BUILDING A GRAPH BASED RDF STORE TO MANIPULATE RDF DATA EFFICIENTLY

Ravindra Sampath Ranwala

138227T

Degree of Master of Science/Master of Engineering

Department Computer Science and Engineering

University of Moratuwa
Sri Lanka

March 2015

BUILDING A GRAPH BASED RDF STORE TO MANIPULATE RDF DATA EFFICIENTLY

Ravindra Sampath Ranwala

138227T

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Thesis/Dissertation submitted in partial fulfillment of the requirements for the degree Master of Science/Master of Engineering in Computer Science and Engineering

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

March 2015

DECLARATION

I declare that this is my own work and this thesis/dissertation 2 does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis/dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date:

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

The above candidate has carried out research for the Masters/MPhil/PhD thesis/Dissertation under my supervision.

Signature of the supervisor: Date:

ABSTRACT

Due to the expansion of semantic web technologies, Resource Description frameworks (RDFs) and triple stores became more prevalent. Since there is a huge amount of RDF data available, managing them in a proper and efficient manner is challenging. Many Triple stores were implemented to support the queries related to semantic web. The queries submitted in this context is called as SPARQL queries which are read dominant. These SPARQL queries needs to be answered quickly and efficiently. RDF data is stored in <subject, predicate, object> form and which is called as a triple. A typical triple store contains billions of triples in the above form.

Much work has been devoted to handle RDF data efficiently. But state of the art systems still cannot handle web scale RDF data effectively. Most existing systems store and index data in particular ways. For an example some systems uses relational tables, bitmap matrix to optimize SPARQL query processing on RDF data. This relational approach suffers from high Join cost and large intermediate results. Some have used prolog inference engine to handle RDF data. This also have some limitations given a huge amount of RDF data.

A modern approach is to model the RDF data in its native Graph form. This approach requires new algorithms to build the graph and graph exploration techniques to answer SPARQL queries. This yields no join cost and very small intermediary results. Also this approach yields less query execution time for complex SPARQL queries.

The objective of this research is to build a graph based triple store for Apache Cassandra. It uses Apache Jena Graph Processing framework to build and explore the RDF graph. Towards the end, it conducts a performance benchmark of this RDF store with some other RDF store implementations using DBPedia dataset and sample queries and proves that this graph based approach outperforms other RDF store implementations.

University of Moratuwa, Sri Lanka
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

ACKNOWLEDGEMENT

I would like to express my special appreciation and thanks to my supervisor Dr. Amal Shehan Perera, you have been a tremendous mentor to me. I would like to thank you for guiding me through your experience to make this research more worthwhile. Also I would like to appreciate the extended support and guidance given by Dr. Srinath Perera through his experience and knowledge, which was really helpful for me to complete this successfully.

I would also like to take this opportunity to thank Dr. Malaka Walpola for guiding us towards this work throughout Research Seminar lecture sessions over a year and for the extended support and kindness granted to us. At last but not the least, I would prefer thank to all the academic staff members for helping, guiding, encouraging us and disseminating knowledge throughout the programme.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

TABLE OF CONTENTS

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Table of Contents

DECLARATION ii

ABSTRACT iii

ACKNOWLEDGEMENT iv

TABLE OF CONTENTS v

LIST OF FIGURES viii

LIST OF TABLES ix

LIST OF ABBREVIATIONS x

1 INTRODUCTION 1

1.1 Semantic web, RDF and Triples 2

1.2 Challenges faced by existing RDF management systems. 3

1.3 New approaches in managing RDF data efficiently 4

1.4 Existing Graph Processing Frameworks. 6

1.5 The Problem/Opportunity 7

1.6 Objectives 7

2 LITERATURE REVIEW 9

2.1. RDF Store - What, Why and How 10

2.2 Apache Cassandra 12

2.3 Different approaches to build a triple store 16

2.3.1 Relational Approach 16

2.3.2 RDF data centric storage 17

2.3.3 RDF graph based approach 18

2.3.4 Hybrid Approaches 25

2.4 Benchmarking RDF stores for performance evaluation 27

3 METHODOLOGY 29

3.1. Existing Solutions 30

3.2 Proposed Solution 31

3.3 Solution Architecture 32

3.4 Solution Implementation 34

3.4.1 Populating data into Cassandra Cluster 36

3.4.2. Building the RDF Graph 37

3.4.3. Querying the RDF Graph 38

3.4.4: Dropping the RDF Store 39

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

3.4.5: Techniques used to render RDF/XML results on the webpage 40
3.4.6: Solution Extensibility and Flexibility...43
4 USE CASE SCENARIOS 44
4.1. Build the RDF graph...45
4.2 Executing SPARQL query..47
4.3 Rendering Results 48
5 EVALUATION AND RESULT...49
5.1 RDF Store Benchmarking 50
5.1.1 Dataset Generation..51
5.1.1.1 RDF/XML 51
5.1.1.2 Turtle 53
5.1.1.3 N-Triples 54
5.1.2 Tested RDF Stores..55
5.1.3 Query Generation 56
5.1.4. Benchmark Configuration 60
5.1.5. Benchmark Metrics...60
6 FUTURE WORK..63
7 CONCLUSION...66
References 68

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

LIST OF FIGURES

Figure Index	Name	Page
Figure 1.1	Set of triples	02
Figure 2.1	RDF Storage Example	12
Figure 2.2	Peer to Peer architecture of Apache Cassandra	13
Figure 2.3	Keyspace in Cassandra	15
Figure 2.4	An example RDF graph	18
Figure 2.5	Distributed query processing framework	19
Figure 2.6	The query graph	20
Figure 2.7:	TripleRush index graph that is created for the triple vertex	22
Figure 2.8	Query execution on the relevant part of the index	23
Figure 2.9	A Motivated Example	26
Figure 2.10	Newly Proposed Idea	26
Figure 3.2.1	Usecase diagram for RDF Graph building	31
Figure: 3.3.1	High level class diagram of the Graph based RDF store	32
Figure 3.4.1	Jena Framework Architecture	35
Figure: 3.4.2	Populating data into Cassandra Cluster	36
Figure: 3.4.3	Building the RDF Graph	37
Figure 3.4.4	Querying the RDF Graph	38
Figure 3.4.5	Drop the RDF Store	39
Figure 3.4.6	SPARQL query result in RDF/XML form	40
Figure 3.4.7	XSLT used for transformation	41
Figure 3.4.8	Sample jsp/struts code used to render results	42
Figure 4.1	Web Client Building RDF Graph	46
Figure 4.2	Execute Query against the RDF graph	47
Figure 4.3	Rendering Results of the SPARQL query	48
Figure 5.1.1.1	Multiple resources as RDF/XML	52
Figure 5.1.1.2	Multiple resources as Turtle	53
Figure 5.1.1.3	Multiple Resources as N-Triples	54
Figure 5.1.3.1	Query 1	56
Figure 5.1.3.2	Query 2	56
Figure 5.1.3.3	Query 3	57
Figure 5.1.3.4	Query 4	58
Figure 5.1.3.5	Query 5	59
Figure 5.1.5.1	SPARQL Query Execution time	61
Figure 6.1	Distributed Implementation of the RDF store	64

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

LIST OF TABLES

Table Index	Name	Page
Table 1	Base tables and bound variables	17
Table 2	Benchmark Configuration	60
Table 3	Performance Benchmark results	60

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

LIST OF ABBREVIATIONS

Abbreviation	Description
CQL	Cassandra Query Language
DBMS	Database Management Systems
RDBMS	Relational Database Management Systems
RDF	Resource Description Framework
SPARQL	RDF Query language

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

1 INTRODUCTION

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

The Introduction section is organized as follows. First some basic concepts related to the RDF data such as triples, SPARQL queries etc is explained. Then the challenges faced by the existing RDF data management systems are elaborated. This opens up avenues to discuss about new approaches to manage RDF data efficiently. One of such a new approach is called Graph processing frameworks which is elaborated next. The problem/opportunity is explicitly stated next while the conclusion section pays attention on objectives of the research.

1.1 Semantic web, RDF and Triples

RDF [1] data is becoming increasingly more available: The semantic web movement towards a web 3.0 world is proliferating a huge amount of RDF data. Commercial search engines including Google and Bing are pushing web sites to use RDF to explicitly express the semantics of their web contents.

Before moving forward some basic concepts related to RDF is explained here. An RDF data set consists of statements in the form of (subject, predicate, object). Each statement, also known as a triple, is about a fact, which can be interpreted as subject has a predicate property whose value is object. For example, a movie knowledge base may contain the triples[2] as shown in figure 1.1.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

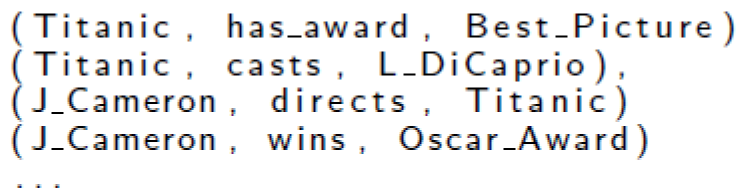


Figure 1.1: Set of triples

These triples can be stored in different ways such as relational tables, Prolog based inference engines or in its native graph form. Those are different approaches in building triple stores.

An RDF dataset can be considered as representing a directed graph, with entities (i.e. subjects and objects) as nodes and relationships (i.e. predicates) as directed edges [3]. SPARQL [4] is the standard query language for retrieving data stored in RDF format.

1.2 Challenges faced by existing RDF management systems.

Mainly RDF data management systems are facing two challenges which are system scalability and generality. The challenge of scalability is particularly important. Tremendous efforts have been devoted to build high performance RDF systems and SPARQL engines. But still the scalability remains as the biggest hurdle. Mainly RDF data is highly connected graph data, and SPARQL queries are like sub graph matching queries. But most approaches model RDF data as a set of triples, and use RDBMS for storing, indexing, and query processing. These approaches do not scale as processing a query often involves a large number of join operations that produce large intermediate results. Several distributed RDF systems such as SHARD [5], YARS [6] were introduced recently. However, they still model RDF data as a set of triples.

The second challenge lies in the generality of RDF systems. State-of-the-art systems are not able to support general purpose queries on RDF data. In fact, most of them are optimized for SPARQL only, but a wide range of meaningful queries and operations on RDF data cannot be expressed in SPARQL.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

There are few distributed in-memory RDF systems that are capable of handling web scale RDF data (billion or even trillion triples). Unlike existing systems that use relational tables (triple stores) or bitmap matrices to manage RDF, these systems model RDF data in its native graph form (i.e., representing entities as graph nodes, and relationships as graph edges). Such a memory-based architecture that logically and physically models RDF in native graphs opens up a new paradigm for RDF data management. It not only leads to new optimization opportunities for SPARQL query processing, but also supports more advanced graph analytics on RDF data [1].

Storing RDF graphs in disk-based triple stores is not a feasible solution since random accesses on hard disks are notoriously slow. Although sophisticated indices can be created to speed up query processing, they introduce excessive join operations, which become a major cost for SPARQL query processing [2].

Some data models RDF data as an in-memory graph. Naturally, it supports fast random accesseson the RDF graph. But in order to process SPARQL queries efficiently, the issues such

as how to reduce the number of join operations, and how to reduce the size of intermediary results needs to be addressed.

1.3 New approaches in managing RDF data efficiently

Some researches addressed novel techniques that use efficient in-memory graph exploration instead of join operations for SPARQL processing. In one approach SPARQL query was decomposed into a set of triple patterns, and conduct a sequence of graph explorations to generate bindings for each of the triple pattern. The exploration-based approach uses the binding information of the explored sub graphs to prune candidate matches in a greedy manner. In contrast, previous approaches isolate individual triple patterns, that is, they generate bindings for them separately, and make excessive use of costly join operations to combine those bindings, which inevitably results in large intermediate results [2].

These new query paradigm greatly reduces the amount of intermediate results, boosts the query performance in a distributed environment, and makes the system scale. Those new systems achieves several orders of magnitude speed-up on web scale RDF data over the state-of-the-art RDF systems.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Since these new approaches models data as a native graph, which enables a large range of advanced graph analytics on RDF data. For example, random walks, regular expression queries, reachability queries, distance oracles, community searches can be performed on web scale RDF data directly [2].

Some approaches build a parallel in-memory triple store designed to address the need for efficient graph stores that quickly answer queries over large-scale graph data. These systems borrow from the database literature to investigate efficient means for storing large graphs and retrieving sub graphs, which are usually defined via a pattern matching language such as SPARQL. Other approaches focus on adapting triple stores to a distributed setting.

Trinity.RDF [2] is a graph engine for SPARQL queries that was built on the Trinity distributed graph processing system. To answer queries, Trinity.RDF represents the graph with adjacency lists and combines traditional query processing with graph exploration.

Joins are the major operator in SPARQL query processing. Trinity.RDF outperforms existing systems by orders of magnitude because it replaces expensive join operations by efficient graph exploration [2].

Another approach introduced by set of researchers was named as TripleRush [7], a triple store which is based on an index graph, where a basic graph pattern SPARQL query is answered by routing partially matched query copies through this index graph. TripleRush routes query descriptions to data. For this reason, TripleRush does not use any joins in the traditional sense but searches the index graph in parallel [7].

TripleRush is built on top of the distributed and parallel graph processing framework known as SIGNAL/COLLECT. [8] The index structure is represented as a graph where each index vertex corresponds to a triple pattern. Partially matched copies of a query are routed in parallel along different paths of this index structure [7].

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

TripleRush takes less than a third of the time to answer queries compared to the fastest of the three state-of-the-art triple stores. On individual queries TripleRush is up to three orders of magnitude faster than other triple stores [7]. This system mainly focuses on investigating efficient means for storing large graphs and retrieving subgraphs, which are usually defined via pattern matching languages such as SPARQL. Also it focuses on adapting a triple store to a distributed setting. MapReduce has been used to aggregate results from multiple single node RDF stores in order to support distributed query processing.

Distributed graph processing frameworks can offer more flexibility for scalable querying of graphs. Some of the triple stores built on top of abstractions such as SIGNAL/COLLECT in TripleRush.

TripleRush, is a triple store which is based on an index graph, where a basic graph pattern SPARQL query is answered by routing partially matched query copies through this index graph. Whilst traditional stores pipe data through query processing operators, TripleRush routes query descriptions to data. For this reason, TripleRush does not use any joins in the traditional sense

but searches the index graph in parallel. TripleRush is implemented on top of Signal/Collect, which is a scalable, distributed, parallel and vertex-centric graph processing framework [7].

It has experimentally been shown that TripleRush outperforms the other triple stores by a factor ranging from 3.7 to 103 times in the geometric mean of all queries. Also the memory usage for TripleRush, which is comparable to that of traditional approaches.

1.4 Existing Graph Processing Frameworks.

SIGNAL/COLLECT [8] is a parallel and distributed large-scale graph processing system written in Scala. The model is suitable for expressing data-flow algorithms, with vertices as processing stages and edges that determine message propagation. In contrast to other systems Signal/Collect supports different vertex types for different processing tasks. Signal/Collect also supports features such as bulk-messaging and Pregel-like message combiners to increase the message-passing efficiency [7].

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Signal/Collect supports asynchronous scheduling, where different partial computations progress at their own pace, without a central bottleneck. The system is based on message-passing, which means that no expensive resource locking is required. These two features are essential for low-latency query processing [7].

Many systems such as SW-Store, Hexastore and RDF-3x are single machine systems. But Trinity is a completely distributed system which spans across multiple nodes. As the size of RDF data keeps soaring, it is not realistic for single-machine approaches to provide good performance. Recently, several distributed RDF systems were introduced to alleviate these challenges.

1.5 The Problem/Opportunity

Apache Cassandra is a Distributed, No-SQL [9], multi-tenant and multi data centric database [10] which is being used heavily these days. Building a scalable triple store for Apache Cassandra exponentially increase the value of Cassandra in semantic web domain. There are more triple stores built for relational databases. Most of these triple stores suffer from performance and scalability issues which are inherent to relational model due to costly joins and large intermediate results. But by building a triple store for a distributed No-SQL database like Apache Cassandra leads to alleviate these issues and build a scalable RDF store which outperforms RDF stores built on top of Relational databases. Since Apache Cassandra is used by eBay, Twitter, Cisco [11] etc it has large active data set. The largest known Cassandra cluster has over 300 TB of data in over 400 machines. This motivates us to build a distributed, scalable RDF store to answer user queries related to them efficiently.

Though this implementation is specific to Apache Cassandra persistence layer, any user can implement their own Data Access Layer implementation to fetch RDF data from any database or file system and pass those data to the RDF graph processing engine to build an in-memory graph. Therefore if someone already has some RDF data in any format, they can plug their data sources easily to this implementation and get the benefits of this graph based approach instead of costly joins. That implies our solution is much more extensible and reusable.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

1.6 Objectives

The main objective of this research is to build a scalable, in memory RDF store for Apache Cassandra. Apache Cassandra is a No-SQL and distributed database system. Even though the current implementation is done only to fetch RDF data from Apache Cassandra, any other RDF data source can be easily plugged with this by merely implementing their own Data Access layer implementation to fetch the data from any custom source and switching the data access implementation used inside the graph processing engine. The custom source can be any other RDBMS, file system resource etc. The plan is to implement an in memory Graph based RDF store for a given set of triples stored in Apache Cassandra. An algorithm needs to be implemented to build an in memory graph given a set of triples. Subjects and objects are represented as nodes in the graph whereas predicates are represented as directed edges in the graph.

Given a SPARQL query an algorithm has to be implemented to explore the directed in memory graph and find the answer to that query. This algorithm should be designed carefully so that the way it explores the graph should NOT deteriorate the performance while answering the SPARQL queries. While answering the SPARQL queries, the order of exploration plays an important role.

This graph based approach is chosen since it does NOT incur any join costs and the intermediate results are much smaller, which is almost negligible compared to the relational approach. After the successful implementation of the RDF graph building and graph exploration algorithms a benchmark needs to be conducted. This benchmark compares the newly implemented system with existing RDF stores and tells whether the new approach outperforms the existing ones or not.

These solutions are planned to be implemented using Java language. But this might have some impact on performance. Ideal language for this would be C language. But given the Object oriented high level API support in java language and due to the high familiarity Java language is used instead of C. But that selection will still lead to some performance impact which may be discovered during the benchmarking stage of the research work.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

2 LITERATURE REVIEW

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

The literature review section is organized as follows. First some basic concepts pertaining to the RDF store is covered. Since the triple store is built on Apache Cassandra, it is explained next. Some time is devoted to elaborate different approaches to build a triple store today. Since benchmarking plays an important role in this context, next section is dedicated to explain benchmarking RDF stores for performance evaluation and which ultimately concludes the literature review section.

2.1. RDF Store - What, Why and How

Resource Description Framework (RDF) was designed with the initial goal of developing metadata for the Internet. [12] The vision of the Semantic Web has brought about new challenges at the intersection of web research and data management. One fundamental research issue at this intersection is the storage of the Resource Description Framework (RDF) data: the model at the core of the Semantic Web. The core of the Semantic Web is built on the Resource Description Framework (RDF) data model. RDF provides a simple syntax, where each data item is broken down into a <subject, predicate, object> triple. This can be interpreted as subject has a predicate property whose value is object [2]. A typical triple store holds a multi millions or billions of such triples within its RDF triple data model. Figure 2.1 shows some sample triple patterns. As an example take the pattern <person1, name, mike>. In this person1 is the subject, Mike is the object. Name is the predicate property here. This triple can be read like, Person1 has a name and its value is Mike. Actually the predicate describes the relationship between the subject and object.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Efficient and scalable management of RDF data is a fundamental challenge at the core of the Semantic Web. Many popular RDF storage solutions use relational databases to achieve this scalability and efficiency. Figure 2.1 shows different RDF storage mechanisms available in relational DBMS.

SPARQL is the typical query language which is submitted to an RDF store. The SPARQL query submitted here is “Find people who has both a name and a website.” This SPARQL query is more or less same as the SQL query sent to a typical relational database. These type of queries are more prominent in Semantic web especially in search engines like Google, yahoo and bing

etc. The main purpose of a triple store is to answer these queries efficiently. A triple store can be built on top of any DBMS, File system etc. It can be a relational DBMS, post Relational DBMS, No-SQL database, distributed file system etc. But building a scalable, high performance RDF store is a huge challenge and it is not an easy task. This challenge is going to be addressed throughout this paper.

Triple stores are the backbone of increasingly many Data Web applications. It is thus evident that the performance of those stores is mission critical for individual projects as well as for data integration on the Data Web in general [13]. With the W3C SPARQL standard [4] a vendor-independent query language for the RDF triple data model exists. SPARQL is based on powerful graph matching allowing to bind variables to fragments in the input RDF graph. In addition, operators akin to the relational joins, unions, left outer joins, selections and projections can be used to build more expressive queries [14].

It is evident that the performance of triple stores offering a SPARQL query interface is mission critical for individual projects as well as for data integration on the Web in general.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

<Person1, Name, Mike>
<Person1, Website, ~mike>
<Person2, Name, Mary>
<Person3, Name, Joe>
<Person4, Name, Kate>
<City1, Population, 200K>
<City2, Population, 300K>

Query
"Find all people that have both a *name* and *website*"

(a) RDF Triples

TS		
Subj	Prop	Obj
Person1	Name	Mike
Person1	Website	~mike
Person2	Name	Mary
Person3	Name	Joe
Person4	Name	Kate
City1	Pop.	200K
City2	Pop.	300K

```
SELECT T1.Obj,T2.Obj
FROM TS T1,TS T2
WHERE T1.Prop=Name AND
T2.Prop=Website AND
T1.Subj=T2.Subj;
```

(b) Triple Store

NameWebsite		
Subj	Name	Website
Person1	Mike	~mike
Person2	Mary	NULL
Person3	Joe	NULL
Person4	Kate	NULL

Population	
Subj	Pop.
City1	200K
City2	300K

```
SELECT T.Name,T.Website
FROM NameWebsite T
Where T.Website IS NOT NULL;
```

(c) N-ary Table

Name	
Subj	Obj
Person1	Mike
Person2	Mary
Person3	Joe
Person4	Kate

Website	
Subj	Obj
Person1	~mike

Population	
Subj	Obj
City1	200K
City2	300K

```
SELECT T1.Obj,T2.Obj
FROM Name T1,Website T2
WHERE T1.Subj=T2.Subj;
```

(d) Binary Tables

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure . 2.1. RDF Storage Example [15]

2.2 Apache Cassandra

Apache Cassandra is a Distributed, high performance, extremely scalable, fault tolerant (i.e. no single point of failure), post relational database solution [11]. Post relational means that Cassandra is not a typical relational database. Cassandra can serve as both operational data store for online/transactional applications, and as a read intensive database for business intelligence systems. Cassandra is basically a combination of Google Bigtable and amazon Dynamo.

Cassandra was designed with the understanding that system/hardware failures can occur. It was designed as a peer to peer distributed database management system as shown in the figure 2.2. All the nodes are the same and there's no concept of a master node or main node here. Cassandra automatically partitions the data you write to the database across all the nodes in the clusters and

you control the data replication within Cassandra to ensure fault tolerance. That means you determine how many copies of the piece of data you want duplicated on the nodes that are participating in a particular database cluster. Because of the peer to peer design Cassandra is read and write anywhere style architecture. You can read from and write to any node in Cassandra. Cassandra uses a gossip protocol to communicate with the various nodes participated in the cluster. When you write data to Cassandra it is first written into the commit log to ensure the durability. Then data is written to an in memory structure called as a memtable. When that memtable becomes full it then pushes to the disk.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure. 2.2. Peer to Peer architecture of Apache Cassandra [10]

The schema used in Cassandra is a row-oriented column structure design. In Cassandra there is a concept called a keyspace which is similar to a database in DBMS world. Column family in Cassandra is the core object which is used to manage the data and this is somewhat similar to a relational database table but it is more flexible/dynamic. Columns and rows in a column family can be indexed, you can have a primary key index or other columns may be indexed as well.

Cassandra has variety of different features and benefits. Cassandra is very well known for being Big-Data capable. You can add new nodes to the cluster online and that gives you linear

performance gains. Cassandra offers no single point of failure. Cassandra is also capable of replicating data between different racks. It is smarter enough to position particular piece of data in one rack and a redundant copy in another rack. If the first rack fails the data is actually safe on the second rack.

Cassandra offers very easy replication capability. Everything is transparently handled by Cassandra. This replication can be done within a single data center or multiple datacenters. It is often easy to replicate data among different geographically dispersed data centers and alike. Because of the design Cassandra exploits all the benefits that you would expect in cloud computing.

There's no need for special type of secondary caching layer and the programming that goes with it in Cassandra. Cassandra also offers what is called tunable data consistency. That means it offers options between very strong data consistency and eventual consistency depending on the use case. This can be handled on a per operation basis. If you have a particular transaction that you want to ensure written to all the nodes you can make sure that happens. All nodes responds or the actual write fails. Whereas in some use cases where it might be ok to have write goes to one node and eventually propagates to other nodes. You can control that data consistency so that it is called as tunable data consistency. Also this tunable data consistency handles multiple datacenter operations.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Cassandra uses column family to store the data inside the Cassandra keyspace as shown in figure 2.3. This schema withcolumn family is much more dynamic and flexible than the schema can be seen in a relational database. It handles structured, semi-structured and unstructured data. Any modification to the column family can be done online without any downtime. Data in Cassandra can be indexed by a primary key as well as secondary indexes that can be created on various columns inside the column family. Cassandra also offers very strong data compression. It uses Google's snappy data compression algorithm. Internal tests at datastax show up to 80% + compression for row data in some use cases. There is no performance penalty due to the compression. Overall performance improves due to less physical I/O that occurs with

compressed data. Cassandra uses CQL which complements to traditional SQL language in relational world.

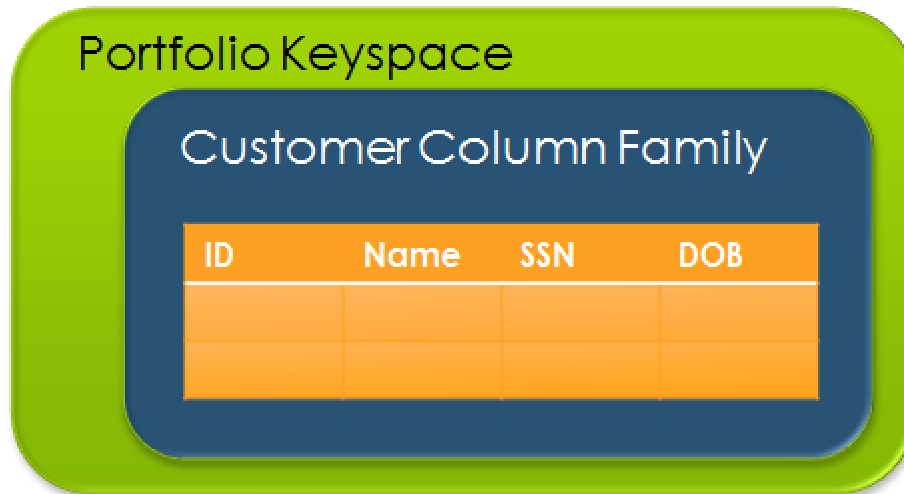


Figure 2.3. Keyspace in Cassandra [11]

All nodes in the cluster communicate with each other through the gossip protocol. If a node goes down, the cluster detects the failure and automatically routes user requests away from the failed machine. Once the failed node is operational again, it rejoins the cluster, and its data is brought back up to date via the other nodes.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Cassandra is a logical choice for enterprises that need high degrees of uptime, reliability, and very fast performance. Leading companies like Netflix, Twitter, Cisco, HP, Motorola, Rackspace, Ooyala, Openwave, and many more rely upon Cassandra to manage the data needs of their critical production applications. [16]

2.3 Different approaches to build a triple store

2.3.1 Relational Approach

The main challenges faced by most of the RDF data management systems are scalability and efficiency. Many different approaches exist to manage RDF data, each with its own advantages and disadvantages. Efficient and scalable management of RDF data is a fundamental challenge at the core of the Semantic Web. Many popular RDF storage solutions use relational databases to achieve this scalability and efficiency. Many state-of-the-art RDF systems store RDF data as a set of triples in relational tables, and therefore, they rely excessively on join operations for processing SPARQL queries.

The simplest way to store RDF triples comprises a relation/table of three columns, one each for subjects, predicates and objects. However, this approach suffers from lack of scalability and abridged query performance, as the single table becomes long and narrow when the number of RDF triples increases. The approach is not scalable since the table is usually located on a single machine. In addition, query performance is diminished since a query requires several self-joins with the same table. [17]

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

In general, query processing consists of two phases [18]: The first phase is known as the scan phase. It decomposes a SPARQL query into a set of triple patterns. For an example consider the Set of triples shown in Figure 1.1. Suppose we need to retrieve the cast of an award-winning movie directed by an award-winning director using the following query [2].

```
SELECT ?movie , ? actor WHERE {
? director wins ? award .
? director directs ? movie .
? movie has_award ? movie_award .
? movie casts ? actor .}
```

The triple patterns for the above query can be written as ?director wins ?award, ?director directs ?movie, ?movie has award ?movie award, and ?movie casts ?actor. For each triple pattern tables or indices are scanned to generate bindings. The base tables that contain bindings

are shown in the table 1. These base tables are joined to find the answers to the queries.

?director	?award
J_Cameron	*Oscar_Award*
G_Lucas	*Saturn_Award*

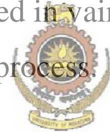
?director	?movie
P_Haggis	*Crash*
J_Cameron	*Titanic*
J_Cameron	*Avatar*

?movie	?movie_award
Titanic	*Best_Picture*
Crash	*Best_Picture*

?movie	?actor
Crash	*D_Cheadle*
Titanic	*L_Dicaprio*
Avatar	*S_Worthington*
Star War VI	*M_Hamill*

Table 1: Base tables and bound variables. [2]

Some sophisticated techniques are used to optimize the order of joins to improve the query performance. Still this approach has two inherent limitations. (1) It uses costly join operations. (2) The scan-join process produces large redundant intermediary results. Most intermediary join results are produced in vain. Moreover useless intermediate results are detected at the later stages of the join process.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

2.3.2 RDF data centric storage

RDF data centric storage is one way of alleviating these tradeoffs inherent in relational model. It improves the relational approach by (1) reducing, on average, the need to join tables in a query by storing as much RDF data together as possible, and (2) reducing the need to process extra data by tuning extra storage (i.e., null storage) to fall below a given threshold. This approach involves two phases: clustering and partitioning.

The clustering phase scans the RDF data to find groups of related properties (i.e., properties that always exist together for a large number of subjects). Properties in a cluster are candidates for storage together in an n-array table. Likewise, properties not in a cluster are candidates for storage in binary tables.

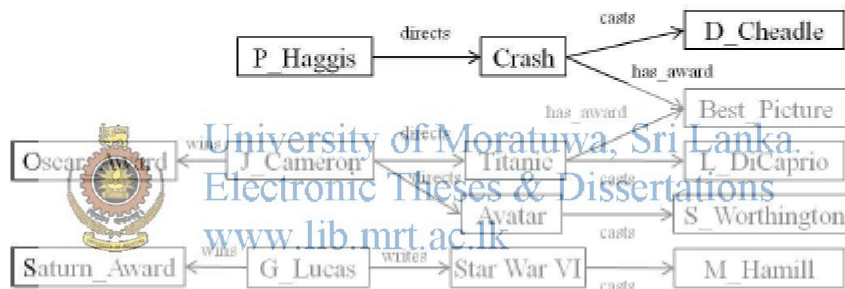
The partitioning phase takes clusters from the clustering phase and balances the tradeoff between storing as many RDF properties in clusters as possible while keeping null storage to a minimum

(i.e., below a given threshold).

By considering all these tradeoffs this approach comes up with an optimal schema to manage EDF data efficiently. The data centric schema creation approach improves query processing compared to previous approaches. Results show that the data centric approach achieves orders of magnitude performance improvements over the triple store.

2.3.3 RDF graph based approach

This is the new approach adopted recently that greatly improves the performance of SPARQL query processing. The idea is to use graph exploration instead of joins. Set of triples shown in Figure 1.1 can be mapped to the RDF graph shown in figure 2.4.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 2.4: An example RDF graph [2]

Using this approach unnecessary intermediary results can be pruned down efficiently. The above intuition is only valid if graph exploration can be implemented more efficiently than join. If the RDF graph is managed by relational tables, triple stores, or disk-based key-value stores, then join operations needs to be used to implement graph explorations which means graph exploration cannot be more efficient than join.

This approach uses native graphs to model RDF data. The order of exploration is important. Starting with the highly selective pattern can prune a lot of unnecessary candidate bindings. Thus, query plans need to be carefully optimized to pick the optimal exploration order, which is not trivial.

One graph based triple store implementations is known as Trinity RDF. Trinity model and store RDF data as a directed graph. Each node in the graph represents a unique entity, which may

appear as a subject and/or an object in an RDF statement. Each RDF statement corresponds to an edge in the graph. Edges are directed, pointing from subjects to objects. Furthermore, edges are labeled with the predicates. To ensure fast random data access in graph exploration, RDF graphs are stored in memory.

Web scale RDF graph may contain billions of entities (nodes) and trillions of triples. It is unlikely that a web scale RDF graph can fit in the RAM of a single machine. Trinity.RDF is based on Trinity [2], which is a distributed in-memory key-value store.

Trinity RDF randomly partitions an RDF graph across a cluster of commodity machines by hashing on the nodes. Thus, each machine holds a disjoint part of the graph. Given a SPARQL query, a search is performed in parallel on each machine. During query processing, machines may need to exchange data as a query pattern may span multiple partitions. Figure 2.5 shows the high level architecture of Trinity.RDF.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

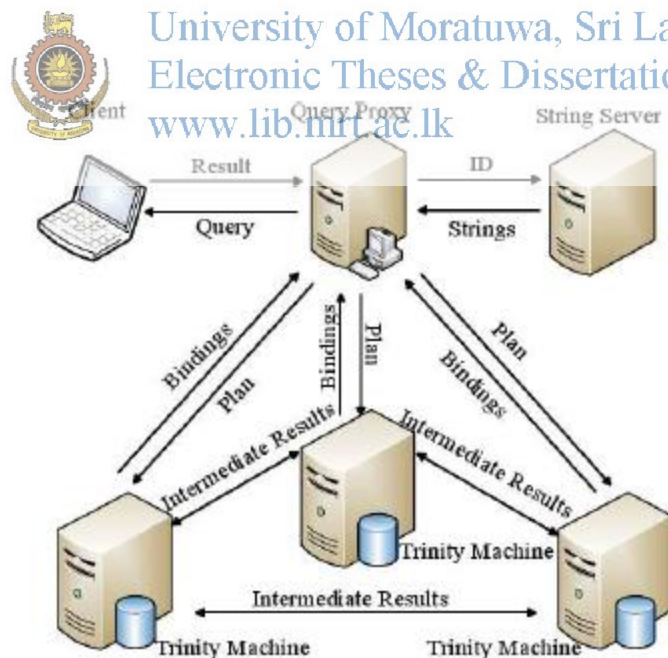
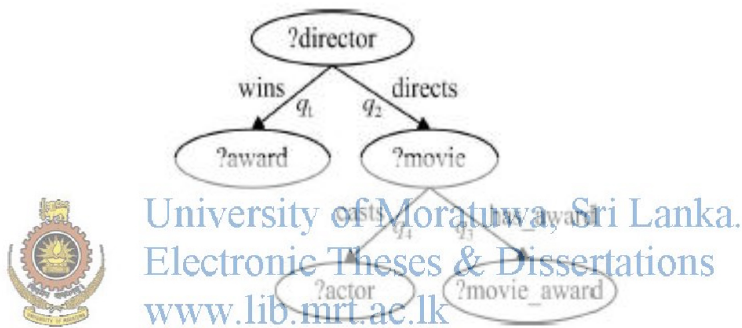


Figure 2.5: Distributed query processing framework [2]

A user submits his query to a proxy. The proxy generates a query plan and delivers the plan to all the Trinity machines, which hold the RDF data. Then, each machine executes the query plan

under the coordination of the proxy. When the bindings for all the variables are resolved, all Trinity machines send back the bindings (answers) to the proxy where the final result is assembled and sent back to the user. The proxy plays an important role in the architecture. Each Trinity machine not only communicates with the proxy. They also communicate among themselves during query execution to exchange intermediary results. All communications are handled by a message passing mechanism built in Trinity.

SPARQL query Q is represented by a query graph G. Nodes in G denote subjects and objects in Q, and directed edges in G denote predicates. Figure 2.6 shows the query graph corresponding to the example query, and lists the 4 triple patterns in the query as q1 to q4.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

- q_1: $(?director\ wins\ ?award)$
- q_2: $(?director\ directs\ ?movie)$
- q_3: $(?movie\ has_award\ ?movie_award)$
- q_4: $(?movie\ casts\ ?actor)$

Figure 2.6 The query graph [2]

With G defined, the problem of SPARQL query processing can be transformed to the problem of subgraph matching. However most existing RDF query processing and subgraph matching algorithms rely excessively on costly joins, which cannot scale to RDF data of billion or even trillion triples. Instead Trinity uses efficient graph exploration in an in-memory key-value store to support fast query processing.

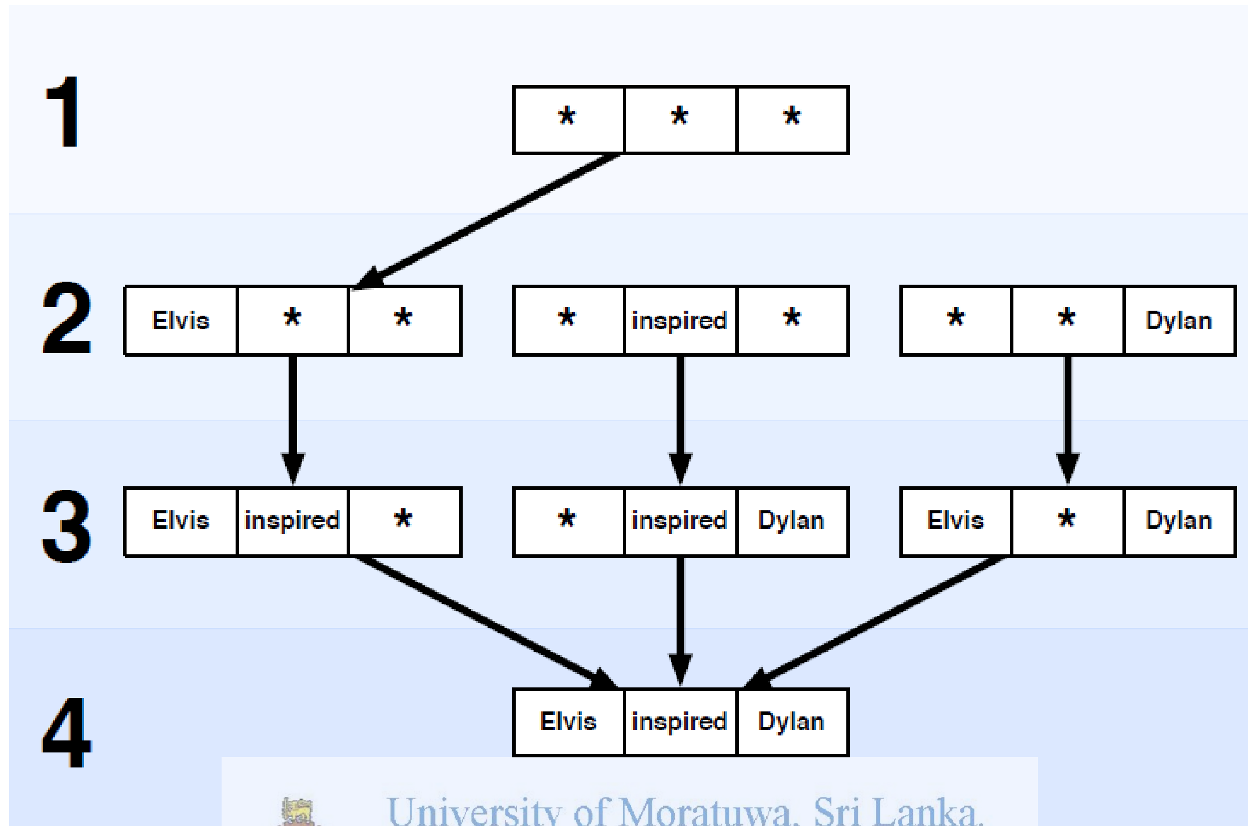
The exploration is conducted as follows: First SPARQL query Q is decomposed into an ordered sequence of triple patterns: q1; ... ; qn. Then, matches for each qi is found, and from each match,

the graph is explored to find matches for qi+1. Furthermore, the exploration is carried out on all distributed machines in parallel. In the final step, matchings for all the individual triple patterns are gathered to a centralized query proxy and combined together to produce the final result.

Another graph based approach to build an RDF store is known as TripleRush. It is a parallel in memory triple store which is designed to answer the queries quickly over large scale graph data. TripleRush is built on top of a parallel graph processing framework known as SIGNAL/COLLECT. TripleRush is a store which is based on an index graph structure. The basic SPARQL query is answered by routing this index graph structure. Instead of using traditional joins like other stores , TripleRush searches the index graph in parallel.

TripleRush is a highly parallel triple store, based on graph-exploration. SIGNAL/COLLECT is a scalable graph processing [19] framework which is used as a foundation for TripleRush. SIGNAL/COLLECT [8] is a parallel and distributed large scale graph processing system written in SCALA. The TripleRush architecture is based on three different types of vertices. Index and triple vertices form the index graph. In addition, the TripleRush graph contains a query vertex for every query that is currently being executed.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 2.7: TripleRush index graph that is created for the triple vertex [7].

Triple vertices are illustrated on level 4 of Figure 2.7 and represent triples in the database. Each contains subject, predicate, and object information.

Index vertices, illustrated in levels 1 to 3 in Figure 2.7, represent triple patterns and are responsible for routing partially matched copies of queries (referred to as query particles) towards triple vertices that match the pattern of the index vertex. Index vertices also contain subject, predicate, and object information, but one or several of them are wildcards. For example, in Fig. 2.7 the index vertex [*inspired *] (in the middle of the gure on level 2) routes to the index vertex [*inspired Dylan], which in turn routes to the triple vertex [Elvis inspired Dylan].

Query vertices, depicted in the example in Figure 2.7, are query dependent. For each query that is being executed, a query vertex is added to the graph. The query vertex emits the rst query particle that traverses the index graph. All query particles successfully matched or not eventually

get routed back to their respective query vertex. Once all query particles have succeeded or failed the query vertex reports the results and removes itself from the graph.

The index graph is built by adding a triple vertex for each RDF triple that is added to TripleRush. These vertices are added to Signal/Collect, which turns them into parallel processing units. Upon initialization, a triple vertex will add its three parent index vertices (on level 3) to the graph and add an edge from these index vertices to itself.

When an index vertex is initialized, it adds its parent index vertex, as well as an edge from this parent index vertex to itself. Note that the parent index vertex always has one more wildcard than its child. The construction process continues recursively until the parent vertex has already been added or the index vertex has no parent. In order to ensure that there is exactly one path from an index vertex to all triple vertices below it, an index vertex adds an edge from at most one parent index vertex.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

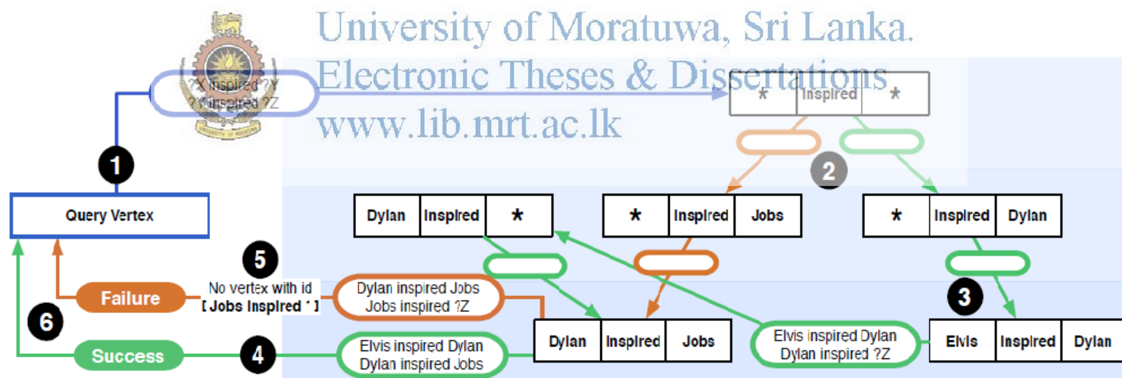


Figure 2.8 Query execution on the relevant part of the index that was created for the triples [Elvis inspired Dylan] and [Dylan inspired Jobs]. [7]

TripleRush is designed for the efficient parallel routing of messages to triples that correspond to a given triple pattern. All vertices that form the index structure are active parallel processing elements that only interact via message passing.

A query is defined by a list of SPARQL triple patterns. Each query execution starts by adding a query vertex to the graph. Upon initialization, a query vertex emits a single query particle. A query particle consists of the list of unmatched triple patterns, the ID of its query vertex, a list of

variable bindings, a number of tickets, and a flag that indicates if the query execution was able to explore all matching patterns in the index graph.

The emitted query particle is routed to the matching index vertex that matches the first unmatched triple pattern. The index vertex then sends the copies of the query particle to all of its immediate child vertices.

When the query particle encounters a triple pattern, it matches the next unmatched pattern to its triple vertex. If this yields success, then variable bindings are created and existing triple patterns are updated with the newly found bindings. The query particle gets routed back to the query vertex only if either all the triple patterns are matched or a match failed. In all the other instances the query particle is sent to either an index or triple vertex which conducts matching intern.

If an index vertex cannot be found for the given id, then the failed query is sent back to the query vertex by undeliverable message handler. Ultimately somehow the query particle is sent back to the query vertex, whether a match is found or not. Each query particle is associated with a number of tickets, to keep track of its query execution state. When the query execution is completed, the query vertex returns the variable bindings for all the successfully matched query particles and removes itself from the query graph.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

For an example which describes a full query execution, consider the graph and queries shown in the figure 2.8. The execution is started from the query vertex.

1. First the query vertex is added to the graph. Just after that, it emits a query particle which is depicted in blue. Then the query particle is routed to the matching index vertex with the id [* inspired *].
2. The query particle is split into two inside the index vertex. For the purpose of illustration one of them is colored in green whereas the other one is colored in orange. Both the particles are sent down to the immediate index vertices.
3. The very first pattern in the green particle is matched with the triple vertex with the id [Elvis inspired Dylan]. The triple vertex then sends the updated particle (unmatched

=[Dylan inspired ?Z]; with the bindings = f ?X=Elvis, ?Y=Dylan g) to the index vertex with ID [Dylan inspired *].

4. From the index vertex, the green particle is routed down to the triple vertex [Dylan inspired Jobs], which binds ?Z to Jobs. As there are no more unmatched triple patterns,the triple vertex routes the particle containing successful bindings for all variables back to its query vertex.
5. The first pattern of the orange particle gets matched in the triple vertex [Dylan inspired Jobs]. The triple vertex sends the updated particle (unmatched = [Jobs inspired ?Z]; bindings = f ?X=Dylan, ?Y=Jobs g) to the index vertex with ID [Jobs inspired *]. Since there is no index vertex with that id, the message cannot be delivered.
6. The query vertex receives both the successfully bound green and the failed orange particle. Query execution has finished, because all tickets that were sent out with the initial blue particle have been received again. The query vertex reports the successful bindings f ?X=Elvis, ?Y=Dylan, ?Z=Jobs g and then removes itself from the graph.

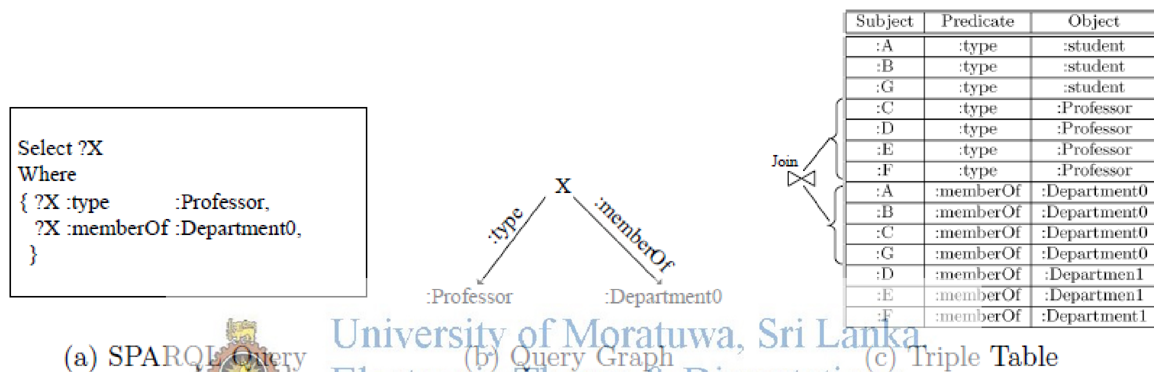
University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

2.3.4 Hybrid Approaches

There are few new hybrid approaches exist to store, index and query RDF data in triple stores efficiently. Graph structure of the RDF data is helped to reduce the join cost while manipulating them. The idea is to partition RDF data into overlapped groups and each group is given a unique identity. These RDF data then stored in a triple table with one more columns to store the group identity for each triple. The join operation is performed only within a given group, such a way that it reduces the join cost drastically. The SPARQL query execution cost can be minimized via this approach. Though RDF data representation is flexible in typical triple stores, it potentially results in serious performance issues since RDF queries involve intensive self-joins over the triple table. The new hybrid approach is going to address these issues in a typical RDF environment.

For an example instance, consider the Figure 2.9 depicted below. This shows that the traditional approach ends up with a fairly high join cost.

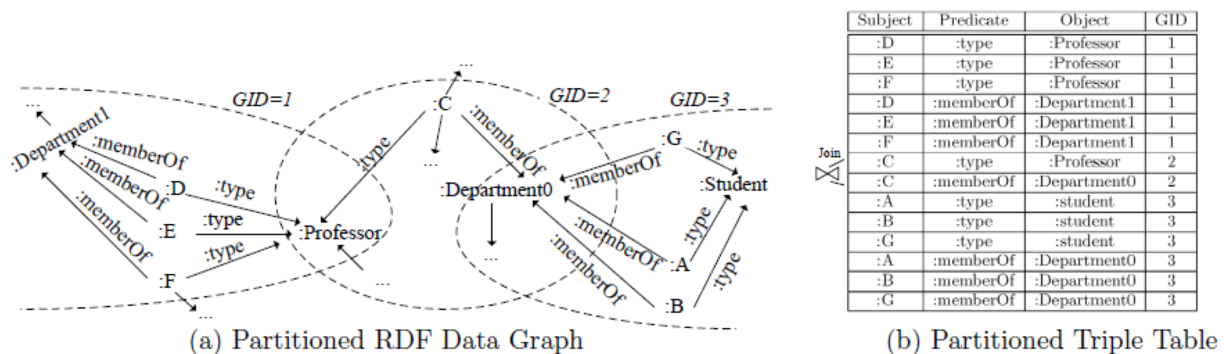
The new approach partitions the RDF graph into three groups as shown in dotted lines in Figure 2.10 below. An additional column is added to the triple table which refers to the unique group id value. Join operation is conducted only within one group, and in between the groups joins are restricted. This yields very less join cost while answering the SPARQL queries. A SPARQL query is decomposed into smaller sub-queries, and for each sub query is mapped into a RDF group using the signature tree index.



Subject	Predicate	Object
:A	:type	:student
:B	:type	:student
:G	:type	:student
:C	:type	:Professor
:D	:type	:Professor
:E	:type	:Professor
:F	:type	:Professor
:A	:memberOf	:Department0
:B	:memberOf	:Department0
:C	:memberOf	:Department0
:G	:memberOf	:Department0
:D	:memberOf	:Departmen1
:E	:memberOf	:Departmen1
:F	:memberOf	:Department1

Figure 2.9: A Motivated Example. [20]

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk



Subject	Predicate	Object	GID
:D	:type	:Professor	1
:E	:type	:Professor	1
:F	:type	:Professor	1
:D	:memberOf	:Department1	1
:E	:memberOf	:Department1	1
:F	:memberOf	:Department1	1
:C	:type	:Professor	2
:C	:memberOf	:Department0	2
:A	:type	:student	3
:B	:type	:student	3
:G	:type	:student	3
:A	:memberOf	:Department0	3
:B	:memberOf	:Department0	3
:G	:memberOf	:Department0	3

Figure 2.10: Newly Proposed Idea. [20]

2.4 Benchmarking RDF stores for performance evaluation

Triple stores are the backbone of increasingly many Data Web applications. It is evident that the performance of those stores is mission critical for individual projects as well as for data integration on the Web in general. Assessing the performance of current triple stores is, therefore, important to observe weaknesses and strengths of current implementations [27]. There are different ways of measuring the performance of triple stores offering a SPARQL query interface. It is consequently of central importance during the implementation of any Data Web application to have a clear picture of the weaknesses and strengths of current triple store implementations [14].

There are few SPARQL benchmarking [22] efforts such as LUBM [23], BSBM [24] and SP2 [25] [276 all of which resemble relational database benchmarks. The data structures behind these benchmarks are basically relational data structures. But there are heterogeneous RDF data stores being used today. Therefore some of those RDF stores does not follow orthodox relational approach and even cannot be represented in that way. DBPSB is a general SPARQL benchmark procedure, which is applied to the DBpedia knowledge base. Therefore DBpedia can be used to measure the performance of both relational and non-relational RDF stores.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

The main idea is to come up with a generic SPARQL benchmarking methodology. The very first step of benchmarking process is to generate the dataset. Generation of a suitable dataset is a crucial step in every benchmarking. The generated dataset should resemble the original data as much as possible. The data generation process should allow generating knowledge bases of various sizes ranging from smaller to larger. The data generation process should be easily repeatable with new versions of the considered dataset.

The second step is Query analysis and clustering which detects prototypical queries sent to the SPARQL endpoint. This approach uses string similarity measures. The query analysis and clustering is a four step approach. First queries that were executed frequently on the input data source are selected. For this SPARQL query log which contains all queries posed to the official SPARQL endpoint for a three-month period is used. In order to obtain a small number of distinctive queries for benchmarking triple stores, those queries are reduced. For an example queries with a low frequency are discarded.

Every SPARQL query contains substrings that segment it into different clauses. Although these strings are essential during the evaluation of the query, they are a major source of noise when computing query similarity. Therefore all SPARQL syntax keywords such as PREFIX, SELECT, FROM and WHERE are stripped from the query. This process is known as string stripping.

The third step is known as the similarity computation which computes the similarity of the stripped queries. The final step of this approach is known as clustering, which applies graph clustering to the query similarity graph computed above. The goal of this step is to discover very similar groups queries out of which prototypical queries can be generated.

Then we execute same set of prototypical SPARQL queries against the same RDF data set for different RDF store implementations which resides on the same hardware configuration, Operating system and compare the results. The main and simplest performance measurement is the SPARQL query execution time and which can be compared between different RDF store implementations. There are other different benchmarking measurements as well and we will discuss them later.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

3 METHODOLOGY

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

This section begins by explaining some existing approaches to build an RDF store. Then it elaborates on proposed solution and it's salient features. Next it explains about the solution architecture using a class diagram, which depicts the design. Solution implementation is covered in the next section with few sequence diagrams and that concludes this chapter.

3.1. Existing Solutions

The main challenges faced by most of the RDF data management systems are scalability and efficiency. Many different approaches exist to manage RDF data, each with its own advantages and disadvantages. Efficient and scalable management of RDF data is a fundamental challenge at the core of the Semantic Web. Many popular RDF storage solutions use relational databases to achieve this scalability and efficiency. Many state-of-the-art RDF systems store RDF data as a set of triples in relational tables, and therefore, they rely excessively on join operations for processing SPARQL queries.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

The simplest way to store RDF triples comprises a relation/table of three columns, one each for subjects, predicates and objects. However, this approach suffers from lack of scalability and poor query performance, as the single table becomes long and narrow when the number of RDF triples increases. The approach is not scalable since the table is usually located on a single machine. In addition, query performance is diminished since a query requires several self-joins with the same table.

RDF graph based approach is the new approach adopted recently that greatly improves the performance of SPARQL query processing. The idea is to use graph exploration instead of joins. Using this approach unnecessary intermediary results can be pruned down efficiently. This approach uses native graphs to model RDF data. One graph based triple store implementations is known as Trinity RDF. Trinity model and store RDF data as a directed graph. This opens up a new arena in the field of RDF data management.

3.2 Proposed Solution

There are more triple stores built for relational databases. Most of these triple stores suffer from performance and scalability issues which are inherent to relational model due to costly joins and large intermediate results. The main idea of this research is to address the aforementioned issues while building a scalable, in memory graph based RDF store for Apache Cassandra.

Figure 3.2.1 depicts all the main use cases pertaining to RDF store. For a given set of triples stored in a data source, an in memory Graph based RDF store needs to be implemented. For the very first time we need to read all the data stored in the database table, in order to build an in memory RDF graph. This incurs some substantial I/O cost but occurs only once. After the graph based triple store is built in memory any number of SPARQL queries can be executed against it. This is where the performance boost comes due to lower I/O cost.

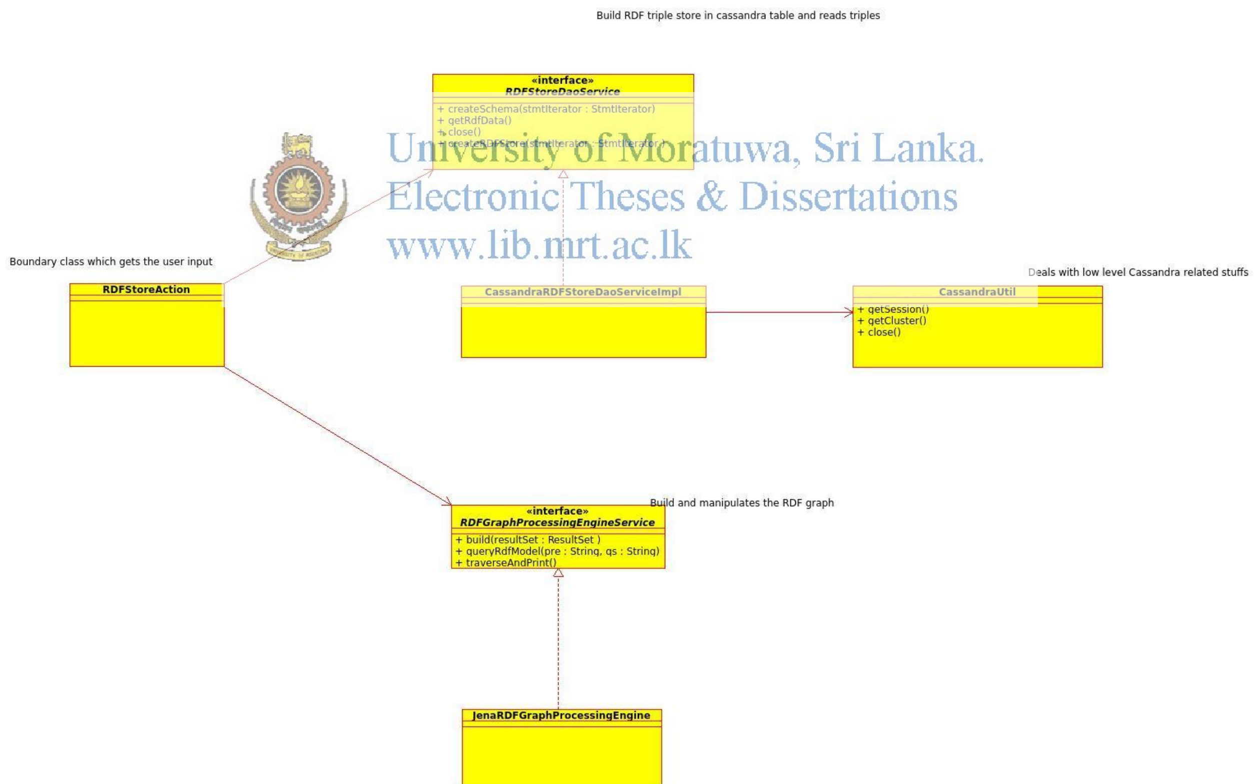
University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 3.2.1: Usecase diagram for RDF Graph building

The intention here is to perform in memory RDF graph explorations instead of joins which will improve the performance and reduce the I/O cost ultimately.

3.3 Solution Architecture

The solution was designed using the Object Oriented Programming methodologies and is stipulated in the figure 3.3.1. Each interface and/or implementation classes are assigned with some cohesive and discrete set of responsibilities that it performs. CassandraUtil class is responsible for handling low level Apache Cassandra related stuffs such as Sessions and clusters etc. RDFStoreDAOService interface and it’s implementation is responsible for performing all the Create, Read, Update, Delete operations related to the triple table. This interface defines all the necessary methods to build an RDF store and query it. It also provides some utility methods to connect/disconnect to the database on which the Triple store is placed.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure: 3.3.1 High level class diagram of the Graph based RDF store.

The current data access layer is only implemented to support RDF data stored in Apache Cassandra. But any RDF data stored in any other data source such as RDBMS, file system can be plugged to this Graph processing engine very easily. All you need to do is implement the Data Access Service which implements necessary logic to manipulate the data stored in that specific data store and pass them in correct format to the RDF graph processing engine. That implies this solution is more flexible and extensible. Therefore if someone is already having an RDF store on hand they don’t need to import all the RDF data to Cassandra to make use of this implementation. What they just need to do is to write their own data access layer implementation which manipulates the data stored in the data store and pass the RDF data in correct form to the graph processing engine. Then the graph processing engine will build the graph in-memory and there after the story remains the same. Also the class CassandraRDFStoreDaoServiceImpl internally uses the CassandraUtil class to perform common database operations such as low-level cluster and session handling.

The RDFGraphProcessingEngineService contract deifnes an API which is used to manipulate an RDF graph. For an example builds an RDF graph given set of RDFTriple instances, traverse the RDF graph model and prints it, query the RDF graph model and retrieves the answers given a SPARQL query.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Finally RDFStoreAction class acts as the boundary class which accepts user input. The user input can be a SPARQL query with it’s prefix which needs to be executed against the RDF graph built. Then this returns the result of the SPARQL query back to the client.

3.4 Solution Implementation

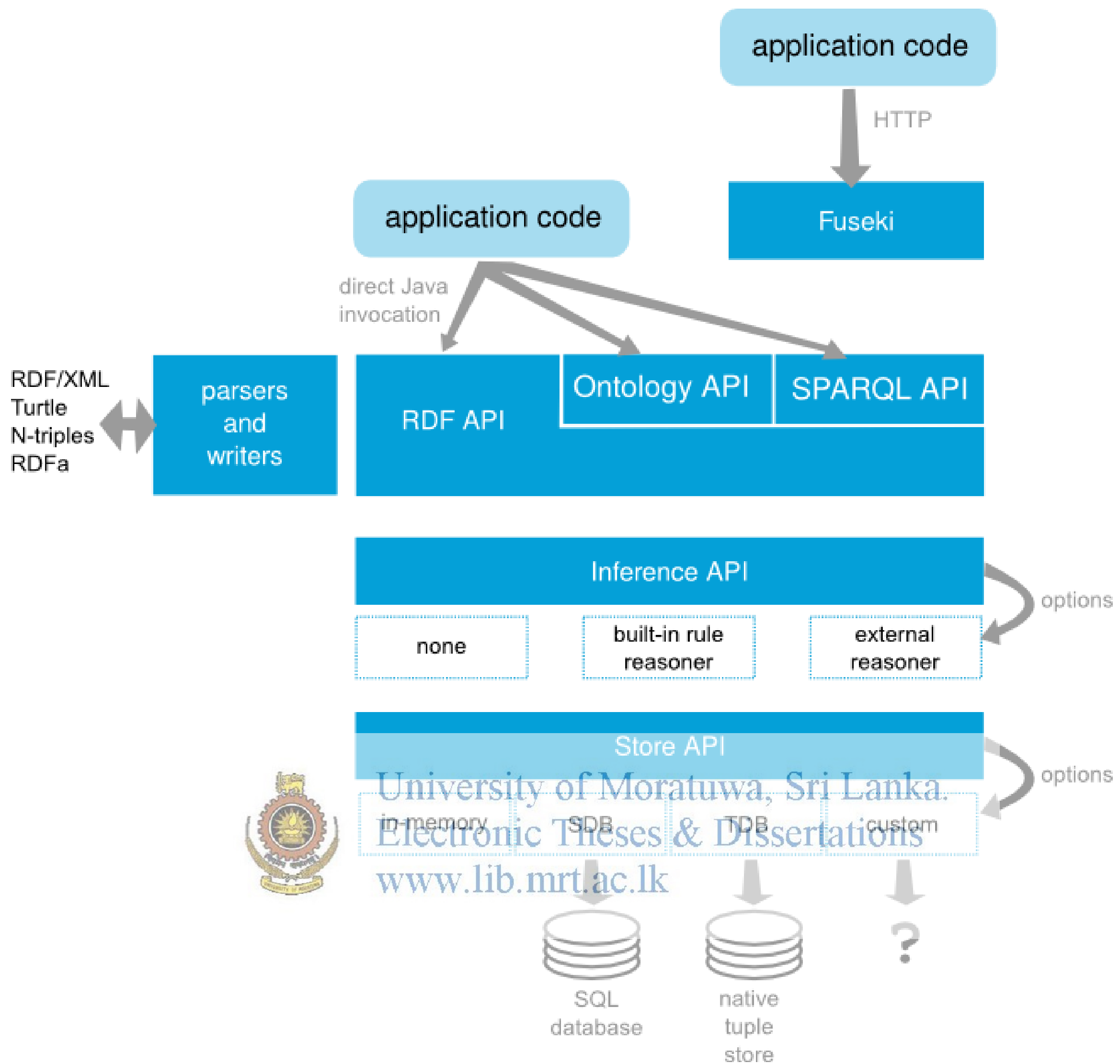
The solution is implemented using the Java Programming language. The aforementioned architecture and design was realized using Object Oriented infrastructure and platform provided by the Java programming language. Also Apache Jena framework which is shown in figure 3.4.1 was used as an RDF Graph processing framework.

Apache Jena is a free and open source Java framework for building Semantic Web and Linked Data applications. The framework is composed of different APIs interacting together to process RDF data. The interaction between the different APIs belongs to Jena is depicted below.

Though there are many different APIs are provided by Jena, main focus here is on the Jena RDF API [28]. The Jena RDF API provides a level of abstraction over RDF data manipulation. Thanks to Jena it is not necessary to implement a new algorithm to build an RDF graph given a set of RDF triples. Jena does that on behalf of us. Also Jena is capable in submitting and executing SPARQL queries against the RDF graph built so that answers can be retrieved. Apache Jena is an open source framework backed by Apache software foundation so that it can be trusted and also widely used in the industry today. Since there is no point of reinventing the wheel, capabilities provided by the Apache Jena framework is used to get the work done.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

The solution implementation can be explained using three main scenarios each of which can be represented using a sequence diagram.

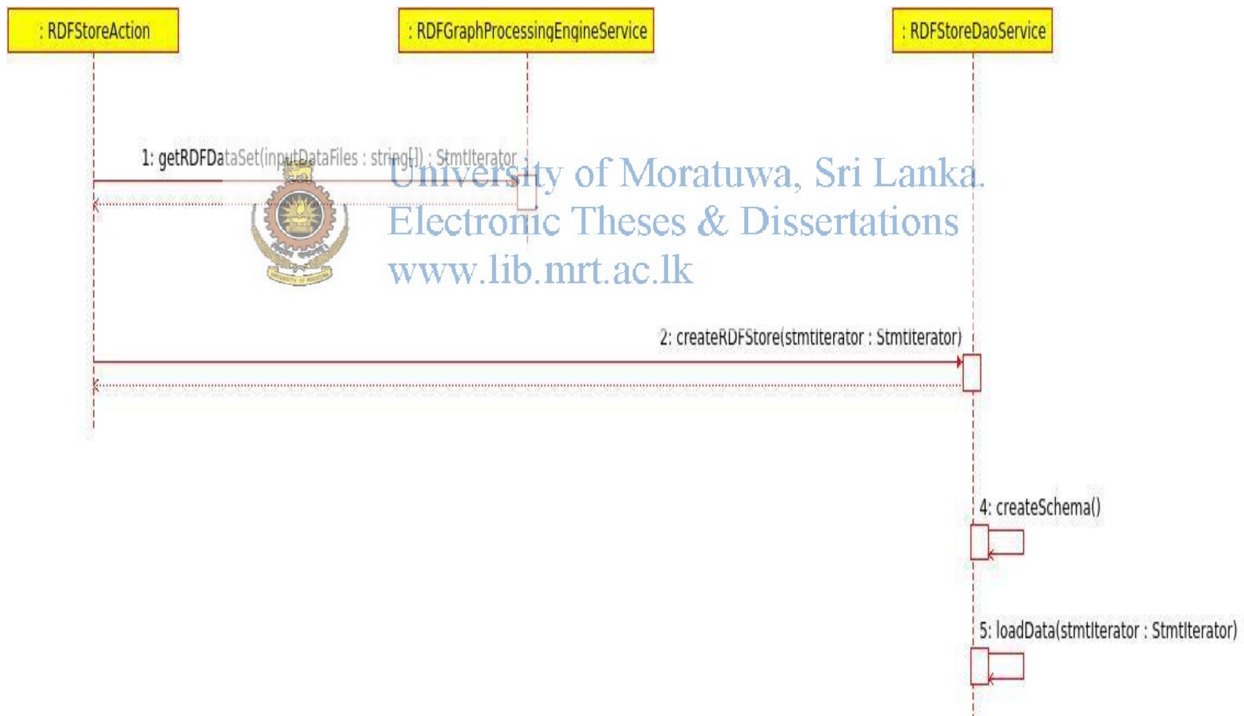


University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 3.4.1: Jena Framework Architecture [29]

3.4.1 Populating data into Cassandra Cluster

The sequence diagram for this scenario is given in figure 3.4.2. When the user clicks on the Create RDF Store hyperlink on the web interface, that event is detected by the Struts RDFStoreAction class and it's createRDFStore method is called. Then RDFStoreAction reads the config.properties file to get the NTRIPLE file(s) which will be used as the dataset to create the RDF store.Here DBPedia homepages and geocoordinate data set was used and together yields to 0.7 million triples. A user can specify any data set by merely giving the path to the NTRIPLE files as a comma separated list. Then using these RDF Statements it creates an RDF store in the Cassandra cluster via the RDFStoreDaoService. First it creates the schema and table structures and then inserts the data read from the Ntriples files into the cassandra cluster. Also note that this is just a onetime operation which is used to populate the triple store with data.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 3.4.2 Populating data into Cassandra Cluster

There are different formats which can be used to represent RDF data. Some of them are orthodox RDF/XML, Turtle format and N Triple format. N-Triples is a simplified version of Turtle that removes most of the shorthand. One line of an N-Triples equals one triple. N-Triples is more verbose than Turtle, but N-Triples is convenient when you need to handle millions of triples.

Many thinks that Turtle and N-Triples are superior replacements for the obsolete RDF/XML format. Turtle is the preferred format to write a few hundred triples by hand, and N-Triples is used to publish large RDF data sets like DBpedia.

3.4.2. Building the RDF Graph

Figure 3.4.3 shows how the RDF graph is built using the RDF triples. First the the user clicks on the Build RDF Graph hyperlink. This triggers the buildGraph method in RDFStoreAction class. This part of the job is done by the Struts 2 framework by using the configurations given in struts.xml file. It then reads the RDF Trples stored in the Cassandra cluster. This returns Apache Jena ResultSet which represents RDF triples stored inside the Cassandra cluster/triple table. RDFStoreAction uses RDFStoreDaoService to access the Cassandra cluster. Then it uses this Apache Jena ResultSet to build the RDF graph. Manipulation of the RDF graph is the responsibility of the RDFGraphProcessingEngineService. Therefore RDFStoreAction uses the RDFGraphProcessingEngineService to build the RDF graph given the set of RDFTriples. The graph needs to be built only once and can be reused for many SPARQL queries afterward.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

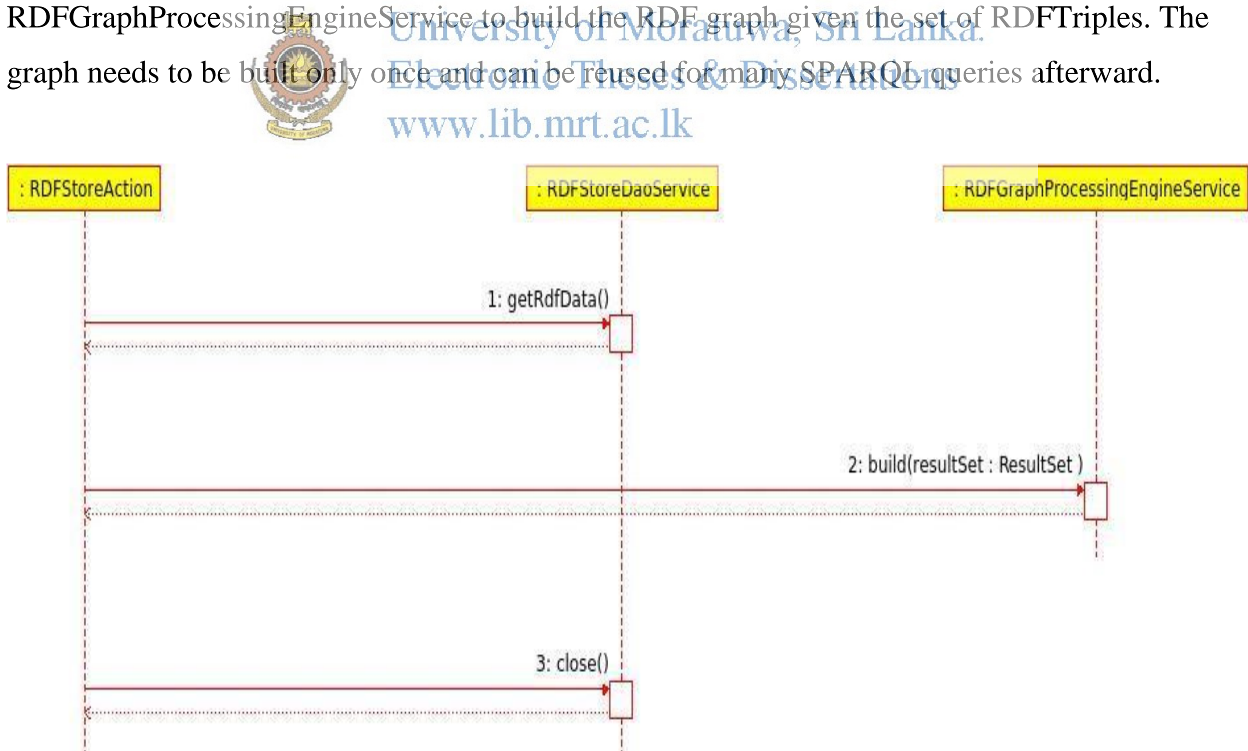
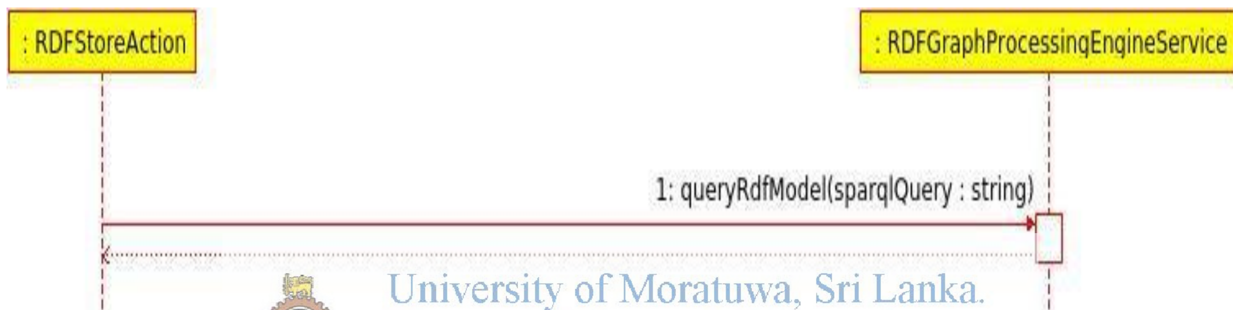


Figure: 3.4.3: Building the RDF Graph

3.4.3. Querying the RDF Graph

The next step is to query the RDF graph built and get the answers to those queries. The figure 3.4.4 depicts this scenario. Mainly DBPedia homepages and geo-coordinates data set is used with few sample queries which will be discussed in a later section in detail. The user needs to enter and submit the query using the web interface of the RDF store. Then it triggers the executeQuery method in RDFStoreAction according to the Struts configuration specified. Then the action class passes that SPARQL query to RDFGraphProcessingEngineService class by invoking the queryRdfModel method. This method is responsible for querying the RDF model using the given SPARQL query and retrieves the answers.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 3.4.4: Querying the RDF Graph

3.4.4: Dropping the RDF Store

This use case is explained in figure 3.4.5 and which begins whenever the user clicks on Drop RDF Store hyperlink. It then triggers the dropRDFStore method in the RDFStoreAction class. That method then will invoke the dropRDFStore method of the RDFStoreDaoService class. This is responsible for closing the database connection and executing the following statement which intern drops the keyspace.

CassandraUtil.getSession().execute("DROP KEYSPACE IF EXISTS rdfstore;");

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 3.4.5: Drop the RDF Store

3.4.5: Techniques used to render RDF/XML results on the webpage

After executing a query against the RDF Graph model, the output which yields the answer to the SPARQL query looks something like below. This is in the form of RDF/XML which is not that human friendly as shown in figure 3.4.6.

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
 <head>
  <variable name="s"/>
  <variable name="homepage"/>
 </head>
 <results>
  <result>
   <binding name="s">
    <uri>http://dbpedia.org/resource/Foreign_Office_%28Germany%29</uri>
   </binding>
   <binding name="homepage">
    <uri>http://www.auswaertiges-amt.de/www/en/index_html</uri>
   </binding>
  </result>
  <result>
   <binding name="s">
    <uri>http://dbpedia.org/resource/Sony_Center</uri>
   </binding>
   <binding name="homepage">
    <uri>http://www.sonycenter.de/aw/~a/Home/?lng=en</uri>
   </binding>
  </result>
…
 </results>
</sparql>
```

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 3.4.6: SPARQL query result in RDF/XML form

Therefore this is transformed into more readable human friendly form using the XSLT shown in figure 3.4.7.

```
<xsl:stylesheet version="1.0" xmlns="http://www.w3.org/1999/xhtml"
        xmlns:sp="http://www.w3.org/2005/sparql-results#" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="sp:sparql">
                <html>
                        <head>
                                <title>XSLT demo</title>
                        </head>
                        <body>
                                <table border="1">
                                        <tr bgcolor="#9acd32">
                                                <th>
                                                        <xsl:value-of select="sp:head/sp:variable[1]/@name" />
                                                </th>
                                                <th>
                                                        <xsl:value-of select="sp:head/sp:variable[2]/@name" />
                                                </th>
                                        </tr>
                                        <xsl:apply-templates />
                                </table>
                        </body>
                </html>
        </xsl:template>
        <xsl:template match="sp:result">
                <tr>
                        <td>
                                <xsl:value-of select="sp:binding[@name][1]" />
                        </td>
                        <td>
                                <xsl:value-of select="sp:binding[@name][2]" />
                        </td>
                </tr>
        </xsl:template>
```

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 3.4.7: XSLT used for transformation

This depicts the result in a more readable tabular format on the web interface of the RDF store. Some XPath expressions are used to access the values of the RDF/XML results obtained. The Welcome.jsp file fetches the results of the SPARQL query execution from the RDFStoreAction via the support provided by the Struts framework. Then it transforms this results using the xslt file provided. The sample coding of this jsp file which does all these magic with the help of Struts tag library is given in Figure 3.4.8.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"
>
<%@ taglib prefix="s" uri="/struts-tags"%>
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x"%>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Welcome Page - Struts 2 - Login Application</title>
</head>
<body>
        <a href="/RDFStoreWebClient/jsp/RDFQuery.jsp">Home</a>
        <h2>Listing Results of the SPARQL Query.</h2>
        <s:set name="xmltext" value="result" />
        <c:import url="/xslt/sparqlResultFormater.xsl" var="xslt" />
        <x:transform xml="${xmltext}" xslt="${xslt}" />

</body>
</html>
```

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 3.4.8: Sample jsp/struts code used to render results

3.4.6: Solution Extensibility and Flexibility

The current data access layer is only implemented to support RDF data stored in Apache Cassandra. But any RDF data stored in any other data source such as RDBMS, File System, etc can be plugged to this Graph processing engine very easily. All you need to do is implement the Data Access Service which implements RDFStoreDaoService contract with necessary logic to manipulate the data stored in that specific data store and pass them in correct format to the RDF graph processing engine. That implies this solution is more flexible and extensible. Therefore if someone is already having an RDF store on the hand they don't need to import all the RDF data to Cassandra to make use of this implementation. What they just need to do is to write their own data access layer implementation which manipulates the data stored in the data source and pass the RDF data in correct form to the graph processing engine. Then the graph processing engine will build the graph in-memory and there after the story remains the same.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

4 USE CASE SCENARIOS

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

The main use case scenario for an RDF store is to execute SPARQL queries against it and fetch the results. But the first step here is to build the RDF graph in memory. This section mainly describes how the new implementation is used by an end user. It mainly focuses on end users perspective of the system.

4.1. Build the RDF graph

An end user needs to build the RDF Store web client [30] application and deploy it into any web container that the user prefers. This section assumes that the user deploys this web client to the Apache Tomcat servlet container. The http://localhost:8080/RDFStoreWebClient/ takes the user to the homepage of the web application. There the user first needs to click on Build RDF Graph hyperlink which is responsible for building the RDF graph in memory. This is a onetime operation and the graph remains until the next server restart assuming that is a fairly large period of time. In Between the server restart any number of queries can be executed against the built RDF graph. The user interface is shown in figure 4.1.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

1. Build RDF Graph
2. Drop RDF Store
3. Create RDF Store

execute

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 4.1: Web Client Building RDF Graph

4.2 Executing SPARQL query

The main objective of a user as far as an RDF store is concerned is to execute SPARQL queries against it to answer questions. So in this implementation the user needs to access the homepage as stated above the hit the query in the text area provided. Then the execute button will submit the query against the RDF store. The user interface is shown in figure 4.2.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 4.2: Execute Query against the RDF graph

4.3 Rendering Results

The successful execution of a query will automatically take the user towards the results rendering page. That displays all the answers to the query submitted. That page is shown in figure 4.3. If the user needs to go back to the home page, she just need to click on the relevant link and execute the next query against the previously build RDF graph.

Home

Listing Results of the SPARQL Query.

s	homepage
http://dbpedia.org/resource/Foreign_Office_%28Germany%29	http://www.auswaertiges-amt.de/www/en/index_html
http://dbpedia.org/resource/Sony_Center	http://www.sonycenter.de/aw/~a/Home/?lng=en
http://dbpedia.org/resource/Topography_of_Terror	http://www.topographie.de/
http://dbpedia.org/resource/Berlin_Hauptbahnhof	http://www.hbf-berlin.de/site/berlin__hauptbahnhof/en/start.html
http://dbpedia.org/resource/Berghain	http://www.berghain.de
http://dbpedia.org/resource/Bayer_Schering_Pharma_AG	http://www.bayerscheringpharma.de
http://dbpedia.org/resource/Berliner_FC_Dynamo	http://www.berlinerfcdynamo.com/
http://dbpedia.org/resource/Embassy_of_the_United_States_in_Berlin	http://www.usembassy.de/
http://dbpedia.org/resource/German_Democratic_Republic	http://www.thelivesofothers.com
http://dbpedia.org/resource/Friedrichshain-Kreuzberg	http://www.friedrichshain-kreuzberg.de/
http://dbpedia.org/resource/Hotel_Adlon	http://www.hotel-adlon.de
http://dbpedia.org/resource/Tresor	http://www.tresorberlin.com/
http://dbpedia.org/resource/E-Werk	http://www.ewerk.net/
http://dbpedia.org/resource/Kunsthaus_Tacheles	http://www.tacheles.de/
http://dbpedia.org/resource/Berlin_University_of_the_Arts	http://www.udk-berlin.de/
http://dbpedia.org/resource/Berlin	http://berlin.de
http://dbpedia.org/resource/O2_World	http://www.o2-world.de
http://dbpedia.org/resource/Fernsehturm_Berlin	http://www.berlinerfernsehturm.de
http://dbpedia.org/resource/Federal_Ministry_of_the_Interior_%28Germany%29	http://www.bmi.bund.de
http://dbpedia.org/resource/Humboldt_University_of_Berlin	http://www.hu-berlin.de
http://dbpedia.org/resource/Bauhaus_Archive	http://www.bauhaus.de/english/index.htm
http://dbpedia.org/resource/Lichtenberg	http://www.berlin.de/ba-lichtenberg/index.html
http://dbpedia.org/resource/Berliner_Verkehrsbetriebe	http://www.bvg.de
http://dbpedia.org/resource/Akademie_der_K%C3%BCnste	http://www.adk.de/
http://dbpedia.org/resource/Zoologischer_Garten_Berlin	http://www.zoo-berlin.de/en.html

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 4.3: Rendering Results of the SPARQL query

5 EVALUATION AND RESULT

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

This section begins with different RDF store benchmarking methodologies. Then it focuses on dataset generation and different RDF data formats. Other RDF stores considered for this benchmarking process and SPARQL queries used for that is elaborated next. The benchmark configuration and benchmarking metrics concludes the section.

5.1 RDF Store Benchmarking

There are different methodologies which can be followed to evaluate and compare performance between triple stores. Some of them are briefly described below.

- Berlin SPARQL Benchmark (BSBM), provides for comparing the performance of RDF and Named Graph stores as well as RDF-mapped relational databases and other systems that expose SPARQL endpoints. Designed along an e-commerce use case. SPARQL and SQL version available. [22]
- Lehigh University Benchmark (LUBM) is developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way. The benchmark is intended to evaluate the performance of those repositories with respect to extensional queries over a large data set that commits to a single realistic ontology.
- The SP^{2}Bench SPARQL Performance Benchmark, provides a scalable RDF data generator and a set of benchmark queries, designed to test typical SPARQL operator constellations and RDF data access patterns.
- DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

There are many other tools and mechanisms which are not stated here. DBPedia benchmarking is used to evaluate the performance between different RDF store implementations in this research. DBPedia uses wikipedia dataset and imposes real queries on real data.

5.1.1 Dataset Generation

DBPedia knowledge base consists of a large dataset, which is backed by wikipedia. Out of the large dataset homepages and geocoordinate data set will be considered in this RDF store benchmark. These two dataset together accounts for around 0.7 million triples together. The first step of the benchmarking process is to load these data into the RDF stores which are used to compare the performance. This Graph based RDF store implementation is compared with well-known 4Store [31] and BigData [32][33] RDF store implementations. The benchmark dataset consists of DBpedia's geocoordinates and homepages [34] datasets with minor corrections.

- **geocoordinates-fixed.nt** (447,517 triples; 64 MB) Based on DBpedia's geocoordinates.nt. Gives geo locations for different real world resources
- **homepages-fixed.nt** (200,036 triples; 24 MB) Based on DBpedia's homepages.nt. Gives home page for some real world resources.

There are different RDF formats which can be used to represent RDF data. Also an RDF graph can be serialized into one of these RDF formats. These formats are explained briefly in the coming section.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

5.1.1.1 RDF/XML

RDF is most commonly expressed in an XML format: *RDF/XML*. The RDF/XML for what we know about the Manchester authority looks like figure 5.1.1.1.

In RDF/XML, the triples for each resource, are contained within<rdf:Description> nodes, with a sub-node for each property and its value. This format has the advantage that most programming languages have support for XML, and XML namespaces can be used to avoid having to use full URIs everywhere, which keeps the size down.

In RDF/XML, the triples for each resource, are contained within<rdf:Description> nodes, with a sub-node for each property and its value.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:j.0="http://data.ordnancesurvey.co.uk/ontology/admingeo/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:j.1="http://xmlns.com/foaf/0.1/"
    xmlns:j.2="http://opendatacommunities.org/def/local-government/"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:j.3="http://statistics.data.gov.uk/def/administrative-geography/"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >
  <rdf:Description rdf:about="http://opendatacommunities.org/id/district-council/b
    <j.2:governs rdf:resource="http://data.ordnancesurvey.co.uk/id/700000000001569
    <j.0:hasCensusCode>42UB</j.0:hasCensusCode>
    <j.2:billingAuthorityCode>E3531</j.2:billingAuthorityCode>
    <rdfs:label>Babergh</rdfs:label>
    <j.1:page rdf:resource="http://www.babergh.gov.uk"/>
    <rdf:type rdf:resource="http://opendatacommunities.org/def/local-government/Lo
    <rdf:type rdf:resource="http://opendatacommunities.org/def/local-government/Di
    <j.0:gssCode>E07000200</j.0:gssCode>
    <owl:sameAs rdf:resource="http://statistics.data.gov.uk/id/local-authority/42U
    <j.3:region rdf:resource="http://statistics.data.gov.uk/id/government-office-r
    <rdf:type rdf:resource="http://opendatacommunities.org/def/local-government/Ci
  </rdf:Description>
  <rdf:Description rdf:about="http://opendatacommunities.org/id/london-borough-cou
    <owl:sameAs rdf:resource="http://statistics.data.gov.uk/id/local-authority/00A
    <j.0:gssCode>E09000002</j.0:gssCode>
    <rdf:type rdf:resource="http://opendatacommunities.org/def/local-government/Ci
    <j.2:openlyLocalUrl rdf:resource="http://openlylocal.com/councils/19-London-Bo
    <rdf:type rdf:resource="http://opendatacommunities.org/def/local-government/Lo
    <j.2:governs rdf:resource="http://data.ordnancesurvey.co.uk/id/700000000001094
    <rdf:type rdf:resource="http://opendatacommunities.org/def/local-government/Lo
    <j.0:hasCensusCode>00AB</j.0:hasCensusCode>
    <j.3:region rdf:resource="http://statistics.data.gov.uk/id/government-office-r
    <j.1:page rdf:resource="http://www.lbbd.gov.uk/"/>
    <j.2:billingAuthorityCode>E5030</j.2:billingAuthorityCode>
    <rdfs:label>Barking and Dagenham</rdfs:label>
  </rdf:Description>
</rdf:RDF>
```

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure: 5.1.1.1 Multiple resources as RDF/XML [35]

5.1.1.2 Turtle

Turtle (Terse RDF Triple Language) is an RDF-specific subset of Tim Berners-Lee's*Notation3* language.

Multiple local authorities could be serialized in Turtle as in figure 5.1.1.2.

```
@prefix gov: <http://opendatacommunities.org/def/local-government/> .
@prefix admingeo: <http://data.ordnancesurvey.co.uk/ontology/admingeo/> .
@prefix statsgeo: <http://statistics.data.gov.uk/def/administrative-geography/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<http://opendatacommunities.org/id/district-council/babergh>
    a gov:LocalAuthority, gov:DistrictCouncil, gov:CivilAdministrativeAuthority ;
    rdfs:label "Babergh" ;
    admingeo:gssCode "E07000200" ;
    admingeo:hasCensusCode "42UB" ;
    admingeo:billingAuthorityCode "E3531" ;
    gov:governs <http://data.ordnancesurvey.co.uk/id/7000000000015692> ;
    statsgeo:region <http://statistics.data.gov.uk/id/government-office-region/G>
    owl:sameAs <http://statistics.data.gov.uk/id/local-authority/42UB> ;
    foaf:page <http://www.babergh.gov.uk> .

<http://opendatacommunities.org/id/london-borough-council/barking-and-dagenham>
    a gov:LocalAuthority, gov:LondonBoroughCouncil, gov:CivilAdministrativeAuthori
    rdfs:label "Barking and Dagenham" ;
    admingeo:gssCode "E09000002" ;
    admingeo:hasCensusCode "00AB" ;
    gov:billingAuthorityCode "E5030" ;
    gov:governs <http://data.ordnancesurvey.co.uk/id/7000000000010949> ;
    gov:openlyLocalUrl <http://openlylocal.com/councils/19-London-Borough-of-Barki
    statsgeo:region <http://statistics.data.gov.uk/id/government-office-region/H>
    owl:sameAs <http://statistics.data.gov.uk/id/local-authority/00AB> ;
    foaf:page <http://www.lbbd.gov.uk/> .
```

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure: 5.1.1.2: Multiple resources as Turtle

The URI for the each resource is followed by the predicates and objects of the triples about it (essentially, as key-value pairs). Each pair is separated by a semi-colon, and the information about a resource is closed off by a dot. Note that the whitespace is not important here, but for readability, the predicates and objects are often indented and appear on separate lines.

It's fairly easy to read for humans due to the lack of punctuation noise, it groups together the triples about each resource.

5.1.1.3 N-Triples

N-Triples are a simplified version of Turtle. Here's the RDF for the Manchester authority again, this time as N-triples is given in figure 5.1.1.3.

```
<http://opendatacommunities.org/id/district-council/allerdale> <http://www.w3.org/
<http://opendatacommunities.org/id/district-council/allerdale> <http://opendatacom
<http://opendatacommunities.org/id/district-council/allerdale> <http://www.w3.org/
<http://opendatacommunities.org/id/district-council/allerdale> <http://opendatacom
<http://opendatacommunities.org/id/district-council/allerdale> <http://opendatacom
<http://opendatacommunities.org/id/district-council/allerdale> <http://www.w3.org/
<http://opendatacommunities.org/id/district-council/allerdale> <http://www.w3.org/
<http://opendatacommunities.org/id/district-council/allerdale> <http://xmlns.com/f
<http://opendatacommunities.org/id/district-council/allerdale> <http://statistics.
<http://opendatacommunities.org/id/district-council/allerdale> <http://data.ordnan
<http://opendatacommunities.org/id/district-council/allerdale> <http://www.w3.org/
<http://opendatacommunities.org/id/district-council/allerdale> <http://data.ordnan
<http://opendatacommunities.org/id/district-council/barrow-in-furness> <http://ope
<http://opendatacommunities.org/id/district-council/barrow-in-furness> <http://www
<http://opendatacommunities.org/id/district-council/barrow-in-furness> <http://www
<http://opendatacommunities.org/id/district-council/barrow-in-furness> <http://ope
<http://opendatacommunities.org/id/district-council/barrow-in-furness> <http://www
<http://opendatacommunities.org/id/district-council/barrow-in-furness> <http://www
<http://opendatacommunities.org/id/district-council/barrow-in-furness> <http://xml
<http://opendatacommunities.org/id/district-council/barrow-in-furness> <http://dat
<http://opendatacommunities.org/id/district-council/barrow-in-furness> <http://dat
<http://opendatacommunities.org/id/district-council/barrow-in-furness> <http://www
<http://opendatacommunities.org/id/district-council/barrow-in-furness> <http://sta
```

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 5.1.1.3: Multiple Resources as N-Triples

With N-triples, each triple appears on its own line, separated by a dot.

N-triples' simplicity makes it easy for software to parse and generate, but it lacks some of the features of RDF/XML and Turtle (such as support for nested resources). Due to the repetition of the resource URIs, it's not as compact as Turtle, and the triples for each resource aren't necessarily grouped together which makes it harder to read by eye.

For the purpose of this benchmark N-Triples RDF format is used hereafter.

5.1.2 Tested RDF Stores

The following stores were selected,

- 4Store
- BigData
- Current Implementation

The 4Store [31] RDF store was selected since it is an efficient, scalable and stable RDF database which is widely used in the industry today. On the other hand BigData [33] is a graph based RDF store which was built on top of the Sesame framework and used for a fair comparison with this graph based implementation. The current Graph based RDF store implementation can be found in the github location [36].

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

5.1.3 Query Generation

Mainly four queries are used to evaluate the performance of different triple store implementations. The queries used are explained in the following section.

Query1:

- Finds the homepage of the Metropolitan museum of Art which is given in figure 5.1.3.1.

```
SELECT ?p ?o WHERE {
  <http://dbpedia.org/resource/Metropolitan_Museum_of_Art> ?p ?o
}

or

PREFIX p: <http://dbpedia.org/resource/>
SELECT ?p ?o WHERE { p:Metropolitan_Museum_of_Art ?p ?o }
```

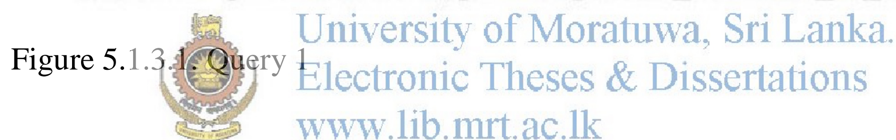

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 5.1.3.1. Query 1

Query 2:

- Finds the Homepage of Kevin_Bacon given in figure 5.1.3.2.

```
PREFIX p: <http://dbpedia.org/resource/>
    SELECT ?p ?o WHERE {p:Kevin_Bacon  ?p ?o}
```

Figure 5.1.3.2. Query 2

Query 3:

- Finds all the resources and their homepages which reside near the area of Berlin. This query is shown in figure 5.1.3.3.

```
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?s ?homepage WHERE {
    <http://dbpedia.org/resource/Berlin> geo:lat ?berlinLat .
    <http://dbpedia.org/resource/Berlin> geo:long ?berlinLong .
    ?s geo:lat ?lat .
    ?s geo:long ?long .
    ?s foaf:homepage ?homepage .
FILTER (
    ?lat    <=   ?berlinLat + 0.03190235436 &&
    ?long   >=   ?berlinLong - 0.08679199218 &&
    ?lat    >=   ?berlinLat - 0.03190235436 &&
    ?long   <=   ?berlinLong + 0.08679199218)
}
```

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 5.1.3.3. Query 3

Above is a fairly complex query with some range checks and joins.

Query 4:

- Finds all the resources and their homepages which reside near the area of New York. This query is shown in figure 5.1.3.4.

```
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX p: <http://dbpedia.org/property/>
SELECT ?s ?homepage WHERE {
    <http://dbpedia.org/resource/New_York_City> geo:lat ?nyLat .
    <http://dbpedia.org/resource/New_York_City> geo:long ?nyLong .
    ?s geo:lat ?lat .
    ?s geo:long ?long .
    ?s foaf:homepage ?homepage .
FILTER (
    ?lat     <=   ?nyLat + 0.3190235436 &&
    ?long    >=   ?nyLong - 0.8679199218 &&
    ?lat     >=   ?nyLat - 0.3190235436 &&
    ?long    <=   ?nyLong + 0.8679199218)
}
```

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 5.1.3.4. Query 4

Query 5:
This query is based on the DBPedia person dataset. It finds all the people who born in England. This query is given in figure 5.1.3.5.

```
PREFIX db: <http://dbpedia.org/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX onto: <http://dbpedia.org/ontology/>

SELECT ?person ?name
WHERE { ?person onto:birthPlace db:England;
foaf:name ?name .}
```

Figure 5.1.3.5. Query 5

This is just to show that our solution is extensible. So a user can point to any dataset, person dataset in this instance and build the RDF graph using that. Only thing she needs to do is stop the server, replace the NTRIPLE file location in config.properties file and restart it. Upon restart she needs to drop the existing RDF store, create a new one using the specified file and build the graph model. Then she can execute any relevant SPARQL query against that graph model. So this solution is extremely flexible and extensible.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

As few data has been prepared for actual use in the application, the queries are mostly of generic nature. They run against the DBpedia geo coordinates and homepages data set and assess performance with varying levels of joins and constraints.

5.1.4. Benchmark Configuration

The configuration used for the benchmarking process is given in the Table 2.

Processor	Intel Pentium CORE i7 vPro
Physical Memory	16 GB
Hard Disk	500 GB
Operating System	Ubuntu Linux 7.14 64-bit

Table 2: Benchmark Configuration

5.1.5. Benchmark Metrics

The main metrics used in DBPSB for performance measurement are:

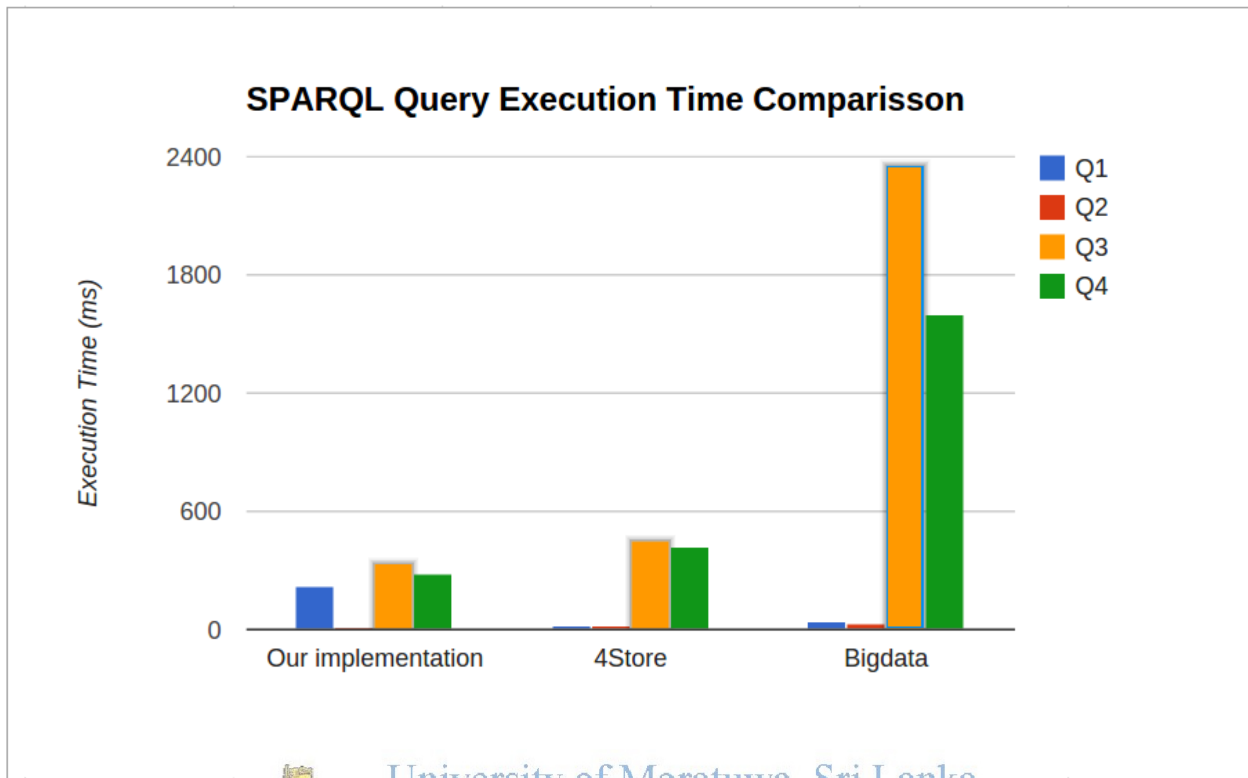
1. Query Mixes per Hour (QMpH), which denotes the number of query mixes posed to the test store in one hour.
2. Queries per Second (QpS), which is the number of queries (query variations of a specific query) the test store can answer in one second.
3. Query execution time in ms.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

The execution time taken by different RDF stores, to execute above four queries is given in table 3. The time is measured in milliseconds. We are comparing 4Store RDF store and Bigdata with our custom implementation. A histogram and a line graph which represents these results in pictorial form is given in figure 5.1.5.1.

	Q1	Q2	Q3	Q4
Our implementation	216ms	7ms	336ms	279ms
4Store	16ms	18ms	455ms	416ms
Bigdata	41ms	30ms	2sec, 355ms	1sec, 600ms

Table 3: Performance Benchmark results



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

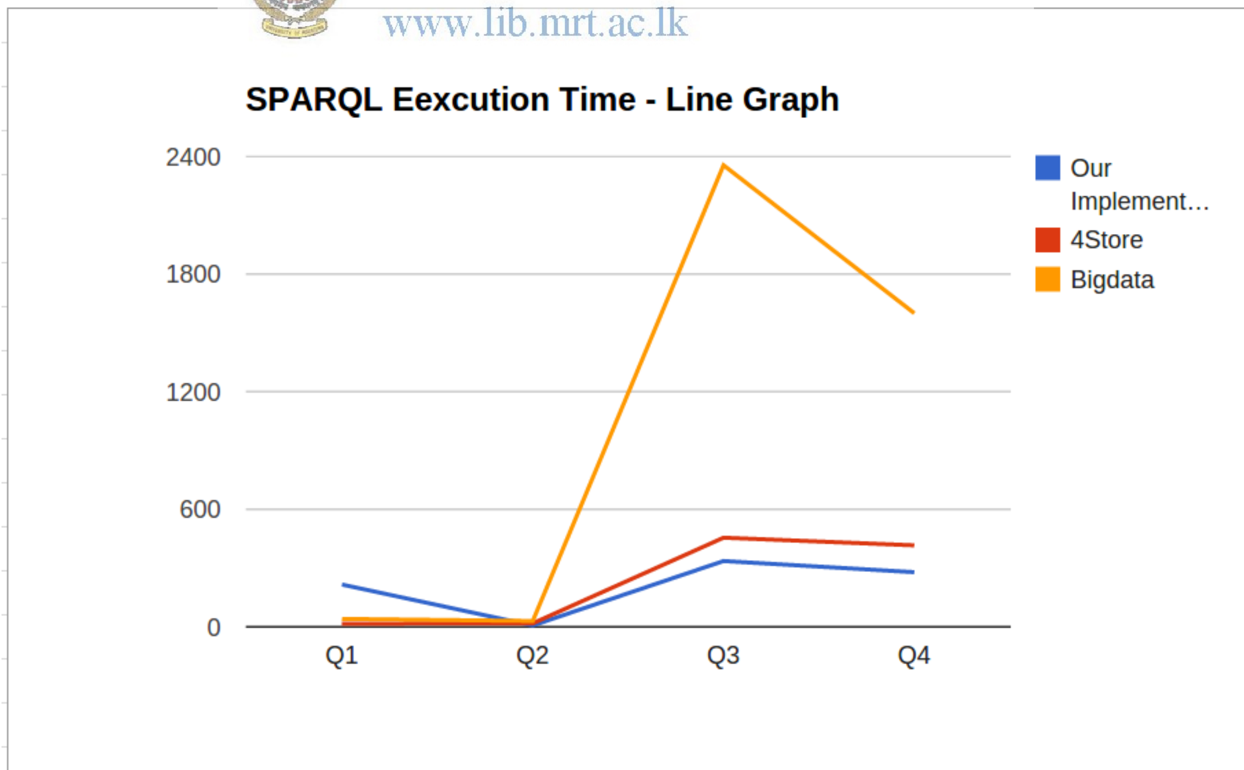


Figure 5.1.5.1: SPARQL Query Execution time

For a fair comparison, the query execution time was compared. Based on this results we may conclude that the Graph based approach yields more performance boosts when query becomes more and more complex compared to other relational approaches. Starting from Query 1 traversing towards Query 3 and Query 4, the complexity becomes higher. The first two queries are simple queries which does not contain any costly join operations whereas the last two queries are complex queries which involves costly join operations on range checks. Based on the above results we can conclude that the graph based implementation outperforms 4Store RDF implementation and Bigdata in most of the cases especially when the SPARQL query becomes more and more complex. In graph based implementation the first query takes more time, since it builds index structure of the graph for the first time. There after it caches them and continues to work as expected.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

6 FUTURE WORK

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

This section briefly discusses about major limitations of this approach and how those limitations can be alleviated by adding some enhancements to this Triple store implementation.

The main limitation of this approach is the scalability. RDF store may contain large amount of RDF triples, possibly several millions or even trillions of triples. If all those triples are loaded in to a graph model in memory at once, there is a high tendency that the system will encounter with an OutOfMemoryError while the RDF graph model is being built. This implementation is constrained by the size of the RDF dataset.

As a solution a distributed implementation of this approach can be suggested. The figure 6.1 illustrates the high level architecture for such a system. The existing system is written in a way such that it can be extended easily to cater these new needs.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

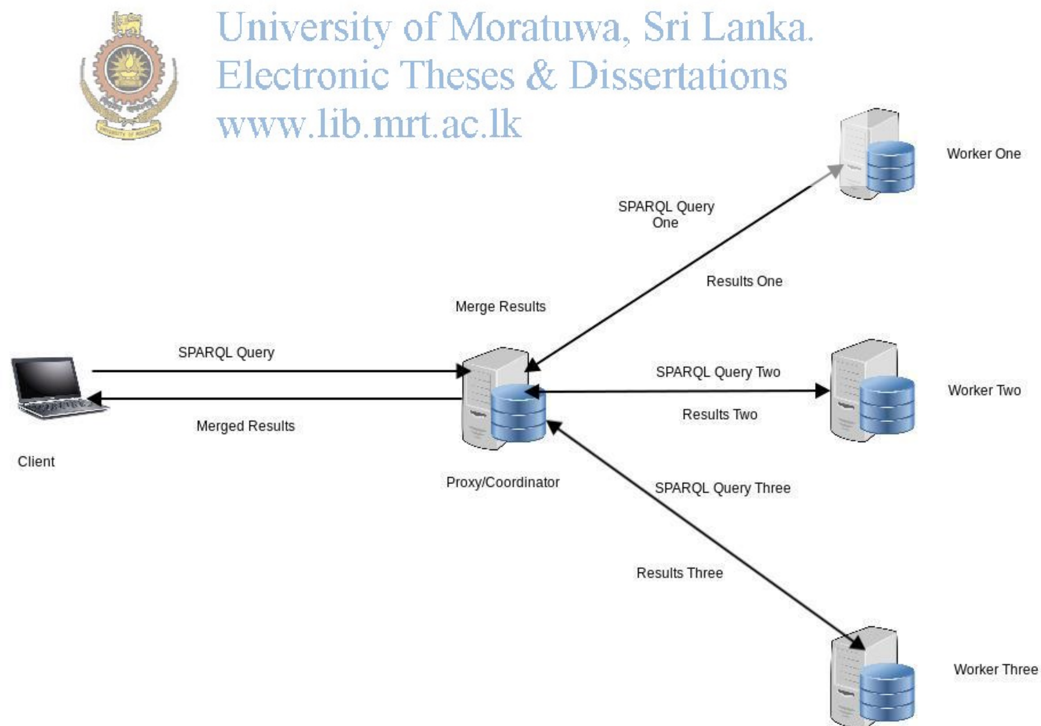


Figure 6.1: Distributed Implementation of the RDF store

The query submitted by the client may contain small segments which would be answered by querying different datasets. For an example suppose a given complex join query can be answered fully using homepages, geo coordinates and people dataset together. The homepages and geo coordinates dataset can be loaded to two different nodes respectively as a graph model. But if the data set is much larger to fit into the memory of a one single worker node, for an example people data set, then that dataset is distributed across multiple worker nodes while many graph models residing on multiple worker nodes representing that dataset. This is a complex situation which needs to be handled carefully.

The client submits a complex query to the coordinator node which then splits that query to few simple queries and submitted to the worker nodes. Worker nodes are responsible for executing the simple queries against the graph model resides in it and results are sent back to the coordinator. Then the coordinator merges the results and pass it back to the client. A distributed coordination support needs to be implemented here. Since Apache Cassandra is distributed, multi-tenant, this will leverage the capabilities of our new implementation. With this enhancement the RDF store will be more scalable, efficient and useable as far as end users are concerned.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

7 CONCLUSION

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

This thesis begins by investigating different approaches used to model and retrieve RDF data efficiently. It compares and contrasts those approaches and outlines their salient features.

Also this thesis discussed a new approach to manage RDF data efficiently. It used graph based approach to model RDF data which yields high performance and efficiency. It explained the design and implementation details of the proposed system. The system use cases are also described to assist end users of the system. It also proved that this new approach outperforms the existing solutions to manage RDF data efficiently especially when the submitted SPARQL queries are more and more complex. The thesis used DBPedia homepages, geo coordinates and people dataset with some sample queries ranging from simple to complex for the performance evaluation. The current implementation is compared with 4Store and Bigdata RDF store implementations and results were explained with the rationale behind it.

Finally it identifies the limitations of this implementation. The major limitation is the size of the RDF dataset which constrains this implementation. If the dataset is much larger the in memory graph consumes more memory space and leads to exhaust the memory in the system. Then it comes up with some enhancements to this implementation which will alleviate those constraints and limitations.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

It suggests a distributed graph based approach to build this RDF store in a worker manager distributed setup. Since Apache Cassandra is distributed and multi-tenant, this approach will leverage the capabilities of the existing implementation. The manager splits a complex query into few simple queries and submitted to the workers. Workers are executing the simple queries against their own graph model and answers are sent back to the manager node. Finally manager merges the results and returns back to the client. The future work and improvements section concludes the thesis.

References

[1] F. Manola and E. Miller. (2004, Feb) RDF primer. W3C recommendation. [Online]. Available: http://www.w3.org/TR/rdf-primer/

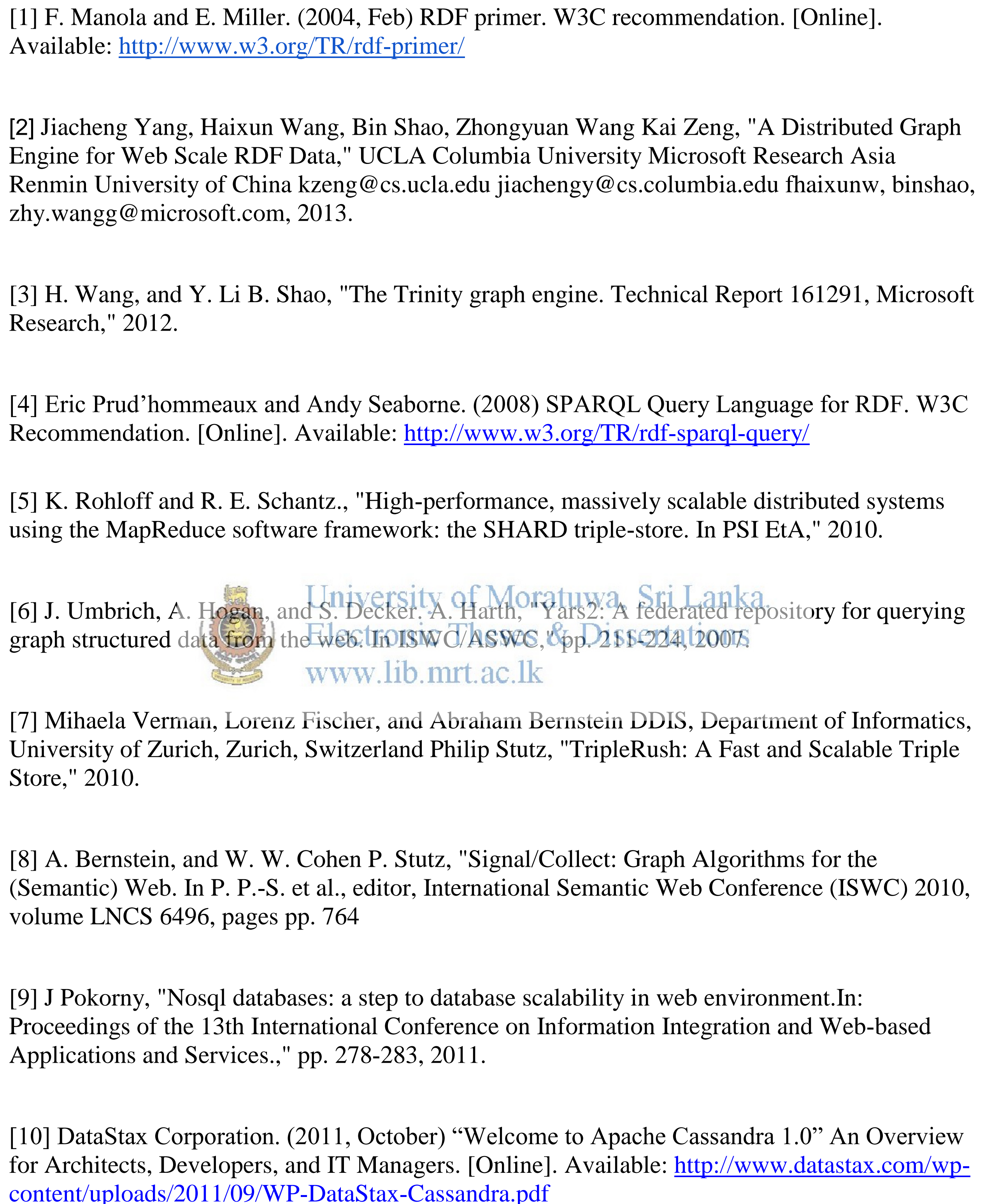
[2] Jiacheng Yang, Haixun Wang, Bin Shao, Zhongyuan Wang Kai Zeng, "A Distributed Graph Engine for Web Scale RDF Data," UCLA Columbia University Microsoft Research Asia Renmin University of China kzeng@cs.ucla.edu jiachengy@cs.columbia.edu fhaixunw, binshao, zhy.wangg@microsoft.com, 2013.

[3] H. Wang, and Y. Li B. Shao, "The Trinity graph engine. Technical Report 161291, Microsoft Research," 2012.

[4] Eric Prud'hommeaux and Andy Seaborne. (2008) SPARQL Query Language for RDF. W3C Recommendation. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[5] K. Rohloff and R. E. Schantz., "High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In PSI EtA," 2010.

[6] J. Umbrich, A. Hogan, and S. Decker. A. Harth, "Yars2: A federated repository for querying graph structured data from the web. In ISWC/ASWC," pp. 211-224, 2007.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

[7] Mihaela Verman, Lorenz Fischer, and Abraham Bernstein DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland Philip Stutz, "TripleRush: A Fast and Scalable Triple Store," 2010.

[8] A. Bernstein, and W. W. Cohen P. Stutz, "Signal/Collect: Graph Algorithms for the (Semantic) Web. In P. P.-S. et al., editor, International Semantic Web Conference (ISWC) 2010, volume LNCS 6496, pages pp. 764

[9] J Pokorny, "Nosql databases: a step to database scalability in web environment.In: Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services.," pp. 278-283, 2011.

[10] DataStax Corporation. (2011, October) "Welcome to Apache Cassandra 1.0" An Overview for Architects, Developers, and IT Managers. [Online]. Available: http://www.datastax.com/wp-content/uploads/2011/09/WP-DataStax-Cassandra.pdf

[11] Datastax. (2011, Nov 10.) An overview of Apache Cassandra. [Online]. Available: http://www.slideshare.net/DataStax/an-overview-of-apache-cassandra?v=qf2&b=&from_search=1

[12] David Rapp Roshan Punnoose Adina Crainiceanu, "Rya: "A Scalable RDF Triple Store for the Clouds" Proteus Technologies US Naval Academy Laboratory for Telecommunication Sciences," 2012.

[13] Chris Bizer, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann Jens Lehmann, "DBpedia - a crystallization point for the web of data. Journal of Web Semantics," vol. 7(3), pp. 154–165, 2009.

[14] Sören Auer, and Axel-Cyrille Ngonga Ngomo Mohamed Morsey Jens Lehmann, "DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data," Department of Computer Science, University of Leipzig Johannisgasse 26, 04103 Leipzig, Germany, 2013.

[15] Justin J. Levandoski F. Mokbel, "RDF Data-Centric Storage," Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

[16] DataStaxMedia. (2011, October 5) DataStax Cassandra Tutorials - Apache Cassandra Overview. [Online]. Available: http://www.youtube.com/watch?v=5qEoEAfAer8

[17] Murat Kantarcioglu, Bhavani Thuraisingham, and Paolo Castagna Vaibhav Khadilkar, "Jena-HBase: A Distributed, Scalable and Efficient RDF Triple Store," 2012.

[18] T. Neumann and G. Weikum., "RDF-3X: a RISC-style," vol. 1(1), 2008.

[19] M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski G. Malewicz, "Pregel: a system for large-scale graph processing. In SIGMOD Conference," pp. 135-146, 2010.

[20] Aoying Zhou , Weining Qian , Li Ma, Yue Pan Ying Yan † Chen Wang, "Efficiently Querying RDF Data in Triple Stores,Department of Computer Science and Engineering, Fudan University," April 2008.

[21] Sören Auer, and Axel-Cyrille ens Lehmann, "Ngonga Ngomo Department of Computer Science, University of Leipzig Mohamed Morsey, "DBpedia SPARQL Benchmark," Performance Assessment with Real Queries on Real Data".

[22] W3C. (2014, November 14). RDF Store Benchmarking. [Online]. Available: http://www.w3.org/wiki/RdfStoreBenchmarking

[23] Yuanbo Guo, , and Je Heflin. LUBM Zhengxiang Pan, "A benchmark for OWL knowledge base systems. In Journal of Web Semantics," vol. 3, pp. 158–182, 2005.

[24] Christian Bizer and Andreas Schultz, "The Berlin SPARQL Benchmark. Int. J. Semantic Web Inf. Sys," pp. 1–24, 2009.

[25] Thomas Hornung, Georg Lausen, and Christoph Pinkel Michael Schmidt, "SP2Bench: A SPARQL performance benchmark. In ICDE," pp. 222–233, 2009.

[26] Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP2Bench Michael Schmidt, "A SPARQL performance benchmark. In ICDE, pages 222–233. IEEE," 2009.
University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk
[27] Mohamed Morsey. (2014, October 17). DBpedia SPARQL Benchmark: a pure RDF benchmark based on actually posed queries [Online]. Available: http://aksw.org/Projects/DBPSB.html

[28] Apache Jena. An Introduction to RDF and the Jena RDF API [Online]. Available: http://jena.apache.org/tutorials/rdf_api.html

[29] Paolo Castagna. Getting started with Apache Jena [Online]. Available: http://jena.apache.org/getting_started/index.html

[30] Ranwala R.S. (2014, November). RDF Store web Client. [Online]. Available: https://github.com/ravindraranwala/RDFStoreWebClient

[31] EdgewallSoftware. (2012, March 20.) 4Store [Online]. Available: http://4store.org/trac/wiki/SparqlServer

[32] Beebs, Mrpersonick, Thompsonbry. (2007, March 21.) **bigdata** Fast, scalable, robust graph database platform [Online]. Available: http://sourceforge.net/projects/bigdata/

[33] bigdata Getting Started [Online]. Available:
http://wiki.bigdata.com/wiki/index.php/GettingStarted

[34] DBpedia. (2008, Jan 10.) RDF Store Benchmarks with DBpedia [Online]. Available:
http://wifo5-03.informatik.uni-mannheim.de/benchmarks-200801/

[35] Ric. (2012, January 30.) Understanding RDF serialization formats [Online]. Available:
http://blog.swirrl.com/articles/rdf-serialisation-formats/

[36] Ranwala R.S. (2014, November). RDF Store - GraphBased. [Online]. Available:
https://github.com/ravindraranwala/RDFStoreForCassandra

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk