UNIVERSITY OF SOUTHAMPTON
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

A mini-thesis submitted for transfer from MPhil to PhD

Supervisor: dr monica mc schraefel, Dr Nick Gibbins
Examiner: Dr Kirk Martinez

An Investigation into Improving RDF Store Performance

by Alisdair Owens

March 13, 2009

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

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

A mini-thesis submitted for transfer from MPhil to PhD

by Alisdair Owens

This report considers the requirement for fast, efficient, and scalable triple stores as part of the effort to enable the Semantic Web. It summarises relevant information in the background field of Database Management Systems (DBMS), and analyses these techniques for the purposes of application in the field of RDF storage. The report concludes that for individuals and organisations to be willing to use Semantic Web technologies, data stores must advance beyond their current state. The report notes that there are several areas with opportunities for development, including scalability, low latency querying, distributed and federated stores, and improved support for updates and deletions. Experiences from the DBMS field can be used to maximise RDF store performance, and suggestions are provided for lines of investigation. Work already performed by the author on benchmarking and distributed storage is described, and a proposal made to research low-latency RDF querying for the purpose of supporting applications that require human interaction. Work packages are provided describing expected timetables for the remainder of the PhD program.

Contents

Acknowledgements **xi**

1 Introduction **1**
- 1.1 Memory Storage for Low Latency RDF Query . . . 2
- 1.2 Overview of the Report . . . 4

2 Background and Research Motivation **5**
- 2.1 The Semantic Web . . . 5
- 2.2 Data Representation . . . 6
  - 2.2.1 RDFS and OWL . . . 9
- 2.3 Data Extraction . . . 10
- 2.4 RDF in Relation to Other Database Models . . . 11
  - 2.4.1 Early Database Models . . . 11
  - 2.4.2 The Relational Data Model . . . 12
  - 2.4.3 Other Data Models . . . 14
  - 2.4.4 Representing RDF . . . 14

3 Exploration of the Problem Domain **19**
- 3.1 Characteristics of Modern Hardware . . . 20
  - 3.1.1 Disk . . . 20
  - 3.1.2 Main Memory . . . 21
  - 3.1.3 CPU . . . 21
    - 3.1.3.1 Superscalar and Pipelined Architectures . . . 22
    - 3.1.3.2 Caching . . . 23
    - 3.1.3.3 Multiple Cores . . . 25
  - 3.1.4 Network . . . 26
  - 3.1.5 Summary . . . 26
- 3.2 Physical Representation: Translating a Data Model into a Performant Storage Layer . . . 27
  - 3.2.1 Physical Representations in DBMSs . . . 27
    - 3.2.1.1 Compression . . . 29
  - 3.2.2 Physical Representation in RDF Stores . . . 30
    - 3.2.2.1 Normalising . . . 32
    - 3.2.2.2 Updates and Deletion . . . 33
  - 3.2.3 Summary . . . 34
- 3.3 Indexing: A Key to High Performance RDF Stores . . . 35
  - 3.3.1 Binary Search Trees . . . 35

3.3.2 B-trees 37
3.3.3 Bitmaps 39
3.3.4 Hash Tables 41
3.3.5 Space Filling Curves 42
3.3.6 Summary 44
3.4 Operator Implementation: The Importance of the Join in RDF Query . . 44
3.4.1 Query Optimisation 45
3.4.2 Types of Join 47
3.4.2.1 Nested Loop 47
3.4.2.2 Merge and Sort/Merge 47
3.4.2.3 Hash 48
3.4.3 Join Minimisation 48
3.4.4 Summary 48
3.5 Scaling to Extremely Large Systems Through Distribution 49
3.5.1 Enabling Parallelism 51
3.5.2 Data Partitioning 52
3.5.3 Distributing RDF Stores 53
3.5.3.1 Distributing Memory Stores 55
3.5.4 Summary 56
3.6 Opportunities 56

4 Measuring RDF Store Performance 59
4.1 Existing RDF Benchmarks 60
4.2 A New RDF Test Set 61
4.3 A Use-Case Based Test 62

5 Future Work 65
5.1 Performance 66
5.2 Storage Capacity 67
5.3 DBMSs on a Virtual Machine 67
5.4 Work Packages 69
5.4.1 WP1 - Characterisation of the Jena Framework 69
5.4.2 WP2 - Investigation into Coding for a Virtual Machine 69
5.4.3 WP3 - Design and Implementation of Prototype(s) 70
5.4.4 WP4 - Testing Against mSpace 70

6 Conclusions 73

A Test Cases 75

B Binary Chop Tests 79
B.1 Java Implementation 79
B.2 C Implementation 80

Bibliography 83

List of Figures

2.1 The Semantic Web layer cake. 7
2.2 Triple Concept. 7
2.3 RDF Triple 8
2.4 RDF Graph 8
2.5 SPARQL Query 10
2.6 SPARQL triple pattern 10
2.7 Illustration of common database operations 13

3.1 Data storage hierarchy. 24
3.2 Cost of Binary Chop as Dataset Increases in Size 25
3.3 3store data schema. 30
3.4 SQL produced by 3Store 31
3.5 Balanced Binary Search Tree 36
3.6 RDF stored using a BST 37
3.7 RDF IDs stored using a B^{+}tree 38
3.8 Querying using a bitmap index 40
3.9 The two dimensional Hilbert curve 42
3.10 Rates of assertion during a Clustered TDB load 55

4.1 Configurable RDF Graph Shapes 62

List of Tables

2.1 Database model comparison 17

3.1 Cost of Binary Chop as Dataset Increases in Size 24

5.1 Comparison of Java and C on an unpredictable large scale binary chop . . 68
5.2 Comparison of Java and C on a predictable large scale binary chop 68

Acknowledgements

Thanks to Andy Seaborne at HP Labs Bristol for so consistently offering his help and advice.

Chapter 1

Introduction

Resource Description Framework (RDF) is a means for expressing knowledge in a generic manner, without requirement for adherence to a strong 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).

This document, submitted for upgrade from MPhil to PhD, describes the author's work in the area of high performance RDF storage and query. It is clearly necessary to support high performance querying over RDF data: development of the Semantic Web implies the encoding of a massive quantity of data, and without the ability to manipulate and extract information in a performant manner it will be impossible to develop the kind of interfaces that popularised the Web.

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.

RDF does, however, exhibit 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 also exhibits an unusually large number of individual data points compared to the amount of information encoded, meaning each operation generally has more datums to process.

Typically, each of these issues inhibits efficient storage and query optimisation, making even advanced RDF stores both slow and lacking in scalability in comparison to their relational peers[1]. The most powerful single machine RDF stores are capable of storing around two billion RDF statements, or in the order of tens of gigabytes of data (Erling and Mikhailov, 2006), and pattern-match queries performed over much smaller datasets can produce unacceptable performance (Smith et al., 2007). This contrasts with commercial RDBMS technologies which are capable of storing terabytes of data whilst preserving real time query performance, and can lead interface designers to abandon RDF stores in favour of faster, but more restrictive systems (Smith et al., 2007).

Given the damaging performance gap between RDF stores and traditional RDBMSs, it is useful to consider the question of how applicable existing DBMS research is to the problem of RDF query, and whether that knowledge is correctly applied in existing RDF stores. This report contains a detailed review of research in the wider field of DBMSs, and analyses it in the context of the unusual requirements of RDF. This review forms a basis for innovation in RDF storage, uncovering useful techniques that can be applied to RDF stores, areas of future work that result from the differences between RDF stores and traditional systems, and offering explanation of the issues that current stores experience. As far as the author is aware, there is no such analysis already in existence, and it is believed that this work will greatly inform the process of RDF store development, and contribute to narrowing the performance gap between RDF stores and RDBMSs.

The performance issues of RDF stores can be categorised as scale and query latency related. 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 little work has examined the creation of stores that offer very 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. An example of an application with such frustrated requirements is already present in the mSpace faceted browser (Smith et al., 2007). This document highlights the need for stores that offer query latencies suitable for human interaction, and it is upon this that the author's future work focuses.

1.1 Memory Storage for Low Latency RDF Query

In practice, the author argues that hardware changes are coming to the fore that will aid the creation of low latency RDF stores. As main memory continues to get cheaper, it can act as a primary storage mechanism for RDF data in interactive systems - a trend that parallels the DBMS community (Stonebraker et al., 2007). Main memory is

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

fundamentally faster than disk based storage, and offers much improved characteristics with regard to the nonsequential access that is common in RDF systems. This hardware shift will result in a change in focus for RDF store optimisation, from hiding the latency created by the use of slow disks, to improving characteristics with regards to physical footprint, cache performance, and CPU utilisation.

Of course, creating new techniques is difficult without a means to measure the improvement that results from them. This report describes work performed by the author on the topic of measuring RDF store performance, with the particular aim of being able to test the individual components of the system, as well as a use-case based benchmark running simulated mSpace sessions.

This work will form the basis of evaluation for the author's proposed future work: an adaptive in-memory storage structure for RDF that is aware of the architecture of modern machines, providing both footprint and performance improvements over existing in-memory RDF stores. Architecture-aware indexes and storage layers have already provided significant performance gains in the DBMS world (Rao and Ross, 2000; Boncz et al., 2005), and it can be expected that such improvements can be realised in RDF stores as well, given the creation of appropriate algorithms.

The work described in this report offers the following contributions:

- A detailed insight into the relationship between traditional RDBMSs and RDF stores, including an analysis of how to close the performance gap between the two.
- Discussion of the design and prototype of a new distributed store designed especially for RDF.
- A new benchmarking/test system that offers the Semantic Web community new ways to test the performance and utility of RDF stores, providing the ability to break down the performance of different components of the tested systems in a manner not supported by existing benchmarks.
- A description of future work in the area of in-memory RDF storage that will show that application of knowledge regarding the underlying machine architecture can provide significant improvements in both space efficiency and performance of RDF stores.

The result of these contributions will be an improved understanding of the problem of storing and querying RDF, and an application of this understanding in the area of low latency RDF stores. This will increase the practicality of delivering rich, interactive RDF-based applications, and thus encourage the large scale knowledge building that the Semantic Web requires.

1.2 Overview of the Report

The report is structured as follows:

- Chapter 2 provides background information on the Semantic Web as a whole, and individual technologies in particular, in order to frame and justify the work undertaken for this mini-thesis. It goes on to consider the data models found in existing DBMSs, with particular focus on the relational model, and relates them to the RDF data model. It considers in particular the question of where the RDF data model differs from existing constructs, framing the available areas of future research.
- Chapter 3 details several areas of research in the DBMS world: translation of logical data model into physical representation, indexing, operator implementation, and distribution. Their implementation is analysed with respect to a thorough background of the characteristics of modern computer hardware.

  This information is analysed in the context of the problem of RDF storage, seeking to discover how lessons learned from the RDBMS world can be applied to the problem of RDF storage, and where new innovations need to be made. Each section provides a summary of the salient points, and the conclusion of the chapter suggests future directions for RDF storage based on this information. Reference is made in Section 3.5 to research performed by the author aimed at the creation of a scalable distributed RDF store.
- Chapter 4 describes the means that exist for testing the performance of RDF stores, including a body of work contributed by the author. This is important in order to validate the future work described in Chapter 5.
- Chapter 5 details the direction that has been decided upon for future research, with justification and reference to a set of work packages that will be required to complete the research.
- Chapter 6 concludes with a summary of the points made, and explanation of the contributions that will be made by this research.

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 Semantic Web

The Semantic Web (Berners-Lee et al., 2001) describes 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 traverse the web. 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, I am undertaking a new fitness regime at the gym. As well as regular exercise, my trainer has recommended me a more healthy diet plan. As a member of the gym, I have complimentary access to a large selection of recipes. Since I feel like trying something new, I ask my agent (accessed through a PDA) to pick one for me. The agent, knowing the foods that I particularly like and dislike, works on finding me 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 I might particularly enjoy. It then presents the best option to me for confirmation, along with a note that I will need to buy more ingredients to be able to cook it. It sounds good, so I accept, and ask the agent to tell me where I can get the items I need from. The agent, knowing that the weather is good and that I like to walk, looks for shops in the immediate area, and suggests two in close proximity that between them should stock everything that I need.

This example shows a variety of benefits, in the elimination of a great deal of drudgery from my life. Of course, if I want to perform any tasks, such as picking the recipe myself, I can, but if I choose I can have large parts of my life automated for me. This example is enabled by the intersection of two concepts: intelligent agents and the semantic web. The agent learns about my 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 I want to know the weather for). The shops' agents understand various kinds of food and whether something is in stock. My 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, RDF-S, OWL, and SPARQL (SPARQL Protocol and RDF Query Language) 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 RDF-S 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,

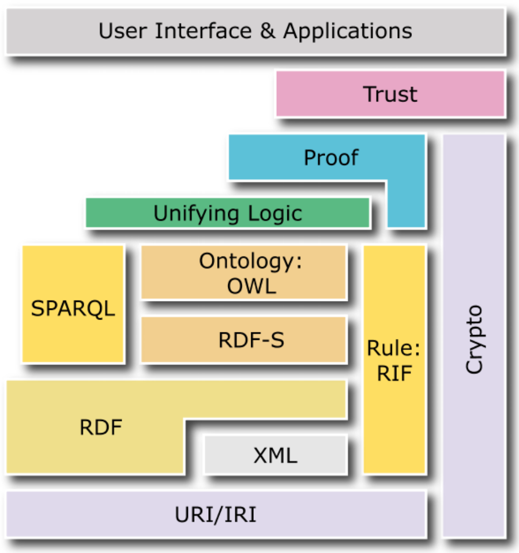


FIGURE 2.1: The Semantic Web layer cake.

composed of subject, predicate, and object. 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.

Alisdair
Has Gender
Male

FIGURE 2.2: Triple Concept.

RDF triples are built out of Uniform Resource Identifiers (URIs) 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.

The use of URIs in RDF makes it easy to find documents that relate to information that I am interested in and understand. For example, if I (or my piece of software) am looking for information about dogs, and I know the URI *http://www.example.com/animals/dog* refers to the concept of a dog, I know that a triple containing that URI is certainly relevant to me.

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.

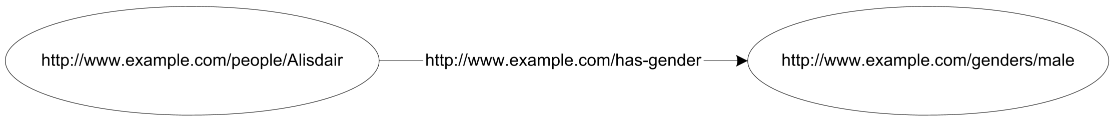

http://www.example.com/people/Alisdair
http://www.example.com/has-gender
http://www.example.com/genders/male

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.

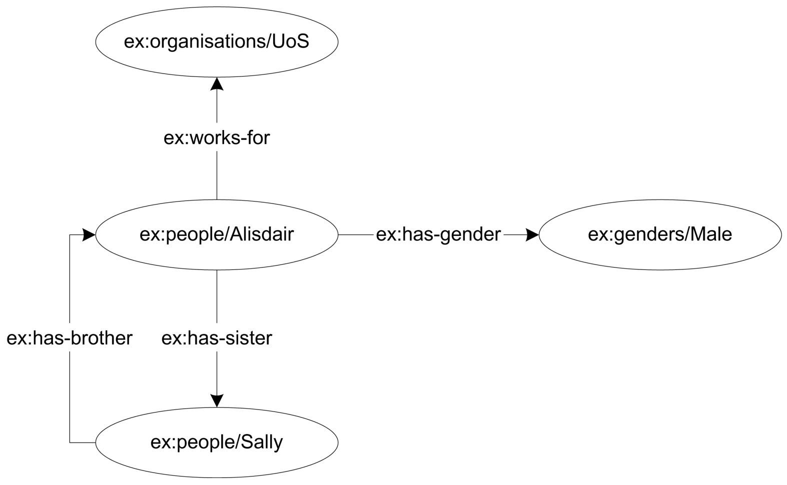


FIGURE 2.4: RDF Graph

RDF, then, offers 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 potentially 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 format, but as with any information, the topic of managing 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 data 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

While not the focus of this document, it is worthwhile to give a brief summary of the languages used to perform inference on the Semantic Web. 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 I 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 us to make statements about the class of objects that can be inserted as the subject and object of particular properties.

OWL adds much more wide ranging capabilities, aimed at providing computers with the ability to share not just information, but vocabulary (Patel-Schneider et al., 2003). 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 the 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 RDF-S and OWL ontologies is complex. Most RDF stores pre-compute much of the entailment of RDF-S data (forward chaining). This effectively determines all the new facts that might be determined by inference 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 likely to be 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. There are a variety of query specifications 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, 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.

This 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, although there are of course complications that aid usability, such as the ability to specify some parts of the pattern as optional, and the ability to order the results. In general, though, SPARQL is a relatively simple language when compared to Sructured Query Language (SQL), the equivalent in the world of RDBMSs.

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

In any database system, data is stored 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, and relates this knowledge back to the RDF data model as described in Section 2.2, asking 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 (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 data 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 modification of 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 data is physically stored: that is, the DBMS may choose to lay the data down on disk 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 operations that return a large number of results: queries that perform operations 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 2.7.

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 individual's 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 2.7: 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. This object stores certain data, and has 'methods' that can modify or retrieve that data. Object-based DBMS have amassed a body of criticism (Date, 1990) due to 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. We will not consider this 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 a 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 allows arbitrary data assertion and retrieval, assuming it conforms to this 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 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, etc. (Date, 1990; Stonebraker et al., 1976)

While the requirement for strict schema definition is usually helpful in traditional database environments, the situation regarding RDF storage is rather different: it is explicitly 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, this generates difficulties in terms of 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).

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 massively slower than doing so through the predicted routes. 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 URIs and literals, with the triple table itself storing 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. In addition, there are several other 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 relatively uncommon to search for the predicate that connects two concepts (Seaborne, Andy, personal communication, August 2008).
- RDF typically has a large number of data points (triples) relative to the physical size of the data.

An attempt to implement a more descriptive schema that adapted to the structure of the data was attempted 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.

	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
Berkeley DB	Application support	Key/value pairs	Unknown level of query predictability, relatively simplistic query support

Table 2.1: Database model comparison

The problem of RDF storage is important to the success of the Semantic Web. If we are to expect individuals or organisations 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 data. If we wish to create interfaces on to RDF data that are suitable for human users, we must maintain the interactive performance to which they have become accustomed.

Chapter 3

Exploration of the Problem Domain

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 datums in a performant manner is a difficult problem.

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 future work 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 to achieve high performance.
- Section 3.3 tightens the focus to indexing algorithms. 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.
- Finally, Section 3.6 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 over a terabyte 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 60MB/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 one simple instruction per cycle, thirty million instructions could be executed in the time it takes to perform a *single* disk seek. Upcoming solid state disk designs are less capacious, but by comparison feature virtually negligible 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. 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. While the bandwidth of RAM is very high, latency can be in excess of 200 processor cycles, making it impractical for modern processors to wait for RAM 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 simple 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 two instructions per cycle on certain workloads (Boncz et al., 2005) - or in the order of six 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).

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' will be transferred from memory at once, which on modern systems is usually 64 or 128 bytes, making subsequent accesses to adjacent data especially fast (Harizopoulos et al., 2006).

Typical CPUs have Level 1 (L1) and Level 2 (L2) caches (some extending to even more). 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). As long as data and instruction flow is sufficiently predictable, or occurs over a sufficiently small set of data, the 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. If the processor knows which 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 this 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, 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

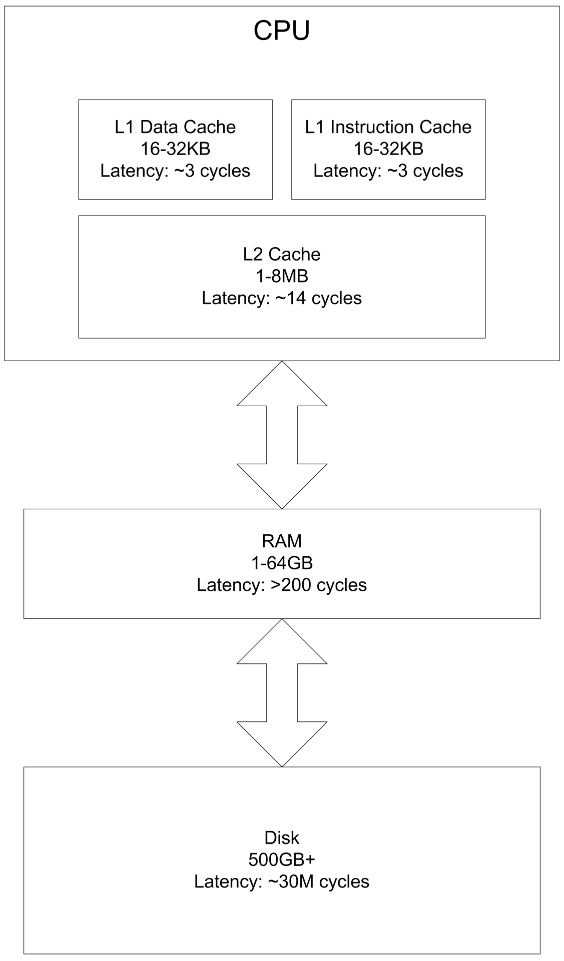


FIGURE 3.1: Data storage hierarchy.

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).

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 Figure 3.2 and Table 3.1, where the author

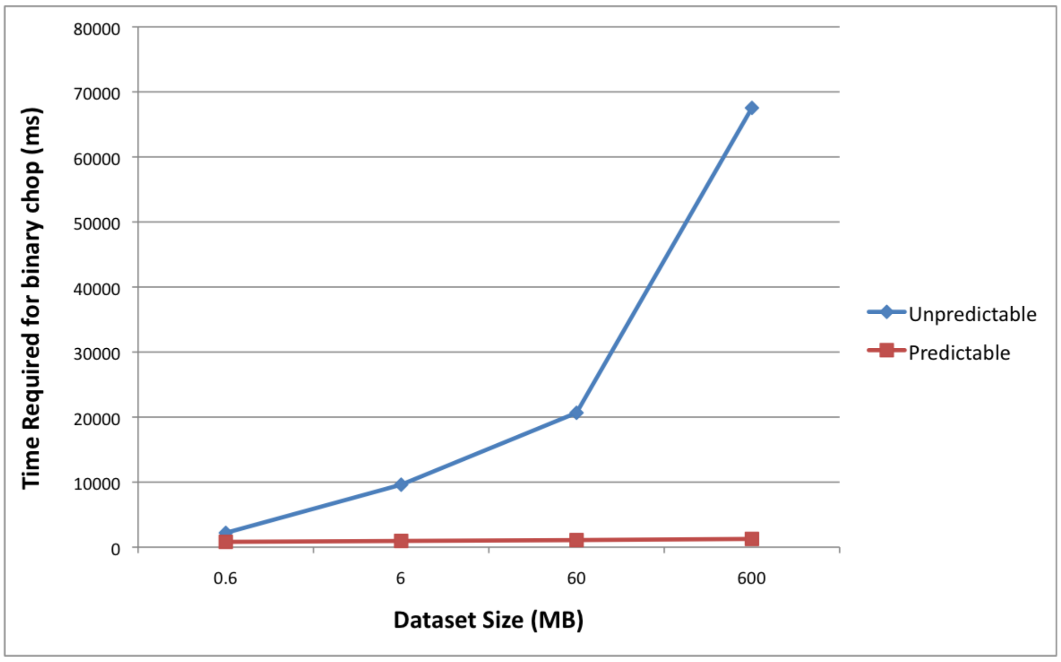


FIGURE 3.2: Cost of Binary Chop as Dataset Increases in Size

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 B

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 recently started to 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, became insufficient, 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.

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

3.1.5 Summary

This section provides 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 nonsequential 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 us with a given 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 64-128 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? Are we looking 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.

Optimising for writes in this fashion can have a significant impact on read performance, however, which is of great performance 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 still 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 piece 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 datums (Abadi et al., 2006).

Both read and write oriented stores may make use of 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 processes such as Lempel-Ziv compression (Abadi et al., 2006), 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.

In Abadi et al. (2006), the authors note that the ultimate 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.

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 contains a hash value, the actual text of which is discovered by joining to another 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 quad 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 well used 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.

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. Indexing is examined in detail in Section 3.3.

3.2.2.1 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 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., 2008b). 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 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. This means that 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 Paradox.

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 general, most RDF stores use hash-based IDs, but this decision would require review in mission-critical systems.

3.2.2.2 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, particularly while under query load, can exhibit much poorer performance.

Deletions offer their own difficulties. In an RDF-only store there is little computational difficulty in eliminating a statement from the system, but recovering the resources it has used is a different matter. 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 (2003) 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 is an example of a relational system that has RDF-specific modifications, and performs extremely well.

Normalisation generally offers a significant performance improvement over storing a triple table in lexical form. 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.

DBMSs researchers are finding that compression can provide a significant performance benefit in disk based systems. 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 data structure that, for a column or set of columns, quickly returns the location of specified datums 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 algorithms, 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 3.5, 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.

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, AVL, Treap and Splay trees. 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

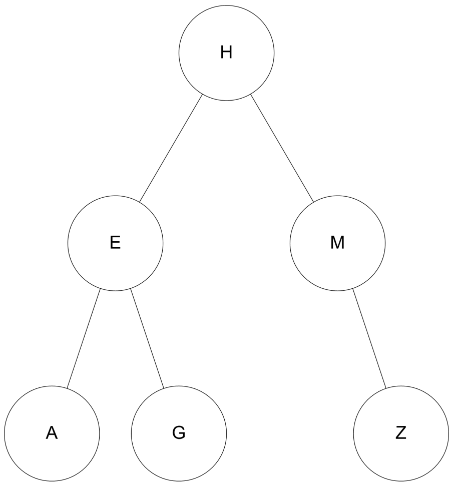


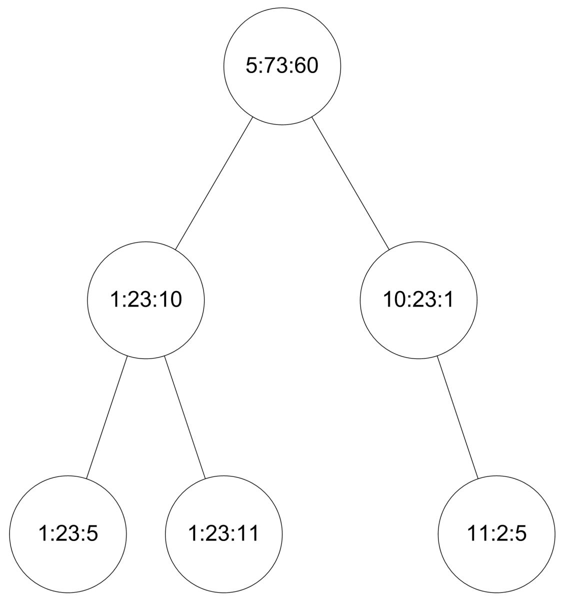
Figure 3.5: Balanced Binary Search Tree

datum. Algorithms 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 generally short retrieval times, high performance 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.

Figure 3.6 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. To produce a truly comprehensive set of indexes using this method, $N!$ indexes are required, where N is the number of columns. Typically, however, a triple store will just use three indexes: Subject-Predicate-Object (SPO), Predicate-Object-Subject (POS), 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.



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 3.6: RDF stored using a BST

Since RDF stores typically have very small node sizes, BSTs cannot be considered 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. This is a space efficiency of just 60%, 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 mostly 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 disk 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.

This format is particularly 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 particular attention to the latency issues that hard disks experience.

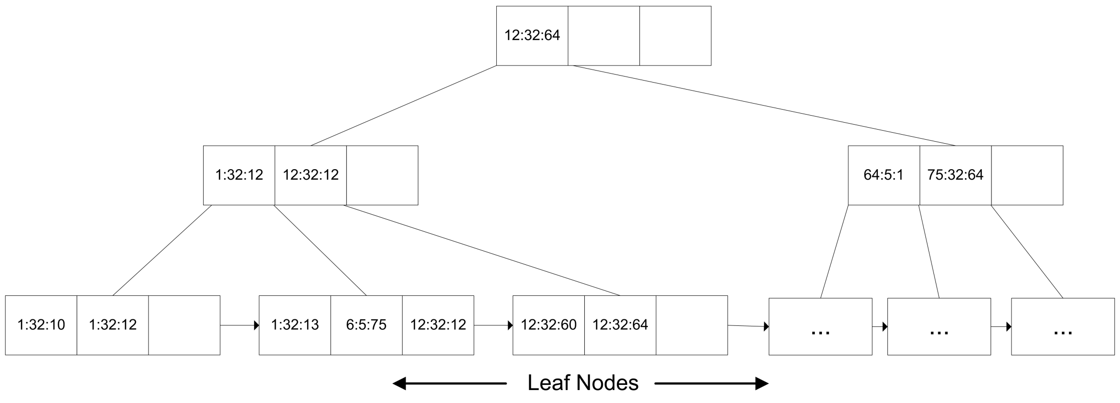


FIGURE 3.7: RDF IDs stored using a B$^+$tree

The B$^+$ variant of this tree (depicted in Figure 3.7) 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 is 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! Another artifact of this 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 algorithm 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., 2008b) 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. Smaller fanouts tend to be more CPU-cache friendly: if cache prefetching can be brought to bear, a small multiple of the size of a cache line offers optimal query performance in memory.

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.

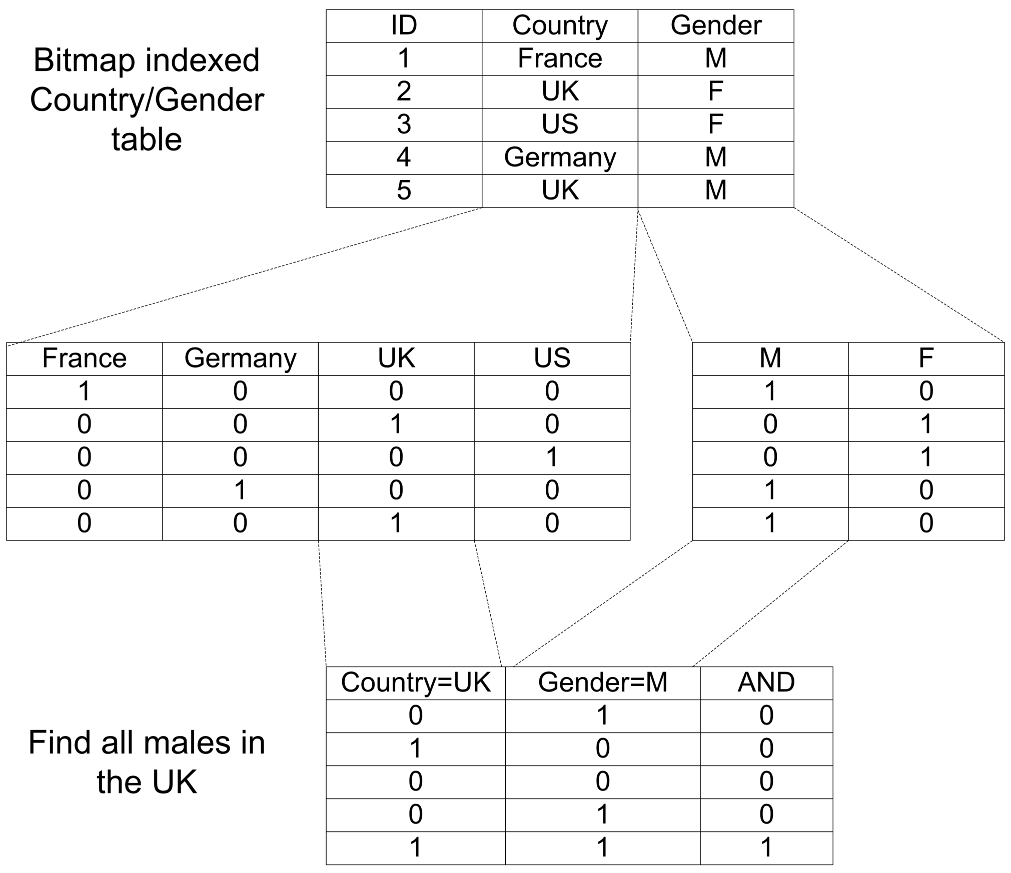
It should be noted, however, that contrary to conventional wisdom B-trees can offer better performance than binary trees for in-memory indexing. If each datum held in a B-tree node is sufficiently small, a B-tree node may hold several, including pointers, in a single cache line. Doing extra processing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

3.3.3 Bitmaps

Bitmap indexes are popular for applications which require extremely high read performance. 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 field 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/hasgender> 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. Bitmap processing 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 Figure 3.8.



ID	Country	Gender
1	France	M
2	UK	F
3	US	F
4	Germany	M
5	UK	M

France	Germany	UK	US
1	0	0	0
0	0	1	0
0	0	0	1
0	1	0	0
0	0	1	0

M	F
1	0
0	1
0	1
1	0
1	0

Country=UK	Gender=M	AND
0	1	0
1	0	0
0	0	0
0	1	0
1	1	1

FIGURE 3.8: Querying using a bitmap index

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 excellent 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.

From the point of view of disk-based RDF stores, Virtuoso has demonstrated that bitmap indexes can produce excellent results (Erling, 2006). It should be noted, however, 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 comprehensive composite tree indexes used in systems such as Jena TDB (Owens et al., 2008b) that can encode all data within the index, and may significantly impact overall performance.

3.3.4 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 do not require any comparisons, the problem of unpredictable branches is 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, it is often impractical to store indexes in RAM, and as soon as the index does drop out of memory, hashes demonstrate 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, particularly when indexing outside of memory, 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 support 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.

From the point of view of RDF/SPARQL, which rarely utilise 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, Andy, personal communication, February 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.

3.3.5 Space Filling Curves

A space filling curve (SFC) is 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 3.9.

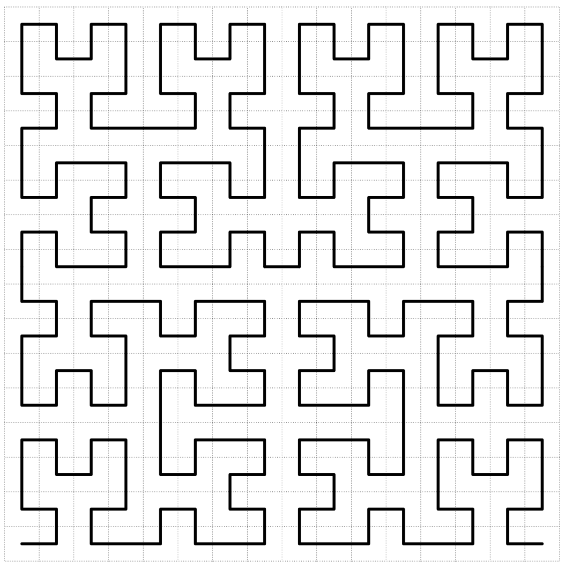


FIGURE 3.9: The two dimensional Hilbert curve

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.

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

Indexing of this structure can occur through a tree-based system in curves such as the Hilbert curve (Lawder and King, 2000). The repeating structure is evidenced at every level of construction of the curve, 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 makes SFC-based indexes vastly 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 datums somewhat close to each other 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 higher proportion of non-contiguous accesses than indexes with a single dimension: this is particularly important for disk storage, where the costs of a seek are crippling. Combined with potentially high costs for calculating the location of a datum, this makes SFC-based indexes an extremely space efficient but potentially slow solution, meriting further investigation for the purposes of high capacity memory stores.

3.3.6 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. Bitmap indexes have also proved effective for RDF storage, and although they typically will require more disk seeks than a dedicated B-tree, they have the advantage of scaling much more effectively to larger numbers of attributes.

For memory-based RDF systems, trees in general are not a good solution. While they provide strong guarantees as regards sorted order, this is more than is required for RDF, which does not generally require in-order traversal. Trees generally waste too much space in pointers and/or empty space, and offer very poor characteristics with regards to contiguity of access. Hash indexes are generally more appropriate in this scenario, as they offer $O(1)$ retrieval and update, and explicit management of large blocks is not required in memory. Hash indexes bring with them their own issues. Care must be taken to ensure efficient use of space when creating hash indexes, and it should be noted that hash indexes do not support prefixes.

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 in sorted order, 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 specifying 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 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 vast 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.

3.4.2 Types of Join

Joining can be an expensive operation, involving as it does two different tables. 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 of $O(n^2)$ with the size of the data being examined, to more useful techniques, such as 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 practical for small datasets.

Since RDF stores will potentially be working with extremely large tables, this naive approach cannot be considered advisable. A more commonly utilised solution is the index nested loop join. In this join, if an index is available on the inner join input, the index is consulted for matches against each row of the outer join input.

Index nested loop joins require very little memory, and are highly effective for some situations: if the outer join input is small and is being matched against a very large inner join input, the selectivity of the index is brought to bear, returning only relevant results and thus reducing computation. 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.

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 the fastest join 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 clearly 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 on disk. 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 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.

3.4.4 Summary

RDF does not offer an unusual problem with regards to operator implementation, and the usual rules of thumb for which join to use apply equally well to RDF. Thanks to the sheer quantity of data points in a typical RDF store, however, RDF does require a special emphasis on minimising the time spent in joins. This can be achieved by a combination of 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. Since it is difficult 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 buy 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 very 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 into prior distributed systems 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 those stores 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.

The differences between these two 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

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

this document, 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 algorithms (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 seen in 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. It should be noted that typically, when reading or writing very small amounts of data, it is desirable to perform the work on one machine. This is because the setup costs will dwarf any advantages gained from partitioning. 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.

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 likely be accessing different pieces of information, so hardware resources can be shared between them and the queries run in parallel. Multi-user systems can 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 disk location) 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 disks 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.

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. In practise, 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 amount of 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.
- 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, 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 and objects, each subject or object being used a relatively small number of times. This makes it advisable to keep all data on a subject or object in an SPO or OSP index on a single machine, as startup costs will remove any gains from increased parallelisation. Properties, 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 properties 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 property-ordered entries to a random server, and flooding all property-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 properties unnecessarily expensive.

In Owens et al. (2008b) we 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 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 P and O, so that queries that supply a property and object can still hit only one machine, while P-only requests gain the benefit of parallelisation. Implementation work is still proceeding on the prototype, but this approach should logically provide improved results.

Aside from these issues, RDF stores usually distribute effectively using traditional techniques. Since range searches are relatively unimportant in RDF query, 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. (2008b).

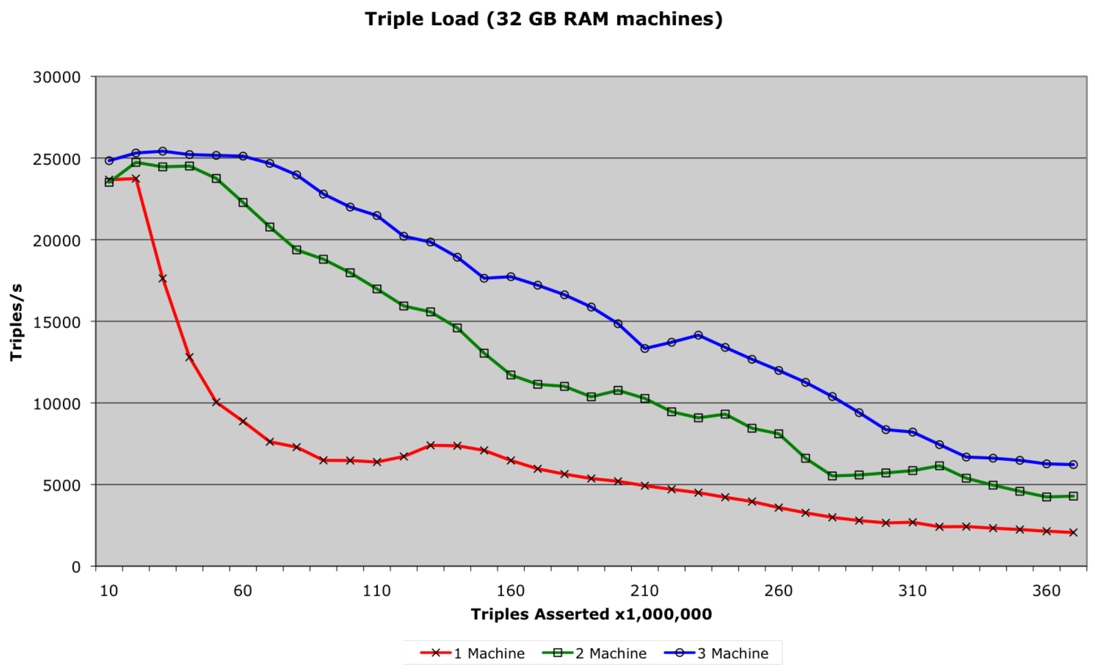


FIGURE 3.10: Rates of assertion during a Clustered TDB load

Figure 3.10 shows the scaling with regards to assertion rates for 1, 2, and 3 machine clusters for the work described in Owens et al. (2008b). 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 slightly different requirements to a disk based environment: in the latter case, the latency of the network is usually significantly 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. In a memory-only or heavily memory-based scenario, the cost of network access is vast compared to an access to the usual storage mechanism (Erling and Mikhailov, 2008). Given this fact, it becomes more important to avoid 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.

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. 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 Virtuoso Cluster (Erling and Mikhailov, 2008), YARS2 (Harth et al., 2007), and the author's own work described in this chapter.

In general, RDF generally 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. 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. It will become more critical to globally cache regularly accessed data, increasing update complexity and compromising linear scalability in the aid of better absolute performance.

3.6 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 generally by far the largest cause of waiting under normal 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 10 items in the outer join input, and a disk seek takes 10ms, the time taken 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 vastly significant 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. On a single machine system, there is an increased focus on efficient utilisation of the CPU and memory architecture, while in a distributed system the latency of the network suddenly becomes a much more important issue.

In a disk bound environment, poor 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 utilisation.

Systems with relatively low I/O latency enable a greater variety of strategies for physical representation. Indexes based on SFCs, for example, are vastly more practical in an environment with lower seek times. Adaptive index structures suited to RDF become plausible in a memory based environment that does not mandate the use of large blocks. Such index structures become particularly attractive since SSDs and RAM typically offer much smaller amounts of space to work with than conventional disks, being significantly more expensive per gigabyte. Space efficiency thus becomes a much greater priority.

Chapter 4

Measuring RDF Store Performance

It is clearly important to be able to show that a given store lives up to its performance claims, and that changes being made to a store actually result in overall performance improvements. This need is satisfied by benchmarks. This chapter describes the background of benchmarking in (O)RDBMSs and RDF stores, and the work performed by the author in this area. This serves as a means to determine the effectiveness of future work.

When considering the benchmarking of RDF stores, or DBMSs in general, there are two schools of thought with regards to how to perform the task. The pre-eminent benchmark set in the DBMS world, that produced by the Transaction Processing Performance Council (TPC) performs a set of high level queries based on a given use case. Benchmarks exist with use cases based around OLTP (described in Council (2001)), and decision support workloads. When a new workload type becomes popular, a new benchmark is added.

These tests produce simple figures for total throughput (overall performance) and throughput per unit cost of the machine. Developers running the tests are expected to provide detailed reports of the hardware upon which the tests were run.

TPC-style benchmarks provide a convenient standard for judging the overall performance of a system, but do not make an effort to individually test the components of that system (DeWitt, 1991), or give a report of what exactly it is that makes the system fast or slow. This is in some ways advantageous, since it simplifies the process of comparison, but does not give much information to those who require information about specific performance characteristics.

From the point of view of the system developer, it is important to be able to investigate the performance of different aspects of the store. This allows the discovery of where most time is spent, and thus enables the targeting of optimisation: even if an improvement is possible in a particular subsystem, it is important to know that sufficient time is spent in that subsystem to make an improvement worthwhile. The TPC family of benchmarks do not provide a sufficiently fine grain to allow such analysis.

An earlier RDBMS benchmark, the Wisconsin benchmark, provides such detailed tests. This benchmark performs a large number of transactions in an attempt to test each subcomponent of the system, such as the query optimiser, join mechanism, and so on. This approach gives detailed feedback regarding the performance characteristics of the DBMS, in particular the intelligence of the query optimiser (Cattell, 1991). It attracted criticism for not reducing the reported figures to a single overall metric (DeWitt, 1991), rendering results harder to interpret and not giving a clear winner when comparing two DBMSs.

TPC-style benchmarks provide a clearer statement of overall performance than the Wisconsin benchmark. Since this is what the majority of clients are interested in for making purchasing decisions, Wisconsin-style benchmarks ultimately proved less popular. Detailed tests, however, are invaluable to developers, and so in Owens et al. (2008a) we describe the creation of a detailed, Wisconsin-style benchmark (including testing software) for RDF stores. This work will provide a foundation for analysing the output produced over the next year.

4.1 Existing RDF Benchmarks

There are a variety of benchmarks currently in existence that are commonly used to test RDF stores. As noted, the Lehigh University Benchmark (Guo et al., 2005)is the most popular of these. LUBM allows the creation of an arbitrarily large amount of RDF data based on a variable number of iterations over a simple OWL ontology. The number of properties attached to any class instance is varied by the use of simple bounded randomisation.

While this benchmark is effective for the purposes of testing OWL inference performance, it is not designed for the purpose it is often used for currently: testing stores based on their RDF query performance. The data produced has a small, heavily repeated structure with few predicates (Weithoner et al., 2006).

The method LUBM uses for querying the stores does not reflect likely use cases for RDF stores: each query is repeated ten times, with the average response time being the result (Guo et al., 2005). This mechanism gives query caches a large influence over the final result. Further, one query is performed at a time, with no provision for concurrent access. This testing mechanism does not accurately reflect the reality of an open data node on the Semantic Web, where a system might expect to be processing many highly unpredictable queries concurrently, potentially requiring the ability to perform updates while under query load. Finally, it should be noted that LUBM is somewhat out of date, and does not test all of the features (such as regular expression object matching) available in SPARQL.

Alternative datasets to that provided by LUBM have emerged. Some of these utilise real world information, such as the UNIPROT and DBpedia sets. While these have the advantage of providing data that is realistic, they are of limited size and do not provide the ability to easily scale the datasets involved.

During the course of the author's work in this area, a new use case-based benchmark for RDF stores was released: the Berlin SPARQL Benchmark (BSBM)[1]. This offers a more comprehensive query set than LUBM, and also tests multi-user scenarios. Explanation is given of the rationale behind each query, and it offers useful overall results. Further, it makes the effort to make minor alterations to SPARQL queries during query repetition, reducing unrealistic caching effects.

Despite these advantages, certain features are missing in BSBM. Little study is made of the effect of changing the dataset, due to its fixed schema. Simple changes such as the sort order of data being input, or the quantity of properties, can have a significant impact on assertion performance. There is no study of deletions, assertions beyond the initial bulk assert, or other features such as multi-user assertions or assertion while under query load. In addition, since the benchmark is use-case based, there is no incremental change in queries: more than one factor is changed at a time between query sets.

These newer test sets can be expected to fill important roles in offering a wider variety of tests of clear format, testing a greater proportion of the features available in SPARQL than are seen in LUBM. It should be noted, however, that they cannot be expected to provide a highly detailed assessment of the stores that they test: they offer a limited number of queries, and a limited structure to their datasets.

4.2 A New RDF Test Set

In designing a new RDF test set, a large number of useful test cases were developed. The entire set of these can be found in Appendix A. These test cases were then applied to an automatically generated dataset, using a set of test scripts that simulated a given number of users. The scripts subsequently generated useful statistics regarding test runtime, average CPU utilisation and average IO utilisation.

Data generation in RDF benchmarks is usually performed by taking a given data schema (or ontology) and generating an amount of data based on a 'scale factor'. The work

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

described in this chapter proceeded along similar lines, with the exception that it was possible to alter the schema for the data. This offers the benefit of testing stores over a variety of different graph shapes, as illustrated in Figure 4.1. An interesting result occurred as a result of this method of data generation: The assertion time of one of the three systems under test, Virtuoso, was affected by the sort order produced by the script, such that it was impossible to complete tests. On the other hand, AllegroGraph and BigOWLIM (the other stores under examination) were able to cope with it.

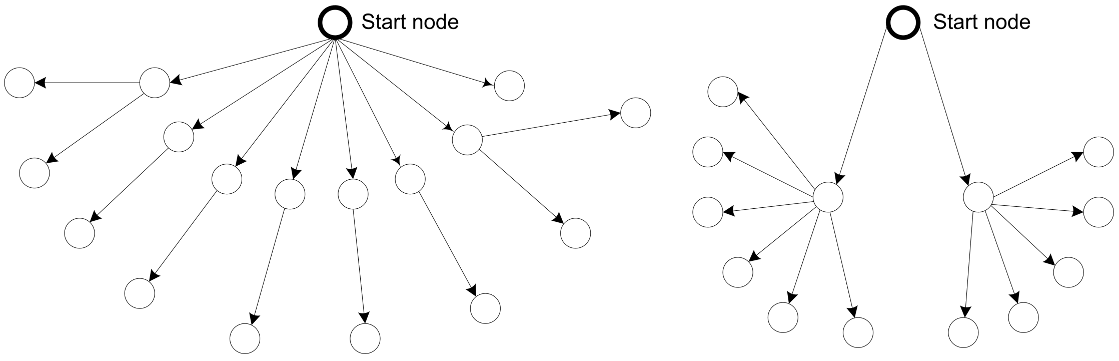


Figure 4.1: Configurable RDF Graph Shapes

The benchmarks showed several more interesting results that might not be clearly exposed by TPC-style benchmarks: BigOWLIM performed very poorly when retrieving data based on a known object. This exposes a relatively slow index over objects. AllegroGraph, which takes advantage of 'inlining' (see Section 3.2.2.1), was capable of answering FILTER queries over a huge number of integers extremely quickly, while BigOWLIM was unable to produce a result. On the other hand, BigOWLIM produced superior assertion times, particularly in situations where concurrent writes were being performed. Full results are available in Owens et al. (2008a).

This test set will be useful in the development of an RDF store, and in the author's proposed future work, as it informs the development process: it is possible to quickly determine how even small design changes affect a particular subsystem.

4.3 A Use-Case Based Test

The mSpace (schraefel et al., 2005) browser provides an excellent real world use case for a large scale RDF application that requires extremely low latency queries. mSpace relies on AJAX to provide a highly interactive system that users expect to respond quickly. The author developed an automatic test system for the purposes of comparing mSpace's performance under 3Store and MySQL in Smith et al. (2007). In this case, it was found that the RDF store's performance lagged far behind a MySQL fixed-schema implementation, a trend that has continued according to the latest BSBM result set.

The mSpace test system does not yet offer the comprehensiveness of either BSBM or the system described in Section 4.2, but it does accurately model a real world situation: a variable number of users are simulated browsing an mSpace system, with configurable average time between clicks, and statistics produced on average query time, CPU utilisation, and I/O utilisation. The test system understands how to browse an mSpace data repository, so little configuration is required, and the dataset can be changed with ease. The test sets are recorded for the sake of repeatability.

Chapter 5

Future Work

There is the potential for a great deal of work in the area of RDF storage. There are a host of use cases: federated systems aggregating a large quantity of data from related institutions, clustered stores designed to maximise performance from a single endpoint, low latency stores designed for highly interactive applications, and smaller stores that fill a role similar to that taken in the RDBMS world by such products as MySQL and Postgres, operating highly efficiently on limited resources. These stores might be read or write optimised: a knowledge repository might often be full of virtually static information, but a store might also be implemented as support for an interactive application, or as a local cache of knowledge on an individual's computer, with information likely to change regularly.

There is a clear need for high-performance RDF storage and retrieval. Recent events such as the billion triples challenge at ISWC 2008 have highlighted the demand for extremely scalable systems, and the scalability of disk-based RDF stores has improved significantly over time. While early systems such as 3store experience limits around the 200 million triple mark, even with access to improved hardware, more advanced stores such as Virtuoso, Jena TDB and BigOWLIM are capable of handling well over a billion triples. These systems are founded on techniques that are familiar from the world of (O)RDBMSs, modified to take into account the special needs of RDF.

Less obviously prominent, yet of significant importance, is a demand for extremely performant, low latency systems to drive flexible RDF-based UIs such as mSpace (Smith et al., 2007). 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 this kind of store 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.

Main memory RDF stores present an attractive alternative for systems such as mSpace. They offer much higher performance over datasets that will fit in RAM, and with clustering can be expanded almost indefinitely. In the enterprise, there is a trend towards main memory DBMSs as RAM prices fall: enterprise systems commonly contain 32-64GB of RAM (Seaborne, Andy, personal communication, August 2008), which makes main memory DBMSs practical for an increasingly large variety of tasks. 32GB of RAM is sufficient to store 200 million statements in the SwiftOWLIM (Kiryakov et al., 2005) memory store, and better space utilisation could lead to even larger capacity. This represents sufficient storage space for current mSpace workloads.

For the remainder of the PhD program, the author proposes to investigate main memory RDF stores as a solution for systems that require extremely performant data access, such as semantic UIs and backward chaining inference engines. This work will be carried out in the context of a performance analysis of the mSpace browser, with the aim of making it practical for such systems to operate at scale on RDF stores. The focus of this work will be in-memory representation, and managing the space/performance tradeoff, the concerns for each of which are described in Sections 5.1 and 5.2. The prototype should be designed such that it is feasible to extend into a distributed system, to enable scalability to even larger datasets.

5.1 Performance

There has been relatively little work on improving the performance of main memory RDF stores. Thanks to changing processor architectures and the increasing importance of processor caches, the model of RAM as purely random access is no longer truly valid. Work in the DBMS field has highlighted the increasing gap between the performance that modern processors are theoretically capable of providing, and the performance that is actually delivered by DBMSs (Ailamaki et al., 1999). Much of this disparity can be traced to two issues: inefficient use of processor caches, and architectures that are not aware of the demands of modern superscalar, pipelined processors. As a result, even relatively recent DBMSs can spend less than half of their time performing actual useful computation. (Ailamaki et al., 1999).

As noted in Section 3.2, cache conscious algorithms and the minimisation of unpredictable branches can result in significantly improved performance for common database workloads, and significantly reduce wasted CPU time. This work will attempt the implementation of cache-friendly indexing mechanisms, while paying attention to storage space requirements. As part of this work, it will be necessary to characterise the performance of a system like Jena in order to discover where most time is spent, such that optimisations are not mistargeted.

5.2 Storage Capacity

Since memory is a much more expensive (and limited) commodity than disk space, it becomes important to minimise the space required by any physical representation. It is potentially challenging to optimise for both storage capacity and high query performance: the ability to store multiple different physical representations of the same data, for example, will improve query performance, at the cost of storage capacity. Similarly, application of compression, popular in disk-based DBMSs for reducing I/O costs and storage footprint (Stonebraker et al., 2005), can result in increased CPU usage and reduced performance in memory stores. A significant portion of this work will be examining the tradeoff between storage capacity and performance.

Fortunately, these concerns are not always in opposition. Cache-friendly algorithms, for example, favour the reduction of data footprint as a way to fit more information on a cache line. The smaller the working set of a given application, the more likely it is to fit into L2 cache, a critical issue for maintaining performance. Smaller working sets also improve utilisation of the Translation Lookaside Buffer, a CPU cache that is important for maintaining the performance of main memory access.

5.3 DBMSs on a Virtual Machine

The two major Semantic Web frameworks, Jena and Sesame, are both written in Java. While it is possible to interface to other languages using Java Native Invocation (JNI), this is complex and has a relatively high overhead (Rao and Ross, 2000), making its suitability for our goals questionable. This means that any new storage and indexing system is likely to be written in Java.

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 early experimentation performed as a test of Java's branch prediction and cache performance. When performing a binary chop of varying predictability over an extremely large array, as illustrated in Table 5.1 and Table 5.1, 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 B

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

Table 5.1: 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 5.2: Comparison of Java and C on a predictable large scale binary chop

In addition, Java has very high space overheads in certain situations: each object has an overhead: in a typical VM, this might be in the order of two words. If a program produces a lot of small objects, as is the case with existing RDF memory stores, space efficiency can be significantly impaired. Paying attention to this issue can result in significantly reduced overhead.

Little study has been made of the details of implementing a DBMS on a virtual machine-based language like Java. A major contribution of this work will be an analysis of the differences between implementing a physical representation on a virtual machine-based language versus a compiled one.

5.4 Work Packages

This section describes the packages of work that will be performed up to the end of this PhD program. The work packages will focus upon the development and testing of the ideas mentioned in this chapter. It is expected that there will be a significant need for prototype-level testing of these ideas, for which it is proposed to build upon the Jena framework, a commonly used standard with which the author is already familiar.

5.4.1 WP1 - Characterisation of the Jena Framework

Before commencing the work, it is important to ensure that it will provide a noticeable benefit. While it has been clearly established that a high performance physical representation is critical in a disk-based system, a study has not been made of memory stores to show the same. This work package will examine in detail the Jena system when backed by a memory store, in order to determine where time is spent, and whether an improved memory store would actually result in significantly better performance.

Tasks (4 weeks total)

- Discovery of and familiarisation with profiling tools (1 week)
  Tools suitable for performance profiling with Java will be determined, installed, and time will be spend gaining familiarity with them.
- Determination of appropriate testing strategies (1 week)
  Existing mechanisms for testing RDF stores will be briefly surveyed, and an appropriate one chosen for examining Jena's performance.
- Profiling of Jena (2 weeks)
  Time will be spent profiling the Jena system under load, in order to determine where most time is spent.

5.4.2 WP2 - Investigation into Coding for a Virtual Machine

This work package will further the author's existing knowledge in the area of high performance coding for virtual machine based languages. While virtual machines can clearly differ between implementations, high performance virtual machines generally share many common characteristics.

The work will contribute an evaluation of the differences that are relevant to DBMSs between coding for virtual machines and compiled languages. This is a significant contribution, given that historically DBMSs have been coded in low level languages, and existing literature focuses on how to optimise in those languages. While an effort will be made to make this study relevant to a broad cross-section of virtual machine-based languages, the focus will be on Sun's reference implementation for Linux. In this work package, small-scale evaluations will be performed in an attempt to confirm expected results.

Tasks (6 weeks total)

- Literature Search (3 weeks)
  Time will be spent reviewing literature on virtual machines (particularly Java), and on code optimisation for such machines, to reinforce the author's existing knowledge.
- Tests (3 weeks)
  Tests will be performed, where feasible, to confirm the knowledge discovered during the literature review.

5.4.3 WP3 - Design and Implementation of Prototype(s)

Work package 3 focuses on the design and creation of the prototype. It is expected that this prototype ought to able to achieve significant improvements in space utilisation and/or performance. In the worst case, a minimum of 10% improvement in space efficiency is desired, but the expected result is an additional 20% improvement in query latency over the existing Jena memory store, along with increased space efficiency. This improvement parallels that found in Rao and Ross (2000). The results of this work package should also be of interest when considering the implementation of a physical representation on an SSD, since these share RAM's relatively low latency.

Tasks (22 weeks total)

- System design (4 weeks)
  The author will build upon existing thoughts on the design for the system, in order to produce an overall system design with the flexibility to be adjusted for space utilisation or performance.
- Prototype Implementation (18 weeks)
  The prototype will be implemented. If time is available, a more basic prototype in a low level language such as C will be produced for the purposes of comparison, and furthering the contribution of WP2.

5.4.4 WP4 - Testing Against mSpace

This work package performs validation of the prototype, and as such will at some points be executed in parallel with WP3. Previous work aimed at performance testing the mSpace RDF browser in a multi-user situation will be reused and updated in order to determine the effectiveness of the prototype, and the results will be compared to existing RDF memory stores, in particular the standard Jena implementation.

Tasks (8 weeks total)

- Updating mSpace benchmarks (2 weeks)
  Time will be spent updating the author's existing mSpace benchmarks in order to ensure that they provide a reasonably realistic user simulation, and that they can interface correctly with the prototype.
- Tests over RDF memory stores (6 weeks)
  Tests will be run over the prototype and selected RDF memory stores (in particular the existing Jena memory backend) in order to validate the prototype. Profiling will also be performed in order to show where gains are being made.

Chapter 6

Conclusions

The Semantic Web offers the potential to vastly 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 report.

This document has described in detail the author's work in the area of RDF storage and query, with a particular focus on physical representation and indexing, and getting the most out of modern computer architectures. It offers several contributions: the report contains a detailed insight into the relationship between traditional RDBMSs and RDF stores, including an explanation of the applicability of techniques from the DBMS world to the problem of storing RDF, and an analysis of where new work is required. This is used to determine and inform the author's future work: in-memory RDF storage structures that will show that application of knowledge regarding the underlying machine architecture can provide significant improvements in both space efficiency and performance of RDF stores.

In addition, the report discusses work performed by the author in the area of clustered RDF storage and RDF store benchmarking, including prototype implementations of each. The work on benchmarking will be used to test and develop the author's research into in-memory RDF storage, offering as it does the ability to break down the performance of different components of the tested systems in a manner not supported by existing benchmarks. Further details on each of these areas of research can be found in Owens et al. (2008a), Owens et al. (2008b) and Smith et al. (2007).

The future work described in this report provides an avenue for significant new research that will benefit the Semantic Web community. High performance memory stores have received relatively little attention, 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 future work will provide significant improvements in performance and storage efficiency of RDF memory stores, thanks to attention being paid to optimisation for modern architectures and virtual machines. Little study has previously been made of implementing a DBMS on a virtual machine, leading to more than one significant contribution from this work.

Appendix A

Test Cases

This appendix details the tests proposed for the benchmarking system described in Chapter 4.

Assertion:

- Bulk assertion time from scratch: this metric tests the stores performance when asserting a new data graph (including creation of appropriate indexes), assuming that the store is under no other activity.
- Assertion time for a significant addition to a previously existing data graph.
- Assertion time for multiple simultaneous writes: This examines the stores locking mechanisms: some schemes require a greater or lesser portion of indexes to be locked to perform an update, which can affect read/write times.
- Assertion time when operating under query load: This again tests locking mechanisms.

Deletion:

- Deletion of entire graphs: most stores offer the ability to create multiple graphs, or models, to separate distinct datasets. If the store does not perform cross-model inference, deletion of a model and its associated statements ought to be simple.
- Deletion of statements: deletion of individual (or patterns of) statements ought to be a relatively simple process for an RDF-only store. If the store produces forward chained entailments for RDF-S or OWL, it becomes a complex issue due to the need to determine how the inferred statements need to be altered (Broekstra and Kampman, 2003).

Query:

- Simple queries that match against a specified subject, predicate, or object (or pair of these), returning a moderate number of results. This provides baseline information, as well as testing storage/indexing methodology: many stores will be indexed and sorted on subjects, and may exhibit decreased performance when testing over predicate or object. The choice of relatively disconnected URIs keeps the result set small for this simple query.
- Repeat of the above, using queries that return large numbers of results. This examines the effect of the number of results returned on query performance, which can be a particular significant bottleneck in stores that internally use id/hash mappings to describe URIs and literals.
- Queries that specify multi-triple graph patterns. These patterns should provide a large capacity for optimisation: one triple should result in the retrieval of very few results compared to the others. This tests the query optimisers intelligence.
- Repeat of the above, specifying the triples in another order. If the same queries are to be used, this test should be run some time after the above, to reduce any effect of caching.
- Queries that specify multi-triple graph patterns, returning many results, not amenable to optimisation, that is, each graph pattern returns many results: this is a test of the stores ability to perform joins over large quantities of data.
- Complex queries that specify many triple patterns. This tests join performance and query optimisation.
- Performance of the system in an environment where it experiences multiple simultaneous queries: does it degrade in performance gracefully as the load increases, or not?
- Queries that return no result. This again tests the query optimisers ability to perform operations in an optimal manner: if the triple that causes no result to be found is run early, this should return very quickly.
- The effect of specifying OPTIONAL sections within a query.
- The effect of the use of the REGEX expression when searching for a range of object values within a result set.
- Performance of numerically restrictive FILTER statements.
- Performance of LIMITed queries versus their non-LIMITed equivalents, and the time taken to retrieve subsequent ranges of information from previously run LIMITed queries.

- Performance of cached queries versus their non-cached equivalents.

Appendix B

Binary Chop Tests

This appendix contains the code for both the Java and C implementations for the tests described in Section 5.3.

B.1 Java Implementation

Run using: java -server -Xmx2048M <filename>

```
public class TestClass {
        final static int size = 150000000;
        final static int iterations = 10000000;

        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();
        }
}
```

B.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 10000000
#define STEP 10
```

```c
#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
```

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 New York, NY, USA, 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.

A. Ailamaki, D.J. DeWitt, M.D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proc. VLDB*, 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 International Conference on Very Large Databases*, pages 266–277, 1999.

DW Anderson, FJ Sparacio, and RM Tomasulo. The IBM System/360 Model 91- machine philosophy and instruction-handling(ibm system/360 model 91 machine organization alleviating disparity between storage time and circuit speed). *IBM Journal of Research and Development*, 11:8–24, 1967.

Malcolm Atkinson, David Dewitt, David Maier, Francois Bancilhon, Klaus Dittrich, and Stanley Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, volume 57, 1989.

D. Beckett. The design and implementation of the Redland RDF application framework. *Computer Networks*, 39(5):577–588, 2002.

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 Int. Semantic Web User Interaction Workshop, Athens, USA*, 2006.

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 at the 5th International Semantic Web Conference*, 2006.

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2004. ACM. ISBN 1-58113-873-3.

P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *Proc. CIDR*, 2005.

H. Boral. Parallelism in bubba. In *Databases in Parallel and Distributed Systems, 1988. Proceedings. International Symposium on*, 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.

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 First International Workshop on Practical and Scalable Semantic Systems*, 2003.

J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An architecture for storing and querying RDF data and schema information. *Spinning the Semantic Web*, pages 197–222, 2003.

J.J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *International World Wide Web Conference*, pages 74–83. ACM Press New York, NY, USA, 2004.

RGG Cattell. The engineering database benchmark. *The Benchmark Handbook for Database and Transaction Processing Systems*, 1991.

S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

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 (CSUR)*, 11(2):121–137, 1979.

T.P.P. Council. TPC Benchmark C, standard specification, revision 5.9. 2001.

C.J. Date. *An Introduction to Database Systems.* Reading, Mass.: Addison-Wesley Pub. Co., 1990.

D.J. DeWitt. The Wisconsin benchmark: Past, present, and future. *The Benchmark Handbook for Database and Transaction Processing Systems*, 1, 1991.

D.J. DeWitt and J. Gray. *Parallel Database Systems: The Future of High Performance Database Processing.* University of Wisconsin-Madison, Computer Sciences Dept., 1992.

Luping Ding, Luping Ding, Kevin Wilkinson, Kevin Wilkinson, Craig Sayers, Craig Sayers, Harumi Kuno, and Harumi Kuno. Application-specific schema design for storing large RDF datasets. In *First International Workshop on Practical and Scalable Semantic Systems*, 2003.

U. Drepper. What every programmer should know about memory, 2007.

O. Erling. Advances in Virtuoso RDF triple storage (bitmap indexing). 2006.

O. Erling and I. Mikhailov. RDF support in the virtuoso DBMS. 2006.

O. Erling and I. Mikhailov. Towards web scale RDF, 2008.

Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

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. *Lecture Notes in Computer Science*, pages 235–244, 2005.

A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. *Lecture Notes in Computer Science*, 4825:211, 2007.

P. Hawthorn and M. Stonebraker. The use of technological advances to enhance database system performance. *Addison-Wesley Series In Computer Science*, pages 106–130, 1986.

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 Sixteenth international conference on Very Large Databases*, pages 493–506, 1990.

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.

A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM - a pragmatic semantic repository for OWL. In *International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2005.

Bob Knighten. Detailed characterization of a quad pentium pro server running TPC-D. In *ICCD '99: Proceedings of the 1999 IEEE International Conference on Computer Design*, page 108, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0406-X.

O. Lassila, R.R. Swick, et al. Resource description framework (RDF) model and syntax specification. 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, pages 20–35. Springer, 2000.

R. Lee. Scalability report on triple store applications. *Massachusetts Institute of Technology*, 2004.

D. Lomet. The evolution of effective B-trees: page organization and techniques: a personal account. *ACM SIGMOD Record*, 30(3):64–69, 2001.

D. Ognyanoff, A. Kiryakov, R. Velkov, and M. Yankova. A scalable repository for massive semantic annotation. *SEKT Technical Report*, 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 Berkeley, CA, USA, 1999.

A. Owens, N. Gibbins, and m. c. schraefel. Effective benchmarking for RDF stores using synthetic datasets. Technical report, University of Southampton, 2008a.

A. Owens, A. Seaborne, N. Gibbins, and m. c. schraefel. Clustered TDB: A clustered triple store for Jena. Technical report, University of Southampton (Work conducted while the author was at HP Labs Bristol), 2008b.

P.F. Patel-Schneider, P. Hayes, I. Horrocks, and F. van Harmelen. OWL web ontology language; semantics and abstract syntax, W3C candidate recommendation. *World Wide Web Consortium*, 2003.

M. Poess and D. Potapov. Data compression in oracle. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 937–947. VLDB Endowment, 2003.

E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. w3c candidate 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 *Proc. VLDB*, pages 263–272, 2000.

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.

L.A. Rowe and M. Stonebraker. The commercial INGRES epilogue. *Addison-Wesley Series In Computer Science*, pages 63–82, 1986.

m. c. schraefel, Nigel R. Shadbolt, Nicholas Gibbins, Stephen Harris, and Hugh Glaser. CS AKTive Space: representing computer science in the semantic web. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 384–392, New York, NY, USA, 2004. ACM. ISBN 1-58113-844-X.

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.

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 Second International Conference on Semantic and Digital Media Technologies*. Springer, 2007.

M. Stonebraker. Retrospection on a database system. *ACM Transactions on Database Systems (TODS)*, 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, 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.

K. Taylor, R. Gledhill, JW Essex, JG Frey, SW 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, RJ Gledhill, J.W. Essex, JG Frey, SW Harris, and DC De Roure. Bringing chemical data onto the semantic web. *Journal of Chemical Information and Modeling*, 46(3):939–952, 2006.

M.L.T. Weithoner, T. Liebig, M. Luther, and S. Bohm. Whats wrong with OWL benchmarks? In *Proceedings of the Second International Semantic Web Conference Workshop on Scalable Semantic Web Knowledge Base Systems*, pages 101–114, 2006.

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 SWDB*, 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.

Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006. ISSN 0362-5915.

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.

M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, 2006.