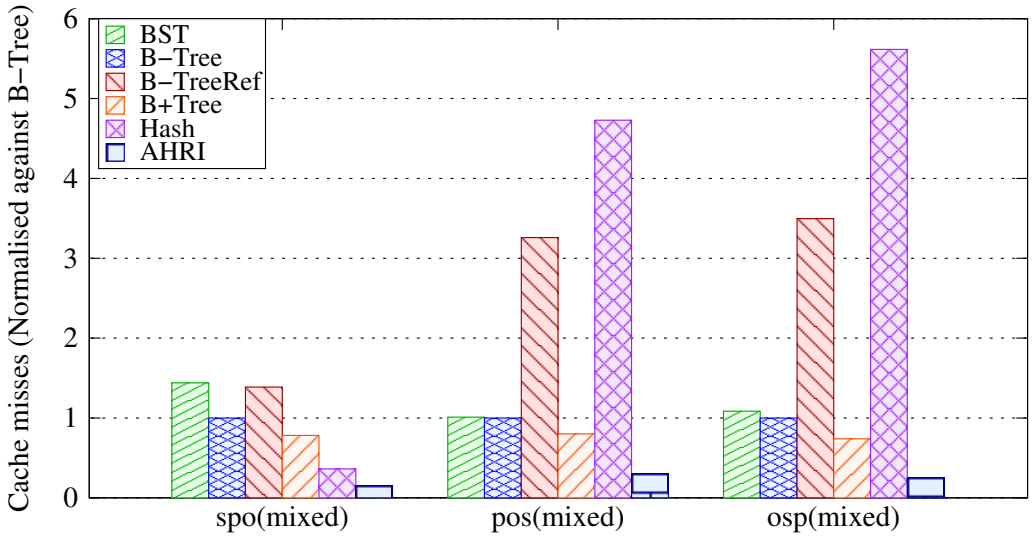


FIGURE 7.13: Cache misses when querying over 5 million triples of BSBM data (lower is better)

Figure 7.13 shows cache misses relative to B-Tree when performing retrievals against BSBM data. Bitmap is not considered in this graph, because it misses cache so much that it renders the other information hard to see. Note again AHRI's very low miss rate relative to the other methods, contributing to its excellent read performance.

Despite B-TreeRef's very poor apparent cache behaviour, it exhibits creditable read performance. A potential explanation for this is that while data was not in cache, it was already in the process of being retrieved (due to predictable access patterns enabling prefetching). B+Tree offers the expected minor improvement over B-Tree: iterating over data requires simply traversing a linked list rather than traversing the tree.

The results for Hash are also of interest. Retrievals over POS and OSP, which are dominated by iteration over found data, cause a lot of cache misses. This is again due to the use of shared triple objects, and is particularly bad due to Hash lacking a level two index, meaning that retrievals have to access an extremely large number of such objects when selecting over two fixed attributes. The cost of using shared triple objects is reflected in poor results for POS(val1) and OSP(val1), described in Section 7.3.3.

The relatively good results of BST are also quite notable. This is due to the manner in which BSTs are bulk constructed in these tests: BST sorts all the data prior to iterating over the sorted array, and creating tree nodes. Tree node objects are thus allocated

in-order in memory, thnks to the JVM's appending allocator. As a result, iterating sequentially over nodes in BST, a behaviour which dominates retrievals from POS and OSP-ordered structures, is very friendly to cache prefetching. It's notable that retrievals from SPO-ordered structures, which are dominated by tree traversal, are more subject to cache misses.

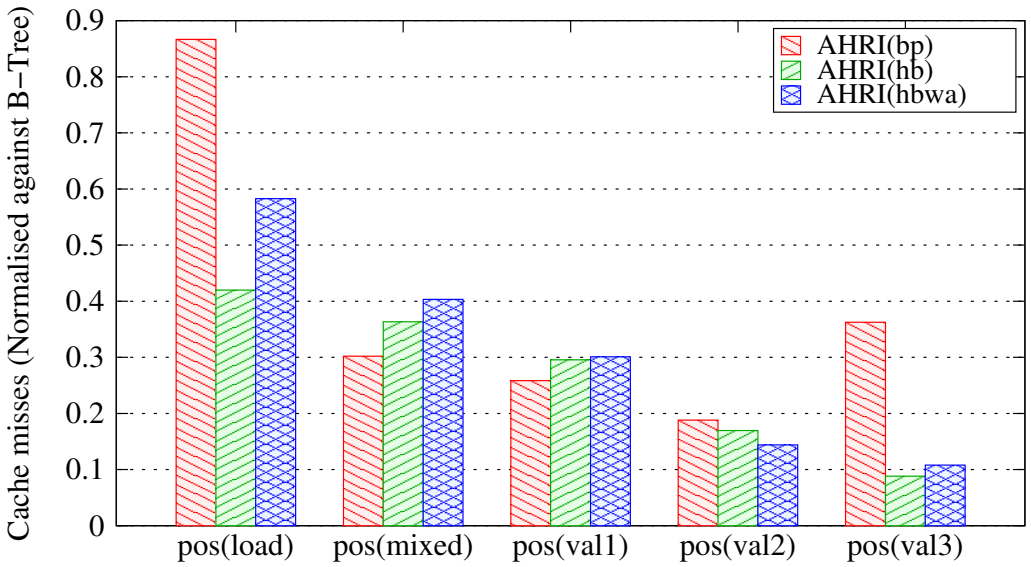


FIGURE 7.14: Cache misses for L3 index variants for a POS index over 5 million triples of BSBM data (lower is better)

The cache miss rates for AHRI L3 index variants are shown in Figure 7.14, and confirm expectations. AHRI-bp has relatively poor cache behaviour for loading and find operations retrievals, due to the cost of unpredictable tree traversals. Otherwise the costs are relatively similar.

7.3.7 TLB misses

Translation Lookaside Buffer misses are related to data cache misses. The TLB is accessed for every time an L1 data cache access is performed, in order to convert virtual memory addresses to physical ones. The TLB can only cache information about a certain number of memory pages, so programs that access data in a very disparate manner, or simply those that access a lot of data, are likely to miss the TLB frequently.

In order to measure TLB miss rates, the following processor events were counted using OProfile:

- DATA_CACHE_ACCESSES (L1 data cache reads, equivalent to L1 TLB reads (Drongowski, 2008))
- L1_DTLB_MISS_AND_L2_DTLB_HIT (L1 TLB misses that hit the L2 TLB)

- L1_DTLB_AND_L2_DTLB_MISS (L2 TLB misses)

Note that in order to calculate the L1 TLB miss rate, it is necessary to add the figures for L1_DTLB_MISS_AND_L2_DTLB_HIT and L1_DTLB_AND_L2_DTLB_MISS (Drongowski, 2008).

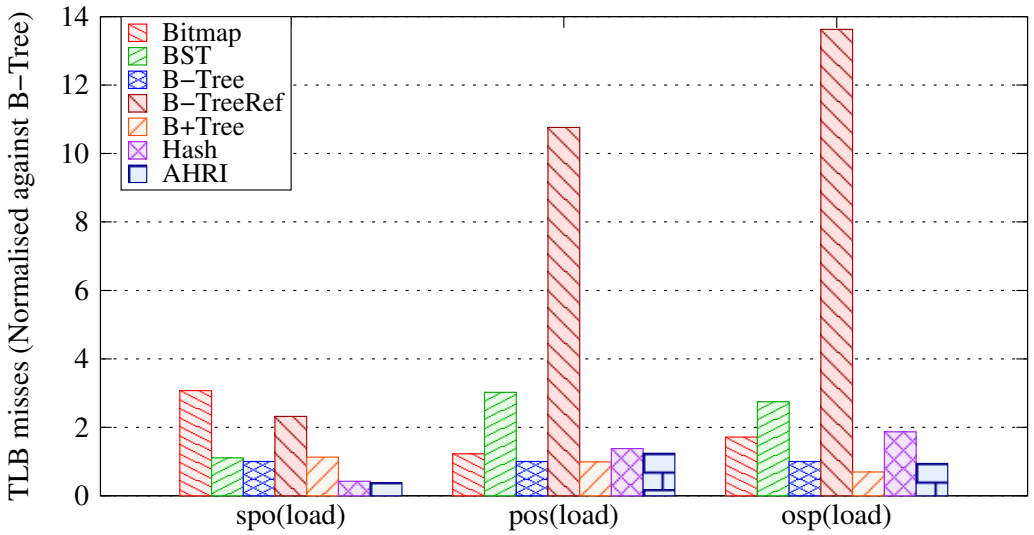


FIGURE 7.15: TLB misses when loading 5 million triples of BSBM data (lower is better)

Figure 7.15 shows the miss rates when loading for each data structure. The poor performance of B-TreeRef is again notable here. This is again due to the cost of shared triple objects, which have to be accessed over a variety of pages.

The apparently varying results for Bitmap are a result of normalisation. For Bitmap, data is always inserted in the order it arrives, whether it is sorted or not. By contrast, structures like trees inherently sort data. Since the data arrives in SPO order, trees insert faster on the SPO index, and so have better results than for POS or OSP. Bitmap's apparently poor performance on SPO-ordered data is actually a result of B-Tree being better, rather than Bitmap getting worse.

TLB miss results for querying over BSBM data are described in Figure 7.16. It's notable that AHRI causes particularly few TLB misses, because relative to the others it touches very little data: for example, if a query iterates over an L3 index, it only has to read one attribute, rather than all three as required by tree indexes.

7.3.8 Branch Mispredictions

Large quantities of branch mispredictions can have a significant impact on index performance. This section considers branch misprediction rate for each of the candidate data structures. Measuring this requires monitoring only two performance counters using OProfile:

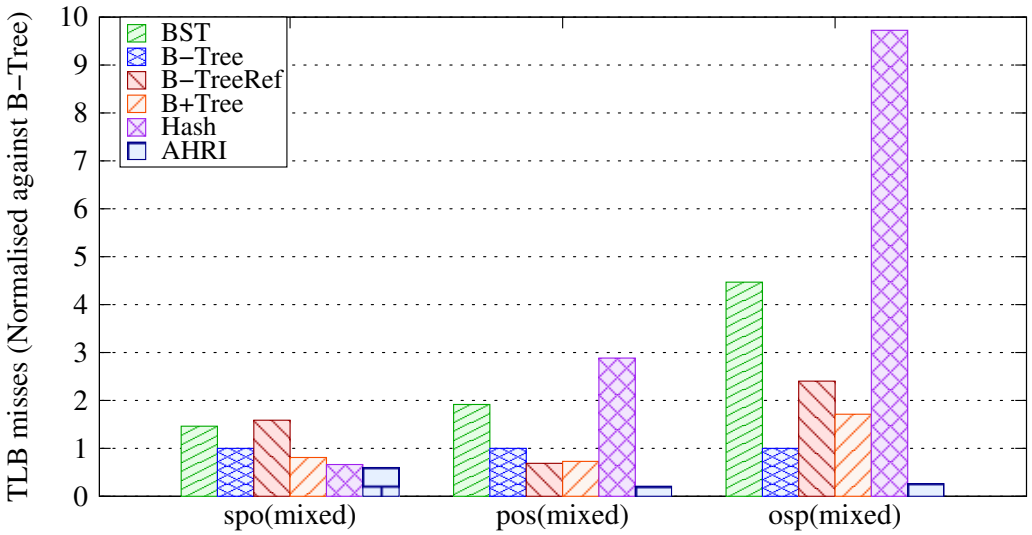


FIGURE 7.16: TLB misses when querying over 5 million triples of BSBM data (lower is better)

- RETIRED_BRANCH_INSTRUCTIONS (Total branch instructions)
- RETIRED_MISPREDICTED_BRANCH_INSTRUCTIONS (Mispredicted branch instructions)

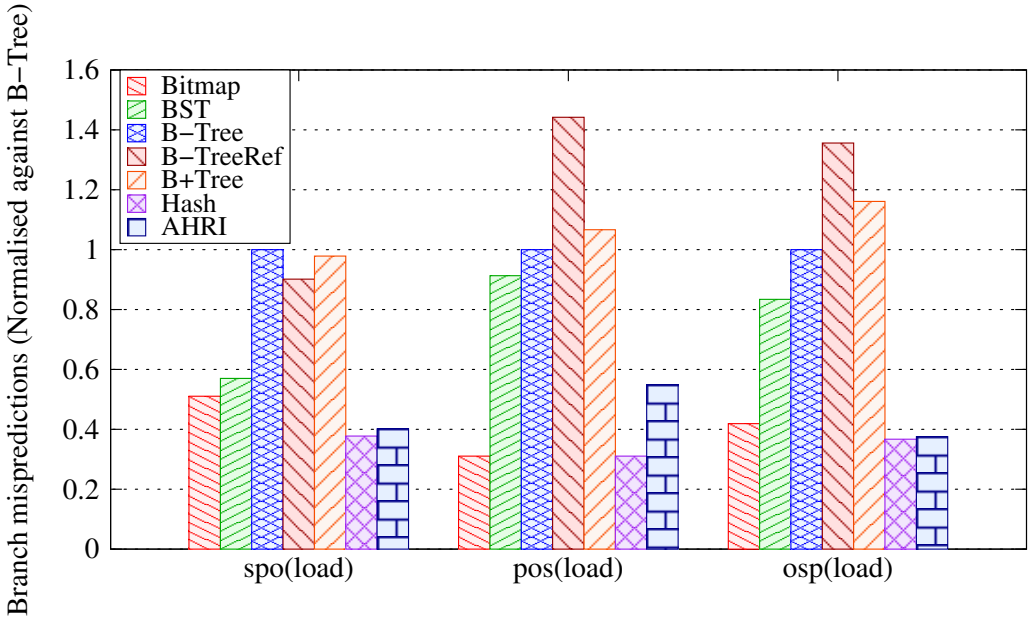


FIGURE 7.17: Branch mispredictions when loading 5 million triples of BSBM data (lower is better)

Figure 7.17 describes the quantity of branch mispredictions when loading BSBM data into each of the candidate data structures. As expected, AHRI does well by this metric, except for POS-ordered data, where it has to load information into B+trees. Note that the reason it does not mispredict as often as the standard tree structures is that there are

a relatively large quantity of L3 B+Trees, and so the height of each tree is, on average, lower.

Hash does the best by this metric, thanks to a very simple insertion, fast assertion mechanism that requires only inserting into a small array or hash set. By contrast, as expected, the B-trees perform poorly. Both BST and Bitmap are built by a bulk, rather than incremental, method, which is why BST achieves better results than the other tree types.

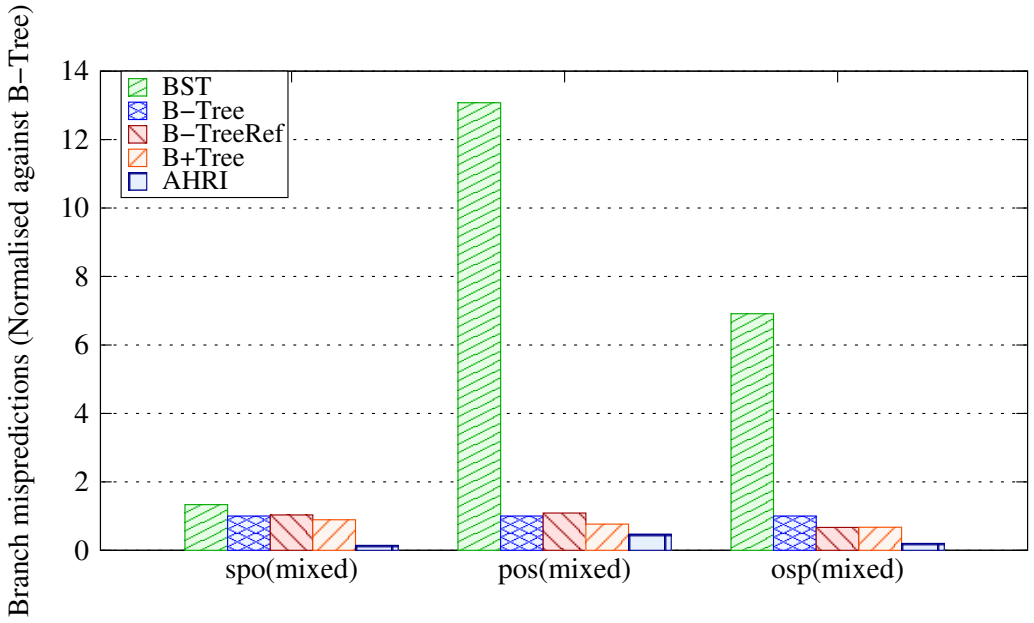


FIGURE 7.18: Branch mispredictions when querying over 5 million triples of BSBM data (lower is better)

Figure 7.18 shows branch mispredictions for each of the data structures. Both Bitmap and Hash had to be eliminated from this chart, due to their distorting the readability of the results. Bitmap causes 90 times as many branch mispredictions as B-Tree for SPO-ordered data, caused by the process of ANDing large compressed bitmaps together. Hash caused over 30 times as many branch mispredictions as B-Tree for POS and OSP ordered data, by virtue of the cost of iterating over hash sets: these data structures inherently require unpredictable comparisons to determine if a slot contains a valid element or not.

AHRI performs exceptionally well by this metric. This is a result of the fact that it requires very few branches in general: when iterating over an L3 index, for example, there is no need to check that elements should be added to the result set: by virtue of the fact that they are in the index, they are part of the result set. This contrasts with tree structures, which do not add data to explicit buckets, and thus have to perform more checks when performing retrievals.

BST's poor results when iterating over data, as shown in the POS and OSP results, reflect the relative complexity of iterating over a binary tree. On arriving at a node, for

example, it's necessary to check whether one or both child nodes has been traversed, and make the decision about where to go next. This contrasts with the relatively simple iteration offered by B-Tree variants.

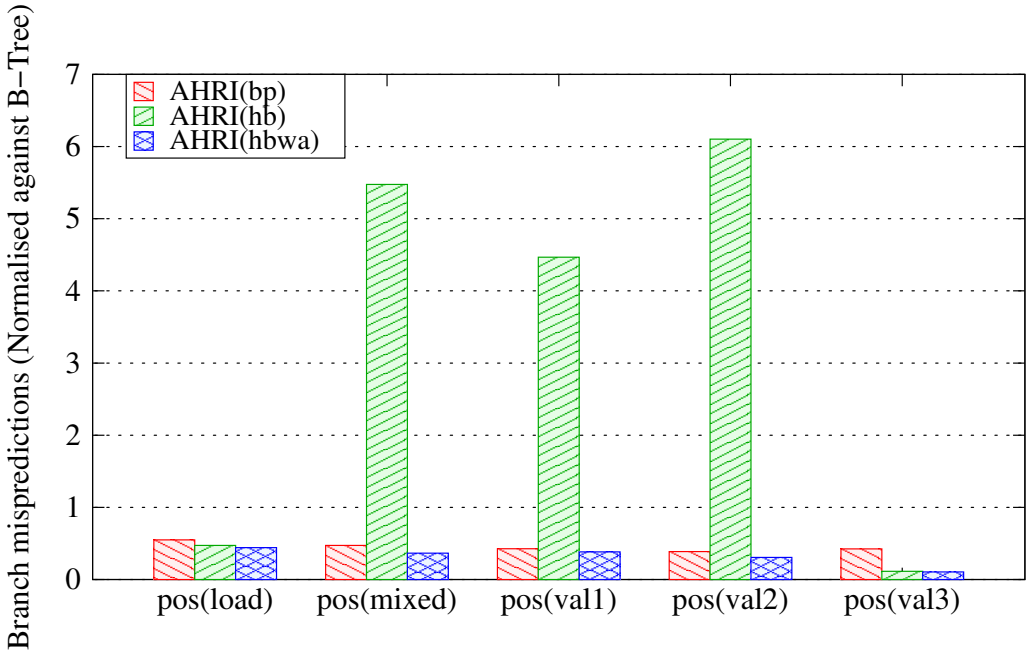


FIGURE 7.19: Branch mispredictions for L3 index variants when loading 5 million triples of BSBM data (lower is better)

The branch misprediction results for the different AHRI L3 indexes (shown in Figure 7.19) are predictable. AHRI-hb performs extremely poorly when forced to iterate over many results. This is because hash sets inherently require a lot of unpredictable comparisons to determine if a slot is filled or not. By contrast, AHRI-bp performs relatively poorly (although less spectacularly so) for find operations.

It should be noted that the cardinality of Predicates, Objects, and PO pairings tends to go up as the size of the dataset increases. This means that the poor results for Hash and AHRI-hb seen in this section are only likely to get worse as the size of the datset increases. However, it is equally quite likely that such high cardinality elements are relatively unlikely to be accessed very often by an intelligent query optimiser, mitigating the impact of this issue.

7.3.9 Discussion

The first round of tests suggest that AHRI substantially outperforms all of the alternative data structures: AHRI is almost universally best on measures of load time, space consumed, and read performance, often by a substantial margin. The analysis of data cache misses, TLB misses, and branch mispredictions described in Sections 7.3.6, 7.3.7,

and 7.3.8 shed light on the reasons behind these results: AHRI performs excellently on each of these metrics.

In terms of load times and performance on some retrieval operations, Hash keeps up with AHRI. Unfortunately, its inability to control space overheads, combined with extremely poor performance on queries that restrict on the second attribute, means that it cannot be considered a suitable competitor on an all-round basis.

In general, the most suitable competitor for AHRI is the B+Tree, which, when compared to B-Tree, trades slightly greater space consumption for faster retrieval times and a simpler codebase. This result was expected, given the B+Tree's recent history of performing well on in-memory systems. In the following sections, B+Tree is used as the only algorithm against which AHRI is compared.

The alternative structures all failed to keep up, particularly in the case of Bitmap. B-Tree and B-TreeRef provided an interesting case study of the differences between structures that refer to shared triple objects and those that encode data inline. This study showed that encoding information inline is a significantly superior in terms of both retrieval performance and space consumed. This conclusion may have to be revisited for tree structures once datasets get so large that they require a 64-bit ID space: B-TreeRef would significantly improve its relative space efficiency in this case.

7.4 Large Scale Tests

To show that AHRI scales effectively, tests were performed on two larger documents: a 350 million triple BSBM dataset (generated using '-fc -pc 1000000'), and the full 230 million triple (after duplicate elimination) DBpedia dataset. As described in Chapter 5, these two represent very different structures of RDF data, and thus provide a good all-round test of AHRI.

In order to work with these datasets, a machine with a substantially larger quantity of RAM was used. This machine was an m2.2xlarge Amazon EC2 instance with the following settings:

- 34.2 GB of memory
- 13 EC2 Compute Units (4 virtual cores with 3.25 EC2 Compute Units each)
- I/O Performance: High
- Linux kernel 2.6.31
- Sun reference JVM (version 1.6.18), run using options '-server -Xms31000M Xmx31000M'

Only the B+Tree and AHRI indexes were tested with this dataset, since they were the most performant options from the initial tests. Unfortunately, the EC2 instance does not provide access to the hardware counters used by OProfile, so it was not possible to obtain cache miss and branch misprediction data on this machine.

7.4.1 Size

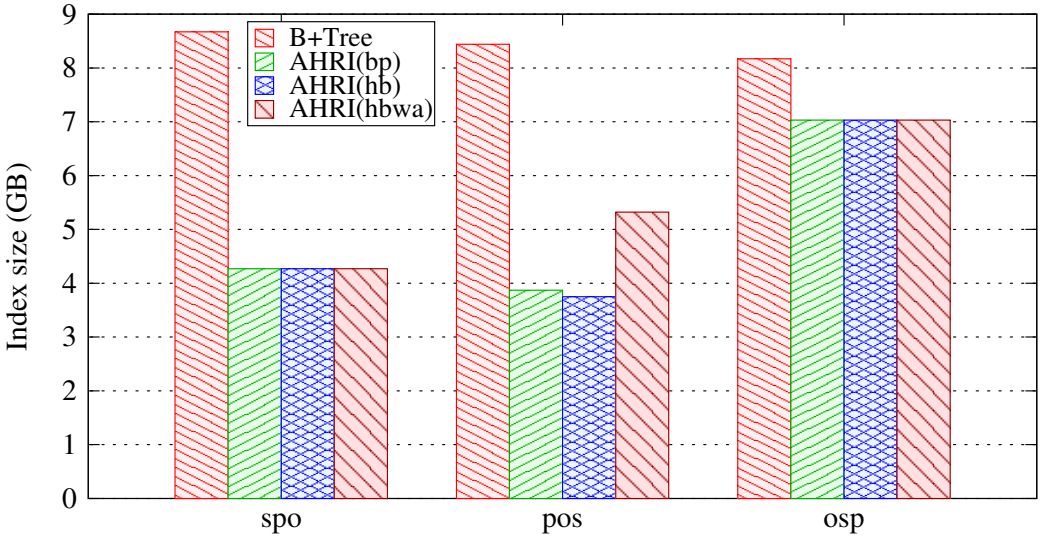


FIGURE 7.20: Index sizes for 350 million triples of BSBM data (lower is better)

Figure 7.20 shows the sizes of each of the AHRI variants, compared against B+Tree, over a 350 million triple BSBM dataset. The relative sizes of AHRI and B+Tree stay similar when compared to the 5 million triple BSBM dataset. This is expected: BSBM is built on repeating patterns, so there is no fundamental change in the characteristics of the dataset when it gets scaled up. While the size of the B+Tree does increase superlinearly with the size of the dataset, this is at a rate of log_n, where n is the average node size. This is too small a factor to have a substantial influence.

As before, AHRI-hb is the smallest of the indexes, followed by closely by AHRI-bp, and trailed substantially by AHRI-hbwa. These results are mirrored when considering the full DBpedia set, shown in Figure 7.21. The results for this dataset are somewhat closer, with the B+Tree achieving a particularly good packing factor on its SPO index. Due to the fact that DBPedia subjects are of relatively high cardinality, AHRI gets relatively little advantage from the small size of its FixedBuckets, increasing overheads somewhat. In general, however, AHRI puts in a strong performance, and remains significantly smaller than the B+Tree.

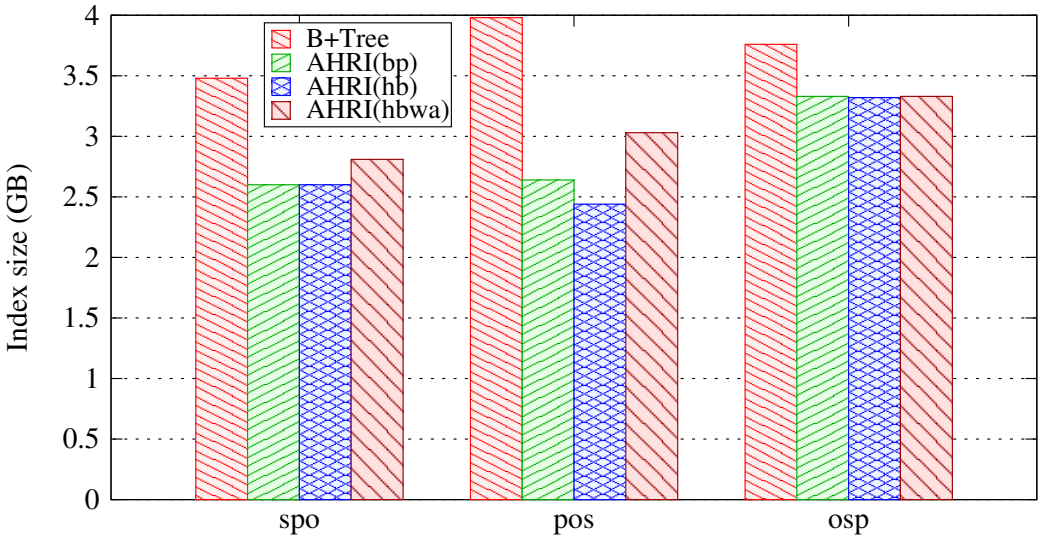


FIGURE 7.21: Index sizes for the full 230 million triple DBPedia dataset (lower is better)

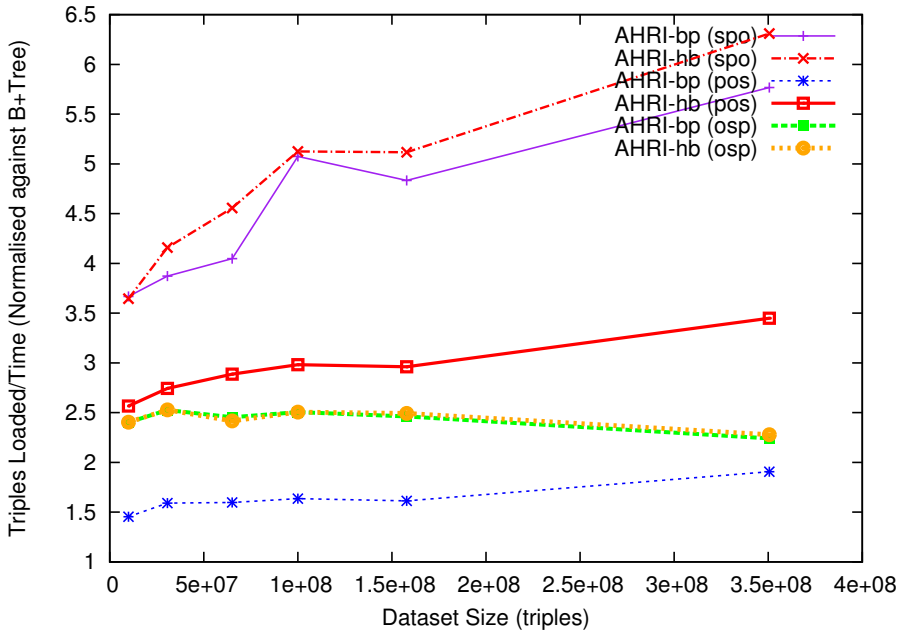


FIGURE 7.22: Load rate with increasing BSBM dataset size (higher is better)

7.4.2 Load

Figure 7.22 shows AHRI's load rate for BSBM data, relative to B+Tree, as the size of the dataset increases. Note, again, that both of these indexes are built using incremental insertion methods, and so these results can be considered indicative of the cost of updating as well as the cost of bulk load.

The load rate of the SPO-ordered index is particularly noticeable here: AHRI's performance relative to B+Tree increases dramatically as the dataset size increases. In reality, this is an artifact of B+Tree's insert rate dropping, while AHRI's stays approximately

constant. AHRI's best insertion rate ocurred on the 30 million triple dataset, averaging 11.5 million triples per second. This high performance is a result of AHRI's fast FixedBucket structures: the only operations that have to be performed are a direct-mapped lookup in an array, followed by a copy and insertion into a very small array of data. This strategy is particularly effective thanks to the fact that data is inserted in SPO order: almost all lookups in the L1 index will be cached, as will the FixedBuckets that get used. B+Tree suffers by comparison due to its relatively wide nodes and long lookup times.

The performance of AHRI's POS indexes is still substantially better than that of B+Tree, although they do not reach the same heights. The POS-ordered structure is heavily influenced by the choice of L3 index: AHRI-bp inserts at a rate of 1.5-2 times that of B+Tree, compared to 2.5-3.5 times for AHRI-hb. The rate for AHRI-bp could be substantially improved by reducing the width of AHRI's L3 B+Tree index nodes, which are currently very wide, at 1000 elements. This would trade off a small amount of space and read performance.

Finally, the insert rate of AHRI's OSP ordered structures is between 2 and 2.5 times that of B+Tree. The rates are extremely similar for the different L3 indexes, as BSBM makes no use of them on the OSP ordering.

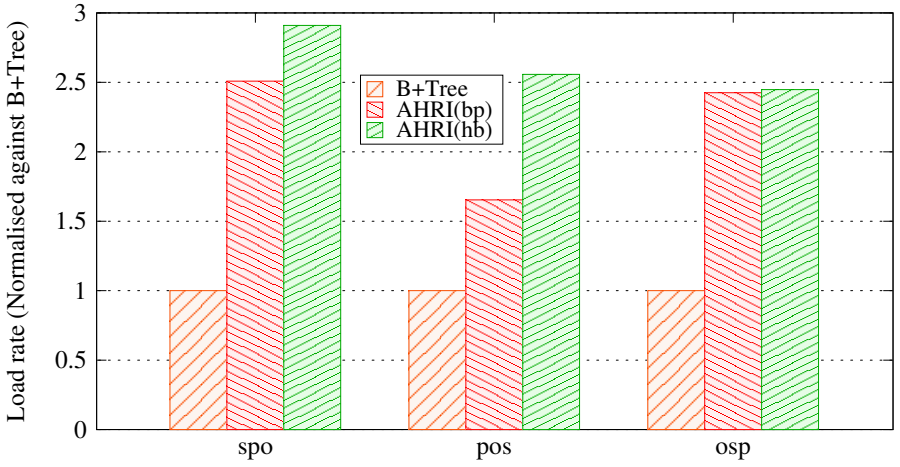


FIGURE 7.23: Load rate for the full 230 million triple DBpedia dataset (higher is better)

Figure 7.23 shows the insertion rate for DBpedia, a dataset that makes much less use of AHRI's FixedBuckets, due to the relatively high cardinality of its Subjects. It can be seen in these results that while performance on SPO-ordered data is still substantially higher than B+Tree, it does not reach the same heights. The insertion rate for this index is just 5 million triples per second. By contrast, the relative load rates for the POS and OSP-ordered indexes are similar to that seen in the BSBM load phase.

7.4.3 Query

This section describes the query (read) performance for each index type, over the two large datasets. Testing was performed as in Section 7.3.3, with the exception that data was retrieved using an iterated method. Instead of passing in an array and writing all results into that array, the index structures returned an object that knew how to iterate over the data structures. This makes it possible to perform element-at-a-time retrieval, necessary to support pipelined operations. This mode of operation is more representative of the conditions found in modern database systems. Tables including the raw figures for each of these tests can be found in Appendix D.

Initial tests at this scale revealed flaws in the design of AHRI. In an iterated environment, where elements are returned from the index one at a time by calling a *next()* method, AHRI slows down significantly. This is due to the cost of calling the next() method. Simple codebases that do not feature much inheritance, like a B+Tree, can have many of their method calls *inlined* - that is, instead of calling the method, the compiler simply replaces the method call with a copy of the method's code, eliminating the call cost. For methods that do not execute extremely regularly, this is a minor operation, but it is of great importance for methods that execute extremely frequently - like next(). Unfortunately, AHRI is more complex than a B+Tree, and the code executed in the next() call is subject to inheritance, which makes it impossible to inline. Since next() gets called for every single element returned, the quantity of method calls slows down execution considerably.

In order to overcome this issue, a version of AHRI that retrieves vectors of results at a time was implemented: effectively buffering calls to next(), and substantially reducing the number of method calls performed. Section 6.3.2.1 discusses the changes made to FixedBuckets to accommodate this change effectively. The alternative structure has negligible differences in terms of load times, and uses the same amount of space, so results for these factors are not discussed here.

7.4.3.1 BSBM

BSBM represents a relatively curated class of information, akin, for example, to the UniProt dataset. Subjects are of predictably very low cardinality, while the dataset contains only a very few predicates, each of which is of very high cardinality.

Figure 7.24 shows the query response rates for the SPO ordering of the BSBM dataset. As expected, AHRI is substantially faster than B+Tree for this ordering. With the higher overheads of the iterated environment mitigating the speed advantage of AHRI's lookups, AHRI manages only an average of 2.5 times improvement upon B+Tree. This is still, however, a very substantial improvement. Since Subject-related queries are always

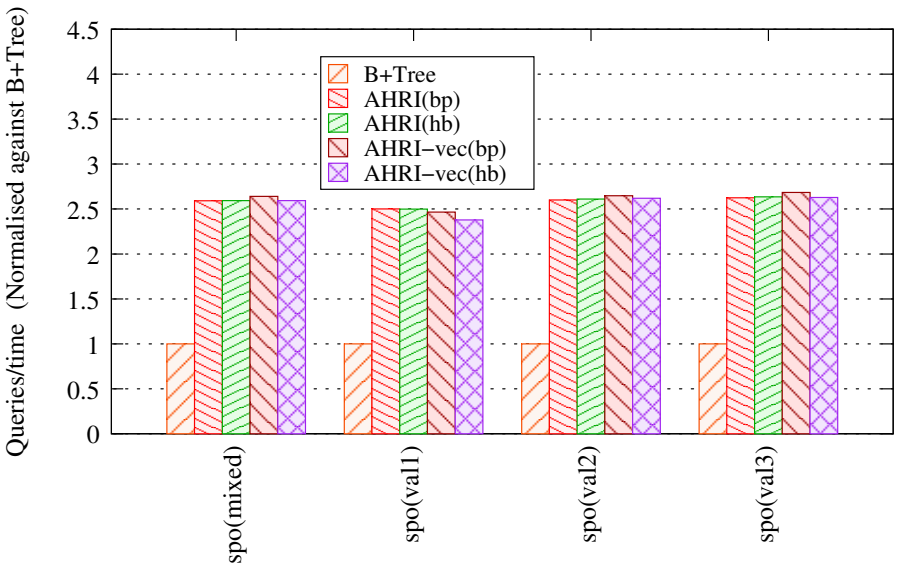


FIGURE 7.24: Query performance over the 350 million triple BSBM dataset using SPO ordering (higher is better)

of low cardinality, and do not generally require an L3 index, there is little to choose between each of the AHRI variants.

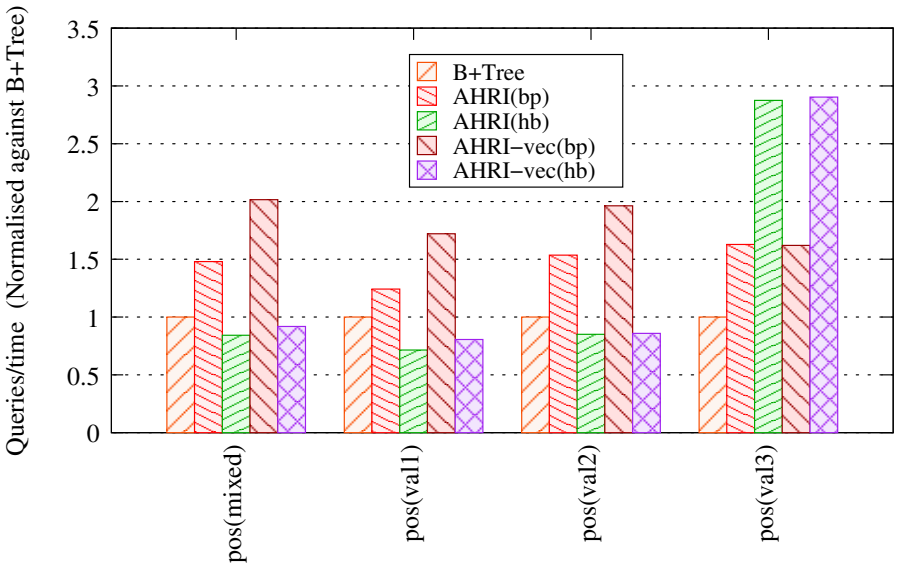


FIGURE 7.25: Query performance over the 350 million triple BSBM dataset using POS ordering (higher is better)

Results for the POS ordering (shown in Figure 7.25) show the real differences between each of the AHRI variants. As expected, the AHRI variants using a hash set L3 index perform poorly for retrievals limited by one or two attributes: they are slow to iterate over. They do, however, perform exceptionally well for find operations. AHRI variants using the B+tree L3 index perform well for retrievals limited by one or two attributes, thanks to very high iteration performance. The difference in iteration performance between the standard and vector models is clearly visible, with the vector model perfoming

substantially better.

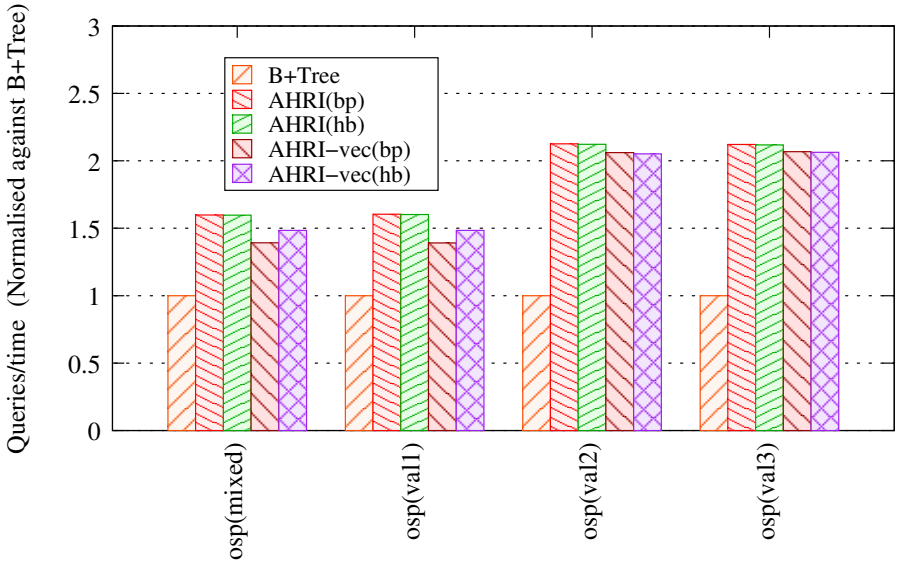


FIGURE 7.26: Query performance over the 350 million triple BSBM dataset using OSP ordering (higher is better)

Figure 7.26 depicts results for the OSP ordering, with AHRI again achieving substantial improvements over B+Tree. Since the OSP ordering makes no use of L3 indexes, there is predictably little difference between the AHRI-hb and AHRI-bp variants. Unexpectedly, the vector model actually causes a slight slowdown when compared to the standard iterated model. Since the expected incidence of queries over the OSP index is low, this is not a substantial concern.

7.4.3.2 DBpedia

The DBpedia dataset represents a significant change from the BSBM one. It is more organic, less managed, and demonstrates substantially different characteristics with respect to the SPO index: a given subject might have a cardinality of several hundred elements, while it exhibits a much greater variety of predicates, a few of which are of low cardinality.

Figure 7.27 shows the results for the SPO ordering. AHRI, while still exhibiting a significant improvement over the B+Tree, does not show the same improvement that it did for BSBM. This is a predictable consequence of the fact that Subjects have substantially higher cardinality in the DBpedia dataset: AHRI's major advantage is in the find time, and iterating over more elements proportionally reduces the influence of this factor. A further consequence of this increased emphasis on iteration performance is the improved results for the vector implementations.

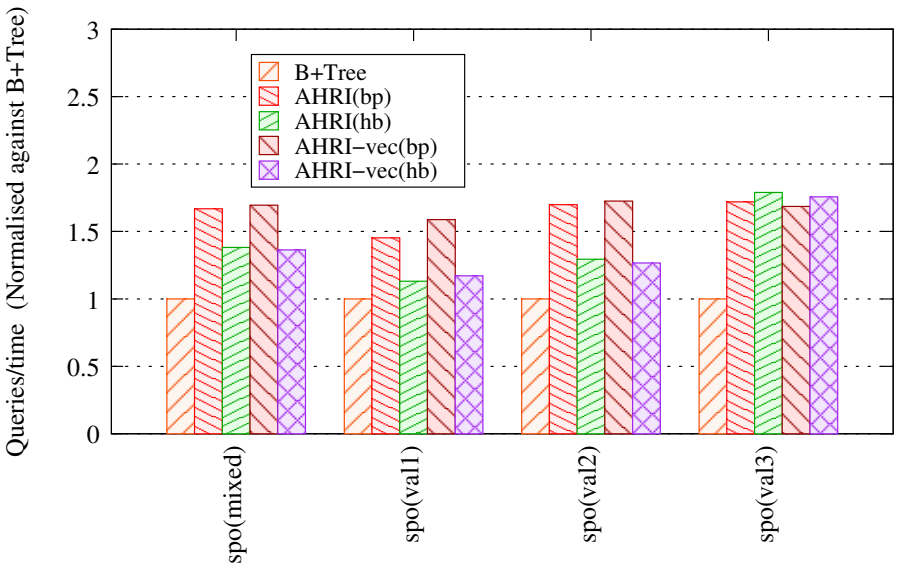


FIGURE 7.27: Query performance over the full DBPedia dataset using SPO ordering (higher is better)

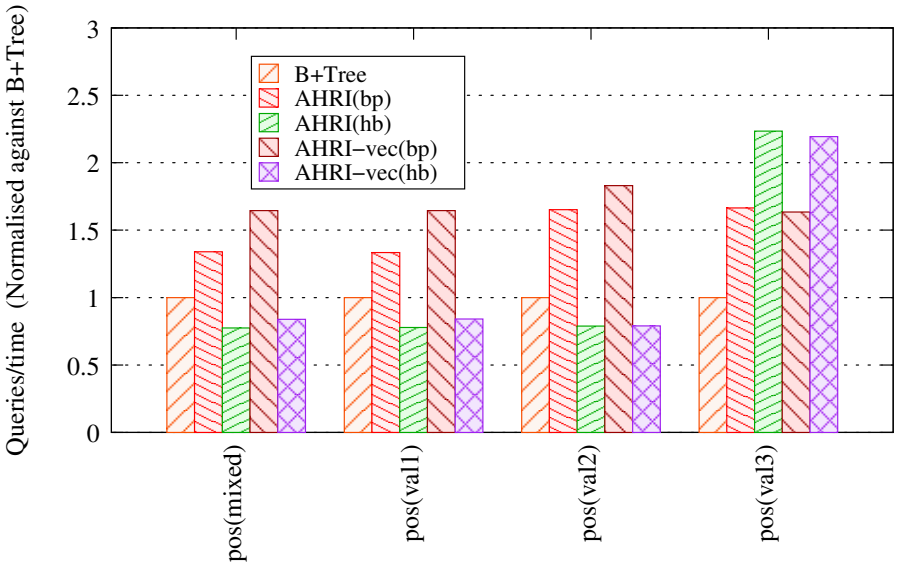


FIGURE 7.28: Query performance over the full DBPedia dataset using POS ordering (higher is better)

Results for the POS index, show in Figure 7.28, are very similar to those for the BSBM dataset. Again, iteration-focused improvements like using the B+tree L3 index and using a vector layout provide substantial benefits.

Finally, the OSP ordering, depicted in Figure 7.29, also provides expected results. AHRI is substantially faster overall, with the vector implementations especially leaping ahead. The limited use of L3 indexes over the OSP ordering means that there is little to choose between the different L3 index implementations.

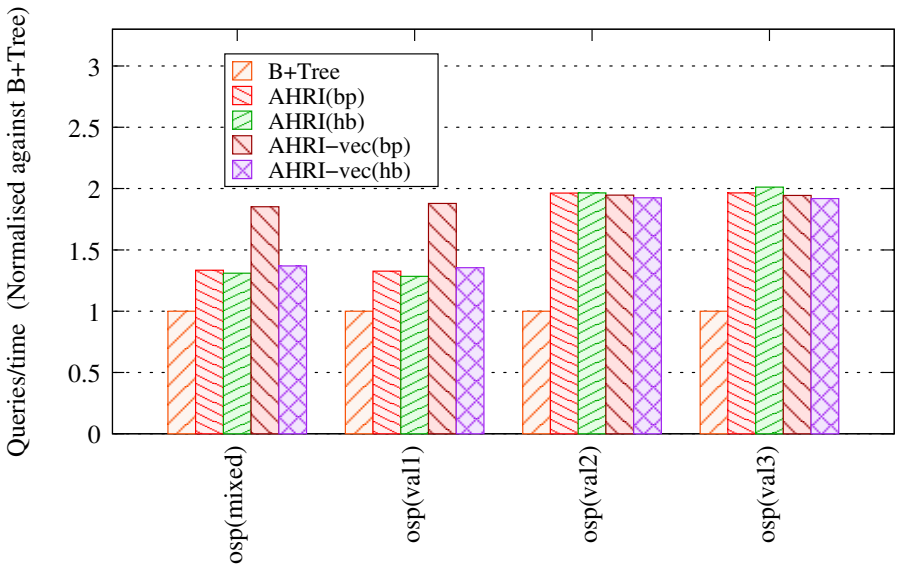


FIGURE 7.29: Query performance over the full DBPedia dataset using OSP ordering (higher is better)

7.4.4 Discussion

Overall, AHRI maintains a substantial advantage over B+Tree for these large datasets. It offers a load rate between 2 and 6 times that of B+Tree, with a substantial reduction in size, and much better query throughput: for most operations AHRI can perform between 1.5 and 3 times as many queries per second as B+Tree.

The different L3 index types, B+tree and hash set, have significantly different behaviour. While they require approximately the same amount of space, AHRI-bp performs significantly better with respect to iteration over large quantities of data, while AHRI-hb is much better for load performance and find operations. The best choice of structure depends largely on workload.

Finally, the size of AHRI's OSP-ordered index is again a noticeable issue. While it is somewhat smaller than the equivalent B+Tree structure, the saving is not especially large. Considering that the OSP index is generally used the least out of all the orderings, there is a clear need to work on a version of AHRI that provides better behaviour with respect to size, even at the cost of query performance.

7.5 Jena Plugin

Testing AHRI's performance as part of a real RDF store is an important part of demonstrating AHRI's real-world utility: while the results show that AHRI's performance is substantially higher than alternative structures in virtually all cases, it must also be

demonstrated that this improvement has a significant impact in the context of all the other operations that a store has to perform.

The AHRI Jena Plugin was created to test AHRI's performance inside a real query engine, and to verify the belief that AHRI would have a substantial impact upon the overall performance of an RDF store. This section considers the size and load/query performance of AJP using the same hardware as that described in Section 7.4.

As well as considering the AHRI and B+Tree index types, performance is compared against two alternative systems that use the Jena framework: The Jena Tuple Database (TDB), a high performance disk-backed store, and the Jena Memory Model (JMM), an example of a current in-memory system. Since these systems all use the same toolkit, and the same indexed nested loops join strategy, as many variables as possible have been eliminated. To ensure, as far as possible, similar query answering strategies, TDB's statistics-based query optimiser was disabled. JMM was also modified to use the same parsing engine as TDB and AJP, an upgrade over its standard parser. The only change to the configuration described in Section 7.4 was for TDB, where the Java options used were '-server -Xmx5000M'. This lower memory limit is because the memory mapping that TDB performs does not count as part of the standard Java heap space. It is thus beneficial to restrict TDB to a smaller amount of memory, in order to ensure that the heap space does not grow to contend with the space available for memory mapping.

Two datasets are used in this section: a 65M triple BSBM file, and a 43.4M triple DBpedia subset, constructed from DBpedia version 3.5, using the files specified by the DBpedia benchmark (Becker, 2008): infoboxes.nt, geocoordinates.nt, and homepages.nt.

7.5.1 Size

Dataset	AHRI-bp	AHRI-hb	B+Tree	TDB	JMM
BSBM (65M)	11.55	11.52	13.38	12.62	31.02*
DBPedia	4.89	4.9	6.0	5.9	13.62

TABLE 7.1: Space consumed (GB) by different RDF stores, loading BSBM and DBpedia datasets. Note that JMM proved unable to load the full BSBM dataset, running out of memory during garbage collection. As a result, figures are linearly projected from a smaller, 30.5 million triple BSBM document.

Table 7.1 breaks down the space consumption of each of the different stores over the two datasets. Results for the vector implementations of AHRI are not included here, since they use the same amount of space as non-vector versions. Amongst these results, JMM's consumption is the noticeable outlier. The reasons for this are twofold:

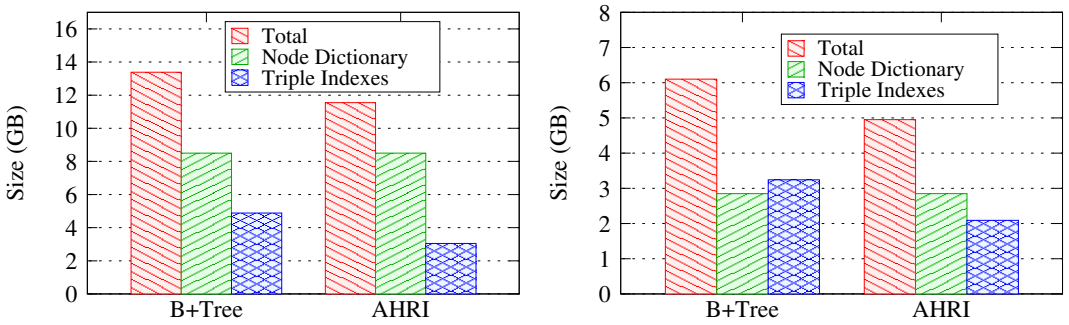
- JMM does not perform full normalisation of string data (as described in Section 4.4), meaning that string data is often stored more than once.

- JMM's triple index structure, similar to the Hash structure used in this chapter, generates a lot of small objects, wasting a lot of space.

TDB's space consumption was determined by examining the size of the data files that it generates, and adding the amount of heap space it consumes. Since TDB uses memory mapping to read and write data, this provides a reasonably accurate figure. TDB's relatively low memory consumption is very noteable, as it comes in at even smaller than AJP's B+Tree implementation. This is an artifact of a kind of implicit compression performed by TDB: when writing to disk, strings are converted into UTF-8 form. Since both of these datasets largely only require one byte per character, UTF-8 encoding saves a substantial amount of space when compared to Java's two byte strings. The downside of this is that encoding and decoding the strings takes a certain amount of time.

Overall, the AHRI implementations are more compact than the alternatives, but there is an unexpectedly small difference in size between B+Tree and AHRI for the BSBM data. This is due to the size of the string dictionary, which consumes 8.5GB in the current uncompressed implementation. This is atypical: BSBM generates an unusually large quantity of text data at 2.5 times as much per triple as the full DBPedia dataset.

Figure 7.30(a) breaks down the amount of space used for string data and triple indexes when loading the BSBM data into AJP, showing that string data consumes the majority of memory space when loading the BSBM dataset, particularly when using AHRI. This is a clear indicator that the next focus for saving space should be on highly performant string compression, or other means of reducing the burden on memory space.



(a) BSBM (b) DPbedia

FIGURE 7.30: Total space consumed by AJP (AHRI-bp) loading a 65M triple BSBM dataset (a) and 43.4M triple DBpedia dataset (b) (lower is better)

Figure 7.30(b) shows results for the 46M triple DBpedia dataset. Despite being over 2/3rds the size of the BSBM file in terms of triple count, it consumes less than half as much space, by virtue of its smaller string dictionary. The advantage of using AHRI is quite significant with this dataset.

7.5.2 Load

Dataset	AHRI (bp)	AHRI (hb)	AHRI-vec(bp)	AHRI-vec(bp)	B+Tree	TDB	JMM
BSBM (65M)	539	513	546	518	603	3768	1221*
DBPedia	268	251	273	274	327	2888	316

TABLE 7.2: Load times in seconds for different RDF stores. Note that JMM proved unable to load the full BSBM dataset, running out of memory during garbage collection. As a result, figures are linearly projected from a smaller, 30.5 million triple BSBM document.

Table 7.2 shows the time required to load data into each RDF store for each dataset. Note that TDB was timed using the included bulk loader. When compared to the results found in earlier tests, the difference in performance between AJP using B+Tree and AHRI is relatively limited. This is an indication that reading and parsing data from disk consumes the majority of AJP's time. TDB is very substantially slower in this test, which is likely to be a consequence of having to occasionally flush data to disk.

AHRI's advantage should become more apparent in systems that are not disk bound, or have greater disk bandwidth. These results indicate that work on faster parsers would be fruitful. One approach might be to work on a parser that uses multiple CPUs.

7.5.3 Query

In order to demonstrate AHRI's query performance, three benchmarks were performed. The first (described in Section 7.5.3.1) was the standard BSBM version 2.0 benchmark, which represents a workload for an e-commerce site: mostly OLTP in nature, with relatively few long running, analytical queries.

In order to demonstrate AHRI's performance in a more analytical situation, four relatively complex custom queries were also created over the BSBM dataset. The BSBM test driver was modified to use these queries in a second round of tests, described in Section 7.5.3.2. For both BSBM tests, the benchmark software was configured to run 100 warmup and 200 normal iterations, ensuring good accuracy of results.

Finally, Section 7.5.3.3 explores AHRI's performance using the DBpedia benchmark. This benchmark is largely of a more analytical nature.

For each test running TDB in this section, the data was reloaded into the store, and the Java object representing TDB kept alive. This is important for ensuring that the store has the best opportunity to cache all its data in RAM: if TDB were restarted after loading data, the only information reliably held in RAM would be that used during the warmup runs. By contrast, loading reliably touches all the data in the system, giving the best odds that all information is cached.

7.5.3.1 BSBM

The standard BSBM suite connects to repositories over HTTP. For the purposes of this experiment, a modified version[3] was used, allowing a direct, local connection. This eliminates the overhead of making the connection, which dominates the cost of some small queries.

BSBM measures performance in terms of sets of queries called 'Query Mixes', reporting an overall metric of 'Query Mixes Per Hour'. This figure is reported here, along with a detailed breakdown of per-query results. Queries 3, 8, and 10 caused excessively long runtimes with AJP due to its inability to work efficiently with OPTIONAL and FILTER statements, and so were not tested. As noted previously, JMM proved unable to load the full 65M triple dataset, and so results are instead reported for the reduced 30.5M triple set.

Store	**QMPH**
AHRI(bp)	1849.3
AHRI(hb)	1856.3
AHRI-vec(bp)	2556.3
AHRI-vec(hb)	2558.0
B+Tree	965.1
TDB	202.2
JMM*	131.34

TABLE 7.3: Query Mixes Per Hour for BSBM. Note that figures for JMM are for a smaller, 30.5 million triple dataset.

Table 7.3 reports the overall QMPH figure for the BSBM run. This shows that AHRI performs extremely well overall, beating B+Tree by a factor of 2-3, and the alternative stores by a factor of ten or more. Since this figure is quite a blunt instrument, per-query results are broken down in Table 7.4.

These results indicate that while AHRI is faster than the alternatives in most situations, the difference is more pronounced on longer running queries. This is not a big surprise: shorter queries are more likely to be dominated by the cost of setup, while longer running ones will work harder on the triple indexes. This effect is mirrored in the comparison between the standard and vector versions of AHRI: it's noticeable that the vector processor provides the largest improvement in the long running query 5.

TDB's poor results for queries 1 and 4 are likely to be a result of taking a different query answering strategy to AJP and JMM: differences as large as a factor of a thousand are unlikely to be down to the performance of the index. Overall, however, it is outperformed by all the AJP implementations, including B+Tree. This is expected: while TDB was

[3] http://github.com/afs/BSBM-Local

Query	AHRI (bp)	AHRI (hb)	AHRI-vec(bp)	AHRI-vec(hb)	B+Tree	TDB	JMM*
1	1136.4	1149.4	1227.0	1149.0	746.3	1.5	270.3
2	1016.1	1014.3	1000.0	988.47	944.4	292.7	526.8
3	-	-	-	-	-	-	-
4	749.1	684.9	813.01	803.2	438.6	0.75	175.3
5	0.55	0.56	0.78	0.79	0.28	0.07	0.04
6	8.3	8.0	8.4	8.4	8.4	0.52	8.73
7	1030.9	1008.8	1034.9	959.23	918.5	351.8	638.5
8	-	-	-	-	-	-	-
9	1822.3	1946.5	1965.6	1980.2	1713.1	918.5	1777.8
10	-	-	-	-	-	-	-
11	3333.3	3225.8	3333.3	2777.8	4255.3	1408.5	2985.1
12	1470.6	1418.4	1418	1526.7	1290.3	682.6	1360.5

TABLE 7.4: BSBM query results. Results indicate the number of queries performed per second for each query type. Note that figures for JMM are for a smaller, 30.5 million triple dataset.

given enough memory to cache all its data, working with memory mapped files is not as fast as working in normal memory.

Finally, JMM produces a creditable performance, particularly for extremely short lived queries. Its generally good per-query results are overwhelmed in in the overall metric by extremely poor performance in query 5.

7.5.3.2 Complex BSBM Queries

Since BSBM mostly consists of an OLTP-like workload, which does not heavily stress the underlying indexes, four queries with a more challenging, analytical nature were generated. These queries are shown in Figure 7.31.

These queries provide relatively little opportunity for restricting the working set, thanks to their low proportion of repeating variables and fixed bindings. As a result, they tend to stress the triple indexes more, as more pieces of data need to be retrieved for joining. This is shown in the results, with queries generally requiring substantially longer runtimes. Table 7.5 shows the QMPH figures for each store. As before, AHRI substantially outperforms the alternative methods, particularly when using the vector version. Unlike the standard BSBM query set, these queries give the B+tree L3 index a significant performance advantage over the hash set. This is because harder queries such as these are more likely to have to iterate over large sets of L3 data.

Table 7.6 shows the per query results for the complex query set. The disk-based store's performance over this dataset is particularly noticeable, and it was unable to complete Query 3 in less than 30 minutes. It is not clear whether this is down to poor query

Store	QMPH
AHRI(bp)	10933.4
AHRI(hb)	10221.9
AHRI-vec(bp)	13768.8
AHRI-vec(hb)	12126.1
B+Tree	5549.6
TDB*	54.27
JMM*	1328.1

TABLE 7.5: Query Mixes Per Hour for the complex query benchmark. Note that figures for JMM are for a smaller, 30.5 million triple dataset, while figures for TDB exclude Query 3, which it was unable to complete.

Query	AHRI (bp)	AHRI (hb)	AHRI-vec(bp)	AHRI-vec(hb)	B+Tree	TDB	JMM*
1	40.62	43.34	47.56	46.53	25.37	0.12	10.22
2	421.94	421.05	441.5	422.83	346.02	64.45	109.83
3	148.92	167.93	187.44	154.8	87.91	-	27.82
4	5.8	4.87	7.83	6.28	2.51	0.06	0.48

TABLE 7.6: Complex query results. Results indicate the number of queries performed per second for each query type. Note that figures for JMM are for a smaller, 30.5 million triple dataset.

optimisation or incomplete caching of its indexes, but the memory-based stores have a clear advantage for these queries.

The performance gap between JMM and AHRI is much larger for these tests, as the latency is less influenced by query startup time. This effect also allows the vector-based versions of AHRI to display their full power, providing significant performance gains on the harder queries.

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>

    SELECT ?vendor ?product WHERE {
    ?offer bsbm:product ?product .
    ?product bsbm:productFeature %ProductFeature1% .
    ?offer bsbm:vendor ?vendor .
}
```

(a) Query 1: Return all vendors who sell product(s) with a given feature, along with those products.

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rev: <http://purl.org/stuff/rev#>

SELECT DISTINCT ?productset WHERE {
    ?prod1 rdf:type %ProductType% .
    ?review bsbm:reviewFor ?prod1 .
    ?review rev:reviewer ?person .
    ?allreviews rev:reviewer ?person .
    ?allreviews bsbm:reviewFor ?productset .
}     LIMIT 1000
```

(b) Query 2: For a given product, return products that have also been reviewed by people who bought it.

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?label ?vendor ?producer ?country WHERE {
    ?product bsbm:producer ?producer .
    ?producer bsbm:country ?country .
    ?offer bsbm:vendor ?vendor .
    ?offer bsbm:product ?product .
    ?vendor bsbm:country ?country .
    ?product rdfs:label ?label .
    ?product rdf:type %ProductType% .
} ORDER BY ?label
```

(c) Query 3: For products of a given type, find the vendors who sell in the country of manufacture.

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name ?review WHERE {
    ?review rev:reviewer ?person .
    ?person foaf:name ?name .
    ?review bsbm:reviewFor ?product .
    ?review bsbm:rating1 2 .
    ?product bsbm:productFeature %ProductFeature1%
}
```

(d) Query 4: For products with a given feature, find the reviews that gave the product a rating of 2.

FIGURE 7.31: Analytical queries for the BSBM dataset

7.5.3.3 DBpedia

The benchmark described in Becker (2008) was used to test query performance over DBpedia data. The results in Section 7.4 indicate that AHRI's advantage when working with DBpedia information is smaller than for BSBM, because DBpedia makes much less use of AHRI's fast FixedBuckets, and so these tests consider AHRI's real-world performance in a more challenging environment. In order to ensure that each store was warmed up effectively, a set of three variants of each query was performed in advance. The test was run three times, with the mean result taken.

Query	AHRI (bp)	AHRI (hb)	AHRI-vec(bp)	AHRI-vec(hb)	B+Tree	TDB	JMM
1	3	4	4	4	4	4	6
2	84	83	87	82	105	225	164
3	69	70	60	64	152	490	17
4	1169	1214	1086	1076	1178	4255	1770
5	1121	1252	1055	1050	1124	4342	1907

TABLE 7.7: Detailed DBpedia query results. Figures indicate the time in milliseconds required for each query.

The per-query results are shown in Table 7.7. On average, AHRI-vec(bp) provided a 38% reduction in the time taken to perform each query when compared to B+Tree, a substantial improvement. AHRI is also generally substantially faster than both JMM and TDB except in one case: JMM performed exceptionally well on query 3.

7.6 Discussion

Overall, the results in this chapter have shown that AHRI provides a substantial improvement upon other index types for storing RDF data in memory. It manages to effectively maintain a compact layout, while offering excellent load/update/query performance. This is accomplished through a few key features, which together fulfil the requirements listed in Section 6.1:

- Inlining of the IDs that represent triples directly into indexes, rather than storing separate, shared triple objects. This is shown to save space, and dramatically reduce cache misses.
- Elimination of repeating IDs, but trying to do so only where the saved space would be higher than the incurred overhead. In general, this approach saves a large quantity of storage.

- Customisation of the structure to fit the shape of the data being stored. This customisation is enabled through the examination of RDF datasets discussed in Chapter 5.
- Avoiding, where possible, structures that require a lot of computation and large numbers of branch operations to locate data. AHRI's L1 and L2 indexes locate data quickly and efficiently, requiring little computation and causing very few branch mispredictions when compared to tree-like structures.
- Limiting the creation of small objects, which tend to have high space overheads in many modern languages.

Section 7.5 showed that these advantages translate effectively into real stores, particularly for long running queries. With further development of AJP to support more standard features such as string compression, and fast multi-threaded parsing, the advantages of AHRI will become even more apparent, as the overheads that mitigate its impact are reduced.

While AHRI shows excellent overall behaviour, the results presented in this chapter have shown that there is room for it to improve further. In particular, it is not especially compact on OSP-ordered data. While there are theoretical limits to the amount of compression that can be performed on OSP-ordered data, AHRI can certainly be improved. The fact that this ordering of data is used relatively infrequently means that there is particular flexibility for improvement: it can be heavily optimised for space consumption rather than time. One could consider, for example, applying a more expensive to read system such as delta-compressed B+Trees.

The hypothesis described in Section 1.3 stated that RDF-specific in-memory structures could substantially outperform more general structures, and that existing in-memory RDF structures could be improved upon substantially. This evaluation has effectively proved this hypothesis, showing that in-memory data structures can benefit substantially from the major points of interest raised in this document: the importance of designing with respect to the underlying structure of RDF datasets, and with the characteristics of modern CPUs and memory latency firmly in mind. These two factors combine to afford AHRI substantial benefits over alternative structures, and offer the ability to significantly improve the performance of RDF storage in general.

Chapter 8

Conclusions and Future Work

The Semantic Web offers the potential to dramatically improve the manner in which we retrieve and interact with data. RDF stores represent a critical requirement for the emergence of this vision: the importance of high performance storage and query over unpredictable RDF data is clear, and has been articulated during the course of this thesis.

RDF does, however, represent a challenge when it comes to delivering performant storage and query solutions. Common DBMSs derive their performance from an expectation of both predictable data structure and predictable query loads, neither of which can be expected for RDF data. A standard approach to fixing this issue has been extreme read-orientation, rendering it infeasible to update information in the store.

These issues represent a threat to the Semantic Web as a whole: modern web applications, for example, have thrived on rich user interaction, which requires both low latency queries and the ability to update information. Without such capabilities, a wide variety of applications that would be useful to Semantic Web users are rendered impractical.

The work in this thesis represents an alternative to existing approaches: it combines research aimed at better understanding the common characteristics of RDF datasets, with a focus on designing practical, RDF-specific in-memory data structures that make excellent use of the features of modern computer hardware. High performance RDF memory stores have received relatively little attention in existing literature, despite the requirements for low latency RDF storage found in interactive applications and reasoners, and the hardware trends that are making main memory storage increasingly practical. This thesis, then, represents significant new contributions to the body of literature on RDF storage.

8.1 AHRI: A Summary

AHRI is a structure that grew out of the analysis of RDF data provided in Chapter 5. It performs particularly well, making it substantially faster to load, update, and query RDF information than alternative structures, while using substantially less space. It does this by adapting its structure to fit the requirements of the information it is storing, based on an in-depth understanding of how RDF datasets behave, and how to get the best performance out of modern computers.

Thanks to a multi-level index structure, AHRI eliminates excessive repetition of values, and their implied wasted space. Its adaptive structure allows the provision of greater or lesser levels of indexing, depending on how much is useful and how it will impact structure size, all the while retaining excellent characteristics with respect to both data locality and the predictability of branch operations. On the whole, this work has shown that RDF data is an excellent candidate for cache-friendly data structures: the size of a single RDF fact is very small, making it possible to fit several such facts into a cache line. It is expected that the work on cache-friendly RDF data structures described in this thesis will be of interest to the designers of existing RDF stores.

Unlike tree structures, where the number of operations required to access information slowly scales up with the size of the dataset, AHRI scales with the number of attributes that the dataset exhibits. Since RDF has only three attributes (Subject, Predicate, and Object), it is an ideal candidate for this sort of approach. Other data description languages with a small number of attributes would also benefit from this approach.

AHRI's design is especially effective when implemented using newer programming languages like Java: its sparing use of small objects substantially reduces memory overheads when compared to many alternative approaches, as well as dramatically reducing load on garbage collectors. Further, it implicitly provides a substantial quantity of statistical information that can be used to improve the quality of query plans.

Overall, AHRI represents a significant step towards a Semantic Web that allows individuals to ask meaningful questions in a timely manner. It also provides a basis for the realisation of further improvements: while AHRI is already faster and smaller than existing alternatives, it leaves open the possibility of integrating future enhancements, through an architecture that encourages plugging in alternative sub-indexes.

Beyond AHRI itself, this thesis provides a foundation for the improvement of RDF storage in general: it provides detailed information on the structure of RDF documents, using a tool that automatically generates human-readable results, as well as a review on the effect of modern computer architectures on RDF data structures. Improvements to AHRI, and future work building on the contributions of this thesis, are described in Section 8.2.

8.2 Future Work

This thesis serves as the basis for a great deal of future work, related to both the further development of AHRI and AJP, and the future of RDF storage in general. This section breaks this future work into short and long term segments. Short term work relates largely to incremental improvements of AHRI and AJP, along with studies to provide data on the query patterns most commonly used in Semantic Web applications.

By contrast, the longer term work features more fundamental research into parallelisation: how RDF stores can make use of multi and many-core processors, along with coping with distribution across networks in in-memory stores.

8.2.1 Short Term

8.2.1.1 AHRI Improvements

There is a variety of work that could be performed to improve both AHRI and its integration into the Jena Semantic Web framework. Perhaps the most important change is to make AHRI still more adaptive. While AHRI performs very well overall, it has relatively poor characteristics over the OSP index.

Since the OSP index is accessed relatively rarely, the focus should be upon keeping its size small rather than achieving extremely high performance. Instead, over this index ordering, AHRI is fast but requires more space than it does in SPO or POS orderings. This is due to the relative lack of repetition in the OSP index, and AHRI's attempts at eliminating repeating data sometimes costing more than the amount of space that is saved.

A remedy to this issue would be to implement more second level index types. The OSP index would have lower overheads using a B+Tree or radix trie as a second level index. While a tree would not provide statistics, this would not be a significant issue due to the fact that OS pairings are almost universally extremely low cardinality, and rarely used anyway. Compression techniques (particularly RDF-3X style encoding) could also be explored, although the low cardinality of OS pairings would limit the success of such approaches.

A second, larger piece of future work is to implement a version of AHRI that supports fully sorted order: If AHRI were to be implemented into systems using a merge and sort/merge strategy, it would be beneficial to provide sorted order in all cases. AHRI currently offers full sorting for some data, with grouping for the rest. Implementing a performant, sorted version of AHRI is impractical in Java due to the fact that Java

does not support software cache prefetching. A fully sorted version requires lower level control, and an implementation in C or C++ would be used for this purpose.

8.2.1.2 Jena Integration

The current AJP implementation is relatively simple, sufficient for prototype-level testing but with room for significant improvement. The most crucial items on the list are simple implementation issues: improved support for OPTIONAL and FILTER patterns in SPARQL queries, which currently rely on a slow fallback mechanism in the Jena framework. FILTER support can be further improved by adding support for inlined IDs (as described in Section 5.4.2), which would reduce space requirements and dramatically speed up integer comparisons.

Space requirements can be further reduced by the introduction of curbs on the size of the string dictionary. While the BSBM test set used in this document has an unusually large amount of string data, space requirements are substantial even for a more normal dataset. As described in Section 5.4.2, common prefix elimination is a simple solution to the issue of the space consumed by URIs, eliminating a large percentage of their overhead, while performing sufficiently well for fast in-memory stores.

Literals are a more complex issue. The majority of space consumed by literals is used by large strings. These are unlikely to be receptive to any simple compression technique, but general purpose algorithms such as Lempel-Ziv may be effective. Such algorithms do have a substantial cost in terms of performance, but the number of literals that would need to be encoded to save a large amount of space is very small, making this avenue worth investigating. One piece of future work, then, is to examine the effect of compression on both URIs and literals.

Another improvement that can be made to AJP is support for persistence. The current implementation offers no persistence, although the data held in AHRI can be easily extracted and saved as a file. Addition of incremental persistence is simply an implementation issue, but choices will have to be made about the required level of transactional integrity: strong guarantees that data will be persisted once asserted into AHRI will inevitably slow down the assertion process.

Finally, the statistics generated by AHRI go unused by the current implementation. These statistics have the potential to substantially improve the performance of AJP, if used in conjunction with a high quality query optimiser. A good quality optimiser will be able to produce better query plans, and even alter the plan on the fly if results suggest that the current approach is poor.

8.2.1.3 Real World Studies

A useful piece of future work would be integrating AHRI into real-world applications, and observing the improvement that it produces. AHRI is well suited to a variety of scenarios. RDF-based human-interactive applications such as mSpace (Smith et al., 2007) require extremely performant, low latency backing stores to drive their flexible UIs. mSpace requires very low latency over queries of significant complexity, with multiple concurrent users. For datasets of any significant size, mSpace has been forced to move from RDF stores to more restrictive RDBMSs in order to attain acceptable performance. The problem in this case is not one of the ability to assert very large numbers of triples, but of being able to perform nontrivial queries in an extremely short period of time.

Other use cases for AHRI exist. Backward chaining reasoners require extremely high performance backing stores to operate effectively, and heavily read oriented DBMSs like C-Store (Stonebraker et al., 2005) (and its descendant, Vertica) use supplemental very low latency memory-based systems to allow incremental assertions and updates. These systems have to be extremely fast in order to not create a noticeable drag on the overall system performance. Such a subsystem would be of use in existing highly read optimised RDF stores such as YARS2, RDF-3X, and Hexastore.

A further helpful outcome from integrating AHRI into real world systems would be a study into the sorts of queries that commonly get performed on RDF stores in the wild. This would be complementary to the work described on ExamineRDF in this document: ExamineRDF inspects the features of datasets, while this piece of work would inspect the characteristics of queries that are run upon them. With both of these pieces of information, RDF store creators would be better equipped to make evidence-based decisions about how to design their systems.

8.2.2 Long Term

8.2.2.1 Multi-processors

As processor manufacturers come up against hard limits to how far they can increase clock speeds, multi-core and multi-processor systems are becoming the norm. While increased transaction throughput is eminently feasible with the application of greater numbers of cores, particularly in a read-oriented environment, achieving speedup is a much harder proposition.

Application of techniques designed for distributed systems described in Section 3.5 could afford real gains in speedup. The index nested loops join used by AJP offers excellent opportunities for parallelisation without having to share (and thus lock) much data. A

major piece of future work in the space of RDF storage will be allowing multi-core, multi-thread systems to create real speedup opportunities.

8.2.2.2 Distribution

While a multi-core extension of AHRI may make use of traditional techniques from the world of distributed DBMSs, a truly distributed system is substantially more challenging. Traditional distributed DBMSs rely on the assumption that network latency is relatively insignificant: a reasonable assertion when relying on disk-backed storage, but very much untrue in a memory-based environment: a typical 200-500μs round trip latency (Erling and Mikhailov, 2008) is very large in such a system.

Distributing a memory store requires a reduction in network latency, which is achievable using advanced hardware: network switches are the largest cause of latency in modern networks (Ousterhout et al., 2010), and high-performance switches can substantially reduce this latency. Expanding a memory-backed store to multiple systems and their extra storage space is highly desirable, and with the use of such hardware, distributing AHRI becomes a feasible and useful piece of future work. AHRI itself is perfectly suited to acting as part of a distributed store. A similar approach to that used by the author in Clustered TDB (described in Section 3.5.3) would be applicable.

8.3 Contributions of this Research

This thesis has yielded a variety of contributions, produced as steps on the road to creating AHRI, the major aim of this work. The first of these steps was an investigation into the behaviour of modern processors, with particular reference to the effects of memory latency and branch misprediction on high performance software. This was combined with a deep review of the structure of existing RDF datasets, including the creation of a tool to produce detailed statistics over RDF data, and identify common factors in those datasets.

Combined with some research into the behaviour of modern JVMs, relevant because an increasing number of high performance systems run on top of virtual machines, the knowledge gained from these investigations enabled the creation of AHRI and AJP. A detailed evaluation showed that AHRI provides superior performance to competing structures, and offers the potential to substantially improve the performance of in-memory RDF stores.

8.4 Final Remarks

AHRI, the RDF data structure described in this thesis, is designed to improve the performance of RDF data stores. It was created using new insights into the structure of RDF datasets, combined with a deep analysis of how RDF stores can fully leverage the power of modern computers. AHRI is intended to break through the limitations of existing data stores, effectively balancing find and iteration performance against space consumption in an in-memory environment. Detailed evaluation of AHRI shows that it achieves these goals.

The aim of this body of work has been to increase the practicality of using RDF data, particularly in interactive environments with a requirement for performant read/write/update operations. The work in this thesis provides a foundation for further improvements in RDF storage and query, with a rich set of future work described in this chapter. It is hoped that ongoing work on providing the ability to update information in RDF stores, while retaining high performance read operations, will provide the inspiration for a wide variety of new applications to drive adoption of the Semantic Web.

Appendix A

Binary Chop Tests

This appendix contains the code for both the Java and C implementations for the tests described in Section 4.1.

A.1 Java Implementation

Run using: java -server -Xmx2048M <filename>

```
public class TestClass {
        final static int size = 150000000;
        final static int iterations = 50000000;

        final static int step = 10;
        int[] cmparr;
        int[] arr1;
        int[] arr2;

        /* arr1 is an array of sorted, randomly increasing integers
         arr2 is filled with 1s.  cmparr contains a pregenerated array
         of random numbers to search for in these arrays.

         Note that this implementation of binary chop does not complete early,
         so it will not finish after one comparison on the predictable data.
         */
        public TestClass() {
                init();

                long time;

                time = System.currentTimeMillis();
                dosearch(arr2,cmparr,iterations);
                System.out.println(System.currentTimeMillis() - time);

                time = System.currentTimeMillis();
                dosearch(arr1,cmparr,iterations);
                System.out.println(System.currentTimeMillis() - time);

        }
```

```
        public void init() {
                for(long l = 0; l < iterations;l++) {
                }

                int incr = 0;

                arr1 = new int[size];
                for(int i = 0; i < size; i++) {
                        incr+=(int)(Math.random()*step);
                        arr1[i] = incr;
                }

                arr2 = new int[size];
                for(int i = 0; i < size; i++) {
                        arr2[i] = 1;
                }

                cmparr = new int[iterations];
                for(int i = 0; i < iterations; i++) {
                        cmparr[i] = (int)(Math.random()*arr1[arr1.length-1]);
                }
        }

        public int dosearch(int[] arr, int[] cmparr, int iterations) {
                for(int i = 0; i < iterations;i++) {
                        int max = size-1;
                        int min = 0;
                        while(min < max) {
                                int pos = min+((max-min)/2);
                                if(arr[pos] > cmparr[i]) {
                                        max = pos;
                                } else {
                                        min = pos+1;
                                }
                        }
                }
                return 0;
        }

        public static void main (String[] args) {
                new TestClass();
        }
}
```

Listing A.1: Java implementation of simple binary search

A.2 C Implementation

Compiled using: gcc -Wall -Werror -O3 -std=c99 <filename>

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#define SIZE 150000000
#define ITERATIONS 50000000
```

```
#define STEP 10
#define IN_OTHER 0

void * domalloc(int size)
{
        void * mem = malloc(size);
        if (mem == NULL)
        {
                exit(0);
        }
        return mem;
}

/*performs a binary chop on arr for each value in cmparr*/
int runarr(unsigned int* restrict arr, unsigned int* restrict cmparr) {
        int i;
        clock_t starttime;

        starttime = clock();

        for(i = 0; i < ITERATIONS;i++) {
                unsigned int min = 0;
                unsigned int max = SIZE-1;

                while(min < max) {
                        int pos = min+((max-min)/2);
        //              printf("Pos: %d Min: %d Max: %d\n",pos, min,max);
                        if(arr[pos] > cmparr[i]) {
                                max = pos;
                        } else {
                                min = pos+1;
                        }
                }
        }
        printf("ms: %d\n",(int)((clock() - starttime)/(int)(CLOCKS_PER_SEC/1000)));
        return 0;


}

/* numbers is an array of sorted, randomly increasing integers
numbers2 is filled with 1s.  cmparr contains a pregenerated array
of random numbers to search for in these arrays.

Note that this implementation of binary chop does not complete early,
so it will not finish after one comparison on the predictable data.
*/
#if IN_OTHER==0
int main(int argc,char *argv[])
{
        unsigned int i;
        unsigned int * restrict numbers;
        unsigned int * restrict numbers2;
        unsigned int * restrict cmparr;
        unsigned int currnum = 0;

        numbers = domalloc(SIZE * sizeof(int));
        numbers2 = domalloc(SIZE * sizeof(int));
        cmparr = domalloc(ITERATIONS * sizeof(int));
```

```
        for(i = 0; i < SIZE; i++) {
                currnum += (rand()/((double)RAND_MAX+1))*STEP;
                numbers[i] = currnum;
                numbers2[i] = 1;

        }
        for(i = 0; i < ITERATIONS; i++) {

                cmparr[i] = 1+(rand()/(((double)RAND_MAX+1))*numbers[SIZE-1]);
        }

        runarr(numbers2,cmparr);
        runarr(numbers,cmparr);

        free(numbers);
        free(numbers2);
        free(cmparr);



        return 0;
}
#endif
```

Listing A.2: C implementation of simple binary search

Appendix B

Test Machine

A variety of small experiments were conducted through the course of this thesis. Except where otherwise mentioned, these were performed on a machine with the following specifications:

- Dual 1.8GHz AMD Opteron, 1MB L2 cache per core
- 8GB RAM
- Linux kernel 2.6.32 (64 bit)
- 64 bit Sun reference JVM (version 1.6.13)
- OProfile 0.9.4

Appendix C

RDF Dataset Statistics

C.1 DBpedia

C.1.1 Summary

Triple Count: 231661194
URI Count: 30218224
Average URI length: 52.93, Standard Deviation: 20.45
Average URI reuse: 20.97
Appeared as (ignoring literals):
S only: 1735317
P only: 1101
S and P: 38559
O only: 11794631
O and S: 16648616
P and O: 0
S, P and O: 0
O including literals: 48039661
Literal Count: 36245030
Average literal length: 76.72, Standard Deviation: 282.03
Average literal reuse: 1.69
Blank Node Count: 0
Average Blank Node reuse: 0.00

C.1.2 Node appearances as S, P, O, SP, PO, OS

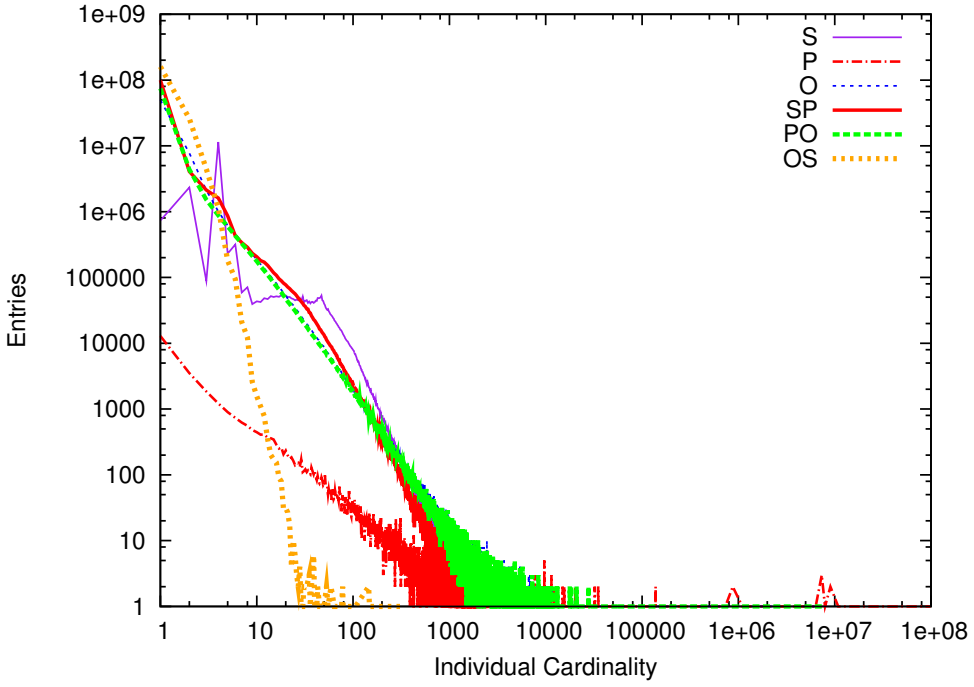


FIGURE C.1: Node and pairing data for DBpedia

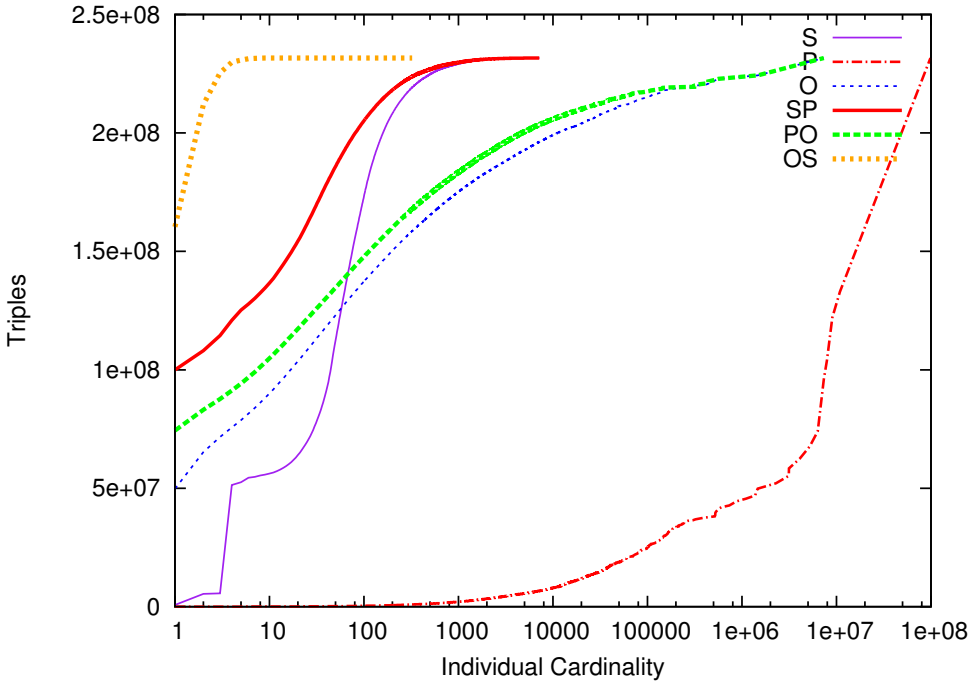


FIGURE C.2: Cumulative node and pairing data for DBpedia

Cardinality	**S**	**P**	**O**	**SP**	**PO**	**OS**
Total	18422492	39660	64688277	112753228	84943262	192062986
1-1	734198	12899	49741498	99958311	74386735	160322033
2-2	2326425	3515	7773391	4071109	4390289	25863273
3-3	87910	1852	2119364	2125134	1513971	4433129
4-4	11433732	1220	1011063	1613126	881030	1130500
5-5	229739	901	615796	862354	597213	179579
6-6	316280	735	462601	428145	430183	94019
7-7	59241	626	341414	339319	328748	21026
8-8	70952	555	274464	287173	261233	12164
9-9	39432	480	225395	233385	212285	2584
10-19	490919	3340	1073029	1334970	989297	4512
20-29	477096	1695	372922	584914	333453	105
30-39	433028	1124	184720	312555	164837	17
40-49	468415	862	109811	175099	98482	7
50-59	326108	655	72882	108358	65772	6
60-69	223578	483	51244	72750	46811	3
70-79	159799	451	37803	49486	34721	5
80-89	115875	350	28151	35472	26518	3
90-99	89577	351	22036	27210	20579	0
100-199	272325	1831	95504	97731	90062	13
200-299	40821	949	29111	20774	27622	7
300-399	12997	593	13227	7053	12645	1
400-499	5664	392	7787	3153	7322	0
500-599	3022	303	5033	1933	4602	0
600-699	1598	306	3483	1068	3184	0
700-799	930	245	2427	630	2256	0
800-899	688	156	1916	441	1837	0
900-999	469	136	1499	325	1418	0
1000-1099	4	3	8	1	10	0
1000-1999	1463	909	6001	1081	5795	0
2000-2999	147	383	1727	113	1768	0
3000-3999	38	211	875	34	812	0
4000-4999	10	126	486	9	460	0
5000-5999	10	89	320	10	298	0
6000-6999	1	81	228	1	185	0
7000-7999	1	49	154	1	119	0

continued on next page

Cardinality	**S**	**P**	**O**	**SP**	**PO**	**OS**
8000-8999	0	64	137	0	113	0
9000-9999	0	47	81	0	69	0
10000-19999	0	273	353	0	303	0
20000-29999	0	111	140	0	94	0
30000-39999	0	80	61	0	45	0
40000-49999	0	35	41	0	20	0
50000-59999	0	29	13	0	14	0
60000-69999	0	17	15	0	18	0
70000-79999	0	17	11	0	5	0
80000-89999	0	17	8	0	1	0
90000-99999	0	7	10	0	3	0
100000-199999	0	61	24	0	14	0
200000-299999	0	13	2	0	0	0
300000-399999	0	3	3	0	4	0
400000-499999	0	1	3	0	3	0
500000-599999	0	7	3	0	2	0
600000-699999	0	1	0	0	0	0
700000-799999	0	1	0	0	0	0
800000-899999	0	2	0	0	0	0
1000000-1999999	0	4	1	0	1	0
2000000-2999999	0	1	0	0	0	0
3000000-3999999	0	2	0	0	0	0
4000000-4999999	0	1	0	0	0	0
5000000-5999999	0	1	0	0	0	0
6000000-6999999	0	1	0	0	0	0
7000000-7999999	0	4	1	0	1	0
9000000-9999999	0	2	0	0	0	0
10000000-19999999	0	1	0	0	0	0
90000000-99999999	0	1	0	0	0	0

Table C.1: Node appearances as S, P, O, SP, PO, OS for DBpedia

C.1.3 Aggregate Node Reuse

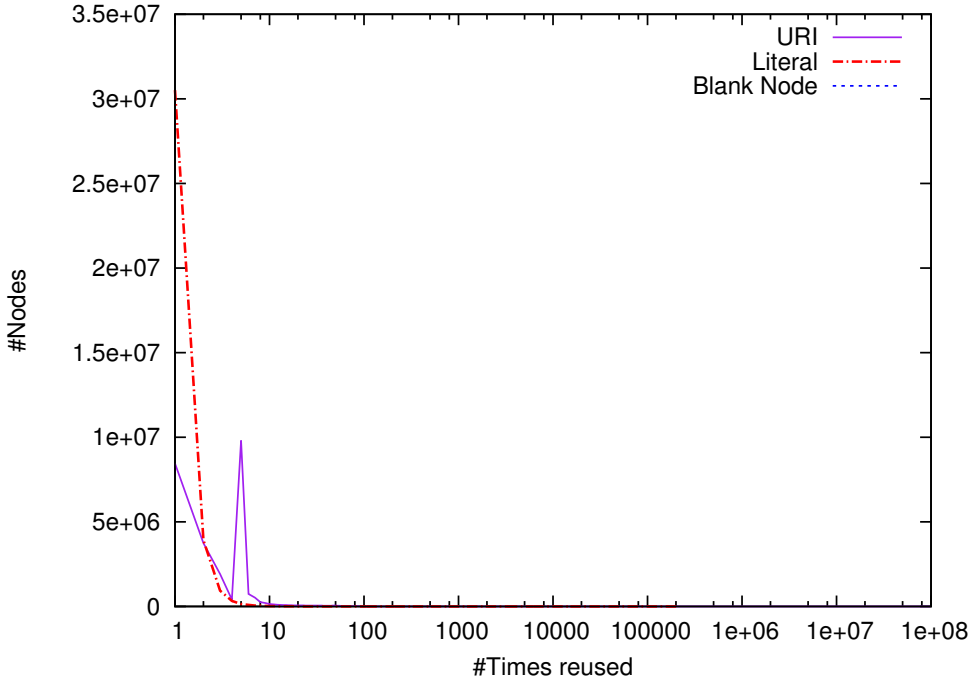


FIGURE C.3: Node reuse data for DBpedia

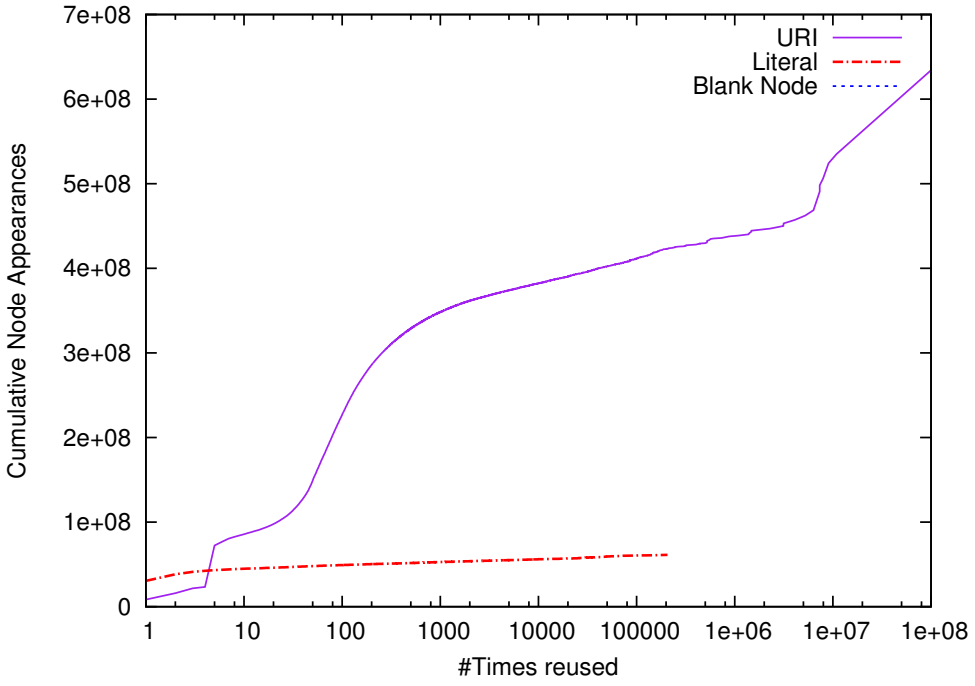


FIGURE C.4: Cumulative node reuse data for DBpedia

#Times reused	URI	Literal	Blank Node
Total	30218224	36245030	0
1-1	8454355	30514732	0
2-2	3738198	3915583	0
3-3	1920471	934156	0
4-4	412706	322932	0
5-5	9810425	145747	0
6-6	739064	95783	0
7-7	518101	50710	0
8-8	253417	36805	0
9-9	203877	25931	0
10-19	887449	104545	0
20-29	553293	34578	0
30-39	473485	18380	0
40-49	451994	10673	0
50-59	399317	7045	0
60-69	276840	4696	0
70-79	207441	3370	0
80-89	157333	2250	0
90-99	120778	1829	0
100-199	446536	7928	0
200-299	92856	2361	0
300-399	36324	1109	0
400-499	18705	768	0
500-599	10831	581	0
600-699	7075	425	0
700-799	4826	283	0
800-899	3497	216	0
900-999	2591	186	0
1000-1099	26	0	0
1000-1999	9910	692	0
2000-2999	2324	244	0
3000-3999	1108	123	0
4000-4999	642	65	0
5000-5999	384	40	0
6000-6999	302	39	0
7000-7999	194	26	0

continued on next page

#Times reused	URI	Literal	Blank Node
8000-8999	192	23	0
9000-9999	113	17	0
10000-19999	582	62	0
20000-29999	213	37	0
30000-39999	124	21	0
40000-49999	56	17	0
50000-59999	39	5	0
60000-69999	27	6	0
70000-79999	24	4	0
80000-89999	25	0	0
90000-99999	15	2	0
100000-199999	81	4	0
200000-299999	14	1	0
300000-399999	6	0	0
400000-499999	4	0	0
500000-599999	10	0	0
600000-699999	1	0	0
700000-799999	1	0	0
800000-899999	2	0	0
1000000-1999999	5	0	0
2000000-2999999	1	0	0
3000000-3999999	2	0	0
4000000-4999999	1	0	0
5000000-5999999	1	0	0
6000000-6999999	1	0	0
7000000-7999999	5	0	0
9000000-9999999	2	0	0
10000000-19999999	1	0	0
90000000-99999999	1	0	0

Table C.2: Node reuse data for DBpedia

C.1.4 Node lengths

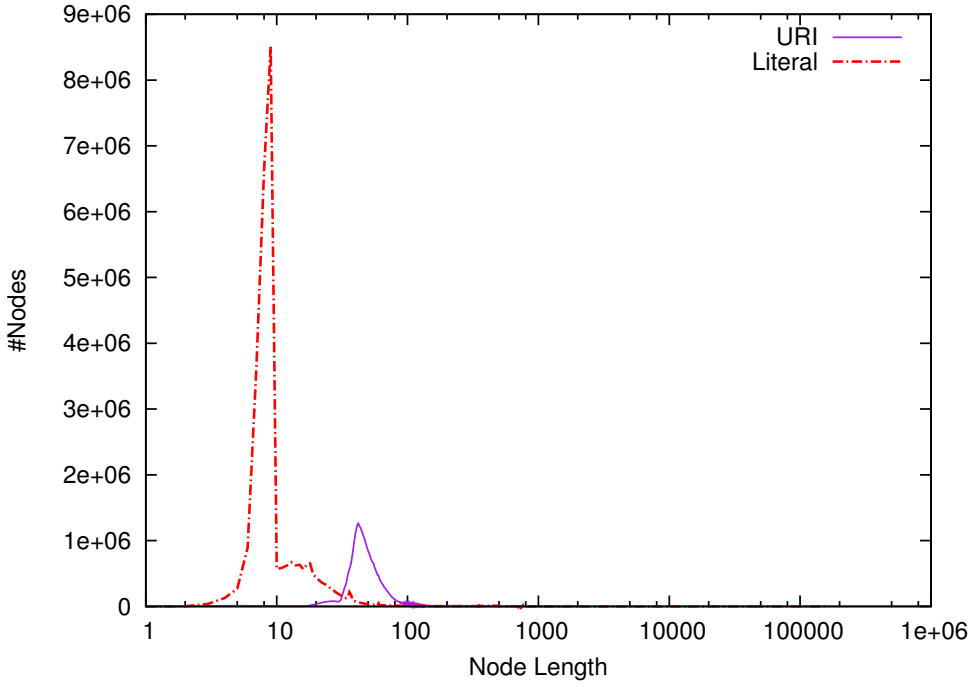


FIGURE C.5: Node length data for DBpedia

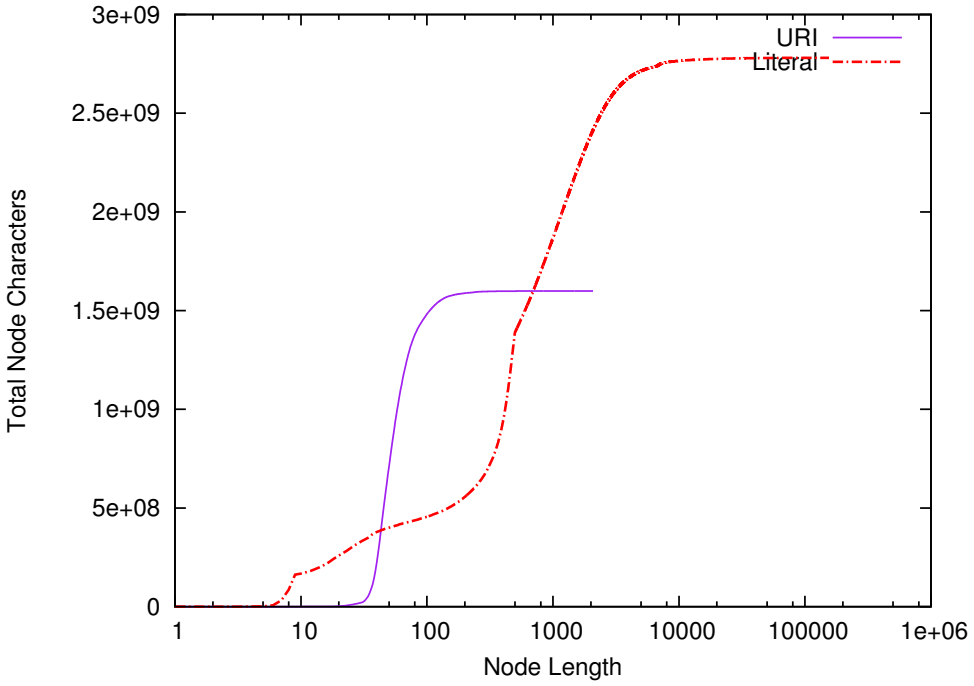


FIGURE C.6: Cumulative node length data for DBpedia

Node Length	**URI**	**Literal**
Total	30218224	36245029
1-1	72	23
2-2	1855	2841
3-3	7132	38834
4-4	9682	125933
5-5	10573	268445
6-6	9322	889580
7-7	9103	3650586
8-8	7736	6631679
9-9	6853	8503520
10-19	70046	6049375
20-29	667438	3127136
30-39	4469975	1493856
40-49	11058969	533083
50-59	6582427	272552
60-69	3431096	203984
70-79	1678006	128919
80-89	775803	115052
90-99	467434	95416
100-199	910083	688230
200-299	38465	541664
300-399	4073	679607
400-499	1011	1019534
500-599	362	201570
600-699	276	157567
700-799	179	128169
800-899	80	103969
900-999	53	85415
1000-1099	1	778
1000-1999	118	379846
2000-2999	1	87226
3000-3999	0	23927
4000-4999	0	7901
5000-5999	0	3145
6000-6999	0	2407
7000-7999	0	1471

continued on next page

Node Length	URI	Literal
8000-8999	0	533
9000-9999	0	326
10000-19999	0	749
20000-29999	0	115
30000-39999	0	34
40000-49999	0	17
50000-59999	0	3
60000-69999	0	6
70000-79999	0	3
90000-99999	0	2
100000-199999	0	1

Table C.3: Node length data for DBpedia

C.2 UniProt

C.2.1 Summary

Triple Count: 2809173894
URI Count: 391273031
Average URI length: 29.08, Standard Deviation: 13.19
Average URI reuse: 18.98
Appeared as (ignoring literals):
S only: 101791597
P only: 104
S and P: 0
O only: 39637426
O and S: 249843881
P and O: 23
S, P and O: 0
O including literals: 93843528
Literal Count: 54206102
Average literal length: 158.32, Standard Deviation: 301.96
Average literal reuse: 18.45
Blank Node Count: 0
Average Blank Node reuse: 0.00

C.2.2 Node appearances as S, P, O, SP, PO, OS

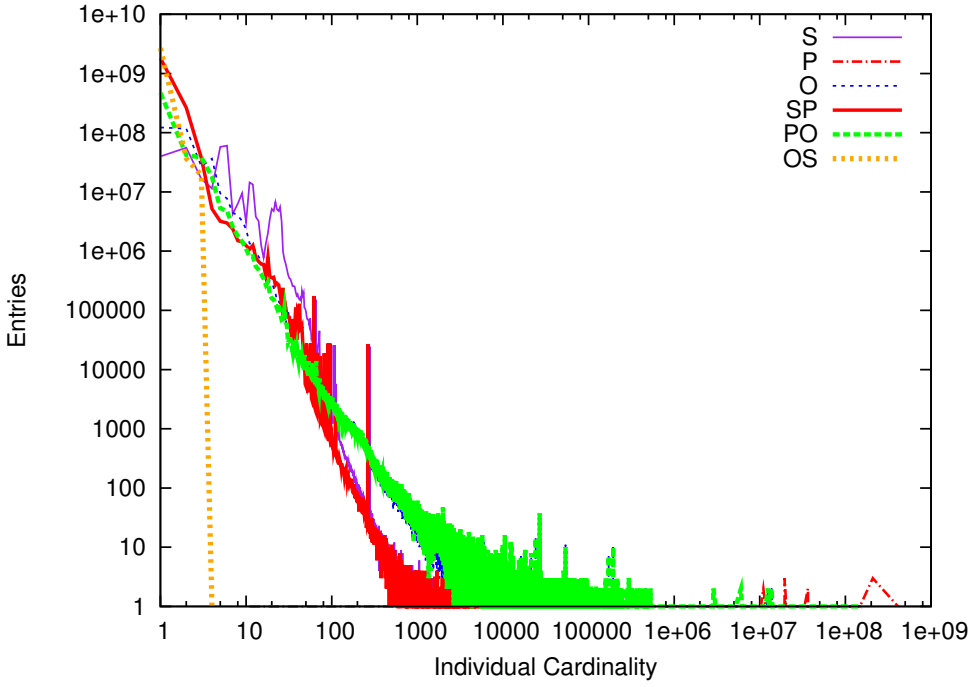


FIGURE C.7: Node and pairing data for UniProt

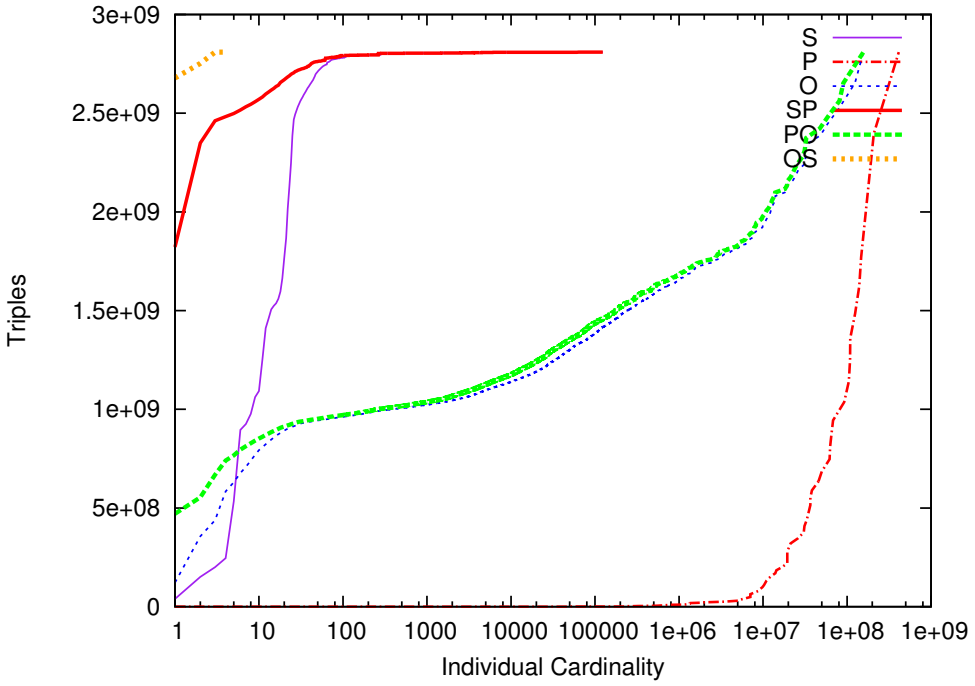


FIGURE C.8: Cumulative node and pairing data for UniProt

Cardinality	**S**	**P**	**O**	**SP**	**PO**	**OS**
Total	351635478	127	343687432	2151619560	591661013	2734351818
1-1	39749143	0	122422328	1822170403	470540556	2678549398
2-2	55590683	0	117036507	263473659	41731214	36782765
3-3	16523708	0	27114367	37550409	38137671	19019654
4-4	11430341	0	36518018	5194729	17976442	1
5-5	57528465	0	9400196	3206989	5296608	0
6-6	60388594	0	7733445	2982967	4789474	0
7-7	4237137	0	4443999	2370572	2562360	0
8-8	6295105	0	4032765	1504296	1797058	0
9-9	9511368	0	3219638	1436581	1462460	0
10-19	46805887	1	9085318	7749391	5252901	0
20-29	37557316	1	1516494	2304913	1082105	0
30-39	3034431	0	432044	622738	286698	0
40-49	1622760	0	157085	537959	146500	0
50-59	627111	0	90547	112841	93396	0
60-69	340188	0	64624	216317	78020	0
70-79	163815	0	51958	29945	45870	0
80-89	46495	0	38348	38188	34461	0
90-99	32414	0	30531	60174	30041	0
100-199	113481	1	143728	21460	141586	0
200-299	32346	1	53746	30725	55520	0
300-399	1511	1	19119	1273	20783	0
400-499	661	2	11004	573	12497	0
500-599	389	1	7632	366	9208	0
600-699	272	1	5914	255	7073	0
700-799	167	0	4419	156	5510	0
800-899	112	1	3538	108	4466	0
900-999	97	0	2815	97	3365	0
1000-1099	1	0	42	1	39	0
1000-1999	522	8	15570	519	20459	0
2000-2999	350	1	7177	350	9697	0
3000-3999	233	2	5196	233	5356	0
4000-4999	205	4	3474	203	3687	0
5000-5999	85	1	2218	85	2617	0
6000-6999	30	0	1377	30	1652	0
7000-7999	19	1	1028	19	1217	0

continued on next page

Cardinality	S	P	O	SP	PO	OS
8000-8999	6	1	841	6	948	0
9000-9999	6	0	708	6	807	0
10000-19999	16	5	3669	16	4725	0
20000-29999	1	2	1904	1	2140	0
30000-39999	2	3	995	2	912	0
40000-49999	2	5	669	2	590	0
50000-59999	0	1	448	0	343	0
60000-69999	2	2	215	2	293	0
70000-79999	0	1	184	0	241	0
80000-89999	0	0	193	0	219	0
90000-99999	0	1	139	0	118	0
100000-199999	1	1	684	1	570	0
200000-299999	0	8	188	0	178	0
300000-399999	0	1	94	0	90	0
400000-499999	0	2	61	0	58	0
500000-599999	0	2	41	0	35	0
600000-699999	0	3	23	0	17	0
700000-799999	0	3	15	0	19	0
800000-899999	0	0	12	0	9	0
900000-999999	0	0	10	0	10	0
1000000-1999999	0	9	62	0	54	0
2000000-2999999	0	0	11	0	12	0
3000000-3999999	0	1	9	0	8	0
4000000-4999999	0	0	5	0	4	0
5000000-5999999	0	2	3	0	3	0
6000000-6999999	0	1	4	0	6	0
7000000-7999999	0	4	5	0	5	0
8000000-8999999	0	1	1	0	3	0
9000000-9999999	0	2	3	0	3	0
10000000-19999999	0	12	15	0	13	0
20000000-29999999	0	3	3	0	5	0
30000000-39999999	0	7	4	0	3	0
40000000-49999999	0	1	1	0	1	0
50000000-59999999	0	1	1	0	1	0
60000000-69999999	0	4	0	0	0	0
80000000-89999999	0	0	1	0	1	0
90000000-99999999	0	1	0	0	1	0

continued on next page

Cardinality	S	P	O	SP	PO	OS
100000000-199999999	0	6	2	0	1	0
200000000-299999999	0	3	0	0	0	0
400000000-499999999	0	1	0	0	0	0

TABLE C.4: Node appearances as S, P, O, SP, PO, OS for UniProt

C.2.3 Aggregate Node Reuse

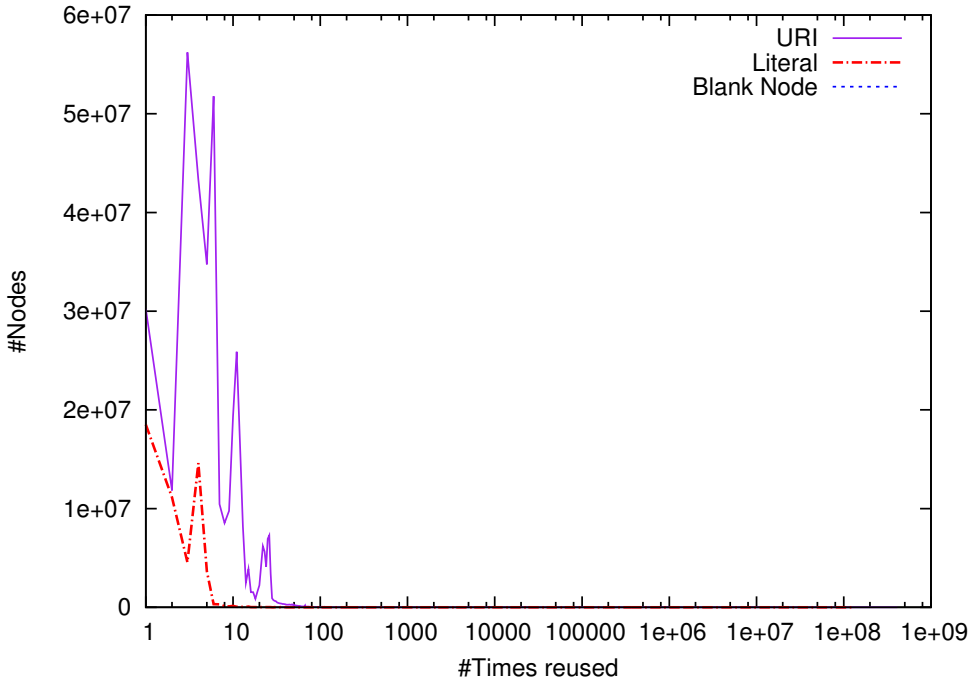


FIGURE C.9: Node reuse data for UniProt

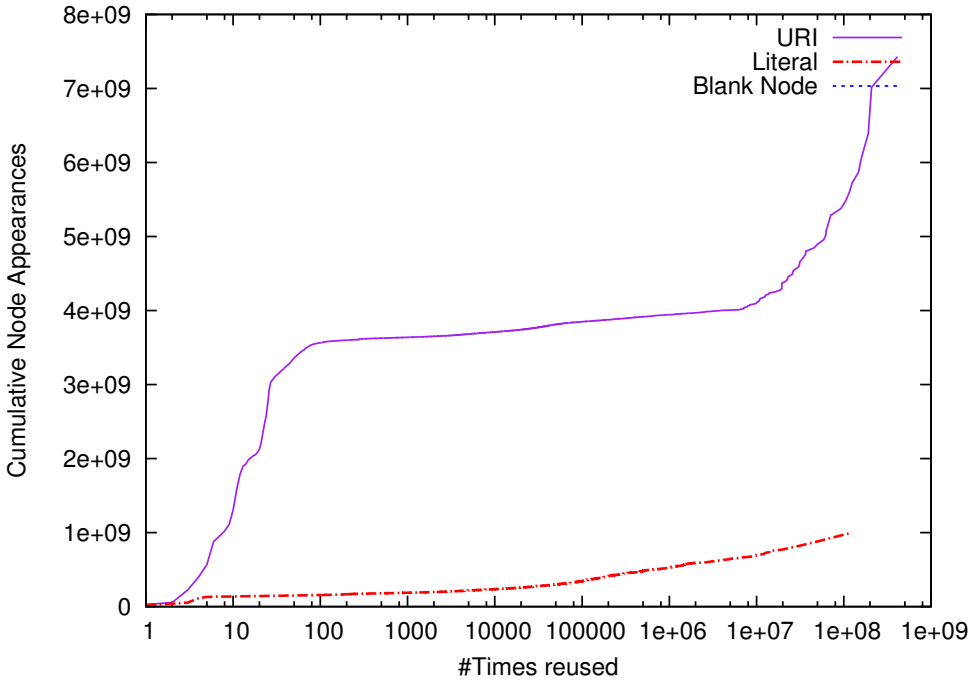


FIGURE C.10: Cumulative node reuse data for UniProt

#Times reused	URI	Literal	Blank Node
Total	391273031	54206102	0
1-1	30238413	18507119	0
2-2	11838128	11128387	0
3-3	56231643	4545853	0
4-4	43321078	14607056	0
5-5	34746557	3650391	0
6-6	51788561	322830	0
7-7	10446355	296079	0
8-8	8530234	100995	0
9-9	9769873	89455	0
10-19	81829719	448770	0
20-29	41641375	151686	0
30-39	4324233	73589	0
40-49	2703065	43194	0
50-59	1649321	29750	0
60-69	919664	21900	0
70-79	568046	16761	0
80-89	202983	13348	0

continued on next page

#Times reused	URI	Literal	Blank Node
90-99	103007	11801	0
100-199	282185	69325	0
200-299	68354	33570	0
300-399	16432	8437	0
400-499	8486	5091	0
500-599	5708	3421	0
600-699	4098	2622	0
700-799	2975	1920	0
800-899	2260	1518	0
900-999	1768	1232	0
1000-1099	10	15	0
1000-1999	9527	6562	0
2000-2999	4173	2967	0
3000-3999	3154	2071	0
4000-4999	2110	1377	0
5000-5999	1402	906	0
6000-6999	890	591	0
7000-7999	714	408	0
8000-8999	590	371	0
9000-9999	466	311	0
10000-19999	2223	1580	0
20000-29999	1091	819	0
30000-39999	720	283	0
40000-49999	430	243	0
50000-59999	236	210	0
60000-69999	135	84	0
70000-79999	77	109	0
80000-89999	53	140	0
90000-99999	47	94	0
100000-199999	196	490	0
200000-299999	69	128	0
300000-399999	39	56	0
400000-499999	20	41	0
500000-599999	14	28	0
600000-699999	12	13	0
700000-799999	8	9	0
800000-899999	5	8	0

continued on next page

#Times reused	URI	Literal	Blank Node
900000-999999	3	7	0
1000000-1999999	19	49	0
2000000-2999999	6	7	0
3000000-3999999	5	4	0
4000000-4999999	0	5	0
5000000-5999999	0	1	0
6000000-6999999	1	2	0
7000000-7999999	7	1	0
8000000-8999999	2	0	0
9000000-9999999	2	3	0
10000000-19999999	19	5	0
20000000-29999999	8	1	0
30000000-39999999	7	1	0
40000000-49999999	1	1	0
50000000-59999999	2	0	0
60000000-69999999	4	0	0
70000000-79999999	1	0	0
90000000-99999999	1	0	0
100000000-199999999	7	1	0
200000000-299999999	3	0	0
400000000-499999999	1	0	0

Table C.5: Node reuse data for UniProt

C.2.4 Node lengths

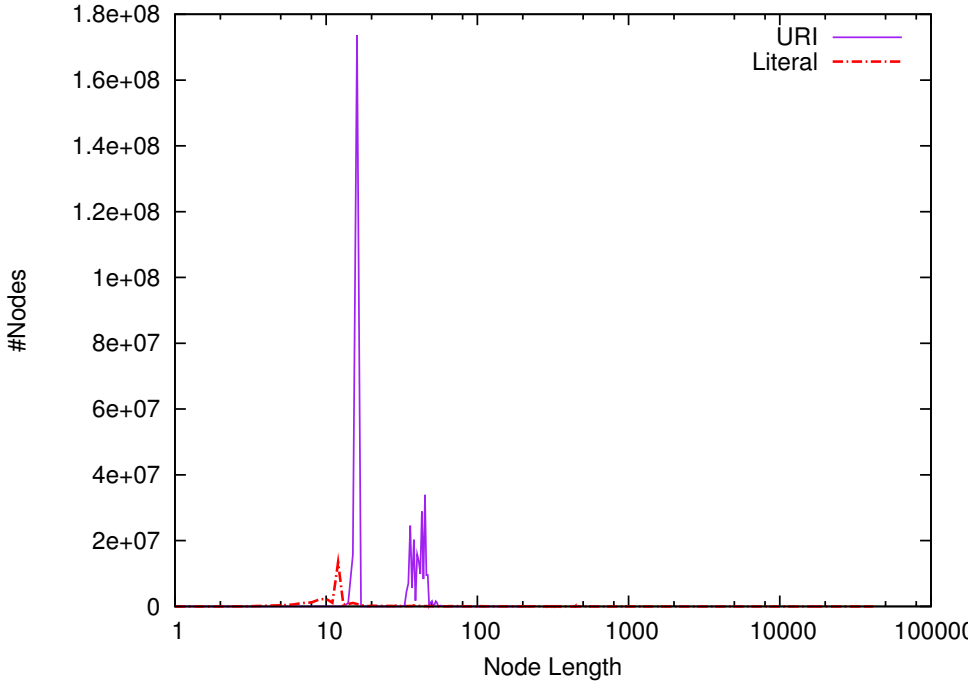


FIGURE C.11: Node length data for UniProt

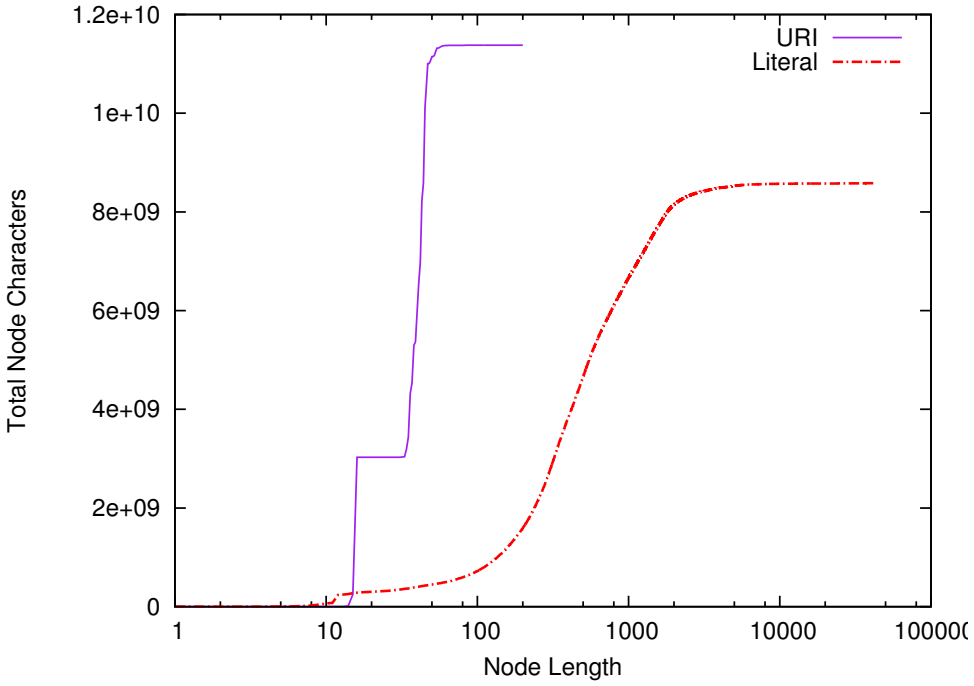


FIGURE C.12: Cumulative node length data for UniProt

Node Length	**URI**	**Literal**
Total	391273031	54206101
1-1	0	64
2-2	0	2190
3-3	0	31265
4-4	0	191842
5-5	0	375910
6-6	0	596624
7-7	0	1031449
8-8	0	1193425
9-9	0	2101651
10-19	190439573	20780878
20-29	124	1672825
30-39	63918056	1560708
40-49	131036014	1207753
50-59	5695675	750303
60-69	161304	701346
70-79	7430	762338
80-89	8280	710313
90-99	2293	684261
100-199	4265	5954322
200-299	17	4616940
300-399	0	3200410
400-499	0	1831947
500-599	0	1203891
600-699	0	696823
700-799	0	473241
800-899	0	351584
900-999	0	278286
1000-1099	0	2449
1000-1999	0	1090341
2000-2999	0	110266
3000-3999	0	23946
4000-4999	0	8578
5000-5999	0	4627
6000-6999	0	1245
7000-7999	0	811

continued on next page

Node Length	URI	Literal
8000-8999	0	383
9000-9999	0	282
10000-19999	0	372
20000-29999	0	141
30000-39999	0	69
40000-49999	0	2

Table C.6: Node length data for UniProt

C.3 CIA World Factbook

C.3.1 Summary

Triple Count: 161489
URI Count: 30338
Average URI length: 48.41, Standard Deviation: 11.24
Average URI reuse: 13.87
Appeared as (ignoring literals):
S only: 9963
P only: 87
S and P: 0
O only: 28
O and S: 20186
P and O: 74
S, P and O: 0
O including literals: 22311
Literal Count: 22283
Average literal length: 42.96, Standard Deviation: 131.07
Average literal reuse: 2.86
Blank Node Count: 0
Average Blank Node reuse: 0.00

C.3.2 Node appearances as S, P, O, SP, PO, OS

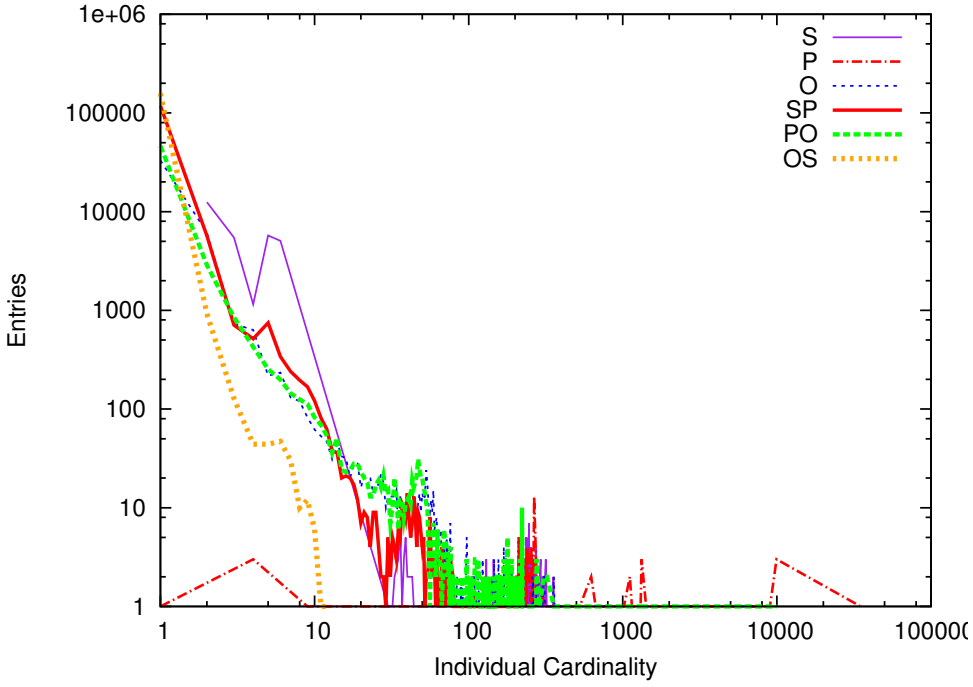


FIGURE C.13: Node and pairing data for CIA World Factbook

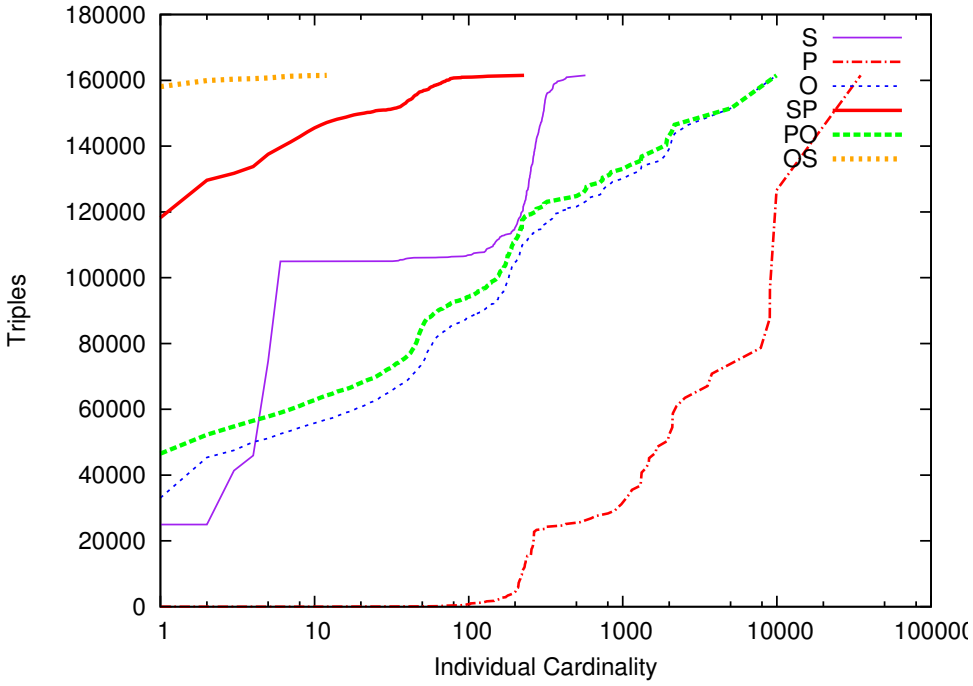


FIGURE C.14: Cumulative node and pairing data for CIA World Factbook

Cardinality	**S**	**P**	**O**	**SP**	**PO**	**OS**
Total	30149	161	42571	127531	52745	159346
1-1	0	1	33095	118222	46517	158100
2-2	12476	0	6125	5700	2878	923
3-3	5463	0	729	708	851	128
4-4	1152	3	633	516	426	44
5-5	5734	0	218	746	254	44
6-6	5058	0	234	340	198	47
7-7	0	0	131	240	144	30
8-8	0	0	120	196	125	10
9-9	0	1	82	168	112	12
10-19	0	1	380	429	419	8
20-29	0	1	176	55	174	0
30-39	22	0	111	59	114	0
40-49	7	0	111	77	186	0
50-59	0	1	135	33	91	0
60-69	2	2	47	21	33	0
70-79	3	1	26	14	19	0
80-89	0	1	11	2	9	0
90-99	5	3	14	1	9	0
100-199	54	24	107	3	110	0
200-299	133	80	43	1	44	0
300-399	36	3	17	0	8	0
400-499	3	2	2	0	1	0
500-599	1	1	5	0	6	0
600-699	0	3	1	0	1	0
700-799	0	0	4	0	3	0
800-899	0	2	2	0	2	0
900-999	0	2	0	0	0	0
1000-1999	0	14	6	0	7	0
2000-2999	0	6	4	0	2	0
3000-3999	0	2	0	0	0	0
5000-5999	0	0	1	0	1	0
7000-7999	0	1	0	0	0	0
8000-8999	0	1	0	0	0	0
9000-9999	0	4	1	0	1	0
30000-39999	0	1	0	0	0	0

TABLE C.7: Node appearances as S, P, O, SP, PO, OS for CIA World Factbook

C.3.3 Aggregate Node Reuse

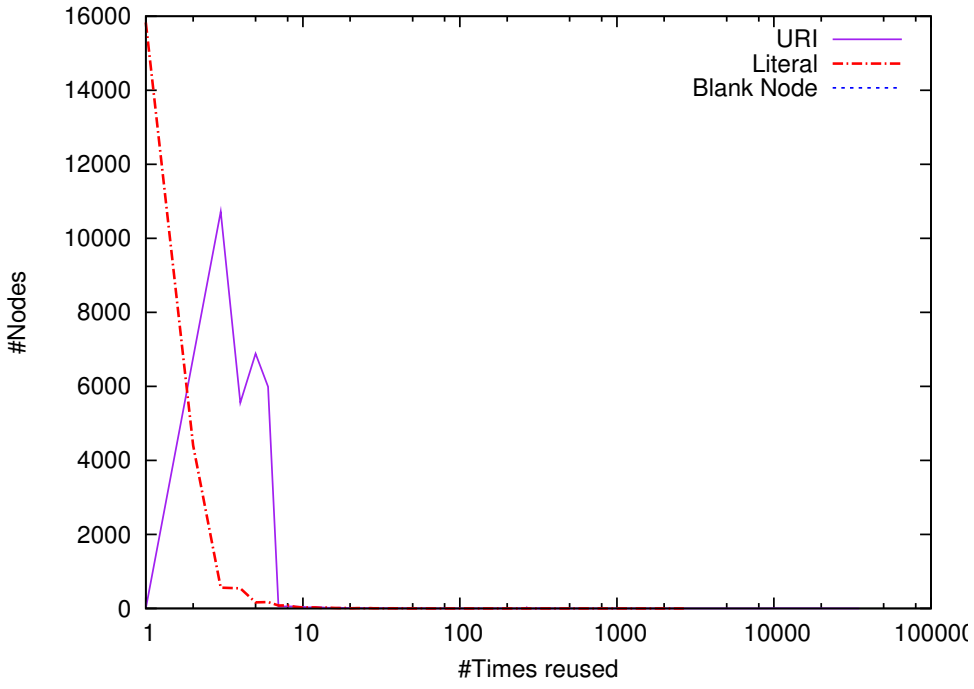


FIGURE C.15: Node reuse data for CIA World Factbook

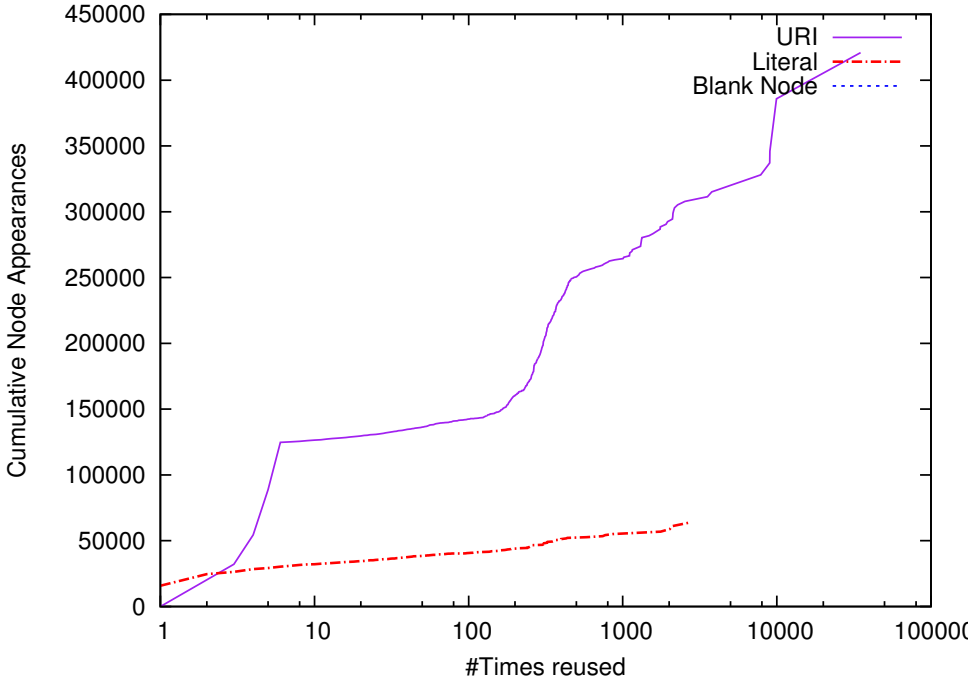


FIGURE C.16: Cumulative node reuse data for CIA World Factbook

#Times reused	URI	Literal	Blank Node
Total	30338	22283	0
1-1	1	15830	0
2-2	0	4405	0
3-3	10723	562	0
4-4	5562	542	0
5-5	6883	163	0
6-6	5991	173	0
7-7	53	80	0
8-8	55	83	0
9-9	52	42	0
10-19	248	181	0
20-29	106	68	0
30-39	68	39	0
40-49	40	23	0
50-59	40	15	0
60-69	23	10	0
70-79	15	6	0
80-89	10	0	0
90-99	8	3	0
100-199	110	23	0
200-299	142	11	0
300-399	109	14	0
400-499	39	2	0
500-599	10	1	0
600-699	4	0	0
700-799	4	2	0
800-899	3	1	0
1000-1999	21	1	0
2000-2999	7	3	0
3000-3999	2	0	0
5000-5999	1	0	0
7000-7999	1	0	0
8000-8999	1	0	0
9000-9999	5	0	0
30000-39999	1	0	0

Table C.8: Node reuse data for CIA World Factbook

C.3.4 Node lengths

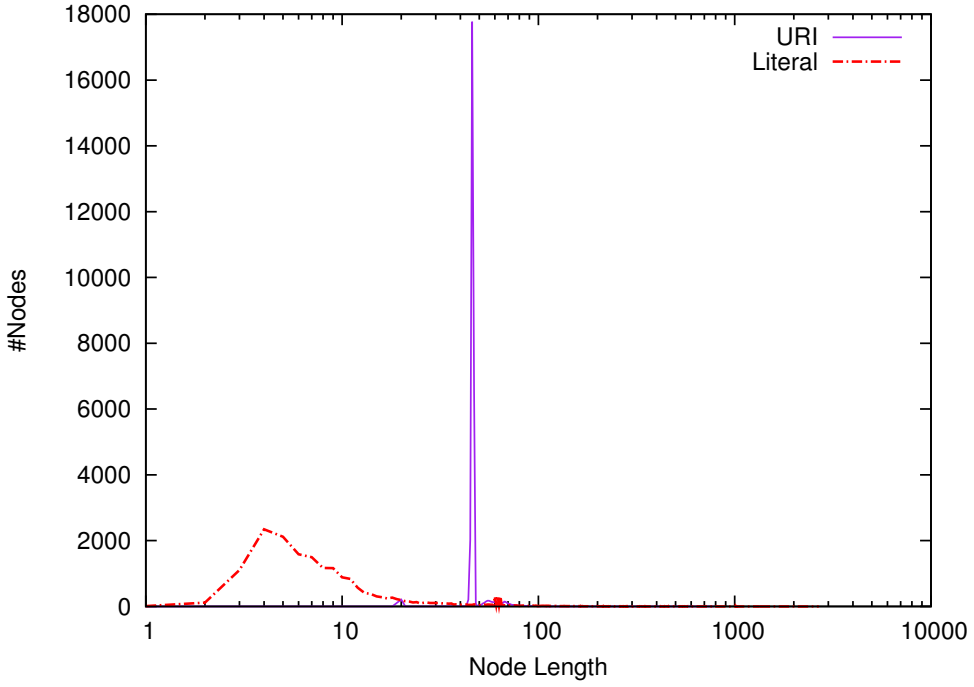


FIGURE C.17: Node length data for CIA World Factbook

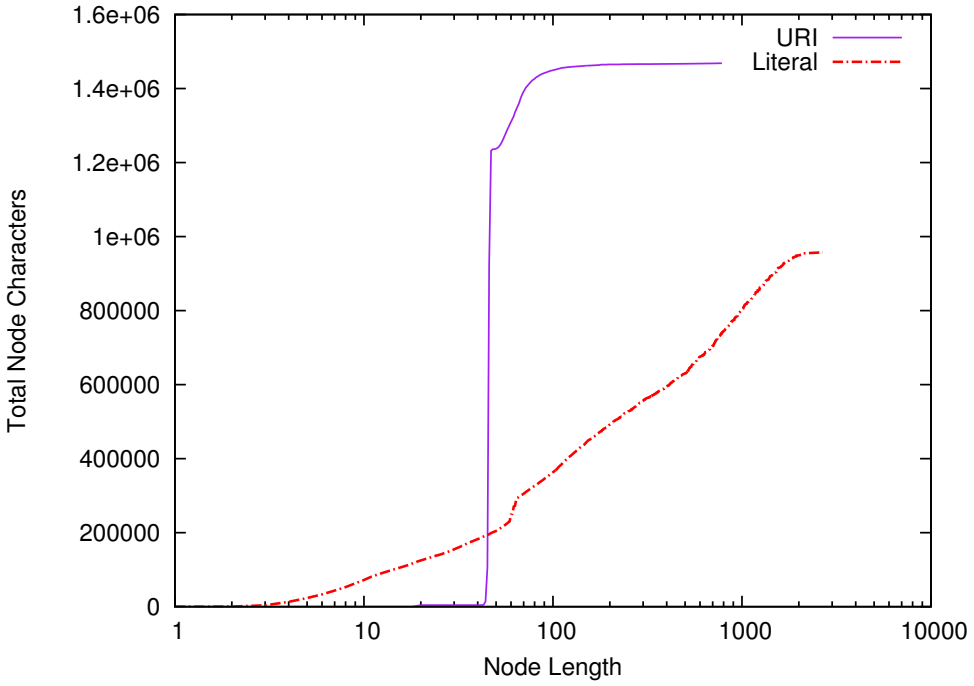


FIGURE C.18: Cumulative node length data for CIA World Factbook

Node Length	**URI**	**Literal**
Total	30338	22282
1-1	0	11
2-2	0	116
3-3	0	1107
4-4	0	2344
5-5	0	2119
6-6	0	1586
7-7	0	1497
8-8	0	1170
9-9	0	1162
10-19	1	4397
20-29	214	1287
30-39	0	837
40-49	26689	503
50-59	1186	506
60-69	1299	1162
70-79	529	292
80-89	195	219
90-99	86	201
100-199	129	949
200-299	5	265
300-399	1	112
400-499	1	81
500-599	1	82
600-699	0	44
700-799	2	54
800-899	0	34
900-999	0	28
1000-1099	0	2
1000-1999	0	111
2000-2999	0	4

Table C.9: Node length data for CIA World Factbook

C.4 Jamendo Music

C.4.1 Summary

Triple Count: 1047950
URI Count: 410929
Average URI length: 48.10, Standard Deviation: 14.35
Average URI reuse: 7.04
Appeared as (ignoring literals):
S only: 45634
P only: 25
S and P: 0
O only: 74979
O and S: 290291
P and O: 0
S, P and O: 0
O including literals: 148663
Literal Count: 73684
Average literal length: 115.42, Standard Deviation: 412.85
Average literal reuse: 3.42
Blank Node Count: 0
Average Blank Node reuse: 0.00

C.4.2 Node appearances as S, P, O, SP, PO, OS

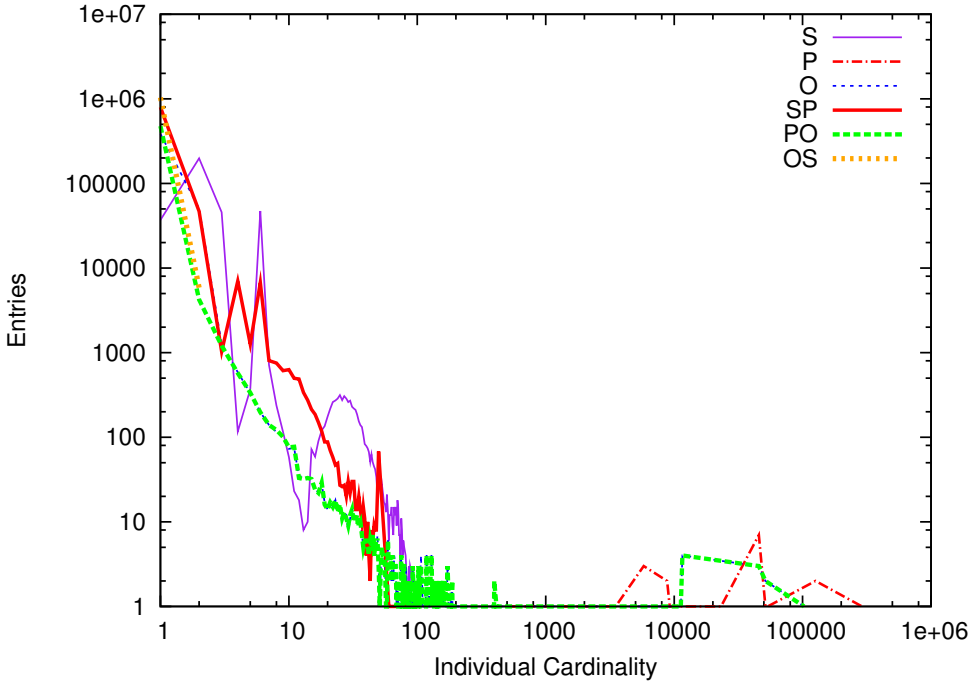


FIGURE C.19: Node and pairing data for Jamendo Music Data

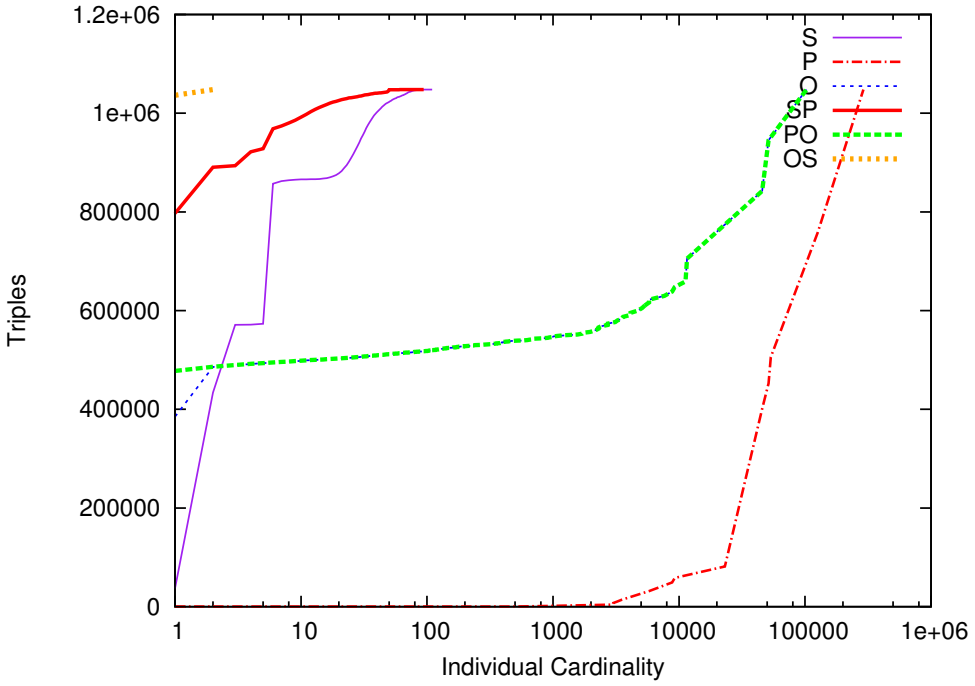


FIGURE C.20: Cumulative node and pairing data for Jamendo Music Data

Cardinality	**S**	**P**	**O**	**SP**	**PO**	**OS**
Total	335925	25	438954	865837	485446	1042078
1-1	36903	0	384903	797072	477630	1036206
2-2	198463	0	50397	46780	4229	5872
3-3	45774	0	1266	1029	1207	0
4-4	117	0	587	7029	573	0
5-5	368	0	335	1258	337	0
6-6	47305	0	198	6737	197	0
7-7	715	0	140	806	140	0
8-8	237	0	115	753	122	0
9-9	116	0	104	612	101	0
10-19	594	0	372	2984	374	0
20-29	2653	0	137	442	137	0
30-39	1730	0	96	180	95	0
40-49	526	0	54	82	54	0
50-59	208	0	31	68	31	0
60-69	112	0	25	3	25	0
70-79	84	0	16	1	17	0
80-89	15	0	12	0	11	0
90-99	4	0	12	1	12	0
100-199	1	0	71	0	71	0
200-299	0	0	14	0	14	0
300-399	0	0	13	0	14	0
400-499	0	1	6	0	5	0
500-599	0	0	2	0	2	0
600-699	0	0	3	0	3	0
700-799	0	0	2	0	2	0
800-899	0	1	1	0	1	0
900-999	0	0	3	0	3	0
1000-1999	0	0	7	0	7	0
2000-2999	0	1	7	0	7	0
3000-3999	0	3	4	0	4	0
4000-4999	0	0	3	0	3	0
5000-5999	0	3	4	0	4	0
7000-7999	0	0	1	0	1	0
8000-8999	0	2	1	0	1	0
9000-9999	0	1	1	0	1	0

continued on next page

Cardinality	S	P	O	SP	PO	OS
10000-19999	0	0	5	0	5	0
20000-29999	0	1	0	0	0	0
40000-49999	0	7	3	0	3	0
50000-59999	0	2	2	0	2	0
100000-199999	0	2	1	0	1	0
200000-299999	0	1	0	0	0	0

TABLE C.10: Node appearances as S, P, O, SP, PO, OS for Jamendo Music Data

C.4.3 Aggregate Node Reuse

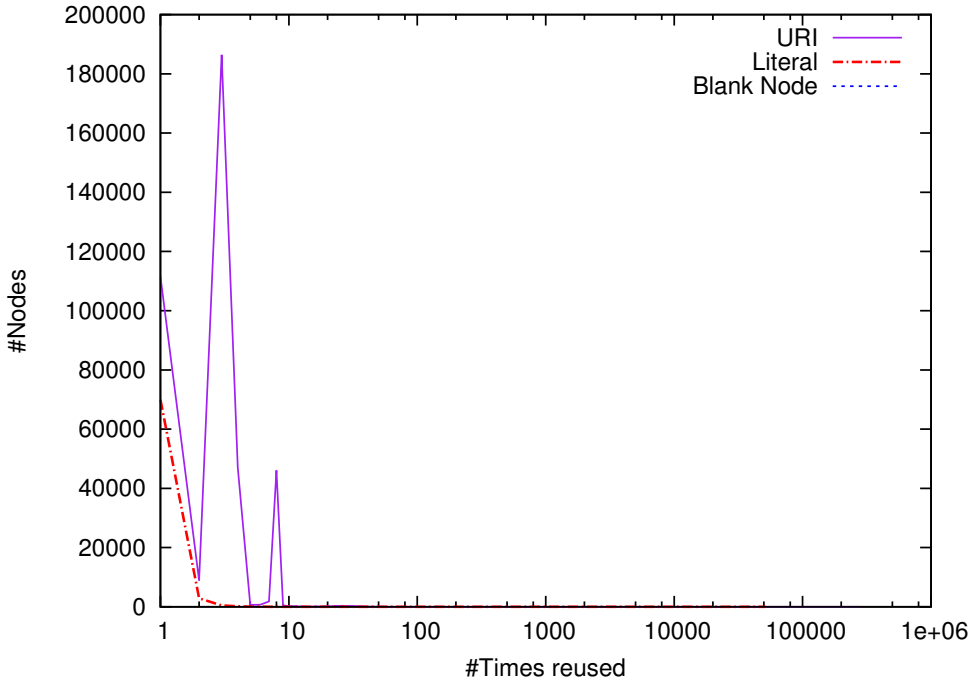


FIGURE C.21: Node reuse data for Jamendo Music Data

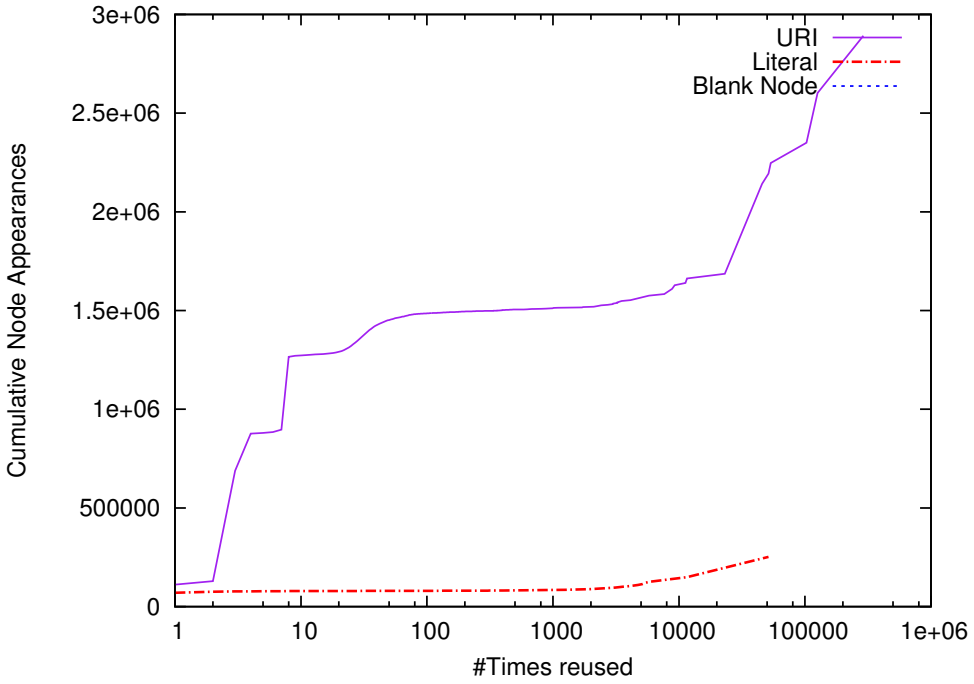


FIGURE C.22: Cumulative node reuse data for Jamendo Music Data

#Times reused	URI	Literal	Blank Node
Total	410929	73684	0
1-1	111515	69938	0
2-2	8920	2857	0
3-3	186421	478	0
4-4	47002	152	0
5-5	662	71	0
6-6	685	37	0
7-7	1811	23	0
8-8	46101	22	0
9-9	553	20	0
10-19	1212	37	0
20-29	2699	9	0
30-39	2004	6	0
40-49	642	3	0
50-59	251	0	0
60-69	144	2	0
70-79	103	0	0
80-89	35	1	0
90-99	15	0	0

continued on next page

#Times reused	URI	Literal	Blank Node
100-199	69	5	0
200-299	13	1	0
300-399	11	2	0
400-499	6	1	0
500-599	1	1	0
600-699	3	0	0
700-799	1	1	0
800-899	2	0	0
900-999	2	1	0
1000-1999	4	3	0
2000-2999	6	2	0
3000-3999	5	2	0
4000-4999	1	2	0
5000-5999	4	3	0
7000-7999	1	0	0
8000-8999	3	0	0
9000-9999	2	0	0
10000-19999	3	2	0
20000-29999	1	0	0
40000-49999	10	0	0
50000-59999	2	2	0
100000-199999	3	0	0
200000-299999	1	0	0

Table C.11: Node reuse data for Jamendo Music Data

C.4.4 Node lengths

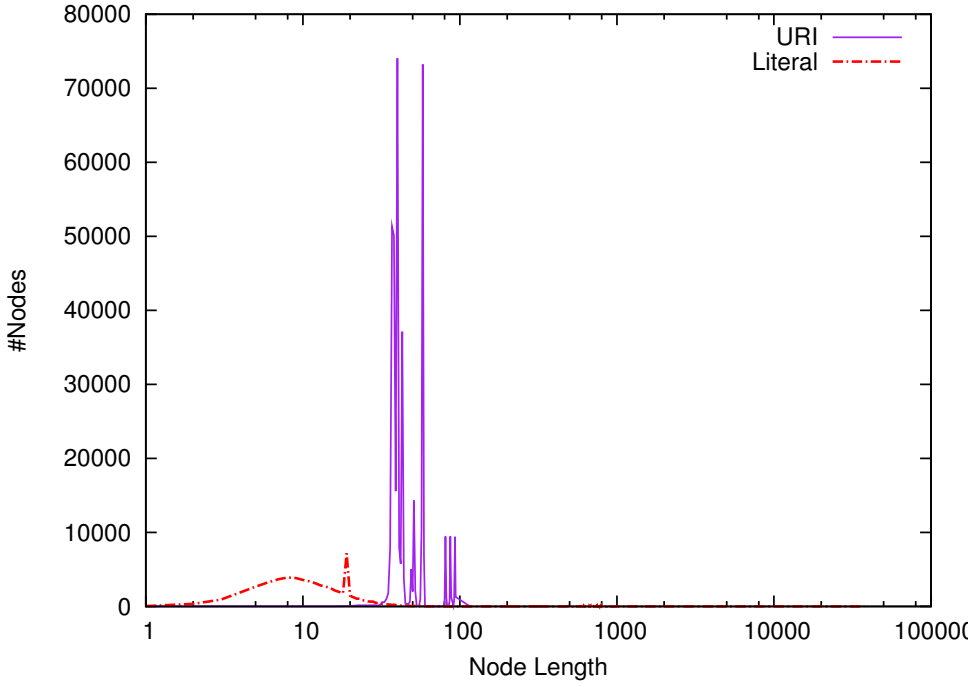


FIGURE C.23: Node length data for Jamendo Music Data

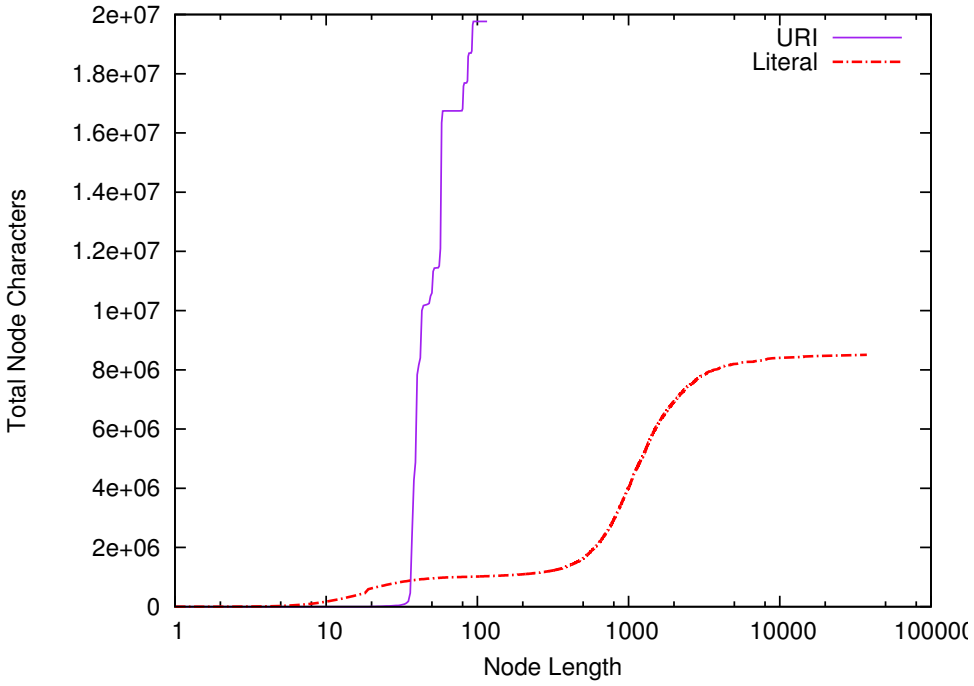


FIGURE C.24: Cumulative node length data for Jamendo Music Data

Node Length	**URI**	**Literal**
Total	410929	73684
1-1	0	52
2-2	0	361
3-3	0	906
4-4	0	1907
5-5	0	2667
6-6	0	3264
7-7	1	3680
8-8	1	3914
9-9	0	3839
10-19	116	31142
20-29	1467	8891
30-39	129084	3018
40-49	135422	1099
50-59	110055	516
60-69	72	221
70-79	69	124
80-89	23105	84
90-99	11536	90
100-199	1	514
200-299	0	428
300-399	0	498
400-499	0	553
500-599	0	684
600-699	0	668
700-799	0	678
800-899	0	643
900-999	0	552
1000-1099	0	4
1000-1999	0	2178
2000-2999	0	359
3000-3999	0	84
4000-4999	0	31
5000-5999	0	12
6000-6999	0	4
7000-7999	0	5

continued on next page

Node Length	URI	Literal
8000-8999	0	7
9000-9999	0	1
10000-19999	0	5
30000-39999	0	1

Table C.12: Node length data for Jamendo Music Data

C.5 GeoSpecies

C.5.1 Summary

Triple Count: 2076380
URI Count: 185147
Average URI length: 48.44, Standard Deviation: 16.94
Average URI reuse: 30.80
Appeared as (ignoring literals):
S only: 619
P only: 173
S and P: 0
O only: 86644
O and S: 97710
P and O: 1
S, P and O: 0
O including literals: 281554
Literal Count: 194910
Average literal length: 33.85, Standard Deviation: 19.36
Average literal reuse: 2.70
Blank Node Count: 1
Average Blank Node reuse: 3.00

C.5.2 Node appearances as S, P, O, SP, PO, OS

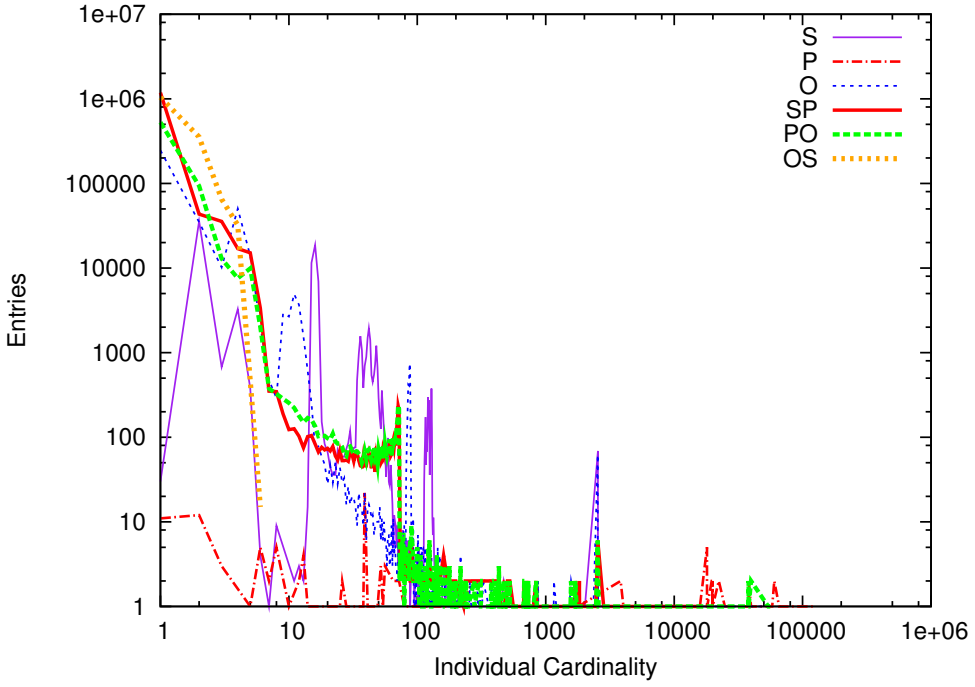


FIGURE C.25: Node and pairing data for GeoSpecies

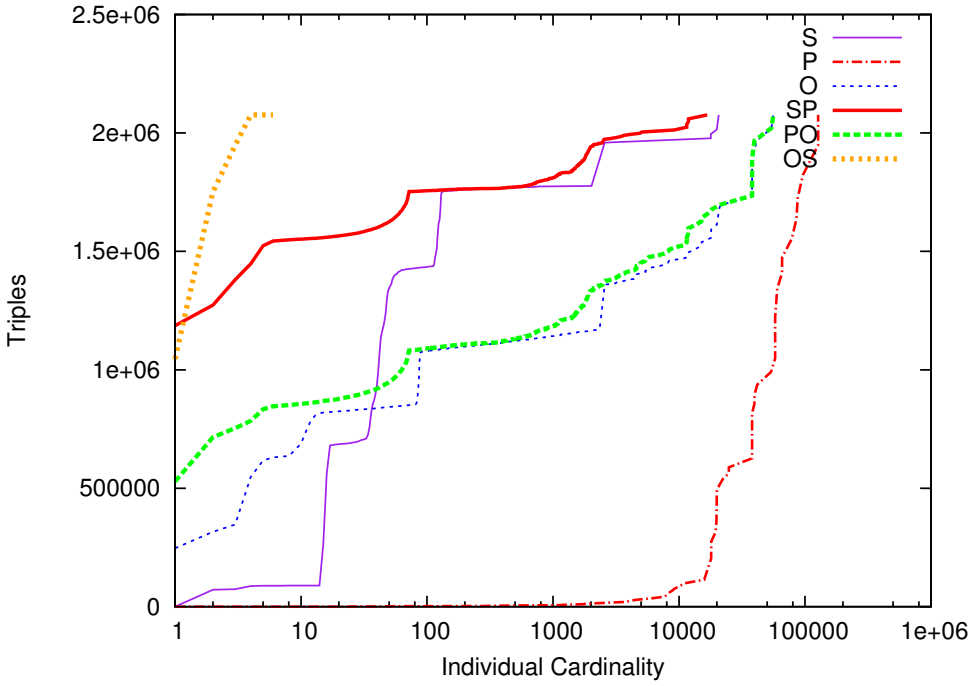


FIGURE C.26: Cumulative node and pairing data for GeoSpecies

Cardinality	**S**	**P**	**O**	**SP**	**PO**	**OS**
Total	98330	174	379266	1306860	661452	1495863
1-1	30	11	246437	1186577	528035	1044944
2-2	35923	12	34904	43459	93688	353634
3-3	677	3	10175	35506	12888	65002
4-4	3262	0	50406	16932	7554	32268
5-5	378	1	13904	15149	9916	0
6-6	4	5	2068	3366	2014	15
7-7	1	2	474	350	377	0
8-8	9	5	290	346	326	0
9-9	0	0	2894	189	285	0
10-19	37489	11	13945	931	1584	0
20-29	589	7	331	617	834	0
30-39	5478	25	161	563	655	0
40-49	9895	5	109	547	611	0
50-59	1537	7	80	610	657	0
60-69	190	2	53	836	882	0
70-79	55	3	49	608	637	0
80-89	35	0	2564	18	38	0
90-99	25	0	37	14	36	0
100-199	2641	1	145	54	124	0
200-299	12	3	36	7	24	0
300-399	5	2	21	6	22	0
400-499	8	2	16	9	22	0
500-599	4	1	9	9	18	0
600-699	1	0	7	13	22	0
700-799	3	0	7	17	17	0
800-899	0	0	7	6	10	0
900-999	0	0	0	9	9	0
1000-1999	0	5	17	85	98	0
2000-2999	73	0	78	14	20	0
3000-3999	0	2	5	4	9	0
4000-4999	0	2	6	2	8	0
5000-5999	0	0	4	1	5	0
6000-6999	0	1	1	0	1	0
7000-7999	0	1	1	0	1	0
8000-8999	0	3	2	0	2	0

continued on next page

Cardinality	S	P	O	SP	PO	OS
9000-9999	0	1	0	1	1	0
10000-19999	3	23	10	5	12	0
20000-29999	3	4	4	0	1	0
30000-39999	0	8	6	0	7	0
40000-49999	0	1	1	0	0	0
50000-59999	0	5	2	0	2	0
60000-69999	0	4	0	0	0	0
70000-79999	0	1	0	0	0	0
80000-89999	0	2	0	0	0	0
90000-99999	0	1	0	0	0	0
100000-199999	0	2	0	0	0	0

TABLE C.13: Node appearances as S, P, O, SP, PO, OS for GeoSpecies

C.5.3 Aggregate Node Reuse

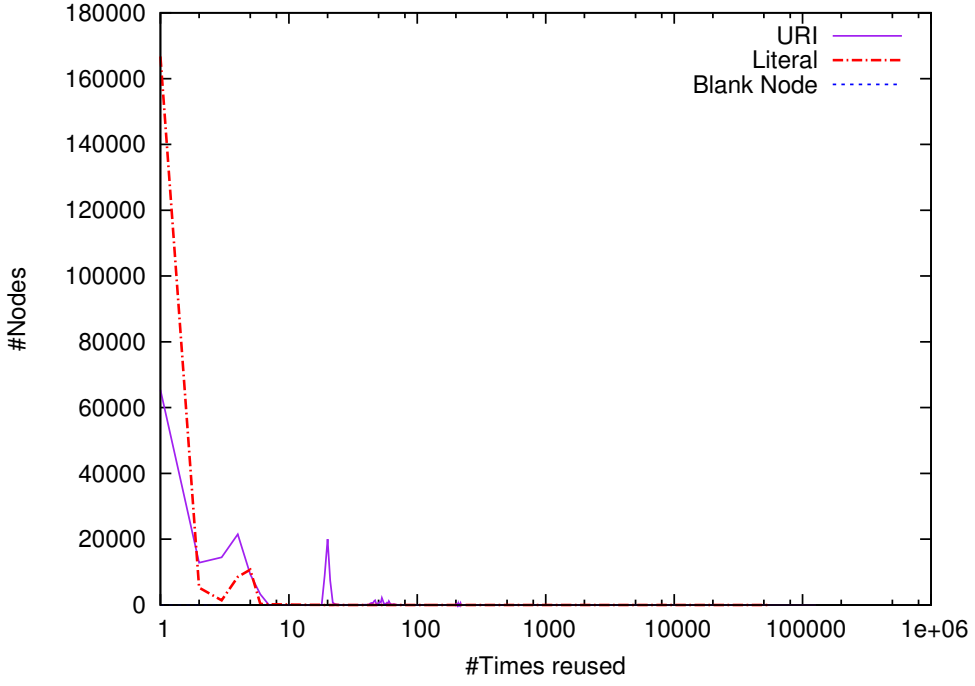


FIGURE C.27: Node reuse data for GeoSpecies

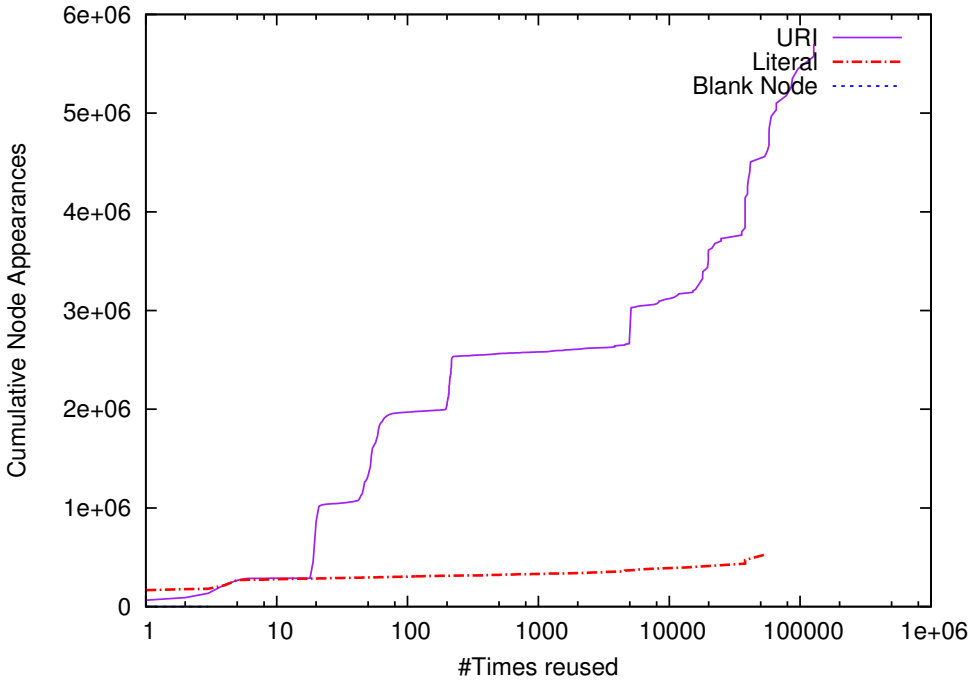


FIGURE C.28: Cumulative node reuse data for GeoSpecies

#Times reused	URI	Literal	Blank Node
Total	185147	194910	1
1-1	65405	166746	0
2-2	12834	5288	0
3-3	14515	1432	1
4-4	21502	8539	0
5-5	9340	10748	0
6-6	3257	314	0
7-7	14	229	0
8-8	10	185	0
9-9	31	146	0
10-19	8967	668	0
20-29	28771	200	0
30-39	625	91	0
40-49	5044	65	0
50-59	8003	45	0
60-69	3243	34	0
70-79	402	25	0
80-89	80	14	0

continued on next page

#Times reused	URI	Literal	Blank Node
90-99	47	18	0
100-199	392	61	0
200-299	2399	12	0
300-399	27	9	0
400-499	25	7	0
500-599	11	5	0
600-699	3	5	0
700-799	6	1	0
800-899	3	2	0
900-999	3	0	0
1000-1999	18	6	0
2000-2999	7	3	0
3000-3999	5	2	0
4000-4999	5	3	0
5000-5999	74	2	0
6000-6999	2	0	0
7000-7999	1	1	0
8000-8999	5	0	0
9000-9999	1	0	0
10000-19999	29	1	0
20000-29999	5	0	0
30000-39999	15	2	0
40000-49999	5	0	0
50000-59999	6	1	0
60000-69999	4	0	0
70000-79999	1	0	0
80000-89999	2	0	0
90000-99999	1	0	0
100000-199999	2	0	0

Table C.14: Node reuse data for GeoSpecies

C.5.4 Node lengths

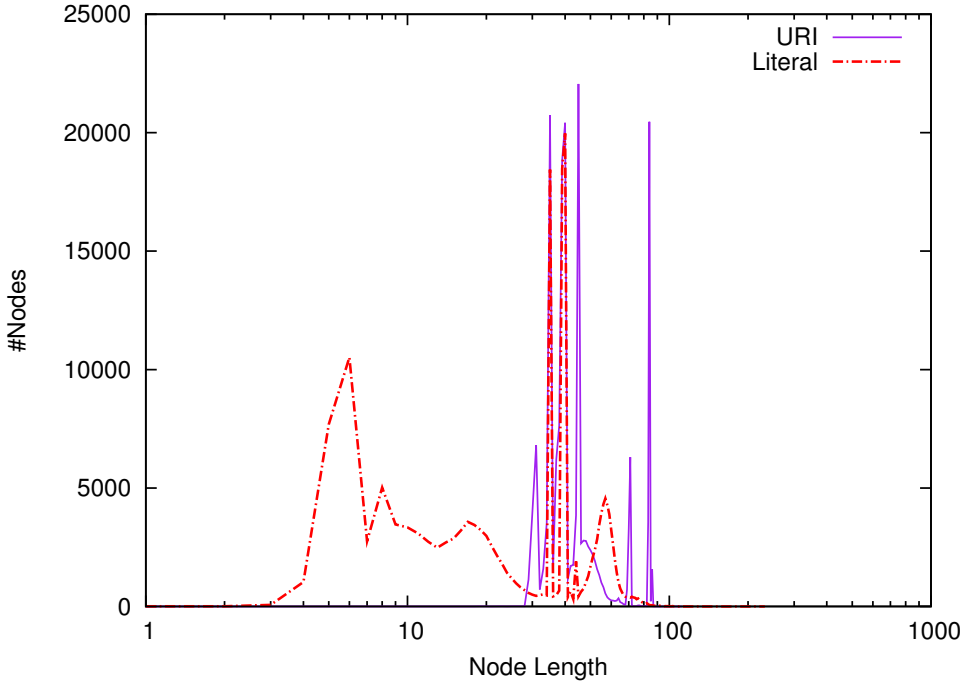


FIGURE C.29: Node length data for GeoSpecies

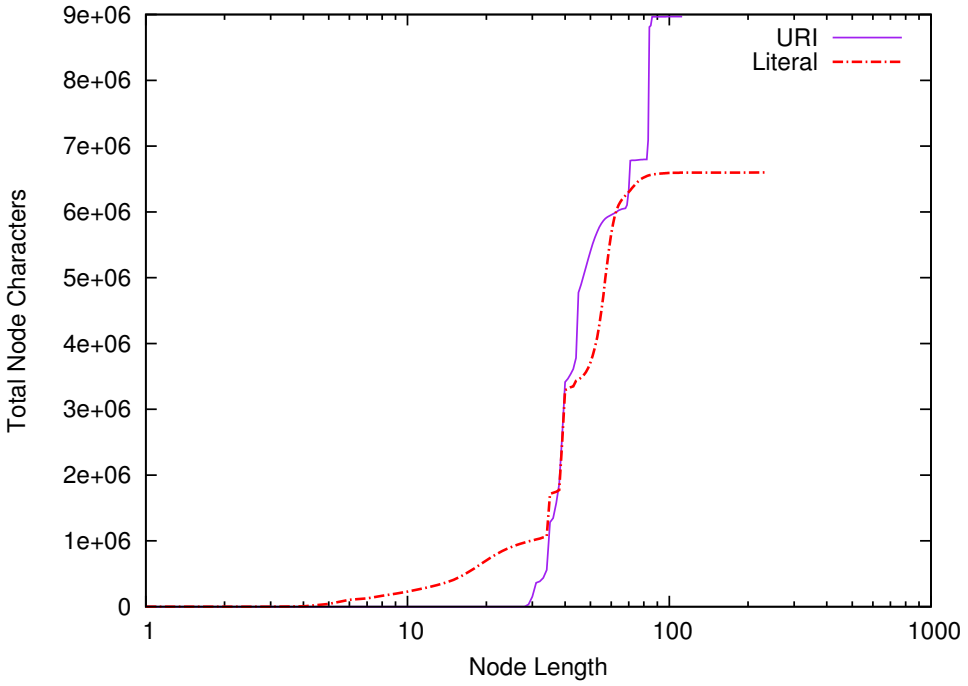


FIGURE C.30: Cumulative node length data for GeoSpecies

Node Length	URI	Literal
Total	185147	194910
1-1	0	4
2-2	0	9
3-3	0	67
4-4	0	1030
5-5	0	7686
6-6	0	10516
7-7	0	2754
8-8	0	5019
9-9	0	3462
10-19	5	30642
20-29	1166	15085
30-39	71673	40885
40-49	61706	27305
50-59	12420	33403
60-69	2533	12845
70-79	9781	3212
80-89	25839	742
90-99	11	172
100-199	13	71
200-299	0	1

Table C.15: Node length data for GeoSpecies

C.6 LinkedCT

C.6.1 Summary

Triple Count: 9804652
URI Count: 1169985
Average URI length: 49.94, Standard Deviation: 5.75
Average URI reuse: 19.31
Appeared as (ignoring literals):
S only: 179319
P only: 80
S and P: 0
O only: 188015
O and S: 802561
P and O: 10

S, P and O: 0
O including literals: 2982612
Literal Count: 2794597
Average literal length: 133.29, Standard Deviation: 532.98
Average literal reuse: 2.44
Blank Node Count: 0
Average Blank Node reuse: 0.00

C.6.2 Node appearances as S, P, O, SP, PO, OS

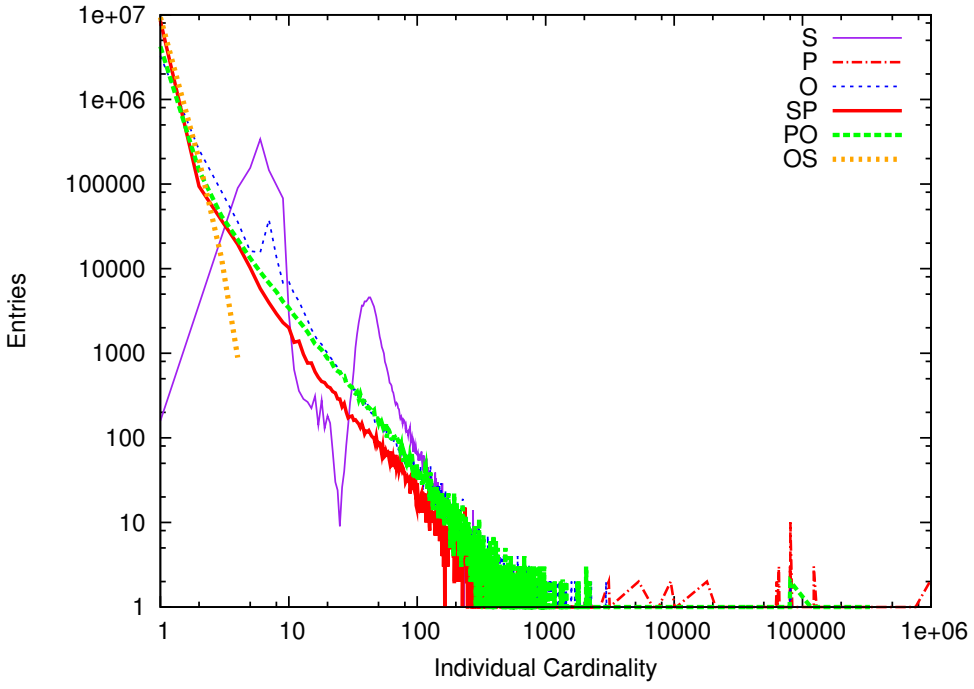


FIGURE C.31: Node and pairing data for LinkedCT

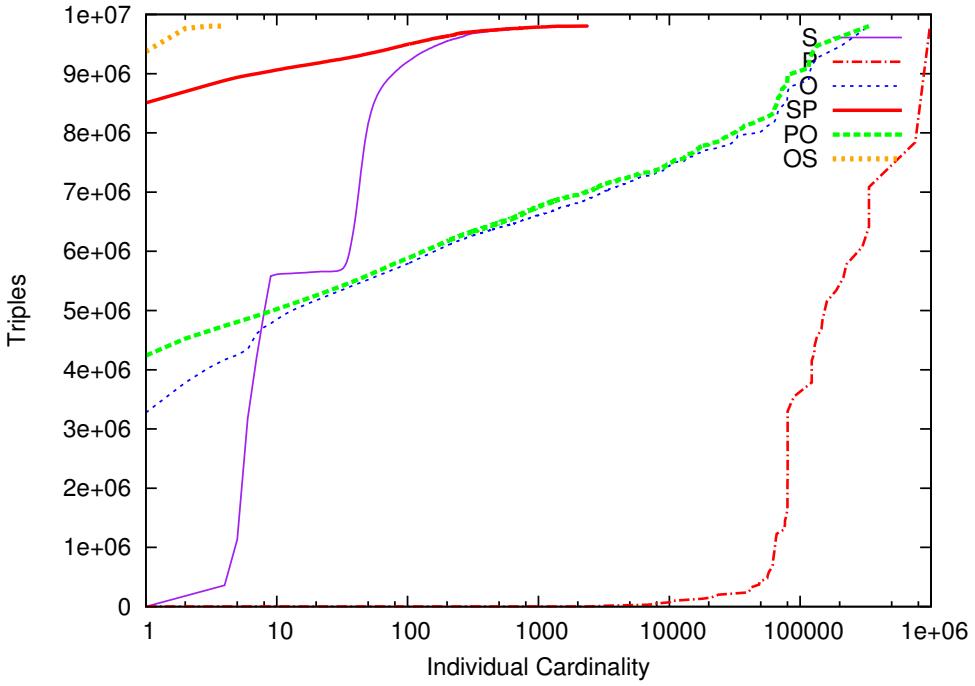


FIGURE C.32: Cumulative node and pairing data for LinkedCT

Cardinality	S	P	O	SP	PO	OS
Total	981880	90	3785183	8702452	4523234	9581290
1-1	159	0	3274987	8507939	4237829	9372840
2-2	0	0	254045	94064	144581	194426
3-3	0	0	81250	37208	41866	13136
4-4	89845	0	35755	19399	22113	888
5-5	154920	0	16082	10246	13301	0
6-6	339532	0	15604	5822	9121	0
7-7	146070	0	37811	3968	6759	0
8-8	97334	0	13777	2935	5350	0
9-9	67890	0	6778	2306	4087	0
10-19	5511	0	29100	9281	18556	0
20-29	753	0	6859	2924	6265	0
30-39	20690	0	3322	1505	3258	0
40-49	37318	0	1948	1053	2047	0
50-59	10935	0	1322	715	1419	0
60-69	3591	0	960	535	1041	0
70-79	1863	0	702	451	773	0
80-89	1238	0	520	312	558	0
90-99	806	0	448	274	451	0

continued on next page

Cardinality	**S**	**P**	**O**	**SP**	**PO**	**OS**
100-199	2665	0	2220	1031	2081	0
200-299	504	0	644	298	632	0
300-399	110	0	248	72	261	0
400-499	54	0	151	39	168	0
500-599	29	0	87	23	102	0
600-699	17	0	66	10	94	0
700-799	11	0	69	15	80	0
800-899	14	0	46	8	59	0
900-999	5	0	30	4	50	0
1000-1999	15	1	0	14	147	0
1000-1099	0	0	1	0	0	0
1000-1999	0	0	148	0	0	0
2000-2999	1	2	68	1	62	0
3000-3999	0	3	33	0	26	0
4000-4999	0	0	19	0	15	0
5000-5999	0	2	14	0	11	0
6000-6999	0	1	3	0	1	0
7000-7999	0	2	7	0	6	0
8000-8999	0	0	4	0	4	0
9000-9999	0	2	8	0	7	0
10000-19999	0	5	20	0	24	0
20000-29999	0	3	5	0	7	0
30000-39999	0	1	4	0	4	0
40000-49999	0	4	1	0	2	0
50000-59999	0	3	2	0	1	0
60000-69999	0	10	4	0	5	0
70000-79999	0	11	2	0	3	0
80000-89999	0	17	3	0	2	0
90000-99999	0	1	0	0	0	0
100000-199999	0	13	4	0	4	0
200000-299999	0	3	1	0	0	0
300000-399999	0	3	1	0	1	0
700000-799999	0	1	0	0	0	0
900000-999999	0	2	0	0	0	0

TABLE C.16: Node appearances as S, P, O, SP, PO, OS for LinkedCT

C.6.3 Aggregate Node Reuse

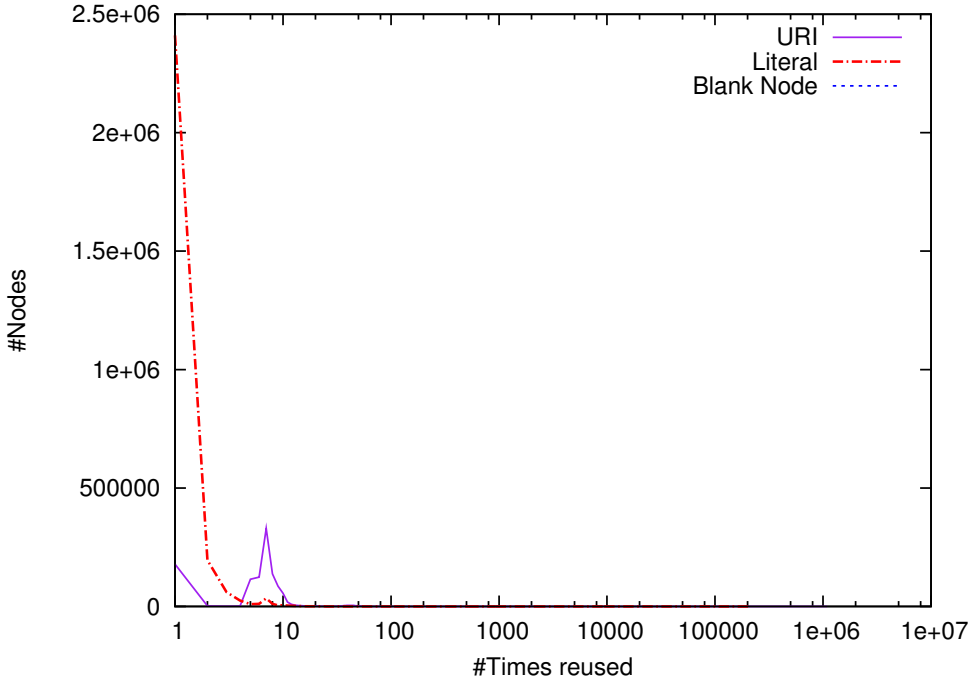


FIGURE C.33: Node reuse data for LinkedCT

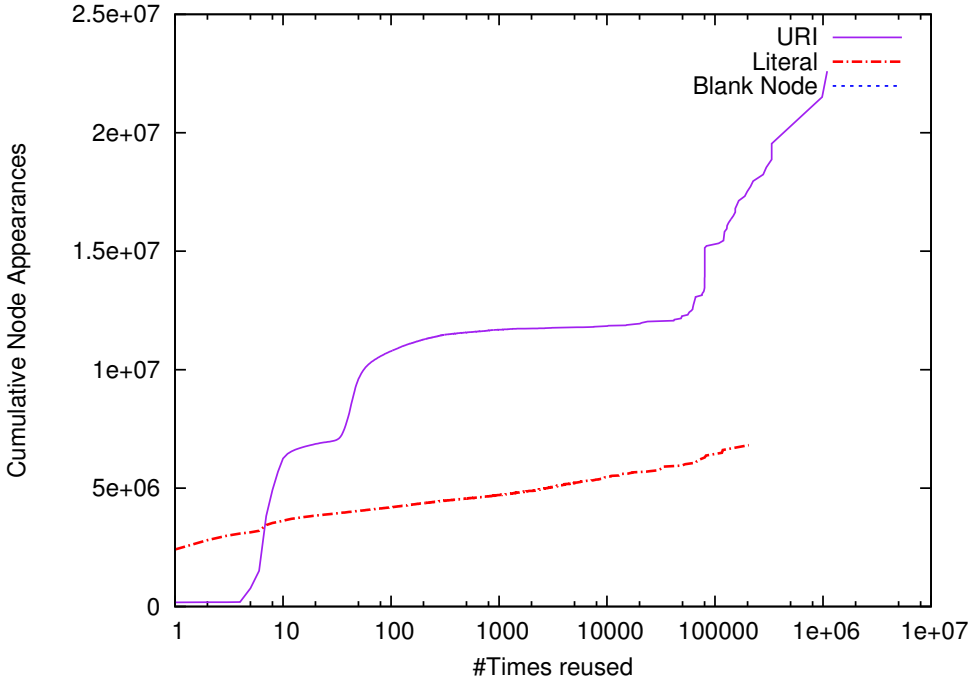


FIGURE C.34: Cumulative node reuse data for LinkedCT

#Times reused	URI	Literal	Blank Node
Total	1169985	2794597	0
1-1	178740	2410524	0
2-2	2171	195527	0
3-3	1031	60756	0
4-4	1318	24973	0
5-5	114789	9666	0
6-6	123971	11054	0
7-7	329675	34553	0
8-8	137704	11124	0
9-9	85809	4672	0
10-19	101529	20121	0
20-29	6429	3912	0
30-39	22863	1868	0
40-49	38453	1130	0
50-59	11614	774	0
60-69	4053	563	0
70-79	2239	396	0
80-89	1486	305	0
90-99	1026	242	0
100-199	3691	1274	0
200-299	784	382	0
300-399	199	162	0
400-499	98	108	0
500-599	58	60	0
600-699	36	46	0
700-799	36	43	0
800-899	28	34	0
900-999	11	24	0
1000-1999	39	0	0
1000-1099	0	1	0
1000-1999	0	125	0
2000-2999	10	61	0
3000-3999	4	31	0
4000-4999	0	19	0
5000-5999	4	12	0
6000-6999	1	3	0

continued on next page

#Times reused	URI	Literal	Blank Node
7000-7999	2	7	0
8000-8999	0	4	0
9000-9999	2	7	0
10000-19999	6	16	0
20000-29999	5	4	0
30000-39999	0	4	0
40000-49999	5	1	0
50000-59999	3	1	0
60000-69999	10	2	0
70000-79999	10	1	0
80000-89999	17	2	0
100000-199999	16	2	0
200000-299999	4	1	0
300000-399999	3	0	0
900000-999999	2	0	0
1000000-1999999	1	0	0

Table C.17: Node reuse data for LinkedCT

C.6.4 Node lengths

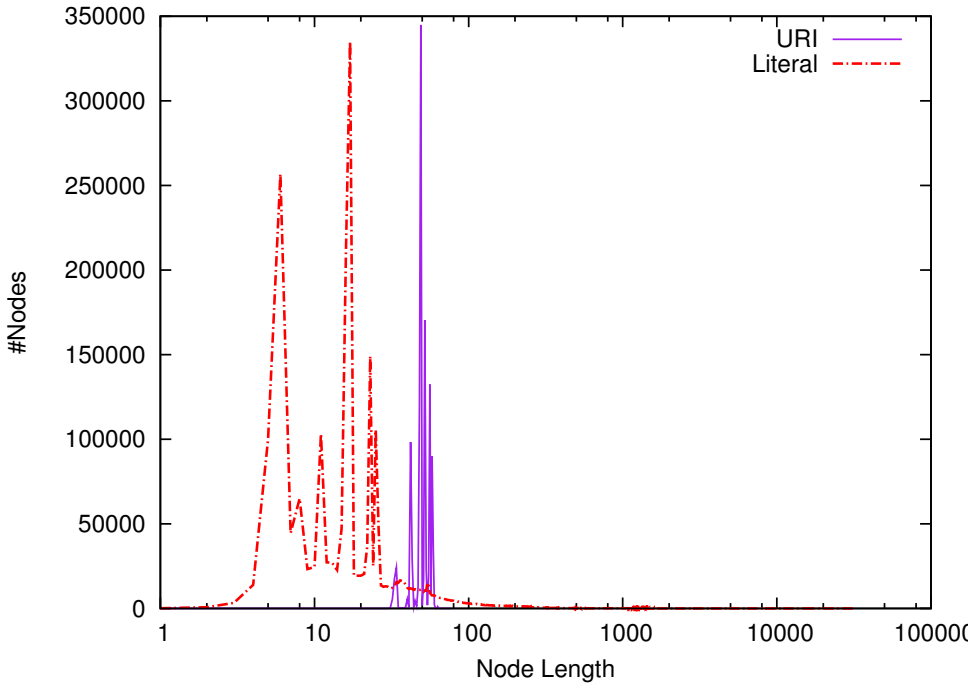


FIGURE C.35: Node length data for LinkedCT

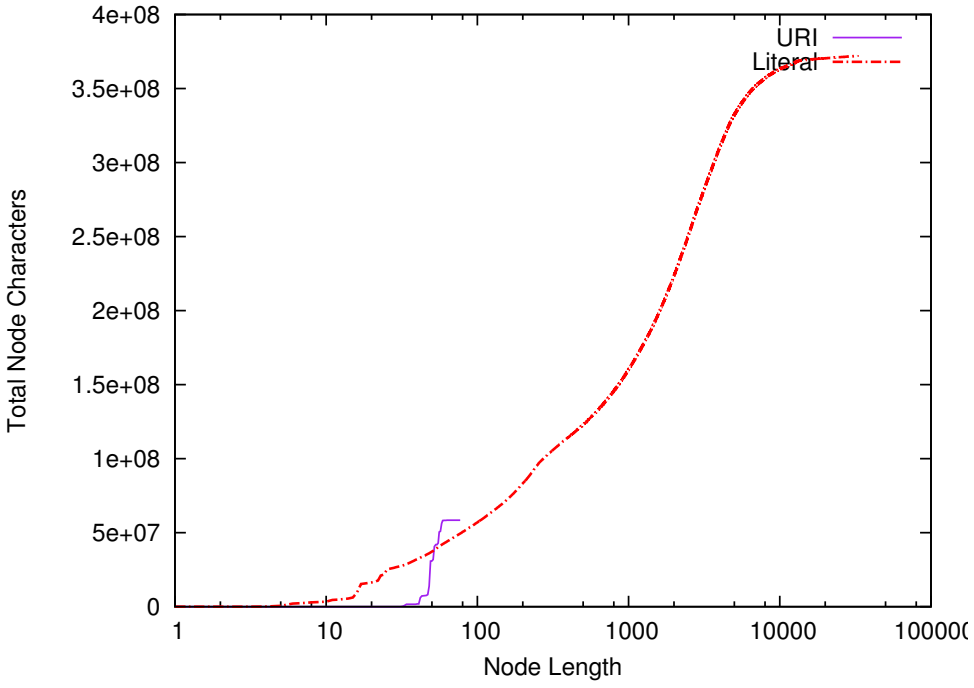


FIGURE C.36: Cumulative node length data for LinkedCT

Node Length	**URI**	**Literal**
Total	1169985	2794596
1-1	0	56
2-2	0	704
3-3	0	3257
4-4	0	13934
5-5	0	100010
6-6	0	256781
7-7	0	44486
8-8	0	64998
9-9	0	23276
10-19	0	835532
20-29	8	444365
30-39	49757	142719
40-49	617098	113396
50-59	499542	103829
60-69	3573	64712
70-79	7	50019
80-89	0	40372
90-99	0	32665
100-199	0	185768
200-299	0	85541
300-399	0	31029
400-499	0	18653
500-599	0	14513
600-699	0	11751
700-799	0	10055
800-899	0	8400
900-999	0	7668
1000-1099	0	66
1000-1999	0	44285
2000-2999	0	21222
3000-3999	0	10056
4000-4999	0	5012
5000-5999	0	2247
6000-6999	0	1196
7000-7999	0	663

continued on next page

Node Length	**URI**	**Literal**
8000-8999	0	416
9000-9999	0	270
10000-19999	0	595
20000-29999	0	56
30000-39999	0	23

Table C.18: Node length data for LinkedCT

Appendix D

Raw Evaluation Data

This appendix contains the raw evaluation data for the evaluation described in Chapter 7. Section D.1 includes the timing-related data, while Section D.2 contains the information on CPU performance counters.

D.1 Timing Data

This section contains the timing data for the evaluation in Chapter 7. Experiments were performed on BSBM datasets, which are unmarked, and DBPedia ones, which are marked with (DBP).

D.1.1 Load Rates

Structure	Attribute Order	Size (M)	Load Time (ms)	Triples Per Second
Bitmap	osp	5.00	3883	1287.66
Bitmap	pos	5.00	4262	1173.16
Bitmap	spo	5.00	4112	1215.95
BPTree	osp	5.00	7411	674.67
BPTree	pos	5.00	8183	611.02
BPTree	spo	5.00	5907	846.45
BST	osp	5.00	12933	386.61
BST	pos	5.00	13408	372.91
BST	spo	5.00	9700	515.46
BTree	osp	5.00	7895	633.31

continued on next page

Structure	Attribute Order	Size (M)	Load Time (ms)	Triples Per Second
BTree	pos	5.00	7891	633.63
BTree	spo	5.00	5933	842.74
BTreeRef	osp	5.00	15840	315.66
BTreeRef	pos	5.00	19048	262.49
BTreeRef	spo	5.00	6073	823.31
AHRI (bp)	osp	5.00	2731	1830.83
AHRI (bp)	pos	5.00	8567	583.63
AHRI (bp)	spo	5.00	1338	3736.91
AHRI (hb)	osp	5.00	3193	1565.92
AHRI (hb)	pos	5.00	2827	1768.66
AHRI (hb)	spo	5.00	1308	3822.62
AHRI (hbwa)	osp	5.00	2703	1849.79
AHRI (hbwa)	pos	5.00	3527	1417.63
AHRI (hbwa)	spo	5.00	1372	3644.31
Hash	osp	5.00	5234	955.29
Hash	pos	5.00	4135	1209.19
Hash	spo	5.00	1877	2663.82
AHRI (vec,bp)	osp	5.00	3116	1604.62
AHRI (vec,bp)	pos	5.00	7582	659.46
AHRI (vec,bp)	spo	5.00	1495	3344.47
AHRI (vec,hb)	osp	5.00	3116	1604.62
AHRI (vec,hb)	pos	5.00	3040	1644.73
AHRI (vec,hb)	spo	5.00	1444	3462.60
AHRI (vec,hbwa)	osp	5.00	3220	1552.79
AHRI (vec,hbwa)	pos	5.00	3345	1494.77
AHRI (vec,hbwa)	spo	5.00	1457	3431.70
BPTree	osp	10.00	7177	1393.33
BPTree	pos	10.00	8220	1216.54
BPTree	spo	10.00	4798	2084.19
AHRI (bp)	osp	10.00	2988	3346.70
AHRI (bp)	pos	10.00	5651	1769.59
AHRI (bp)	spo	10.00	1308	7645.21
AHRI (hb)	osp	10.00	2985	3350.06
AHRI (hb)	pos	10.00	3202	3123.03
AHRI (hb)	spo	10.00	1316	7598.73

continued on next page

Structure	Attribute Order	Size (M)	Load Time (ms)	Triples Per Second
AHRI (hbwa)	osp	10.00	3073	3254.13
AHRI (hbwa)	pos	10.00	3153	3171.56
AHRI (hbwa)	spo	10.00	1307	7651.06
AHRI (vec,bp)	osp	10.00	3042	3287.29
AHRI (vec,bp)	pos	10.00	5595	1787.30
AHRI (vec,bp)	spo	10.00	1415	7067.09
AHRI (vec,hb)	osp	10.00	3014	3317.83
AHRI (vec,hb)	pos	10.00	3077	3249.90
AHRI (vec,hb)	spo	10.00	1454	6877.53
AHRI (vec,hbwa)	osp	10.00	3039	3290.53
AHRI (vec,hbwa)	pos	10.00	3225	3100.75
AHRI (vec,hbwa)	spo	10.00	1442	6934.77
BPTree	osp	30.69	25948	1182.58
BPTree	pos	30.69	29520	1039.48
BPTree	spo	30.69	16159	1898.98
AHRI (bp)	osp	30.69	10271	2987.59
AHRI (bp)	pos	30.69	18560	1653.32
AHRI (bp)	spo	30.69	4173	7353.36
AHRI (hb)	osp	30.69	10262	2990.21
AHRI (hb)	pos	30.69	10759	2852.08
AHRI (hb)	spo	30.69	3886	7896.44
AHRI (hbwa)	osp	30.69	11025	2783.27
AHRI (hbwa)	pos	30.69	11258	2725.67
AHRI (hbwa)	spo	30.69	3934	7800.09
AHRI (vec,bp)	osp	30.69	11251	2727.36
AHRI (vec,bp)	pos	30.69	20546	1493.51
AHRI (vec,bp)	spo	30.69	6061	5062.79
AHRI (vec,hb)	osp	30.69	11230	2732.46
AHRI (vec,hb)	pos	30.69	11213	2736.61
AHRI (vec,hb)	spo	30.69	5699	5384.38
AHRI (vec,hbwa)	osp	30.69	11148	2752.56
AHRI (vec,hbwa)	pos	30.69	13002	2360.07
AHRI (vec,hbwa)	spo	30.69	4480	6849.46
BPTree	osp	46.14 (DBP)	44128	1045.53
BPTree	pos	46.14 (DBP)	48815	945.14

continued on next page

Structure	Attribute Order	Size (M)	Load Time (ms)	Triples Per Second
BPTree	spo	46.14 (DBP)	28609	1612.68
AHRI (bp)	osp	46.14 (DBP)	18565	2485.16
AHRI (bp)	pos	46.14 (DBP)	33836	1363.55
AHRI (bp)	spo	46.14 (DBP)	9552	4830.09
AHRI (hb)	osp	46.14 (DBP)	18670	2471.19
AHRI (hb)	pos	46.14 (DBP)	22639	2037.95
AHRI (hb)	spo	46.14 (DBP)	9091	5075.03
AHRI (hbwa)	osp	46.14 (DBP)	18322	2518.12
AHRI (hbwa)	pos	46.14 (DBP)	23407	1971.08
AHRI (hbwa)	spo	46.14 (DBP)	9124	5056.67
AHRI (vec,bp)	osp	46.14 (DBP)	18881	2443.57
AHRI (vec,bp)	pos	46.14 (DBP)	38767	1190.11
AHRI (vec,bp)	spo	46.14 (DBP)	10076	4578.91
AHRI (vec,hb)	osp	46.14 (DBP)	20387	2263.06
AHRI (vec,hb)	pos	46.14 (DBP)	23453	1967.21
AHRI (vec,hb)	spo	46.14 (DBP)	11172	4129.71
AHRI (vec,hbwa)	osp	46.14 (DBP)	18691	2468.41
AHRI (vec,hbwa)	pos	46.14 (DBP)	22836	2020.37
AHRI (vec,hbwa)	spo	46.14 (DBP)	9966	4629.45
BPTree	osp	65.04	56423	1152.74
BPTree	pos	65.04	63877	1018.22
BPTree	spo	65.04	31479	2066.16
AHRI (bp)	osp	65.04	22989	2829.21
AHRI (bp)	pos	65.04	39988	1626.51
AHRI (bp)	spo	65.04	7777	8363.22
AHRI (hb)	osp	65.04	23340	2786.67
AHRI (hb)	pos	65.04	22129	2939.16
AHRI (hb)	spo	65.04	6909	9413.92
AHRI (hbwa)	osp	65.04	23245	2798.05
AHRI (hbwa)	pos	65.04	23650	2750.14
AHRI (hbwa)	spo	65.04	6888	9442.62
AHRI (vec,bp)	osp	65.04	23665	2748.40
AHRI (vec,bp)	pos	65.04	40108	1621.64
AHRI (vec,bp)	spo	65.04	7754	8388.03
AHRI (vec,hb)	osp	65.04	23540	2762.99

continued on next page

Structure	Attribute Order	Size (M)	Load Time (ms)	Triples Per Second
AHRI (vec,hb)	pos	65.04	22691	2866.37
AHRI (vec,hb)	spo	65.04	7850	8285.45
AHRI (vec,hbwa)	osp	65.04	23684	2746.19
AHRI (vec,hbwa)	pos	65.04	23511	2766.40
AHRI (vec,hbwa)	spo	65.04	7965	8165.82
BPTree	osp	95.47 (DBP)	85273	1119.60
BPTree	pos	95.47 (DBP)	88878	1074.19
BPTree	spo	95.47 (DBP)	48762	1957.91
AHRI (bp)	osp	95.47 (DBP)	33866	2819.10
AHRI (bp)	pos	95.47 (DBP)	54207	1761.25
AHRI (bp)	spo	95.47 (DBP)	15402	6198.66
AHRI (hb)	osp	95.47 (DBP)	33742	2829.46
AHRI (hb)	pos	95.47 (DBP)	36397	2623.07
AHRI (hb)	spo	95.47 (DBP)	15973	5977.07
AHRI (hbwa)	osp	95.47 (DBP)	33422	2856.56
AHRI (hbwa)	pos	95.47 (DBP)	37951	2515.66
AHRI (hbwa)	spo	95.47 (DBP)	15011	6360.12
AHRI (vec,bp)	osp	95.47 (DBP)	35310	2703.82
AHRI (vec,bp)	pos	95.47 (DBP)	54426	1754.16
AHRI (vec,bp)	spo	95.47 (DBP)	16403	5820.39
AHRI (vec,hb)	osp	95.47 (DBP)	36857	2590.33
AHRI (vec,hb)	pos	95.47 (DBP)	36946	2584.09
AHRI (vec,hb)	spo	95.47 (DBP)	16240	5878.81
AHRI (vec,hbwa)	osp	95.47 (DBP)	35032	2725.27
AHRI (vec,hbwa)	pos	95.47 (DBP)	38144	2502.93
AHRI (vec,hbwa)	spo	95.47 (DBP)	16212	5888.96
BPTree	osp	100.00	89010	1123.47
BPTree	pos	100.00	101502	985.20
BPTree	spo	100.00	48897	2045.12
AHRI (bp)	osp	100.00	35527	2814.76
AHRI (bp)	pos	100.00	62061	1611.32
AHRI (bp)	spo	100.00	9635	10378.84
AHRI (hb)	osp	100.00	35534	2814.21
AHRI (hb)	pos	100.00	34038	2937.90
AHRI (hb)	spo	100.00	9542	10479.99

continued on next page

Structure	Attribute Order	Size (M)	Load Time (ms)	Triples Per Second
AHRI (hbwa)	osp	100.00	34959	2860.50
AHRI (hbwa)	pos	100.00	35651	2804.97
AHRI (hbwa)	spo	100.00	9516	10508.63
AHRI (vec,bp)	osp	100.00	35924	2783.66
AHRI (vec,bp)	pos	100.00	62040	1611.87
AHRI (vec,bp)	spo	100.00	10324	9686.18
AHRI (vec,hb)	osp	100.00	36165	2765.11
AHRI (vec,hb)	pos	100.00	34559	2893.61
AHRI (vec,hb)	spo	100.00	10785	9272.15
AHRI (vec,hbwa)	osp	100.00	36179	2764.04
AHRI (vec,hbwa)	pos	100.00	37154	2691.50
AHRI (vec,hbwa)	spo	100.00	10792	9266.13
BPTree	osp	157.84	144978	1088.73
BPTree	pos	157.84	162540	971.10
BPTree	spo	157.84	76768	2056.10
AHRI (bp)	osp	157.84	58899	2679.88
AHRI (bp)	pos	157.84	100776	1566.27
AHRI (bp)	spo	157.84	15883	9937.83
AHRI (hb)	osp	157.84	58166	2713.66
AHRI (hb)	pos	157.84	54886	2875.82
AHRI (hb)	spo	157.84	15003	10520.73
AHRI (hbwa)	osp	157.84	59286	2662.39
AHRI (hbwa)	pos	157.84	57886	2726.78
AHRI (hbwa)	spo	157.84	15161	10411.09
AHRI (vec,bp)	osp	157.84	60132	2624.93
AHRI (vec,bp)	pos	157.84	101866	1549.51
AHRI (vec,bp)	spo	157.84	16631	9490.86
AHRI (vec,hb)	osp	157.84	60765	2597.59
AHRI (vec,hb)	pos	157.84	55927	2822.30
AHRI (vec,hb)	spo	157.84	19159	8238.56
AHRI (vec,hbwa)	osp	157.84	65678	2403.28
AHRI (vec,hbwa)	pos	157.84	59563	2650.01
AHRI (vec,hbwa)	spo	157.84	17175	9190.25
BPTree	osp	189.57 (DBP)	215461	879.85
BPTree	pos	189.57 (DBP)	211123	897.93

continued on next page

Structure	Attribute Order	Size (M)	Load Time (ms)	Triples Per Second
BPTree	spo	189.57 (DBP)	106513	1779.82
AHRI (bp)	osp	189.57 (DBP)	88834	2134.02
AHRI (bp)	pos	189.57 (DBP)	127632	1485.31
AHRI (bp)	spo	189.57 (DBP)	42457	4465.07
AHRI (hb)	osp	189.57 (DBP)	88027	2153.59
AHRI (hb)	pos	189.57 (DBP)	82563	2296.11
AHRI (hb)	spo	189.57 (DBP)	36598	5179.89
AHRI (hbwa)	osp	189.57 (DBP)	88250	2148.14
AHRI (hbwa)	pos	189.57 (DBP)	84749	2236.88
AHRI (hbwa)	spo	189.57 (DBP)	33996	5576.35
AHRI (vec,bp)	osp	189.57 (DBP)	94295	2010.43
AHRI (vec,bp)	pos	189.57 (DBP)	129061	1468.87
AHRI (vec,bp)	spo	189.57 (DBP)	46565	4071.16
AHRI (vec,hb)	osp	189.57 (DBP)	93610	2025.14
AHRI (vec,hb)	pos	189.57 (DBP)	84226	2250.77
AHRI (vec,hb)	spo	189.57 (DBP)	40350	4698.23
AHRI (vec,hbwa)	osp	189.57 (DBP)	93762	2021.86
AHRI (vec,hbwa)	pos	189.57 (DBP)	87997	2154.32
AHRI (vec,hbwa)	spo	189.57 (DBP)	39210	4834.83
BPTree	osp	350.56	411739	851.41
BPTree	pos	350.56	460948	760.52
BPTree	spo	350.56	244327	1434.79
AHRI (bp)	osp	350.56	183657	1908.77
AHRI (bp)	pos	350.56	241811	1449.72
AHRI (bp)	spo	350.56	42363	8275.12
AHRI (hb)	osp	350.56	180659	1940.45
AHRI (hb)	pos	350.56	133678	2622.41
AHRI (hb)	spo	350.56	38720	9053.69
AHRI (hbwa)	osp	350.56	180158	1945.84
AHRI (hbwa)	pos	350.56	140387	2497.09
AHRI (hbwa)	spo	350.56	38208	9175.01
AHRI (vec,bp)	osp	350.56	192900	1817.31
AHRI (vec,bp)	pos	350.56	240258	1459.09
AHRI (vec,bp)	spo	350.56	51673	6784.18
AHRI (vec,hb)	osp	350.56	167737	2089.93

continued on next page

Structure	Attribute Order	Size (M)	Load Time (ms)	Triples Per Second
AHRI (vec,hb)	pos	350.56	135644	2584.40
AHRI (vec,hb)	spo	350.56	44985	7792.80
AHRI (vec,hbwa)	osp	350.56	186607	1878.59
AHRI (vec,hbwa)	pos	350.56	142832	2454.34
AHRI (vec,hbwa)	spo	350.56	44433	7889.61

Table D.1: Load rates

D.1.2 Restriction by One Attribute

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
Bitmap	osp	5.00	500	8363	59.79
Bitmap	pos	5.00	60	5783	10.38
Bitmap	spo	5.00	2000000	6795	294334.07
BPTree	osp	5.00	3000	5918	506.93
BPTree	pos	5.00	600	11053	54.28
BPTree	spo	5.00	2000000	5998	333444.48
BST	osp	5.00	3000	12726	235.74
BST	pos	5.00	600	21415	28.02
BST	spo	5.00	2000000	11357	176102.84
BTree	osp	5.00	3000	7650	392.16
BTree	pos	5.00	600	13844	43.34
BTree	spo	5.00	2000000	7314	273448.18
BTreeRef	osp	5.00	3000	6701	447.69
BTreeRef	pos	5.00	600	13978	42.92
BTreeRef	spo	5.00	2000000	9912	201775.63
AHRI (bp)	osp	5.00	3000	3156	950.57
AHRI (bp)	pos	5.00	600	5378	111.57
AHRI (bp)	spo	5.00	2000000	1458	1371742.11
AHRI (hb)	osp	5.00	3000	3198	938.09
AHRI (hb)	pos	5.00	600	9133	65.70
AHRI (hb)	spo	5.00	2000000	1780	1123595.51
AHRI (hbwa)	osp	5.00	3000	3083	973.08
AHRI (hbwa)	pos	5.00	600	5489	109.31

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (hbwa)	spo	5.00	2000000	1785	1120448.18
Hash	osp	5.00	3000	9169	327.19
Hash	pos	5.00	600	18383	32.64
Hash	spo	5.00	2000000	6025	331950.21
AHRI (vec,bp)	osp	5.00	3000	2817	1064.96
AHRI (vec,bp)	pos	5.00	600	5276	113.72
AHRI (vec,bp)	spo	5.00	2000000	1636	1222493.89
AHRI (vec,hb)	osp	5.00	3000	2835	1058.20
AHRI (vec,hb)	pos	5.00	600	9035	66.41
AHRI (vec,hb)	spo	5.00	2000000	1633	1224739.74
AHRI (vec,hbwa)	osp	5.00	3000	2786	1076.81
AHRI (vec,hbwa)	pos	5.00	600	5351	112.13
AHRI (vec,hbwa)	spo	5.00	2000000	1334	1499250.37
BPTree	osp	10.00	2000	1722	1161.44
BPTree	pos	10.00	300	2245	133.63
BPTree	spo	10.00	4000000	4505	887902.33
AHRI (bp)	osp	10.00	2000	1373	1456.66
AHRI (bp)	pos	10.00	300	1915	156.66
AHRI (bp)	spo	10.00	4000000	1579	2533248.89
AHRI (hb)	osp	10.00	2000	1362	1468.43
AHRI (hb)	pos	10.00	300	3434	87.36
AHRI (hb)	spo	10.00	4000000	1724	2320185.61
AHRI (hbwa)	osp	10.00	2000	1354	1477.10
AHRI (hbwa)	pos	10.00	300	1951	153.77
AHRI (hbwa)	spo	10.00	4000000	1721	2324230.10
AHRI (vec,bp)	osp	10.00	2000	988	2024.29
AHRI (vec,bp)	pos	10.00	300	1391	215.67
AHRI (vec,bp)	spo	10.00	4000000	1985	2015113.35
AHRI (vec,hb)	osp	10.00	2000	990	2020.20
AHRI (vec,hb)	pos	10.00	300	2998	100.07
AHRI (vec,hb)	spo	10.00	4000000	1998	2002002.00
AHRI (vec,hbwa)	osp	10.00	2000	989	2022.24
AHRI (vec,hbwa)	pos	10.00	300	1398	214.59
AHRI (vec,hbwa)	spo	10.00	4000000	2008	1992031.87

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
BPTree	osp	30.69	2000	6063	329.87
BPTree	pos	30.69	300	6486	46.25
BPTree	spo	30.69	4000000	5832	685871.06
AHRI (bp)	osp	30.69	2000	4577	436.97
AHRI (bp)	pos	30.69	300	5746	52.21
AHRI (bp)	spo	30.69	4000000	1867	2142474.56
AHRI (hb)	osp	30.69	2000	4566	438.02
AHRI (hb)	pos	30.69	300	9915	30.26
AHRI (hb)	spo	30.69	4000000	1890	2116402.12
AHRI (hbwa)	osp	30.69	2000	4646	430.48
AHRI (hbwa)	pos	30.69	300	5688	52.74
AHRI (hbwa)	spo	30.69	4000000	1874	2134471.72
AHRI (vec,bp)	osp	30.69	2000	4342	460.62
AHRI (vec,bp)	pos	30.69	300	4104	73.10
AHRI (vec,bp)	spo	30.69	4000000	2148	1862197.39
AHRI (vec,hb)	osp	30.69	2000	4335	461.36
AHRI (vec,hb)	pos	30.69	300	8827	33.99
AHRI (vec,hb)	spo	30.69	4000000	1850	2162162.16
AHRI (vec,hbwa)	osp	30.69	2000	4339	460.94
AHRI (vec,hbwa)	pos	30.69	300	4044	74.18
AHRI (vec,hbwa)	spo	30.69	4000000	2132	1876172.61
BPTree	osp	46.14 (DBP)	2000	1109	1803.43
BPTree	pos	46.14 (DBP)	300	2675	112.15
BPTree	spo	46.14 (DBP)	4000000	8710	459242.25
AHRI (bp)	osp	46.14 (DBP)	2000	1015	1970.44
AHRI (bp)	pos	46.14 (DBP)	300	2260	132.74
AHRI (bp)	spo	46.14 (DBP)	4000000	5175	772946.86
AHRI (hb)	osp	46.14 (DBP)	2000	1133	1765.23
AHRI (hb)	pos	46.14 (DBP)	300	3723	80.58
AHRI (hb)	spo	46.14 (DBP)	4000000	5882	680040.80
AHRI (hbwa)	osp	46.14 (DBP)	2000	882	2267.57
AHRI (hbwa)	pos	46.14 (DBP)	300	2594	115.65
AHRI (hbwa)	spo	46.14 (DBP)	4000000	5372	744601.64
AHRI (vec,bp)	osp	46.14 (DBP)	2000	911	2195.39

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (vec,bp)	pos	46.14 (DBP)	300	1544	194.30
AHRI (vec,bp)	spo	46.14 (DBP)	4000000	5301	754574.61
AHRI (vec,hb)	osp	46.14 (DBP)	2000	995	2010.05
AHRI (vec,hb)	pos	46.14 (DBP)	300	3165	94.79
AHRI (vec,hb)	spo	46.14 (DBP)	4000000	6166	648718.78
AHRI (vec,hbwa)	osp	46.14 (DBP)	2000	675	2962.96
AHRI (vec,hbwa)	pos	46.14 (DBP)	300	1514	198.15
AHRI (vec,hbwa)	spo	46.14 (DBP)	4000000	5390	742115.03
BPTree	osp	65.04	2000	10962	182.45
BPTree	pos	65.04	300	15062	19.92
BPTree	spo	65.04	4000000	6554	610314.31
AHRI (bp)	osp	65.04	2000	7766	257.53
AHRI (bp)	pos	65.04	300	12523	23.96
AHRI (bp)	spo	65.04	4000000	2573	1554605.52
AHRI (hb)	osp	65.04	2000	7663	260.99
AHRI (hb)	pos	65.04	300	22242	13.49
AHRI (hb)	spo	65.04	4000000	2705	1478743.07
AHRI (hbwa)	osp	65.04	2000	7716	259.20
AHRI (hbwa)	pos	65.04	300	11245	26.68
AHRI (hbwa)	spo	65.04	4000000	2693	1485332.34
AHRI (vec,bp)	osp	65.04	2000	8332	240.04
AHRI (vec,bp)	pos	65.04	300	10411	28.82
AHRI (vec,bp)	spo	65.04	4000000	2828	1414427.16
AHRI (vec,hb)	osp	65.04	2000	8286	241.37
AHRI (vec,hb)	pos	65.04	300	19033	15.76
AHRI (vec,hb)	spo	65.04	4000000	2833	1411930.82
AHRI (vec,hbwa)	osp	65.04	2000	8329	240.12
AHRI (vec,hbwa)	pos	65.04	300	10297	29.13
AHRI (vec,hbwa)	spo	65.04	4000000	2806	1425516.75
BPTree	osp	95.47 (DBP)	2000	1330	1503.76
BPTree	pos	95.47 (DBP)	300	6760	44.38
BPTree	spo	95.47 (DBP)	4000000	9825	407124.68
AHRI (bp)	osp	95.47 (DBP)	2000	1063	1881.47
AHRI (bp)	pos	95.47 (DBP)	300	6246	48.03

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (bp)	spo	95.47 (DBP)	4000000	5808	688705.23
AHRI (hb)	osp	95.47 (DBP)	2000	1075	1860.47
AHRI (hb)	pos	95.47 (DBP)	300	11184	26.82
AHRI (hb)	spo	95.47 (DBP)	4000000	6250	640000.00
AHRI (hbwa)	osp	95.47 (DBP)	2000	1012	1976.28
AHRI (hbwa)	pos	95.47 (DBP)	300	6978	42.99
AHRI (hbwa)	spo	95.47 (DBP)	4000000	5955	671704.45
AHRI (vec,bp)	osp	95.47 (DBP)	2000	1019	1962.71
AHRI (vec,bp)	pos	95.47 (DBP)	300	4771	62.88
AHRI (vec,bp)	spo	95.47 (DBP)	4000000	5936	673854.45
AHRI (vec,hb)	osp	95.47 (DBP)	2000	1041	1921.23
AHRI (vec,hb)	pos	95.47 (DBP)	300	9000	33.33
AHRI (vec,hb)	spo	95.47 (DBP)	4000000	6410	624024.96
AHRI (vec,hbwa)	osp	95.47 (DBP)	2000	1024	1953.12
AHRI (vec,hbwa)	pos	95.47 (DBP)	300	4940	60.73
AHRI (vec,hbwa)	spo	95.47 (DBP)	4000000	5983	668560.92
BPTree	osp	100.00	2000	17020	117.51
BPTree	pos	100.00	300	22991	13.05
BPTree	spo	100.00	4000000	6946	575871.00
AHRI (bp)	osp	100.00	2000	12346	162.00
AHRI (bp)	pos	100.00	300	19340	15.51
AHRI (bp)	spo	100.00	4000000	2837	1409940.08
AHRI (hb)	osp	100.00	2000	12348	161.97
AHRI (hb)	pos	100.00	300	32138	9.33
AHRI (hb)	spo	100.00	4000000	2835	1410934.74
AHRI (hbwa)	osp	100.00	2000	12333	162.17
AHRI (hbwa)	pos	100.00	300	19306	15.54
AHRI (hbwa)	spo	100.00	4000000	2851	1403016.49
AHRI (vec,bp)	osp	100.00	2000	13393	149.33
AHRI (vec,bp)	pos	100.00	300	13885	21.61
AHRI (vec,bp)	spo	100.00	4000000	2954	1354096.14
AHRI (vec,hb)	osp	100.00	2000	14163	141.21
AHRI (vec,hb)	pos	100.00	300	28949	10.36
AHRI (vec,hb)	spo	100.00	4000000	2821	1417936.90

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (vec,hbwa)	osp	100.00	2000	13273	150.68
AHRI (vec,hbwa)	pos	100.00	300	15607	19.22
AHRI (vec,hbwa)	spo	100.00	4000000	2943	1359157.32
BPTree	osp	157.84	2000	31311	63.88
BPTree	pos	157.84	300	39364	7.62
BPTree	spo	157.84	4000000	7414	539519.83
AHRI (bp)	osp	157.84	2000	23145	86.41
AHRI (bp)	pos	157.84	300	32341	9.28
AHRI (bp)	spo	157.84	4000000	2755	1451905.63
AHRI (hb)	osp	157.84	2000	20400	98.04
AHRI (hb)	pos	157.84	300	54046	5.55
AHRI (hb)	spo	157.84	4000000	2804	1426533.52
AHRI (hbwa)	osp	157.84	2000	20420	97.94
AHRI (hbwa)	pos	157.84	300	31922	9.40
AHRI (hbwa)	spo	157.84	4000000	2760	1449275.36
AHRI (vec,bp)	osp	157.84	2000	23489	85.15
AHRI (vec,bp)	pos	157.84	300	26468	11.33
AHRI (vec,bp)	spo	157.84	4000000	2910	1374570.45
AHRI (vec,hb)	osp	157.84	2000	23502	85.10
AHRI (vec,hb)	pos	157.84	300	48742	6.15
AHRI (vec,hb)	spo	157.84	4000000	3072	1302083.33
AHRI (vec,hbwa)	osp	157.84	2000	22125	90.40
AHRI (vec,hbwa)	pos	157.84	300	25986	11.54
AHRI (vec,hbwa)	spo	157.84	4000000	2918	1370801.92
BPTree	osp	189.57 (DBP)	2000	817	2447.98
BPTree	pos	189.57 (DBP)	300	148098	2.03
BPTree	spo	189.57 (DBP)	4000000	14241	280879.15
AHRI (bp)	osp	189.57 (DBP)	2000	616	3246.75
AHRI (bp)	pos	189.57 (DBP)	300	111043	2.70
AHRI (bp)	spo	189.57 (DBP)	4000000	9802	408079.98
AHRI (hb)	osp	189.57 (DBP)	2000	636	3144.65
AHRI (hb)	pos	189.57 (DBP)	300	190195	1.58
AHRI (hb)	spo	189.57 (DBP)	4000000	12592	317662.01
AHRI (hbwa)	osp	189.57 (DBP)	2000	547	3656.31

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (hbwa)	pos	189.57 (DBP)	300	138117	2.17
AHRI (hbwa)	spo	189.57 (DBP)	4000000	10179	392965.91
AHRI (vec,bp)	osp	189.57 (DBP)	2000	435	4597.70
AHRI (vec,bp)	pos	189.57 (DBP)	300	90015	3.33
AHRI (vec,bp)	spo	189.57 (DBP)	4000000	8967	446080.07
AHRI (vec,hb)	osp	189.57 (DBP)	2000	603	3316.75
AHRI (vec,hb)	pos	189.57 (DBP)	300	175997	1.70
AHRI (vec,hb)	spo	189.57 (DBP)	4000000	12161	328920.32
AHRI (vec,hbwa)	osp	189.57 (DBP)	2000	586	3412.97
AHRI (vec,hbwa)	pos	189.57 (DBP)	300	90657	3.31
AHRI (vec,hbwa)	spo	189.57 (DBP)	4000000	9123	438452.26
BPTree	osp	350.56	2000	71195	28.09
BPTree	pos	350.56	300	84671	3.54
BPTree	spo	350.56	4000000	7777	514337.15
AHRI (bp)	osp	350.56	2000	44390	45.06
AHRI (bp)	pos	350.56	300	68191	4.40
AHRI (bp)	spo	350.56	4000000	3109	1286587.33
AHRI (hb)	osp	350.56	2000	44445	45.00
AHRI (hb)	pos	350.56	300	118479	2.53
AHRI (hb)	spo	350.56	4000000	3113	1284934.15
AHRI (hbwa)	osp	350.56	2000	44421	45.02
AHRI (hbwa)	pos	350.56	300	67441	4.45
AHRI (hbwa)	spo	350.56	4000000	3130	1277955.27
AHRI (vec,bp)	osp	350.56	2000	51194	39.07
AHRI (vec,bp)	pos	350.56	300	49219	6.10
AHRI (vec,bp)	spo	350.56	4000000	3154	1268230.82
AHRI (vec,hb)	osp	350.56	2000	47964	41.70
AHRI (vec,hb)	pos	350.56	300	105132	2.85
AHRI (vec,hb)	spo	350.56	4000000	3271	1222867.62
AHRI (vec,hbwa)	osp	350.56	2000	47961	41.70
AHRI (vec,hbwa)	pos	350.56	300	48413	6.20
AHRI (vec,hbwa)	spo	350.56	4000000	3153	1268633.05

Table D.2: Restriction by one attribute

D.1.3 Restriction by Two Attributes

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
Bitmap	osp	5.00	10000	2684	3725.78
Bitmap	pos	5.00	300	4404	68.12
Bitmap	spo	5.00	10000	10551	947.78
BPTree	osp	5.00	2000000	4075	490797.55
BPTree	pos	5.00	3000	5025	597.01
BPTree	spo	5.00	2000000	4733	422564.97
BST	osp	5.00	2000000	5848	341997.26
BST	pos	5.00	3000	9570	313.48
BST	spo	5.00	2000000	6878	290782.20
BTree	osp	5.00	2000000	5301	377287.30
BTree	pos	5.00	3000	6349	472.52
BTree	spo	5.00	2000000	4866	411015.21
BTreeRef	osp	5.00	2000000	5820	343642.61
BTreeRef	pos	5.00	3000	6418	467.44
BTreeRef	spo	5.00	2000000	7146	279876.85
AHRI (bp)	osp	5.00	2000000	1086	1841620.63
AHRI (bp)	pos	5.00	3000	2121	1414.43
AHRI (bp)	spo	5.00	2000000	1090	1834862.39
AHRI (hb)	osp	5.00	2000000	1389	1439884.81
AHRI (hb)	pos	5.00	3000	4294	698.65
AHRI (hb)	spo	5.00	2000000	1358	1472754.05
AHRI (hbwa)	osp	5.00	2000000	1347	1484780.99
AHRI (hbwa)	pos	5.00	3000	2282	1314.64
AHRI (hbwa)	spo	5.00	2000000	1359	1471670.35
Hash	osp	5.00	6000	16293	368.26
Hash	pos	5.00	2000	46834	42.70
Hash	spo	5.00	2000000	4718	423908.44
AHRI (vec,bp)	osp	5.00	2000000	1377	1452432.82
AHRI (vec,bp)	pos	5.00	3000	2174	1379.94
AHRI (vec,bp)	spo	5.00	2000000	1378	1451378.81
AHRI (vec,hb)	osp	5.00	2000000	1367	1463057.79
AHRI (vec,hb)	pos	5.00	3000	4323	693.96
AHRI (vec,hb)	spo	5.00	2000000	1390	1438848.92
AHRI (vec,hbwa)	osp	5.00	2000000	1132	1766784.45

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (vec,hbwa)	pos	5.00	3000	2186	1372.37
AHRI (vec,hbwa)	spo	5.00	2000000	1124	1779359.43
BPTree	osp	10.00	4000000	3697	1081958.34
BPTree	pos	10.00	2000	1286	1555.21
BPTree	spo	10.00	4000000	3801	1052354.64
AHRI (bp)	osp	10.00	4000000	1417	2822865.21
AHRI (bp)	pos	10.00	2000	1115	1793.72
AHRI (bp)	spo	10.00	4000000	1307	3060443.76
AHRI (hb)	osp	10.00	4000000	1444	2770083.10
AHRI (hb)	pos	10.00	2000	1672	1196.17
AHRI (hb)	spo	10.00	4000000	1346	2971768.20
AHRI (hbwa)	osp	10.00	4000000	1448	2762430.94
AHRI (hbwa)	pos	10.00	2000	1167	1713.80
AHRI (hbwa)	spo	10.00	4000000	1363	2934702.86
AHRI (vec,bp)	osp	10.00	4000000	1489	2686366.69
AHRI (vec,bp)	pos	10.00	2000	700	2857.14
AHRI (vec,bp)	spo	10.00	4000000	1331	3005259.20
AHRI (vec,hb)	osp	10.00	4000000	1517	2636783.12
AHRI (vec,hb)	pos	10.00	2000	1659	1205.55
AHRI (vec,hb)	spo	10.00	4000000	1333	3000750.19
AHRI (vec,hbwa)	osp	10.00	4000000	1546	2587322.12
AHRI (vec,hbwa)	pos	10.00	2000	686	2915.45
AHRI (vec,hbwa)	spo	10.00	4000000	1335	2996254.68
BPTree	osp	30.69	4000000	5080	787401.57
BPTree	pos	30.69	2000	3924	509.68
BPTree	spo	30.69	4000000	5064	789889.42
AHRI (bp)	osp	30.69	4000000	1694	2361275.09
AHRI (bp)	pos	30.69	2000	2877	695.17
AHRI (bp)	spo	30.69	4000000	1479	2704530.09
AHRI (hb)	osp	30.69	4000000	1688	2369668.25
AHRI (hb)	pos	30.69	2000	5309	376.72
AHRI (hb)	spo	30.69	4000000	1532	2610966.06
AHRI (hbwa)	osp	30.69	4000000	1699	2354326.07
AHRI (hbwa)	pos	30.69	2000	2635	759.01

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (hbwa)	spo	30.69	4000000	1479	2704530.09
AHRI (vec,bp)	osp	30.69	4000000	1800	2222222.22
AHRI (vec,bp)	pos	30.69	2000	2232	896.06
AHRI (vec,bp)	spo	30.69	4000000	1459	2741603.84
AHRI (vec,hb)	osp	30.69	4000000	1850	2162162.16
AHRI (vec,hb)	pos	30.69	2000	5228	382.56
AHRI (vec,hb)	spo	30.69	4000000	1455	2749140.89
AHRI (vec,hbwa)	osp	30.69	4000000	1825	2191780.82
AHRI (vec,hbwa)	pos	30.69	2000	2134	937.21
AHRI (vec,hbwa)	spo	30.69	4000000	1586	2522068.10
BPTree	osp	46.14 (DBP)	4000000	5603	713903.27
BPTree	pos	46.14 (DBP)	2000	514	3891.05
BPTree	spo	46.14 (DBP)	4000000	5957	671478.93
AHRI (bp)	osp	46.14 (DBP)	4000000	2414	1657000.83
AHRI (bp)	pos	46.14 (DBP)	2000	357	5602.24
AHRI (bp)	spo	46.14 (DBP)	4000000	2802	1427551.75
AHRI (hb)	osp	46.14 (DBP)	4000000	2465	1622718.05
AHRI (hb)	pos	46.14 (DBP)	2000	700	2857.14
AHRI (hb)	spo	46.14 (DBP)	4000000	3055	1309328.97
AHRI (hbwa)	osp	46.14 (DBP)	4000000	2357	1697072.55
AHRI (hbwa)	pos	46.14 (DBP)	2000	432	4629.63
AHRI (hbwa)	spo	46.14 (DBP)	4000000	2806	1425516.75
AHRI (vec,bp)	osp	46.14 (DBP)	4000000	2429	1646768.22
AHRI (vec,bp)	pos	46.14 (DBP)	2000	294	6802.72
AHRI (vec,bp)	spo	46.14 (DBP)	4000000	2932	1364256.48
AHRI (vec,hb)	osp	46.14 (DBP)	4000000	2491	1605780.81
AHRI (vec,hb)	pos	46.14 (DBP)	2000	691	2894.36
AHRI (vec,hb)	spo	46.14 (DBP)	4000000	3119	1282462.33
AHRI (vec,hbwa)	osp	46.14 (DBP)	4000000	2367	1689902.83
AHRI (vec,hbwa)	pos	46.14 (DBP)	2000	281	7117.44
AHRI (vec,hbwa)	spo	46.14 (DBP)	4000000	2918	1370801.92
BPTree	osp	65.04	4000000	6055	660611.07
BPTree	pos	65.04	2000	8894	224.87
BPTree	spo	65.04	4000000	5899	678081.03

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (bp)	osp	65.04	4000000	2694	1484780.99
AHRI (bp)	pos	65.04	2000	6614	302.39
AHRI (bp)	spo	65.04	4000000	2254	1774622.89
AHRI (hb)	osp	65.04	4000000	2697	1483129.40
AHRI (hb)	pos	65.04	2000	12710	157.36
AHRI (hb)	spo	65.04	4000000	2308	1733102.25
AHRI (hbwa)	osp	65.04	4000000	2688	1488095.24
AHRI (hbwa)	pos	65.04	2000	7431	269.14
AHRI (hbwa)	spo	65.04	4000000	2310	1731601.73
AHRI (vec,bp)	osp	65.04	4000000	2747	1456133.96
AHRI (vec,bp)	pos	65.04	2000	6162	324.57
AHRI (vec,bp)	spo	65.04	4000000	2310	1731601.73
AHRI (vec,hb)	osp	65.04	4000000	2787	1435235.02
AHRI (vec,hb)	pos	65.04	2000	12514	159.82
AHRI (vec,hb)	spo	65.04	4000000	2284	1751313.49
AHRI (vec,hbwa)	osp	65.04	4000000	2751	1454016.72
AHRI (vec,hbwa)	pos	65.04	2000	5863	341.12
AHRI (vec,hbwa)	spo	65.04	4000000	2300	1739130.43
BPTree	osp	95.47 (DBP)	4000000	6324	632511.07
BPTree	pos	95.47 (DBP)	2000	891	2244.67
BPTree	spo	95.47 (DBP)	4000000	6551	610593.80
AHRI (bp)	osp	95.47 (DBP)	4000000	3001	1332889.04
AHRI (bp)	pos	95.47 (DBP)	2000	725	2758.62
AHRI (bp)	spo	95.47 (DBP)	4000000	3477	1150417.03
AHRI (hb)	osp	95.47 (DBP)	4000000	3003	1332001.33
AHRI (hb)	pos	95.47 (DBP)	2000	1237	1616.81
AHRI (hb)	spo	95.47 (DBP)	4000000	3645	1097393.69
AHRI (hbwa)	osp	95.47 (DBP)	4000000	3004	1331557.92
AHRI (hbwa)	pos	95.47 (DBP)	2000	603	3316.75
AHRI (hbwa)	spo	95.47 (DBP)	4000000	3538	1130582.25
AHRI (vec,bp)	osp	95.47 (DBP)	4000000	3097	1291572.49
AHRI (vec,bp)	pos	95.47 (DBP)	2000	492	4065.04
AHRI (vec,bp)	spo	95.47 (DBP)	4000000	3641	1098599.29
AHRI (vec,hb)	osp	95.47 (DBP)	4000000	3076	1300390.12

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (vec,hb)	pos	95.47 (DBP)	2000	1187	1684.92
AHRI (vec,hb)	spo	95.47 (DBP)	4000000	3757	1064679.27
AHRI (vec,hbwa)	osp	95.47 (DBP)	4000000	3103	1289075.09
AHRI (vec,hbwa)	pos	95.47 (DBP)	2000	468	4273.50
AHRI (vec,hbwa)	spo	95.47 (DBP)	4000000	3648	1096491.23
BPTree	osp	100.00	4000000	6433	621793.88
BPTree	pos	100.00	2000	13223	151.25
BPTree	spo	100.00	4000000	6305	634417.13
AHRI (bp)	osp	100.00	4000000	2917	1371271.85
AHRI (bp)	pos	100.00	2000	9598	208.38
AHRI (bp)	spo	100.00	4000000	2443	1637331.15
AHRI (hb)	osp	100.00	4000000	2928	1366120.22
AHRI (hb)	pos	100.00	2000	17728	112.82
AHRI (hb)	spo	100.00	4000000	2437	1641362.33
AHRI (hbwa)	osp	100.00	4000000	2922	1368925.39
AHRI (hbwa)	pos	100.00	2000	8900	224.72
AHRI (hbwa)	spo	100.00	4000000	2454	1629991.85
AHRI (vec,bp)	osp	100.00	4000000	2979	1342732.46
AHRI (vec,bp)	pos	100.00	2000	7508	266.38
AHRI (vec,bp)	spo	100.00	4000000	2424	1650165.02
AHRI (vec,hb)	osp	100.00	4000000	3019	1324942.03
AHRI (vec,hb)	pos	100.00	2000	17882	111.84
AHRI (vec,hb)	spo	100.00	4000000	2412	1658374.79
AHRI (vec,hbwa)	osp	100.00	4000000	3014	1327140.01
AHRI (vec,hbwa)	pos	100.00	2000	8351	239.49
AHRI (vec,hbwa)	spo	100.00	4000000	2416	1655629.14
BPTree	osp	157.84	4000000	6641	602318.93
BPTree	pos	157.84	2000	20705	96.60
BPTree	spo	157.84	4000000	6638	602591.14
AHRI (bp)	osp	157.84	4000000	3103	1289075.09
AHRI (bp)	pos	157.84	2000	14604	136.95
AHRI (bp)	spo	157.84	4000000	2500	1600000.00
AHRI (hb)	osp	157.84	4000000	3121	1281640.50
AHRI (hb)	pos	157.84	2000	25988	76.96

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (hb)	spo	157.84	4000000	2536	1577287.07
AHRI (hbwa)	osp	157.84	4000000	3125	1280000.00
AHRI (hbwa)	pos	157.84	2000	13503	148.12
AHRI (hbwa)	spo	157.84	4000000	2512	1592356.69
AHRI (vec,bp)	osp	157.84	4000000	3195	1251956.18
AHRI (vec,bp)	pos	157.84	2000	13367	149.62
AHRI (vec,bp)	spo	157.84	4000000	2494	1603849.24
AHRI (vec,hb)	osp	157.84	4000000	3193	1252740.37
AHRI (vec,hb)	pos	157.84	2000	26042	76.80
AHRI (vec,hb)	spo	157.84	4000000	2535	1577909.27
AHRI (vec,hbwa)	osp	157.84	4000000	3174	1260239.45
AHRI (vec,hbwa)	pos	157.84	2000	12941	154.55
AHRI (vec,hbwa)	spo	157.84	4000000	2501	1599360.26
BPTree	osp	189.57 (DBP)	4000000	6796	588581.52
BPTree	pos	189.57 (DBP)	2000	527	3795.07
BPTree	spo	189.57 (DBP)	4000000	10259	389901.55
AHRI (bp)	osp	189.57 (DBP)	4000000	3461	1155735.34
AHRI (bp)	pos	189.57 (DBP)	2000	319	6269.59
AHRI (bp)	spo	189.57 (DBP)	4000000	6040	662251.66
AHRI (hb)	osp	189.57 (DBP)	4000000	3460	1156069.36
AHRI (hb)	pos	189.57 (DBP)	2000	668	2994.01
AHRI (hb)	spo	189.57 (DBP)	4000000	7929	504477.24
AHRI (hbwa)	osp	189.57 (DBP)	4000000	3464	1154734.41
AHRI (hbwa)	pos	189.57 (DBP)	2000	418	4784.69
AHRI (hbwa)	spo	189.57 (DBP)	4000000	6501	615289.96
AHRI (vec,bp)	osp	189.57 (DBP)	4000000	3492	1145475.37
AHRI (vec,bp)	pos	189.57 (DBP)	2000	288	6944.44
AHRI (vec,bp)	spo	189.57 (DBP)	4000000	5947	672608.04
AHRI (vec,hb)	osp	189.57 (DBP)	4000000	3531	1132823.56
AHRI (vec,hb)	pos	189.57 (DBP)	2000	667	2998.50
AHRI (vec,hb)	spo	189.57 (DBP)	4000000	8102	493705.26
AHRI (vec,hbwa)	osp	189.57 (DBP)	4000000	3554	1125492.40
AHRI (vec,hbwa)	pos	189.57 (DBP)	2000	273	7326.01
AHRI (vec,hbwa)	spo	189.57 (DBP)	4000000	5965	670578.37

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
BPTree	osp	350.56	4000000	7223	553786.52
BPTree	pos	350.56	2000	47821	41.82
BPTree	spo	350.56	4000000	7134	560695.26
AHRI (bp)	osp	350.56	4000000	3398	1177163.04
AHRI (bp)	pos	350.56	2000	31121	64.27
AHRI (bp)	spo	350.56	4000000	2744	1457725.95
AHRI (hb)	osp	350.56	4000000	3403	1175433.44
AHRI (hb)	pos	350.56	2000	56159	35.61
AHRI (hb)	spo	350.56	4000000	2732	1464128.84
AHRI (hbwa)	osp	350.56	4000000	3439	1163128.82
AHRI (hbwa)	pos	350.56	2000	28728	69.62
AHRI (hbwa)	spo	350.56	4000000	2733	1463593.12
AHRI (vec,bp)	osp	350.56	4000000	3504	1141552.51
AHRI (vec,bp)	pos	350.56	2000	24363	82.09
AHRI (vec,bp)	spo	350.56	4000000	2695	1484230.06
AHRI (vec,hb)	osp	350.56	4000000	3521	1136040.90
AHRI (vec,hb)	pos	350.56	2000	55638	35.95
AHRI (vec,hb)	spo	350.56	4000000	2724	1468428.78
AHRI (vec,hbwa)	osp	350.56	4000000	3533	1132182.28
AHRI (vec,hbwa)	pos	350.56	2000	23085	86.64
AHRI (vec,hbwa)	spo	350.56	4000000	2688	1488095.24

Table D.3: Restriction by two attributes

D.1.4 Restriction by Three Attributes

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
Bitmap	osp	5.00	10000	12963	771.43
Bitmap	pos	5.00	10000	31867	313.80
Bitmap	spo	5.00	10000	13460	742.94
BPTree	osp	5.00	2000000	4695	425985.09
BPTree	pos	5.00	2000000	5086	393236.34
BPTree	spo	5.00	2000000	4562	438404.21
BST	osp	5.00	2000000	5335	374882.85

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
BST	pos	5.00	2000000	6124	326583.93
BST	spo	5.00	2000000	6354	314762.35
BTree	osp	5.00	2000000	5282	378644.45
BTree	pos	5.00	2000000	5432	368188.51
BTree	spo	5.00	2000000	5143	388878.09
BTreeRef	osp	5.00	2000000	6763	295726.75
BTreeRef	pos	5.00	2000000	6792	294464.08
BTreeRef	spo	5.00	2000000	6830	292825.77
AHRI (bp)	osp	5.00	2000000	1083	1846722.07
AHRI (bp)	pos	5.00	2000000	1956	1022494.89
AHRI (bp)	spo	5.00	2000000	991	2018163.47
AHRI (hb)	osp	5.00	2000000	1418	1410437.24
AHRI (hb)	pos	5.00	2000000	1174	1703577.51
AHRI (hb)	spo	5.00	2000000	1255	1593625.50
AHRI (hbwa)	osp	5.00	2000000	1350	1481481.48
AHRI (hbwa)	pos	5.00	2000000	1017	1966568.34
AHRI (hbwa)	spo	5.00	2000000	1257	1591089.90
Hash	osp	5.00	2000000	2159	926354.79
Hash	pos	5.00	2000000	1315	1520912.55
Hash	spo	5.00	2000000	2330	858369.10
AHRI (vec,bp)	osp	5.00	2000000	1421	1407459.54
AHRI (vec,bp)	pos	5.00	2000000	1963	1018848.70
AHRI (vec,bp)	spo	5.00	2000000	1280	1562500.00
AHRI (vec,hb)	osp	5.00	2000000	1404	1424501.42
AHRI (vec,hb)	pos	5.00	2000000	1153	1734605.38
AHRI (vec,hb)	spo	5.00	2000000	1292	1547987.62
AHRI (vec,hbwa)	osp	5.00	2000000	1123	1780943.90
AHRI (vec,hbwa)	pos	5.00	2000000	991	2018163.47
AHRI (vec,hbwa)	spo	5.00	2000000	1035	1932367.15
BPTree	osp	10.00	4000000	3685	1085481.68
BPTree	pos	10.00	4000000	3946	1013684.74
BPTree	spo	10.00	4000000	3710	1078167.12
AHRI (bp)	osp	10.00	4000000	1433	2791346.82
AHRI (bp)	pos	10.00	4000000	2070	1932367.15

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (bp)	spo	10.00	4000000	1261	3172085.65
AHRI (hb)	osp	10.00	4000000	1451	2756719.50
AHRI (hb)	pos	10.00	4000000	1072	3731343.28
AHRI (hb)	spo	10.00	4000000	1284	3115264.80
AHRI (hbwa)	osp	10.00	4000000	1447	2764340.01
AHRI (hbwa)	pos	10.00	4000000	1082	3696857.67
AHRI (hbwa)	spo	10.00	4000000	1281	3122560.50
AHRI (vec,bp)	osp	10.00	4000000	1503	2661343.98
AHRI (vec,bp)	pos	10.00	4000000	2143	1866542.23
AHRI (vec,bp)	spo	10.00	4000000	1284	3115264.80
AHRI (vec,hb)	osp	10.00	4000000	1525	2622950.82
AHRI (vec,hb)	pos	10.00	4000000	1101	3633060.85
AHRI (vec,hb)	spo	10.00	4000000	1284	3115264.80
AHRI (vec,hbwa)	osp	10.00	4000000	1549	2582311.17
AHRI (vec,hbwa)	pos	10.00	4000000	1129	3542958.37
AHRI (vec,hbwa)	spo	10.00	4000000	1290	3100775.19
BPTree	osp	30.69	4000000	5162	774893.45
BPTree	pos	30.69	4000000	5277	758006.44
BPTree	spo	30.69	4000000	4827	828672.05
AHRI (bp)	osp	30.69	4000000	1712	2336448.60
AHRI (bp)	pos	30.69	4000000	2554	1566170.71
AHRI (bp)	spo	30.69	4000000	1419	2818886.54
AHRI (hb)	osp	30.69	4000000	1708	2341920.37
AHRI (hb)	pos	30.69	4000000	1151	3475238.92
AHRI (hb)	spo	30.69	4000000	1468	2724795.64
AHRI (hbwa)	osp	30.69	4000000	1718	2328288.71
AHRI (hbwa)	pos	30.69	4000000	1227	3259983.70
AHRI (hbwa)	spo	30.69	4000000	1407	2842928.22
AHRI (vec,bp)	osp	30.69	4000000	1806	2214839.42
AHRI (vec,bp)	pos	30.69	4000000	2622	1525553.01
AHRI (vec,bp)	spo	30.69	4000000	1409	2838892.83
AHRI (vec,hb)	osp	30.69	4000000	1849	2163331.53
AHRI (vec,hb)	pos	30.69	4000000	1195	3347280.33
AHRI (vec,hb)	spo	30.69	4000000	1375	2909090.91

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (vec,hbwa)	osp	30.69	4000000	1828	2188183.81
AHRI (vec,hbwa)	pos	30.69	4000000	1261	3172085.65
AHRI (vec,hbwa)	spo	30.69	4000000	1545	2588996.76
BPTree	osp	46.14 (DBP)	4000000	5539	722152.01
BPTree	pos	46.14 (DBP)	4000000	5578	717102.90
BPTree	spo	46.14 (DBP)	4000000	5462	732332.48
AHRI (bp)	osp	46.14 (DBP)	4000000	2443	1637331.15
AHRI (bp)	pos	46.14 (DBP)	4000000	3248	1231527.09
AHRI (bp)	spo	46.14 (DBP)	4000000	2526	1583531.27
AHRI (hb)	osp	46.14 (DBP)	4000000	2359	1695633.74
AHRI (hb)	pos	46.14 (DBP)	4000000	2177	1837390.90
AHRI (hb)	spo	46.14 (DBP)	4000000	2541	1574183.39
AHRI (hbwa)	osp	46.14 (DBP)	4000000	2357	1697072.55
AHRI (hbwa)	pos	46.14 (DBP)	4000000	2264	1766784.45
AHRI (hbwa)	spo	46.14 (DBP)	4000000	2491	1605780.81
AHRI (vec,bp)	osp	46.14 (DBP)	4000000	2385	1677148.85
AHRI (vec,bp)	pos	46.14 (DBP)	4000000	3260	1226993.87
AHRI (vec,bp)	spo	46.14 (DBP)	4000000	2592	1543209.88
AHRI (vec,hb)	osp	46.14 (DBP)	4000000	2365	1691331.92
AHRI (vec,hb)	pos	46.14 (DBP)	4000000	2298	1740644.04
AHRI (vec,hb)	spo	46.14 (DBP)	4000000	2539	1575423.40
AHRI (vec,hbwa)	osp	46.14 (DBP)	4000000	2319	1724881.41
AHRI (vec,hbwa)	pos	46.14 (DBP)	4000000	2302	1737619.46
AHRI (vec,hbwa)	spo	46.14 (DBP)	4000000	2578	1551590.38
BPTree	osp	65.04	4000000	6075	658436.21
BPTree	pos	65.04	4000000	6317	633211.97
BPTree	spo	65.04	4000000	5831	685988.68
AHRI (bp)	osp	65.04	4000000	2713	1474382.60
AHRI (bp)	pos	65.04	4000000	3731	1072098.63
AHRI (bp)	spo	65.04	4000000	2218	1803426.51
AHRI (hb)	osp	65.04	4000000	2713	1474382.60
AHRI (hb)	pos	65.04	4000000	2052	1949317.74
AHRI (hb)	spo	65.04	4000000	2259	1770695.00
AHRI (hbwa)	osp	65.04	4000000	2727	1466813.35

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (hbwa)	pos	65.04	4000000	2132	1876172.61
AHRI (hbwa)	spo	65.04	4000000	2248	1779359.43
AHRI (vec,bp)	osp	65.04	4000000	2742	1458789.20
AHRI (vec,bp)	pos	65.04	4000000	3772	1060445.39
AHRI (vec,bp)	spo	65.04	4000000	2251	1776988.01
AHRI (vec,hb)	osp	65.04	4000000	2780	1438848.92
AHRI (vec,hb)	pos	65.04	4000000	2076	1926782.27
AHRI (vec,hb)	spo	65.04	4000000	2247	1780151.31
AHRI (vec,hbwa)	osp	65.04	4000000	2752	1453488.37
AHRI (vec,hbwa)	pos	65.04	4000000	2112	1893939.39
AHRI (vec,hbwa)	spo	65.04	4000000	2251	1776988.01
BPTree	osp	95.47 (DBP)	4000000	6252	639795.27
BPTree	pos	95.47 (DBP)	4000000	6457	619482.73
BPTree	spo	95.47 (DBP)	4000000	6256	639386.19
AHRI (bp)	osp	95.47 (DBP)	4000000	2964	1349527.67
AHRI (bp)	pos	95.47 (DBP)	4000000	3796	1053740.78
AHRI (bp)	spo	95.47 (DBP)	4000000	3281	1219140.51
AHRI (hb)	osp	95.47 (DBP)	4000000	2950	1355932.20
AHRI (hb)	pos	95.47 (DBP)	4000000	2810	1423487.54
AHRI (hb)	spo	95.47 (DBP)	4000000	3254	1229256.30
AHRI (hbwa)	osp	95.47 (DBP)	4000000	2962	1350438.89
AHRI (hbwa)	pos	95.47 (DBP)	4000000	2907	1375988.99
AHRI (hbwa)	spo	95.47 (DBP)	4000000	3272	1222493.89
AHRI (vec,bp)	osp	95.47 (DBP)	4000000	3005	1331114.81
AHRI (vec,bp)	pos	95.47 (DBP)	4000000	3751	1066382.30
AHRI (vec,bp)	spo	95.47 (DBP)	4000000	3393	1178897.73
AHRI (vec,hb)	osp	95.47 (DBP)	4000000	2991	1337345.37
AHRI (vec,hb)	pos	95.47 (DBP)	4000000	2851	1403016.49
AHRI (vec,hb)	spo	95.47 (DBP)	4000000	3338	1198322.35
AHRI (vec,hbwa)	osp	95.47 (DBP)	4000000	3021	1324064.88
AHRI (vec,hbwa)	pos	95.47 (DBP)	4000000	2857	1400070.00
AHRI (vec,hbwa)	spo	95.47 (DBP)	4000000	3346	1195457.26
BPTree	osp	100.00	4000000	6457	619482.73
BPTree	pos	100.00	4000000	6553	610407.45

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
BPTree	spo	100.00	4000000	6220	643086.82
AHRI (bp)	osp	100.00	4000000	2938	1361470.39
AHRI (bp)	pos	100.00	4000000	3937	1016002.03
AHRI (bp)	spo	100.00	4000000	2381	1679966.40
AHRI (hb)	osp	100.00	4000000	2956	1353179.97
AHRI (hb)	pos	100.00	4000000	2239	1786511.84
AHRI (hb)	spo	100.00	4000000	2387	1675743.61
AHRI (hbwa)	osp	100.00	4000000	2942	1359619.31
AHRI (hbwa)	pos	100.00	4000000	2258	1771479.19
AHRI (hbwa)	spo	100.00	4000000	2400	1666666.67
AHRI (vec,bp)	osp	100.00	4000000	2997	1334668.00
AHRI (vec,bp)	pos	100.00	4000000	4011	997257.54
AHRI (vec,bp)	spo	100.00	4000000	2389	1674340.73
AHRI (vec,hb)	osp	100.00	4000000	3011	1328462.30
AHRI (vec,hb)	pos	100.00	4000000	2214	1806684.73
AHRI (vec,hb)	spo	100.00	4000000	2390	1673640.17
AHRI (vec,hbwa)	osp	100.00	4000000	3001	1332889.04
AHRI (vec,hbwa)	pos	100.00	4000000	2243	1783325.90
AHRI (vec,hbwa)	spo	100.00	4000000	2379	1681378.73
BPTree	osp	157.84	4000000	6642	602228.24
BPTree	pos	157.84	4000000	6904	579374.28
BPTree	spo	157.84	4000000	6548	610873.55
AHRI (bp)	osp	157.84	4000000	3145	1271860.10
AHRI (bp)	pos	157.84	4000000	4163	960845.54
AHRI (bp)	spo	157.84	4000000	2442	1638001.64
AHRI (hb)	osp	157.84	4000000	3138	1274697.26
AHRI (hb)	pos	157.84	4000000	2320	1724137.93
AHRI (hb)	spo	157.84	4000000	2489	1607071.11
AHRI (hbwa)	osp	157.84	4000000	3148	1270648.03
AHRI (hbwa)	pos	157.84	4000000	2396	1669449.08
AHRI (hbwa)	spo	157.84	4000000	2459	1626677.51
AHRI (vec,bp)	osp	157.84	4000000	3180	1257861.64
AHRI (vec,bp)	pos	157.84	4000000	4243	942729.20
AHRI (vec,bp)	spo	157.84	4000000	2424	1650165.02

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (vec,hb)	osp	157.84	4000000	3197	1251172.97
AHRI (vec,hb)	pos	157.84	4000000	2353	1699957.50
AHRI (vec,hb)	spo	157.84	4000000	2488	1607717.04
AHRI (vec,hbwa)	osp	157.84	4000000	3175	1259842.52
AHRI (vec,hbwa)	pos	157.84	4000000	2386	1676445.93
AHRI (vec,hbwa)	spo	157.84	4000000	2478	1614205.00
BPTree	osp	189.57 (DBP)	4000000	6754	592241.63
BPTree	pos	189.57 (DBP)	4000000	6794	588754.78
BPTree	spo	189.57 (DBP)	4000000	6701	596925.83
AHRI (bp)	osp	189.57 (DBP)	4000000	3438	1163467.13
AHRI (bp)	pos	189.57 (DBP)	4000000	4080	980392.16
AHRI (bp)	spo	189.57 (DBP)	4000000	3897	1026430.59
AHRI (hb)	osp	189.57 (DBP)	4000000	3357	1191540.07
AHRI (hb)	pos	189.57 (DBP)	4000000	3040	1315789.47
AHRI (hb)	spo	189.57 (DBP)	4000000	3745	1068090.79
AHRI (hbwa)	osp	189.57 (DBP)	4000000	3443	1161777.52
AHRI (hbwa)	pos	189.57 (DBP)	4000000	3115	1284109.15
AHRI (hbwa)	spo	189.57 (DBP)	4000000	3777	1059041.57
AHRI (vec,bp)	osp	189.57 (DBP)	4000000	3475	1151079.14
AHRI (vec,bp)	pos	189.57 (DBP)	4000000	4157	962232.38
AHRI (vec,bp)	spo	189.57 (DBP)	4000000	3976	1006036.22
AHRI (vec,hb)	osp	189.57 (DBP)	4000000	3521	1136040.90
AHRI (vec,hb)	pos	189.57 (DBP)	4000000	3097	1291572.49
AHRI (vec,hb)	spo	189.57 (DBP)	4000000	3815	1048492.79
AHRI (vec,hbwa)	osp	189.57 (DBP)	4000000	3478	1150086.26
AHRI (vec,hbwa)	pos	189.57 (DBP)	4000000	3119	1282462.33
AHRI (vec,hbwa)	spo	189.57 (DBP)	4000000	3825	1045751.63
BPTree	osp	350.56	4000000	7224	553709.86
BPTree	pos	350.56	4000000	7476	535045.48
BPTree	spo	350.56	4000000	7056	566893.42
AHRI (bp)	osp	350.56	4000000	3405	1174743.02
AHRI (bp)	pos	350.56	4000000	4590	871459.69
AHRI (bp)	spo	350.56	4000000	2688	1488095.24
AHRI (hb)	osp	350.56	4000000	3411	1172676.63

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (hb)	pos	350.56	4000000	2600	1538461.54
AHRI (hb)	spo	350.56	4000000	2678	1493651.98
AHRI (hbwa)	osp	350.56	4000000	3442	1162115.05
AHRI (hbwa)	pos	350.56	4000000	2556	1564945.23
AHRI (hbwa)	spo	350.56	4000000	2690	1486988.85
AHRI (vec,bp)	osp	350.56	4000000	3493	1145147.44
AHRI (vec,bp)	pos	350.56	4000000	4614	866926.74
AHRI (vec,bp)	spo	350.56	4000000	2629	1521491.06
AHRI (vec,hb)	osp	350.56	4000000	3501	1142530.71
AHRI (vec,hb)	pos	350.56	4000000	2575	1553398.06
AHRI (vec,hb)	spo	350.56	4000000	2685	1489757.91
AHRI (vec,hbwa)	osp	350.56	4000000	3542	1129305.48
AHRI (vec,hbwa)	pos	350.56	4000000	2627	1522649.41
AHRI (vec,hbwa)	spo	350.56	4000000	2624	1524390.24

Table D.4: Restriction by three attributes

D.1.5 Restriction by Mixed Attributes

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
Bitmap	osp	5.00	1000	2517	397.30
Bitmap	pos	5.00	200	3921	51.01
Bitmap	spo	5.00	10000	10459	956.11
BPTree	osp	5.00	15000	3093	4849.66
BPTree	pos	5.00	1500	3679	407.72
BPTree	spo	5.00	2000000	4786	417885.50
BST	osp	5.00	15000	7159	2095.26
BST	pos	5.00	1500	7417	202.24
BST	spo	5.00	2000000	7167	279056.79
BTree	osp	5.00	15000	4178	3590.23
BTree	pos	5.00	1500	4532	330.98
BTree	spo	5.00	2000000	5557	359906.42
BTreeRef	osp	5.00	15000	3378	4440.50
BTreeRef	pos	5.00	1500	4716	318.07

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
BTreeRef	spo	5.00	2000000	6213	321905.68
AHRI (bp)	osp	5.00	15000	1721	8715.86
AHRI (bp)	pos	5.00	1500	1803	831.95
AHRI (bp)	spo	5.00	2000000	1106	1808318.26
AHRI (hb)	osp	5.00	15000	1736	8640.55
AHRI (hb)	pos	5.00	1500	3133	478.77
AHRI (hb)	spo	5.00	2000000	1291	1549186.68
AHRI (hbwa)	osp	5.00	15000	1717	8736.17
AHRI (hbwa)	pos	5.00	1500	1836	816.99
AHRI (hbwa)	spo	5.00	2000000	1381	1448225.92
Hash	osp	5.00	15000	24061	623.42
Hash	pos	5.00	1500	21021	71.36
Hash	spo	5.00	2000000	4312	463821.89
AHRI (vec,bp)	osp	5.00	15000	1538	9752.93
AHRI (vec,bp)	pos	5.00	1500	1764	850.34
AHRI (vec,bp)	spo	5.00	2000000	1382	1447178.00
AHRI (vec,hb)	osp	5.00	15000	1506	9960.16
AHRI (vec,hb)	pos	5.00	1500	3130	479.23
AHRI (vec,hb)	spo	5.00	2000000	1208	1655629.14
AHRI (vec,hbwa)	osp	5.00	15000	1514	9907.53
AHRI (vec,hbwa)	pos	5.00	1500	1821	823.72
AHRI (vec,hbwa)	spo	5.00	2000000	1124	1779359.43
BPTree	osp	10.00	15000	1545	9708.74
BPTree	pos	10.00	1000	1134	881.83
BPTree	spo	10.00	4000000	3881	1030662.20
AHRI (bp)	osp	10.00	15000	1213	12366.03
AHRI (bp)	pos	10.00	1000	965	1036.27
AHRI (bp)	spo	10.00	4000000	1339	2987303.96
AHRI (hb)	osp	10.00	15000	1203	12468.83
AHRI (hb)	pos	10.00	1000	1650	606.06
AHRI (hb)	spo	10.00	4000000	1374	2911208.15
AHRI (hbwa)	osp	10.00	15000	1206	12437.81
AHRI (hbwa)	pos	10.00	1000	992	1008.06
AHRI (hbwa)	spo	10.00	4000000	1371	2917578.41

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (vec,bp)	osp	10.00	15000	880	17045.45
AHRI (vec,bp)	pos	10.00	1000	680	1470.59
AHRI (vec,bp)	spo	10.00	4000000	1380	2898550.72
AHRI (vec,hb)	osp	10.00	15000	881	17026.11
AHRI (vec,hb)	pos	10.00	1000	1505	664.45
AHRI (vec,hb)	spo	10.00	4000000	1385	2888086.64
AHRI (vec,hbwa)	osp	10.00	15000	885	16949.15
AHRI (vec,hbwa)	pos	10.00	1000	674	1483.68
AHRI (vec,hbwa)	spo	10.00	4000000	1387	2883922.13
BPTree	osp	30.69	15000	4635	3236.25
BPTree	pos	30.69	1000	3313	301.84
BPTree	spo	30.69	4000000	5105	783545.54
AHRI (bp)	osp	30.69	15000	3499	4286.94
AHRI (bp)	pos	30.69	1000	2742	364.70
AHRI (bp)	spo	30.69	4000000	1510	2649006.62
AHRI (hb)	osp	30.69	15000	3488	4300.46
AHRI (hb)	pos	30.69	1000	4888	204.58
AHRI (hb)	spo	30.69	4000000	1557	2569043.03
AHRI (hbwa)	osp	30.69	15000	3543	4233.70
AHRI (hbwa)	pos	30.69	1000	2661	375.80
AHRI (hbwa)	spo	30.69	4000000	1515	2640264.03
AHRI (vec,bp)	osp	30.69	15000	3318	4520.80
AHRI (vec,bp)	pos	30.69	1000	2004	499.00
AHRI (vec,bp)	spo	30.69	4000000	1526	2621231.98
AHRI (vec,hb)	osp	30.69	15000	3315	4524.89
AHRI (vec,hb)	pos	30.69	1000	4493	222.57
AHRI (vec,hb)	spo	30.69	4000000	1480	2702702.70
AHRI (vec,hbwa)	osp	30.69	15000	3307	4535.83
AHRI (vec,hbwa)	pos	30.69	1000	1951	512.56
AHRI (vec,hbwa)	spo	30.69	4000000	1642	2436053.59
BPTree	osp	46.14 (DBP)	15000	824	18203.88
BPTree	pos	46.14 (DBP)	1000	787	1270.65
BPTree	spo	46.14 (DBP)	4000000	6081	657786.55
AHRI (bp)	osp	46.14 (DBP)	15000	751	19973.37

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (bp)	pos	46.14 (DBP)	1000	625	1600.00
AHRI (bp)	spo	46.14 (DBP)	4000000	2956	1353179.97
AHRI (hb)	osp	46.14 (DBP)	15000	813	18450.18
AHRI (hb)	pos	46.14 (DBP)	1000	1055	947.87
AHRI (hb)	spo	46.14 (DBP)	4000000	3163	1264622.19
AHRI (hbwa)	osp	46.14 (DBP)	15000	651	23041.47
AHRI (hbwa)	pos	46.14 (DBP)	1000	719	1390.82
AHRI (hbwa)	spo	46.14 (DBP)	4000000	2972	1345895.02
AHRI (vec,bp)	osp	46.14 (DBP)	15000	678	22123.89
AHRI (vec,bp)	pos	46.14 (DBP)	1000	442	2262.44
AHRI (vec,bp)	spo	46.14 (DBP)	4000000	3072	1302083.33
AHRI (vec,hb)	osp	46.14 (DBP)	15000	714	21008.40
AHRI (vec,hb)	pos	46.14 (DBP)	1000	934	1070.66
AHRI (vec,hb)	spo	46.14 (DBP)	4000000	3235	1236476.04
AHRI (vec,hbwa)	osp	46.14 (DBP)	15000	517	29013.54
AHRI (vec,hbwa)	pos	46.14 (DBP)	1000	433	2309.47
AHRI (vec,hbwa)	spo	46.14 (DBP)	4000000	3068	1303780.96
BPTree	osp	65.04	15000	10029	1495.66
BPTree	pos	65.04	1000	7305	136.89
BPTree	spo	65.04	4000000	5952	672043.01
AHRI (bp)	osp	65.04	15000	7090	2115.66
AHRI (bp)	pos	65.04	1000	5896	169.61
AHRI (bp)	spo	65.04	4000000	2280	1754385.96
AHRI (hb)	osp	65.04	15000	7024	2135.54
AHRI (hb)	pos	65.04	1000	10564	94.66
AHRI (hb)	spo	65.04	4000000	2338	1710863.99
AHRI (hbwa)	osp	65.04	15000	7057	2125.55
AHRI (hbwa)	pos	65.04	1000	5593	178.79
AHRI (hbwa)	spo	65.04	4000000	2342	1707941.93
AHRI (vec,bp)	osp	65.04	15000	7629	1966.18
AHRI (vec,bp)	pos	65.04	1000	5017	199.32
AHRI (vec,bp)	spo	65.04	4000000	2354	1699235.34
AHRI (vec,hb)	osp	65.04	15000	7613	1970.31
AHRI (vec,hb)	pos	65.04	1000	9411	106.26

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (vec,hb)	spo	65.04	4000000	2340	1709401.71
AHRI (vec,hbwa)	osp	65.04	15000	7640	1963.35
AHRI (vec,hbwa)	pos	65.04	1000	4896	204.25
AHRI (vec,hbwa)	spo	65.04	4000000	2342	1707941.93
BPTree	osp	95.47 (DBP)	15000	966	15527.95
BPTree	pos	95.47 (DBP)	1000	2628	380.52
BPTree	spo	95.47 (DBP)	4000000	6824	586166.47
AHRI (bp)	osp	95.47 (DBP)	15000	766	19582.25
AHRI (bp)	pos	95.47 (DBP)	1000	2317	431.59
AHRI (bp)	spo	95.47 (DBP)	4000000	3648	1096491.23
AHRI (hb)	osp	95.47 (DBP)	15000	790	18987.34
AHRI (hb)	pos	95.47 (DBP)	1000	3958	252.65
AHRI (hb)	spo	95.47 (DBP)	4000000	3738	1070090.96
AHRI (hbwa)	osp	95.47 (DBP)	15000	726	20661.16
AHRI (hbwa)	pos	95.47 (DBP)	1000	2489	401.77
AHRI (hbwa)	spo	95.47 (DBP)	4000000	3693	1083130.25
AHRI (vec,bp)	osp	95.47 (DBP)	15000	740	20270.27
AHRI (vec,bp)	pos	95.47 (DBP)	1000	1721	581.06
AHRI (vec,bp)	spo	95.47 (DBP)	4000000	3800	1052631.58
AHRI (vec,hb)	osp	95.47 (DBP)	15000	763	19659.24
AHRI (vec,hb)	pos	95.47 (DBP)	1000	3182	314.27
AHRI (vec,hb)	spo	95.47 (DBP)	4000000	3882	1030396.70
AHRI (vec,hbwa)	osp	95.47 (DBP)	15000	744	20161.29
AHRI (vec,hbwa)	pos	95.47 (DBP)	1000	1740	574.71
AHRI (vec,hbwa)	spo	95.47 (DBP)	4000000	3784	1057082.45
BPTree	osp	100.00	15000	12987	1155.00
BPTree	pos	100.00	1000	11245	88.93
BPTree	spo	100.00	4000000	6379	627057.53
AHRI (bp)	osp	100.00	15000	9540	1572.33
AHRI (bp)	pos	100.00	1000	8979	111.37
AHRI (bp)	spo	100.00	4000000	2475	1616161.62
AHRI (hb)	osp	100.00	15000	9542	1572.00
AHRI (hb)	pos	100.00	1000	15570	64.23
AHRI (hb)	spo	100.00	4000000	2476	1615508.89

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (hbwa)	osp	100.00	15000	9544	1571.67
AHRI (hbwa)	pos	100.00	1000	8810	113.51
AHRI (hbwa)	spo	100.00	4000000	2491	1605780.81
AHRI (vec,bp)	osp	100.00	15000	10247	1463.84
AHRI (vec,bp)	pos	100.00	1000	6578	152.02
AHRI (vec,bp)	spo	100.00	4000000	2485	1609657.95
AHRI (vec,hb)	osp	100.00	15000	10865	1380.58
AHRI (vec,hb)	pos	100.00	1000	14548	68.74
AHRI (vec,hb)	spo	100.00	4000000	2455	1629327.90
AHRI (vec,hbwa)	osp	100.00	15000	10180	1473.48
AHRI (vec,hbwa)	pos	100.00	1000	7362	135.83
AHRI (vec,hbwa)	spo	100.00	4000000	2484	1610305.96
BPTree	osp	157.84	15000	26156	573.48
BPTree	pos	157.84	1000	18599	53.77
BPTree	spo	157.84	4000000	6692	597728.63
AHRI (bp)	osp	157.84	15000	19486	769.78
AHRI (bp)	pos	157.84	1000	14490	69.01
AHRI (bp)	spo	157.84	4000000	2515	1590457.26
AHRI (hb)	osp	157.84	15000	17047	879.92
AHRI (hb)	pos	157.84	1000	24184	41.35
AHRI (hb)	spo	157.84	4000000	2553	1566784.18
AHRI (hbwa)	osp	157.84	15000	17052	879.66
AHRI (hbwa)	pos	157.84	1000	14113	70.86
AHRI (hbwa)	spo	157.84	4000000	2537	1576665.35
AHRI (vec,bp)	osp	157.84	15000	19543	767.54
AHRI (vec,bp)	pos	157.84	1000	12219	81.84
AHRI (vec,bp)	spo	157.84	4000000	2524	1584786.05
AHRI (vec,hb)	osp	157.84	15000	19563	766.75
AHRI (vec,hb)	pos	157.84	1000	22382	44.68
AHRI (vec,hb)	spo	157.84	4000000	2580	1550387.60
AHRI (vec,hbwa)	osp	157.84	15000	18361	816.95
AHRI (vec,hbwa)	pos	157.84	1000	11971	83.54
AHRI (vec,hbwa)	spo	157.84	4000000	2542	1573564.12
BPTree	osp	189.57 (DBP)	15000	635	23622.05

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
BPTree	pos	189.57 (DBP)	1000	56862	17.59
BPTree	spo	189.57 (DBP)	4000000	9310	429645.54
AHRI (bp)	osp	189.57 (DBP)	15000	476	31512.61
AHRI (bp)	pos	189.57 (DBP)	1000	42433	23.57
AHRI (bp)	spo	189.57 (DBP)	4000000	5580	716845.88
AHRI (hb)	osp	189.57 (DBP)	15000	485	30927.84
AHRI (hb)	pos	189.57 (DBP)	1000	73359	13.63
AHRI (hb)	spo	189.57 (DBP)	4000000	6739	593559.88
AHRI (hbwa)	osp	189.57 (DBP)	15000	427	35128.81
AHRI (hbwa)	pos	189.57 (DBP)	1000	53298	18.76
AHRI (hbwa)	spo	189.57 (DBP)	4000000	5799	689774.10
AHRI (vec,bp)	osp	189.57 (DBP)	15000	343	43731.78
AHRI (vec,bp)	pos	189.57 (DBP)	1000	34574	28.92
AHRI (vec,bp)	spo	189.57 (DBP)	4000000	5493	728199.53
AHRI (vec,hb)	osp	189.57 (DBP)	15000	464	32327.59
AHRI (vec,hb)	pos	189.57 (DBP)	1000	67765	14.76
AHRI (vec,hb)	spo	189.57 (DBP)	4000000	6829	585737.30
AHRI (vec,hbwa)	osp	189.57 (DBP)	15000	453	33112.58
AHRI (vec,hbwa)	pos	189.57 (DBP)	1000	34798	28.74
AHRI (vec,hbwa)	spo	189.57 (DBP)	4000000	5459	732734.93
BPTree	osp	350.56	15000	54773	273.86
BPTree	pos	350.56	1000	45050	22.20
BPTree	spo	350.56	4000000	7203	555324.17
AHRI (bp)	osp	350.56	15000	34272	437.68
AHRI (bp)	pos	350.56	1000	30428	32.86
AHRI (bp)	spo	350.56	4000000	2779	1439366.68
AHRI (hb)	osp	350.56	15000	34296	437.37
AHRI (hb)	pos	350.56	1000	53431	18.72
AHRI (hb)	spo	350.56	4000000	2777	1440403.31
AHRI (hbwa)	osp	350.56	15000	34302	437.29
AHRI (hbwa)	pos	350.56	1000	29467	33.94
AHRI (hbwa)	spo	350.56	4000000	2778	1439884.81
AHRI (vec,bp)	osp	350.56	15000	39331	381.38
AHRI (vec,bp)	pos	350.56	1000	22348	44.75

continued on next page

Structure	Attribute Order	Size (M)	Iterations	Runtime (ms)	Queries Per Second
AHRI (vec,bp)	spo	350.56	4000000	2727	1466813.35
AHRI (vec,hb)	osp	350.56	15000	36900	406.50
AHRI (vec,hb)	pos	350.56	1000	48993	20.41
AHRI (vec,hb)	spo	350.56	4000000	2778	1439884.81
AHRI (vec,hbwa)	osp	350.56	15000	36931	406.16
AHRI (vec,hbwa)	pos	350.56	1000	21674	46.14 (DBP)
AHRI (vec,hbwa)	spo	350.56	4000000	2724	1468428.78

TABLE D.5: Restriction by mixed attributes

D.2 Performance Counter Information

This section contains all the results for using OProfile to measure CPU events such as cache misses. For this section, the only dataset used was the 5 million triple BSBM dataset.

D.2.1 Cache

D.2.1.1 Load

Structure	Attribute Order	Iterations	L1 Accesses (x10,000)	L1 Misses (x10,000)	L2 Misses (x10,000)
Bitmap	osp	4000000	81655	2038	960
Bitmap	pos	4000000	87740	2292	1056
Bitmap	spo	4000000	87129	2118	1024
BPTree	osp	4000000	177244	4965	1673
BPTree	pos	4000000	205859	5794	1734
BPTree	spo	4000000	145776	1973	995
BST	osp	4000000	150785	5985	3432
BST	pos	4000000	160001	6194	3586
BST	spo	4000000	85321	4121	3718
BTree	osp	4000000	174034	3963	1446
BTree	pos	4000000	195188	5119	1603
BTree	spo	4000000	149238	1877	1039

continued on next page

Structure	Attribute Order	Iterations	L1 Accesses (x10,000)	L1 Misses (x10,000)	L2 Misses (x10,000)
BTreeRef	osp	4000000	203798	16172	6558
BTreeRef	pos	4000000	243735	22594	7679
BTreeRef	spo	4000000	142726	2000	929
AHRI (vec,cbp)	osp	4000000	53869	1902	931
AHRI (vec,cbp)	pos	4000000	168948	6090	1373
AHRI (vec,cbp)	spo	4000000	40676	436	168
AHRI (vec,hb)	osp	4000000	54778	1924	901
AHRI (vec,hb)	pos	4000000	52080	1649	656
AHRI (vec,hb)	spo	4000000	38578	413	159
AHRI (vec,hbwa)	osp	4000000	54081	1981	924
AHRI (vec,hbwa)	pos	4000000	56407	2000	800
AHRI (vec,hbwa)	spo	4000000	44441	459	171
Hash	osp	4000000	62187	3342	1754
Hash	pos	4000000	52712	1997	1209
Hash	spo	4000000	42415	683	407
AHRI (cbp)	osp	4000000	63510	2038	973
AHRI (cbp)	pos	4000000	168261	5969	1389
AHRI (cbp)	spo	4000000	44030	454	181
AHRI (hb)	osp	4000000	60225	2058	969
AHRI (hb)	pos	4000000	61581	1764	673
AHRI (hb)	spo	4000000	44175	438	184
AHRI (hbwa)	osp	4000000	64089	2042	983
AHRI (hbwa)	pos	4000000	75352	2300	934
AHRI (hbwa)	spo	4000000	49252	551	200

Table D.6: Cache performance counters: load

D.2.1.2 Restriction by One Attribute

Structure	Attribute Order	Iterations	L1 Accesses (x10,000)	L1 Misses (x10,000)	L2 Misses (x10,000)
Bitmap	osp	1000	149461	3234	2256
Bitmap	pos	120	117398	4092	2604
Bitmap	spo	4000000	199989	2442	1799
BPTree	osp	6000	265167	3081	678

continued on next page

Structure	Attribute Order	Iterations	L1 Accesses (x10,000)	L1 Misses (x10,000)	L2 Misses (x10,000)
BPTree	pos	1200	463970	5301	1214
BPTree	spo	4000000	183055	2676	1592
BST	osp	6000	483722	6698	992
BST	pos	1200	942348	12530	1486
BST	spo	4000000	367837	5387	3228
BTree	osp	6000	310660	3141	878
BTree	pos	1200	618005	6601	1699
BTree	spo	4000000	233145	3164	2013
BTreeRef	osp	6000	199248	7067	3272
BTreeRef	pos	1200	353629	12472	6047
BTreeRef	spo	4000000	266324	5494	3421
AHRI (vec,cbp)	osp	6000	138923	1859	148
AHRI (vec,cbp)	pos	1200	230529	3108	460
AHRI (vec,cbp)	spo	4000000	60668	479	265
AHRI (vec,hb)	osp	6000	157105	2210	159
AHRI (vec,hb)	pos	1200	403993	4336	476
AHRI (vec,hb)	spo	4000000	100007	552	296
AHRI (vec,hbwa)	osp	6000	141194	1895	135
AHRI (vec,hbwa)	pos	1200	274041	3802	560
AHRI (vec,hbwa)	spo	4000000	59899	476	261
Hash	osp	6000	321727	6745	610
Hash	pos	1200	812817	14240	1255
Hash	spo	4000000	191510	1804	858
AHRI (cbp)	osp	6000	128552	2059	174
AHRI (cbp)	pos	1200	232347	3418	439
AHRI (cbp)	spo	4000000	54163	538	331
AHRI (hb)	osp	6000	117814	1950	159
AHRI (hb)	pos	1200	391816	4237	502
AHRI (hb)	spo	4000000	58339	587	373
AHRI (hbwa)	osp	6000	131768	2124	185
AHRI (hbwa)	pos	1200	247698	3540	511
AHRI (hbwa)	spo	4000000	60906	636	398

TABLE D.7: Cache performance counters: restriction by one attribute

D.2.1.3 Restriction by Two Attributes

Structure	Attribute Order	Iterations	L1 Accesses (x10,000)	L1 Misses (x10,000)	L2 Misses (x10,000)
Bitmap	osp	20000	61258	1092	44
Bitmap	pos	600	87784	2157	1282
Bitmap	spo	20000	248147	4967	164
BPTree	osp	4000000	102638	2496	1501
BPTree	pos	6000	203520	2336	542
BPTree	spo	4000000	114541	2872	1753
BST	osp	4000000	126748	4532	3341
BST	pos	6000	388945	5292	687
BST	spo	4000000	159658	4310	3156
BTree	osp	4000000	136802	2865	1924
BTree	pos	6000	294761	2993	813
BTree	spo	4000000	121266	3039	2019
BTreeRef	osp	4000000	135560	5131	3392
BTreeRef	pos	6000	162777	5738	2571
BTreeRef	spo	4000000	140899	4835	3103
AHRI (vec,cbp)	osp	4000000	25024	691	309
AHRI (vec,cbp)	pos	6000	104748	1382	178
AHRI (vec,cbp)	spo	4000000	34537	447	265
AHRI (vec,hb)	osp	4000000	25028	720	301
AHRI (vec,hb)	pos	6000	181869	1816	137
AHRI (vec,hb)	spo	4000000	36092	437	262
AHRI (vec,hbwa)	osp	4000000	25425	689	311
AHRI (vec,hbwa)	pos	6000	109395	1481	133
AHRI (vec,hbwa)	spo	4000000	35167	457	264
Hash	osp	12000	569344	14671	1866
Hash	pos	4000	1363738	39791	8336
Hash	spo	4000000	137536	1870	977
AHRI (cbp)	osp	4000000	27258	785	331
AHRI (cbp)	pos	6000	95194	1250	153
AHRI (cbp)	spo	4000000	42773	558	350
AHRI (hb)	osp	4000000	28424	870	373
AHRI (hb)	pos	6000	203653	2032	138
AHRI (hb)	spo	4000000	44846	574	384
AHRI (hbwa)	osp	4000000	29396	859	384

continued on next page

Structure	Attribute Order	Iterations	L1 Accesses (x10,000)	L1 Misses (x10,000)	L2 Misses (x10,000)
AHRI (hbwa)	pos	6000	100068	1361	117
AHRI (hbwa)	spo	4000000	53456	737	452

TABLE D.8: Cache performance counters: restriction by two attributes

D.2.1.4 Restriction by Three Attributes

Structure	Attribute Order	Iterations	L1 Accesses (x10,000)	L1 Misses (x10,000)	L2 Misses (x10,000)
Bitmap	osp	20000	308682	6043	227
Bitmap	pos	20000	1013952	12158	2561
Bitmap	spo	20000	319682	6276	229
BPTree	osp	4000000	100976	2465	1508
BPTree	pos	4000000	118092	2728	1631
BPTree	spo	4000000	102545	2772	1749
BST	osp	4000000	122988	4464	3340
BST	pos	4000000	141424	4759	3661
BST	spo	4000000	145935	4505	3425
BTree	osp	4000000	134008	2760	1897
BTree	pos	4000000	131672	3054	2082
BTree	spo	4000000	105264	2925	2019
BTreeRef	osp	4000000	120902	4471	3013
BTreeRef	pos	4000000	127252	4360	2986
BTreeRef	spo	4000000	122286	4583	3012
AHRI (vec,cbp)	osp	4000000	25524	631	307
AHRI (vec,cbp)	pos	4000000	61468	1456	874
AHRI (vec,cbp)	spo	4000000	28030	401	270
AHRI (vec,hb)	osp	4000000	26231	660	302
AHRI (vec,hb)	pos	4000000	28117	481	191
AHRI (vec,hb)	spo	4000000	29312	401	263
AHRI (vec,hbwa)	osp	4000000	28148	708	346
AHRI (vec,hbwa)	pos	4000000	28465	538	199
AHRI (vec,hbwa)	spo	4000000	28911	391	270
Hash	osp	4000000	26106	801	451

continued on next page

Structure	Attribute Order	Iterations	L1 Accesses (x10,000)	L1 Misses (x10,000)	L2 Misses (x10,000)
Hash	pos	4000000	24002	444	261
Hash	spo	4000000	28946	865	669
AHRI (cbp)	osp	4000000	26338	705	330
AHRI (cbp)	pos	4000000	53756	1275	755
AHRI (cbp)	spo	4000000	35320	465	329
AHRI (hb)	osp	4000000	28827	799	359
AHRI (hb)	pos	4000000	27374	473	184
AHRI (hb)	spo	4000000	36922	486	365
AHRI (hbwa)	osp	4000000	27008	735	351
AHRI (hbwa)	pos	4000000	31283	618	225
AHRI (hbwa)	spo	4000000	37699	517	363

Table D.9: Cache performance counters: restriction by three attributes

D.2.1.5 Restriction by Mixed Attributes

Structure	Attribute Order	Iterations	L1 Accesses (x10,000)	L1 Misses (x10,000)	L2 Misses (x10,000)
Bitmap	osp	2000	49388	1062	530
Bitmap	pos	400	86648	2601	1588
Bitmap	spo	20000	256033	5104	189
BPTree	osp	30000	139612	1630	372
BPTree	pos	3000	154290	1791	419
BPTree	spo	4000000	116373	2782	1709
BST	osp	30000	291112	4113	546
BST	pos	3000	289999	3853	529
BST	spo	4000000	170723	4306	3143
BTree	osp	30000	168810	1727	503
BTree	pos	3000	186458	1872	523
BTree	spo	4000000	137493	3342	2184
BTreeRef	osp	30000	104624	3709	1759
BTreeRef	pos	3000	128119	4575	1704
BTreeRef	spo	4000000	143827	4751	3029
AHRI (vec,cbp)	osp	30000	99042	1363	112
AHRI (vec,cbp)	pos	3000	79101	1092	189

continued on next page

Structure	Attribute Order	Iterations	L1 Accesses (x10,000)	L1 Misses (x10,000)	L2 Misses (x10,000)
AHRI (vec,cbp)	spo	4000000	35162	457	267
AHRI (vec,hb)	osp	30000	79942	1164	88
AHRI (vec,hb)	pos	3000	118801	1273	180
AHRI (vec,hb)	spo	4000000	44288	498	300
AHRI (vec,hbwa)	osp	30000	79704	1100	98
AHRI (vec,hbwa)	pos	3000	78688	1115	163
AHRI (vec,hbwa)	spo	4000000	35792	456	267
Hash	osp	30000	794429	19569	2825
Hash	pos	3000	721939	19188	2473
Hash	spo	4000000	95303	1386	792
AHRI (cbp)	osp	30000	65297	1133	126
AHRI (cbp)	pos	3000	75662	1059	158
AHRI (cbp)	spo	4000000	39983	517	334
AHRI (hb)	osp	30000	66553	1160	116
AHRI (hb)	pos	3000	135983	1473	190
AHRI (hb)	spo	4000000	45546	589	394
AHRI (hbwa)	osp	30000	68699	1151	108
AHRI (hbwa)	pos	3000	94760	1378	211
AHRI (hbwa)	spo	4000000	41687	575	357

TABLE D.10: Cache performance counters: restriction by mixed attributes

D.2.2 TLB

D.2.2.1 Load

Structure	Attribute Order	Iterations	L1 TLB Accesses (x10,000)	L1 TLB Misses (x10,000)	L2 TLB Misses (x10,000)
Bitmap	osp	4000000	81655	6047	1448
Bitmap	pos	4000000	87740	5827	1285
Bitmap	spo	4000000	87129	5834	1373
BPTree	osp	4000000	177244	1881	588
BPTree	pos	4000000	205859	2771	1038
BPTree	spo	4000000	145776	690	504
BST	osp	4000000	150785	5000	2319

continued on next page

Structure	Attribute Order	Iterations	L1 TLB Accesses (x10,000)	L1 TLB Misses (x10,000)	L2 TLB Misses (x10,000)
BST	pos	4000000	160001	6690	3169
BST	spo	4000000	85321	700	495
BTree	osp	4000000	174034	2326	844
BTree	pos	4000000	195188	2850	1048
BTree	spo	4000000	149238	653	447
BTreeRef	osp	4000000	203798	26783	11504
BTreeRef	pos	4000000	243735	27001	11278
BTreeRef	spo	4000000	142726	10569	1037
AHRI (vec,cbp)	osp	4000000	53869	3485	758
AHRI (vec,cbp)	pos	4000000	168948	5425	1032
AHRI (vec,cbp)	spo	4000000	40676	996	153
AHRI (vec,hb)	osp	4000000	54778	3889	1015
AHRI (vec,hb)	pos	4000000	52080	4223	863
AHRI (vec,hb)	spo	4000000	38578	873	50
AHRI (vec,hbwa)	osp	4000000	54081	3431	690
AHRI (vec,hbwa)	pos	4000000	56407	4410	828
AHRI (vec,hbwa)	spo	4000000	44441	975	103
Hash	osp	4000000	62187	4504	1576
Hash	pos	4000000	52712	4620	1446
Hash	spo	4000000	42415	900	190
AHRI (cbp)	osp	4000000	63510	3804	798
AHRI (cbp)	pos	4000000	168261	6142	1302
AHRI (cbp)	spo	4000000	44030	1315	169
AHRI (hb)	osp	4000000	60225	4421	1053
AHRI (hb)	pos	4000000	61581	4939	979
AHRI (hb)	spo	4000000	44175	1563	142
AHRI (hbwa)	osp	4000000	64089	4044	940
AHRI (hbwa)	pos	4000000	75352	4443	818
AHRI (hbwa)	spo	4000000	49252	1294	182

Table D.11: TLB performance counters: load

D.2.2.2 Restriction by One Attribute

Structure	Attribute Order	Iterations	L1 TLB Accesses (x10,000)	L1 TLB Misses (x10,000)	L2 TLB Misses (x10,000)
Bitmap	osp	1000	149461	1525	1047
Bitmap	pos	120	117398	1029	765
Bitmap	spo	4000000	199989	2327	1249
BPTree	osp	6000	265167	413	361
BPTree	pos	1200	463970	1170	1089
BPTree	spo	4000000	183055	1588	770
BST	osp	6000	483722	1986	1869
BST	pos	1200	942348	3797	3566
BST	spo	4000000	367837	2664	1110
BTree	osp	6000	310660	735	555
BTree	pos	1200	618005	2315	2055
BTree	spo	4000000	233145	1781	992
BTreeRef	osp	6000	199248	1350	1079
BTreeRef	pos	1200	353629	1529	1025
BTreeRef	spo	4000000	266324	2898	1783
AHRI (vec,cbp)	osp	6000	138923	239	193
AHRI (vec,cbp)	pos	1200	230529	943	882
AHRI (vec,cbp)	spo	4000000	60668	644	336
AHRI (vec,hb)	osp	6000	157105	375	336
AHRI (vec,hb)	pos	1200	403993	1068	999
AHRI (vec,hb)	spo	4000000	100007	602	339
AHRI (vec,hbwa)	osp	6000	141194	240	195
AHRI (vec,hbwa)	pos	1200	274041	664	599
AHRI (vec,hbwa)	spo	4000000	59899	629	372
Hash	osp	6000	321727	745	620
Hash	pos	1200	812817	2353	2140
Hash	spo	4000000	191510	1570	1123
AHRI (cbp)	osp	6000	128552	223	181
AHRI (cbp)	pos	1200	232347	851	792
AHRI (cbp)	spo	4000000	54163	930	538
AHRI (hb)	osp	6000	117814	286	242
AHRI (hb)	pos	1200	391816	1033	970
AHRI (hb)	spo	4000000	58339	1001	585
AHRI (hbwa)	osp	6000	131768	264	223

continued on next page

Structure	Attribute Order	Iterations	L1 TLB Accesses (x10,000)	L1 TLB Misses (x10,000)	L2 TLB Misses (x10,000)
AHRI (hbwa)	pos	1200	247698	746	685
AHRI (hbwa)	spo	4000000	60906	1002	624

TABLE D.12: TLB performance counters: restriction by one attribute

D.2.2.3 Restriction by Two Attributes

Structure	Attribute Order	Iterations	L1 TLB Accesses (x10,000)	L1 TLB Misses (x10,000)	L2 TLB Misses (x10,000)
Bitmap	osp	20000	61258	129	61
Bitmap	pos	600	87784	714	482
Bitmap	spo	20000	248147	902	769
BPTree	osp	4000000	102638	1365	720
BPTree	pos	6000	203520	799	744
BPTree	spo	4000000	114541	1465	750
BST	osp	4000000	126748	2501	1157
BST	pos	6000	388945	906	783
BST	spo	4000000	159658	2935	1390
BTree	osp	4000000	136802	1331	670
BTree	pos	6000	294761	1338	1194
BTree	spo	4000000	121266	1498	780
BTreeRef	osp	4000000	135560	2230	1249
BTreeRef	pos	6000	162777	576	315
BTreeRef	spo	4000000	140899	3059	1998
AHRI (vec,cbp)	osp	4000000	25024	1193	517
AHRI (vec,cbp)	pos	6000	104748	358	325
AHRI (vec,cbp)	spo	4000000	34537	572	253
AHRI (vec,hb)	osp	4000000	25028	1195	537
AHRI (vec,hb)	pos	6000	181869	362	324
AHRI (vec,hb)	spo	4000000	36092	582	277
AHRI (vec,hbwa)	osp	4000000	25425	1129	486
AHRI (vec,hbwa)	pos	6000	109395	230	195
AHRI (vec,hbwa)	spo	4000000	35167	569	290
Hash	osp	12000	569344	1386	1227

continued on next page

Structure	Attribute Order	Iterations	L1 TLB Accesses (x10,000)	L1 TLB Misses (x10,000)	L2 TLB Misses (x10,000)
Hash	pos	4000	1363738	5590	5045
Hash	spo	4000000	137536	1422	952
AHRI (cbp)	osp	4000000	27258	1273	592
AHRI (cbp)	pos	6000	95194	104	67
AHRI (cbp)	spo	4000000	42773	949	528
AHRI (hb)	osp	4000000	28424	1203	612
AHRI (hb)	pos	6000	203653	477	440
AHRI (hb)	spo	4000000	44846	784	385
AHRI (hbwa)	osp	4000000	29396	1212	592
AHRI (hbwa)	pos	6000	100068	243	212
AHRI (hbwa)	spo	4000000	53456	827	439

Table D.13: TLB performance counters: restriction by two attributes

D.2.2.4 Restriction by Three Attributes

Structure	Attribute Order	Iterations	L1 TLB Accesses (x10,000)	L1 TLB Misses (x10,000)	L2 TLB Misses (x10,000)
Bitmap	osp	20000	308682	681	534
Bitmap	pos	20000	1013952	2790	2474
Bitmap	spo	20000	319682	597	447
BPTree	osp	4000000	100976	984	398
BPTree	pos	4000000	118092	1278	657
BPTree	spo	4000000	102545	1497	765
BST	osp	4000000	122988	2172	837
BST	pos	4000000	141424	2390	1184
BST	spo	4000000	145935	2909	1420
BTree	osp	4000000	134008	1734	1029
BTree	pos	4000000	131672	1758	1109
BTree	spo	4000000	105264	1257	546
BTreeRef	osp	4000000	120902	1990	1128
BTreeRef	pos	4000000	127252	1770	858
BTreeRef	spo	4000000	122286	2566	1424
AHRI (vec,cbp)	osp	4000000	25524	1105	440
AHRI (vec,cbp)	pos	4000000	61468	1523	568

continued on next page

Structure	Attribute Order	Iterations	L1 TLB Accesses (x10,000)	L1 TLB Misses (x10,000)	L2 TLB Misses (x10,000)
AHRI (vec,cbp)	spo	4000000	28030	535	243
AHRI (vec,hb)	osp	4000000	26231	1169	517
AHRI (vec,hb)	pos	4000000	28117	1153	370
AHRI (vec,hb)	spo	4000000	29312	559	269
AHRI (vec,hbwa)	osp	4000000	28148	1168	529
AHRI (vec,hbwa)	pos	4000000	28465	1008	286
AHRI (vec,hbwa)	spo	4000000	28911	576	297
Hash	osp	4000000	26106	1419	811
Hash	pos	4000000	24002	1481	537
Hash	spo	4000000	28946	1117	695
AHRI (cbp)	osp	4000000	26338	1254	580
AHRI (cbp)	pos	4000000	53756	1471	380
AHRI (cbp)	spo	4000000	35320	865	460
AHRI (hb)	osp	4000000	28827	1254	567
AHRI (hb)	pos	4000000	27374	1205	376
AHRI (hb)	spo	4000000	36922	828	402
AHRI (hbwa)	osp	4000000	27008	1346	711
AHRI (hbwa)	pos	4000000	31283	1126	344
AHRI (hbwa)	spo	4000000	37699	891	481

Table D.14: TLB performance counters: restriction by three attributes

D.2.2.5 Restriction by Mixed Attributes

Structure	Attribute Order	Iterations	L1 TLB Accesses (x10,000)	L1 TLB Misses (x10,000)	L2 TLB Misses (x10,000)
Bitmap	osp	2000	49388	300	192
Bitmap	pos	400	86648	458	234
Bitmap	spo	20000	256033	1410	1292
BPTree	osp	30000	139612	409	373
BPTree	pos	3000	154290	580	545
BPTree	spo	4000000	116373	1612	878
BST	osp	30000	291112	1066	974
BST	pos	3000	289999	1516	1432
BST	spo	4000000	170723	3202	1584

continued on next page

Structure	Attribute Order	Iterations	L1 TLB Accesses (x10,000)	L1 TLB Misses (x10,000)	L2 TLB Misses (x10,000)
BTree	osp	30000	168810	315	218
BTree	pos	3000	186458	835	748
BTree	spo	4000000	137493	1779	1083
BTreeRef	osp	30000	104624	699	524
BTreeRef	pos	3000	128119	708	515
BTreeRef	spo	4000000	143827	2842	1718
AHRI (vec,cbp)	osp	30000	99042	188	156
AHRI (vec,cbp)	pos	3000	79101	216	189
AHRI (vec,cbp)	spo	4000000	35162	645	340
AHRI (vec,hb)	osp	30000	79942	271	246
AHRI (vec,hb)	pos	3000	118801	361	330
AHRI (vec,hb)	spo	4000000	44288	589	298
AHRI (vec,hbwa)	osp	30000	79704	99	71
AHRI (vec,hbwa)	pos	3000	78688	238	207
AHRI (vec,hbwa)	spo	4000000	35792	641	336
Hash	osp	30000	794429	2380	2120
Hash	pos	3000	721939	2395	2158
Hash	spo	4000000	95303	1154	719
AHRI (cbp)	osp	30000	65297	82	57
AHRI (cbp)	pos	3000	75662	183	156
AHRI (cbp)	spo	4000000	39983	1069	651
AHRI (hb)	osp	30000	66553	230	207
AHRI (hb)	pos	3000	135983	457	429
AHRI (hb)	spo	4000000	45546	743	360
AHRI (hbwa)	osp	30000	68699	222	197
AHRI (hbwa)	pos	3000	94760	198	167
AHRI (hbwa)	spo	4000000	41687	1016	591

Table D.15: TLB performance counters: restriction by mixed attributes

D.2.3 Branch

D.2.3.1 Load

Structure	Attribute Order	Iterations	Branch Operations (x10,000)	Branch Mispredictions (x10,000)
Bitmap	osp	4000000	39205	1994
Bitmap	pos	4000000	40244	1530
Bitmap	spo	4000000	42627	1668
BPTree	osp	4000000	200393	5525
BPTree	pos	4000000	213223	5254
BPTree	spo	4000000	184619	3196
BST	osp	4000000	68658	3969
BST	pos	4000000	75886	4497
BST	spo	4000000	61036	1862
BTree	osp	4000000	183760	4758
BTree	pos	4000000	210191	4926
BTree	spo	4000000	196933	3267
BTreeRef	osp	4000000	226495	6453
BTreeRef	pos	4000000	255684	7103
BTreeRef	spo	4000000	168458	2946
AHRI (vec,cbp)	osp	4000000	28340	1646
AHRI (vec,cbp)	pos	4000000	254296	3023
AHRI (vec,cbp)	spo	4000000	21834	1517
AHRI (vec,hb)	osp	4000000	27350	1493
AHRI (vec,hb)	pos	4000000	34473	2177
AHRI (vec,hb)	spo	4000000	22481	1412
AHRI (vec,hbwa)	osp	4000000	27034	1742
AHRI (vec,hbwa)	pos	4000000	32476	1991
AHRI (vec,hbwa)	spo	4000000	19026	1255
Hash	osp	4000000	26451	1746
Hash	pos	4000000	24761	1529
Hash	spo	4000000	21590	1234
AHRI (cbp)	osp	4000000	27990	1786
AHRI (cbp)	pos	4000000	251303	2701
AHRI (cbp)	spo	4000000	22689	1311
AHRI (hb)	osp	4000000	31821	2134
AHRI (hb)	pos	4000000	37196	2328
AHRI (hb)	spo	4000000	26289	1634
AHRI (hbwa)	osp	4000000	29319	1863

continued on next page

Structure	Attribute Order	Iterations	Branch Operations (x10,000)	Branch Mispredictions (x10,000)
AHRI (hbwa)	pos	4000000	35446	2173
AHRI (hbwa)	spo	4000000	26574	1704

TABLE D.16: Branch performance counters: load

D.2.3.2 Restriction by One Attribute

Structure	Attribute Order	Iterations	Branch Operations (x10,000)	Branch Mispredictions (x10,000)
Bitmap	osp	1000	73365	3848
Bitmap	pos	120	61759	2598
Bitmap	spo	4000000	80628	5452
BPTree	osp	6000	100529	1186
BPTree	pos	1200	185622	1807
BPTree	spo	4000000	101830	4049
BST	osp	6000	292042	12765
BST	pos	1200	525635	22660
BST	spo	4000000	210896	10194
BTree	osp	6000	104562	1606
BTree	pos	1200	187697	2325
BTree	spo	4000000	119806	4589
BTreeRef	osp	6000	109242	1093
BTreeRef	pos	1200	192785	2490
BTreeRef	spo	4000000	125952	5424
AHRI (vec,cbp)	osp	6000	101274	597
AHRI (vec,cbp)	pos	1200	116792	1003
AHRI (vec,cbp)	spo	4000000	29716	456
AHRI (vec,hb)	osp	6000	91734	509
AHRI (vec,hb)	pos	1200	185003	11425
AHRI (vec,hb)	spo	4000000	30345	619
AHRI (vec,hbwa)	osp	6000	98997	485
AHRI (vec,hbwa)	pos	1200	144604	1052
AHRI (vec,hbwa)	spo	4000000	29955	461
Hash	osp	6000	189475	9438

continued on next page

Structure	Attribute Order	Iterations	Branch Operations (x10,000)	Branch Mispredictions (x10,000)
Hash	pos	1200	393819	18850
Hash	spo	4000000	80311	3574
AHRI (cbp)	osp	6000	98023	572
AHRI (cbp)	pos	1200	122701	994
AHRI (cbp)	spo	4000000	30346	468
AHRI (hb)	osp	6000	90794	474
AHRI (hb)	pos	1200	179381	10386
AHRI (hb)	spo	4000000	30315	454
AHRI (hbwa)	osp	6000	100495	564
AHRI (hbwa)	pos	1200	142665	888
AHRI (hbwa)	spo	4000000	30328	454

TABLE D.17: Branch performance counters: restriction by one attribute

D.2.3.3 Restriction by Two Attributes

Structure	Attribute Order	Iterations	Branch Operations (x10,000)	Branch Mispredictions (x10,000)
Bitmap	osp	20000	113580	2137
Bitmap	pos	600	56072	1858
Bitmap	spo	20000	422127	1617
BPTree	osp	4000000	70111	4492
BPTree	pos	6000	101303	819
BPTree	spo	4000000	66725	4000
BST	osp	4000000	70611	4335
BST	pos	6000	257133	10271
BST	spo	4000000	91264	5831
BTree	osp	4000000	71610	4577
BTree	pos	6000	83582	1033
BTree	spo	4000000	73716	4296
BTreeRef	osp	4000000	71775	4687
BTreeRef	pos	6000	85630	1197
BTreeRef	spo	4000000	76701	5071
AHRI (vec,cbp)	osp	4000000	14347	834
AHRI (vec,cbp)	pos	6000	45384	354

continued on next page

Structure	Attribute Order	Iterations	Branch Operations (x10,000)	Branch Mispredictions (x10,000)
AHRI (vec,cbp)	spo	4000000	22919	569
AHRI (vec,hb)	osp	4000000	12603	840
AHRI (vec,hb)	pos	6000	91873	7037
AHRI (vec,hb)	spo	4000000	25191	735
AHRI (vec,hbwa)	osp	4000000	13319	531
AHRI (vec,hbwa)	pos	6000	55421	381
AHRI (vec,hbwa)	spo	4000000	26586	640
Hash	osp	12000	343757	19153
Hash	pos	4000	979724	63736
Hash	spo	4000000	75173	4353
AHRI (cbp)	osp	4000000	14949	595
AHRI (cbp)	pos	6000	49915	400
AHRI (cbp)	spo	4000000	25633	575
AHRI (hb)	osp	4000000	13270	531
AHRI (hb)	pos	6000	84908	6304
AHRI (hb)	spo	4000000	28877	652
AHRI (hbwa)	osp	4000000	13788	561
AHRI (hbwa)	pos	6000	61027	316
AHRI (hbwa)	spo	4000000	23360	548

TABLE D.18: Branch performance counters: restriction by two attributes

D.2.3.4 Restriction by Three Attributes

Structure	Attribute Order	Iterations	Branch Operations (x10,000)	Branch Mispredictions (x10,000)
Bitmap	osp	20000	582056	4202
Bitmap	pos	20000	913879	10748
Bitmap	spo	20000	570845	3766
BPTree	osp	4000000	68723	4479
BPTree	pos	4000000	77821	4880
BPTree	spo	4000000	64739	4155
BST	osp	4000000	69004	4224
BST	pos	4000000	81926	4751
BST	spo	4000000	79531	5257

continued on next page

Structure	Attribute Order	Iterations	Branch Operations (x10,000)	Branch Mispredictions (x10,000)
BTree	osp	4000000	68070	4406
BTree	pos	4000000	83937	4990
BTree	spo	4000000	65653	4144
BTreeRef	osp	4000000	70226	4643
BTreeRef	pos	4000000	78654	5240
BTreeRef	spo	4000000	65053	5098
AHRI (vec,cbp)	osp	4000000	13970	768
AHRI (vec,cbp)	pos	4000000	28048	1779
AHRI (vec,cbp)	spo	4000000	24270	552
AHRI (vec,hb)	osp	4000000	13129	831
AHRI (vec,hb)	pos	4000000	15769	680
AHRI (vec,hb)	spo	4000000	23148	669
AHRI (vec,hbwa)	osp	4000000	12234	435
AHRI (vec,hbwa)	pos	4000000	14898	657
AHRI (vec,hbwa)	spo	4000000	23944	587
Hash	osp	4000000	12394	443
Hash	pos	4000000	11765	366
Hash	spo	4000000	12865	471
AHRI (cbp)	osp	4000000	13821	932
AHRI (cbp)	pos	4000000	26316	2115
AHRI (cbp)	spo	4000000	23341	506
AHRI (hb)	osp	4000000	12223	526
AHRI (hb)	pos	4000000	15664	568
AHRI (hb)	spo	4000000	22887	535
AHRI (hbwa)	osp	4000000	13929	549
AHRI (hbwa)	pos	4000000	14442	524
AHRI (hbwa)	spo	4000000	21409	510

TABLE D.19: Branch performance counters: restriction by three attributes

D.2.3.5 Restriction by Mixed Attributes

Structure	Attribute Order	Iterations	Branch Operations (x10,000)	Branch Mispredictions (x10,000)

continued on next page

Structure	Attribute Order	Iterations	Branch Operations (x10,000)	Branch Mispredictions (x10,000)
Bitmap	osp	2000	46804	1040
Bitmap	pos	400	55979	1618
Bitmap	spo	20000	427699	2252
BPTree	osp	30000	56937	610
BPTree	pos	3000	65766	591
BPTree	spo	4000000	73897	4460
BST	osp	30000	151845	6276
BST	pos	3000	193102	10108
BST	spo	4000000	98621	6665
BTree	osp	30000	59608	908
BTree	pos	3000	64316	773
BTree	spo	4000000	84834	4988
BTreeRef	osp	30000	56705	607
BTreeRef	pos	3000	59250	843
BTreeRef	spo	4000000	77092	5156
AHRI (vec,cbp)	osp	30000	58711	361
AHRI (vec,cbp)	pos	3000	41899	358
AHRI (vec,cbp)	spo	4000000	26095	783
AHRI (vec,hb)	osp	30000	49050	261
AHRI (vec,hb)	pos	3000	57121	3953
AHRI (vec,hb)	spo	4000000	26354	932
AHRI (vec,hbwa)	osp	30000	54817	326
AHRI (vec,hbwa)	pos	3000	48116	401
AHRI (vec,hbwa)	spo	4000000	26420	767
Hash	osp	30000	531052	28884
Hash	pos	3000	453220	27860
Hash	spo	4000000	54755	3189
AHRI (cbp)	osp	30000	44123	182
AHRI (cbp)	pos	3000	40507	365
AHRI (cbp)	spo	4000000	25025	724
AHRI (hb)	osp	30000	49931	259
AHRI (hb)	pos	3000	63345	4232
AHRI (hb)	spo	4000000	25240	700
AHRI (hbwa)	osp	30000	51401	310

continued on next page

Structure	**Attribute Order**	**Iterations**	**Branch Operations (x10,000)**	**Branch Mispredictions (x10,000)**
AHRI (hbwa)	pos	3000	42697	283
AHRI (hbwa)	spo	4000000	25569	722

TABLE D.20: Branch performance counters: restriction by mixed attributes

Bibliography

D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 671–682. ACM Press, 2006.

D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Databases*, pages 411–422, 2007.

M. Adelson-Velskii and E.M Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:1259–1262, 1962.

A Adl-Tabatabai, M Cierniak, G Lueh, V. M Parikh, and J. M Stichnoth. Fast, effective code generation in a just-in-time Java compiler. *SIGPLAN Notices*, 33(5):280–290, 1998.

V. Agarwal, MS Hrishikesh, S.W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. *ACM SIGARCH Computer Architecture News*, 28(2):259, 2000.

Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A practical scalable distributed B-tree. *Proceedings of the Very Large Databases Endowment*, 1(1):598–609, 2008.

A. Ailamaki, D.J. DeWitt, M.D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Databases*, 2001.

A. Ailamaki, D.J. DeWitt, M.D. Hill, and D.A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Databases*, pages 266–277, 1999.

D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo. The IBM system/360 model 91: machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11:8–24, 1967.

A.W. Appel and J. Palsberg. *Modern compiler implementation in Java*. Cambridge University Press, 2002.

R. Apweiler, A. Bairoch, C.H. Wu, W.C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, et al. UniProt: the universal protein knowledgebase. *Nucleic acids research*, 32:D115, 2004.

C.R. Aragon and R.G. Seidel. Randomized search trees. *Annual IEEE Symposium on Foundations of Computer Science*, 0:540–545, 1989.

Malcolm Atkinson, David Dewitt, David Maier, Francois Bancilhon, Klaus Dittrich, and Stanley Zdonik. The object-oriented database system manifesto. In *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases*, volume 57, pages 223–40. North-Holland/Elsevier, 1989.

S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: A nucleus for a web of open data. In *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007.

A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. *ACM SIGPLAN Notices*, 32:134–145, 1997.

D.F. Bacon, S.J. Fink, and D. Grove. Space-and time-efficient implementation of the Java object model. In *ECOOP 2002 Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 13–27. Springer, 2002.

A.H. Badawy, A. Aggarwal, D. Yeung, and C.W. Tseng. The efficacy of software prefetching and locality optimizations on future memory systems. *Journal of Instruction-Level Parallelism*, 6(7), 2004.

D. Battre, F. Heine, A. Hoing, and O. Kao. Load-balancing in P2P based RDF stores. In *Proceedings of the Second International Workshop on Scalable Semantic Web Knowledge Base Systems*, Athens, Georgia, 2006.

R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.

R. Bayer and V. Markl. The UB-tree: Performance of multidimensional range queries. Technical report, Technische Universitat Munchen, Informatik, Munchen Technical Report TUM- I, 1998.

C. Becker. RDF store benchmarks with DBpedia. http://www4.wiwiss.fu-berlin.de/benchmarks-200801/ , 2008.

D. Beckett. The design and implementation of the Redland RDF application framework. *Computer Networks*, 39(5):577–588, 2002.

F. Belleau, M.A. Nolin, N. Tourigny, P. Rigault, and J. Morissette. Bio2rdf: Towards a mashup to build bioinformatics knowledge systems. *Journal of Biomedical Informatics*, 41(5):706–716, 2008.

T. Berners-Lee, Y. Chen, L. Chilton, D. Connolly, R. Dhanaraj, J. Hollenbach, A. Lerer, and D. Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *Proceedings of the 3rd International Semantic Web User Interaction Workshop, Athens, USA*, 2006.

T. Berners-Lee, R. Fielding, and L. Masinter. RFC2396: Uniform Resource Identifiers (URI): Generic syntax, 1998.

T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):28–37, 2001.

C. Bizer and R. Cyganiak. D2R Server–publishing relational databases on the semantic web. In *Poster and Demo Proceedings of the 5th International Semantic Web Conference, Athens, Georgia*, 2006.

C Bizer and A Schultz. The Berlin SPARQL benchmark. *International Journal On Semantic Web and Information Systems*, 5(2):1–24, 2009.

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. *SIGMETRICS Performance Evaluation Review*, 32(1):25–36, 2004.

Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.K. Su. Myrinet: A gigabit-per-second local area network. *Micro, IEEE*, 15(1): 29–36, 1995. ISSN 0272-1732.

P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Databases*, pages 54–65, 1999.

P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research*, 2005.

H. Boral. Parallelism in Bubba. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, pages 68–71, 1988.

H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, 1990.

D. Borthakur. The Hadoop distributed file system: Architecture and design. Technical report, The Apache Software Foundation, http://hadoop.apache.org/common/docs/r0.18.0/hdfs_design.pdf , 2007.

D. Brickley, R.V. Guha, and B. McBride. RDF vocabulary description language 1.0: RDF Schema. W3C recommendation, http://www.w3.org/TR/rdf-schema/ , 2004.

S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN systems*, 30(1-7):107–117, 1998.

J. Broekstra and A. Kampman. Inferencing and truth maintenance in RDF Schema. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems*, 2003a.

J. Broekstra and A. Kampman. SeRQL: a second generation RDF query language. In *Proceedings of the SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, 2003b.

J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proceedings of the 1st International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2002.

J. Broekstra, A. Kampman, and F. van Harmelen. *Sesame: An Architecture for Storing and Querying RDF Data and Schema Information*. MIT Press, 2003.

M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer networks. In *Proceedings of the 13th International World Wide Web Conference*, pages 650–657. ACM Press, 2004.

J.J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *Proceedings of the 13th International World Wide Web Conference*, pages 74–83. ACM Press, 2004.

J Casazza. Intel Core i7-800 processor series and the Intel Core i5-700 processor series based on Intel Microarchitecture (Nehalem). Technical report, Intel Corporation, http://download.intel.com/products/processor/corei7/319724.pdf , 2009.

S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

S. Chen, P.B. Gibbons, and T.C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 235–246. ACM, 2001.

EF Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

D. Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 1990.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*. USENIX Association, 2004.

D.J. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of the 11th International Conference on Very Large Databases, Stockholm, Sweden*, pages 151–164, 1985.

D.J. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 35(6):85–98, 1992.

L. Ding, K. Wilkinson, C. Sayers, and H. Kuno. Application-specific schema design for storing large RDF datasets. In *1st International Workshop on Practical and Scalable Semantic Systems*, 2003.

U. Drepper. What every programmer should know about memory. http://lwn.net/Articles/250967/ , 2007.

P. J. Drongowski. Basic performance measurements for AMD Athlon 64, AMD Opteron and AMD Phenom processors. AMD, http://developer.amd.com/documentation/articles/pages/1212200690_3.aspx , 2008.

O. Erling. Advances in Virtuoso RDF triple storage (bitmap indexing). Technical report, Openlink Software Inc, http://virtuoso.openlinksw.com/wiki/main/Main/VOSBitmapIndexing , 2006.

O. Erling. Faceted views over large-scale linked data. In *Proceedings of the Linked Data on the Web Workshop, Madrid, Spain*, 2009.

O. Erling and I. Mikhailov. Towards web scale RDF. In *Proceedings of the 4th International Workshop on Scalable Semantic Web Knowledge*. www.openlinksw.com/weblog/oerling/2008iswc_webscale_rdf. pd, 2008.

O. Erling and I. Mikhailov. RDF support in the virtuoso DBMS. In *Networked Knowledge - Networked Media*, volume 221, pages 7–24. Springer, 2009.

B. Fulgham and I. Gouy. The computer language benchmarks game. http://shootout.alioth.debian.org/ .

R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious b+-trees. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 283–294. ACM, 2003.

S. Harizopoulos, V. Liang, D.J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the 32nd International Conference on Very Large Databases*, pages 487–498. VLDB Endowment, 2006.

S. Harris. SPARQL query processing with conventional relational database systems. In *Web Information Systems Engineering–WISE 2005 Workshops*, volume 3807 of *Lecture Notes in Computer Science*, pages 235–244, 2005.

S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, page 86.

A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. In *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2007.

P. Hawthorn and M. Stonebraker. The use of technological advances to enhance database system performance. In *The INGRES papers: anatomy of a relational database system*, pages 106–130. Addison-Wesley Longman, 1986.

F. Heine, M. Hovestadt, and O. Kao. Processing complex RDF queries over P2P networks. In *Proceedings of the 2005 ACM Workshop on Information Retrieval in Peer-to-Peer Networks*, pages 41–48. ACM Press, 2005.

J. Hendler. Agents and the Semantic Web. *IEEE Intelligent Systems*, 16(2):30–37, 2001.

K.A. Hua and C. Lee. An adaptive data placement scheme for parallel database computer systems. In *Proceedings of the 16th International Conference on Very Large Databases*, pages 493–506, 1990.

InfiniBand Trade Association. InfiniBand Architecture Specification, 2001.

E. Jain, A. Bairoch, S. Duvaud, I. Phan, N. Redaschi, B.E. Suzek, M.J. Martin, P. McGarvey, and E. Gasteiger. Infrastructure for the life sciences: design and implementation of the UniProt website. *BMC bioinformatics*, 10:136, 2009.

G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: a declarative query language for RDF. In *Proceedings of the 11th International World Wide Web Conference*, page 603. ACM, 2002.

K. Keeton, D.A. Patterson, Y.Q. He, R.C. Raphael, and W.E. Baker. Performance characterization of a quad Pentium Pro SMP using OLTP workloads. *ACM SIGARCH Computer Architecture News*, 26(3):15–26, 1998.

MF Khan, R. Paul, I. Ahmed, and A. Ghafoor. Intensive data management in parallel systems: A survey. *Distributed and Parallel Databases*, 7(4):383–414, 1999.

B Knighten. Detailed characterization of a quad pentium pro server running TPC-D. In *Proceedings of the 1999 IEEE International Conference on Computer Design*, page 108. IEEE Computer Society, 1999. ISBN 0-7695-0406-X.

T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):1–32, 2008.

A. Langegger and W. Wöß. RDFStats-an extensible RDF statistics generator and library. In *Proceedings of the 2009 20th International Workshop on Database and Expert Systems Application*, pages 79–83. IEEE Computer Society, 2009.

O. Lassila, R.R. Swick, et al. Resource description framework (RDF) model and syntax specification. W3C Recommendation, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/ , 1999.

J.K. Lawder and P.J.H. King. Using space-filling curves for multi-dimensional indexing. In *Proceedings of the 17th British National Conference on Databases*, volume 1832 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2000.

R. Lee. Scalability report on triple store applications. Massachusetts Institute of Technology, http://simile.mit.edu/reports/stores/ , 2004.

S. Lee, B. Moon, and C. Park. Advances in flash memory SSD technology for enterprise database applications. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 863–870, New York, NY, USA, 2009. ACM.

T. J. Lehman and M. J. Carey. Query processing in main memory database management systems. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 239–250. ACM, 1986. ISBN 0-89791-191-1.

H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):429, 1983.

D. Lomet. The evolution of effective B-trees: page organization and techniques: a personal account. *ACM SIGMOD Record*, 30(3):64–69, 2001.

M. Mehta and D.J. DeWitt. Data placement in shared-nothing parallel database systems. *The International Journal on Very Large Data Bases*, 6(1):53–72, 1997.

B. Motik, P.F. Patel-Schneider, B. Parsia, C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, et al. OWL 2 web ontology language: Structural specification and functional-style syntax. W3C recommendation, http://www.w3.org/TR/2009/REC-owl2-overview-20091027/ , 2009.

Rene Mueller and Jens Teubner. FPGA: what's in it for a database? In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 999–1004, New York, NY, USA, 2009. ACM.

W. Nejdl, W. Siberski, and M. Sintek. Design issues and challenges for RDF-and Schema-based peer-to-peer systems. *ACM SIGMOD Record*, 32(3):41–46, 2003.

T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. In *Proceedings of the Very Large Databases Endowment*, volume 1, pages 647–659. VLDB Endowment, 2008.

D. Ognyanoff, A. Kiryakov, R. Velkov, and M. Yankova. A scalable repository for massive semantic annotation. Technical Report D2.6.3, SEKT, 2007.

M.A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 43–43. USENIX Association, 1999.

J.K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, M. Rosenblum, S.M. Rumble, et al. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

A. Owens, A. Seaborne, N. Gibbins, and m. c. schraefel. Clustered TDB: A clustered triple store for Jena, 2008.

J.W. Palmer. Web site usability, design, and performance metrics. *Information Systems Research*, 13(2):151–167, 2003.

P.F. Patel-Schneider, P. Hayes, I. Horrocks, and F. van Harmelen. OWL web ontology language; semantics and abstract syntax, W3C recommendation. http://www.w3.org/TR/2003/CR-owl-semantics-20030818/ , 2004.

M. Poess and D. Potapov. Data compression in Oracle. In *Proceedings of the 29th International Conference on Very Large Databases*, pages 937–947. VLDB Endowment, 2003.

E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation. *World Wide Web Consortium*, 2006.

F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the UB-tree into a database system kernel. In *Proceedings of the 26th International Conference on Very Large Databases*, pages 263–272, 2000.

J. Rao and K.A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th Very Large Databases Conference*, pages 78–89, 1999.

J. Rao and K.A. Ross. Making B+trees cache conscious in main memory. *ACM SIGMOD Record*, 29(2):475–486, 2000.

EM Riseman and CC Foster. The Inhibition of Potential Parallelism by Conditional Jumps. *Transactions on Computers*, 100(21):1405–1411, 1972.

K. Rohloff, M. Dean, I. Emmons, D. Ryder, and J. Sumner. An evaluation of triple-store technologies for large data stores. In *Proceedings of the 2007 OTM Confederated International Conference*, volume 4806 of *Lecture Notes in Computer Science*, page 1105. Springer, 2007.

L.A. Rowe and M. Stonebraker. The commercial INGRES epilogue. In *The INGRES papers: anatomy of a relational database system*, pages 63–82. Addison-Wesley Longman, 1986.

K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *SIGPLAN Notices*, 41(10):263–272, 2006.

H. Sagan and J. Holbrook. *Space-filling curves*. Springer-Verlag New York, 1994.

M. Schmidt, T. Hornung, N. Kchlin, G. Lausen, and C. Pinkel. An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario. In *The Semantic Web - ISWC 2008*, volume 5318, pages 82–97. Springer, 2010.

m. c. schraefel, Nigel R. Shadbolt, Nicholas Gibbins, Stephen Harris, and Hugh Glaser. CS AKTive Space: representing computer science in the semantic web. In *Proceedings of the 13th International World Wide Web Conference*, pages 384–392, New York, NY, USA, 2004. ACM.

m.c. schraefel, D.A. Smith, A. Owens, A. Russell, C. Harris, and M. Wilson. The evolving mSpace platform: Leveraging the semantic web on the trail of the memex. In *Proceedings of the 16th ACM conference on Hypertext and Hypermedia*, pages 174–183. ACM Press New York, NY, USA, 2005.

A. Seaborne. RDQL - a query language for RDF. W3C Member Submission, http://www.w3.org/Submission/RDQL/ , 2004.

Andy Seaborne. personal communication, August 2008.

Andy Seaborne. personal communication, February 2009.

M. Shao, J. Schindler, S.W. Schlosser, A. Ailamaki, and G.R. Ganger. Clotho: Decoupling memory page layout from storage organization. In *Proceedings of the 30th International Conference on Very Large Databases*, pages 696–707, 2004.

J.P. Sheu and T.H. Thai. Partitioning and mapping nested loops on multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):430–439, 1991.

D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

D. Smith, A. Owens, m. c. schraefel, P Sinclair, P. Andre, M.L. Wilson, A. Russell, K. Martinez, and P. Lewis. Challenges in supporting faceted semantic browsing of multimedia collections. In *Proceedings of the Semantic and Digital Media Technologies 2nd International Conference on Semantic Multimedia*, pages 280–283. Springer, 2007.

M. Stonebraker. Retrospection on a database system. *ACM Transactions on Database Systems*, 5(2):225–240, 1980.

M. Stonebraker. The case for shared nothing. *Database Engineering Bulletin*, 9(1):4–9, 1986.

M. Stonebraker, G. Held, E. Wong, and P. Kreps. The design and implementation of INGRES. *ACM Transactions on Database Systems (TODS)*, 1(3):189–222, 1976.

M. Stonebraker and G. Kemnitz. The POSTGRES next generation database management systems. *Communications of the ACM*, 34(10):78–92, 1991.

M. Stonebraker, S. Madden, D.J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Databases*, pages 1150–1160. VLDB Endowment, 2007.

M. Stonebraker, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, S. Zdonik, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, et al. C-store: a column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Databases*, pages 553–564. VLDB Endowment, 2005.

Sun Microsystems. Memory management in the Java HotSpot virtual machine. Technical report, http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf , 2006.

Sun Microsystems. Java SE 6 performance white paper. Technical report, http://java.sun.com/performance/reference/whitepapers/6_performance.html , 2008.

H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):16–20, 2005.

K. Taylor, R. Gledhill, J.W. Essex, J.G. Frey, S.W. Harris, and D. De Roure. A semantic datagrid for combinatorial chemistry. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 148–155, 2005.

K.R. Taylor, R.J. Gledhill, J.W. Essex, J.G. Frey, S.W. Harris, and D. De Roure. Bringing chemical data onto the semantic web. *Journal of Chemical Information and Modeling*, 46(3):939–952, 2006.

Robert W. Taylor and Randall L. Frank. Codasyl data-base management systems. *ACM Comput. Surv.*, 8(1):67–103, 1976.

D. C. Tsichritzis and F. H. Lochovsky. Hierarchical data-base management: A survey. *ACM Comput. Surv.*, 8(1):105–123, 1976.

D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. ACM, 1984.

C. Weiss and A. Bernstein. On-disk storage techniques for semantic web data-are b-trees always the optimal solution? In *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, page 49, 2009.

C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the Very Large Databases Endowment archive*, 1 (1):1008–1019, 2008.

T White. *Hadoop: The Definitive Guide*. O'Reilly, 2009.

K. Wilkinson. Jena property table design. In *Proceedings of the Jena Users Conference, Bristol, England*, 2006.

K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proceedings of the First International Workshop on Semantic Web and Databases*, volume 3, pages 7–8, 2003.

D. Wood, P. Gearon, and T. Adams. Kowari: A platform for semantic web storage and analysis. In *Proceedings of the 14th International World Wide Web Conference*, 2005.

K Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006. ISSN 0362-5915.

T. Würthinger, C. Wimmer, and H. Mössenböck. Array bounds check elimination for the Java HotSpot client compiler. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 125–133. ACM New York, NY, USA, 2007.

T.Y. Yeh and Y.N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61. ACM, 1991. ISBN 0897914600.

K. Youssefi and E. Wong. Query processing in a relational database management system. In *The Fifth International Conference on Very Large Data Bases*, pages 409–417, 1979.

J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, page 59, 2006.