DISTRIBUTED RDF QUERY PROCESSING AND REASONING FOR BIG DATA / LINKED DATA

A THESIS IN
Computer Science

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

by
ANUDEEP PERASANI

B.E., Osmania University, 2012

Kansas City, Missouri
2014

DISTRIBUTED RDF QUERY PROCESSING AND REASONING FOR BIG DATA / LINKED DATA

Anudeep Perasani, Candidate for the Master of Science Degree

University of Missouri-Kansas City, 2014

ABSTRACT

The Linked Data Movement is aimed at converting unstructured and semi-structured data on the documents to semantically connected documents called the "web of data." This is based on Resource Description Framework (RDF) that represents the semantic data and a collection of such statements shapes an RDF graph. SPARQL is a query language designed specifically to query RDF data. Linked Data faces the same challenge that Big Data does. We now lead the way to a new wave of a new paradigm, Big Data and Linked Data that identify massive amounts of data in a connected form. Indeed, utilizing Linked Data and Big Data continue to be in high demand. Therefore, we need a scalable and accessible query system for the reusability and availability of existing web data. However, existing SPAQL query systems are not sufficiently scalable for Big Data and Linked Data.

In this thesis, we address an issue of how to improve the scalability and performance of query processing with Big Data / Linked Data. Our aim is to evaluate and assess presently available SPARQL query engines and develop an effective model to query RDF data that should be scalable with reasoning capabilities. We designed an efficient and distributed SPARQL engine using MapReduce (parallel and distributed processing for large data sets on a cluster) and the Apache Cassandra database (scalable and highly available peer to peer

distributed database system). We evaluated an existing in-memory based ARQ engine provided by Jena framework and found that it cannot handle large datasets, as it only works based on the in-memory feature of the system. It was shown that the proposed model had powerful reasoning capabilities and dealt efficiently with big datasets.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a thesis titled "Distributed RDF Query Processing and Reasoning for Big Data / Linked Data," presented by Anudeep Perasani, candidate for the Master of Science degree, and hereby certify that in their opinion, it is worthy of acceptance.

Supervisory Committee

Yugyung Lee, Ph.D., Committee Chair
School of Computing and Engineering

Praveen Rao, Ph.D., Committee Member
School of Computing and Engineering

Yongjie Zheng, Ph.D., Committee Member
School of Computing and Engineering

TABLE OF CONTENTS

ILLUSTRATIONS

Figure Page

TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

The semantic web [1][2] is one of the fastest evolving technologies. Tim Berners-Lee first introduces the term "semantic web". The main idea behind the concept was to develop a technology that needs to connect all the web documents around the globe, make them available to public, and machine understandable. This connected data is also called *linked data* [3]. Figure 1.1 represents the view of semantic web. Many enterprises and organizations are constantly producing vast amount of heterogeneous data on the web. If we could combine and convert this large data into a semantically machine understandable format, then we can take advantage of sharing and reusing existing data. Applications are being developed to crawl information from different knowledge base systems across the globe.

The World Wide Web Consortium (W3C) proposed the Resource Description Framework (RDF) [4][5] to represent the semantic data, where information can be seen as simple statements like subject-predicate-object. Each such RDF statement is called a triple. In RDF, every resource is assigned with a Uniform Resource Identifier (URI). The RDF has a feature that connects each resource to a literal or to a resource and forms a connected graph named as RDF graph. As many organizations and companies are moving towards working on semantic data, RDF data is growing rapidly. According to the data collected by W3C in 2010, we had nearly 19 billion triples on the web, and this data is increasing very fast. We presently have many varieties of triple stores to store the RDF data. As data is growing rapidly, we need to have an efficient RDF processing system that should be scalable, and

highly available such that it can handle huge amount of data. One of the greatest problems with RDF processing systems is the lack of an efficient query engine. There are many RDF query languages [6] such as DQL, N3QL, RDFQ, and RDQ etc., but they possess high latency when querying RDF data. SPARQL [7][8] is one of the best RDF query language proposed by W3C and it is observed that current implementations of SPARQL are not efficient on large data sets. Researchers and organizations are doing active research to develop optimal solutions to run queries efficiently over large RDF data sets.

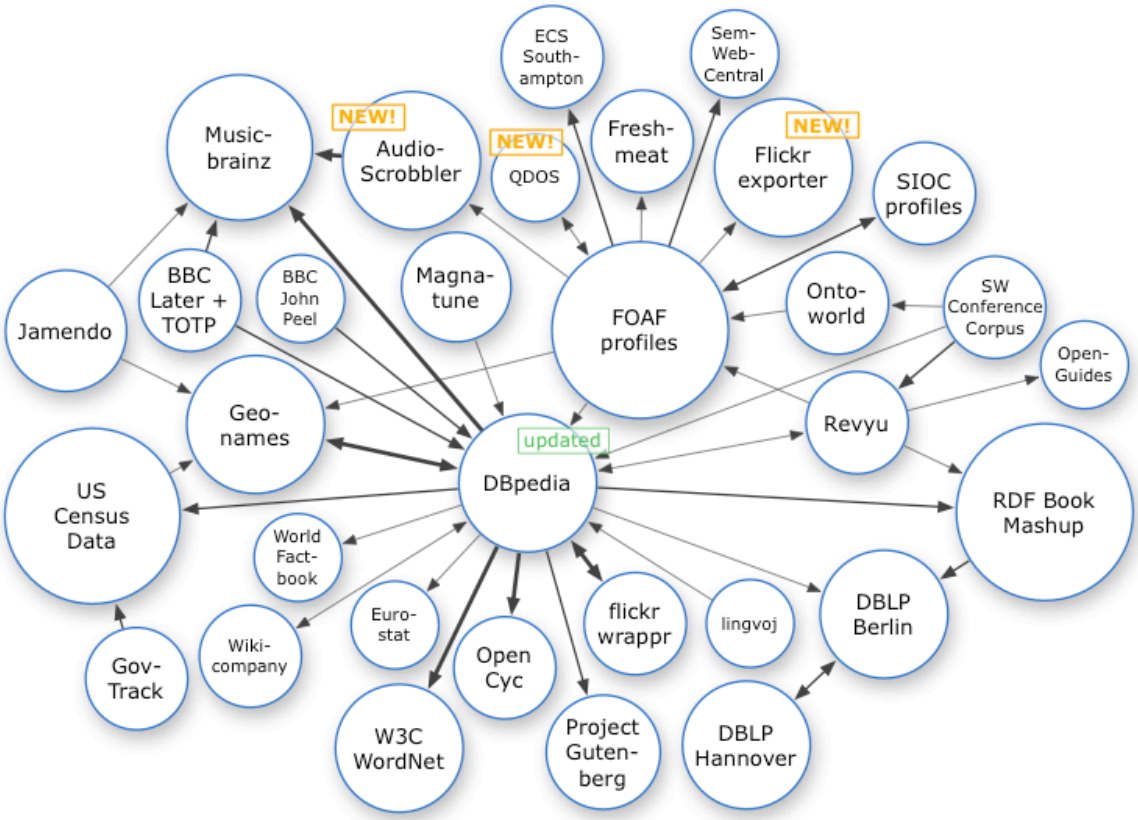


Figure 3.1 View of Semantic Web [3]

1.2 Problem Statement

The question here is of developing a distributed storage system to store linked data (RDF data) that can be fed by multiple feeders and can be used to serve many client requests concurrently. We can share our knowledge, and reuse the existing knowledge, if we have a standard distributed data storage system for RDF. As data is generated in high volumes, we need to have the storage systems that are scalable, efficient, and highly available.

The second problem is the need to have efficient query engines to serve RDF data. SPARQL Protocol and RDF Query Language (SPARQL) [7][8] is one of the RDF query languages that can be used to query RDF data. The W3C has made SPARQL a standard, and it is one of the important semantic technologies. The present implementations of SPARQL show that SPARQL queries face high latency when dealing with high volume of data. The main aim of this project is to evaluate the existing SPARQL systems, find key components that can be improved, develop a new system that can run SPARQL on huge data, and finally, compare the performance of the new system with that of the existing systems.

1.3 Thesis Outline

In this thesis, we address the issue of how to improve the scalability and performance of query processing with Big Data / Linked Data. Our aim is to evaluate and assess presently available SPARQL query engines and develop an effective model to query RDF data that should be scalable with reasoning capabilities. We have proposed an efficient and scalable distributed data storage system to handle large RDF data sets. We proposed a caching mechanism called Graph-store to save reasoning information extracted from ontologies. We designed an efficient and distributed SPARQL engine using MapReduce (parallel and distributed processing for large data sets on a cluster) [10] and the Apache Cassandra

database (scalable and highly available peer to peer distributed database system) [11]. We evaluated an existing in-memory based ARQ engine [13] provided by Jena framework and found that it cannot handle large datasets, because it only works on the in-memory feature of the system. It was shown that the proposed model has powerful reasoning capabilities and efficiently deals with big datasets.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we will briefly discuss the technologies that we have used. It includes RDF [4][5], SPARQL [7][8], Hadoop MapReduce paradigm [10], Cassandra distributed data storage [11], Apache Jena [14], Neo4j [15], and OWL [16]. Readers having basic knowledge of these technologies, can skip the background section of this chapter. In section 2.2, we have discussed related work done by other researches on this topic.

2.1 Background

Background section of this chapter discuss about related technologies we have used in our system implementation, which includes details about RDF, OWL, SPARQL, Neo4j, Cassandra, and MapReduce.

2.1.1 RDF

Resource Description Framework (RDF) [4][5] is a framework or model that was proposed by the W3C, to represent linked data on the web. It is used to describe a model that makes statements to connect two resources or to connect resource to its value. RDF does not have pre-defined schemas to represent data. However, RDF has its own schema language called Resource Description Framework Schema (RDFS). RDF statements are represented as subject-predicate-object. It is called a triple. It is similar to the object, relationship, and value in the object oriented paradigm, but RDF is not actually object oriented. Here, the subject can be mapped to the object, the predicate can be mapped to the relationship between the object and it's value, and the object can be mapped to the value. For example, RDF statement like *":student :takesCourse :graduateCourse1"* represents :student is a subject, :graduateCourse1 is an object and :takesCourse is a predicate to connect both. All such RDF

statements form a RDF graph, where nodes represents subject, object and links define the relationship among them. A typical complex RDF graph is represented as shown in figure 2.2.

RDF data is represented as directed graphs. So, we can infer basic semantics from the information when no RDF schema is available. RDF has also the ability to merge two different data sources without having defined schemas. Hence, it can be used to merge unstructured, semi-structured data and share it across the network. Producers can produce the RDF data and share on the network where as consumers can crawl it to use in their applications. This improves the reusability of existing information without having to create new one.

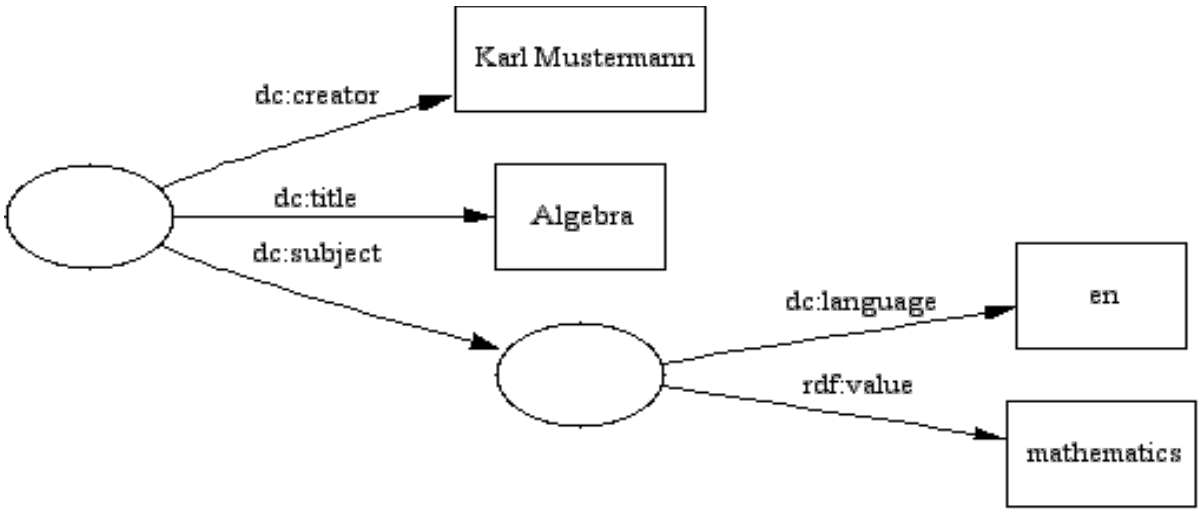


Figure 4.1 A Simple RDF Graph [17]

According to W3C standards and conventions, resources in the RDF graph are represented with circle, literals are represented with rectangle and predicate is an arrow from subject to object as shown in figure 2.1. So, nodes in the RDF graph can be a circle or a rectangle. Every subject in the RDF has a Unique Resource Identifier (URI) or a blank node.

A resource representing blank node is also called anonymous resource. A predicate is an URI, which also indicates a resource. An object can be an URI, blank node or a literal.

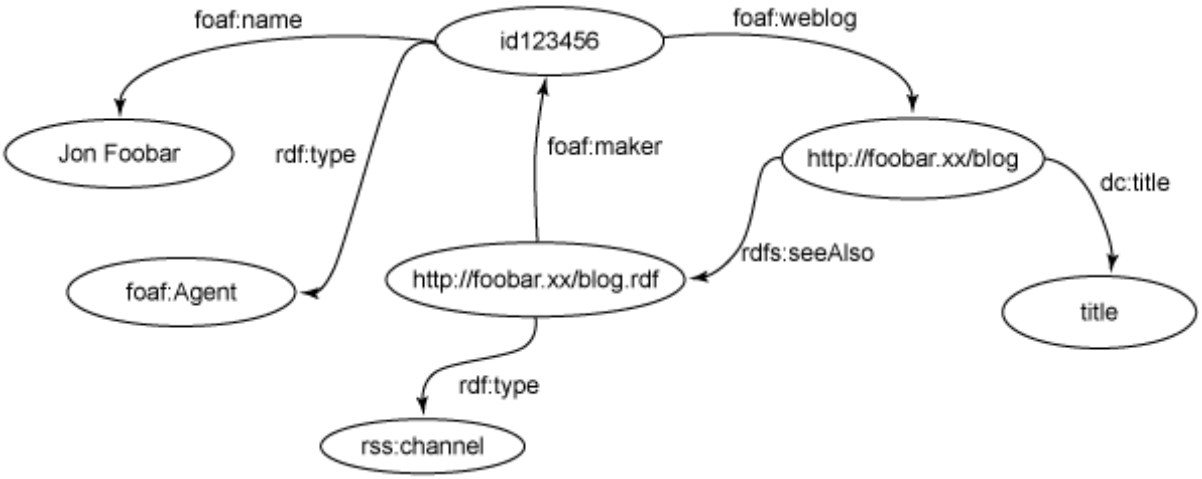


Figure 2.2 A Complex RDF Graph [18]

RDF has several serialization formats, each having its own syntax and specifications to encode the data. Some of the common serialization formats that RDF supports are Turtle, N-Triples, N-Quads, JSON-LD, N3 and RDF/XML.

Table 2.1 Sample RDF Data [17]

```
<?xml version="1.0"?> <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
ns#" xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description>
    <dc:creator>Karl Mustermann</dc:creator>
    <dc:title>Algebra</dc:title>
    <dc:subject>mathematics</dc:subject>
```

```
    <dc:language>EN</dc:language>
    <dc:description>An introduction to algebra</dc:description>
  </rdf:Description>
</rdf:RDF>
```

2.1.2 SPARQL

SPARQL [7][8] also stands for SPARQL Protocol and RDF Query Language. SPARQL, as its name suggests, is a RDF query language, which is used to get the information stored in the Resource Description Framework (RDF) format. It was recommended by W3C. A simple SPARQL query consists of triple patterns, which represents subject, predicate, and object. Complex SPARQL queries also consist of conjunctions, disjunctions, and optional patterns. There are many tools available to construct SPARQL queries, to translate SPARQL queries to other query languages such as SQL, to run SPARQL queries on NoSQL databases such as MongoDB, Cassandra.

Table 2.2 illustrates a simple SPARQL query, based on "friend-of-friend" ontology. More specifically, it should return the data set of all the person names. It has two query patterns. One is to find the triples of type person and other is to find the persons who have name associated with them.

Table 2.2 SPARQL Query Example [7]

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
    ?person a foaf:Person.
```

```
    ?person foaf:name ?name
    }
```

This is the simplest query, as it needs to match only two triple patterns. The complex URIs is split into prefixes and we include them on top of the query. “PREFIX” in the query represents namespace foaf: that can be used instead of full URI like <http://xmlns.com/foaf/0.1/>. “SELECT” identifies the variables to appear in the query results.. In our example we need to output person’s names from the final dataset. Within the “WHERE” clause, we have two triple patterns. First pattern, matches all the subjects of type person. Second pattern matches all the triples having predicate foaf:name. SPARQL should join the results from pattern 1 and pattern 2 and output the combined result dataset. A placeholder (“?”) in front of a variable indicates binding the variable to corresponding parts of each triple. SPARQL follows a procedure called federated query, where it will distribute the query to end points, computed, and results submitted.

SPARQL supports four different query forms in order to read RDF data from the databases. SELECT Query is used to get information stored in the database and results are stored in tabular format. Construct query is used to get information from the repositories and results will be converted to RDF format. ASK query is used to get simple true or false information. DESCRIBE query is used to get the RDF graph from the SPARQL endpoints.

2.1.3 Apache Cassandra Distributed Data Storage

Apache Cassandra [11] is a distributed data storage system to handle vast amount of data by providing high availability and no single point of failure. Cassandra was initially developed by Facebook and later it became Apache incubator project. Cassandra is proven

to be excellent in performance and scalability. It also provides tunable consistency and replication across the multiple datacenters. Cassandra is proven to be good at writes. In Cassandra cluster, there is no concept of master and slaves. All the individuals act as peers and they communicate using GOSIP protocol. GOSIP protocol is used to send node information to and from peers and to get ring information. Initially, when we set up the cluster we assign seed nodes. Seed nodes are regular nodes in the Cassandra cluster, which are used to start the cluster and provide ring information to the new machine joining the cluster. They are only used for initial handshaking. Ideally, one node from each data center will act as seed node.

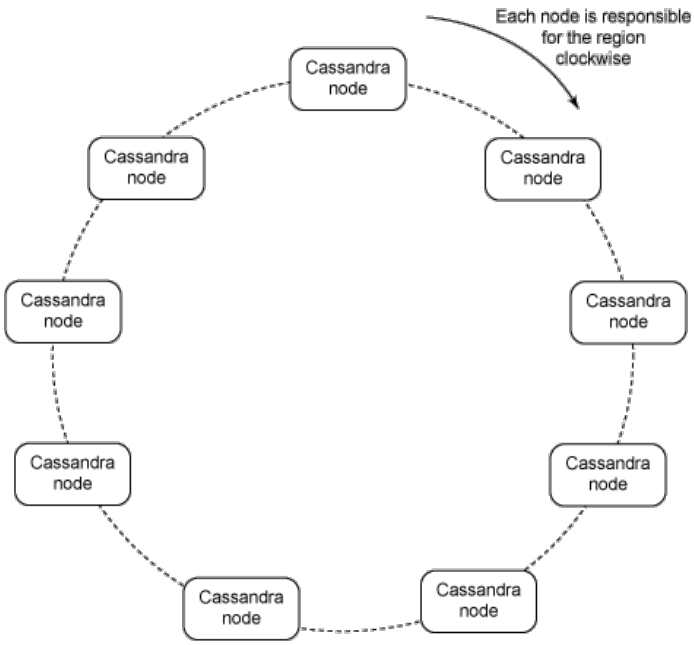


Figure 2.3 Apache Cassandra Cluster Ring [19]

In this section, we will discuss the special features that Apache Cassandra possesses. One of the important things is that Cassandra is fully decentralized. As we discussed earlier, there is no single point of failure. Even if one node stops working due to failure, other

nodes/peers continue to work properly. There is no need to install backup nodes for masters as in other distributed storages like Hadoop File System (HDFS). Each node has an equal role and data is distributed across the cluster using specified practitioner. It supports to keep multiple copies of the same data across multiple nodes on multiple data centers. Replication in Cassandra is customizable. Cassandra read/write performance increases with scalability. More the number of nodes added to the cluster, the faster reads/writes we can expect. As it keeps the data across the cluster, it is highly fault tolerant. If a node with particular data is down, then it uses other nodes in the cluster having the same data as backup. Cassandra follows Quorum protocol for tunable consistency.

Cassandra also has the support of MapReduce [12] to perform online analytics. This is useful when we want to perform analytics on data, while serving user requests. In such situations, we dedicate some nodes to serve user requests, while others run analytics on them. Besides MapReduce, other Hadoop ecosystem components like Hive, PigLatin, Oozie are also integrated with Cassandra [12].

Cassandra has its own query language known as "Cassandra Query Language (CQL)". CQL is a SQL like alternative for Cassandra, which has command line interface. CQL supports wide variety of operations like in RDMSs such as create, insert, delete, describe, alter etc. Cassandra has default Thrift [20] interface for communicating with client applications. Client applications to deal with Cassandra can be written in many languages like java, python.

Cassandra data storage is similar to that of Relational DBMs. However, there are many differences between the two. The smallest unit in the Cassandra data storage is called a column. Every column has a name, a value and a timestamp. It is similar to a table column in

relational data bases. A group of logically related columns can be referred as a row. Each row has its own primary key to uniquely identify it. The difference between a row in Cassandra

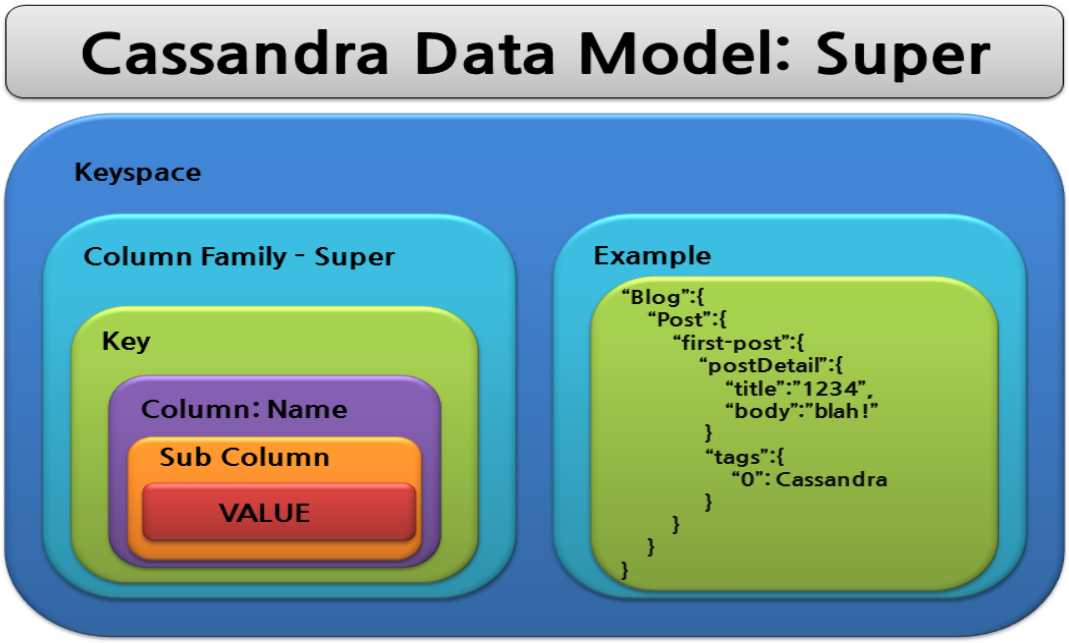


Figure 2.4 Apache Cassandra Data Architecture [21]

column family and a row in relational data base is that a row can have any number of columns in Cassandra, where as traditional databases have fixed number of columns according to its schema. Unlike in a traditional database, there is no need to define column schema at the time of table creation. Columns can be added and deleted to any row dynamically. A set of logical rows forms a Column family like Table in relational database. A group of logically related Column families form a Keyspace. Moreover, applications can use Super Columns, Super Column family to sort inner columns.

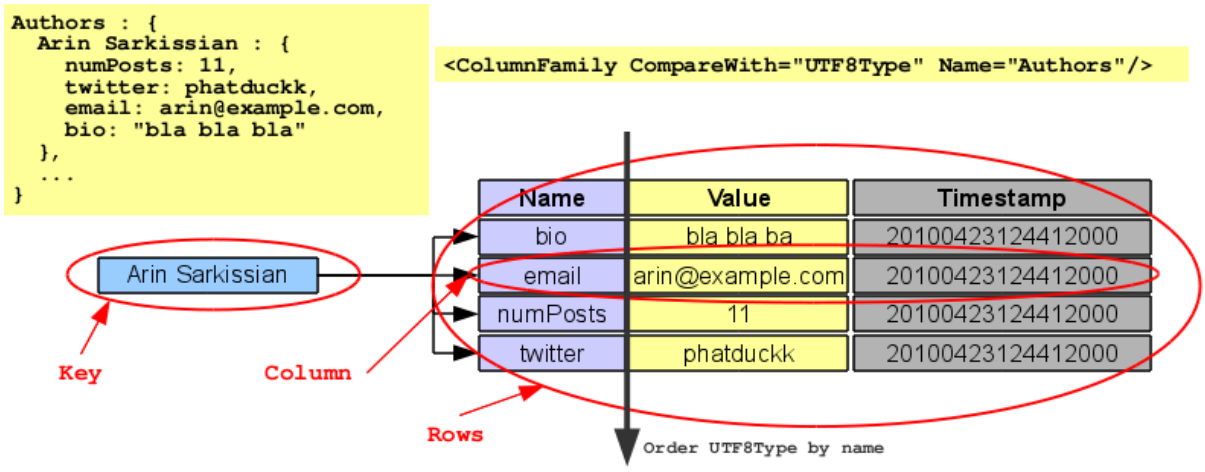


Figure 2.5 Apache Cassandra Data Storage -1 [22]

Figure 2.5 depicts illustration of Cassandra data model without super Column families. This data model consists of a Column family known as “Authors”. It has only one row with row key as “Artin Sarkissain”. This row has 4 columns and each column has name, value and timestamp associated with it. Column name “bio” has value “bla bla ba”. So, we need row key and column key to access a particular column value.

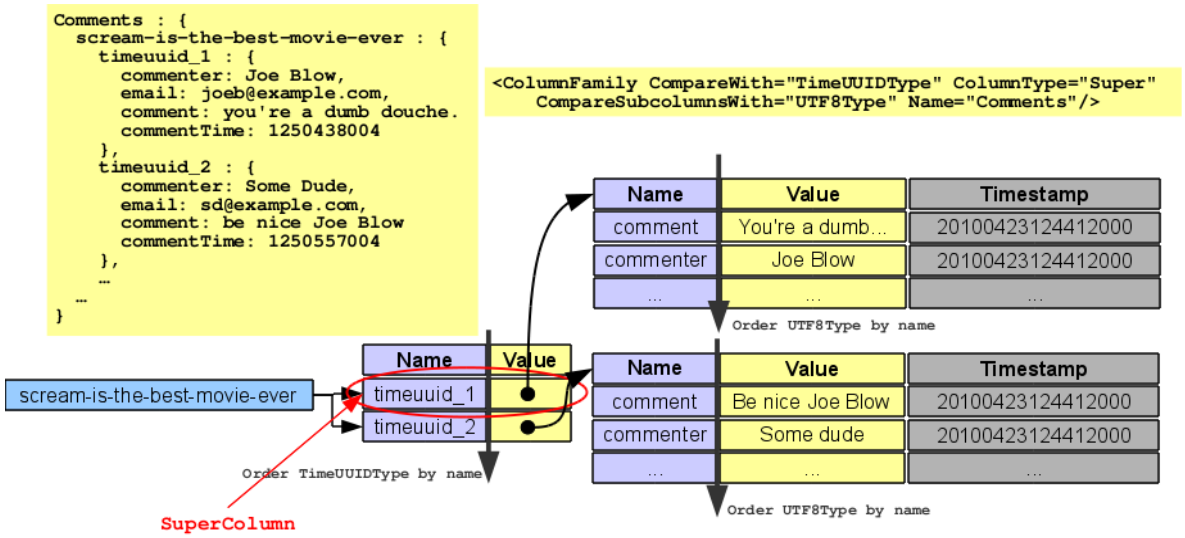


Figure 2.6 Apache Cassandra Data Storage -2 [22]

Figure 2.6 depicts Cassandra data model with super columns. In this data model, inner columns are sub divided into two sets. Division is based on super column *timeuuid*. It has column family name "Comments" and it has one row with row key as "scream-is-the-best-movie-ever". Column value field of the row consists of sub columns. Similarly, we can also define super column families. For example, Column families Department and Chair can be defined under Super Column family School.

2.1.4 MapReduce Programming Model

MapReduce [10] is a programming model to deal with large data. MapReduce executes in parallel and distributed way. It is called massive parallel processing. The main advantage of MapReduce paradigm is that it never sends data to the computation instead it sends computation near data. MapReduce consists of Map() and Reduce() functions. Map function does filter and map the relevant data. The input of the Map phase will be list of key-value pairs. We don't need to supply data as key-values. MapReduce framework will take care of everything transparently. The outcome of the Map phase will also be list of key-value pairs. The output of the Map phase will go to shuffle and sort phase, where data is shuffled and output will be pairs of key and list of values. The output of the shuffle and sort phase will be input to the Reducer() function. Reducer function does summary operations and final output of reducer function is list of key – value pairs.

$$Mapper(key1,value1) \rightarrow sorted\_list(key2,value2)$$

$$Reducer(key2, sorted\_list\ (value2)) \rightarrow result\_list(value3)$$

The MapReduce framework, automatically, splits the input data and passes them to map tasks. That is reason for being called as massive parallel programming. Moreover, Hadoop framework will speculate the MapReduce program execution.

2.1.5 Apache Jena

Apache Jena [14] provides a framework to build semantic applications. It includes a rule based inference engine and environment for RDFS, RDF, OWL and SPARQL. Apache Jena is developed in Java programming language. It provides API to write data to and get data from RDF graphs. It also provides ARQ engine, which runs SPARQL queries on RDF data. To query RDF data, a model will be created using RDF statements and SPARQL will execute on this model.

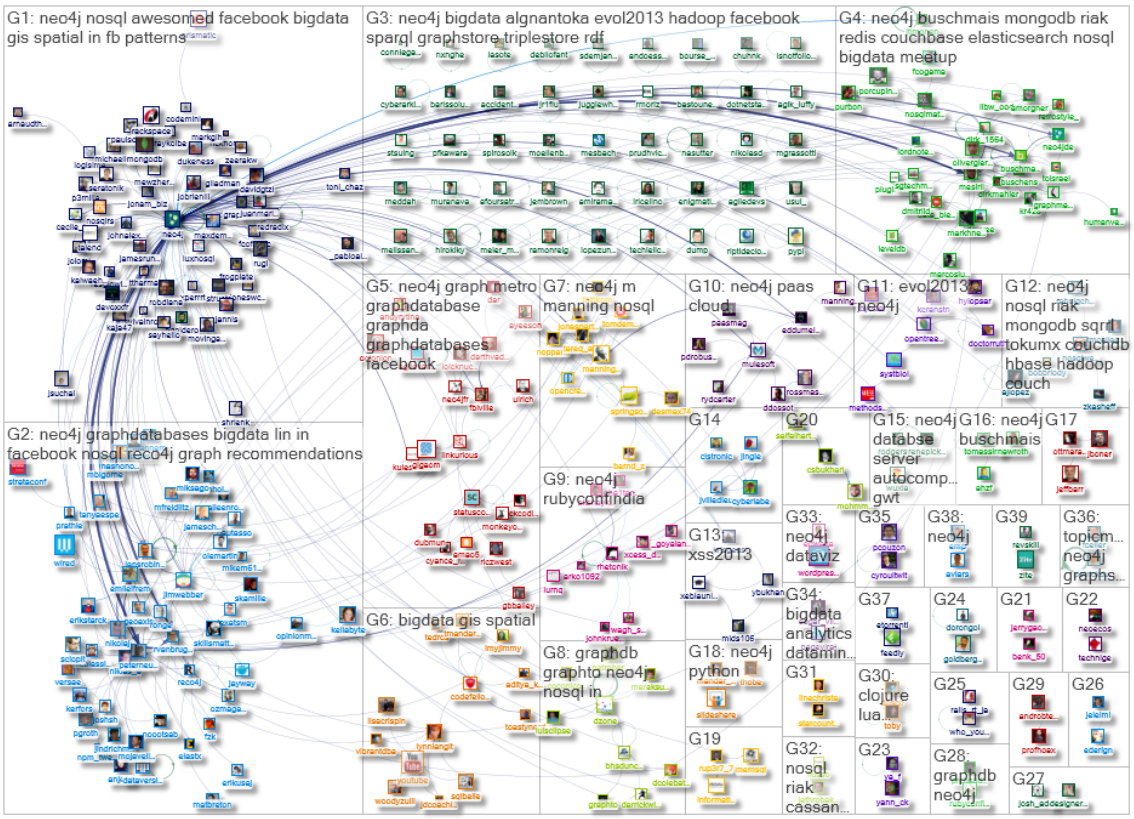


Figure 2.7 Sample Neo4j Graph Database

2.1.6 Neo4j

Neo4j [15] is a graph database to store graph patterns. It is highly scalable and robust database. Neo4j is implemented in Java and it is open source project. We used Neo4j to store

Graph-store, where global ontology will be stored. Resources and literals can be represented as a node and relationships between them are represented as links between nodes in the Neo4j databases. Neo4j is the most popular graph database available in the market.

2.1.7 Web Ontology Language (OWL)

Web Ontology Language (OWL) [16] is a part of semantic frameworks, used to represent rich, complex information (knowledge) about individuals, and relation between individuals. OWL is part of semantic web technologies. The documents represented in OWL are called ontologies. Each ontology consists of complete knowledge about a specific domain. OWL language is more useful to represent semantics. Documents represented in OWL can be interpreted and verified by computers to ensure their consistency with respect to knowledge.

2.2 Related Work

This section talks about related work done by other researchers on RDF data processing, which includes different RDF data storage systems and various RDF query execution models.

2.2.1 Efficient Processing of Semantic Web Queries in HBase and MySQL Cluster

Craig Franke et al. [23] propose a scalable resource description framework (RDF) database system to efficiently process SPARQL queries over large RDF datasets. They have designed two RDF distributed database management systems. One is using cloud technologies such as HBase, and other is using relational database such as MySQL. They also described querying schemes for HBase and MySQL cluster. They have designed algorithms for converting SPARQL queries to SQL queries , and SPARQL queries to Java programs. They implemented the prototype and compared the two models using data and

queries from the Lehigh University Benchmark (LUBM) [24]. Their main focus is on comparing two approaches for scalable semantic web data management. They concluded from their results that HBase model can deal with significantly larger data sets than MySQL model, but they did not talk about reasoning of SPARQL queries, accuracy of results. We have also concentrated on reasoning and accuracy of SPARQL queries along with distributed database management.

2.2.2 Distributed SPARQL Query Processing on RDF Data using MapReduce

P. Kulakarni implemented a way of executing distributed SPARQL on RDF data using MapReduce [25]. Their approach is based on parsing SPARQL query into sub queries and executing them on distributed RDF data using MapReduce. In the map phase, system collects all the triples matched by at least one of the query pattern specified in the SPARQL query. Later in reduce phase, the system implements multi-way joins to grab the triples satisfied by all the sub query patterns. Their implementation is focused only on query responsiveness. It does not have reasoning capabilities. But our approach is concentrated on areas, query responsiveness as well as reasoning.

2.2.3 Scalable Distributed Reasoning using MapReduce Framework

Urbani [26] proposes and implements scalable distributed reasoning using MapReduce framework. Their approach is based on applying each RDFS/OWL rule to scan the entire dataset for finding reasoning information. So, for each RDFS rule a separate MapReduce job is executed on the total dataset. Dictionary encoding techniques are used to reduce the size of the data on which reasoning has to be done. Urbani's work addressed the issue of scalable reasoning over large datasets, but executing SPARQL queries on soft-real time systems does not require complex reasoning capabilities rather fast query processing.

Our work focuses on implementing scalable RDF store and distributed RDF processing with basic reasoning capabilities involved.

2.2.4 An Efficient SQL based RDF Querying Scheme

Oracle [27] introduced a scheme for efficient and scalable querying of Resource Description Framework (RDF). It is mainly developed for querying RDF data stored in relational databases. They induced a function called RDF_MATCH that matches SQL queries against RDF data including inferencing based on RDFS rules. This approach is more suitable for small scale and enterprise level applications, but for large scale applications relational databases are not suitable in terms of handling heavy work load. Our work focuses on storing large scale RDF data on distributed commodity hardware and querying.

2.2.5 SPARQL using PigLatin

PigLatin is a script language built on top of Hadoop framework to support massive parallel processing of data. Along with parallel processing, it has extensive power of operations that are similar to SQL or relational databases. Yahoo research group developed PigLatin. As it is built for the Hadoop framework, it automatically converts all PigLatin scripts to MapReduce programs. PigLatin is useful, when running complex queries involving joins on huge data. Yahoo research group also provided a way of mapping SPARQL queries to Pig scripts [28]. This also has similar approach, as we discussed above. It performs joining of two query patterns matching on the same key. It does not have the ability to perform multi-way joins. This leads to slow performance. Yahoo's method used LUBM benchmark, which is benchmarked to test reasoning systems. Our approach is not based on mapping sub query patterns. We used Graph-store to save reasoning information and reasoning has to be done only once when the ontology is introduced.

2.2.6 Executing SPARQL Queries over the Web of Linked Data

Hartig [29] provides a method of SPARQL execution on web of linked data. This approach is used to fetch relevant RDF data at runtime during query execution time. Based upon the URIs specified in the query, RDF dataset will be resolved over HTTP and added to the queried dataset. Although it reduces the amount of data on which query is executed, HTTP network speed is a bottleneck for this approach. Our system assumes that the dataset will be fed to the system before feeding the query to be executed.

Table 2.3 Comparing Different SPARQL Models

Model Name	Query Evaluation	Data Storage	Scalability	Reasoning	Limitation
Efficient Processing of Semantic Web Queries in HBase and MySQL Cluster[23]	SPARQL to SQL, SPARQL to Java program translations	Hbase, MySQL	Uses distributed data storage and query processing	Did not talk about reasoning	Concentrated on data storage, did not talk about accuracy of results
Distributed SPARQL engine using MapReduce[25]	Basic Graph Pattern Matching and joins	HBase	Uses distributed storage system and MapReduce jobs for achieving scalability	Not concentrated on reasoning	Implemented to handle simple queries, cannot handle complex queries
ARQ engine provided by Jena [13]	SPARQL	In-memory	Cannot scale because it is based on in-memory	Has full reasoning capabilities	Not able to handle large datasets.
An Efficient SQL-based RDF Querying Scheme [27]	RDF_MATCH function was used	MySQL	Uses multiple systems to store the data in MySQL	It has reasoning capabilities	Using relational database. So, it has same limitations as relational databases expose.

RDFS/OWL reasoning using MapReduce Framework [26]	Use RDFS/OWL rules for reasoning	Hadoop File system	Uses MapReduce jobs for data storage, reasoning	Full RDFS/OWL reasoning	For every query it needs to scan entire dataset using all predefined rules

In this chapter, we have discussed briefly about RDF, SPARQL, OWL, Neo4j, Cassandra, MapReduce and Apache Jena. We have also discussed the related work done on RDF data processing. In the next chapter, we will present Graph-store based SPARQL model.

Chapter 3

GRAPH-STORE BASED SPARQL MODEL

In this chapter, we present a Graph-store based SPARQL model. It includes accepting ontologies, extracting semantic data from them, storing RDF triples, mapping SPARQL patterns with the Graph-store data and querying SPARQL on the stored data. Here, we store the reasoning information extracted from the ontologies in a graph database (Graph-store) and RDF triples in a NoSQL distributed data storage system. We used our own graph-store mapping algorithm to map SPARQL patterns with Graph-Store to infer the semantic information. Information from the graph-store helps us to fetch the relevant RDF data from the RDF-store. Figure 3.1 explains the architecture of the system.

3.1 System Architecture

The basic idea of this model is to store reasoning information in a separate graph based database and use this data efficiently at the time of query processing. This reduces the time spent for performing inference operations. This kind of system is more suitable to soft-real time systems, wherein query performance is a major factor than performing complex query operations. We used a graph database (Graph-store) to store reasoning information, a NoSQL distributed database storage system (RDF-store) to store RDF data in different column families, and a DL-reasoner for extracting the semantic data from ontology. Our system architecture consists of several software components and sub-components. There are three main components that form the core of our system architecture, namely, Ontology registration, RDF storage engine and SPRAQL query execution engine.

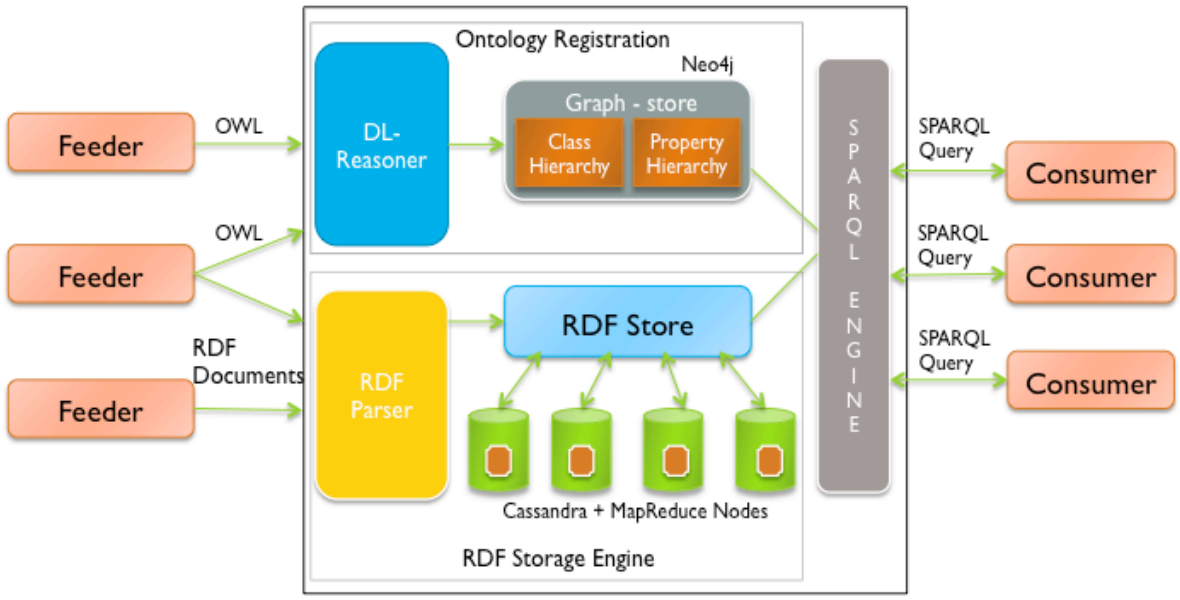


Figure 3.1 System Architecture

Figure 3.1 illustrates the overall architecture of the system. In the Ontology Registration, feeders, who want to store RDF data, should register their domain specific ontology (OWL) to the system. This will create a graph, which we call global ontology, and store it in a graph database (Graph-store). After accepting ontology (OWL) from the users and creating/updating Graph-store comes under ontology registration component. The input RDF data from the feeders and the information stored in the Graph-store is used to store the RDF triples on distributed data storage system, called the RDF-store. RDF storage engine is responsible for collecting and storing RDF data. In the SPARQL query engine, it allows the consumers to enter SPARQL queries in the form of string or text. SPARQL engine will perform actions according to Graph-store mapping algorithm presented in section 3.1.3.1, fetch the relevant data from the RDF-store, and present it to the consumers. We will elaborate each component in next section.

3.1.1 Ontology Registration

We have designed our system, so as to store RDF data from multiple domains. That being said, we need a global ontology which answers the SPARQL queries effectively because SPARQL in turn needs OWL or RDFs for inference. In the registration component, we will consider the URL of specific domain ontology and create graph which consists reasoning information from the same ontology. Before the data is stored in RDF store, data specific domain ontology must be registered. For instance, if we want to load the RDF data generated by LUBM data generator tool [24], first, we need to register LUBM university ontology [24].

Ontology registration component will take the URL of the specific domain ontology as an input. It could either be a web URL or an absolute path of an ontology file stored in the system. It recursively checks for other dependent ontologies and resolves them dynamically. This component accepts only OWL representation. We used DL-Reasoner to extract the semantic information from the ontology. It uses information extracted by DL-Reasoner to build class hierarchy and property hierarchy. Class hierarchy stores the information related to classes, inheriting classes, inherited classes. And property hierarchy stores the information related to properties and sub properties. Both these hierarchies are stored in a graph database. As it contains information from multiple domains, we call it as Global Ontology. It is a combination of different domain ontologies under the same root. It consists of all known ontology axioms such as classes, subclasses, properties, sub-properties, and relationship among them. Root node of each domain will have a link to global root node. Each node in the Graph-store is a full Internationalized Resource Identifier (IRI) of the axiom from known

ontology. This Graph-store will be updated when new ontologies are registered to the system. Duplicate ontologies will be discarded.

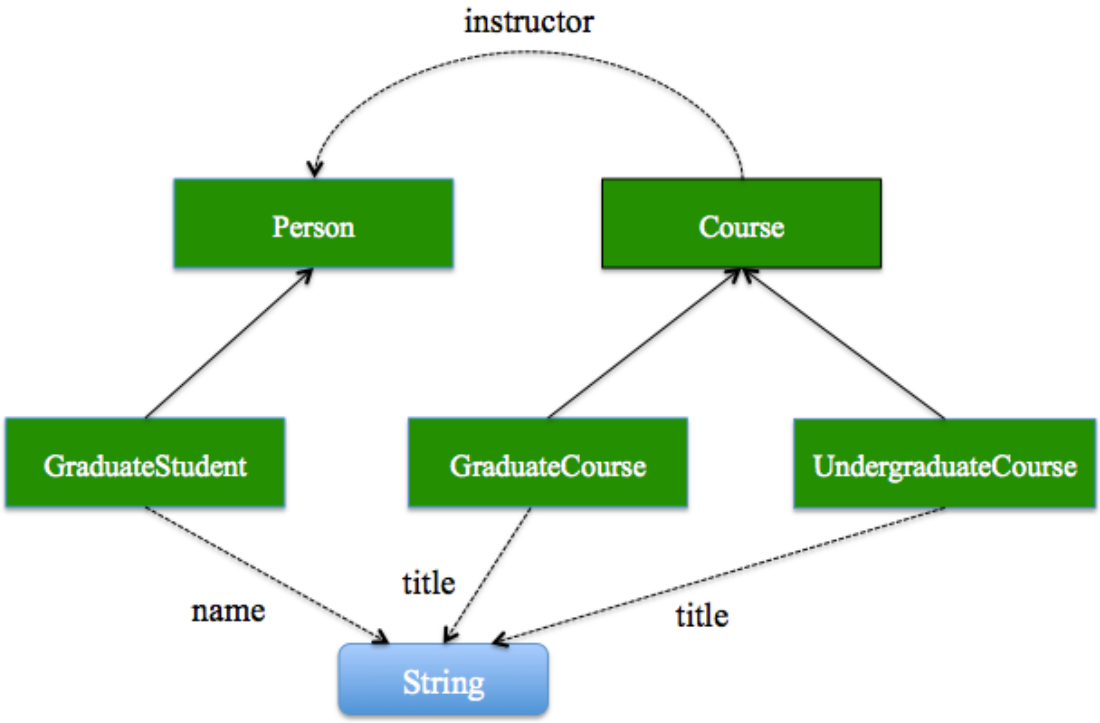


Figure 3.2 Class Hierarchy

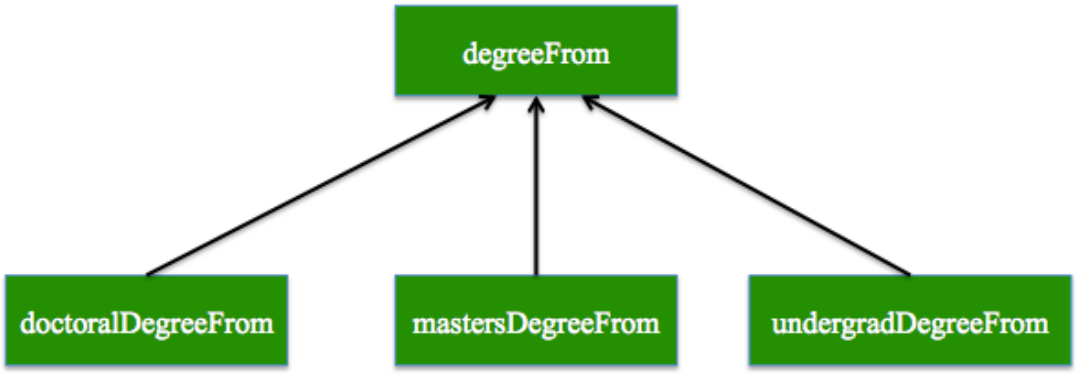


Figure 3.3 Property Hierarchy

Class hierarchy, also encodes the object properties. A link is created from the domain to range of the property. For data-type properties, range is coerced to the string

representation. Figure 3.2 represents a class hierarchy, where, classes, sub-classes, object properties and data-type properties are shown. Here, *GraduateCourse* and *UndergraduateCourse* are the sub-classes of *Course* class and object property *Instructor* has domain as *Course* class and range as *Person* class. Figure 3.3 represents property hierarchy where properties and their sub-properties are depicted. Here *doctoralDegreeFrom*, *mastersDegreeFrom*, and *undergradDegreeFrom* are sub-properties of *degreeFrom*. We have not considered consistency of graph-store, as we are updating a single ontology at a time to this extent, but have taken care of duplicates by comparing ontology URLs. This graph-store information is useful while storing RDF triples and querying RDF data. We will discuss them in detail in later sections.

3.1.2 RDF Storage Engine

We have two sub-components for RDF data storage system, a data reader and saving RDF triples to RDF-store. The data reader component uses Apache Jena [14] framework to parse the RDF data. Each RDF statement is retrieved from RDF document and stored in distributed RDF-store in such a way that it is easily accessible for retrieving. We used Cassandra NoSQL database as RDF-store. Cassandra is a key-value based storage system with replication and tunable consistency. It also supports two level key-based mechanisms using the concept of super-columns. We used four different column-families for storing RDF data namely *Concepts*, *SPOData*, *OPSData* and *InstanceData*. *Concepts* column-family is used to store classes and their instances in which, each row key is the complete IRI of a class from the registered ontologies. Each IRI of an instance of that class comes under column-value where, column key is the hash code of its corresponding column value. Each row also stores the instances of its subclasses. Since this system has been designed mainly for data

mining, the SPARQL responses are not just sets of triples associated with variables, but are complete information of each variable in the result. So, in order to retrieve full RDF information efficiently, we store complete RDF information of every instance in separate column family named *InstanceData*, in which each row key is the IRI of an instance, and column value is the full RDF information of that instance. In *SPOData* column-family, we have used subject as a row key, predicate as a super-column key, and each object as one column. In *OPSData* column-family, we used object as a row key, predicate as a super-column key and each subject as one column.

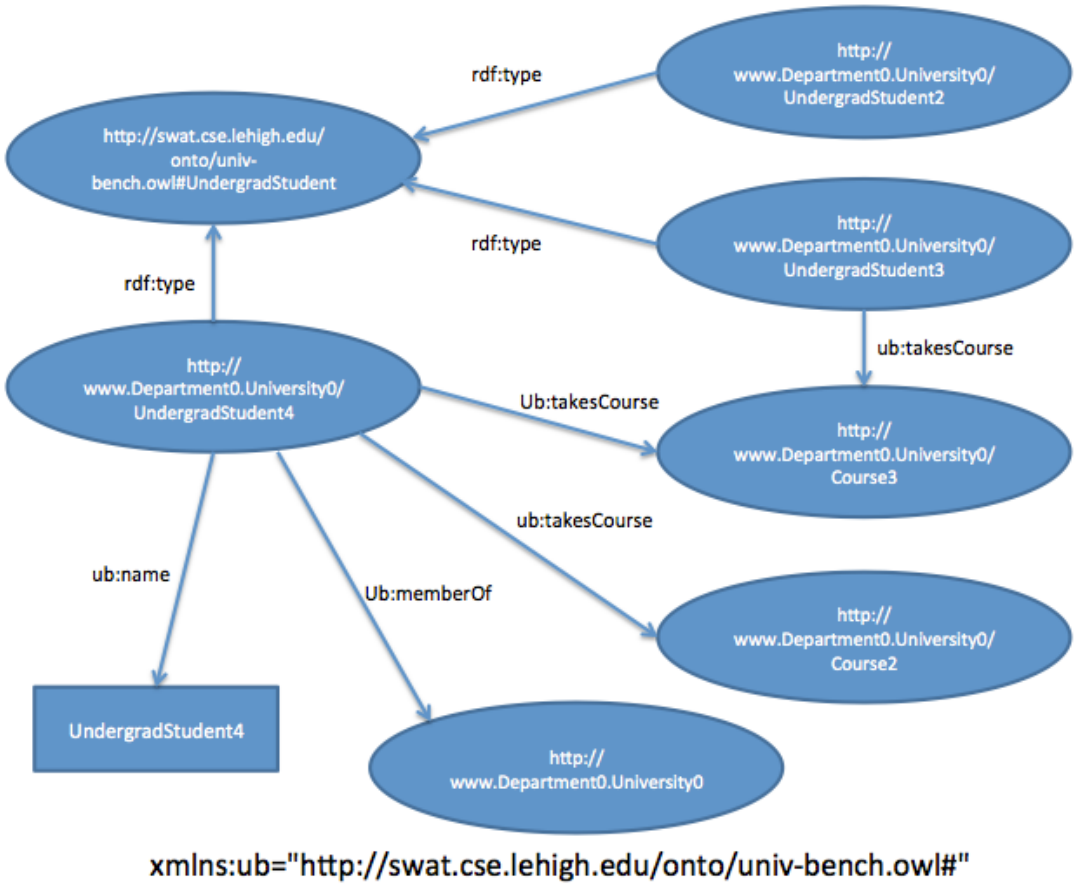


Figure 3.4 RDF Graph

Figure 3.4 is a part of full RDF graph generated by LUBM data generator tool. In this section, we will explain how we store these RDF triples in different column-families. Let us consider a RDF triple "http://www.Department0.University0/UndergraduateStudent2 -> rdf:type -> http://swat.cse.lehigh.edu/onto/univ-bench.owl#UndergradateStudent". Here http://www.Department0.University0/UndergraduateStudent2 is a subject, rdf:type is a predicate and http://swat.cse.lehigh.edu/onto/univ-bench.owl#UndergradateStudent is an object. Whenever the system receives this triple, our algorithm first checks whether predicate is of type "rdf:type" or not. If so, it recognizes that object is a full IRI from the known ontology and subject is an instance of the object. It creates a row in the *Concepts* column-family with row key as "http://swat.cse.lehigh.edu/onto/univ-bench.owl#UndergradateStudent" and column value as "http://www.Department0.University0/UndergraduateStudent2". Other instances of the same class will come under same row, but as different columns. We, also store the instances of sub-classes under the same row, as it is useful when executing SPARQL queries. For example "http://www.Department0.University0/UndergraduateStudent2" is also stored under this row with row key "http://swat.cse.lehigh.edu/onto/univ-bench.owl#Student" because

Row key	Column1	Column2	Column3
http://swat.cse.lehigh.edu/onto/univ-bench.owl#UndergradStudent	http://www.University0.Department1.edu#UndergradStudent2	http://www.University0.Department0.edu#UndergradStudent3	http://www.University0.Department0.edu#UndergradStudent4

Figure 3.5 Concepts Column – Family Layout

UndergraduateStudent is a subclass of *Student* class. Graph-store information is of great use while performing these types of inferences. Figure 3.5 is an example of a row in *Concepts* column-family.

Row key	Column1
http://www.Department0.University0.edu/ UndergradStudent4	<ub:UndergradStudent rdf:about="http:// www.Department0.University0.edu/ UndergradStudent4"> <ub:name>UndergradStudent4</ub:name> <ub:memberOf rdf:resource="http:// www.Department0.University0.edu"/> <ub:takesCourse rdf:resource="http:// www.Department0.University0.edu/Course3"/> <ub:takesCourse rdf:resource="http:// www.Department0.University0.edu/Course2"/> </ub:UndergraduateStudent>

Figure 3.6 InstanceData Column-Family Layout

As we discussed earlier, this system has been designed mainly for data mining, and the SPARQL responses are not just sets of triples associated with variables, but are complete information of each variable in the result. Figure 3.6 represents a row in *InstanceData* column-family, where http://www.Department0.University0.edu/UndergradStudent4 is a row key and column value is the complete information regarding this instance, UndergradStudent4.

Figure 3.7 represents *SPOData* column-family layout. Each subject-predicate-object triple obtained from the RDF document is stored in *SPOData* column-family. Here, we take advantage of super-columns in Cassandra, to represent relevant information under one column. In figure 3.7, http://www.Department0.University0.edu/Course3(subject) is a row key, http://swat.cse.lehigh.edu/onto/univ-bench.owl#takesCourse (predicate) is a super

column key, and http://www.Department0.University0.edu/UndergradStudent4, http://www.Department0.University0.edu/UndergradStudent3 (objects) are column-values. If object is an URI and predicate is not of type "rdf:type", then, we store object-predicate-subject triples in the *OPSData* column-family. Figure 3.8 shows *OPSData* column-family layout. As in *SPOData* column-family, predicate is a super-column key, but the difference here is object acts as row key and subject(s) are column values. Figure 3.9 illustrates that the relationship among different column-families.

Row key	Super-Column	
	http://swat.cse.lehigh.edu/onto/univ-bench.owl#takesCourse	
http://www.Department0.University0.edu/UndergradStudent4	http://www.Department0.University0.edu/Course3	http://www.Department0.University0.edu/Course2

Figure 3.7 SPOData Column-Family Layout

Row key	Super-Column	
	http://swat.cse.lehigh.edu/onto/univ-bench.owl#takesCourse	
http://www.Department0.University0.edu/Course3	http://www.Department0.University0.edu/UndergradStudent4	http://www.Department0.University0.edu/UndergradStudent3

Figure 3.8 OPSData Column-Family Layout

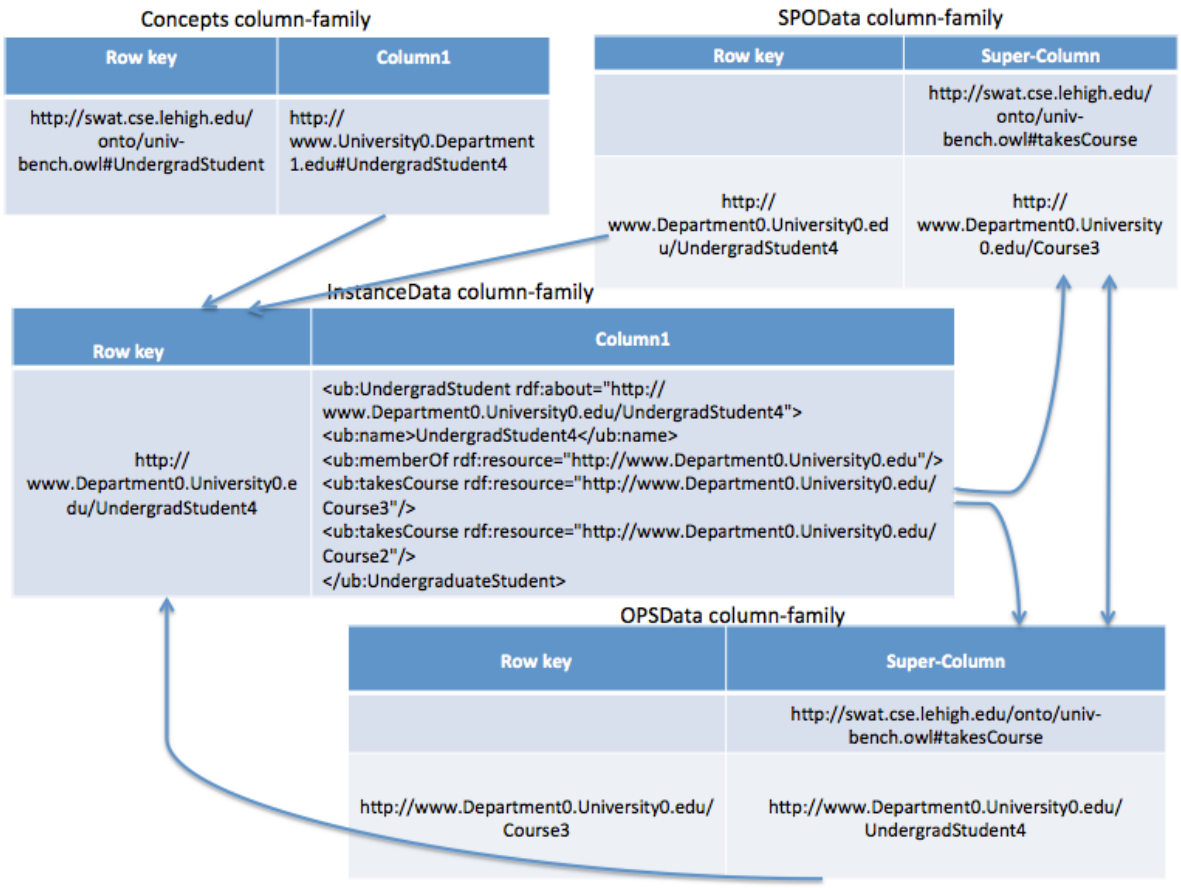


Figure 3.9 Relationship among Column – Families

3.1.3 SPARQL Query Execution Engine

Our SPARQL query execution consists of two sub-components: query mapping and query retrieval. Figure 3.10 illustrates the design of SPARQL query execution component. The system allows the end users to enter SPARQL query as input to the system in string or text format. The SPARQL interface is responsible for collecting input data from clients and presenting the output results to users. After the SPARQL interface collects the query in either string or text format, it converts the query into Query object and passes the Query object to the Query Validation component. As this system currently supports the queries of type SELECT only, the Query Validation component will check whether the input string is of

SELECT type or not. If query passes validation successfully, Query Validation module passes the Query object to the Parser. Else it will throw an error message stating that query is not valid.

Parser takes the Query object as it's input and breaks it into sub queries called triples. Each triple or pattern will be passed to an Analyzer, which uses Cassandra Hector API or Cassandra and MapReduce API to match each pattern with set of triples stored in the Cassandra nodes using the reasoning information stored in the Graph-store. We will present the algorithm used for Graph-store mapping in the next section. Analyzer executes each sub-query pattern at a time sequentially. Finally, Analyzer sends the results to SPARQL interface, which is responsible for displaying query results to the client.

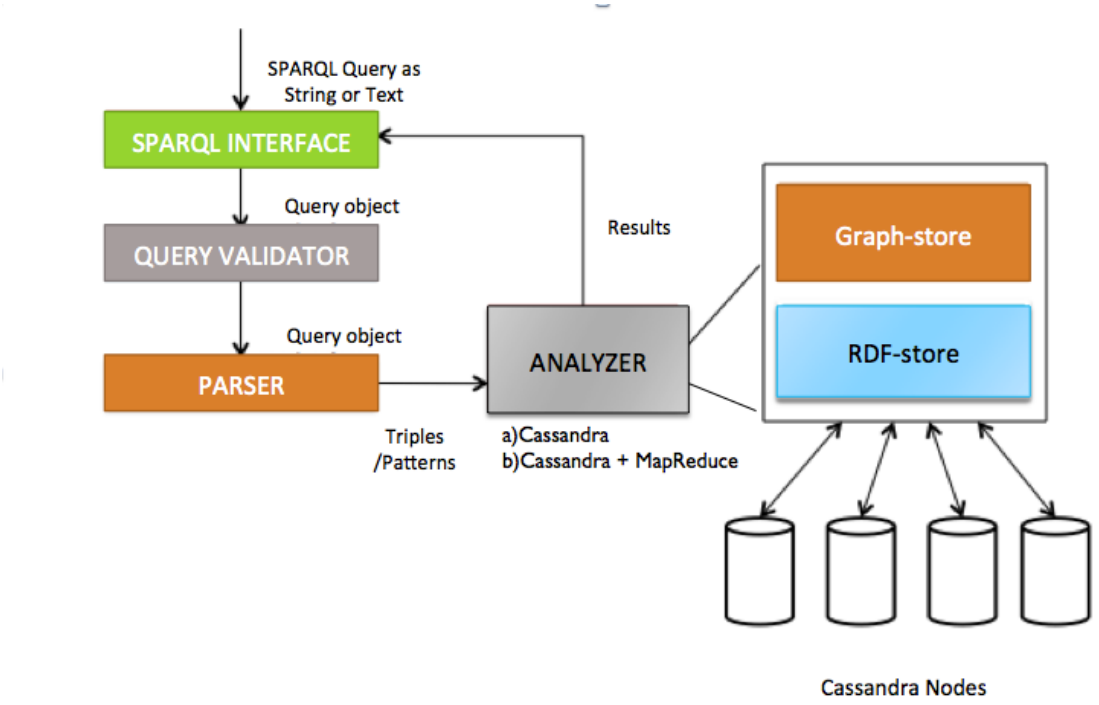
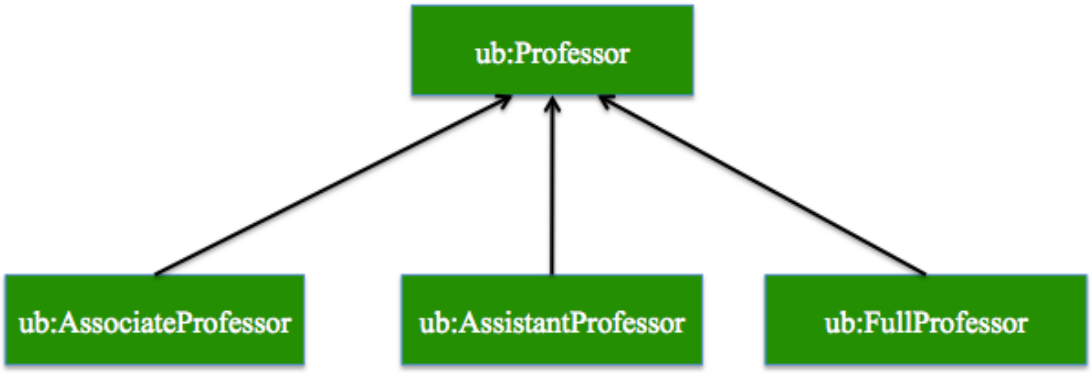


Figure 3.10 Architecture of Query Processing Component



Xmlns:ub=”http://swat.cse.lehigh.edu/onto/univ-bench.owl#”

Figure 3.11 Graph-store Information

Analyzer takes a SPARQL query object and splits it into sub queries, where each sub query is a SPARQL query pattern provided in the query. Each sub query maps to the Graph-store for performing inferences. For example, a sub query pattern *“?X rdf:type ub:professor”* maps to the Graph-store as shown in figure 3.11 and resolves to *“?X rdf:type ub:associateProfessor”, “?X rdf:type ub:assistantProfessor”, and “?X rdf:type ub:fullProfessor”* because Professor class has AssociateProfessor, AssistantProfessor and FullProfessor as subclasses in the class hierarchy of Graph-Store as in figure 3.11. The resulting three patterns will again be mapped to Graph-store to resolve further inferences. For an instance, if FullProfessor has subclasses Chair and Dean, then, it would yield *“?X rdf:type ub:chair”, and “?X rdf:type ub:dean”*. This process will continue, until, there are no subclasses or equivalent classes found in the class hierarchy. In the same way, properties are also resolved using property hierarchy information stored in the Graph-store. Once the query patterns are resolved, they get executed on the data stored in the RDF-store using the algorithm provided.

Each sub query will result in set of keys depending on the type of predicate, type of subject and type of object. For example a sub query "*?X rdf:type ub:fullProfessor"* will result in all column values of the row with row-key *"ub:fullProfessor"* from the *Concepts* column-family. Here column values are nothing but instances of the class "*FullProfessor"*. As another example, a sub query *"?X ub:name 'XYZ'"* will get the values of columns, under the super column *"ub:name" from SPOData* column-family. In the same way, all sub queries will be executed one by one according to the algorithm. At present we are doing sequential query execution, which is executing single query pattern at a time. On the flip side, it is taking more time to process. So, we are looking forward to make it parallel. The output of query mapping phase is set of IRIs, which are row keys of Cassandra column-family. Finally, the result from the last query pattern is used to fetch the required RDF data from *InstanceData* column-family. Here, mapping each sub-pattern with the Graph-store is called query mapping and retrieving data from RDF –store is called data retrieval. So, when we query for data, we need to take care of time taken by two components. The total time taken by each query is the sum of query mapping time and query retrieval time.

In the SPARQL query engine, we used two different models. The first model uses Cassandra API and the second model uses Cassandra and MapReduce API. In the first model, we fetch Cassandra column-family rows using IRIs (keys) obtained in the query mapping phase. For example, if we get 10 row keys after query mapping, 10 keys will be executed one by one to get the entire 10 rows from the Cassandra database. This is inefficient when keyset size increases. So, we integrated MapReduce paradigm with Cassandra to make data retrieval efficient. In this model, all the row keys or IRIs are written to a text file. MapReduce job reads this text file in the setup phase of MapReduce and reads only the

specified rows from the Cassandra column-family. We will explain query mapping functionality and data retrieval functionality in section 3.1.3.3 with help of an example.

3.1.3.1 Algorithm for Graph-store Mapping

Here, we will present the abstract of the algorithm that we have used for Graph-store mapping. The algorithm represents basic matching of a triple pattern to the Graph-store information.

Table 3.2 Graph-store Mapping Pseudo Code

```
Algorithm Graph-store-Mapping()
#input is query object qm
#output is Cassandra row keys
{
   if(Predicate is URI and is of RDF.type)
   {
      if(Object is not URI)
         outputs error message that a predicate expects URI object
      else
         get URI's of objects from column family "Concepts"
   }
   else if(predicate is URI)
   {
       check internal relationships of subject, predicate, object
   }
   else if(predicate if of type ANY)
   {
      if(subject and object are also type ANY)
          output error as predicate ANY cannot have subject and object as ANY
      else if(subject is URI and URI is not null)
           check internal relationships of subject, predicate, object where subject is concept
       else if(subject is URI)
           check internal relationships of subject, predicate, object where subject is
instance
       else is( subject is ANY)
       {
            if(object is URI and not null)
                check internal relationships of subject, predicate, object where subject is
ANY
           else if(object is URI)
               get object keys from column-family
           else if(object is literal)
```

```
                output error
        }
 else  output error }
```

3.1.3.2 Simple SPARQL Query Execution

We make an assumption of having a simple data layout, which consists of RDF triples in the form of subject-predicate-object. Suppose, we have sample data as shown in table 3.2.

Table 3.2 Sample RDF Dataset

Subject	Predicate	Object
X1	Name	John
X1	Email	john@mail.com
X2	Name	Steve
X2	Email	stev@mail.com
X2	Phone	408-333-5643

Consider, a simple SPARQL query like the one in table 3.3. The query first matches all the records having names and all the records having phone numbers as separate result sets. Each query pattern in the SPARQL query will produce a corresponding result set. The final result is the common data among all the result sets, which is bound to the required variables.

?X phone X2

?X name ?Y ⇔ {X1 name "John", X2 name "Steve"}

Answer is ?X, ?Y ⇔ {X2, steve}

Table 3.3 SPARQL Query-1

```
SELECT   ?X, ?Y   WHERE
{  ?X  Name   ?Y
   ?X  Phone  ?Z
}
```

3.1.3.3 Graph-store based SPARQL Query Execution

Consider a simple data storage layout, which stores the RDF data in the form of subject-predicate-object and object-predicate-subject. This storage layout is same as what we have used for storing RDF data in Cassandra storage system. We also have Concepts column-family, which stores classes and their instances. A subject can have more than one object and an object can have more than one subject as shown in table 3.4 and 3.5.

Table 3.4 Data Storage Layout -SPOData

Subject	Predicate	Object(s)
D1	Type	DoctoralStudent
D1	memberOf	University0
D1	doctoralDegreeFrom	University0
G2	Type	GraduateStudent
G2	memberOf	University2
G2	name	graduatestudent2

Table 3.5 Data Storage Layout – OPSData

Object	Predicate	Subject(s)		
DoctoralStudent	Type	D1	D2	

Consider a simple SPARQL query as an example as shown in table 3.6. This query will return all the instances of RDF graph matching the given SPARQL query pattern i.e. it will return all the variables which are of Type "Student", takes degreeFrom "University0". SPARQL query execution will start with the very first pattern and it maps the predicate *Type* Student with the information stored in the Graph-store.

Table 3.6 SPARQL Query-2

```
SELECT   ?X   WHERE
{
?X   Type   Student.                                    Pattern -1
?X   degreeFrom   ?Y.                                  Pattern -2
?Y  name   "University0". }                            Pattern -3
```

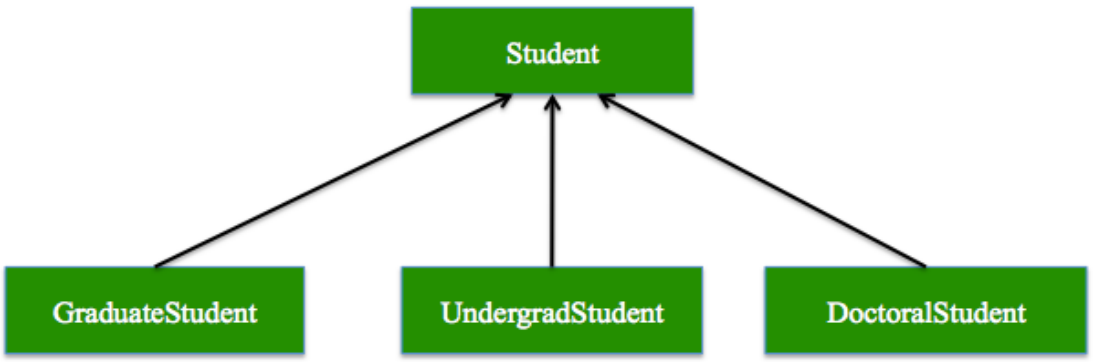


Figure 3.12 Sample Graph-store

As shown in figure 3.12, in the Graph-store mapping, system maps Student with GraduateStudent, UndergradStudent, and DoctoralStudent, as they are subclasses of Student class. Then, our system will get all the instances of type GraduateStudent, UndergradStudent, and DoctoralStudent from the "Concept" column-family. We store all the reasoning information related to classes, relationships and their instances in the column-family called "Concepts".

Table 3.7 Concepts Column – Family

Concept	Column1	Column2	Column3	….
DoctoralStudent	D1	D2		
GraduateStudent	G1	G3	G6	
UndergradStudent	U3	U5		

Table 3.8 Result from First Query Pattern

D1 is a Student because D1 is a DoctoralStudent
D2 is a Student because D2 is a DoctoralStudent
G1 is a Student because G1 is a GraduateStudent
G3 is a Student because G3 is a GraduateStudent
G6 is a Student because G6 is a GraduateStudent
U3 is a Student because U3 is an UndergradStudent
U5 is a Student because U5 is an UndergradStudent

So, the first part of the query pattern is resolved by binding the variable X with D1, D2, G1, G3, U3, U5. Now, the query system will resolve the second pattern by considering the result achieved in the first phase. First, it checks the graph-store to get the reasoning information. Here, doctoralDegreeFrom, undergradDegreeFrom, mastersDegreeFrom are sub properties of degreeFrom as shown in figure 3.13. So, it will search all the triples, which have predicate as doctoralDegreeFrom, undergradDegreeFrom, mastersDegreeFrom and subject as D1, D2, G1, G3, U3, U5. From the second query pattern, it will get the result as {Univ0, Univ1}. The third part of the query is to get all the triples that have predicate as "Name", object as "University0" and subject would be "Univ0, Univ1". We have subject Univ0 has name "University0".

?Y is Univ0

?x maker ?Y ⇔ ?x maker Univ0

?x is D1, G1

So, the answer ?X is {D1, G1}.

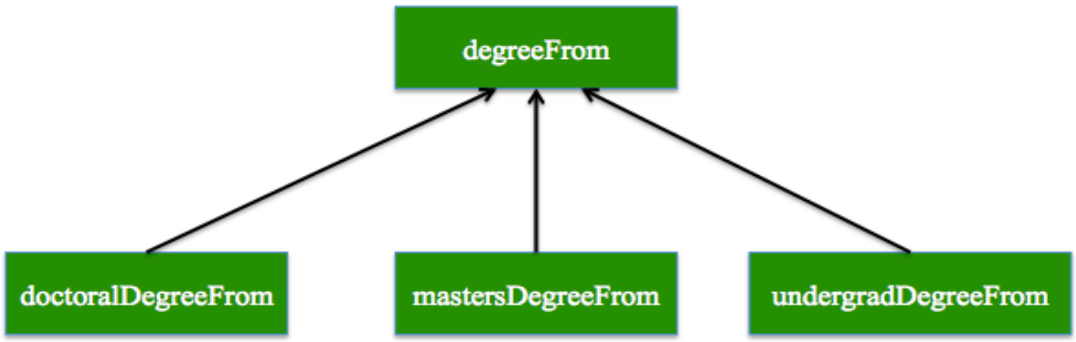


Figure 3.13 Graph-store Property Hierarchy

In this way, first, query patterns are mapped to Graph-store to check whether there is any inference to be made and then it pulls required data from Cassandra column-families. Query mapping time is same in both Cassandra, and Cassandra & MapReduce API models, the only difference is the query retrieval time.

In this chapter, we discussed Graph-store based SPARQL model, which includes ontology registration, RDF storage engine and SPARQL execution engine. In the next chapter, we will discuss implementation aspects of our system.

CHAPTER 4

GRAPH-STORE BASED SPARQL MODEL IMPLEMENTATION

In this chapter, we will discuss the detailed implementation of our system. Implementation section includes interacting with system components, process of executing SPARQL Queries on RDF using Cassandra API, process of executing SPARQL Queries on RDF using Cassandra and MapReduce, and algorithm used for Graph-store mapping.

4.1 Interacting with System Components

Our system consists of mainly three components: Ontology registration, RDF storage engine and SPARQL engine. We have defined different RESTful web services to interact with individual components in the system. We used Jersey [30] to develop RESTful web services and created HTTP web server using Grizzly [31]. Grizzly provides Java API for creating and interacting with HTTP web server. So, calling respective RESTful services via HTTP server is doing all the interactions with the system. Each service performs required actions by executing specific code section.

4.2 Executing SPARQL Queries with Jena API

Apache Jena [14] supports execution of SPARQL queries via Jena's *com.hp.hpl.jena.query* package. It contains several classes and methods to run SPARQL queries on RDF data. *QueryFactory* is one of the classes, which has several *create()* methods to read a string query or a textual query. *QueryFactory* class supports several different parsers. Every *create()* method in *QueryFactory* class will output a *Query* object, which is a data structure to represent the queries. *Query* object holds an instance of query execution engine. This *Query* object is input to the *QueryExecution*, which will execute the query on the model.

The other input to Jena's *QueryExecution* is the RDF graph. We can create an empty RDF model using *ModelFactory.createModel()*, which will return an instance of *Model* class. Once we have created an empty model, we can read a specific RDF graph into the empty model using *read(input, null)* method.

The next step in this process is executing a constructed SPARQL *Query* on the RDF data, which is loaded into an empty model. To execute the *Query* object on the RDF Model, we need to create an instance of the *QueryExecution*, that represents a single time execution of query. *QueryExecutionFactory.create(query, Model)* will help us to create an instance of *QueryExecution*. This method requires two parameters, the query to be executed and model on which the query should be executed. The *QueryExecution* interface has various execution methods, where each method executes a different type of query. Since this system is designed for only simple SELECT queries, we use *execSelect()* method only, for the execution of SELECT query. This method will then return *ResultSet object*. We then iterate over *ResultSet* using *hasNext()* method to output all the solutions bound to the query pattern. Each solution is saved as an instance of *QuerySolution,* which is a class that holds single answer from a SELECT query. In the next section, we will discuss the implementation of Graph-store based SPARQL query model.

Table 4.1 Executing a Simple SELECT Query using Jena API [18]

```
Import com.hp.hpl.jena.query.*;
Model model = ….;
String query = " …. ";
Query q = QueryFactory.create(query);
QueryExecution queryExec = QueryExecutionFactory.create(q,model);
```

```
try {
ResultSet results = queryExec.execSelect();
        for (; results.hasNext() ; )
        {
          QuerySolution querySoln = results.nextSolution();
          RDFNode x = querySoln.get("varName");    //Get a result variable by name
          Resource r = querySoln.getResource("varR");//Get a result variable–must be a
resource
          Literal l = querySoln.getLiteral("VarL"); //Get a result variable – must be a literal
        }
 }
 finally { queryExec.close();  }
```

4.3 SPARQL Query Processing using Cassandra Hector API

We have used two different models for SPARQL query execution. In the first model, we have executed SPARQL queries on RDF data by using Cassandra Hector API [32]. Hector is a high level java client for Apache Cassandra distributed data storage system. It has several features that include, simple object – oriented interface, pooling mechanism, Java Management Extensions (JMX) for management, automatic discovery of Cassandra cluster, automatic discovery of failed hosts, suspension of downed or unresponsive hosts, load balancing etc. Cassandra in-built feature provides low level Thrift API [20] for clients, but, Hector is a high level object oriented interface for Cassandra, which encapsulates the underlying Thrift API methods and structs. That is why we used Hector API.

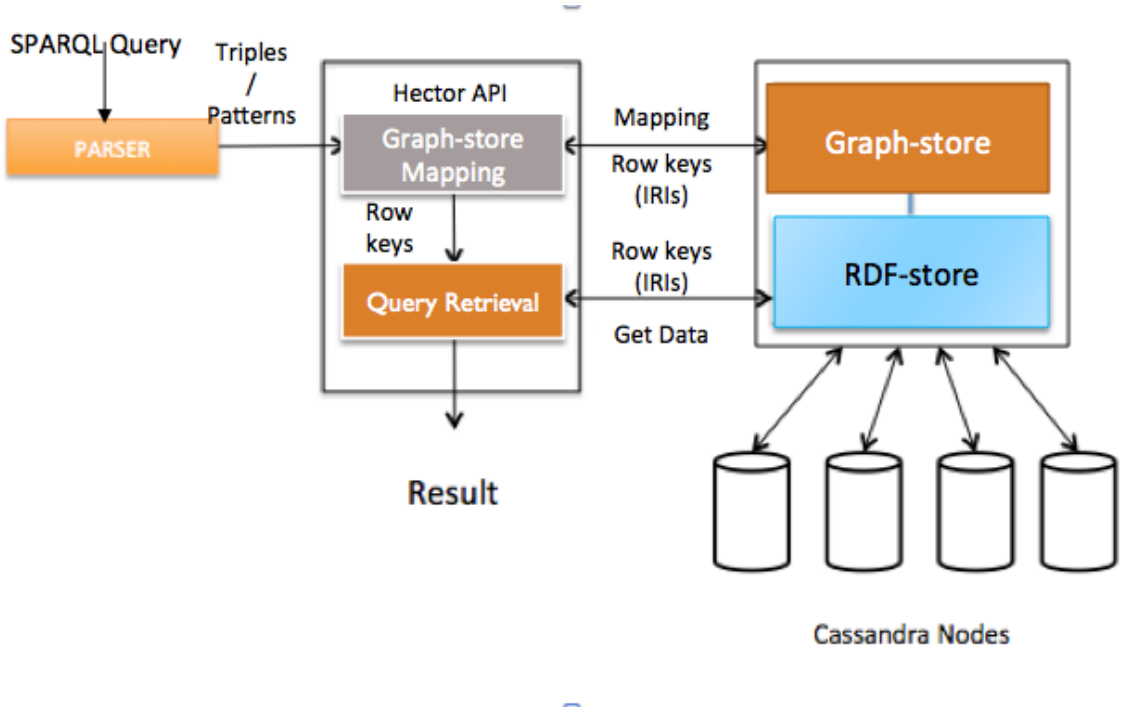


Figure 4.1 SPARQL Query Model using Cassandra Hector API

SliceQuery is an interface supported by Hector API, which has several methods that can be extended to query Cassandra keyspace. HFactory is a class in Hector API to create mutations and queries. It has bunch of static factory methods. Using HFactory, we can create Cassandra keyspaces, column families, columns, rows as well as queries. In the above example, *createSliceQuery* method takes input as Keyspace name, type of row serializer, type of name serializer and value serializer. This method creates a query, which is an instance of *SliceQuery*. *setKey* method is used to set row key and *setColumnFamily* is used to set Cassandra column family in which data is stored. Table 4.3 shows piece of code for retrieving data from Cassandra data store using Hector API.

Table 4.2 Data retrieval Code - Hector API

```
SliceQuery<String, String, String> query = HFactory.createSliceQuery(keyspace,
key_serializer, name_serializer, value_serializer);

        query.setKey(key);
```

```
query.setColumnFamily("column_family_name");
```

4.4 SPARQL Query Processing using Cassandra and MapReduce API

Hadoop integration in Cassandra [12] was added from version 0.6. There are many packages that are added to support MapReduce operations on Cassandra data. *Org.apache.cassandra.hadoop.**, *org.apache.cassandra.db.**, *org.apache.cassandra.** are some packages usually required to integrate MapReduce functionality on top of Cassandra nodes. To support data retrieval from Cassandra, we have *InputSplit, InputFormat* and *RecordReader* classes. *RecordReader* is used to read data, record by record. *InputFormat* is used to specify the type of input data for MapReduce. For Cassandra data, it is *ColumnFamilyInputFormat* . Using *ConfigHelper* class, we have to specify the input RPC port number, initial address of Cassandra cluster, input partitioner, input column family etc. If we want to store the output of Map or reduce in Cassandra column family, we also need to give output RPC port number, output initial address, output partitioner, output column family name etc. *SlicePredicate* in Cassandra will take input as a list of column names or slice range. It tells MapReduce as to which of the columns from each row it has to read.

Table 4.3 Job Configuration Settings – Cassandra and MapReduce API

```
job.setInputFormatClass(ColumnFamilyInputFormat.class);
ConfigHelper.setInputRpcPort(job.getConfiguration(), "9160");
ConfigHelper.setInputInitialAddress(job.getConfiguration(), "localhost");
ConfigHelper.setInputPartitioner(job.getConfiguration(),
"org.apache.cassandra.dht.Murmur3Partitioner");
ConfigHelper.setInputColumnFamily(job.getConfiguration(), KEYSPACE,

COLUMN_FAMILY);

SlicePredicate predicate = new
```

```
SlicePredicate().setColumn_names(Arrays.asList(ByteBufferUtil.bytes(columnName)));
ConfigHelper.setInputSlicePredicate(job.getConfiguration(), predicate);
```

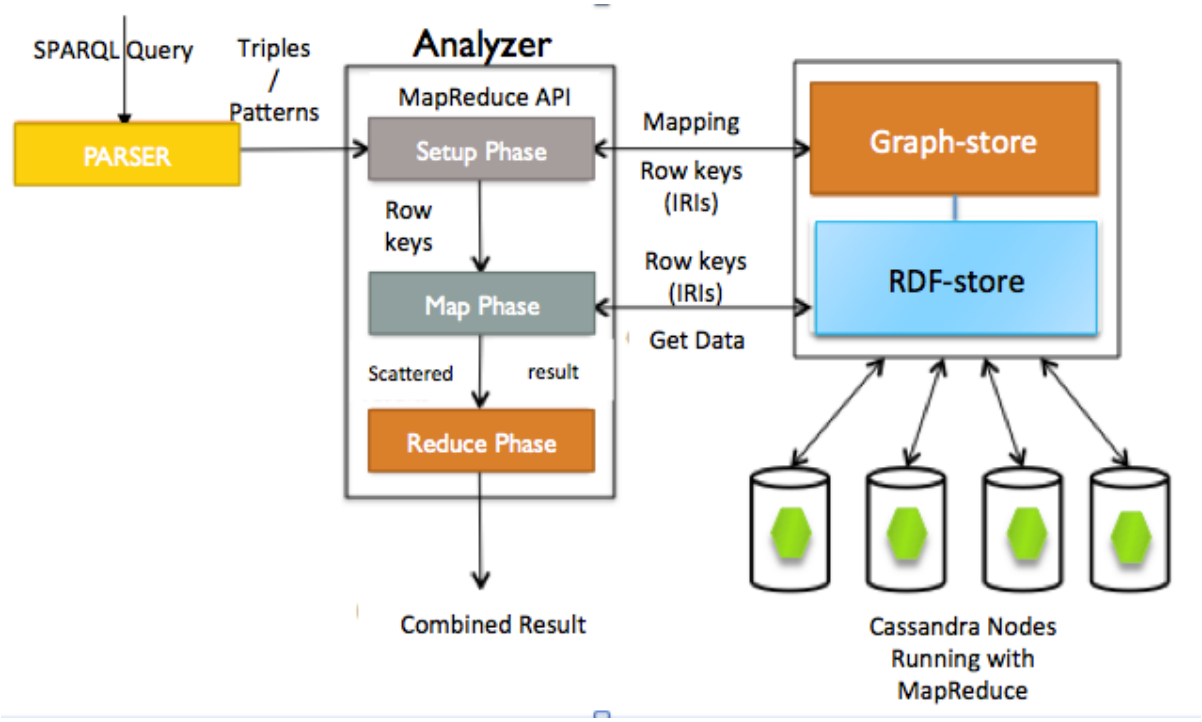


Figure 4.2 SPARQL Query Model using Cassandra and MapReduce API

Input to the Map phase is Cassandra rows. It reads each Cassandra row key and associated columns in the format of *SoredMap<ByteBuffer, IColumn>*. *ICoumn* is an interface that represents a column in the Cassandra. Using *get(), value()* methods, we can retrieve particular column value associated with column name.

Table 4.4 Map Function

```
public void map(ByteBuffer key, SortedMap<ByteBuffer, IColumn> columns, Context
context) throws IOException, InterruptedException
    {
      IColumn column = columns.get(sourceColumn);
      if (column == null)
```

```
            return;
        else map logic goes here;
}
```

In this chapter, we have discussed technical and implementation aspects of our system. We discussed about interaction with system components, SPARQL query execution using Cassandra Hector API, and SPARQL query execution using Cassandra & MapReduce API. In the next chapter, we will discuss our system evaluation.

CHAPTER 5

RESULTS AND EVALUATION

In this chapter, we discuss the test environment setup and the test results. Tests were performed with different datasets under different testing environments. First, we discuss the test environment characteristics and test data characteristics. At the end, we will compare and discuss the evaluation results.

5.1 Performance Results

In this section, we discuss about performance of each component in our system. First, we discuss test environment used for system evaluation such as system configuration, data set and queries. Next, we will present time taken for ontology registration, time taken for RDF storage engine and time taken for executing different SPARQL queries.

5.1.1 Single Node Cluster Setup

We have used Microsoft Windows Azure [33] instance running Ubuntu 12.04 LTS software for all the tests on single node machine. Tests were run on hardware configuration of 3.5GB RAM, 64-bit AMD Opteron™ processor 4171 HE X 2 , and 30GB Hard Drive. JVM 1.7.0_51 was used. DataStax Enterprise edition 3.2.5 [34] was used for Cassandra and Hadoop eco system components. Neo4j community edition 1.9.2 was used for Graph-store storage.

5.1.2 Multi Node Cluster Setup

A setup of four (4) node Microsoft Windows Azure VM's cluster was used. Ubuntu 12.04 LTS environment was used on each machine. Each instance had AMD Opteron™ processor 4171 HE X 2 and 3.5GB of RAM. DataStax Enterprise edition 3.2.5 was installed

on each machine to run Cassandra and Hadoop MapReduce. One of the four nodes, ran the system and one node acted as job tracker while others acted as data nodes.

Table 5.1 Test Data Characteristics

LUBM Data/Number of Instances	LUBM(5,0)	LUBM(10,0)	LUBM(20,0)
Classes #	43	43	43
Object Properties #	25	25	25
Data-type Properties #	7	7	7
Class Instances #	129533	263427	556572
Property Instances #	516116	1052895	2224750
Triples #	646128	1316993	2782419
Data Size	44.8 MB	124.6 MB	259.2 MB

5.1.3 Data and Queries

We have generated three different datasets, using LUBM UBA data generator tool. LUBM(5,0) represents the data generated for 5 universities, starting from index 0, LUBM(10,0) represents the data generated for 10 universities, starting from index 0, and LUBM(20,0) represents the data generated for 20 universities, starting from index 0. The total number of class instances and property instances for each data set is as shown in the table 5.1. The reason behind taking LUBM(5,0), LUBM(10,0) and LUBM(20,0) datasets is for easy comparison of our experimental with the LUBM benchmark results for accuracy. LUBM benchmark provides 14 queries to test knowledge base systems. Queries 11,12,13 are

not applicable to this system, as we are not performing full OWL reasoning yet. Queries 2, 7, 8, 9 are also excluded because of no response from the system. This might occur because of timeout or Jena memory restriction. Since we are focusing on the response time, basic reasoning and scalability of the system, we did not implement queries 2, 7, 8, 9, 11,12,13. We ran each test query 5 times and we excluded the first tow (2) test results. We averaged remaining three test results and presented in this report. Below are the seven (7) LUBM benchmark queries, we used for testing.

Table 5.2 Nature of Queries

Query	Nature	Can our system handle?
Q1	Basic join	Yes
Q2	Subclass, sub-property, Transitive	No
Q3	Subclass	Yes
Q4	Subclass, Data property	Yes
Q5	Subclass, Sub-property	Yes
Q6	Subclass	Yes
Q7	Subclass	No
Q8	Sub-class, Sub-property, Data property	No
Q9	subclass	No
Q10	subclass	Yes
Q11	Subclass, transitive property	No
Q12	Subclass, transitive property	No
Q13	Subclass, inverse property, sub property	No
Q14	subclass	Yes

Table 5.3 LUBM Query – 1 [24]

```
# This query bears large input and high selectivity. It queries about just one class and
# one property and does not assume any hierarchy information or inference.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:GraduateStudent .
  ?X ub:takesCourse
http://www.Department0.University0.edu/GraduateCourse0}
```

Table 5.4 LUBM Query – 3 [24]

```
# This query is similar to Query 1 but class Publication has a wide hierarchy.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:Publication .
  ?X ub:publicationAuthor
http://www.Department0.University0.edu/AssistantProfessor0}
```

Table 5.5 LUBM Query – 4 [24]

```
# This query has small input and high selectivity. It assumes subClassOf relationship
# between Professor and its subclasses. Class Professor has a wide hierarchy. Another
# feature is that it queries about multiple properties of a single class.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y1, ?Y2, ?Y3
WHERE
{?X rdf:type ub:Professor .
  ?X ub:worksFor <http://www.Department0.University0.edu> .
  ?X ub:name ?Y1 .
  ?X ub:emailAddress ?Y2 .
  ?X ub:telephone ?Y3}
```

Table 5.6 LUBM Query – 5 [24]

```
# This query assumes subClassOf relationship between Person and its subclasses
# and subPropertyOf relationship between memberOf and its subproperties.
# Moreover, class Person features a deep and wide hierarchy.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:Person
 ?X ub:memberOf <http://www.Department0.University0.edu>}
```

Table 5.7 LUBM Query – 6 [24]

```
# This query queries about only one class. But it assumes both the explicit
# subClassOf relationship between UndergraduateStudent and Student and the
# implicit one between GraduateStudent and Student. In addition, it has large
# input and low selectivity.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {?X rdf:type ub:Student}
```

Table 5.8 LUBM Query – 10 [24]

```
# In this it only requires the (Implicit) subClassOf relationship between GraduateStudent
#and Student, i.e., subClassOf relationship between UndergraduateStudent and Student
#does not add to the results.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:Student .
  ?X ub:takesCourse
        <http://www.Department0.University0.edu/GraduateCourse0>}
```

Table 5.9 LUBM Query – 14 [24]

```
# This query is the simplest in the test set.
# This query represents those with large input and low selectivity and does not assume any #
hierarchy information or inference.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {?X rdf: ype ub:UndergraduateStudent}
```

5.1.4 Ontology Registration Time

Ontology registration time is the time taken to build class and property hierarchies. We ran 5 tests for each ontology and then averaged them. The LUBM ontology consists of 43 classes, 25 object properties and 7 datatype properties. It takes about 2570 ms to build class and property hierarchies in the Graph-store. The DBpedia ontology version 3.6 consists of 272 classes, 629 object properties and 706 datatype properties takes 4640 ms to update the Graph-store. We have observed from the above results that ontology registration is quite efficient even when data size increases.

5.1.5 Data Loading Time

Data loading time is the time taken to parse a RDF document and save the triples in the RDF-store. We have used LUBM synthetic dataset for evaluating data loading time. We ran each test 5 times and taken their average. Each RDF file, with size ranging from 400 – 600 KB, takes about 3 sec to 6 sec for loading into RDF-store. LUBM (5,0) has 93 RDF files and it takes 372 sec for parsing and loading entire dataset. LUBM (10,0) has 189 RDF files and takes about 854 sec for loading into RDF –store. LUBM (20,0) having 402 RDF files takes 1708 sec for data loading. We can say that data loading is efficient and scalable even data size increases.

5.1.6 Query Processing Time

Query processing time is sum of the time taken to map the query to the Graph-store and the time taken to fetch the RDF-store. We have evaluated query mapping time and data retrieval time separately. We have used LUBM queries 1, 3, 4, 5 ,6 , 10, 14 to evaluate query processing time on 3 different datasets (LUBM(5,0), LUBM(10,0), LUBM (20,0)). Each query was executed on Single node machine and 4-node cluster. Each query has performed on every dataset. We have two separate querying systems, one uses Cassandra API to query the data, while other uses Cassandra and MapReduce API.

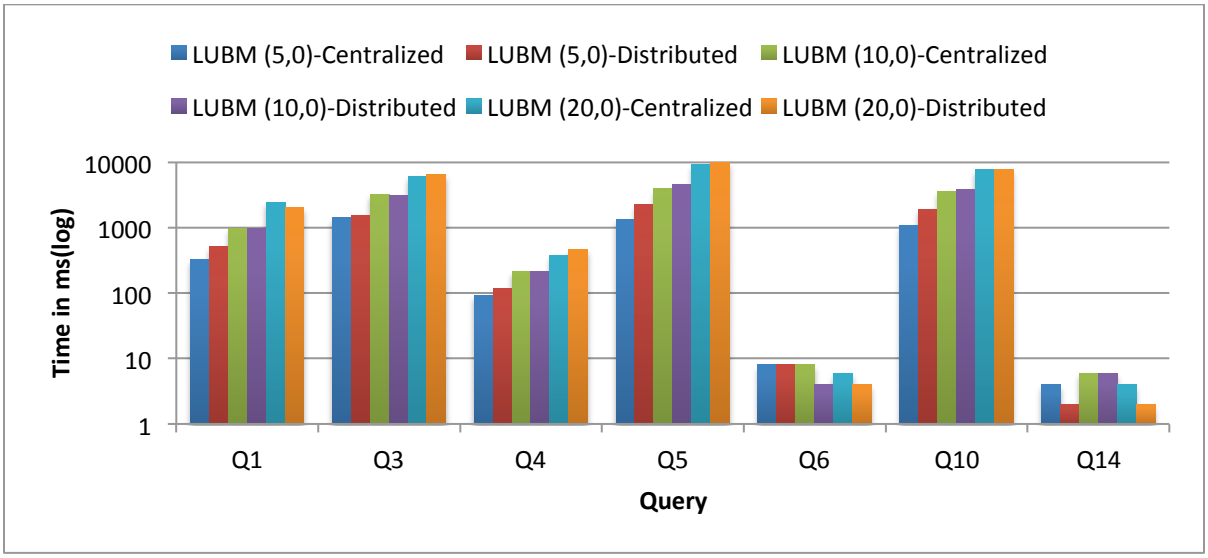


Figure 5.1 Query Mapping Time

We observed that query mapping time is quite efficient in both centralized and distributed models because query resolution is being done on one machine only. It is also observed that queries Q6 and Q14 takes less than 10ms for graph-mapping because they do not need much reasoning information.

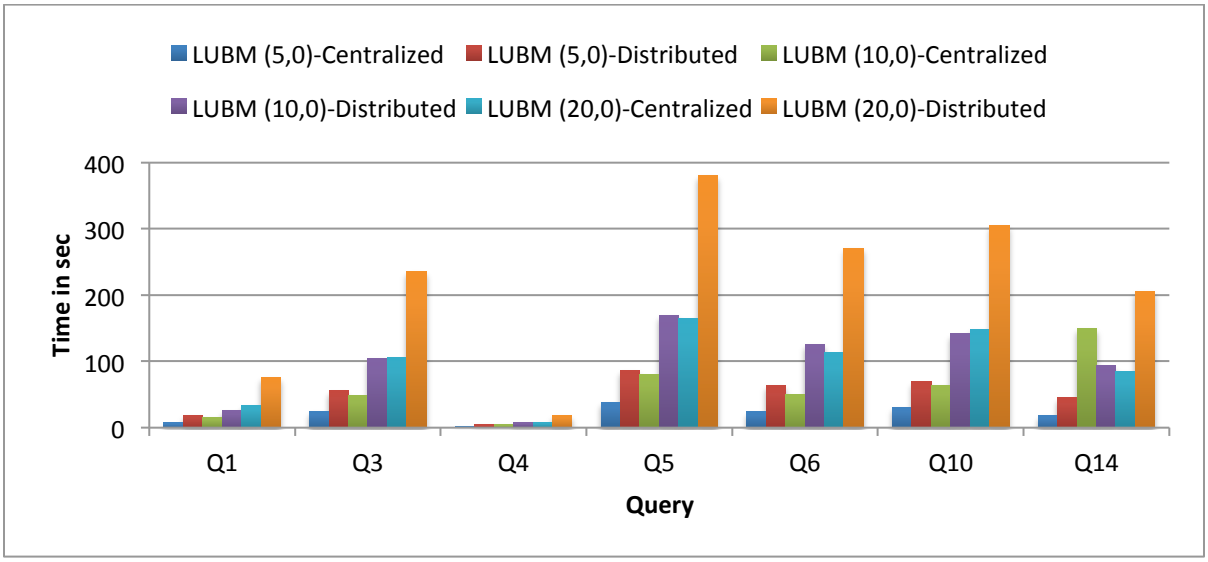


Figure 5.2 Data Retrieval Time using Cassandra API

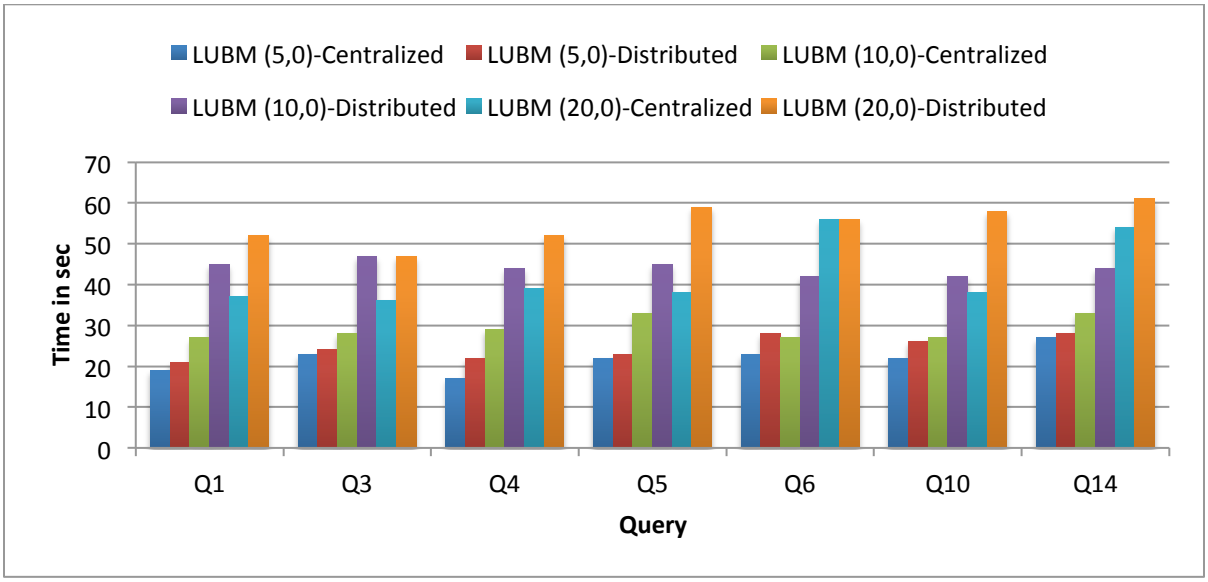


Figure 5.3 Data Retrieval Time using Cassandra and MapReduce API

Data retrieval time is better in a centralized system than a distributed system when using Cassandra API. It is improved when Cassandra nodes integrated with MapReduce because MapReduce can perform massive parallel work on the data, but it is still better in centralized system. This is because of doing sequential query processing.

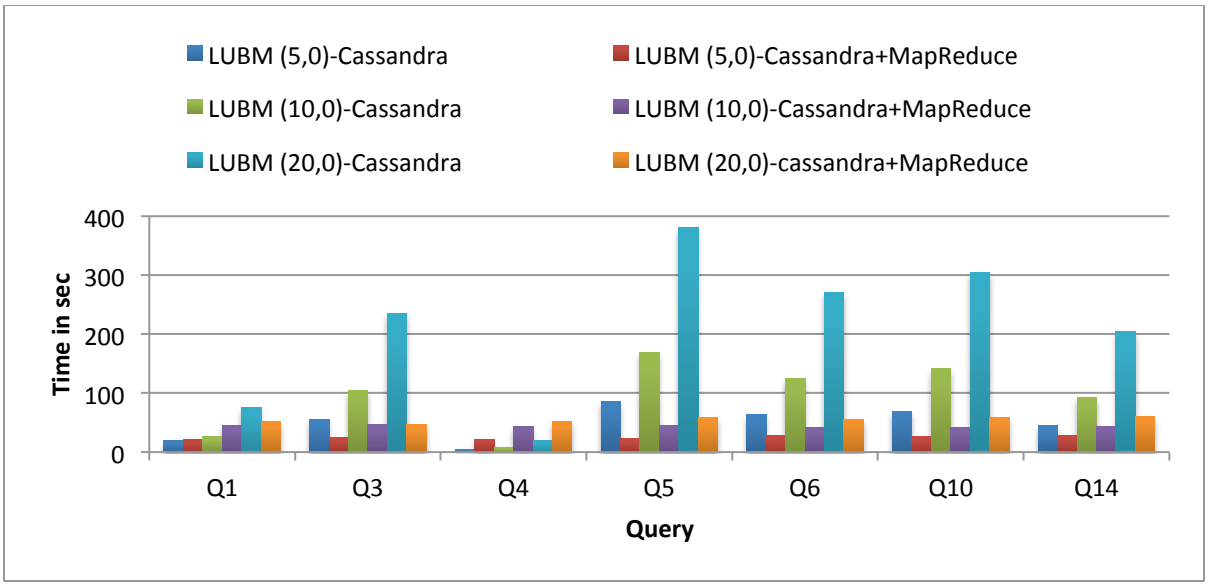


Figure 5.4 Query Processing Time

The results show that MapReduce API performs better for queries, which involve less reasoning operations and more data fetching from database. Cassandra API performs better for queries that need to perform complex reasoning and have small result set. Although both models show performance drop in distributed system compared to centralized system, Cassandra API integrated with MapReduce has a comparatively better performance than MapReduce API. The problems identified with distributed model are SPARQL query resolution is being done on only one machine and network I/O.

5.2 Accuracy Results

To test the accuracy of the system, we have compared our system results with the LUBM benchmark results. We got 100% accurate results for all the queries except query 5, which output a result set of 678 rows, but a result set of 719 rows was expected. Results are same when executed on single node and distributed nodes.

Table 5.10 Results Compared with LUBM Results

Query	Graph-store based SPARQL (5,0)	Graph-store based SPARQL (10,0)	Graph-store based SPARQL (20,0)	LUBM (5,0)	LUBM (10,0)	LUBM (20,0)
Q1	4	4	4	4	4	4
Q3	6	6	6	6	6	6
Q4	34	34	34	34	34	34
Q5	678	678	678	719	719	719
Q6	48582	99566	210603	48582	99566	210603
Q10	4	4	4	4	4	4
Q14	36682	75547	160120	36682	75547	160120

5.3 Comparing with In-memory based ARQ Engine

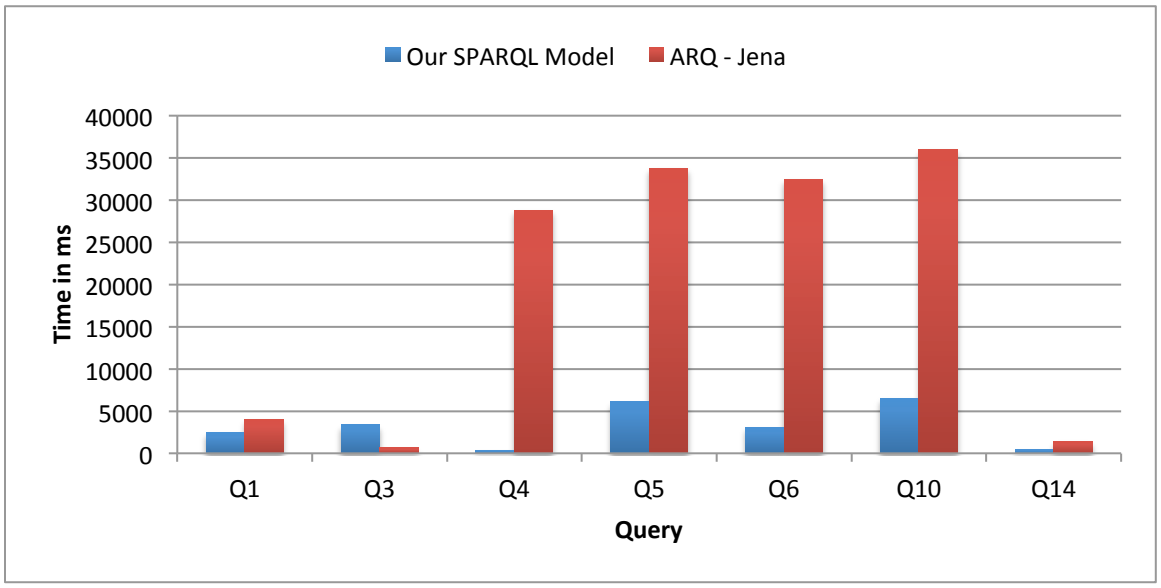


Figure 5.5 Query Processing Time Compared with In-Memory based ARQ Engine

We have also compared our SPARQL model with original ARQ engine provided by Jena. We used LUBM (1,0) dataset to evaluate ARQ engine versus Graph-store based SPARQL model. We observed that our approach works well compared to ARQ engine. Except query Q3, all the remaining queries performed well in our model. We have also conducted the memory test and observed that ARQ engine cannot handle more than 1.23 GB RDF data when tested on MacBook Pro having 4 GB RAM. Our model handled 3 GB of RDF data perfectly. Theoretically, it could handle more than 3GB, but due to the limited resources, we restrict ourselves to test with 3 GB data.

CHAPTER 6

CONCUSION AND FUTURE WORK

6.1 Conclusion

In this section, we will summarize the contribution of this thesis. We proposed a distributed data storage layout for storing large set of RDF triples, and it has worked well. We came up with idea of caching reasoning information separately in Graph-store. We successfully implemented Graph-store based SPARQL query system that can take huge RDF dataset as input, can perform some reasoning operations using Graph-store, and retrieve relevant data from Cassandra storage system.

We evaluated Cassandra and MapReduce based SPARQL query system along with Cassandra Hector based SPARQL system on centralized and distributed systems. Although, the results were still getting better on centralized system for most of the benchmark queries, we improved the query responsiveness by introducing massive parallelism concept with MapReduce. We also compared our model with in-memory based ARQ engine and concluded that our model performed very well compared to ARQ engine.

6.2 Future Work

There are some limitations in the project, but there is also scope for further enhancement. The current system does not support complex reasoning capabilities such as transitive properties over relationships. We will extend the system by introducing Full OWL properties. Also, the query resolution is being done on only one machine. We need to definitely work on the distributed SPARQL resolution. This might help to improve query response time in distributed systems. We have not really tested our system using big data

because of resources limitation. We will to use huge data set for future evaluations. We will use ARQ new versions such as ARQ SDB, and ARQ TDB for evaluation.

REFERENCES

1. The Sematic Web (2014). The Wikipedia. Available: http://en.wikipedia.org/wiki/Semantic_Web.
2. The Semantic Web (2013). The W3C. Available: http://www.w3.org/standards/semanticweb/.
3. The Linked Data (2014). The Linked Data Community. Available: http://linkeddata.org/.
4. The Resource Description Framework (RDF) (2014). The W3C. Available: http://www.w3.org/RDF/.
5. The Resource Description Framework (2014). The Wikipedia. Available: http://en.wikipedia.org/wiki/Resource_Description_Framework.
6. The RDF Query Languages (2014). The Wikipedia. Available: http://en.wikipedia.org/wiki/RDF_query_language.
7. The SPARQL (2014). The Wikipedia. Available: http://en.wikipedia.org/wiki/SPARQL
8. The SPARQL Query Language for RDF (2008). The W3C. Available: http://www.w3.org/TR/rdf-sparql-query/
9. The Foundation (2012). The apache software foundation. Available: http://www.apache.org/foundation/.
10. MapReduce. Hadoop (2014). The Apache Software Foundation. Available: http://hadoop.apache.org/.
11. Cassandra wiki (2013). The Apache Software Foundation. Available: http://wiki.apache.org/cassandra/.

12. Hadoop Support - Cassandra wiki (2014). The Apache Software Foundation. Available: http://wiki.apache.org/cassandra/HadoopSupport.
13. The ARQ Engine (2014). The Apache Jena. Available: http://jena.apache.org/documentation/query/
14. The Jena Framework (2011). The Apache Jena. Available: http://jena.apache.org/.
15. The Neo4j (2014). The Neo Technology, Inc. Available: http://www.neo4j.org/.
16. The Web Ontology Language (OWL). The W3C. Available: http://www.w3.org/2001/sw/wiki/OWL
17. Stefan Kokkelink. Roland Schwanzl (2002). "Expressing Qualified Dublin Core in RDF / XML". The Dublin Core Metadata Initiative. Available: http://dublincore.org/documents/dcq-rdf-xml/
18. McCarthy, Philip (2005). "Search RDF Data with SPARQL". The IBM Developer Works. Available: https://www.ibm.com/developerworks/library/j-sparql/.
19. Srinath, Perera (2012). "Consider the Apache Cassandra Database". The IBM Developer Works. Available: http://www.ibm.com/developerworks/library/os-apache-cassandra/
20. Thrift wiki (2013). The Apache Software Foundation. Available: http://wiki.apache.org/thrift/.
21. Charsyam. "Cassandra Data Model". Available: http://charsyam.wordpress.com/tag/cassandra-data-model/
22. Recardo Martin, Camarero. (2010). "Getting Started with Cassandra". Available: http://blogs.nologin.es/rickyepoderi/index.php?/archives/9-Getting-Started-With-Cassandra.html

23. Franke, Craig, et al. "Efficient Processing of Semantic Web Queries in HBase and MySQL Cluster." IT Professional 15.3 (2013): 36-43.
24. Guo, Yuanbo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems." Web Semantics: Science, Services and Agents on the World Wide Web 3.2 (2005): 158-182.
25. Kulkarni, Prasad. "Distributed SPARQL query engine using MapReduce."Master of Science, Computer Science, School of Informatics, University of Edinburgh (2010). Available: http://www.inf.ed.ac.uk/publications/thesis/online/IM100832.pdf.
26. Urbani, Jacopo, et al. "Scalable distributed reasoning using mapreduce." The Semantic Web-ISWC 2009. Springer Berlin Heidelberg, 2009. 634-649.
27. Chong, Eugene Inseok, et al. "An efficient SQL-based RDF querying scheme."Proceedings of the 31st international conference on Very large data bases. VLDB Endowment, 2005.
28. Mika, Peter, and Giovanni Tummarello. "Web semantics in the clouds." Intelligent Systems, IEEE 23.5 (2008): 82-87.
29. Hartig, Olaf, Christian Bizer, and Johann-Christoph Freytag. "Executing SPARQL queries over the web of linked data." The Semantic Web-ISWC 2009. Springer Berlin Heidelberg, 2009. 293-309.
30. The Jersey API for REST Web Services (2014). The Oracle Corporation. Available: https://jersey.java.net/
31. The Grizzly API for HTTP Server (2014). The Oracle Corporation. Available: https://grizzly.java.net/

32. Hector - Cassandra wiki (2014). The Apache Software Foundation. Available: http://wiki.apache.org/cassandra/Hector.

33. The Microsoft Azure Cloud (2014). The Microsoft Corporation. Available: http://azure.microsoft.com/en-us/

34. The DataStax Enterprise Edition (2013). The DataStax. Available: http://www.datastax.com/

35. Heart proposal (2010). The Apache Software Foundation. Available: http://wiki.apache.org/incubator/HeartProposal.

36. Auer, Sören, et al. "Dbpedia: A nucleus for a web of open data." The semantic web. Springer Berlin Heidelberg, 2007. 722-735.

37. Harth, Andreas, et al. "Yars2: A federated repository for querying graph structured data from the web." The Semantic Web. Springer Berlin Heidelberg, 2007. 211-224.

38. Ladwig, Günter, and Andreas Harth. "CumulusRDF: Linked data management on nested key-value stores." The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011). 2011.

39. Weaver, Jesse, and James A. Hendler. "Parallel materialization of the finite rdfs closure for hundreds of millions of triples." The Semantic Web-ISWC 2009. Springer Berlin Heidelberg, 2009. 682-697. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04930-9 43

40. Chang, Fay, et al. "Bigtable: A distributed storage system for structured data."ACM Transactions on Computer Systems (TOCS) 26.2 (2008): 4.

41. DeCandia, Giuseppe, et al. "Dynamo: amazon's highly available key-value store." ACM SIGOPS Operating Systems Review. Vol. 41. No. 6. ACM, 2007.

42. Horridge, Matthew, and Sean Bechhofer. "The owl api: A java api for owl ontologies." Semantic Web 2.1 (2011): 11-21.
43. Myung, Jaeseok, Jongheum Yeon, and Sang-goo Lee. "SPARQL basic graph pattern processing with iterative MapReduce." Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud. ACM, 2010.
44. Husain, Mohammad Farhan, et al. "Storage and retrieval of large rdf graph using hadoop and mapreduce." Cloud Computing. Springer Berlin Heidelberg, 2009. 680-686. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10665-1 72

VITA

Anudeep Perasani was born on July 02, 1991, in Khammam, Andhra Pradesh, India. He completed his schooling in Khammam and graduated high school in 2008. He then completed his Bachelor's degree in Computer Science & Engineering from Maturi Venkata Subba Rao Engineering College, Hyderabad, India in 2012. Upon the completion of his Bachelor's, he was placed in Cognizant Technology Solutions as a Software Engineer.

In August 2012, Mr. Anudeep Perasani came to United States to study Computer Science at the University of Missouri- Kansas City (UMKC), specializing in Software Engineering. Upon completion of his requirements for the Master's Program, Mr. Anudeep Perasani plans to work for Cerner, Kansas City.