A Nine Month Progress Report on an Investigation into Mechanisms for Improving Triple Store Performance

Alisdair W. Owens

2 October 2007

Abstract

This report considers the requirement for fast, efficient, and scalable triple stores as part of the effort to produce the Semantic Web. It summarises relevant information in the major background field of Database Management Systems (DBMS), and provides an overview of the techniques currently in use amongst the triple store community. The report concludes that for individuals and organisations to be willing to provide large amounts of information as openly-accessible nodes on the Semantic Web, storage and querying of the data must be cheaper and faster than it is currently. Experiences from the DBMS field can be used to maximise triple store performance, and suggestions are provided for lines of investigation in areas of storage, indexing, and query optimisation. Finally, work packages are provided describing expected timetables for further study of these topics.

Table of Contents

1 Introduction

This is a nine month progress report detailing my research into triple stores, specifically focussing on the performance issues found in these storage engines and their potential resolution.

Resource Description Framework (RDF) is one of the fundamental languages produced by the W3C to realise the Semantic Web vision (Berners-Lee, Hendler et al. 2001). RDF is a means for expressing knowledge in a generic manner. It is fundamentally based upon the concept of triples, that is, statements in the form of subject-predicate-object. This format allows the creation of an arbitrarily-shaped graph of knowledge, and is used as a basis for other languages in the W3C's roadmap such as RDF Schema (RDFS) (Lassila, Swick et al. 1999) and the Web Ontology Language (OWL) (Patel-Schneider, Hayes et al. 2003). It is designed to provide a flexible means to support simple data aggregation, discovery, and interchange. The 'layer cake' of semantic web technologies can be seen in Figure 1.

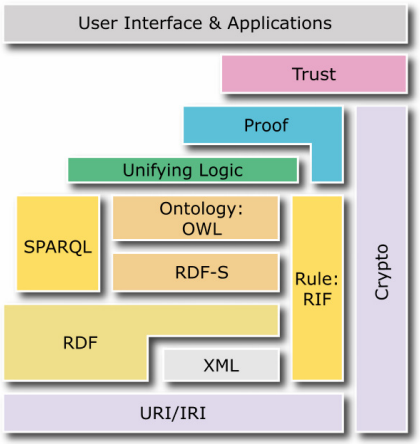

Figure 1: Semantic Web Layer Cake[1]

Since RDF triples do not enforce a limited structure upon data, RDF is ideal for applications where the structure of data added to the store is not well known in advance, is liable to change rapidly, or where there are many different structures being linked together, as found in data aggregation projects. Further, the RDF Schema and OWL specifications built on top of RDF can allow the inference of new data from that originally asserted into a store. Ultimately, of course, RDF provides support for the Semantic Web, and the massive quantity of aggregated knowledge that this implies. The requirement for large-scale storage of RDF data is clear. Triple stores are the database management systems (DBMS) of the Semantic Web world, and exist to support bulk storage and querying of RDF data.

RDF(S) is experiencing use in e-science projects (Taylor, Gledhill et al. 2005) (Taylor, Gledhill et al. 2006), particularly in the area of bioinformatics, which often use RDF as a data interchange format. Bioinformatics applications can produce extremely large amounts of data: the UniProt[2] dataset, for example, sums around 262 million triples. The use of RDF in this area is sometimes limited to data exchange rather than data storage, as this amount of data is impractical for performing fast queries under current stores. Further, many faceted browsers, such as mSpace (Smith, Owens et al. 2005) and CS AKTive Space (Shadbolt, Gibbins et al. 2004), use RDF as their knowledge base language. RDF offers a data model that is significantly more flexible than that seen in previous models, which is useful in situations where the structure of the data being produced is not well known in advance, or is liable to change rapidly. In situations such as this working with existing databases can become challenging, due to their expectation of consistent data structure.

While progress on producing language specifications has been swift, it takes time for supporting technologies to mature: the most powerful triple stores as of writing are capable of storing around two billion RDF statements, or in the order of tens of gigabytes of data (Errling 2006). However, pattern-

[1] http://www.w3.org/2007/03/layerCake.png
[2] http://dev.isb-sib.ch/projects/uniprot-rdf/

match queries performed over much smaller datasets can produce unacceptable performance (Smith, Owens et al. 2007), posing a significant challenge as datasets grow larger. This contrasts with commercial relational DBMS (RDBMS) technologies which are capable of storing terabytes of data whilst preserving real time query performance for many concurrent users.

This report describes the utility of the Semantic Web and its various related data representation languages, the unusual problems that RDF brings to the area of data storage, the current state of the art in terms of RDF storage technology, and goes on to describe how lessons learned from the RDBMS world might be applicable to the problem of storing triples. Finally, we consider how the needs of this new data environment can change the focus of storage optimisation.

2 Background & Research Motivation

This section describes the Semantic Web and some of its backing technologies, presenting the case for supporting the development effort in the form of satisfying the need for advanced data storage technologies.

2.1 The Semantic Web

The Semantic Web (Berners-Lee, Hendler 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 simply traverse the web, and enable them to communicate with each other, even in situations where they were not expressly designed to, through the power of shared understanding. 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 complementary 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 a triple 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 (De Roure, Jennings et al. 2005) are using it to advertise the availability of computing resources. E-Science researchers (Taylor, Gledhill et al. 2006) are using Semantic Web languages to exchange and aggregate data. There are Semantic Web browsers such as Tabulator (Berners-Lee, Chen et al. 2006) that offer individuals the ability to browse Semantic Web data for themselves. Faceted browsers like mSpace (Smith, Owens 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 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, Swick et al. 1999). It is expressed in the simple manner of a triple, composed of subject, predicate, and object. This is roughly analogous to the subject verb and object of a simple sentence (Berners-Lee, Hendler et al. 2001): for example:

Subject: Alisdair
Predicate: Has Gender
Object: Male

This is expressed visually in figure 2.

Figure 2: Triple concept

RDF triples are built out of Uniform Resource Identifiers (URIs) and literals, as seen in figure 1. 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 3.

Figure 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, Hendler et al. 2001). This is illustrated in Figure 4, where for simplicity we use the prefix 'ex:' 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.

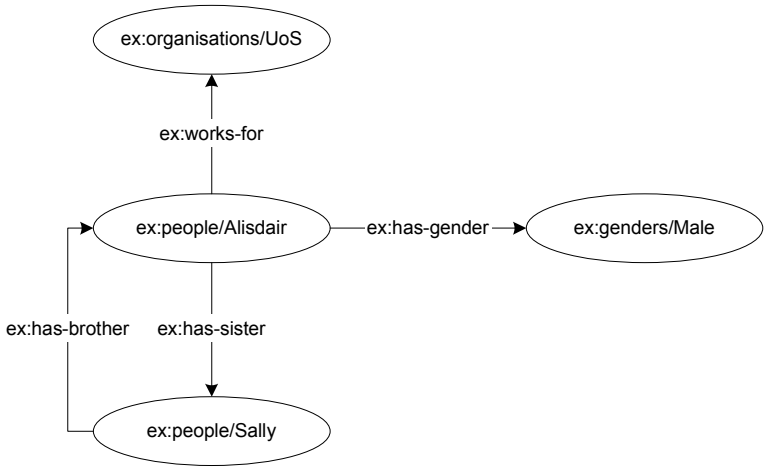

Figure 4: RDF graph

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, Gledhill 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.

2.2.1 RDFS & OWL

While not the focus of this document, it is worthwhile to give a 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, Swick 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. Note that this does not restrict any data from being asserted: it merely allows reasoning to be performed. For example, if I have a property 'Plays Instrument', which has a domain of 'Musician' and a range of 'Musical Instrument', and I assert the data 'Greg Plays Instrument Trumpet', it can be inferred that Greg is a Musician, and Trumpet is a Musical Instrument. There are a variety of other useful inclusions, such as the standardisation of the property 'rdfs:label' to describe human readable versions of a resource's name.

OWL adds much more wide ranging capabilities, aimed at providing computers with the ability to share not just information, but vocabulary (Patel-Schneider, Hayes 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.

Clearly, for the purposes of reasoning these sublanguages become progressively more powerful and harder to work with.

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. For example:

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

Figure 5: SPARQL Query

The query shown in Figure 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 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 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 SQL.

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 Triple Stores

Clearly, in the case of small datasets, it may be sufficient to simply statically store an RDF(S) file, and allow clients to process the data as they wish. However, there will also be many occasions where a large repository of data is in existence, and it is impractical to simply download and process it. Triple stores are the DBMSs of the Semantic Web world, and allow a repository of RDF data to be queried in place, using a language such as SPARQL.

While the purpose of triple stores is similar to that of conventional database systems such as the dominant RDBMSs, Object-Relational DBMSs (ORDBMS) 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 1975), triple stores allow

arbitrary assertion of knowledge in the form of triples. 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 1975).
- It offers cheap, detailed information to the DBMS on how the data is structured: how it might best be laid down onto disk, how queries can be optimised using knowledge of indexes, row lengths, etc. (Date 1975) (Stonebraker, Held et al. 1976)

While the requirement for strict schema definition is usually of use in traditional database situations, 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, Gledhill et al. 2006). 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, Dickinson et al. 2004) (Smith, Owens et al. 2007). Current triple 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 shape of query that it had not previously had to satisfy often.

Another significant difficulty for triple stores is the requirement for support for reasoning over RDF-S data and OWL ontologies. Current triple 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. It should be noted that the number of triples that have to be stored increases, with the associated performance implications (Harris 2005). It is often felt that a purely forward chaining environment would lead to an impractical amount of data being asserted, but this is refuted by Broekstra and Kampman (Broekstra and Kampman 2003). They also note that a rarely considered side effect of forward chaining is the difficulty of removing data – since the precomputed inferences need to be adjusted too.

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 potentially too expensive to support interactive-time query satisfaction. This problem is largely outside the scope of this document, as we focus on the problem of storing and querying the RDF graph, rather than performing efficient reasoning.

The difficulty of creating high speed triple stores presents a problem for the Semantic Web as a whole. If we are to expect individuals or organisations to host data and allow users to query it, particularly in a free or advertising supported environment, it has to be both feasible and cheap to store large quantities of data with many concurrent users performing significant queries upon it.

3 Related Work

This section considers work directly related to the development of triple stores. Initially, it considers existing database models and the work that has already gone into implementing them, with particular emphasis on the pre-eminent storage paradigm, the relational model. This is instructive, as triple stores are simply another case of DBMS, and all DBMS share common design concerns. It goes on to consider

the development that has gone into current triple stores, with reference to how previously existing ideas are applied to these new storage engines.

3.1 Previous Database Systems

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. Database systems can perform operations on the database such as:

- Add new file
- Remove file
- Insert data into file
- Remove data from file
- Retrieve data
- Change 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 1975).

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 hierarchical model also suffered particularly in instances where data was being accessed in a manner which did not conform to the defined hierarchy. 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.

3.1.1 The Relational Data Model

A radical diversion from early approaches was proposed by E. F. Codd in his paper *A relational model of data for large shared data banks* (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, Held 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 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 7: Illustration of common database operations

3.1.2 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 (Atkinson, Bancilhon et al. 1989). This is based on the familiar principles found in object-oriented programming, and indeed these databases 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 1975) 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 the majority of triple 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. OLAP technologies, for example, offer the ability to prevent work being repeated inside the DBMS when asking several related questions at once (Chaudhuri and Dayal 1997).

Finally, a common model used by applications for data persistence is simple key/value pair storage, as evidenced in Berkeley DB (Olson, Bostic et al.). 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 1 offers a brief overview of the differences between current models.

	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 1: Database model comparison

3.1.3 Implementing a DBMS

Most of the issues described in this section are relevant to most forms of DBMS, but information is drawn largely from work on relational DBMS, since there is a large body of work in this area, and it is considered more relevant to triple storage than other highly developed models such as the object data model. It should be notes that a large proportion of the design considerations listed in this section are relevant to all forms of databases. There are a host of important issues to consider (Date 1975) (Hawthorn and Stonebraker 1986) (Stonebraker 1980):

- What is the optimal manner in which to store the data, both on disk and in memory? Are we looking to optimise for small database footprint or performance? If the answer is performance, are we most interested in read or write performance?
- How will the DBMS support multiple users?

    - How does the DBMS resolve issues such as allowing simultaneous reads and writes to the database?
- How can it efficiently satisfy a query expressed in a declarative language?
    - Can the work be distributed across multiple processors or processor cores?
- Do we wish to distribute the database over multiple machines? If so, how can this be accomplished, and what benefits can be derived in terms of performance, scalability, and reliability?

This section considers some of these issues in detail, omitting distribution over multiple machines as an area for future study if it is chosen to pursue work in that area:

3.1.3.1 Storing Data

Clearly, one of the most important considerations is how to represent the database on the machine(s) on which it is to be stored. This has a large effect on such factors as:

- Retrieval performance
- Write performance
- Space

There is often a requirement for trading off between these points, and we choose where to focus depending on the expected usage profile of the DBMS. Physical representation is also heavily dependent upon storage mechanism (such as RAM, hard drive, or even flash memory). The vast majority of database systems run on machines with a limited amount of main memory and a large hard drive, and it is on this standard that we will focus.

Storing data in RAM is a relatively simple matter: RAM has the ideal characteristics of constant access time combined with fast retrieval for any piece of data. This means that there is no requirement for pieces of logically contiguous data to be placed next to each other, making RAM extremely easy to work with: one might simply use hashing algorithms (for example, hashing the key of a table) to determine a random position in memory for a piece of data, and the data could then be retrieved in constant time. Contiguous pieces

The difficulties in working with RAM are that it is limited in size and not persistent. While some observers (Stonebraker, Madden et al.) propose that in future it will be practical to store most current databases in main memory, this is certainly not currently the case. Further, lack of persistence means that it must be possible to reconstruct the database into RAM from a persistent store (usually a hard disk) in case of failure. The combination of these limitations result in most databases using RAM for storage of metadata, working data, and cache, leaving the actual dataset stored on hard disk.

Unfortunately, while the speed of computers has continued to rise dramatically, the performance of hard disks has not kept pace (Stonebraker, Madden et al.). The speed of data transfer off the disk is quite slow, and writing is slower again. Even more critically, there is a seek time associated with travelling from one block of data to another non-sequential block in the order of 10ms. This storage medium is a major limiting factor in both read and write performance in any DBMS.

In practise, most modern (O)RDBMSs store their data in a relatively simple fashion, writing it to disk row by row, starting with row metadata. Order is often loosely maintained. This means that if one wishes to add a new record to the database, it can be accomplished in a single write. If there is an expectation of an approximately equal number of reads and writes, this is a very sensible approach: writing to disk is an inherently slower operation. DBMSs of this form are usually considered ‘write optimised’.

Optimising for writes in this fashion can have a significant impact on read performance, however. This is caused by several issues:

- Data order may not be maintained. Since reordering data to keep it contiguous is very expensive, and this would be a regular requirement when data is added, maintaining strict order might be considered too expensive. This, however, can mean that additional disk seeks are required when retrieving data. Further, it may make join operations much more expensive.

- Row-orientation means that when we perform a select based on a single column, we still have to read the entirety of each row into memory. This results in greater data transfer and more memory use, and is particularly damaging on wide tables.

If, however, we expect our database use to be heavily read-biased (for example, in applications such as data warehousing), we might choose to optimise for reads. Characteristically, a read-optimised DBMS will store its data in columns: that is, each column of data will be written to disk contiguously. This benefits read performance massively when working with specified columns over a larger table, as irrelevant columns can simply be ignored. Further advantages can be gained, in that it is easier to perform compression, reducing time taken to read from disk (Stonebraker, O'Neil et al. 2005).

It should be noted that attention must be paid to the operating system when attempting to ensure that data is written in a particular order (Stonebraker 1981). File systems for modern OSs are designed to support simple file creation, deletion, and modification, and in doing so may create data in a non-contiguous manner. While it will of course present the data as contiguous to any application accessing it, there will still be a requirement for performance-sapping seeks when travelling between blocks of data. This is highly undesirable behaviour in the specific case of a database, and should be avoided where possible.

3.1.3.2 Indexing Data

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 1975). This process creates a data structure that, for a column or set of columns, quickly returns the location of specified data items within those columns.

A commonly used index for RAM-based storage is the hash map. 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 RAM can retrieve data from any position in constant time, extremely fast.

Unfortunately, it is often impractical to store indexes in RAM, and as soon as the index does drop out of memory, hash indexes 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 we 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, when indexing outside of memory, hash indexes are used only in situations where queries are operating on discrete specified values, not over a range.

When attempting to store data on disk, data is usually indexed using B-trees (Comer 1979) (or variants such as B+ or B* trees). B-trees are self-balancing tree structures where each node can have multiple child nodes. This format is very useful for block-based storage such as hard disks, if we size the nodes to correspond with the size of a disk block: in a disk, the cost of getting to a node is high (since there will be a disk seek for each node traversal), but the cost of examining data within the node is low, since the entire node is read into memory in a single read. The B-tree's characteristic wide nodes and shallow depth are well suited to this application. The B+tree variant is particularly common, and modifies the structure such that all actual data is stored in the leaf nodes of the tree. This offers the significant advantage that the data can be stored separately to the index, making it simple to index on several items in a single table without duplicating data. As shown in figure 8, it is possible to create links between leaf nodes in the B+tree, easing the process of scanning sequential items. While the B-tree and its variants are excellent indexing structures for disk-based storage, they do not usually fill up each node with data, leading to significant amounts of wasted space. This is inefficient for memory-based storage, and other data structures are preferable for this application (Wood, Gearon et al. 2005). If data is stored in sorted (or near-sorted) order on an index, it is referred to as 'clustered' on that index.

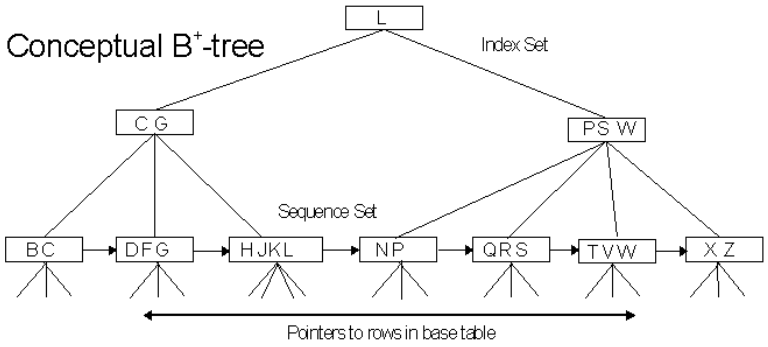

Figure 8: B+tree[3]

A further alternative for disk-based storage is the bitmap index. This is traditionally used for low-cardinality attributes such as 'Gender', or 'Country', but has been shown to be applicable even to columns with a high degree of unique values (Date 1975). A bitmap index simply creates a bitmap for each unique value that an item in a column could be set to. Each bitmap contains a bit for every item in the attribute, showing whether the field contains that value or not. Compressing the bitmaps using Run Length Encoding (RLE) based mechanisms is usually very simple and worthwhile.

Each bitmap in a bitmap index, once compressed, is quite small, and tractable to load into memory. They can be quickly scanned to find desired matching pieces of data, and bitwise operations can be performed on bitmaps related to different columns in a table, speeding the performance of complex select queries. While bitmap indexes in general offer excellent read performance, they may be inefficient in terms of space required when indexing columns with a high degree of uniqueness. Further, 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.

3.1.3.3 Operations

As noted, the three most common operations in (O)RDBMS are select, project, and join. The efficiency of all three of these operations is impacted significantly by the size of the dataset, the level of uniqueness of the data, the availability of indexes, and whether the dataset is in sorted order.

Select is, in ideal conditions, an extremely fast operation. 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 fundamentally 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.

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 (particularly large if no indexes are available), to more useful techniques, such as merge, sort/merge, and hash joins (Date 1975):

- **Merge** joins assume that both tables are sorted in order on the column that is being joined on. With this being the case, a simple scan of both tables 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.
- **Sort/Merge** simply sorts the tables as required, then performs a merge join on the resulting data.

[3] http://www.ianywhere.com/images/whitepapers/1003010_1.gif

- **Hash** joining also performs a single scan over each table. It creates a hash table on the first relation, with a pointer to the corresponding tuple on disk. When scanning the second table, it compares against that hash table to produce the joined table. This technique scales in linear fashion with the amount of data scanned, and does not require tables to be sorted to work efficiently (although, of course, sorting will ensure that less seeks have to be performed). 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.

3.1.3.4 Querying Data

As noted, in early DBMS data retrieval was performed through procedural coding mechanisms. Programmers defined exactly what path through the database was taken, and exactly how data was to be retrieved. In the leap from these 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: 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 by C.J Date, there are four steps to query optimisation (Date 1975):

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 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.

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 access with great ease.

The creation of procedures for step 3 might come about through a variety of processes. One example, created for the Ingres database, is called query decomposition (Youssefi and Wong 1979). This mechanism consists of two operations: ‘detachment’ and ‘tuple substitution’. Detachment breaks portions of the query with just one variable in common with the rest of the query. Once a stage has been reached where no more detachment can be applied, tuple substitution substitutes real data one piece at a time into one of the variables in the remaining query. Detached queries can be answered in any order, or even simultaneously, since they are independent of the rest of the query. By answering detached queries

in an optimal fashion, the working set of data that can be substituted in during tuple substitution is reduced, providing improved performance.

3.2 Existing Triple Stores

Current triple stores are developed in a variety of styles. There are developments such as widely distributed peer-to-peer stores such as RDFPeers (Cai and Frank 2004), but we largely focus on the most common kind of repository: those that are situated on a single computer (or as a cluster), and behave much like the conventional DBMSs of today.

It is notable, in fact, that most current stores are actually based upon existing DBMS technologies: 3Store (Harris 2005), Virtuoso (Erling and Mikhailov 2006) and D2R Server (Bizer and Cyganiak 2006) are layers on top of existing DBMS, as are the best performing backends for other stores such as Sesame (Broekstra, Kampman et al. 2003) and Jena (Wilkinson, Sayers et al. 2003). This approach certainly appears to make sense: existing DBMS technologies have been refined and optimised over many years, are stable, and relatively simple to develop on top of. This section will spend time considering whether this approach is worthwhile in the long term. Stores such as Redland (Beckett 2002), Sesame and Jena also have in-memory backends, while Kowari (Wood, Gearon et al. 2005) has its own dedicated data storage layer.

In the following subsections, we consider the developments made in some of the most popular triple stores currently in existence, making an attempt to focus on those that offer varying storage paradigms.

3.2.1 3Store

3Store (Harris 2005) is fast C library that runs on top of the MySQL RDBMS. It is a triple store of moderate performance, known to store at least 30 million triples with reasonable performance on simple queries[4].

3Store uses a simple schema in which to store the graph shape (as quads, since it adds another field to denote provenance, or 'model') consisting of:

Model	Subject	Predicate	Object
64 bit int	64 bit int	64 bit int	64 bit int

Figure 9: 3Store triples table (Harris 2005)

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, which 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: the SPARQL is translated into an SQL query that the RDBMS can answer. For example, if one wished to answer the SPARQL query in Figure 5, 3Store might perform the following SQL upon the triples table:

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

Figure 10: SQL produced by 3Store

Clearly, additional SQL is required to determine the lexical representation of the hashed values that would be returned, 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

[4] http://www.aktors.org/technologies/3store/

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 triple stores. However, while it is relatively simple to implement, and provides a moderate level of performance, it should be noted that RDBMS generally expect a somewhat normalised schema to afford efficient indexing and provide the query optimiser with information.

3.2.2 Kowari

Kowari (Wood, Gearon et al. 2005) differs from most triple stores in that it implements its own custom Java-based backend. It uses AVL trees to index RDF statements stored in flat files. In much the same way as 3Store, it separates the conceptual shape of the graph from the physical representation of the URIs, into a 'Statement Pool' and a 'String Pool' respectively.

Kowari, like 3Store, stores statements as quads including provenance information. However, it stores the statements six times, corresponding to the six different ways in which an RDF triple can be ordered (SPO, SOP, etc.). This enables Kowari's indexing structure, and also provides efficient disk access, since there is no need for the disk to seek to multiple locations when retrieving all triples which mention specific URIs.

Kowari indexes its Statement Pool using Adelson-Velskii and Landis (AVL) trees, a self-balancing binary structure. There are six trees, corresponding again to the possible orderings of a triple. Tree depth is kept small by having each node in the tree represent a block of triples, making it fast to traverse and increasing the likelihood of keeping the index in memory.

Kowari offers performance broadly comparable to 3Store (Taylor, Gledhill et al. 2006).

3.2.3 Jena

Jena (Carroll, Dickinson et al. 2004) is a Java API designed to support the creation and manipulation of Semantic Web data. In contrast to most of the other stores mentioned, Jena is intended to provide library level RDF support for applications, rather than acting as a network-available knowledge store. Initial iterations supported RDF, but subsequent revisions have added RDF-S and limited OWL support.

Jena offers a variety of mechanisms for storing data (Wilkinson, Sayers et al. 2003): as well as the usual RDBMS backends, it offers in memory and Berkeley DB backends. Berkeley DB (Olson, Bostic et al.) is a low level embedded DBMS, offering simple key/value pair storage. It is primarily intended for application data persistence, but offers reasonable performance under Jena. There is innovation to be found in the RDBMS level storage as well, however. As well as a simple triple table, it offers a mechanism known as Property Tables (Wilkinson 2006). Instead of storing the triples in one long list, it is possible to specify separate tables for properties that relate subject-value pairs to each other. In the case of properties with a maximum cardinality of one, it is even possible to store multiple properties in the same table. This is illustrated in figures 11 and 12.

Subject	First name	Surname	Height In cm
<Alisdair Owens>	Alisdair	Owens	185
<John Smith>	John	Smith	192
<Melissa Arnold>	Melissa	Arnold	NULL

Figure 11: Property table containing many properties with a cardinality of 0 or 1

Subject	JobTitle
<Alisdair Owens>	PhD Candidate
<John Smith>	PhD Candidate
<John Smith>	<Research Assistant>
<Melissa Arnold>	<Research Assistant>

Figure 12: Property Table for one property with a cardinality over 1

In this schema, details such as first name and surname have a maximum of one value, whereas an individual may have more than one job. As soon as the property has a cardinality of greater than one, it has to be split out into a separate table as seen in Figure 12. A crucial feature of this schema is that to avoid extreme complexity, a property may only appear in one table.

While property tables are shown to have performance advantages (Wilkinson 2006), determining the table layout to use is a difficult problem (Abadi, Marcus et al. 2007): ideally, it is important to store subjects and values that tend to be accessed together in a single table, allowing the use of selectivity rather than joins in the querying process. However, it is also important to make sure that the table is as densely populated as possible, as excessive populations of NULL values, and the storage requirements thereof, can overwhelm the benefits gained through the use of a better schema. While steps have been taken in an attempt to perform this work automatically (Ding, Wilkinson et al.), it is still largely a complex manual task.

3.2.4 D2R Server

D2R Server (Bizer and Cyganiak 2006) is not strictly a triple store *per se*, but rather a layer that maps SPARQL queries onto an existing RDBMS schema. This makes publishing the wealth of data stored in relational databases around the world relatively easy, as it allows the database to be accessed as if it natively stored RDF. This approach offers considerable advantages: data can be stored for internal consumption by applications in a traditional RDBMS fashion, with the associated benefits of performance and guaranteed data integrity, while also being published for the consumption of the wider Semantic Web. There is also no need to maintain separate versions of data in SQL and RDF formats, and no need to run an entirely new database system to store RDF: just a relatively simple application running on top of the RDBMS.

Clearly, this different approach has significant benefits over and above those offered by pure triple stores, but it is worth considering situations where it is disadvantageous:

- The ability to assert data outside of a defined schema is lost.
- As previously mentioned, relational databases are usually optimised for specific access patterns. If arbitrary SPARQL access is allowed through D2R, it is likely that data access patterns will vary in an unpredictable manner. This means that many queries may be very inefficient.
- Changes to the schema of the database necessitate changes to the D2R mapping, increasing maintenance costs.

If it is expected that the D2R server will experience significant use, it would be advisable to analyse queries that are likely to be performed, and ensure that indexes are created to support these: indeed, this is a function that could potentially be automated through analysis of queries actually being performed on the database. This would significantly improve the performance of this mechanism of RDF storage for the purposes of acting as an accessible node on the Semantic Web.

3.2.5 Virtuoso

Virtuoso Universal Server is a high performance DBMS with the ability to store a variety of data such as XML, RDF, and web server applications. At its core it is an Object-Relational DBMS (ORDBMS) with other data items translated to fit within this model. The creators of Virtuoso claim to be able to process a LeHigh University Benchmark (LUBM) (Guo, Qasem et al. 2006) of over a billion triples effectively, although relatively little study is made of query response times.

Virtuoso uses a familiar model for storing its RDF data (although it also offers a D2R-like translation mechanism) (Erling and Mikhailov 2006): the shape of the graph is stored as a long list of triples, as seen in applications like 3Store, and indeed many other stores. The ideas that differentiate Virtuoso from the pack, allowing it to process such large quantities of triples, are centred around the minimisation of storage space required per triple, and the use of bitmap indexes (Errling 2006). Since it is only possible to store quantities of triples in the low millions in the memory available in standard computer systems of today, the designers of Virtuoso concluded that it was inevitable that hard drive storage would be necessary, and that a low storage footprint per triple would minimise the problems associated with the slow transfer rates of this medium.

A low space usage per triple is accomplished through a variety of means, including splitting triples into prefix and local parts, storing each prefix uniquely. Further, bitmap indexes are used to significantly reduce the storage required for the indexes and speed up queries, particularly for datasets where there is a relatively low level of unique URIs.

3.2.6 Vertical Partitioning

Vertical Partitioning (Abadi, Marcus et al. 2007) is a recent innovation with its foundations in the idea of Property Tables. Abadi et al. note that there is a great deal of complexity in determining the correct schema when considering the tables containing properties with a cardinality of 0 or 1. They further consider the fact that in traditional RDBMS that store data on disk row by row ('row oriented'), the amount of data that has to be read per row increases greatly as the table gets wider. Since it is so difficult to work with the traditional property table model, they propose instead the use of 'Vertical Partitioning'. This is equivalent to storing all data in the model illustrated in Figure 12: a separate table for each property.

Although this approach loses a major benefit of Property Tables, in that many joins are still required to answer queries, the denormalisation of the data into many separate, sorted tables reduces the depth of indexes, eliminates the existence of NULLs, and allows the use of fast merge joins.

The authors champion the use of C-Store (Stonebraker, O'Neil et al. 2005), a column oriented DBMSs where data is stored on disk column by column to store RDF data. On the surface, this appears to have limited value, since the tables are so thin that the amount of extraneous data loaded is minimal, but several advantages are noted, including:

- Tuple headers are stored separately. Given that the headers are so large in most RDBMS, they tend to overwhelm the storage requirement for the data itself.
- Column oriented data compression: since data in an individual column is likely to have a high repeating factor, having the data stored column-wise on disk aids data compression, reducing the time spent reading data from disk.

The authors submit a further innovation in 'Materialised Path Expressions', which are in essence the joining of multiple properties to appear as one in the database. Data such as 'The authors of books written in 1860' can thus be precalculated, saving work on queries that utilise these paths.

The combination of the vertical partitioning schema and the use of a column-oriented DBMS appears to offer significant performance gains, with the authors claiming orders of magnitude improvements over previous stores. It should be noted, however, that C-Store, as a heavily read-oriented DBMS is known to be slow at inserting data after the initial write. The authors make no study of the stores' relative performance on data inserts.

3.2.7 Space Filling Curves

The TriStarp[5] project has utilised space filling curves to store and index data in a non-RDF triple store. A space filling curve is essentially a continuous curve that fill up any given square or cube (or even a hypercube of any dimension), assuming that object is constructed of discrete units. Space filling curves are usually repeating patterns that are constructed iteratively. Well-known examples of these are Z-order and Hilbert curves (Lawder and King).

Space filling cubes can be applied to triple storage and indexing. If we take RDF as a three dimensional storage problem (ignoring, for now, provenance), we can imagine it as a cube, with each dimension

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

being one of subject, predicate, and object. An RDF triple is a point within the cube. The fact that RDF has more three dimensions is a problem when attempting to store it contiguously - in a one dimensional manner. Space filling curves can be applied to this problem: the curve passes through every point in the cube (or every triple, in this case). If we store the triples on disk in the order in which they are traversed by the curve, then we have a one dimensional representation of our three dimensional structure. A good curve will keep spatially related data items somewhat close to each other on disk.

Indexing of this structure can occur through a tree-based system in curves such as the Hilbert curve (Lawder and King). 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.

Storing and indexing via space filling curves has the important property that no one dimension is dominant, as is the case with one dimensional indexing techniques such as B-trees. It is possible to retrieve data by any combination of the three 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 than if only one is).

Space filling curves are of special use in range selections over more than one dimension. Traditional DBMSs perform poorly at this task, since it is necessary to scan all data items that satisfy one of the ranges, and then restrict the resultant output by the other specified ranges. In the case where only broad ranges are required, or data is of low cardinality, this is extremely inefficient. Using space filling curve-based techniques, a volume is designated for retrieval, the points at which the curve intersects that volume computed, and all of those points retrieved.

There appears to be little evaluation of the performance of space filling curve based techniques as applied to triple graphs in the TriStarp system.

3.2.8 Distributed Stores

Triple stores such as RDFPeers (Cai and Frank 2004) and YARS2 (Harth, Umbrich et al.) allow triples to be stored over a widely distributed network of computers. YARS2 is particularly notable, with studies showing it functioning with a 7 billion triple dataset.

YARS2 is a heavily read optimised store, using six different indexes into six data orderings, supporting full retrieval of RDF quads. The index type used is called a ‘sparse’ index, which is an in memory index into a sorted and blocked data file. To retrieve data, a binary search is performed upon the index, and the closest block of data is retrieved. To enable it to stay in memory, the index gets less specific as the dataset gets larger. This results in near-constant retrieval time with respect to index size, as disk seeks are minimised, and the major cost is the disk seek rather than the amount of data retrieved: if we assume that a disk seek takes 10ms, and the disk reads at 50MB/s, it is possible to read 500KB of data in the time a seek would take. It should be noted, however, that very large reads will use up significant amounts of main memory, and may flush useful data out of the operating system hard disk cache.

In many distributed stores, distribution of triples occurs via a hash of the triple or some part of it, using a hash function that is known to all participating stores. This lightweight mechanism allows any store to know where any given triple in the system can be found. YARS2, for example, uses a hash of the first part of the quad, modulated by the number of machines in the store. This mechanism can keep closely related data clustered on a single machine (which reduces the amount of time-consuming communication between machines), but can be disadvantageous when considering data orderings that are predicate-first (Battre, Heine et al.). The solution used by YARS2 is to randomly distribute predicate-first orderings, and flood queries that require this ordering to all machines. It is not clear how the hash function will continue to work with addition or removal of machines.

4 Lessons Learned

This section considers the lessons that can be learned from the implementation of existing DBMSs, and how they can be applied to the problem of storing and querying RDF data, as well as the ways in which this problem requires new solutions to produce acceptable performance.

4.1 The RDF Data Model

In some ways, the storage of RDF data is extremely complex: its relatively unstructured nature does not lend itself to storage in anything but the most broad of data structures, and this inhibits high performance retrieval. However, combined with the simplicity of SPARQL, this does offer certain advantages. In creating a DBMS to store RDF, while we do not know the conceptual structure of the data that might be asserted, we do have strong knowledge of the data structure that we will be storing it in – for example, a triple table, or a vertically partitioned structure. Despite the unpredictable nature of the data, the manner in which we are to store and retrieve that data can actually be actually far more predictable than it is in a standard (O)RDBMS. Assuming we discount the standard property table model as overly complex, there is no need to offer arbitrarily wide tables, support for three-valued logic, duplicate rows, or objects.

While these generalised models of the structure of arbitrary RDF data are very difficult to optimise for high performance retrieval, they do offer the opportunity to focus, without having to deal with the massive complexities that, for example, SQL based DBMSs have to overcome. With a triple store, there is prior knowledge of the way the data will be stored at *code-time*, as well as the fact that most of the operations in any query will be selections and joins, and we can heavily optimise these operations, in particular ensuring that joins exhibiting linear or near-linear behaviour are used. Thus, there is in many ways less complication to the process of storing and performing query optimisation over RDF data. Further, such issues as the unnecessarily large (for the purposes of RDF storage) tuple headers in general purpose DBMSs could be completely eliminated in a dedicated store.

While the relational model is perfectly capable of modelling RDF data, it is questionable as to whether it is advisable to continue using conventional RDBMSs as storage engines. These tend to be optimised for write operations, and even in read-optimised engines, issues remain due to the fact that the RDF is being stored through two layers of logical model: the RDF layer, mapping on to the relational layer, which finally maps on to physical storage. The characteristics of most applications of (O)RDBMS and those of RDF storage are radically different: triple stores' fixed, thin tables require the storage of a great deal of data-centric information to allow accurate query optimisation, for example. This is not the case with the majority of (O)RDBMS use cases, where more schema-centric optimisation is simple and fruitful.

A clear illustration of the issues that can be created by the application of standard (O)RDBMSs to the RDF storage problem comes from considering a step most triple stores use: storing URIs and literals as fixed length hashes rather than the variable length strings that they truly are. Keeping rows at a fixed length provides improved performance, and the hashes are usually much smaller than the URIs they replace, reducing the amount of data that has to be transferred to and from disk. Unfortunately, this also destroys any notion of meaningful ordering. This means that if we wish to specify a range of values, many more disk seeks will be required to satisfy them than if the data were in truly sorted order. This is a negligible issue when considering the subject and predicate, as range queries are quite unlikely (and usually have little meaning) over URIs, but such an operation could not be considered unlikely over literal-containing objects. A specialised triple store might be able to account for this and order data appropriately, but this is unlikely to be simple in a general-purpose (O)RDBMS. Berkeley DB appears to offer an attractive alternative as a backend, with its ability to store arbitrary data structures, but there appears to be little scope for optimisation once the basic triple store model (such as that found in Jena) has been implemented.

While it might be considered unwise to recreate the many years of optimisation that have gone into the creation of existing DBMS, the need for very specialised behaviour to extract worthwhile performance from stored RDF informs the author's opinion that high performance triple stores will either have to move away from existing DBMSs, or be built on DBMSs that have received significant customisations to optimise them for storage of RDF data.

4.2 Storage

One of the lessons that is apparent from the review of existing database systems is the importance of the manner in which data is stored. While RAM is both simple to work with and fast, it is still too rare a commodity for it to be possible to hold all of a store's data in main memory, although this may change in the future. The ability to work with secondary storage, usually hard disks, is an essential feature for almost any triple store. Particularly notable is the manner in which the issue of secondary storage access almost dominates other concerns, due to the failure of secondary storage to improve in performance in line with the other components within the modern computer.

Upcoming technologies like Flash memory exhibit behaviour more in line with that of RAM, having very short access times, and may well result in changes to the way in which DBMSs store and access data. For the foreseeable future, however, hard disks continue to be the dominant technology. The importance, then, of storing data in a manner suited to this underlying storage mechanism cannot be overestimated:

- *When attempting to optimise data assertion performance, it is important to minimise the amount of data written to disk.* This includes reordering of data: for example, if we wish to keep data in sorted order on disk, 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 the disk.* This does not necessarily mean that the data footprint should be small: if the data is stored in several representations, we need only read from the one that will allow us to retrieve the data is the quickest time. It is useful to maintain data in sorted order, contiguously on the disk, as this will greatly reduce either the amount of data that has to be read, or the number of seeks that have to be performed.
- For both cases, it is important to read or write the data as contiguously as possible to prevent slow disk seeks.

As can be seen, optimising for read performance tends to compromise write performance, and vice-versa. This effect is seen in the rest of the DBMS world. General purpose (O)RDBMS tend to be row oriented, and optimised for fast writes. For purposes where the query performance of these DBMSs is insufficient, column stores or even more exotic data storage mechanisms may be used. It should be noted, however, that inserting data in a triple store that performs significant levels of pre-computed reasoning will require write *and* read activity from the underlying storage, so the case for write vs read optimising is not so clear cut as it is in the area of (O)RDBMSs. Further, the current read performance of triple stores is poor enough that in datasets that are unlikely to experience great change, read optimisation would be highly advisable.

With this said, a further point to note is that column storage is of questionable benefit, once the store is being created solely for RDF storage. While (Abadi, Marcus et al. 2007) showed some significant and well-justified performance improvements when using a column store over a write-optimised database such as Postgres, many of these benefits were by virtue of the fact that the underlying store, C-Store, was read optimised, not that it was column-oriented. Such facts as C-Store's storing all data in sorted order meant that very fast merge joins could be performed, rather than the slower operations such as sort-merge that Postgres would have to perform. Given that in a large proportion of cases it will be necessary to read the entirety of a triple anyway, it is unlikely that column orientation will be a great deal faster than row in equally read-optimised DBMSs: particularly, use of a column store may tend to result in more disk seeks.

It should be noted that while the separation of the conceptual model of the graph by creating hashes from the text of URIs and literals bears many advantages, particularly in the way it makes each row in tables describing the graph small and fixed in size per tuple (and thus quick to access), it also causes significant problems in terms of creating additional disk accesses. At least one disk access will be required in the translation of every unique URI or literal in the results set back to its textual form. This is a crippling burden on queries that return a lot of results. However, complex queries that may result in a lot of data being processed, but have a small result set, benefit massively from the hashing approach. This issue might be mitigated by a variety of means:

- Storing URI prefixes in memory, and the rest of the URI directly in the table. If the URI is over a certain small fixed width, a hash could still be used.
- Column-oriented data compression.
- Storing the hash and plain text of URIs/literals against a hash of each of the URIs they are connected to. This would mean that simple queries that produce a lot of results could require much fewer disk accesses, as related data would more often be available in a somewhat contiguous manner. This method would significantly increase storage requirements, however.

Finally, it is important to optimise the use of what main memory is available. One might choose to cater for the specific case by using caches for single queries, or for the general case by storing more metadata on the structure of data in the store to give hints to the query optimiser. This will be discussed further in the following subsections.

4.3 Indexing

The indexing requirements of triple stores differ from those of most (O)RDBMS applications, although some (O)RDBMS do offer a great deal of flexibility in this regard. As noted in section 3.3.1, it is likely that most queries specifying subjects or objects will be related to a single URI, or a discrete set of them. In this case, the B+tree's advantage of providing fast range queries is of limited interest, as a range is unlikely to be specified. A hash-based index could provide indexing at a constant cost per URI, likely much less than that required by a tree method. For regularly accessed data, this is a very good candidate for caching. As a secondary index, however, B+trees may be of use: if we consider the query:

SELECT ?z WHERE {?x has-gender ?z}

Let us assume that ?z has a small cardinality, while ?x has a very high cardinality. While we could use a hash-based index to find all uses of 'has-gender', we could still be left with a very large number of results to consider, perhaps having to perform an expensive projection over the column holding all possible ?z. Using a tree index, though, it would be very easy to answer. While we are unlikely to have a range query over the entire table, we are very likely to have to deal with queries that request all objects for a certain property (in effect a range of the entire sub-index). In the case above, if we attempt to do this by scanning the actual data, or projecting over the object column, this could take a very long time. Accessing the information through a tree index would be much easier: since we are looking for all unique pieces of data, and the index lists only the unique pieces of data with no repetition, much less information has to be scanned. It is notable that this type of query can perform very poorly on 3Store, despite it being based on a RDBMS that uses B+trees for indexes. This is likely to be a query optimisation issue.

Another mechanism that is clearly worth investigation is bitmap indexing. Virtuoso, despite having an otherwise normal triple table storage mechanism, manages to store billions of triples using this method. It should be noted that this method does have significant performance implications if we expect writes to be performed regularly.

Finally, the sparse index structure used in YARS2 is of interest. It supports prefix lookups, where the index can be queried by only part of the string: this could be of great use in object-centric orderings, where partial-text and range queries are likely to be a regular feature. The major disadvantage of this index type is the large memory footprint it entails, through both the index itself and the large block sizes used as the index scales up. This index type may also be of great use as a secondary index for a primary hash index: much smaller sub-indexes could be loaded into memory as needed.

It is very much worth considering the implications of caching on the indexes generated for the triple store: these stores, with their potentially large join operations, are likely to make heavy use of whatever indexes are available, and the caching of upper levels of a tree-based index (for example), or regularly used triples, could save many disk accesses over just a single query. It should be noted that in practise it is quite possible that the operating system disk cache would provide this functionality to some extent anyway, but it would be unwise to rely on this.

4.4 Operations

As noted, unlike most SQL products, triple stores cannot necessarily expect certain queries to be performed, or certain ones to be excluded (although preventing extremely expensive queries from being run may be a necessary step in some situations). In particular, it is by no means unlikely that SPARQL queries will cause a great deal of data to be worked with, even if they are quite simple in appearance. This being the case, it is extremely important that operations such as select, join, and project scale in a manner that does not cause large stores to become prohibitive.

In practise, ensuring that data is sorted and clustered is quite attractive. Joins are regular operations in triple stores, and can be extremely expensive. It is important, if possible, to keep data in sorted order and as clustered as possible, maximising the possibility of merge joins being possible.

The availability of sorted data means that it can be accomplished in linear time with a fast algorithm, the merge join. Further, if all data is sorted and stored in contiguous fashion for each possible ordering, there are no unnecessary disk seeks. In this case, with an appropriate indexing mechanism, triple stores really could perform in an effective manner for large datasets.

In practise, however, this rigid scheme is likely to cause problems for a system that experiences even relatively rare writes. This mechanism would mean storing information six times over, and moving a vast quantity of data to accommodate every write, to ensure clustering. While it is the opinion of the author that triple stores should focus to some degree on read optimisation, since the emergence of reasoning systems will likely call upon read performance even when writing, this is excessive. There are a variety of compromises that might be enacted:

- Vertical partitioning has many excellent characteristics. Its implementation requires only two sorted orders to perform at peak efficiency. It should be noted, however, that this scheme complicates any situation involving more than one property, or where the property is unknown. This is a *relatively* rare scenario, but it is by no means inconceivable that one might run a query where a subject is given and no other data, hoping to retrieve all data that a source has on that subject. Storing the data in additional forms (even an additional triple table format) might mitigate this issue to some extent.
- The Subject-Object-Predicate and Object-Subject-Predicate orderings are so rarely required as to be virtually useless. These orderings could be eliminated.
- The Predicate-Object-Subject and Predicate-Subject-Object orderings could be implemented only as unclustered indexes, rather than physical data orderings. This would provide some of the required functionality – queries such as that mentioned in section 4.3 could still be easily answered through index work alone.
- Data clustering is largely only important on the secondary part of orders that do not have objects as the primary concern, due to the unlikelihood of range queries on URIs. This allows data to be split along the boundaries of the primary parts of these orders, with no performance penalty for the vast majority of queries. It is further important to note that if we use URI/Literal -> hash mappings, for object-centric orderings it is important to maintain order based on the plain text, not the hash.
- Data could be divided horizontally, and clustering allowed to break along these lines. The size of each cluster could be varied as required, and if intelligently placed cause relatively little performance degradation.
- Newly written data could be stored in a small write-friendly store, and batch-updated into the main database as necessary. To prevent unnecessary disk reads, the data could be mirrored in memory if possible. Deletions could be accomplished in a 'lazy' fashion, by simply marking data to be ignored, for purging at a later date.

4.5 Query Optimisation

SPARQL, although having a superficially similar syntax to SQL, is very different. It is very much simpler, and operates at a higher level: SQL makes it possible to provide hints to the query optimiser as to the most efficient manner in which to retrieve the data, whereas SPARQL does not. Further, there are differences in the manner in which each language is used: as noted, (O)RDBMSs are generally used with by a fixed application pool, where the vast majority of queries can be anticipated and optimised for. SPARQL, on the other hand, is expected to afford general-purpose access to data, while maintaining reasonable performance.

Another difference between designing a query optimiser for SPARQL queries and SQL is the relative lack of clues available in RDF storage schemas. Relying purely on this small amount of information will result in a poor data traversal strategy, so there is a clear imperative to optimise based on the data items in the database, not just on its schema. An example of where this might be vitally important can be found in (Barabasi and Bonabeau 2003): in the case of a scale free network being formed by the RDF graph, it might be extremely important to know that certain nodes have an overwhelmingly large number of connections when compared to the rest.

With these points in mind, lessons can be learned from the world of existing DBMS on how to construct a query optimiser:

- It is important to know the fastest way to perform an individual segment of a query: a store should be aware of the indexes that are available to be used.
- Generally speaking, it is important to perform the segments of a query that are most selective first: these most reduce the working set, or the set of data that is of interest.
- Knowledge of the underlying data structure is important. There might, for example, be an index on a particular column, but when considering when to run the query segment that this index is

related to, it is also useful to know information like whether the data is clustered on that index. If it is not, access might still be quite slow.
- Plans do not have to be final: it is perfectly reasonable to iterate and construct a new query plan after each segment of the plan has been performed. This allows new information brought about by the results returned to refine the plan.
- If operating in an environment where writes are at all likely to occur, the query optimiser should be aware of locks on the database, and work in such a manner as to avoid waiting on locks where possible.

A dedicated query optimiser for a triple store would probably be somewhat simpler than that for an SQL DBMS, due to the simplicity of SPARQL. We have prior knowledge that most of operations will be selections and joins, and of the structure of all data, at code-time. It is quite possible that we will know the status with regards to sorting, as well. With a large variety of metadata to aid good decision-making, a query optimiser that produces worthwhile results could be implemented.

5 Future Work

There is the potential for a great deal of work in the area of RDF triple storage. There are a host of use cases: distributed, peer to peer stores aggregating a large quantity of data from related institutions, clustered stores designed to maximise performance from a single data node, 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 base 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 individuals computer, with information likely to change regularly.

This report has chosen to focus on issues that affect triple stores of all types, and will offer a solid background for future research should there be a decision to specialise further into areas such as distributed stores. Despite the recent improvements shown by stores such as Virtuoso and YARS2, which offer multiple-billion triple storage, there is a clear need for stores that can handle RDF more efficiently. In section 4 we explored improvements that could be made to existing triple stores with reference to previous database research, and there is a clear opportunity to make RDF storage and retrieval a more feasible operation.

In the future I plan to examine the various subtopics described in this report (in particular storage, indexing, and query optimisation) and work on producing specialised systems to test ideas such as those described in section 4, in order to improve RDF storage.

5.1 Pre-Mini-Thesis Work Packages

The Pre-Mini-Thesis work will focus upon the development and testing of ideas mentioned in Section 4. It is expected that there will be a significant need for prototype-level testing of both these ideas and those implemented in previous systems in order to show which approaches are of value, and in what situations. It is desirable to avoid a full scale implementation of these approaches, as this drains time, and does not provide significant added value.

5.1.1 WP1 – Storage and Indexing

Work Package 1 focuses on analysing different mechanisms for storing and indexing RDF data, with emphasis on discovering which approaches have value for a variety of different purposes (for example, read versus write optimisation). This will involve prototype creation to test theories, in particular some of those mentioned in Section 4, and it is expected that this work package will involve a brief skills update to ensure that the author is capable of implementing these efficiently. This will focus on sharpening skills in low level languages suited to the low level operations that may be necessary, particularly C. It is expected that this work package will at the least provide a comparison of existing techniques and why and when they are effective, and ideally will provide evidence for new ideas that may speed up the process of storing and retrieving RDF data. It is expected that this work package may overlap to some extent with Work Package 2, as a complete evaluation stage may require insight into how the storage and indexing scheme affects the performance of operations.

Method (18 weeks total)
Further reading will be performed in this area to ensure that the author has as complete an understanding as possible. Discover how other models have been tested: ideally it is desirable to avoid anything

beyond proof of concept-level implementation. Packages that allow arbitrary backends (such as Sesame and Jena) may provide some help in this regard.

Tasks

- Literature Search (2 weeks)
  I will revisit literature in the area of storage and indexing, ensuring that I have a complete understanding in this area.
- Algorithm Selection (1 week)
  I will select storage and indexing methods from my own ideas, and those that have already been implemented in the RDF storage world for implementation. The implementation of modes of storage and indexing from existing stores will provide a fair comparison base for new ideas.
- Skills Update (3 weeks)
  I will update my programming skills, particularly focussing on improving my knowledge of low level languages such as C, to ensure my skill set is suitable to the implementation of these prototypes.
- Prototype Implementation (estimate 10 weeks)
  Prototypes of the selected ideas will be implemented.
- Testing (estimate 2 weeks)
  Tests will be devised to examine the efficacy of the selected ideas. There is potential for a publication if these tests provide useful data.

5.1.2 WP2 – Efficient Operations

Work Package 2 considers mechanisms by which operations (such as selections and joins) can be performed efficiently on stored RDF data. This package will build upon work performed in WP1, considering in particular what operations are appropriate for the storage and indexing schemes considered worthwhile in that work package. As noted, this package may overlap with WP1. The package will provide information on how operations can be most efficiently performed, and how they affect both read and write performance. This will inform further development on producing triple stores for any purpose.

Method (12 weeks total)
Similar to that seen in WP1: a literature review around the area, followed by prototype generation and testing.

Tasks

- Literature Search (2 weeks)
  I will revisit literature in the area of efficient operations, ensuring that I have a complete understanding in this area.
- Algorithm Selection (1 week)
  I will select relevant operations for implementation. These will certainly include basic operations like select and join, but may also include more exotic operations such as ‘find neighbour’ if a space filling cube storage/indexing mechanism is implemented in WP1.
- Prototype Implementation (estimate 7 weeks)
  Prototypes of the selected ideas will be implemented.
- Testing (estimate 2 weeks)
  Tests will be devised to examine the efficacy of the selected ideas. There is potential for a publication if these tests provide useful data.

5.2 Post-Mini-Thesis Work Packages

In the final portion of my PhD, I expect to build upon the work created in WP1 and WP2. This area is left somewhat open ended to allow for different directions based on the changing needs of Semantic Web research.

5.2.1 WP3 – Query Optimisation

Query optimisation is a massive item of consideration when attempting to optimise RDF storage and retrieval. Since this area has been less exhaustively researched than others in the preparation of this report, it is expected that there will be a significant amount of further reading required, prior to experimentation with various methods of query optimisation. It is important to consider a variety of issues, such as:

- Effective multi-threading: There is a trend for multiple processor cores to be employed even in basic computer systems, as increasing individual core speed becomes progressively more challenging. Optimisers that can efficiently distribute load amongst these cores would be advantageous.
- Statistics and memory use: Which statistics can affect performance, and how can variable amounts of memory be most efficiently used to speed up queries.

5.2.2 WP4 - Inference

While investigation into inference has not been the focus of this document, performance of reasoning over stored triples is clearly important to the success of the Semantic Web. Research into means of trading off levels of completeness and soundness to allow for queries being performed in a time scale useful to a given end user would certainly be of interest.

References

Abadi, D., A. Marcus, et al. (2007). "Scalable Semantic Web Data Management Using Vertical Partitioning." Proc. VLDB.

Atkinson, M., F. Bancilhon, et al. (1989). "The Object-Oriented Database System Manifesto."

Barabasi, A. and E. Bonabeau (2003). "Scale-free networks." Sci Am **288**(5): 60-9.

Battre, D., F. Heine, et al. "Load-balancing in P2P based RDF stores." Second International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006).

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

Berners-Lee, T., Y. Chen, et al. (2006). "Tabulator: Exploring and Analyzing linked data on the Semantic Web." Proceedings of the 3rd Int. Semantic Web User Interaction Workshop, Athens, USA.

Berners-Lee, T., J. Hendler, et al. (2001). "The Semantic Web." Scientific American **284**(5): 28-37.

Bizer, C. and R. Cyganiak (2006). "D2R Server-Publishing Relational Databases on the Semantic Web." Poster at the 5th International Semantic Web Conference, Athens, USA, November.

Broekstra, J. and A. Kampman (2003). "Inferencing and Truth Maintenance in RDF Schema." Proceedings of the First International Workshop on Practical and Scalable Semantic Systems.

Broekstra, J., A. Kampman, et al. (2003). "Sesame: An Architecture for Storing and Querying RDF Data and Schema Information." Spinning the Semantic Web: 197-222.

Cai, M. and M. Frank (2004). "RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network." Proceedings of the 13th conference on World Wide Web: 650-657.

Carroll, J. J., I. Dickinson, et al. (2004). "Jena: implementing the semantic web recommendations." International World Wide Web Conference: 74-83.

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

Codd, E. (1970). "A relational model of data for large shared data banks." Communications of the ACM **13**(6): 377-387.

Comer, D. (1979). "Ubiquitous B-Tree." ACM Computing Surveys (CSUR) **11**(2): 121-137.

Date, C. J. (1975). An introduction to database systems, Reading, Mass.: Addison-Wesley Pub. Co.

De Roure, D., N. Jennings, et al. (2005). "The Semantic Grid: Past, Present, and Future." Proceedings of the IEEE **93**(3): 669-681.

Ding, L., K. Wilkinson, et al. "Application-Specific Schema Design for Storing Large RDF Datasets." Proc. of the PSSS **3**.

Erling, O. and I. Mikhailov (2006). "RDF Support in the Virtuoso DBMS."

Errling, O. (2006). "Advances in Virtuoso RDF Triple Storage (Bitmap Indexing)."

Guo, Y., A. Qasem, et al. (2006). "Large Scale Knowledge Base Systems: An Empirical Evaluation Perspective." Proc. of the 21st National Conf. on Artificial Intelligence (AAAI 2006), Boston, USA.

Harris, S. (2005). "SPARQL query processing with conventional relational database systems."

Harth, A., J. Umbrich, et al. "YARS2: A Federated Repository for Querying Graph Structured Data from the Web."

Hawthorn, P. and M. Stonebraker (1986). "The use of technological advances to enhance database system performance." Addison-Wesley Series In Computer Science: 106-130.

Hendler, J. (2001). "Agents and the Semantic Web." Intelligent Systems, IEEE [see also IEEE Intelligent Systems and Their Applications] **16**(2): 30-37.

Lassila, O., R. R. Swick, et al. (1999). "Resource Description Framework (RDF) Model and Syntax Specification."

Lawder, J. K. and P. J. H. King "Using Space-Filling Curves for Multi-dimensional Indexing." proceedings of the 17th British National Conference on Databases (BNCOD 17) **1832**: 20-35.

Lee, R. (2004). "Scalability Report on Triple Store Applications." Massachusetts institute of technology.

Olson, M. A., K. Bostic, et al. "Berkeley DB."

Patel-Schneider, P. F., P. Hayes, et al. (2003). "OWL Web Ontology Language; Semantics and Abstract Syntax, W3C Candidate Recommendation." World Wide Web Consortium, http://www.w3.org/TR/2003/CR-owl-semantics-20030818/, August.

Prud'hommeaux, E. and A. Seaborne (2006). "SPARQL Query Language for RDF. W3C Candidate Recommendation." World Wide Web Consortium, April.

Shadbolt, N., N. Gibbins, et al. (2004). "CS AKTive Space, or how we learned to stop worrying and love the semantic Web." Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems] **19**(3): 41-47.

Smith, D. A., A. Owens, et al. (2005). "The evolving mSpace platform: leveraging the semantic web on the trail of the memex." Proceedings of the sixteenth ACM conference on Hypertext and hypermedia: 174-183.

Smith, D. A., A. Owens, et al. (2007). "Challenges in Supporting Faceted Semantic Browsing of Multimedia Collections."

Stonebraker, M. (1980). "Retrospection on a database system." ACM Transactions on Database Systems (TODS) **5**(2): 225-240.

Stonebraker, M. (1981). "Operating System Support for Database Management." Communications of the ACM **24**(7): 412-418.

Stonebraker, M., G. Held, et al. (1976). "The design and implementation of INGRES." ACM Transactions on Database Systems (TODS) **1**(3): 189-222.

Stonebraker, M., S. Madden, et al. "The End of an Architectural Era (It's Time for a Complete Rewrite)."

Stonebraker, M., E. O'Neil, et al. (2005). "C-store: a column-oriented DBMS." Proceedings of the 31st international conference on Very large data bases: 553-564.

Taylor, K., R. Gledhill, et al. (2005). "A Semantic Datagrid for Combinatorial Chemistry." Grid Computing, 2005. The 6th IEEE/ACM International Workshop on: 148-155.

Taylor, K. R., R. Gledhill, et al. (2006). "Bringing Chemical Data onto the Semantic Web." J. Chem. Inf. Model **46**(3): 939-952.

Wilkinson, K. (2006). "Jena Property Table Design." Proceedings of the Jena Users Conference, Bristol, England, May.

Wilkinson, K., C. Sayers, et al. (2003). "Efficient RDF Storage and Retrieval in Jena2." Proceedings of SWDB **3**: 7-8.

Wood, D., P. Gearon, et al. (2005). "Kowari: A Platform for Semantic Web Storage and Analysis." Proceedings of the 14th International WWW Conference.

Youssefi, K. and E. Wong (1979). "Query Processing In A Relational Database Management System." Very Large Data Bases, 1979. Fifth International Conference on: 409-417.