Distributed SPARQL over Big RDF Data, A Comparative Analysis Using Presto and MapReduce

by

Mulugeta Mammo

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2014 by the
Graduate Supervisory Committee:

Srividya Bansal, Chair
Ajay Bansal
Timothy Lindquist

ARIZONA STATE UNIVERSITY

December 2014

ABSTRACT

The processing of large volumes of RDF data require an efficient storage and query processing engine that can scale well with the volume of data. The initial attempts to address this issue focused on optimizing native RDF stores as well as conventional relational databases management systems. But as the volume of RDF data grew to exponential proportions, the limitations of these systems became apparent and researchers began to focus on using big data analysis tools, most notably Hadoop, to process RDF data. Various studies and benchmarks that evaluate these tools for RDF data processing have been published. In the past two and half years, however, heavy users of big data systems, like Facebook, noted limitations with the query performance of these big data systems and began to develop new distributed query engines for big data that do not rely on map-reduce. Facebook's Presto is one such example.

This thesis deals with evaluating the performance of Presto in processing big RDF data against Apache Hive. A comparative analysis was also conducted against 4store, a native RDF store. To evaluate the performance Presto for big RDF data processing, a map-reduce program and a compiler, based on Flex and Bison, were implemented. The map-reduce program loads RDF data into HDFS while the compiler translates SPARQL queries into a subset of SQL that Presto (and Hive) can understand. The evaluation was done on four and eight node Linux clusters installed on Microsoft Windows Azure platform with RDF datasets of size 10, 20, and 30 million triples. The results of the experiment show that Presto has a much higher performance than Hive can be used to process big RDF data. The thesis also proposes an architecture based on Presto, Presto-RDF, that can be used to process big RDF data.

DEDICATION

This thesis is dedicated my parents for their love and support and to my brother who initiated the idea of coming to the US for my master’s study and who sponsored my education.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Srividya Bansal, for her guidance and encouragement during the course of the development of this thesis. My appreciation also goes to Prof. Timothy Lindquist for introducing me to Flex and Bison, which were used in this thesis to build the SPARQL to SQL compiler. Last, but not least, I would like to thank Dr. Ajay Bansal for his constant encouragement and positivity he showed me throughout my stay in ASU.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER 1

INTRODUCTION

1.1 Motivation

The Semantic Web is a technology and a set of standards for sharing data using a model called the Resource Description Framework (RDF). RDF enables the representation of data as a set of linked statements, each of which consists of a subject, predicate, and object called a triple. RDF datasets, consisting of millions of triples, form a network of directed graph (DG) and are stored in systems called triple-stores. A query language standard, SPARQL, has also been developed to query RDF datasets.

For the Semantic Web to work, both triple-stores and SPARQL query processing engines have to scale well with the size of data. This is especially true when the size of RDF data is too big such that it is difficult, if not impossible, for conventional triple-stores to work with. In the past few years, however, new advances have been made in the processing of large volumes of data sets, aka big data, which can be made to use for processing big RDF data. In this regard, various researches that study the use of big data technologies for RDF data processing have been published.

In the past two and half-years, new trends in big data technology have emerged that use distributed in-memory query processing engines based on SQL syntax. Some of these tools include: Facebook Presto, Apache Shark, and Cloudera Impala. These tools promise to deliver high performance query execution than traditional Hadoop system like Hive. It is the motivation of this thesis to validate this claim for big RDF data – i.e. if these new in-memory query processing models work well to deliver faster response times for SPARQL queries, which must be translated to SQL.

1.2 Problem Statement

The problem statement of this thesis can be summarized as:

> *Can we gain query execution performance, compared to Hive and 4store, by storing big RDF data in HDFS and using in-memory processing engines instead of MapReduce?*

Specifically, it addresses these questions:

- Is it feasible to store big RDF data in HDFS and get improved query execution time, compared to Hive and native RDF stores like 4store, by translating SPARQL queries into SQL and then using the Presto distributed SQL query processing engine to run the translated queries?
- How much improvement, in query response time, can be attained by using in-memory query processing engine, e.g. Presto, against native RDF stores, like 4store, and other query processing engines based on MapReduce, like Hive?
- How do different RDF storage schemes in HDFS affect the performance of SPARQL queries? And is it possible to construct an end-to-end distributed architecture to store and query RDF datasets?

Based on the above questions, the following hypothesis are put forward:

1. As the size of RDF data increases to big-data levels, RDF stores based on Hadoop outperform native RDF stores like 4store.
2. Distributed in-memory query processing engines deliver faster response time on big RDF datasets than query processing engines that rely on MapReduce.
3. Vertical partitioning scheme for RDF data gives better performance than other RDF storage schemes on Hive.
4. Increasing number of processing nodes dramatically improves query performance.

CHAPTER 2

THE SEMANTIC WEB

This thesis is based on two technologies – the semantic web and big data. This chapter presents a summarized review of the semantic web.

2.1 Introduction

The original paper on semantic web, written in 2001 by Tim–Berners Lee, defines the semantic web as a web of data that is an extension of the current web but one in which information has semantics or well–defined meaning [**1**]. The motivation behind the creation was to address the fundamental limitation of the current World Wide Web – which is a web of *pages* and not of *data*.

Consider an example of consulting a trips advisory website to choose a hotel for your summer vacation. You browse the hotels listed in the trips advisory website and choose one hotel. But when trying to book from the hotel's website you discover that they have closed their branch. This looks a simple outdated data issue on part of the trips advisory website, however in its essence it points to a major limitation in the current web architecture – data is linked at *presentation* level and not at a *representation* level [**2**].

In the current web architecture, the problem stated above can be solved in a variety of ways. The simplest solution can be making data available as lined data and updating it regularly. But if the linked data changes frequently, this solution would be very hard, if not impossible, to implement. The other option would be to back up both sites with the same relational database so that when the linked data is updated by one party, the other party gets it automatically. This approach would require that a trust relationship be established between the parties. Yet another option, which is popular, is to publish functions as web services so that parties who need the published functionalities would get needed data by consuming the web services.

Though both the relational database and web services approach can solve the data linking problem at an enterprise level, they are not feasible solutions in the context of the entire World Wide Web – which has an open and distributed data architecture that grows organically with contributions coming from different sources.

The semantic web attempts to solve the data linking problem by creating a web infrastructure, on top of the current web infrastructure, that would make it possible for one data item to point to another, using global references called Uniform Resource Identifiers (URIs), thereby creating a single, distributed web of data.

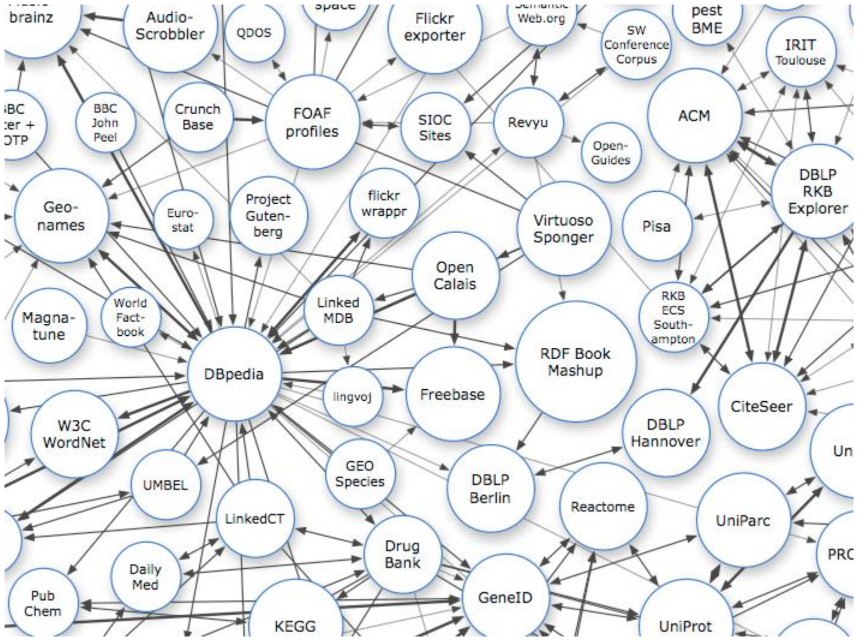

Figure 1 Semantic Web as a Web of Data [3]

In the Semantic web, a website would not just publish its data at the presentation level (i.e. for human consumption) but also a machine–readable description of the data. The Semantic web infrastructure provides a data model, called Resource Description Framework (RDF), that can be used to encode this machine–readable data. Semantic web comprises of web pages that implement this data model.

The ultimate aim of the Semantic web, according to Tim–Berners Lee, is to transform the current web of documents (pages) to web of data thereby bridging the divide that exists between the web, *which is unstructured and human–readable*, and databases, *that are well structured and machine–readable*.

2.2 Challenges of Implementing the Semantic Web

The Semantic web infrastructure is based on the existing web infrastructure and hence inherits its features. These features, however, become more problematic to handle because the entity that is being linked in the semantic web is not pages, but data [**4**]:

- Anyone can say anything about any topic assumption – in the current web context, this means that anyone can create and publish a *document* online which others can link to. In the context of the semantic web, however, it means that anyone can create a *data item* about any entity (called a resource in the Semantic web terminology) in a way that can be *integrated* with other data items from other sources.
- Open world assumption – the Semantic web must assume that new data items can be introduced to the distributed web of data at any time and no conclusions may be drawn based on the assumption that all data is available.

- Non–unique naming assumption – different names can refer to the same resource and the Semantic web has to assume, until otherwise told, that a resource can be referred to using different names by different sources.
- Network effect and synergy – the semantic web infrastructure must enable knowledge synergy possible – more people joining creates more value which attracts more people who would in turn add more values that would expand and increase knowledge sharing exponentially.

Because the Semantic web is a distributed web of data from different sources and its aim is to create a coherent web of data that is useful, the chaos that arises from non–unique naming, subtleties in the meaning of terms, etc. have to be resolved [**4**]. For example, the Semantic web infrastructure has to provide a mechanism to differentiate the word "apple" from two sources that talk about fruit and the multibillion-dollar company, Apple. The semantic web achieves this by providing a number of modeling languages that can be used to express meaning, semantics, at different levels of detail.

2.3 Semantic Modeling

The semantic web provides a number of modeling languages that can express meaning at different levels. These modeling languages are defined as standards by the W3C group and can be used to express, i.e. give semantics to, different kinds of data. The standard organizes these modeling languages as a stack of layers where a layer is dependent upon the layers under it. The expressiveness that can be achieved increases as one goes from lower to higher layers [**4**].

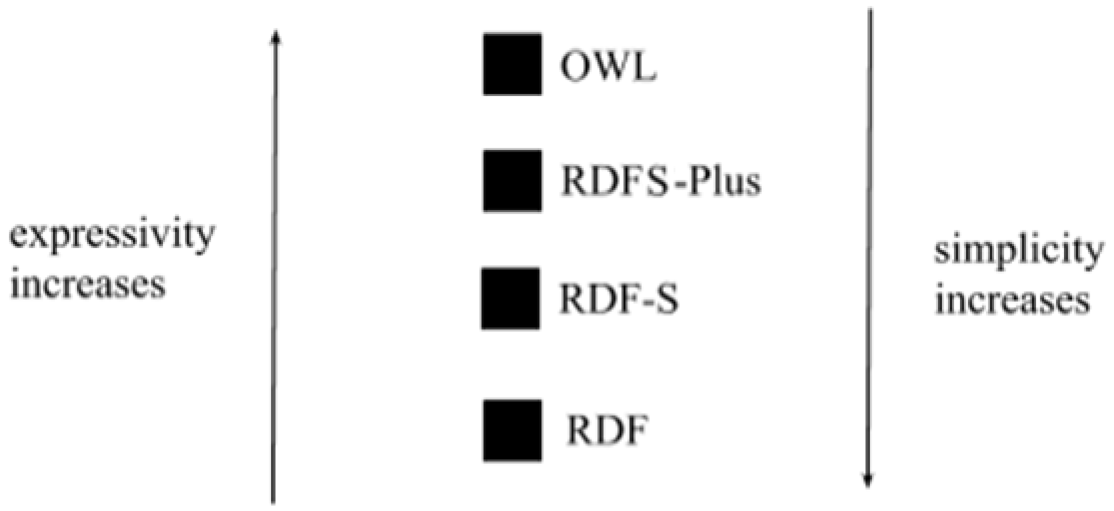

Figure 2 Semantic Web Modeling Languages

- RDF – the Resource Description Framework – is the bottom layer in the stack of semantic web modeling languages and is used to express a basic statement about a resource [**5**].
- RDFS – the RDF Schema language – is the next more expressive layer above the RDF layer. It can be used to express commonality and/or variability between resources. RDFS, aka RDF-S, can be likened to how classes and class hierarchies are organized in object-oriented programming languages [**4, 6**].
- RDFS–Plus – is the next top layer above RDF-S but below OWL – hence, it is more expressive than RDF-S, but less expressive and complex than OWL. Currently, there are no standards defined for RDFS–plus. RDFS-Plus can express the relationship and constraints that exist between properties of resources. Since it sits on top of the RDFS and RDF, it can express semantics that can be expressed by RDFS and RDF [**4**].

- OWL – the Web Ontology Language – is the top layer of the stack and can be used to express semantics that cannot be expressed by the other models. OWL is more complex and can be used to express detailed constraints between resources [**4, 7**].

2.4 Identifying Resources – the URI

A resource, in the semantic web technology, refers to a specific entity or thing that can be identified. The standard way to identify a resource in the web is to give it a URI – Uniform Resource Identifier. A URI is a string of characters with a well-defined format. The Internet standard STD 66 (also RFC 3986), defines the generic syntax for a URI as [**8**]:

<scheme name> : <hierarchical part> [? <query>] [# <fragment]

Where:

- Scheme name – is a sequence of characters beginning with a letter and followed by a combination of letters, digits, the plus character ('+'), period ('.'), or hyphen ('-'). Scheme names are case-insensitive but the standard is to write them in small letters.
- Hierarchal part – holds identification information and may or may not begin with double forward slashes. If it begins with double forward slashes, then authority and path parts follow, if not path follows.
    - Authority part – can hold a hostname, an optional port that is preceded by a colon (':') or a user information that is terminated with '@'.
    - Path – is a sequence of segments separated by a forward slash '/'. A path may begin with a forward slash but may not begin with two forward slashes.
- Query – is optional and holds additional information that is not hierarchical.
- Fragment – holds additional information that identifies a secondary resource. For example, a section in a document. Fragment, just like query, is optional.

A URI can be used to identify and name anything – from physical structures, like buildings, to abstract concepts like subclass. A URI must not contain any embedded spaces and is guaranteed to be unique.

A related term, URL (Uniform Resource Locator), is a subset of URI that, in addition to identifying a web resource, can be used to specify the means of locating the resource. It does this by specifying the resource's primary access mechanism.

URIs can be also expressed using **q**ualified **n**ames, or *qnames*. A *qname* has two parts: a namespace and an identifier that are separated with a colon [**9**]. For example the URI:

http://purl.org/dc/elements/1.1/creator

Can be expressed as the qname:

dc: creator

Where dc, the namespace, represents http://purl.org/dc/elements/1.1/. Namespaces in qnames, just like namespace in object-oriented programming languages, are used to group related identifiers together and avoid name collisions.

There are a number of namespaces that have been defined by the W3C and the Semantic Web standards group for use with web and Semantic Web technologies. For example, W3C defines xsd as a namespace for XML schema definitions. Likewise, the Semantic Web defines namespaces for its different layers [**10**]:

- rdf – is the namespace for identifiers used in RDF. The URI for the namespace is http://www.w3.org/1999/02/22-rdf-syntax-ns#
- rdfs – is the namespace for identifiers used in RDF schema, RDFS. The URI for the namespace is http://www.w3.org/2000/01/rdf-schema#
- owl – is the namespace used for owl, the Web Ontology Language..

2.5 RDF – The Resource Description Framework

The Resource Description Framework, RDF, is the basic construct of the Semantic Web that is used to make a statement about an entity (subject or resource). An RDF statement is composed of three parts: *subject*, *predicate* and *object* (aka *spo*). The subject denotes the entity the statement is made about while the predicate denotes some attribute or aspect of the entity [**5**]. The object is the value of the predicate for the subject. An RDF statement is also known as a triple.

For example, the statement:

Einstein was born in 1879.

Can be represented as an RDF statement with *spo*:

$$(s, p, o) = (\text{Einstein, wasBorn, 1879}).$$

A triple can also be represented as the labeled directed graph (DG):

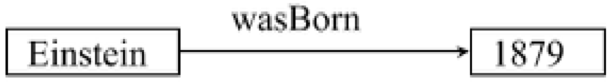

Figure 3 A Triple as a Labeled Directed Graph

Multiple statements can also be made about a subject and a subject may have multiple values for a predicate. Multiple RDF statements are called triples. Triples can be represented as a table of *subject-predicate-object* values:

Subject	**Predicate**	**Object**
Einstein	wasBorn	1879
Einstein	developed	Special Relativity
Einstein	developed	General Relativity

Einstein	diedIn	1955
Newton	wasBorn	1642
Galileo	died	1642
General Relativity	dealsWith	Gravity
Newton	discoveredLawsOf	Gravity

Table 1 A Table of Triples

Triples can also be represented using a network of labeled directed graphs (DG):

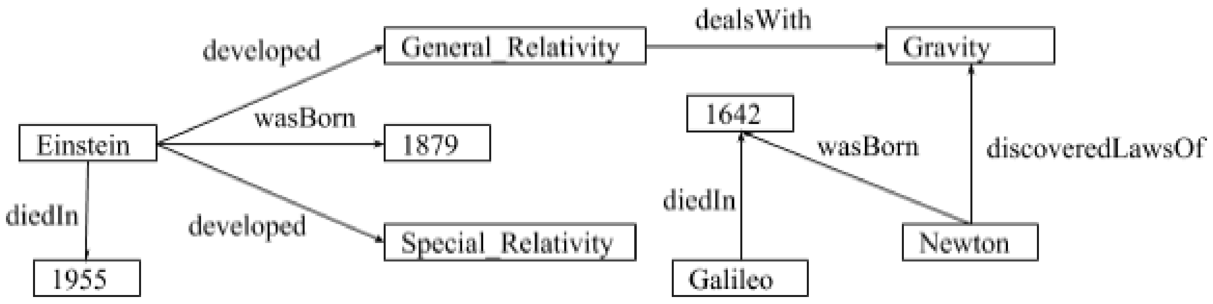

Figure 4 A Labeled Directed Graph of Triples

2.6 RDF Serialization Formats

RDF data can be serialized, i.e. stored in a file system, in different formats. The W3C defines four serialization formats for RDF data – RDF/XML, Turtle, N–Triples, and N3 [**11**]. There are also other serialization formats – *RDFa, microdata, rdf-json*, etc.

RDF/XML – serializes RDF data (graph) as an XML file where the nodes and edges of the RDF graph are represented using XML elements, attribute and text values. RDF/XML was the first RDF serialization format adopted by the W3C [**12**]. The RDF graph in Figure 4 can be represented in XML as:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
 xmlns:ns1="http://www.eg.com/ex/description#"
 xmlns:ns2="http://www.eg.com/ex/yr#"
 xmlns:ns3="http://www.eg.com/ex/work#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
<rdf:Description rdf:about="http://www.eg.com/ex/subject#General_Relativity">
  <ns1:dealsWith rdf:resource="http://www.eg.com/ex/subject#Gravity"/>
</rdf:Description>
<rdf:Description rdf:about="http://www.eg.com/ex/person/name#Einstein">
  <ns3:developed rdf:resource="http://www.eg.com/ex/subject#General_Relativity"/>
  <ns2:wasBorn>1879</ns2:wasBorn>
  <ns3:developed rdf:resource="http://www.eg.com/ex/subject#Special_Relativity"/>
  <ns2:diedIn>1955</ns2:diedIn>
</rdf:Description>
<rdf:Description rdf:about="http://www.eg.com/ex/person/name#Newton">
  <ns3:discoveredLawsOf rdf:resource="http://www.eg.com/ex/subject#Gravity"/>
  <ns2:wasBorn>1642</ns2:wasBorn>
</rdf:Description>
<rdf:Description rdf:about="http://www.eg.com/ex/person/name#Galileo">
  <ns2:diedIn>1642</ns2:diedIn>
</rdf:Description>
</rdf:RDF>
```

The main advantage of RDF/XML over the other serialization formats is that it can be used readily with system and programming environments that are based on XML. However, not all RDF triples can be represented in RDF/XML due to a limitation XML imposes on its syntax.

Turtle –Terse RDF Triple Language – is a serialization format created by Tim Berners–Lee to encode RDF graphs in a compact form that is also readable for humans [**13**]. Unlike the RDF/XML serialization format, the Turtle format can be used to encode any type of RDF graph. A turtle document consists of a list of directives and triples. The RDF graph in Figure 4, for example, can be represented in turtle format as:

```
@prefix ns0: <http://www.eg.com/ex/yr#> .
@prefix ns1: <http://www.eg.com/ex/work#> .
@prefix ns2: <http://www.eg.com/ex/description#> .
```

```
@prefix ns3: <http://www.eg.com/ex/person/name#> .
@prefix ns4: <http://www.eg.com/ex/subject#>

ns3:Einstein ns0:wasBorn “1879”^xsd:integer .
ns3:Einstein ns1:developed “Special_Relativity” .
ns3:Einstein ns1:developed “General_Relativity” .
ns3:Einstein ns0:diedIn “1955”^xsd:integer .
ns4:General_Relativity ns2:dealsWith ns4:Gravity .
ns3:Newton ns1:discoveredLawsOf ns4:Gravity .
ns3:Newton ns0:wasBorn “1642”^xsd:integer .
ns3:Galileo ns0:diedIn “1642”^xsd:integer .
```

Turtle is a subset of, and is compatible with, the N3 serialization format and has the MITME type text/turtle.

N–triples – Notation of Triples – is another RDF serialization format that is simpler than the turtle format but not as compact as the Turtle format. The RDF graph in Figure 4, for example, can be represented in N-triples as:

```
<http://www.eg.com/ex/person/name#Einstein> <http://www.eg.com/ex/yr#wasBorn>
“1879”^xsd:integer.
<http://www.eg.com/ex/person/name#Einstein> <http://www.eg.com/ex/work#developed>
<http://www.eg.com/ex/subject#Special_Relativity> .
<http://www.eg.com/ex/person/name#Einstein> <http://www.eg.com/ex/work#developed>
<http://www.eg.com/ex/subject#General_Relativity> .
<http://www.eg.com/ex/person/name#Einstein> <http://www.eg.com/ex/yr#diedIn>
“1955”^xsd:integer.
<http://www.eg.com/ex/person/name#Einstein> <http://www.eg.com/ex/yr#wasBorn>
“1879”^xsd:integer.
<http://www.eg.com/ex/subject#General_Relativity>
<http://www.eg.com/ex/description#dealsWith> <http://www.eg.com/ex/subject#Gravity>.
<http://www.eg.com/ex/person/name#Newton>
<http://www.eg.com/ex/work#discoveredLawsOf> <http://www.eg.com/ex/subject#Gravity>.
<http://www.eg.com/ex/person/name#Newton> <http://www.eg.com/ex/yr#wasBorn>
“1642”^xsd:integer.
<http://www.eg.com/ex/person/name#Galileo> <http://www.eg.com/ex/yr#diedIn>
“1642”^xsd:integer
```

N–triples has a text/turtle MIME type.

N–3 – Notation–3 – is another serialization format for RDF. N-3 is a superset of the Turtle format and has been designed with human readability in mind. The RDF graph in Figure 4 can be represented in N-3 as:

```
@prefix ns1: <http://www.eg.com/ex/description#> .
@prefix ns2: <http://www.eg.com/ex/yr#> .
@prefix ns3: <http://www.eg.com/ex/work#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xhv: <http://www.w3.org/1999/xhtml/vocab#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://www.eg.com/ex/person/name#Einstein> a rdfs:Resource ;
  ns3:developed <http://www.eg.com/ex/subject#General_Relativity>,
     <http://www.eg.com/ex/subject#Special_Relativity> ;
  ns2:diedIn “1955”^xsd:integer ;
  ns2:wasBorn “1879”^xsd:integer .

<http://www.eg.com/ex/person/name#Galileo> a rdfs:Resource ;
  ns2:diedIn “1642”^xsd:integer .

<http://www.eg.com/ex/person/name#Newton> a rdfs:Resource ;
  ns3:discoveredLawsOf <http://www.eg.com/ex/subject#Gravity> ;
  ns2:wasBorn “1642”^xsd:integer .

<http://www.eg.com/ex/subject#General_Relativity> a rdfs:Resource ;
  ns1:dealsWith <http://www.eg.com/ex/subject#Gravity> .
```

RDFa – Resource Description Framework in Attributes – is a W3C is recommendation that adds meta-data information to HTML (or XHTML) documents by extending the attributes of elements. The RDF graph from Figure 4 can be represented using RDFa as:

```
<div xmlns="http://www.w3.org/1999/xhtml"
 prefix="
  ns3: http://www.eg.com/ex/work#
  ns1: http://www.eg.com/ex/description#
  ns2: http://www.eg.com/ex/yr#
  rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
  rdfs: http://www.w3.org/2000/01/rdf-schema#"
```

```
>
 <div typeof="rdfs:Resource" about="http://www.eg.com/ex/person/name#Newton">
   <div rel="ns3:discoveredLawsOf" resource="http://www.eg.com/ex/subject#Gravity"></div>
   <div property="ns2:wasBorn" content="1642"^xsd:integer></div>
 </div>
 <div typeof="rdfs:Resource" about="http://www.eg.com/ex/person/name#Einstein">
   <div property="ns2:wasBorn" content="1879"^xsd:integer></div>
   <div property="ns2:diedIn" content="1955"^xsd:integer></div>
   <div rel="ns3:developed">
     <div typeof="rdfs:Resource" about="http://www.eg.com/ex/subject#General_Relativity">
       <div rel="ns1:dealsWith" resource="http://www.eg.com/ex/subject#Gravity"></div>
     </div>
   </div>
   <div rel="ns3:developed"
resource="http://www.eg.com/ex/subject#Special_Relativity"></div>
 </div>
 <div typeof="rdfs:Resource" about="http://www.eg.com/ex/person/name#Galileo">
   <div property="ns2:diedIn" content="1642"^xsd:integer></div>
 </div>
</div>
```

2.7 RDF-Schema

RDF–Schema (RDF-S) is a data modeling vocabulary for RDF data. According to the W3C, "RDF–S provides mechanisms for specifying groups of related resources and the relationship between these resources".

In RDF-S, resources are grouped into classes. A resource that belongs to a class is called an instance of the class. In many respects, RDF-S classes are similar to classes found in object-oriented programming languages like Java and C++. However, unlike classes in object-oriented programming languages where a class is defined in terms of the attributes of its instances, RDF-S defines properties in terms of the classes to which they belong (apply). For example, in Java, one can define a class called Person with an attribute (property) Age whose type int. In RDF-S, same idea can be expressed by defining a property Age with *domain* Person and *range* integer [**12**].

2.8 OWL – the Web Ontology Language

OWL is a W3C standard designed for use by web applications that need to define and process semantic data. OWL provides a much richer set of vocabularies than RDF–S and can be used to express and relate complex knowledge about things and their relationships. OWL has three sub-languages – *OWL Lite, OWL DL,* and *OWL Full* [**4**].

OWL Lite – is a smaller subset of OWL that can be used to express classification hierarchy of resources and simple constraints. OWL Lite provides a set of vocabularies that are classified into seven groups based on their purpose:

- RDF-S features – *Class*, *rdfs:subClassOf*, *rdf:Property*, *rdfs:subPropertyOf*, *rdfs:domain*, *rdfs:range*, *Individual*.
- Equality and inequality – *equivalentClass*, *equivalentProperty, sameAs, differentFrom, AllDifferent, distinctMembers*.
- Property characterstics – *ObjectProperty*, *DatatypeProperty*, *inverseOf, TransitiveProperty*, *SymmetricProperty*, *FunctionalProperty*, *InverseFunctionalProperty*.
- Property restrictions – *Restriction*, *onProperty*, *allValuesFrom*, *someValuesFrom*.
- Restricted Cardinality – *minCardinality*, *maxCardinality*, *cardinality*
- Header Information – *ontology, imports*
- Class Intersection – *intersectionOf*
- Datatypes – *xsd:datatypes*
- Annotation properties – *rdfs:label, rdfs:comment, rdfs:seeAlso, rdfs:isDefinedBy, AnnotationProperty, OntologyProperty*

OWL Lite guarantees computational completeness and decidability. Computational completeness means that all conclusions that can be drawn are computable while decidability means computations can finish in finite time.

OWL DL – OWL Description Logics – is a subset of OWL that is more expressive than OWL Lite but less expressive and complex than OWL Full. OWL DL also guarantees computational completeness and decidability.

OWL Full – is a super set of OWL data and provides maximum expressiveness. However, unlike OWL DL and OWL Lite, it *cannot* guarantee computational completeness.

2.9 SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is a query language defined by the W3C to query RDF data. The “Protocol” in the SPARQL deals with how a client program and a SPARQL processing engine interact to exchange queries and results from queries. W3C specifies the rules of these interactions. SPARQL has been design to query any RDF data regardless of its storage format – RDF/XML, Turtle, N–Triples, N–3, JSON, etc. SPARQL queries can also be run against a combination of these different storage formats.

Basic SPARQL Query – The most basic SPARQL query consists of a SELECT clause that identifies *variables* that appear in the result set and a WHERE clause which provides a graph pattern to match against the RDF graph the query is being run on.

For example, given the following RDF data set from Figure 3:

```
@prefix ns0: <http://www.eg.com/ex/yr#> .
@prefix ns1: <http://www.eg.com/ex/work#> .
@prefix ns2: <http://www.eg.com/ex/description#> .
@prefix ns3: <http://www.eg.com/ex/person/name#> .
@prefix ns4: <http://www.eg.com/ex/subject#>
```

```
ns3:Einstein ns0:wasBorn “1879”^xsd:integer .
ns3:Einstein ns1:developed “Special Relativity” .
ns3:Einstein ns1:developed “General Relativity” .
ns3:Einstein ns0:diedIn “1955”^xsd:integer .
ns4:General_Relativity ns2:dealsWith ns4:Gravity .
ns3:Newton ns1:discoveredLawsOf ns4:Gravity .
ns3:Newton ns0:wasBorn “1642”^xsd:integer .
ns3:Galileo ns0:diedIn “1642”^xsd:integer .
```

A SPARQL query that returns the theories Einstein developed can be written as:

```
SELECT ?theory
WHERE
{
<http://www.eg.com/ex/person/name#Einstein>
<http://www.eg.com/ex/work#developed> ?theory.
}
```

Returning:

“Special Relativity”
“General Relativity”

Same query can also be expressed using prefixes:

```
@prefix ns1: <http://www.eg.com/ex/work#> .
@prefix ns3: <http://www.eg.com/ex/person/name#> .

SELECT ?theory
WHERE
{
ns3:Einstein ns1:developed ?theory.
}
```

Multiple matches – multiple graph pattern conditions can be stated in a SPARQL query by adding more triples and variables.

```
@prefix ns0: <http://www.eg.com/ex/yr#> .

SELECT ?x ?z
WHERE
```

```
{
?x ns0:wasBorn ?y.
?x ns0:diedIn ?z.
}
```

Running the above query on the dataset from Figure 3, gives the result:

```
“Einstein”    1955
```

The first pattern, ?x ns:wasBorn ?y, selects:

```
“Einstein”    1879
“Newton”      1642
```

While the second pattern, ?x ns0:diedIn ?z, selects:

```
“Einstein”    1955
“Galileo”     1642
```

Because the first pattern and the second pattern are linked by their subject, ?x, and the variable ?z relates to the ns0:diedIn predicate, the returned result is:

```
“Einstein”    1955
```

Matching RDF Literals – SPARQL also supports querying based on string, integer and other types of literals.

RDF Constraints – SPARQL allows one to specify constraints that filter bindings of variables to RDF terms.

Filtering string values – the SPARQL FILTER function, regex, can be applied on string literals to do regular expression pattern matching:

```
@prefix ns1: <http://www.eg.com/ex/work#> .
SELECT  ?y
WHERE   {
   ?x ns1:developed ?y
   FILTER regex(?y, "Special%")
}
```

Returns the result “Special Relativity”.

Filtering numeric values – SPARQL also allows filters to be applied to queries:

```
@prefix ns0: <http://www.eg.com/ex/yr#> .
SELECT ?x
WHERE
{
   ?x ns0:wasBorn ?y.
   FILTER (?y < 1800)
}
```

Returns the result “Newton”

SPARQL data types – SPARQL also supports integer, floating-point, string, boolean and dateTime literals.

Blank Nodes – are denoted by _:someLabel, can appear in an RDF data. When a blank node appears in RDF data, it indicates either a lack of value – if it appears in the object position of a triple – or an unknown resource – if it appears in the subject position. When a blank node is used in SPARQL query, however, it is treated just like a variable. For example, the two queries below have identical semantics:

```
SELECT ?a ?b
WHERE {
?a :predicate _:blankNode .
_:blankNode :otherPredicate ?b .
}
```

```
SELECT ?a ?b
WHERE {
?a :predicate ?variable .
?c :otherPredicate ?b .
}
```

CHAPTER 3

BIG DATA

The other core technology, besides the Semantic Web, that this thesis relies on is big data. This chapter reviews the latest information and articles written about big data. It also presents one of its very important and popular open–source application framework – Apache Hadoop.

3.1 Definition

There are a number of definitions that have been suggested for big data. IBM, for example, defines big data as a term that describes quintillion bytes of data aggregated from various sources in different file formats and that which grows rapidly [**15**]. Likewise, in an article published by Microsoft in 2013, big data is a term that is used to describe the process of applying serious computing power to seriously massive and often highly complex sets of information [**16**]. Many technologists also have proposed various definitions of big data. John Worthington from Tech Republic, for example, defines big data as a large amount of data that is moving at a rapid pace and upon its valuable analysis a company's existence is based on [**17**]. On the other hand, Edd Dumbill from O'Reilly Strata, defines big data as data that exceeds the processing power of conventional database systems and that which is too big and too fast to fit into these systems [**18**].

Though each of the definitions above emphasizes one or another aspect of big data, all definitions agree that big data is big in size, is unstructured, holds lots of different data formats (text, video, sensor data, financial transactions, etc.) and is very difficult to process using conventional database systems.

This thesis assumes the definition of big data as proposed by Bernard Marr in 2013. Bernard Marr defines big data as data that is characterized by four parameters [**19**]:

1. Volume – big data holds vast amounts of data, usually in TB and PB – i.e. big data is big.
2. Velocity – big data is generated rapidly and moves around fast. For example social media messages that are created rapidly and going viral in seconds.
3. Variety – big data holds increasingly different types of data from various sources – social media feeds, financial data, photos, sensor data and so forth.
4. Veracity – big data is incredibly messy and unstructured and may involve inconsistencies.

From the perspective of this thesis, big RDF data refers to vast amounts of RDF triples with hundreds of millions of interconnections.

3.2 Apache Hadoop

The Apache Hadoop framework is an open–source software framework for storing and processing large data sets over a cluster of commodity machines. Hadoop is designed to scale horizontally and can run on few to thousands of machines. Each machine in a Hadoop cluster can be used both to store and compute data. Hadoop is designed to be fault–tolerant and can detect and handle failures at an application level gracefully – there by delivering a high–availability service [**20**].

The Apache Hadoop is composed of four modules – Hadoop Common, Hadoop Distributed File System (HDFS), Hadoop YARN, and Hadoop MapReduce.

- Hadoop Common – is the module that is responsible for providing a set of utility (foundation) classes used by the other modules.
- Hadoop Distributed File System (HDFS) – is a distributed file system that is designed to store big data over a cluster of commodity machines.

- Hadoop YARN – Hadoop Yet another Resource Negotiator – is framework that is used for job scheduling and resource management over a cluster.
- Hadoop MapReduce – is an open–source implementation of the MapReduce programming model.

The Apache Hadoop project consists of other Hadoop-related projects: *Ambari*, *Avro*, *Cassandra*, *Chukwa*, *HBase*, *Hive*, *Mahout*, *Pig*, *Spark*, *Tez*, and *Zookeeper*. Apache Hadoop can therefore be regarded as an ecosystem of frameworks and utilities designed to handle use cases that relate to the storage and processing of big data. This thesis makes use of the HDFS, MapReduce and Hive parts of this ecosystem [**20**].

3.3 HDFS – The Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is a distributed file system that is designed to store big data over a cluster of commodity machines. HDFS was originally developed by Doug Cutting and has its origin in the Apache Nutch project – an open–source web search engine project [**21, 22**].

HDFS has been designed to cope with hardware failures and has an error-detection and automatic recovery mechanism when faults occur. HDFS also provides high aggregate data bandwidth and can scale horizontally to thousands of nodes. Files in HDFS are created are written once but can be read many times. This model, also known as the write–once–read–many model, guarantees data coherency and enables a high throughput data access. High throughput is also achieved by localizing data and computation together so that network congestion is minimized. HDFS, as a distributed file system for big data, emphasizes high throughput of data access rather than low latency.

HDFS uses a master/slave architecture that consists of one master node, the NameNode, and multiple slave nodes. The NameNode is responsible for managing the file system and regulate file access by clients. Each node in the HDFS cluster, including the NameNode, has DataNodes that manages its storage. The data nodes are also responsible for providing file read and write requests from client applications, performing data creation, replication and deletion upon a request from the NameNode.

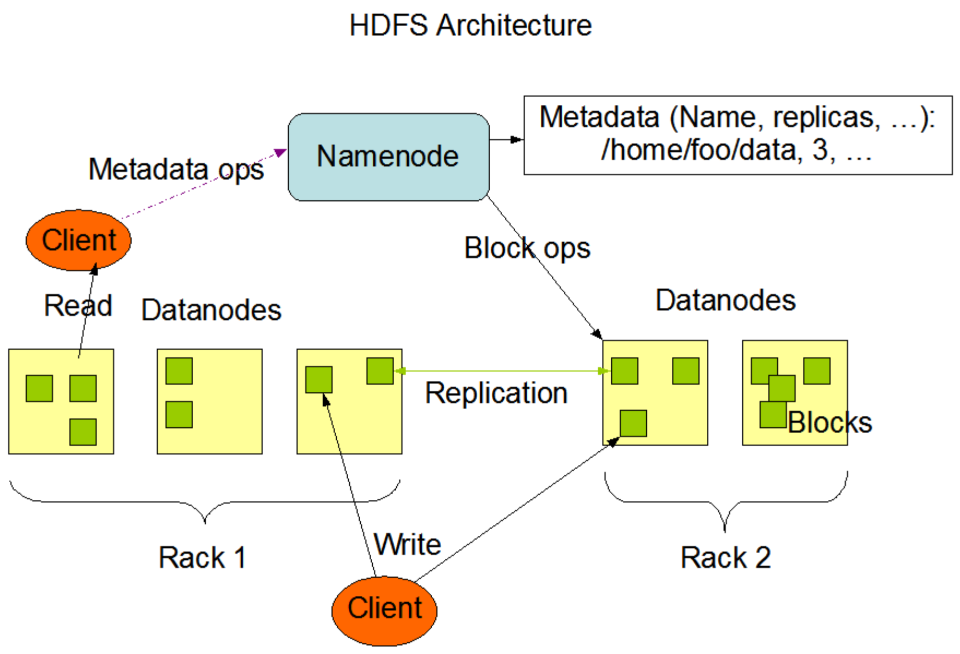

Figure 5 HDFS Architecture [21]

HDFS supports a hierarchical file organization similar to POSIX systems where the root directory is represented by "/". Applications can create, remove, and rename files and directories inside this root directory. Unlike POSIX systems, however, HDFS does not support user quotas, hard links as well as soft links. Any changes to the file system, including its properties, are recorded and managed by the NameNode.

A file in HDFS is stored as a sequence of equally sized blocks where the size of each block is defined by applications. HDFS ensures fault–tolerance and high–reliability by replicating these data blocks across the cluster. Applications can also specify the number of replicas of a file, at file creation time or later. The NameNode makes all decisions regarding the replication of blocks including which data nodes will be used to store replicas. Replicas are essential in HDFS to guarantee fault–tolerance and performance. HDFS has a replica placement policy that is “rack aware”. This policy is based on the experience that network bandwidth between machines in the same rack is greater than between machines on different racks. Hence, when HDFS receives a read request, it tries to answer the request by consulting the closer replica to the reader. If the replica exists in the same rack as the reader then that replica is chosen. HDFS uses the TCP/IP protocol for communication [**21, 22**].

Data upload in HDFS is accomplished in a sequence of steps. When an application wants to create a file, it sends a file creation request to the HDFS client. Upon receiving the request, the HDFS client creates a temporary local file to which the application writes data. When the size of the temporary local file reaches the block size, typically 64 MB, the HDFS client contacts the NameNode. When the NameNode receives the request, it sends the destination DataNode and block back to the client. The client would then flush the data from the temporary file to the destination DataNode and block and notifies and notifies the NameNode when the transfer is complete. Upon receiving a completion message, the NameNode commits the file creation and persistently stores the data.

3.4 Hadoop MapReduce

Hadoop MapReduce is Apache’s implementation of the map-reduce programming model that was popularized by Google in 2004. The Hadoop MapReduce framework can be used to write client applications that process large amounts of data (usually in hundreds of Giga and Tera Byes) in parallel, over a cluster of machines. HDFS and MapReduce can be considered as the two most important components of the Hadoop ecosystem – HDFS is responsible for handling the storage and management of data while MapReduce is responsible for computations over the data.

A computation is much more efficient when it is executed near the data it operates on [**22**]. This is particularly important when dealing with big data because it minimizes network congestion that would otherwise have been overloaded by transferring vast amount of data back and forth. Hence, moving computation is much cheaper than moving data and for this reason HDFS and MapReduce are typically configured to run on the same node. HDFS also provides interfaces to applications to move themselves closer to where the data resides.

The MapReduce programming can be defined as a programming model and an associated implementation for processing and generating large data sets using a parallel, distributed algorithm over a cluster of commodity hardware. MapReduce is inspired by the map and reduce primitives that exist in functional programming languages like Lisp, Haskell, and others.

A map function takes in a <key, value> input pair and generates a set of intermediate <key, values> pairs. The intermediate <key, value> pairs are then be taken by the MapReduce library, which groups and sorts together all intermediate key values associated with the same intermediate key, in a process called sort-and-shuffle, and passes it to the reducer function. The reducer function, upon receiving an intermediate key and a list of values for the key, merges the list of values together to output a possibly smaller set of values – typically just zero or one output value.

Both the map and reduce functions are written by the user. User programs can also specify a comparison function that would be used in the sort–and–shuffle step to compare and sort intermediate keys.

map: $<key_1, value_1> \rightarrow$ list $(<key_2, value_2>)$

reduce: $<key_2,$ list $(value_{2)}> \rightarrow$ list $(value_3)$

The creators of the MapReduce programming model, Jeffery Dean and Sanjay Ghemawat [**23**] argue that many real world tasks can be expressed using this simple model and that the model can be easily used to parallelize large computation over a cluster of machines.

During the execution of a MapReduce program each machine holds a copy of the program and the map and reduce functions are executed in parallel by the different machines across the cluster. MapReduce uses a hash-based algorithm to splits its input data.

According to Jeffery Dean and Sanjay Ghemawat, the execution of MapReduce programs follows the following sequence of steps [**23**]:

1. User's MapReduce program is loaded into memory.
2. The input files are split into M blocks, each with size 16 – 64 MB size. The master node, aka the NameNode, distributes these blocks across the cluster.
3. The master starts copies of the program across the cluster.
4. The master picks up idle workers and assigns a map task or a reduce task.
5. A slave/worker node that is assigned a map task reads the contents of its assigned input split, parses the <key, value> pairs out of the input data, and calls the map function on each <key, value> pair. The set of intermediate key value pairs produced by the map function are buffered in the worker's memory.
6. The buffered pairs are written periodically to local disk, which is partitioned into regions by a partitioning function. Upon completion of the write operations, the worker node notifies back the master of these locations which updates its file system namespace and would then notifies a reducer of these locations.

7. When a reducer receives the location of the intermediate/key value pairs produced by map workers, it does a remote procedure call to these map workers, reads the intermediate data, and sorts it by the intermediate keys.
8. For each unique intermediate key in the sorted set, the reducer calls the reduce function and appends the output of the reduce function to a final output file.
9. When all the map and reduce tasks are complete control is returned back to the user code.

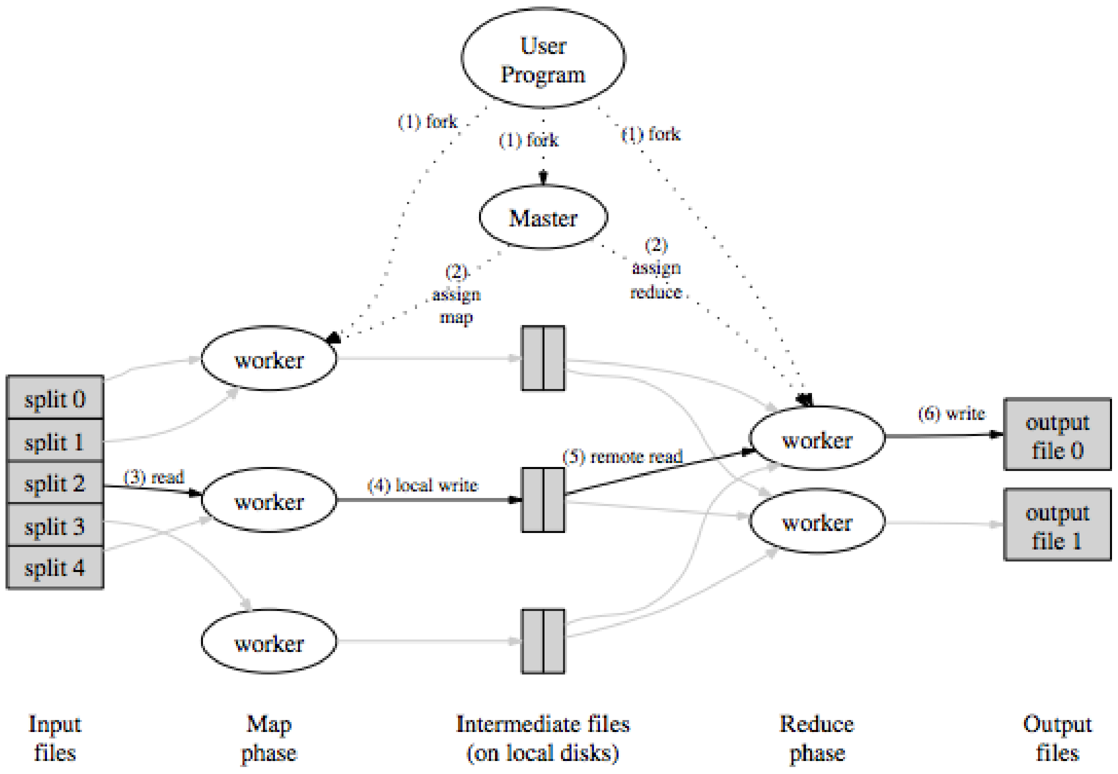

Figure 6 MapReduce Execution Steps [23]

The MapReduce model is designed to run on a cluster of hundreds and thousands of worker nodes controlled by a single master node. The master node maintains a data structure that holds the state of each map and reduce task – which could be in idle, in–progress or completed state. The MapReduce model also has an inherent support to gracefully handle worker node crashes.

The master nodes periodically listens for a workers’ heartbeat and if no response is received, the master node marks the worker node as dead and all map and reduce tasks that are in progress in the assigned node are set to idle states and become candidates for task re–scheduling. The model can also handle a master node crash by periodically writing checkpoints so that when it does fail a new copy can be started from the most recent check–pointed stated.

CHAPTER 4

RDF STORES

A number of articles and conference presentations that propose and evaluate different RDF storage managers have been published during the past five years. This chapter presents a review of some these papers that are relevant to this thesis.

4.1 Introduction

RDF stores, also known as triple stores, are data management systems that are used to store and query RDF data. RDF stores also provide an interface, called a SPARQL end–point, which can be used to submit SPARQL queries. Some triple stores, like Sesame, also provide APIs that can programmers can use to submit SPARQL queries and get results.

RDF storage managers can be broadly classified into three categories:

1. Native triple stores – are stores that are built from scratch to store RDF triples. Native triple stores make a direct use of the RDF data model – a labeled directed graph – to store and access RDF data. These triple stores usually store RDF data in a custom binary format. Examples are 4store [**25**], Jena TDB [**26**], and Sesame [**27**].
2. Relational–backed triple stores – are triple stores that are built on top of traditional RDBMs. Because RDBMs have a number of well-advanced features that have developed over the years, triple stores based RDBMs benefit from these features. Examples are 3store [**28**] and Jena SDB [**26**].
3. NoSQL triple stores – are triple stores based on NoSQL systems and by far are the largest triple stores in number. This group includes triples stores based on NoSQL systems as well as systems based on the Hadoop ecosystem. Examples include: Hive+HBase [**29**], CumulusRDF, Couchbase, and many more. Because NoSQL systems include a wide

range of systems, including graph databases, some researches have classified AllegroGraph [**30**], a graph database, both as a native as well a NoSQL store.

Despite the large number of RDF triple stores that have been developed and proposed by researchers, very few attempts have been made to systematically parameterize triple stores based on specific implementation parameters. The next section presents a review of three such papers.

4.2 Parameterized Classification of RDF Stores

Kiyoshi Nitaa and Iztok Savnik [**32**] proposed a parameterized view of RDF stores that is based on “single” and “multi–process” attribute sets. In this view, an RDF Store Manager, RSM, can be parameterized as function of single–process, S, and multi–process attributes, M:

RSM = f (S, M) = (<Ts, Is, Qs, Js, Cs, Ds, Fs>, <Dm, Qm, Sm, Am>)

The single–process attributes, S, this paper identifies are:

- Ts – Is the type of triple table structure the store uses – vertical, property-table, or horizontal.
- Is – is the index structure type – 6–independent, GSPO–OGPS, O matrix.
- Qs – Indicates whether a SPARQL end–point is implemented.
- Ss – Indicates the translation method type of IRI and literal strings – URI, literal, long, or none.
- Js – the join optimization method – RDBMS–based, column–store based, conventional ordering, pruning, or none.
- Cs – the cache type used – materialized–path–index, reified–statement, or none.
- Ds – whether the store relies on traditional RDBMS or uses its own custom database engine, and
- Fs – the type of inference type it supports – TBOX, ABox or none.

While multi–process attributes, M, are:

- Dm – data distribution method type – hash, data–source or none.
- Qm – query process distribution method type – data-parallel, data–replication, or none.
- Sm – stream process type – pipeline or none.
- Am – resource-sharing architecture – memory, disk, or nothing.

Based on the above parameterized classification, 4store has the characteristics: Ts = vertical, Qs = SPARQL, Ss = string id, Js = conventional ordering, Ds = RDB, Dm = hash, Qm = data parallel, Am = nothing, while Hadoop/HBase has Ts = horizontal, Ds = custom, Qm = data parallel, Am = memory.

RDF triple stores, specifically relational–backed triple stores, can also be characterized based on how they shred RDF data, triples, into relational tables. Y. Theoharis, V. Christophides, and G. Karvounarakis, in their 2005 paper [**32**] identify three shredding mechanisms: *schema–oblivious*, *schema–aware* and *hybrid*. In the schema–oblivious shredding, a single table is used to store RDF data as rows of triples of the form subject-predicate-object. In a schema-aware shredding, each RDF property or class is represented as a table. In hybrid shredding, on the other hand, one table is created per an RDF meta-class by distinguishing the range type of properties.

Some researchers [**33**] have also characterized and classified RDF stores as *in–memory*, *native*, and *non–native-non–memory* stores based on the implementation architecture they use. According to these researchers, in–memory RDF stores are useful to benchmark inference and reasoning operations by loading data from remote sites and storing in memory but are not useful to process large sets of triples. The native stores provide persistent storage with own database implementation that is modelled after the RDF model while the non–native-non–memory stores are setup to run on top of traditional databases like MySQL, PostsgreSQL, Oracle, etc.

4.3 Native RDF Stores

Native RDF stores provide a direct implementation of the RDF data model described in chapter 2 where the subject and object of a triple are represented as nodes, and the predicate as a labeled edge from the subject to the object – resulting in a network of labeled direct graph (DG). RDF data that has millions of triples and association among the triples would end up as a directed graph with hundreds of millions of interconnections. 4store and Jena TDB are two examples of native triple stores:

- 4store – is a native RDF store implemented in C and that, according to its designers, was designed to handle loads of 1 billion triples and above. 4store can run on a single machine or in a cluster of 32 machines (nodes). Garlik, the company behind 4store, reported that they were able to load and process more than 15 billion triples[1]. 4store, like other native stores, has a SPARQL end–point [**25**].
- Jena TDB – is another native RDF store from the Apache Jena project. Jena TDB comes with a full set of APIs that are used to store and query RDF data. Jena TDB has a SPARQL end–point and supports a number of extensions – e.g. property functions, aggregates, and arbitrary length property paths. Jena TDB is implemented in Java and employs a memory mapped I/O and a custom implementation of B+ Trees. The major limitation of Jena TDB is that it runs only on a single machine. However, it was reported that TDB was able to process 1.7 billion triples on a single machine with 64-bit hardware within 36 hours – i.e. 12k triples/second [**26**].

[1] In comparison, DBpedia 580 million triples (for English edition of Wikipedia)

4.4 Relational–based RDF Stores

Relational RDF stores use RDBMs to store triples. This section presents two such RDF stores – Apache Jena SDB and Sesame. The main characteristics that distinguish relational–backed RDF stores from native RDF is the presence of a mapping layer that maps RDF graphs into relational tables and SPARQL queries into SQL queries.

Apache Jena SDB is designed to run on top of a number of relational database management systems – MySQL, PostgeSQL, Oracle, BerkeleyDB, to name a few. Jena SDB, just like Jena TDB, was designed to run on a single machine. Application store RDF triples in Jena SDB using a JDBC connector. Jena SDB also requires that only single Java Virtual Machine should be involved in the triple writing, else data corruption may result. Jena SDB supports a number of RDF parsers and I/O modules for N3, N–triples and XML/RDF serialization formats [**26**].

Jena SDB also supports a simple triple store as well as three different kinds of property tables – a single–valued property table, a multi–valued property table and a property–class table. Each of these will be reviewed in detail in the next chapter.

4.5 NoSQL RDF Stores

NoSQL, Not Only SQL, stores model their data storage and retrieval in ways other than the traditional relational database management systems. NoSQL database are characterized by being non–relational, distributed, open–source, schema–free, horizontally scalable and able to handle huge amount of data [**34**]. NoSQL databases do not support relational ACID properties – Atomicity, Consistency, Isolation, and Durability. Instead, they support the BASE guarantee – Basic Availability, Soft–state and Eventual consistency.

The BASE guarantee is based on the CAP theorem formulated in 2001 by Eric Brewer. The theorem states that a distributed computer system cannot guarantee *consistency*, *availability* and *partition tolerance* at the same time. A BASE guarantee gives up on consistency and relies on an eventual consistency [**35**]. Base guarantee offers:

- Basic availability – system is guaranteed for availability.
- Soft state – indicates that the state of the system may change over time even without input because of the eventual consistency property.
- Eventual consistency – indicates that the system will become consistent over time, and not after every transaction.

NoSQL databases, and therefore NoSQL RDF stores, are classified based on their data storage model. The primary classifications are: *Key-Value stores*, *Document stores*, *Column family stores*, and *Graph databases*.

Key–Value stores – are the simplest NoSQL databases where every single record in the database is stored as an attribute name, key, and its associated value. Key-value stores use the map or dictionary data model where data is accessed and addressed via the key. Because the values stored in key-value stores are independent of each other, the relationship among them has to be established at an application level - i.e. key-value stores are schema-free. In key-value stores new entries can be added dynamically without affecting the system's availability. Amazon DynamoDB and Apache Cassandra are key-value stores. CumulusRDF [**36**], based on Apache Cassandra, is a key-value RDF store.

Document stores – are NoSQL stores that are similar to key-value stores but extend the functionality of key-value stores. The main notion in document stores is a document - which is a collection of data items (or values) that are organized and can be treated as a single item. Unlike key-value stores where values are opaque to the system, values in document stores have an open structure and queries can be written against them. The key-value pairs in document stores are usually encapsulated in JSON or a JSON like structure that allows for nested representation of key-value pairs. NoSQL document stores, just like key-value stores, are schema free. MongoDB and CouchDB are the most popular document stores. Both CouchDB and MongoDB have been used to store and query RDF data.

Wide-column stores – aka column-family stores or just column stores – store data as sections of columns-of-data rather than as rows-of-data. The basic unit in wide-column stores is a column and consists of a key-value pair. Some wide-column stores, for example Google Big Table [**37**], also add a timestamp attribute to a column to indicate when the data is added to the database or when it was last updated.

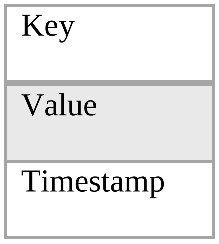

Figure 7 A Column in Wide-column Store

Unlike relational tables, where the number and type of columns are defined prior to the insertion of rows, a column in a wide-column store can be dynamically created as needed. A row in standard column-family store consists of a key, which is unique, and a collection of columns.

There are two types of column-family stores – standard column-family and super column-family. Examples of column stores include BigTable [**37**], HBase [**38**], and HyperTable [**38**]. RDF stores based on column stores have also been developed.

Graph databases – graph databases are another variant of NoSQL databases. According to [**40**] a graph database is “an online database system that exposes a graph data model and supports traditional CRUD methods - Create, Read, Update, and Delete". Graph databases are mostly built for use with OLTP – Online Transaction Processing – systems. RDF data, being a labeled directed graph, can be suitably represented and stored in graph databases. Examples of graph databases include Neo4j, AllegroGraph, and OpenLink Viruoso. AllegroGraph, according to its website, is "a modern, high-performance, persistence graph databases". AllegroGraph is designed to store RDF data and fully compliant with SPARQL 1.0. Because AllegroGraph has been designed from start to store RDF data, some researchers have classified it as a native triple store.

CHAPTER 5

STORING RDF DATA IN HDFS

The RDF data model, which is based on a tuple of subject, predicate and object values, is a flexible way of representing information about resources and relationships between resources. This chapter presents a review of how RDF data, triples, can be stored using facilities provided by the Hadoop ecosystem, most notably Hive.

5.1 RDF and HDFS

RDF triples can be stored and accessed in HDFS by creating a relational layer on top of HDFS that maps triples into relational schemas. Hive, for example, allows storing data in HDFS based on a relational schema that defined by the user.

Though there are some discrepancies among researchers regarding the naming and classification of relational schemas for RDF data, most researchers classify these schemas in to three groups [**40, 41, 42, 43**]:

- Triple table – the entire RDF data is stored as a single table with three columns – subject, predicate and object. Each triple is stored as a row in this table.
- Property-table – triples are grouped together by predicate name. In this scheme, all triples with the same predicate are stored in a separate table. Some researchers call property tables vertical partitioning.
- Cluster-property tables – in this scheme triples are grouped into classes based on correlation and occurrence of predicates. A triple is stored in the table based on the predicate class it belongs to.

5.2 Triple-store

The triple-store schema, aka triple table, is the simplest and straightforward to implement. Though the triple-store schema may be suitable for very simple queries, running complex queries will require many self-joins. In a big RDF data context, a triple table may hold hundreds of millions of records and self-joining such table would have a dramatic impact on performance. Table 1, from Chapter 2, can be represented using triple table as:

Subject	**Predicate**	**Object**
Einstein	wasBorn	1879
Einstein	developed	Special Relativity
Einstein	developed	General Relativity
Einstein	diedIn	1955
Newton	wasBorn	1642
Galileo	diedIn	1642
General Relativity	dealsWith	Gravity
Newton	discoveredLawsOf	Gravity

Table 2 Triple Table Representation of RDF Graph

5.3 Property Table

The property table schema can have many variants. In one variant, called vertical partitioning, each predicate is separated out into its own table having two columns – subject and object. Subjects that have the same predicate are stored in the same table. For the RDF data from Chapter 2, a vertical partitioning scheme results in five tables:

WasBorn:

Subject	**Object**
Einstein	1879
Newton	1642

Developed:

Subject	**Object**
Einstein	Special Relativity
Einstein	General Relativity

DiedIn:

Subject	**Object**
Einstein	1955
Galileo	1642

DealsWith:

Subject	**Object**
General Relativity	Gravity

DiscoveredLawsOf:

Subject	**Object**
Newton	Gravity

Figure 8 Vertical Representation of RDF Triples

Another variant of the property table, which is the opposite extreme of the vertical partitioning, lays out all the predicates in the RDF data set as columns of a table. The resulting structure is one table that a subject column and as many predicate columns as there are in the dataset. This structure is sometimes called horizontal. The horizontal representation[2] for the RDF data from Chapter 2 is:

Subject	**WasBorn**	**Developed**	**DiedIn**	**DealsWith**	**Dis.LawsOf**
Einstein	1879	Special R.	1955	Null	Null
Einstein	1879	General R.	1955	Null	Null
Newton	1642	Null	Null	Null	Gravity
Galileo	Null	Null	1642	Null	Null

Figure 9 Horizontal Representation of RDF Triples

As can be observed from the above table, the horizontal representation has many null values for undefined subject-predicate values.

[2] There are discrepancies among researchers regarding the naming [40, 41].

5.4 Clustered-property Tables

In the clustered-property table representation[3], if many subjects all have the same set of predicates, then a table with these predicates as columns may be created. Considering the example from the previous sections, the predicates *WasBorn* and *DiedIn* are shared by two subjects and hence can be joined together as a single table.

Subject	**WasBorn**	**DiedIn**
Einstein	1879	1879
Newton	1642	Null
Galileo	Null	1642

Figure 10 Clustered-property Table Representation of RDF Triples

Predicates that occur less in the RDF data set will be represented using their own tables, just like the vertical partitioning scheme.

Developed:

Subject	**Predicate**
Einstein	Special Relativity
Einstein	General Relativity

[3] Some researchers call clustered-property representation property-class representation.

DealsWith:

Subject	**Predicate**
General Relativity	Gravity

DiscoveredLawsOf:

Subject	Predicate
Newton	Gravity

Figure 11 Clustered-property Table Representation (contd.)

CHAPTER 6

SPARQL BENCHMARKS

RDF benchmarks are models that test the scalability, performance, and efficiency of RDF stores in a standard and systematic way. These models define a finite set of carefully designed queries that vary in their selectivity, output size, and depth – i.e. on the number of nodes the query visits in the RDF graph. The scalability, performance and efficiency of an RDF store are measured by running these benchmark queries on a SPARQL end–point. Each of these models also provides utility application that can be used to generate RDF data. This chapter reviews four such benchmarks – DBpedia, SP2Bench, BSBM, and LUBM. The chapter concludes by reviewing some of the criticisms that have been leveled against the benchmarks.

6.1 DBpedia

The DBpedia SPARQL benchmark is a generic SPARQL benchmark creation methodology that is based on query–log mining, clustering and SPARQL feature analysis. The methodology has been applied on different RDF data sizes from the DBpedia dataset (RDF dataset derived from Wikipedia). In order to construct benchmark queries that are prototypical to real queries, the creators performed query analysis and clustering on queries sent to DBpedia SPARQL end–point – http://dbpedia.org/sparql. Based on the highest ranked query clusters they derived a set of 25 SPARQL query templates to which parameterization is applied to generate the actual benchmark queries [**44**, **45**].

The creators of the benchmark ran the benchmark queries against the Virtuoso, Sesame, Jena–TDB, and Big OWLIM triple stores and found out that the performance of these triples stores is far less homogenous than what other benchmarks suggested. Furthermore, they argue that their benchmark is a pure benchmark and is more suitable than other benchmarks, like LUBM, BSBM and SP2B, to evaluate the performance of triple stores because the underlying data structure these benchmarks use, they state, are relational in nature and are most suitable to benchmark relationally–backed RDF stores.

DBpedia SPARQL benchmark project is currently deprecated and replaced by project Mosquito, https://github.com/AKSW/mosquito, which is a program and an API framework that can be used to benchmark different types of SPARQL end–points.

6.2 LUBM

The Lehigh University Benchmark (LUBM) is a benchmark that is created to facilitate the evaluation of RDF data in a standard and systematic way. The benchmark is based on ontology for the university domain and can generate synthetic data of arbitrary size and provides fourteen queries that represent a variety of RDF graph properties and several performance metrics [**46**, **47**].

The LUBM benchmark, according to its creators, has been designed with the following goals:

1. Support for extensional, rather than intentional queries – the creators of the benchmark conjecture that the majority of semantic web applications will use queries about the data instances, extensional queries, rather than queries about classes and properties, intentional queries. The benchmark, therefore, focuses on extensional queries.
2. Support for data scalability – the LUBM creators predict that, in the long run, data will outnumber ontologies and the benchmark therefore has to be scalable.
3. Support for a moderate size and complex ontology – the benchmarks tries to strike a balance between queries that manipulate large and complex ontologies versus queries that manipulate ontologies based on OWL Lite.

LUBM provides fourteen benchmark queries that vary in their input size, selectivity, complexity, class and property hierarchy, as well as the degree of logical inference that is required to answer the query.

LUBM measures input size as the proportion of the class instances involved in a query to the total number of class instances in the benchmark data. Input size is considered large if the proportion is greater than 5%. Selectivity, on the other hand, measures the ratio of the number class instances involved in a query to the number of class instances that satisfy the query. A query is said to have high complexity if this proportion is less than 10%. The creators of the benchmark argue that query complexity can be implicitly measured by the number of classes and properties that are involved in the query as well as by the depth and width of the class hierarchies involved in the query. The fourteen benchmark queries LUBM uses have different levels of input size, selectivity and complexity [**47**].

The dataset generated by LUBM consists of fifteen to twenty five departments per university, each described in a separate OWL file. The LUBM data generator generates a data set based on the number of universities users specify.

LUBM is used in this thesis to validate the SPARQL to SQL compiler, RQ2SQL, component of the Presto–RDF implementation this thesis proposes to analyze big RDF data. Because Presto–RDF does not support RDFS inference, LUBM was not chosen as a benchmark in this thesis.

6.3 BSBM

The Berlin SPARQL Benchmark (BSBM), proposed in 2008, is another benchmark that can measure the performance of native RDF stores against systems that convert SPARQL to SQL, also known as SPARQL–to–SQL rewriters [**48**, **49**]. Examples are D2R Server and Virtuoso

RDF Views. Presto–RDF, the architecture this thesis proposes, can be considered in this group because it translates SPARQL to SQL, i.e. it is a SPARQL–to–SQL re–writer architecture.

The benchmark is based on an e–commerce use case in which different vendors can offer different sets of products and consumers can write reviews on these products. The benchmark has an abstract data model based on the following classes: *Product, ProductType*, *ProductFeature*, *Producer*, *Vendor*, *Offer*, *Review*, and *Person*. Based on these classes and a set of data production rules the benchmark can generate RDF data of different sizes. The benchmark also offers a relational representation of the RDF data.

The BSBM data generator works based on the number of products that a user specifies. Each product is described by rdfs:label and rdfs:comment and can have 3 to 5 textual properties whose value can consist of 5 to 15 words that are randomly chosen from a dictionary. A product also has 3 to 5 numeric properties whose values range from 1 to 2000, with a normal distribution. Each product also has a type that is part of a type hierarchy where the depth and width of the subsumption hierarchy depends on a scaling factor that users can specify.

The BSBM benchmark queries are 25 in number and, just like the other benchmarks reviewed in this thesis, satisfy two principles for the design of benchmark queries, aka, query mixes [**48**]:

1. Benchmark queries should be designed to test specific features of the query language, SPARQL, or the data management, RDF.
2. Benchmark queries should be based on real world use cases.

BSBM places much more emphasis on the second criteria and its 25 benchmark queries try to simulate a realistic search and navigation pattern by an e–commerce user looking for a product.

The creators of the benchmark tested the 25 queries against four RDF triple stores – Sesame, Virtuoso, Jena TDB, and Jena SDB – and two SPARQL–to–SQL rewriters – D2R Server and

Virtuoso RDF Views. Their results indicate that Virtuoso triple store has the best overall performance against the other three RDF triple stores while Virtuoso RDF Views outperformed D2R Server.

6.4 SP^{2}Bench

SP^{2}Bench, proposed in 2008, is another SPARQL benchmark that is designed to test SPARQL queries over RDF triples stores as well as SPARQL–to–SQL re–write systems. SP^{2}Bench focuses on how well an RDF store supports the different SPARQL operators and their combination – known as operator constellations [**50**, **51**]. This is different from BSBM, which places more emphasis on creating benchmark queries that are closer to the real world. And, unlike LUBM, SP2Bench presents a set of benchmark queries that are based on the UNION and OPTIONAL operators that are central to SPARQL.

SP^{2}Bench data model is based on the DBLP, http://www.informatik.uni–trier.de/~ley/db/, a computer science bibliography created in the 1980s and currently featuring more than 2.3 million articles. The SP^{2}Bench data generator can generate any number of triples based on what a user specifies. For the experiments conducted in this thesis, for example, triples of size 10, 20, and 30 million were generated. The SP^{2}Bench data model defines classes such as: Person, Document, Journal, Article, Book, PhD Thesis, etc.

SP^{2}Bench benchmark queries are 12 in number and are designed to evaluate an RDF storage scheme by imposing various data accesses – for example, through a triple subject, predicate or object. Unlike BSBM and LUBM, which focus on common SPARQL patterns, SP^{2}Bench also provides queries (e.g. Query 9 (Q9) and 10 (Q10)) that test uncommon access patterns. Q9, for example, returns incoming and outgoing properties of Persons.

The creators of the SP^{2}Bench tested their benchmark queries on Jena TDB, Redland RDF Processor, Sesame and Virtuoso and were able to discover that, despite claims that a vertically partitioned RDF store is superior to a simple triple store, some queries performed well on simple triple store than a vertically partitioned store. This thesis also verifies this observation.

6.5 Rationale for Choosing SP^{2}Bench

Though there was a plan to conduct the experiments of this thesis based on the four benchmarks discussed in the previous section, the time and money involved in running the benchmark queries over a cluster of machines proved difficult. The author of this thesis also noted, based on the literature review he conducted, that no single benchmark is good enough to characterize the performance of RDF stores. Instead, a combination of them taken in a bottom–up approach may yield a better benchmarking. More specifically, the author proposes the following benchmarking guideline that can be used while developing and benchmarking an RDF store:

SP^{2}Bench → BSBM → LUBM → DBpedia

Start with SP^{2}Bench, which evaluates an RDF store at low–level, i.e. at SPARQL operator constellations level, and progress toward DBpedia – which is a higher level measure that benchmarks on real data with real queries. Starting with SP^{2}Bench, the author of this thesis believes, would help optimize RDF store designers optimize their SPARQL end–point. The guideline can be also be viewed from the perspective of an RDF store designer and user. An RDF store user, while evaluating an RDF store, may rely on the high–level benchmarks (e.g. DBpedia) and ignore SP^{2}Bench. On the other hand, an RDF store designer can’t do without SP^{2}Bench as it focuses on low-level implementation and how efficiently an RDF implementation implements the SPARQL operators.

6.6 Benchmarking the Benchmarks

LUBM, BSBM, and SP^{2}Bench benchmarks all offer applications that can be used to generate RDF data set. Each generator provides its own command line interface that takes in different sets of parameters.

The LUBM data set generator supports only the OWL format. BSBM can generate data sets in n–triples, turtle, RDF/XML, TriG and MySQL dump. SP^{2}Bench supports the n–triples format.

To evaluate the dataset generation performance, triples of size 10 million, 20 million, and 30 million were generated. The evaluation was conducted using the latest versions of the data set generators – UBA 1.7 for LUBM[4], version 3.1 for BSBM[5], and SP2B 1.0.1 for SP^{2}Bench. The data generators were run on a 16–core machine with 16GB RAM running Linux Mint 17 (Quiana).

The results of the evaluation show that both BSBM and SP^{2}Bench have comparable dataset generation performance while LUBM takes almost twice the amount of time to generate the same number of triples. This could be due to the fact that LUBM generates multiple OWL files which require many I/O operations.

[4] LUBM generates OWL files - LUBM(72), LUBM(144), LUBM(215) were used to generate ~ 10M, 20M, and 30M triples

[5] BSBM generates triples based on products - 28480 products correspond to 10 million triples.

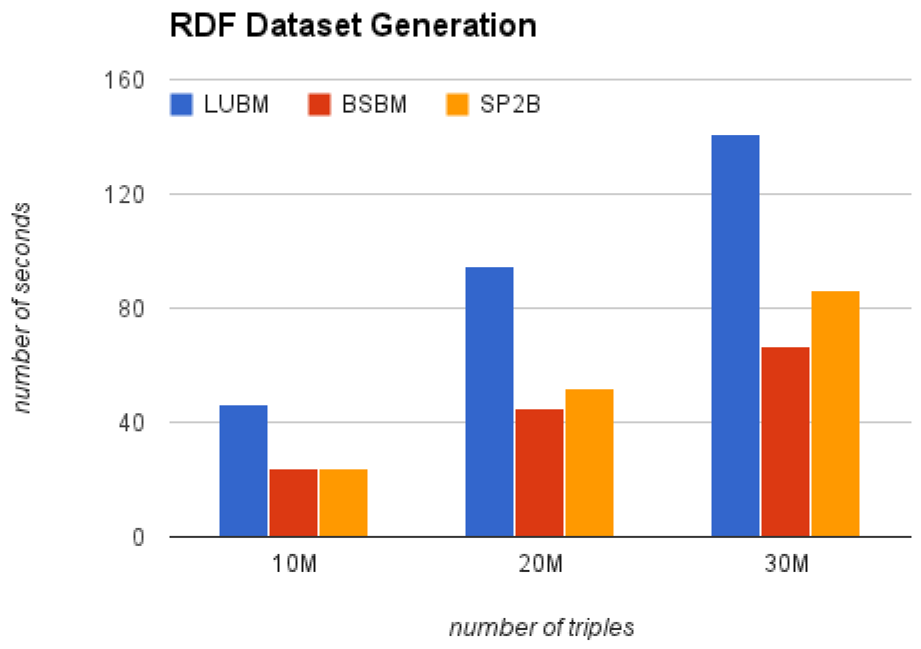

Figure 12 Performance Evaluation of RDF Dataset Generation

In terms of the size of the dataset that is generated, LUBM has the smallest size of dataset per the number of triples. For 20 million triples, for example, LUBM generates a total of 1.6 GB files while SP^{2}Bench generates a file size of 2.3 GB. BSBM, on the other hand, generates a 5.2 GB file - i.e. for 20 million triples.

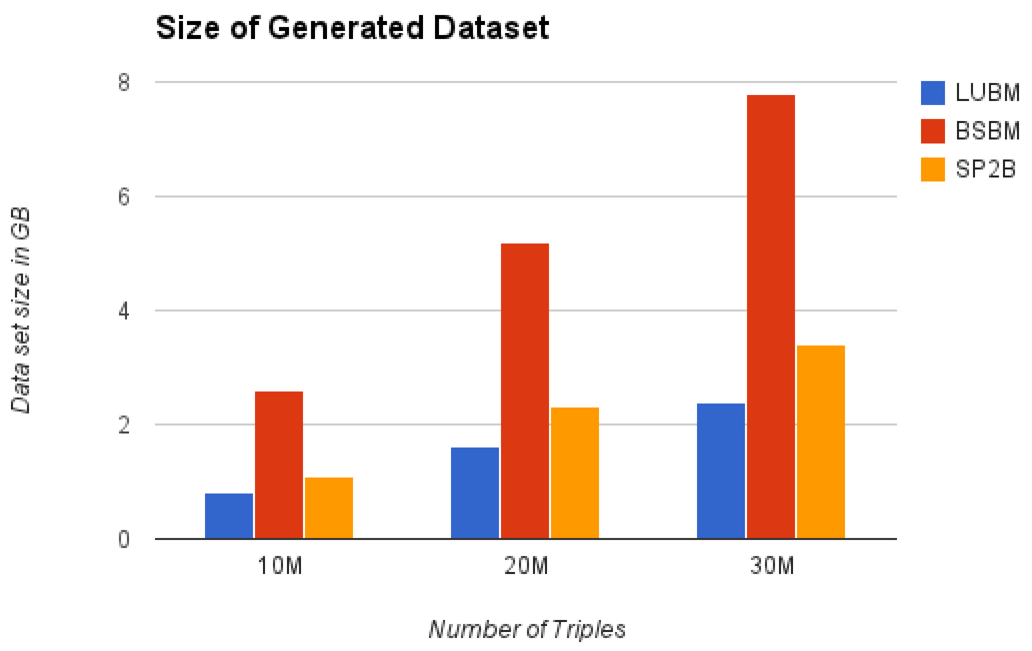

Figure 13 Size of Generated RDF Dataset

6.7 Critique of the Standard Benchmarks

A number of papers that evaluate different RDF stores using the SPARQL benchmarks discussed in the previous sections have been published. However, only a single paper was found that looked critically into how well the data sets generated by these benchmarks resembles real data, from different sources, in its structure [**52**]. The paper argues that, though primitive RDF data set metrics, such as number of subjects, predicates, objects, number of predicates per type, etc. are useful, they offer no little insight into the structure of an RDF dataset. The authors developed a mathematical formula, based on the primitive metrics, that evaluates the structured-ness of an RDF dataset using a 0 – 1 scale where a zero represents a data set that is completely unstructured while one denotes a data set that is well structured – e.g. relational data. The authors evaluated DBpedia, LUBM, BSBM, SP^{2}Bench and other benchmarks and found out that

DBpedia is less structured, with a value of 0.025, and resembles real data. LUBM and BSBM both have the highest value, 0.975, implying a high structure in their generated data set. SP2Bench has a value 0.775.

CHAPTER 7

PRESTO, HIVE, AND 4STORE

Presto–RDF, the big RDF query analyzer this thesis proposes, is based on Presto and uses a Hive connector to get data that resides in HDFS. The thesis benchmarks Presto–RDF against Hive and 4store, a native RDF store. This chapter gives an overview of each of these systems. Presto–RDF is discussed in the next chapter.

7.1 Facebook Presto

According to Facebook, “Presto is an open source distributed SQL query engine for running interactive analytic queries against data sources of all sizes ranging from gigabytes to petabytes”. Presto can query data that resides in different systems – Hive, Cassandra, relational databases as well as other proprietary data sources (as long as its interface requirements are met). The motivation behind the creation of Presto is the high latency BI tools have while visualize Tera and Peta bytes of Facebook data. Currently Presto is deployed on Facebook running against thousands of data stores that are distributed at different geographic locations as well as an internal 300PB data warehouse.

7.2 Presto Architecture

Presto is a distributed SQL query engine that runs on a cluster of machines controlled by a single coordinator with hundreds or thousands of worker nodes. Presto is optimized for ad–hoc analysis and supports standard ANSI SQL, including complex queries, aggregation, joins, and window function [**53**].

The simplified architecture of Presto is presented in the diagram below. The client sends SQL query using the Presto command line interface to the coordinator that would then parse, analyze and plan the query execution. The scheduler, component within the coordinator, connects together the execution pipeline and assigns and monitors work to worker nodes that are closer to the data. The client gets data from the output stage, one of the worker nodes, which in turn pulls data from the underlying stages.

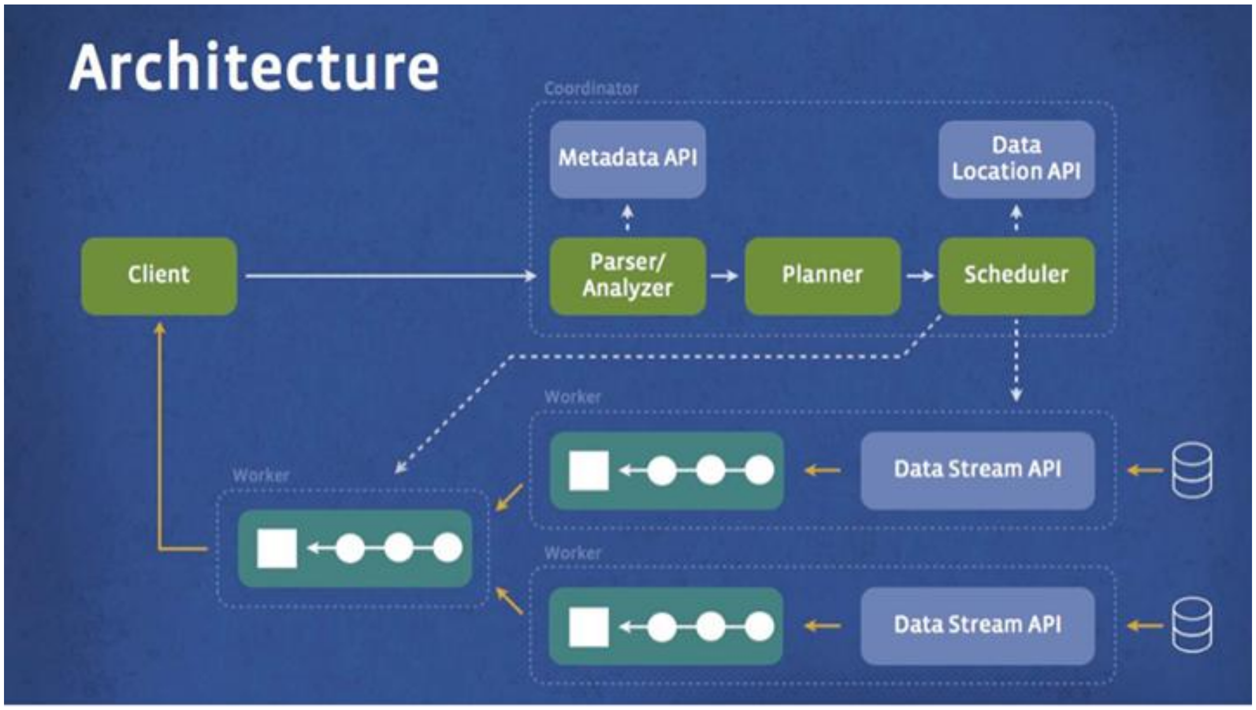

Figure 14 Presto Architecture [53]

7.3 Presto versus MapReduce

SQL queries executed by Hive are implemented using MapReduce where each query is translated into more than one MapReduce tasks that executed over one after another. Each MapReduce task reads intermediate inputs from disk and writes back intermediate results back to disk and finally performs reduces tasks to give the result back to the user. Presto, on the other hand, does not use MapReduce. Instead, it uses a query and execution engine with operators that supports SQL semantics. Presto does all its processing in memory and pipelines its processes across the cluster between stages. The creators believe this avoids unnecessary IO operations and reduces latency [**53**].

Presto is also designed to run against different types of data sources using storage plugins, also known as connectors. Presto relies on interfaces provided by these connectors. These interfaces must provide, according to the Presto specification, a way to fetch metadata, get data locations and access the underlying data. Hence, Presto can ran against any data source as long as the data source provides a connector that can subscribe to the Presto interface. Multiple connectors can also be dynamically installed which would enable Presto to work on many different data sources at the same time. Presto also provides predefined connectors for Hive, Cassandra and TPC–H.

7.4 Apache Hive

Apache Hive is an open–source data warehouse solution for querying and analyzing large data sets residing in Hadoop Distributed File System (HDFS). Hive provides a mechanism to define a relational structure on top of HDFS and supports a SQL–like query language, called HiveQL, which one can use to query data [**54**].

The Hive data model is organized into Tables, Partitions and Buckets:

- Tables – Hive tables are analogous to relational database tables. Each table in Hive is mapped to a specific HDFS directory. A table in Hive stores its data as files inside its directory.
- Partitions – Each table in Hive, which has its own directory in HDFS, can further distribute its data inside subdirectories, called partitions.
- Buckets – Data residing in each partition of a Hive table can further be divided into buckets where each bucket is a file that is stored inside a specific partition. The partitioning of data into buckets is based on the hash of a column in the table.

Hive table columns can be defined with one of these types – integer, float, string, date, boolean, array, and map. Hive also allows programmers to define their own column types.

7.5 Apache Hive Architecture

The paper [**55**] describes the architecture of Hive as a system composed of six main components – External Interfaces, Hive Thrift Server, MetaStore, Driver, Compiler, and Execution Engine.

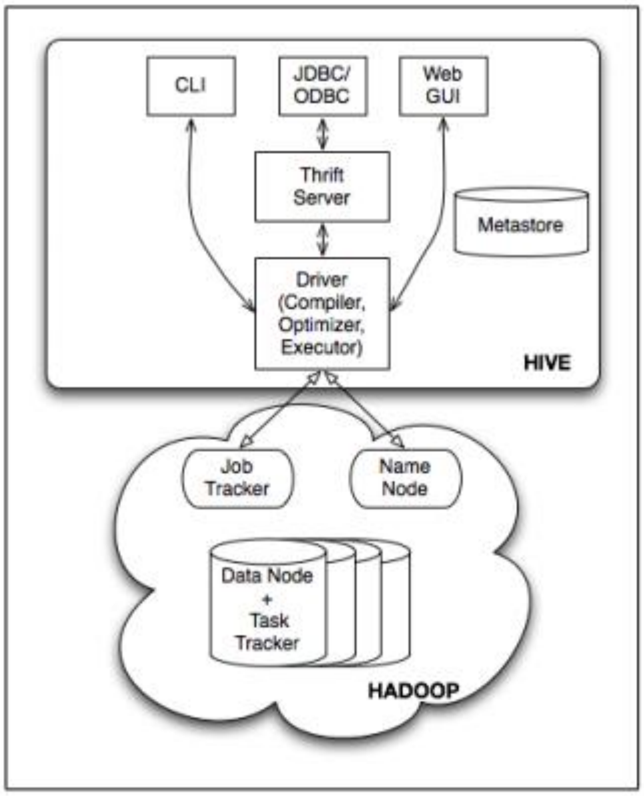

Figure 15 Apache Hive Architecture [55]

- The External Interfaces component provides a command line interface (CLI), a web UI, and application programming interfaces (APIs) that can be used to programmatically connect to Hive, create tables and submit queries.
- Hive Thrift Server exposes a simple API that can be used to execute HiveQL statements.
- The metastore functions as a system catalog that stores the metadata about the tables that are stored in Hive. The metadata is specified during table creation and is re–used every time the table is referenced, in HiveQL. The metastore is Hive’s data warehouse.

- The Driver component manages the life cycle of HiveQL statements during compilation, optimization and execution. The driver is also responsible for submitting map–reduce jobs from the DAG graph which is created by the compiler.
- The Compiler is invoked by the driver upon receiving HiveQL statements and is responsible for translating the statements into a directed acyclic graph (DAG) of map–reduce jobs.
- The Execution engine receives map–reduce jobs, from the driver, and executes it in Hadoop.

Hive was created in Facebook before becoming an Apache project. At Facebook, the Hive warehouse contains more than 700 TB of data and manages more than 5000 queries on daily basis.

7.6 4store

4store was developed at Garlik, www.garlik.com, by Steve Harris to support the company's semantic web application need. 4store is regarded as a native RDF store and has been used by Garlik to store and analyze RDF data of over 15TB. 4store is written in ANSI C99 and was designed to run on UNIX–like systems. According to its creator, 4store is optimized to run on shared–nothing clusters of up to 32 nodes linked with gigabit Ethernet.

Many research papers, including this thesis, have used 4store to benchmark native RDF stores with non–native RDF stores.

CHAPTER 8

PRESTO-RDF

This chapter proposes architecture, called Presto–RDF, which can be used to store and query big RDF data using the Hadoop Distributed File System (HDFS) and Facebook Presto. It also presents RDF–Loader, a component of the architecture, which is used to read, parse and store RDF triples. The next chapter will cover RQ2SQL – a SPARQL to SQL compiler developed as part of the architecture.

8.1 Architecture

Presto–RDF consists of the following components: a command line interface (CLI), a SPARQL to SQL compiler (RQ2SQL), Facebook Presto, Hive Metastore, HDFS, and RDF–Loader. The following diagram illustrates these components:

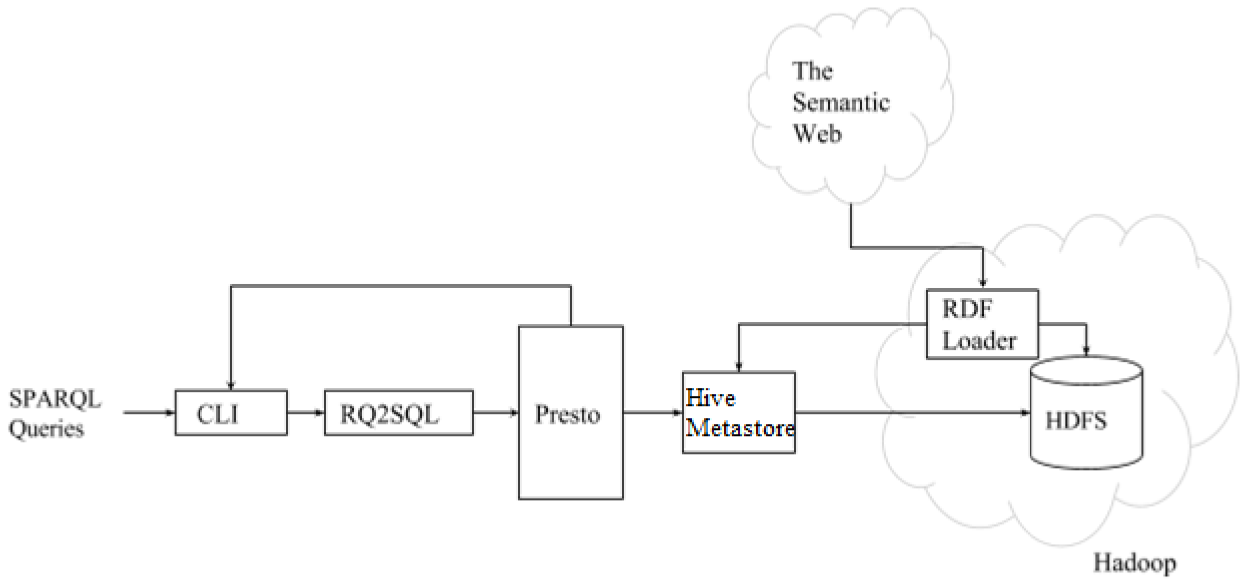

Figure 16 Presto-RDF Architecture

RDF data that extracted from the Semantic Web is parsed and loaded into HDFS using a custom–made loader, RDF–loader, which will also store metadata information on Hive Thrift

Server. When a user submits a SPARQL query over a command line interface, the query is processed by a custom–made SPARQL to SQL converter, RQ2SQL, that translates the the SPARQL query into SQL which would then be submitted to Facebook Presto. Presto, using its Hive connector and Hive Thrift Server, runs the SQL against HDFS and returns the result back to the CLI.

8.2 RDF–Loader

The purpose of the RDF–Loader is to load, parse and store RDF data in HDFS. RDF–Loader implements four different RDF storage schemes and creates external Hive tables whose metadata is stored in the Hive Thrift server.

Before the RDF–Loader can run, the raw RDF data to be first processed and loaded into HDFS using this command:

```
hadoop fs –put file hdfs–dir
```

Once the raw RDF data is uploaded, RDF–Loader runs several MapReduce jobs and stores the output back into HDFS. The structured of the data is defined by the schema that users can specify.

In order for the RDF–Loader to run and process raw RDF, the following input parameters are required:

- *database* – is the name of the database that will be created.
- *target* – is the type of RDF storage structure, i.e. the type of schema. There are four options: *triples, vertical, wide,* and *horizontal*.
- *expand* – this option indicates if *qnames* are to be expanded[6].

[6] The current implementation forces a true value for this option.

- *server* – is the DNS name or IP address of the master node, NameNode, of the Hadoop cluster.
- *port* – is the port number Hadoop listens to connections.
- *input* – is the path of the HDFS directory that holds the raw RDF data.
- *output* – is the path of the HDFS directory the processed RDF data will be stored into.
- *format* – defines the format of the output files as they are stored in HDFS. The current version of the Hive meta–store supports five different formats: SEQUENCEFILE, TEXTFILE, RCFILE, ORC, and AVRO. This thesis makes use of the TEXTFILE format.

The following sections discuss four different RDF storage strategies implemented by the RDF–Loader. The next chapter presents a performance report on each of these storage strategies.

8.3 Triple-store

In the triples store storage scheme, an RDF triple is stored as is – resulting in a table with three columns: subject, predicate and object. If the raw RDF data has 30 million triples, the triple store strategy will have one table with 30 million rows.

The map–reduce algorithm that transforms the raw RDF data into the triples table is quite simple:

```
map (String key, String value)
    // key: RDF file name
    // value: file contents
    for each triple in value
        emit_intermediate (subject + '\t ' + predicate, object)

reduce (String key, Iterator values)
    // key: subject and predicate delimited by tab
    // values: list of object values
    for each v in values
        emit (subject + '\t ' + predicate + object);
```

For an RDF dataset with n number of triples, the map algorithm has O(n) running time while the reducer, which is called once for each unique subject, has O(*s*o*) running time, where *s* is the number of unique subjects and *o* are the average number of object values per subject.

8.4 Wide–table

In the wide table RDF storage scheme, the raw RDF data is parsed and stored as a single table having one column for subject values, and multiple predicate columns for object values. The resulting table has the following schema:

WideTable (String subject, String predicate_1, String predicate_2, …, String predicate_n)

Because it is unlikely that a subject has all the predicates found in the data set, this storage strategy will have a number of null values.

For an RDF data set that has unique object values for a subject–predicate pair, this scheme would result in a table that has *s* number of rows, where *s* is the number of subjects in the data set. For example, given the triple set:

subject_1	predicate_1	object_1
subject_1	predicate_1	object_3
subject_2	predicate_2	object_3

The wide table representation for the triples would be:

subject	predicate_1	predicate_2
subject_1	object_1	object_3

subject_2	null	object_3

If the dataset, however, contains multiple values for the same subject–predicate pair, the table will have multiple rows for the same subject. For example, given the triple set:

subject_1	predicate_1	object_1
subject_1	predicate_1	object_2
subject_2	predicate_2	object_3

The wide table representation for the triples would be:

subject	predicate_1	predicate_2
subject_1	object_1	null
subject_1	object_2	null
subject_2	null	object_3

The algorithm for storing triples using the wide table storage scheme would get complicated if the data set contains subjects with multiple predicates (which is natural) and multiple object values for the same predicate. For example, given the triple set:

subject_1	predicate_1	object_1
subject_1	predicate_1	object_2
subject_1	predicate_2	object_3

subject_2	predicate_2	object_4

The wide table representation for the triples would be:

subject	predicate_1	predicate_2
subject_1	object_1	null
subject_1	object_2	null
subject_1	null	object_3
subject_2	null	object_4

The storage scheme, thus, forces new rows to be created for each unique subject–predicate pair.

The map–reduce algorithm for the wide table storage scheme, as implemented in this thesis, is:

```
map (String key, String value)
    // key: RDF file name
    // value: file contents
    for each triple in value
        emit_intermediate (<subject, predicate>, <predicate, object>)

reduce (String key, Iterator values)
    // key: a <subject, predicate> pair
    // values: list of <predicate, object> pairs
    String subject = key.getSubject();
    String[] row = new String[1 + num_unique_predicates];
    int i = 0
    for each v in values
        row[i] = v.getObject();
        i++;
    emit (subject, row);
```

8.5 Horizontal-store Scheme

The horizontal storage scheme is similar to the wide table storage scheme in terms of the schema of the table. However, unlike the wide–table scheme, it optimizes the number of rows stored for subjects that have multiple object values for the same predicate. Given the example presented in the previous section:

subject_1	predicate_1	object_1
subject_1	predicate_1	object_2
subject_1	predicate_2	object_3
subject_2	predicate_2	object_4

The horizontal storage scheme stores the triples as:

subject	predicate_1	predicate_2
subject_1	object_1	object_3
subject_1	object_2	null
subject_2	null	object_4

In this scheme, it is not necessary to create new rows for each unique subject–predicate pair. Instead, rows that are already created for the same subject, but for a different predicate will be used.

7.6 Vertical-store Scheme

In the vertical storage scheme implemented in this thesis, the raw RDF data is partitioned into different tables based on the predicate values of the triples in the data with each table having two columns – the subject and object values of the triple. Thus, if the raw RDF data has 30 million triples that have 20 unique predicates, the vertical storage scheme will create 20 tables and stores the subject and object values of triples that share the same predicate in the same table.

The map–reduce algorithm works with predicate as a key value and a pair of subject and object values as value:

```
map (String key, String value)
   // key: RDF file name
   // value: file contents
   for each triple in value
      emit_intermediate (predicate, <subject, object>);

reduce (String key, Iterator values)
   // key: predicate
   // values: list of <subject, predicate> pairs
   String table = key.replace_unwanted('_');
   MultipleOutputs<String, String> mos;
   for each v in values
      // create a directory table
     // write the subject, values inside the directory
   mos.write (v.getFirst(), v.getSecond(), table);
```

Because predicate values are URIs that contain non–alpha numeric characters, e.g. http://www.w3.org/1999/02/22–rdf–syntax–ns#, which cannot be used in naming directories, the reducer has to replace these characters with some other character, for example the underscore character, http___www_w3_org_1999_02_22_rdf_syntax_ns_, and creates the directory (which is considered as a table for the Hive Metastore).

In the vertical storage scheme, for a raw RDF data that contains *n* number of triples, the mapper runs at *O(n)* while the reducer runs at O(*p*x*) where *p* and *s* are the number of unique predicates and subjects in the data set, respectively. In the worst case scenario, where there are as many unique predicates and subjects, the number of triples, the map-reduce algorithm for the vertical storage scheme runs at $O\ (n^2)$.

CHAPTER 9

RQ2SQL - SPARQL TO SQL COMPILER

This chapter presents a review of basic SPARQL graph patterns and RQ2SQL – a SPARQL to SQL mini-compiler that is developed as part of Presto–RDF. RQ2SQL (**R**DF **Q**uery to **SQL**) converts SPARQL queries into SQL statements that can be run on Presto and Hive. RQ2SQL is implemented in Flex and Bison using C++11.

9.1 SPARQL Graph Patterns

SPARQL query processing is based on graph pattern matching. Complex graph patterns can be constructed by combining few basic graph pattern techniques. The W3C classifies SPARQL graph pattern matching into five smaller patterns [**14**]: *Basic Graph Pattern*, *Group Graph Pattern*, *Optional Graph Pattern*, *Alternate Graph Pattern* and *Named Graph Pattern*.

- Basic graph patterns – are set of triple patterns where the pattern matching is defined in terms of joining the results from individual triples. A single graph pattern is composed of a sequence of triples that may optionally be interrupted by filter expressions. The below example is a basic graph pattern.

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
SELECT ?name ?age
WHERE  {
   ?x foaf:name ?name .
   ?x foaf:age ?age .
}
```

- RQ2SQL translates the above SPARQL into the SQL[7]:

```
SELECT T0.object,
       T1.object
FROM http___xmlns_com_foaf_0_1_name T0
JOIN http___xmlns_com_foaf_0_1_age T1 ON (T1.subject = T0.subject)
```

- Group graph pattern – a group graph pattern is specified by delimiting it with braces. The below example specifies one graph pattern with two basic graph patterns.

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
SELECT ?name ?age
WHERE  {
   { ?x foaf:name ?name . }
   { ?x foaf:age ?age . }
}
```

The RQ2SQL translation, for the vertical store, is the same as the previous:

```
SELECT T0.object,
       T1.object
FROM http___xmlns_com_foaf_0_1_name T0
JOIN http___xmlns_com_foaf_0_1_age T1 ON (T1.subject = T0.subject)
```

- Optional graph pattern – optional graph patterns are specified using the OPTIONAL keyword. The semantics of the optional graph pattern matching is that it either adds additional binding to the solution or would leave it unchanged.

  Given the RDF data:

```
@prefix foaf:      <http://xmlns.com/foaf/0.1/> .
@prefix rdf:       <http://www.w3.org/1999/02/22–rdf–syntax–ns#> .
```

[7] The translation given here is for the vertical store

```
_:a  rdf:type            foaf:Person .
_:a  foaf:name "Michael" .
_:a  foaf:email <mailto:michael@example.com> .
_:a  foaf:email <mailto:michael@hahusofware.com> .

_:b  rdf:type            foaf:Person .
_:b  foaf:name "Mulugeta" .
```

And a SPARQL optional graph pattern query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
WHERE  {
        ?x foaf:name  ?name .
        OPTIONAL {
                ?x  foaf:email  ?email
        }
}
```

Would give the result:

name	**email**
“Michael”	<mailto:michael@example.com>
“Michael”	<mailto:michael@hahusoftware.com>
“Mulugeta”	

The RQ2SQL translation, for the vertical store, is the same as the previous:

```
SELECT T0.object,
     T1.object
FROM http___xmlns_com_foaf_0_1_name T0
LEFT JOIN http___xmlns_com_foaf_0_1_email T1 ON (T1.subject =
T0.subject)
```

Constraints can also be applied to optional graph patterns.

- Alternate graph Pattern – are constructed by specifying the keyword UNION between two graph patterns.
- Named graph patterns – are constructed by specifying a FROM NAMED IRI where each IRI is used to provide one named graph in the RDF dataset. Using same IRI in two or more NAMED clauses would result in one named graph.

```
# Graph: http://example.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Mulugeta" .
_:a foaf:email <mailto:mulugeta@hahusoftware.com> .

# Graph: http://example.org/alice
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:a foaf:name "Michael" .
_:a foaf:email <mailto:michael@example.com> .

...
FROM NAMED <http://example.org/michael>
FROM NAMED <http://example.org/mulugeta>
...
```

RQ2SQL *does not* support Named graph patterns.

9.2 SPARQL Solution Sequences and Modifiers

The results returned from a SPARQL query are unordered collection of single or composite values that, according the W3C, can be regarded as solution sequences with no specific order. SPARQL defines six solution modifiers: *order*, *projection*, *distinct*, *reduced*, *offset* and *limit*.

- **Order modifier** – is specified by the ORDER BY clause and forms the order of a solution sequence. Ordering can be qualified as ASC for ascending or DESC for descending.

- **Projection modifier** – is specified by listing a subset of variables defined in the pattern-matching clause.
- **Distinct modifier** – is specified by the DISTINCT keyword and filters out duplicates from the solution sequence.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?name WHERE { ?x foaf:name ?name }
```

RQ2SQL translation:

```
SELECT DISTINCT T0.object
FROM http___xmlns_com_foaf_0_1_name T0
```

- **Reduced modifier** – unlike the distinct modifiers which ensures that duplicate solutions are eliminated from the solution sequence, the reduced modifier, specified by the REDUCED keyword, *permits* them to be eliminated. The result set of a solution sequence with a reduced modifier is at least one and at most the cardinality of the solution sequence without the distinct and reduce modifiers.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT REDUCED ?name WHERE {
?x foaf:name ?name
}
```

RQ2SQL *does not* support the REDUCED keyword.

- **Offset modifier** – just like SQL, the offset modifier, specified by the OFFSET keyword, returns results of the solution sequence starting at the specified offset value. Offset value of 0 has no effect.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>

SELECT  ?name
WHERE   {
        ?x foaf:name ?name
}
ORDER BY ?name
LIMIT   5
OFFSET  10
```

RQ2SQL translation:

```
SELECT TOP 5 T0.object
FROM http___xmlns_com_foaf_0_1_name T0
ORDER BY http___xmlns_com_foaf_0_1_name
```

Both Presto and Hive do not support the OFFSET keyword.

- **Limit modifier** – just like SQL, the LIMIT modifier puts an upper bound to the number of solution sequences returned. A limit value of 0 would return no results. A negative limit value is not valid.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
        ?x foaf:name ?name
}
LIMIT 20
```

RQ2SQL translation:

```
SELECT TOP 20 T0.object
FROM http___xmlns_com_foaf_0_1_name T0
```

- The ASK query modifier – SPARQL queries specified using the ASK form test whether or not a SPARQL query has a solution.

Given the following triples:

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .
_:a  foaf:name       "Michael" .
_:a  foaf:homepage   <http://work.example.org/michael/> .
_:b  foaf:name       "Mulugeta" .
_:b  foaf:mbox       <mailto:mulugeta@hahusoftware.com> .
```

Running the SPARQL query:

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
ASK  { ?x foaf:name  "Michael" }
```

Returns the value: yes.

RQ2SQL *does not* support the ASK query modifier.

9.3 RQ2SQL

RQ2SQL is a mini SPARQL to SQL compiler built using Flex – a lexical analyzer creator – and Bison – a parser generator creator. RQ2SQL supports basic SPARQL queries including OPTIONALS, FILTERS as well as ORDER BY, DISTINCT, projection and LIMIT modifiers. However, it does *not* support UNION, ASK, named graph patterns as well as group graph patterns. RQ2SQL generates SQL queries for the four different RDF storage schemas explained in chapter 7 – triple, wide, horizontal, and vertical.

Translating LUBM queries to SQL

RQ2SQL was tested for correctness by compiling the 14 LUBM benchmark queries against Presto and then comparing the result with the output generated after running same queries on 4store. This section presents selected queries from LUBM and their RQ2SQL translation for the vertical storage scheme.

- **Q1**

```
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>.
}
```

Graph representation:

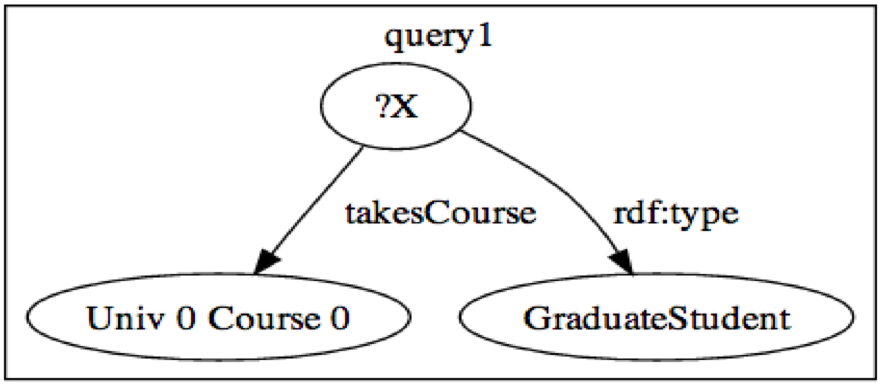

Figure 17 LUBM Query 1 [47]

RQ2SQL translation – for the vertical storage scheme:

```
SELECT T1.Subject
FROM http___www_w3_org_1999_02_22_rdf_syntax_ns_type T0
JOIN http___www_lehigh_edu__zhp2_2004_0401_univ_bench_owl_takesCourse T1
ON (T1.Subject = T0.Subject)
```

- **Q4**

```
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.
}
```

Graph representation:

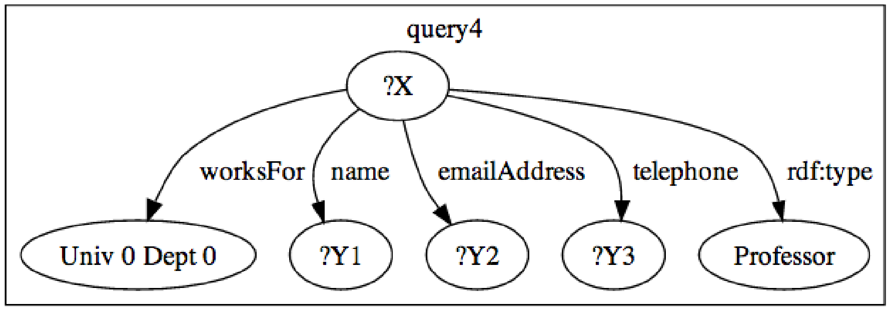

Figure 18 LUBM Query 4 [47]

RQ2SQL translation:

```
SELECT T4.Subject,
       T2.Object,
       T3.Object,
       T4.Object
FROM http___www_w3_org_1999_02_22_rdf_syntax_ns_type T0
JOIN http___www_lehigh_edu__zhp2_2004_0401_univ_bench_owl_worksFor T1 ON
(T1.Subject = T0.Subject)
JOIN http___www_lehigh_edu__zhp2_2004_0401_univ_bench_owl_name T2 ON
(T2.Subject = T1.Subject)
JOIN http___www_lehigh_edu__zhp2_2004_0401_univ_bench_owl_emailAddress T3
ON (T3.Subject = T2.Subject)
JOIN http___www_lehigh_edu__zhp2_2004_0401_univ_bench_owl_telephone T4 ON
(T4.Subject = T3.Subject)
```

- **Q12**

```
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 ?Y
WHERE {
          ?X rdf:type ub:Chair .
          ?Y rdf:type ub:Department .
          ?X ub:worksFor ?Y .
          ?Y ub:subOrganizationOf <http://www.University0.edu>
}
```

Graph representation:

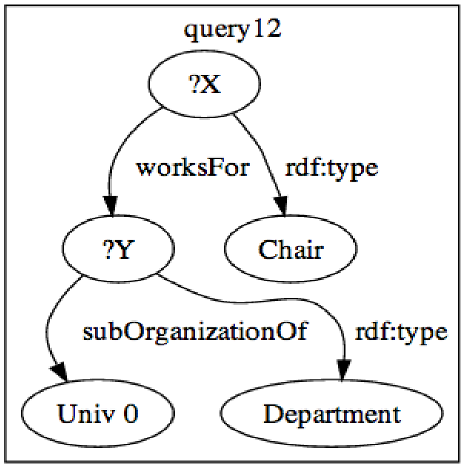

Figure 19 LUBM Query 12 [47]

RQ2SQL translation:

```
SELECT T2.Subject,
       T3.Subject
FROM http___www_w3_org_1999_02_22_rdf_syntax_ns_type T0
JOIN http___www_w3_org_1999_02_22_rdf_syntax_ns_type T1
JOIN http___www_lehigh_edu__zhp2_2004_0401_univ_bench_owl_worksFor T2 ON
(T2.Object = T1.Subject AND T2.Subject = T0.Subject)
JOIN ttp___www_lehigh_edu__zhp2_2004_0401_univ_bench_owl_subOrganizationOf
T3 ON (T3.Subject = T2.Object)
```

- **Q13**

```
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.
        <http://www.University0.edu> ub:hasAlumnus ?x.
}
```

Graph representation

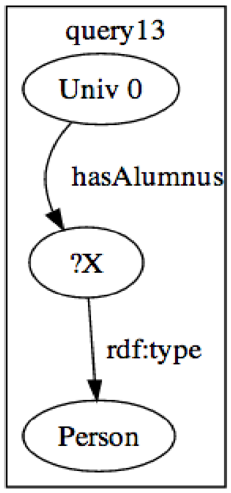

Figure 20 LUBM Query 13 [47]

RQ2SQL translation:

```
SELECT T1.Object
FROM http___www_w3_org_1999_02_22_rdf_syntax_ns_type T0
JOIN http___www_lehigh_edu__zhp2_2004_0401_univ_bench_owl_hasAlumnus T1
ON (T1.Object = T0.Subject)
```

- **Q14**

```
SELECT ?x WHERE {
   ?x rdf:type ub:UndergraduateStudent.
}
```

Graph representation:

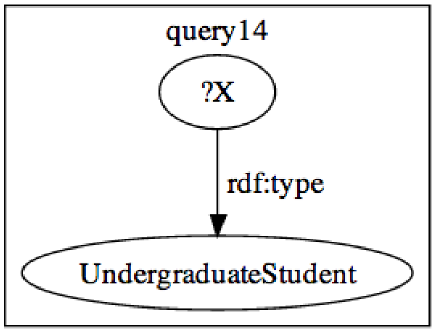

Figure 21 LUBM Query 14 [47]

RQ2SQL translation:

```
SELECT T0.Subject
FROM http___www_w3_org_1999_02_22_rdf_syntax_ns_type T0
```

CHAPTER 10

BENCHMARKING PRESTO-RDF

This chapter presents the experiment and the results conducted to benchmark the performance of Presto-RDF against Hive. A comparative measurement was also done on 4store – a native RDF store. Overall, two experimental setups were constructed for benchmarking the performance of Presto-RDF. The first setup was a 4-node cluster virtualized on a single 16GB memory machine. The second setup was 8-node cluster virtualized on the Windows Azure platform. The second setup was required because the experiments conducted used up the hard disk space and it was not possible to run queries on triples of more than 4 million. For the experiment, four benchmark queries from SP^{2}Bench were used and three different RDF storage schemes were evaluated – triple, vertical and horizontal stores.

10.1 Benchmark Queries

The experiment was based on running four benchmark queries, from SP^{2}Bench[8], with different degrees of complexity – query 1, 6, 8, and 11. The SP^{2}Bench use case is based on the DBLP[9].

Query 1: return the year of publication of journal 1

```
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX bench:   <http://localhost/vocabulary/bench/>
PREFIX xsd:     <http://www.w3.org/2001/XMLSchema#>

SELECT ?yr
WHERE {
  ?journal rdf:type bench:Journal .
  ?journal dc:title "Journal 1 (1940)"^^xsd:string .
  ?journal dcterms:issued ?yr
}
```

[8] The reasons behind choosing SP^{2}Bench can be found in section 5.5.
[9] http://www.informatik.uni-trier.de/~ley/db/

Query 6: return, for each year, the set of all publications authored by persons that have not published in years before.

```
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>

SELECT ?yr ?name ?document
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?document rdf:type ?class .
  ?document dcterms:issued ?yr .
  ?document dc:creator ?author .
  ?author foaf:name ?name
  OPTIONAL {
    ?class2 rdfs:subClassOf foaf:Document .
    ?document2 rdf:type ?class2 .
    ?document2 dcterms:issued ?yr2 .
    ?document2 dc:creator ?author2
    FILTER (?author=?author2 && ?yr2<?yr)
  } FILTER (!bound(?author2))
}
```

Query 8: Compute authors that have published with Paul Erdoes, or with an author that has published with Paul Erdoes.

```
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>

SELECT DISTINCT ?name
WHERE {
  ?erdoes rdf:type foaf:Person .
  ?erdoes foaf:name "Paul Erdoes"^^xsd:string .
  {
    ?document dc:creator ?erdoes .
    ?document dc:creator ?author .
    ?document2 dc:creator ?author .
    ?document2 dc:creator ?author2 .
```

```
   ?author2 foaf:name ?name
   FILTER (?author!=?erdoes &&
           ?document2!=?document &&
           ?author2!=?erdoes &&
           ?author2!=?author)
  } UNION {
   ?document dc:creator ?erdoes.
   ?document dc:creator ?author.
   ?author foaf:name ?name
   FILTER (?author!=?erdoes)
  }
}
```

Query 11: Return (up to) 10 electronic edition URLs starting from the 51th publication, in lexicographical order.

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

SELECT ?ee
WHERE {
  ?publication rdfs:seeAlso ?ee
}
ORDER BY ?ee
LIMIT 10
OFFSET 50
```

10.2 Four Node Cluster Setup

The first setup was a virtualized four node cluster on a machine with 16GB of memory and 300GB of hard disk. Each of the four virtual machines had 2GB of memory and 32GB of hard disk. To evaluate the performance of Presto-RDF, an RDF dataset with three million (3M) triples was generated and the four benchmark queries from section 9.1 were ran on 2-Nodes and 4-Nodes for each of the three storage schemes - triples, vertical, and horizontal.

10.3 Loading Time for 3M Triples

Once the 3M triples were generated using the SP2Bench dataset generator, it was copied into a specific HDFS directory. The copying took 27 seconds. After the triples data was loaded into

HDFS, RDF-Loader was run on the data to parse the triples and store them in the three storage schemas - triples, vertical and horizontal storage schemes. Figure 22 shows a comparison of the loading times.

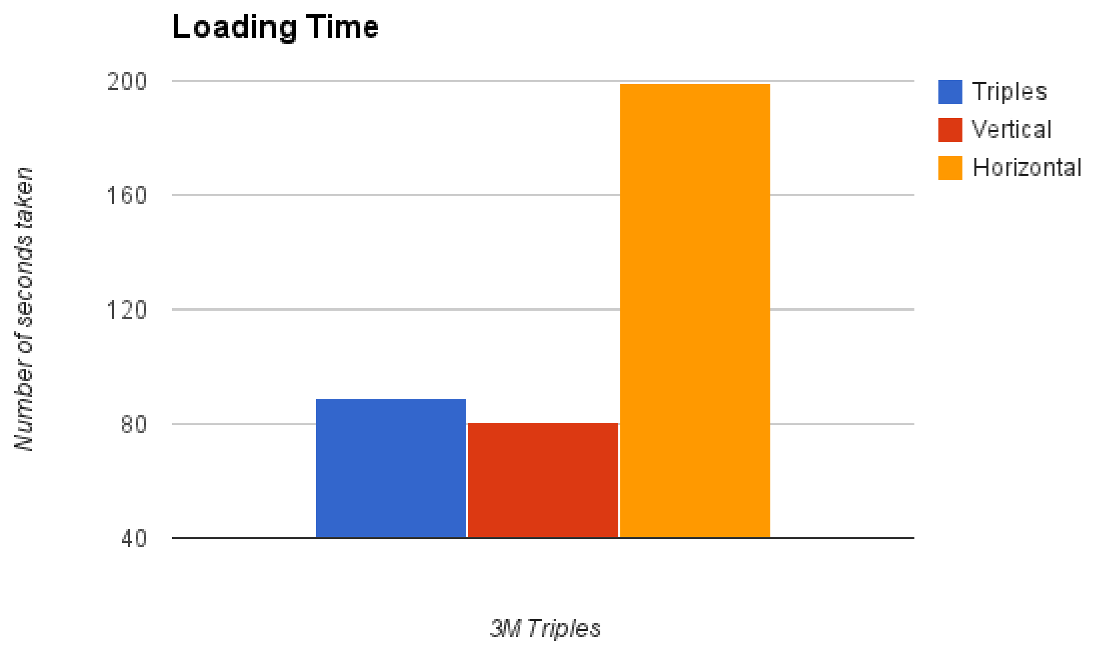

Figure 22 Loading Time for 3M Triples

The results above measure the performance of the RDF-Loader component of Presto-RDF in parsing and loading 3M triples for the three different storage strategies under study - triples, vertical and horizontal. The results are in accordance with the complexity of the map-reduce algorithms (from Chapter 7) used to parse, decompose and store the 3M triples for the three storage strategies.

10.4 Benchmarking using 3M Triples

After the RDF-Loader loaded the 3M triples into HDFS and the corresponding external tables have been created on Hive metastore, the SQL equivalent of the four benchmark queries from section 9.1 were run on a 2-Node[10] and 4-Node cluster.

10.4.1 Performance Comparison - Triples Storage Scheme[11] -

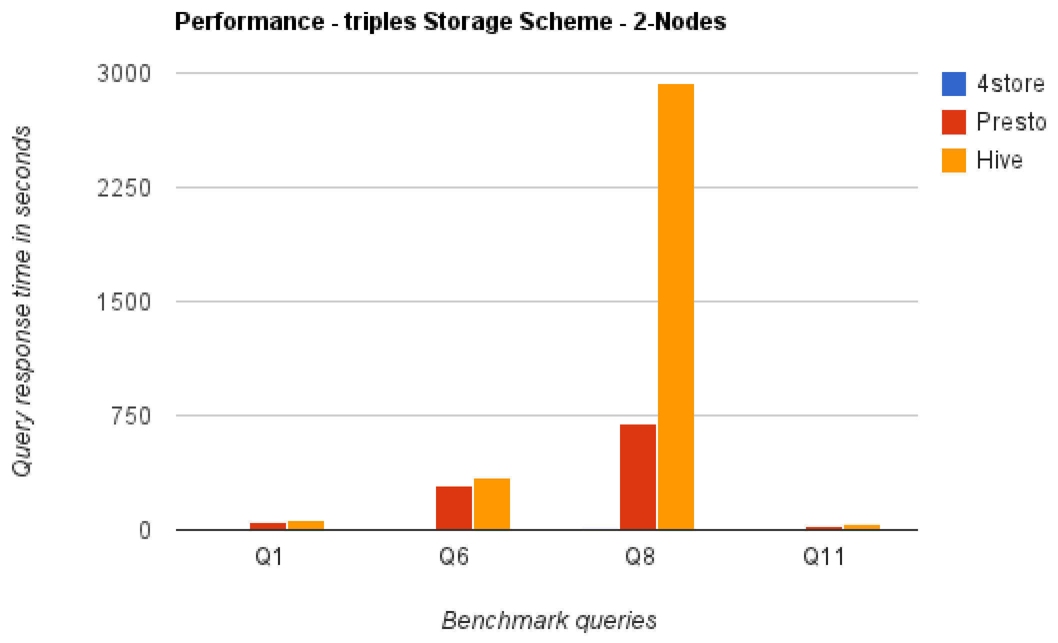

Figure 23 Query Response Time [3M Triples, 2-Node Cluster, Triples Storage Scheme]

As can be seen from the chart, the query response time for 4store is very fast compared to Presto and Hive. 4store took 0, 8, 9, and 7 seconds to respond to queries Q1, Q6, Q8, and Q11 respectively. Presto also performed much better than Hive. Hive was very slow on Query 8, which involved UNION.

The performance results over a 4-node cluster are show below in Figure 24.

[10] Simulating a 2-Node cluster simply involves shutting down two nodes out of the four nodes.

[11] Note that 4store has no partitioning scheme and was run on a single-node cluster. The results are included in the chart to give how well Presto and Hive perform in comparison to it.

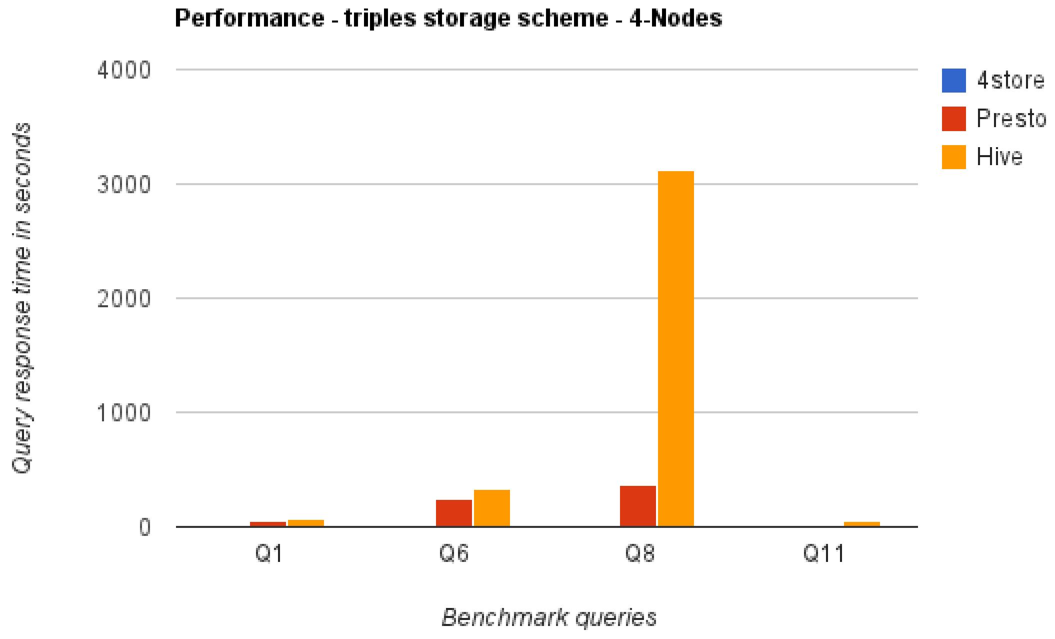

Figure 24 Query Response Time [3M Triples, 4-Node Cluster, Triples Storage Scheme]

Similar behavior was observed for Presto and Hive over a 4-node cluster - Presto was much faster than Hive. One interesting result that was observed was, as the number of nodes increase from 2 to 4, Presto showed some improvement (i.e. speed up in query response time) while Hive showed a delay - i.e. the query response time for Hive got longer when more nodes are added.

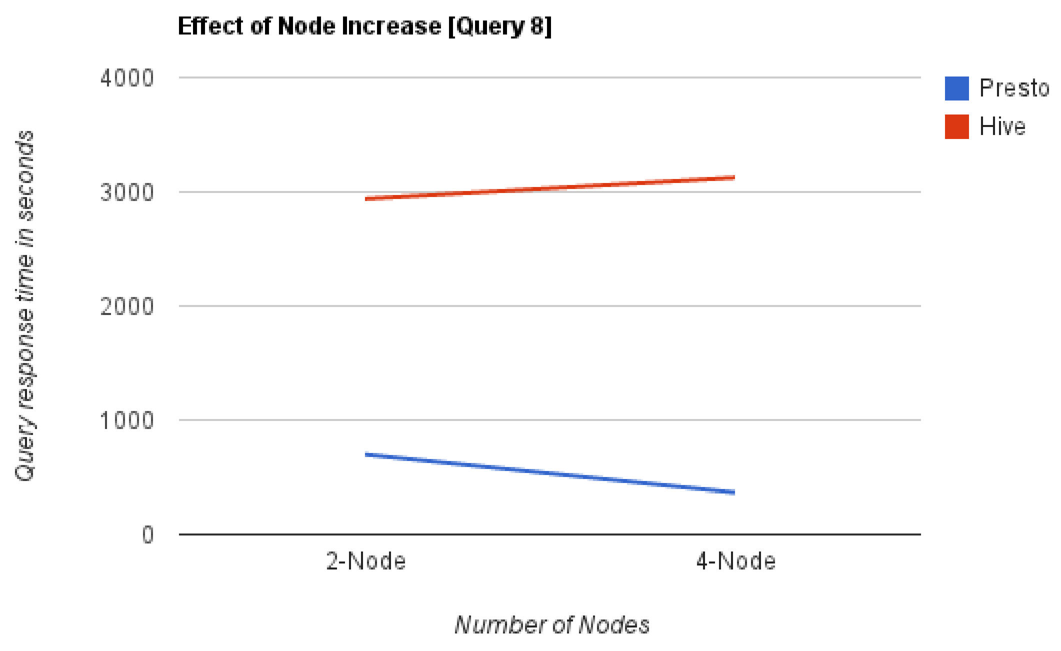

Figure 25 Effect of Node Increase on Query Performance

10.4.2 Performance Comparison - Vertical Storage Scheme

Figure 26 below show the query response time of 3M triples using the vertical storage scheme. The results are very similar to the triples storage scheme - Presto performs much faster than Hive. Moreover, for the same query, the vertical storage scheme has a better performance than the triples storage scheme.

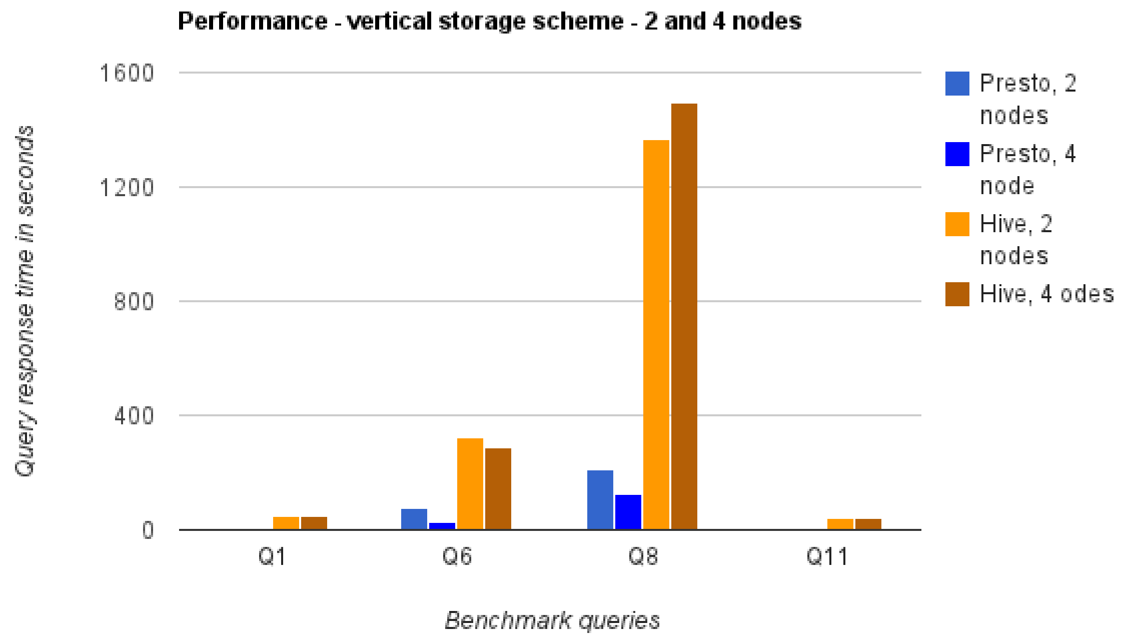

Figure 26 Presto vs. Hive using the Vertical Storage Scheme

The results for the vertical storage scheme show that, once again, Presto performed much faster than Hive. Moreover, the node increase from 2 to 4 also increased the performance of Presto but not Hive.

10.4.3 Performance Comparison - Horizontal Storage Scheme

Figure 27 shows the performance comparison of Presto and Hive for the horizontal storage scheme. Once again, Presto has a better performance than Hive. For queries that does not involve UNIONs (Q1, Q6, and Q11), the performance of Presto over Hive was not as significant as it was for the triples and vertical storage schemes.

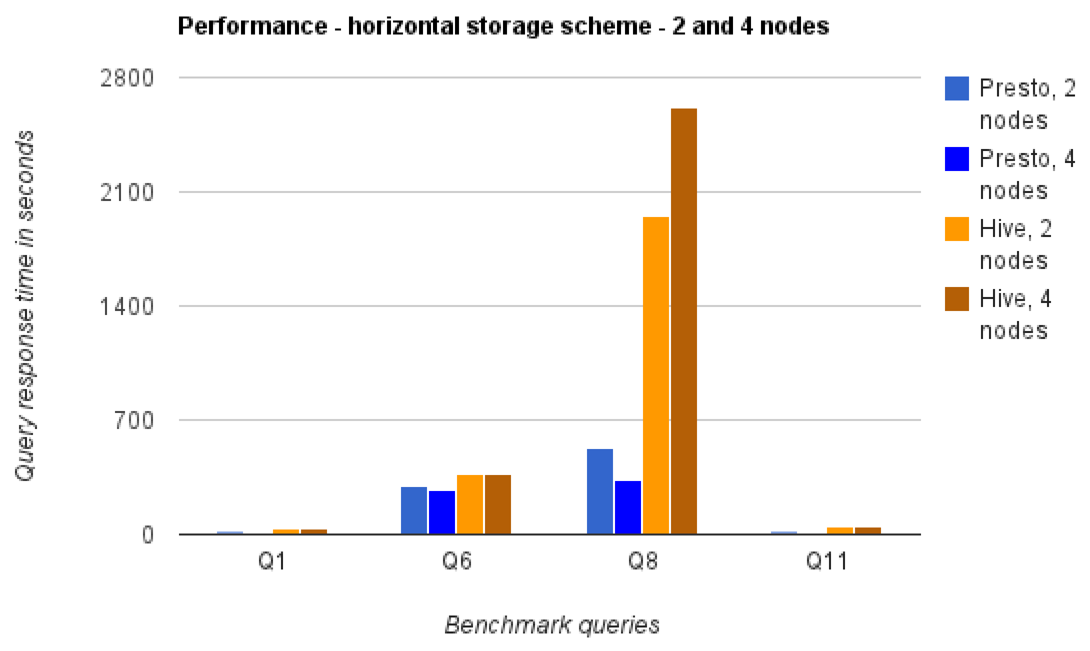

Figure 27 Presto vs. Hive using the Horizontal Storage Scheme

The results for the vertical storage scheme show that, once again, Presto performed much faster than Hive. Moreover, the node increase from 2 to 4 also increased the performance of Presto but not Hive.

10.4.4 Comparison of Triples, Vertical and Horizontal Storage Schemes

Figure 28 below shows the performance comparison of the three storage schemes for Presto and Hive over a 4 node cluster. The results indicate that the vertical storage scheme outperforms both the triples and horizontal storage schemes on both Presto and Hive. For queries Q1, Q8 and Q11, the horizontal storage scheme has a slightly better performance than the triples storage scheme.

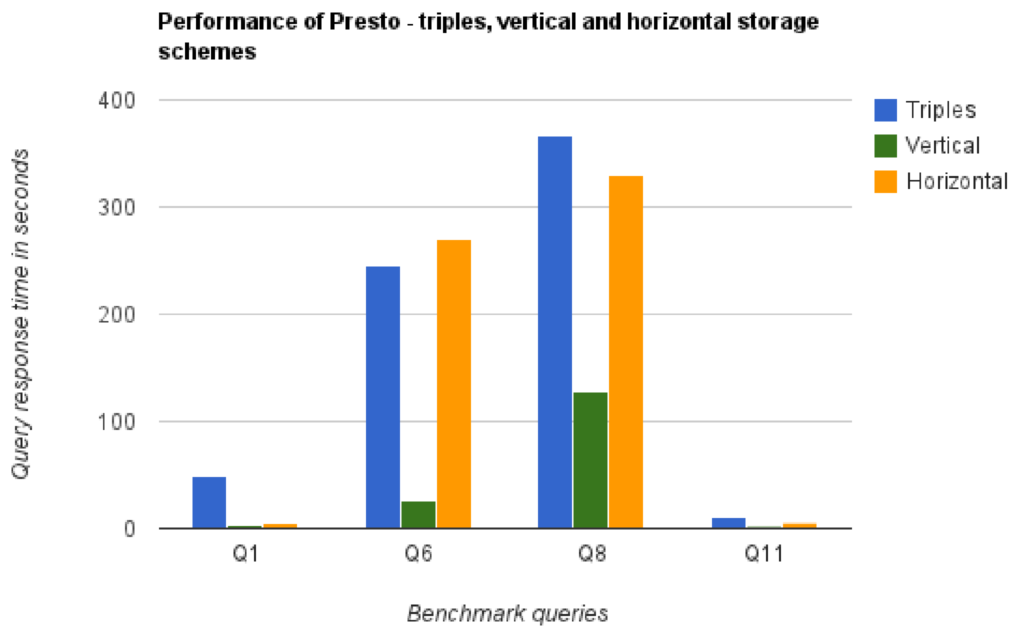

Figure 28 Performance Comparison of the Three Storage Schemes on Presto

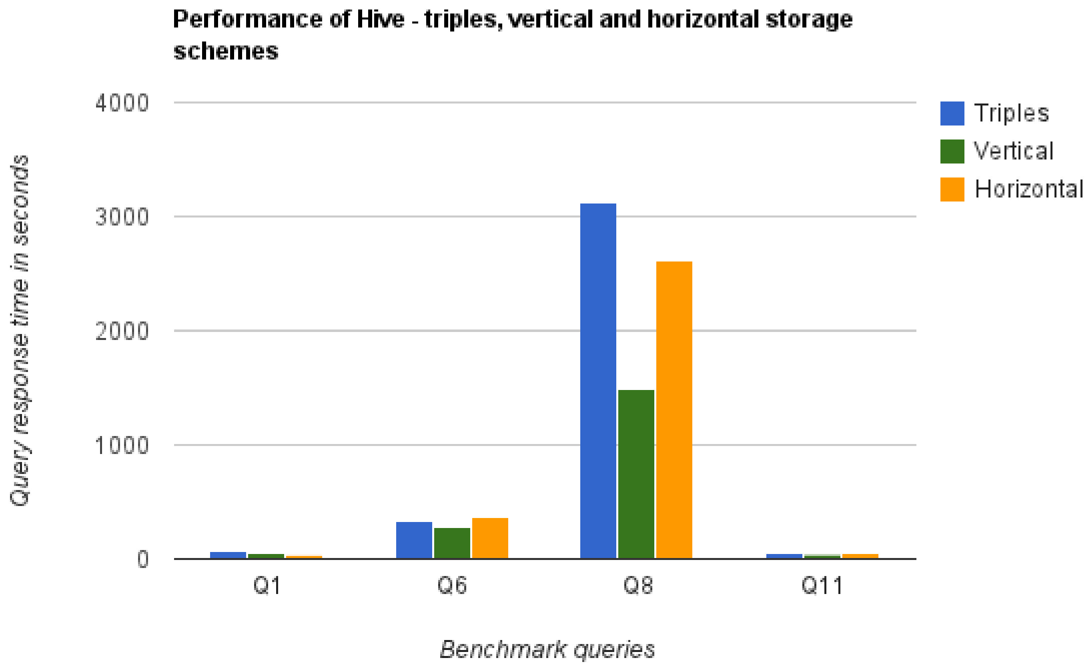

Figure 29 Performance Comparison of the Three Storage Schemes on Hive

10.5 Benchmarking Presto-RDF using 10, 20, and 30M Triples

The second experimental setup that was conducted involved setting up four and eight node clusters on Microsoft Windows Azure Platform. Each node in the cluster had a 2-core x86-64 processor, 14GB of memory, and 1TB of hard disk. Measurements were conducted for the four benchmark queries listed in section 9.1 for 10, 20, and 30 million triples.

10.5.1 Loading Time

Once the RDF dataset is copied into HDFS, the RDF-Loader will parse and run a map-reduce job to convert the raw dataset to a structured dataset based on three storage schemas – triple-store, vertical and horizontal. The results of the measurement are shown in Figure 30 below.

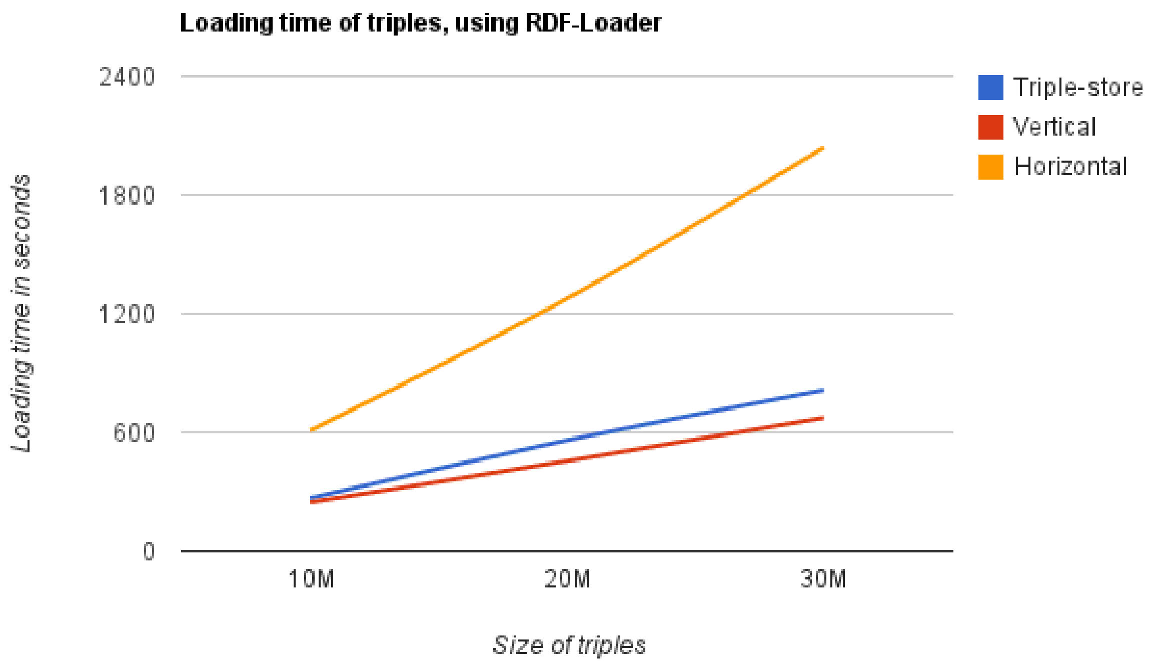

Figure 30 Loading Time of RDF Triples

The performance of the RDF-loader has a linear relationship with the size of the triples. The horizontal store map-reduce algorithm always took much longer time than the triple-store and vertical store schemes.

10.5.2 Evaluation Result for Q1

The equivalent SQL translations of SPARQL Q1 for the three storage schemes are given below.

Q1 Triple-store SQL:

```
SELECT
   T3.Object AS yr
FROM Triples T1
   JOIN Triples T2 ON T1.subject = T2.subject
   JOIN Triples T3 ON T1.subject = T3.subject
WHERE
   T1.Predicate = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'
   AND T2.Predicate = '<http://purl.org/dc/elements/1.1/title>'
   AND T3.Predicate = '<http://purl.org/dc/terms/issued>'
   AND T1.Object = '<http://localhost/vocabulary/bench/Journal>'
   AND T2.Object = '"Journal 1 (1940)"^^<http://www.w3.org/2001/XMLSchema#string>'
;
```

Q1 vertical-store SQL:

```
SELECT
   T3.Object AS yr
FROM http___www_w3_org_1999_02_22_rdf_syntax_ns_type T1
   JOIN http___purl_org_dc_elements_1_1_title T2 ON T1.subject = T2.subject
   JOIN http___purl_org_dc_terms_issued T3 ON T1.subject = T3.subject
WHERE
   T1.Object = '<http://localhost/vocabulary/bench/Journal>'
   AND T2.Object = '"Journal 1 (1940)"^^<http://www.w3.org/2001/XMLSchema#string>'
;
```

Q1 horizontal-store SQL:

```
SELECT
   T.dcterms_issued AS yr
FROM HorizontalTable T
WHERE
```

```
    T.rdf_type = '<http://localhost/vocabulary/bench/Journal>'
    AND T.dc_title = '"Journal 1 (1940)"^^<http://www.w3.org/2001/XMLSchema#string>'
;
```

The result of running the above queries on Presto for a 4-node and 8-node cluster setup are shown in the figures below.

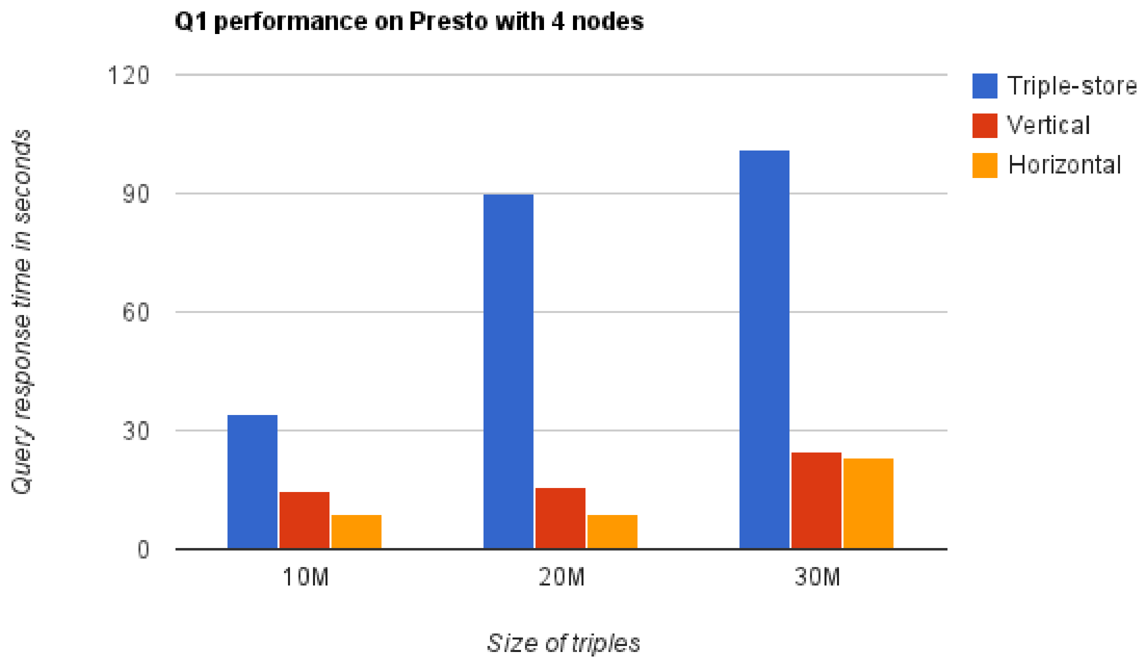

Figure 31 Performance of Q1 over a 4-Node Cluster

For Q1, the vertical and horizontal stores have a much better performance than the triple-store scheme. This can be explained by looking into the SQL translations of the vertical and horizontal storage schemes – which have lesser rows involved in JOINs. This fact remains true when the number of nodes is increased from 4 to 8 – Figure 32.

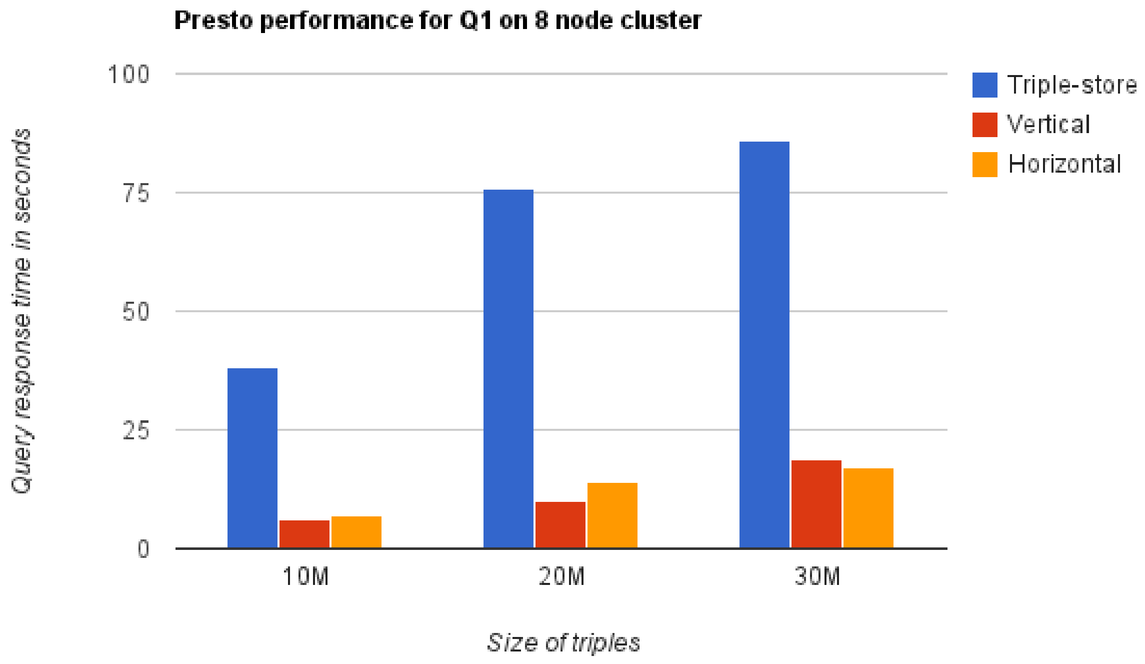

Figure 32 Performance of Q1 over 8-Node Cluster

For Q1, increasing number of nodes resulted in performance improvement for the three storage schemes. See Figure 33 below.

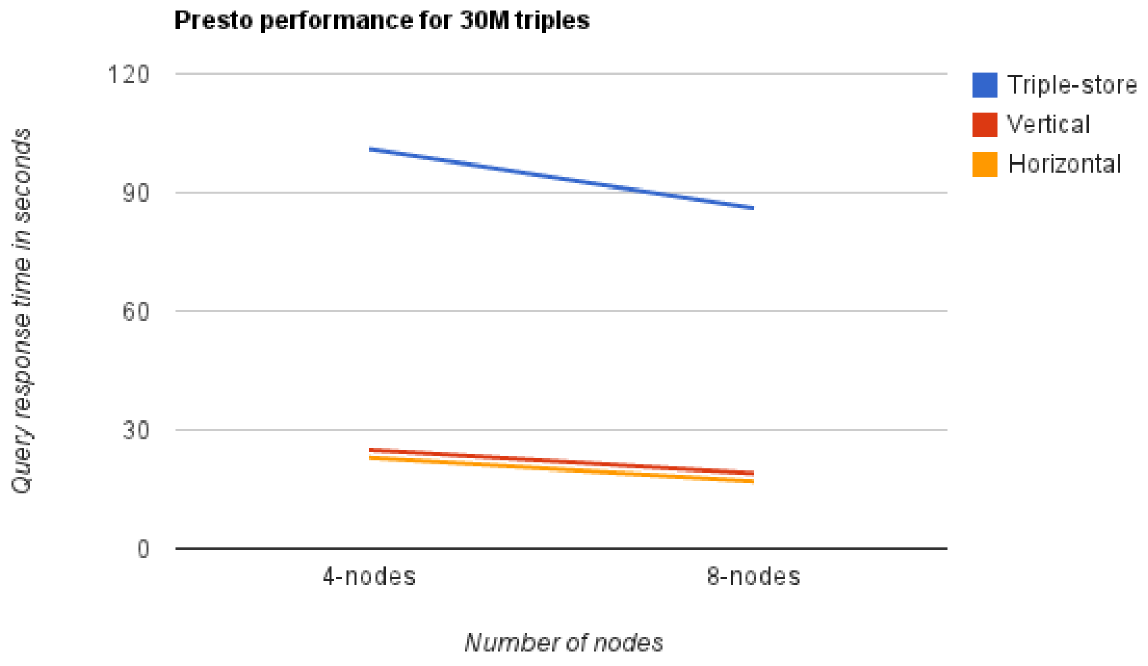

Figure 33 Effect of Node Increase on Presto for Q1

Presto vs. Hive:

Compared to Hive, Presto once again has a much higher performance. Figure 34 shows a comparison of Presto and Hive for 30M triples.

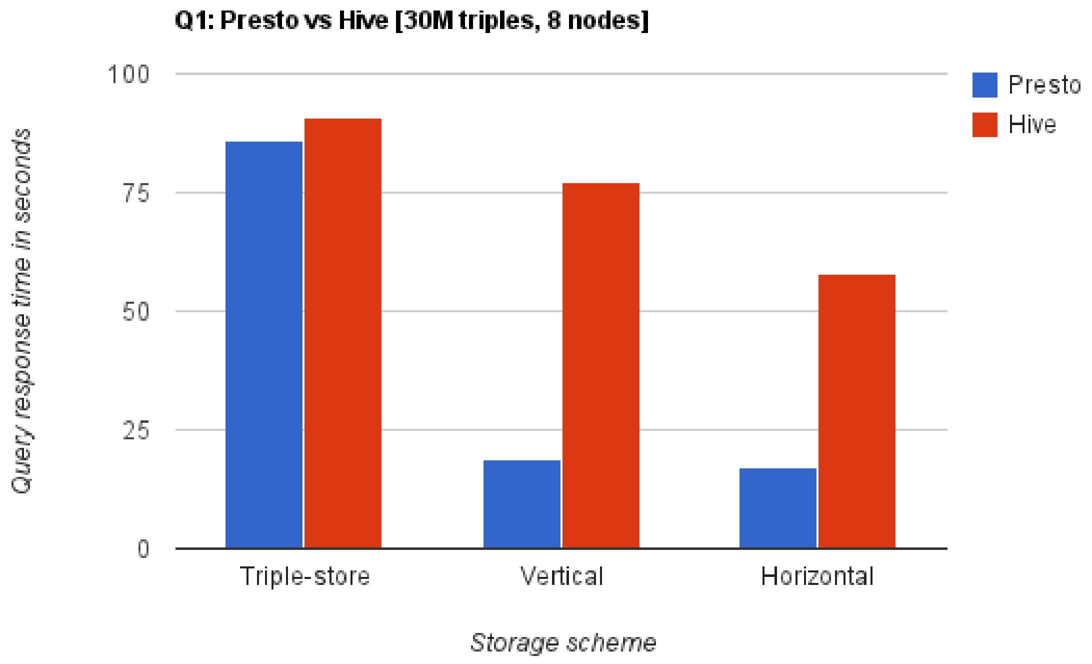

Figure 34 Q1 Performance [30M Triples over 8-Node Cluster]

10.5.3 Evaluation Result for Q6

The SQL translations for Q6, unlike Q1, involve multiple JOINs for each of the three storage schemes and are given in Appendix A. The results of the evaluation on a 4-node and 8-node cluster are shown in Figure 35 and 36 below.

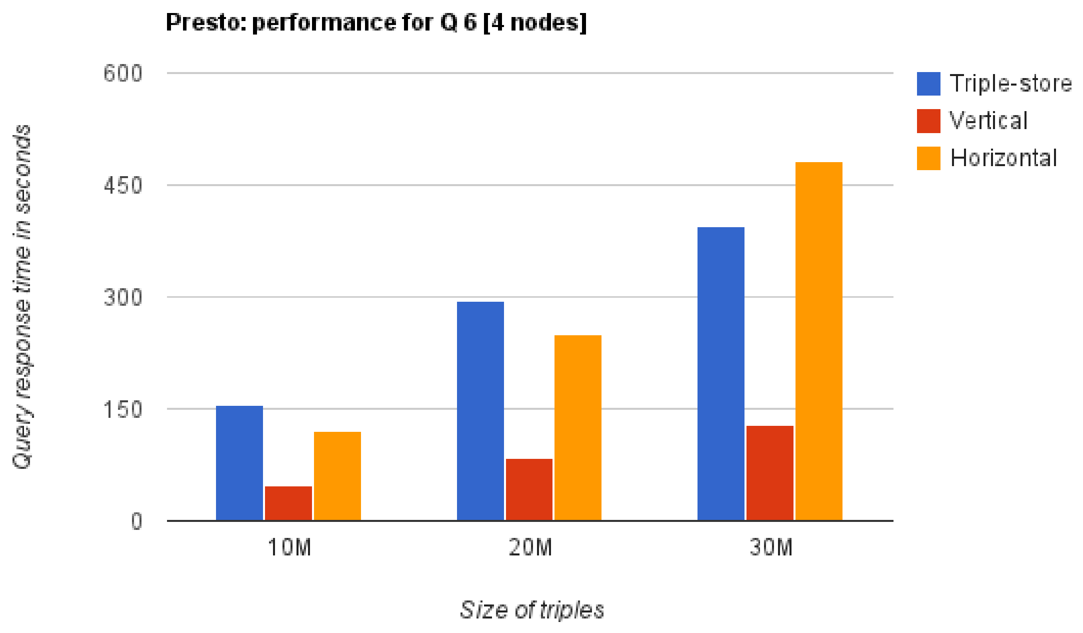

Figure 35 Q6 Performance on Presto with 4 Nodes

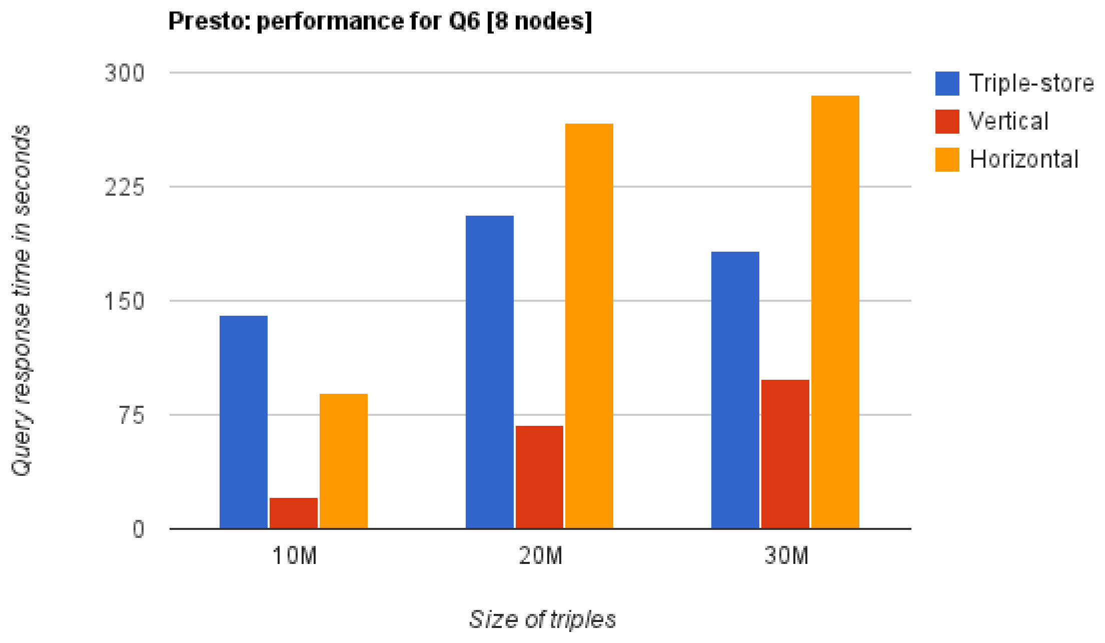

Figure 36 Q6 Performance on Presto with 8 Nodes

The results of the evaluation above (Figure 35 and 36) indicate that the performance increased with increase in the number of nodes – see Figure 37 below. The vertical store, again, has a much better performance than the triple-store and horizontal store. Unlike Q1, however, where the horizontal store had a slightly better performance than the triple-store, the triple-store in Q6 had a slightly better performance than the horizontal store, especially as the size of the triples increases. This result can be explained by the fact that the horizontal store SQL for Q6, unlike the triple-store, involves multiple selections before making JOINs.

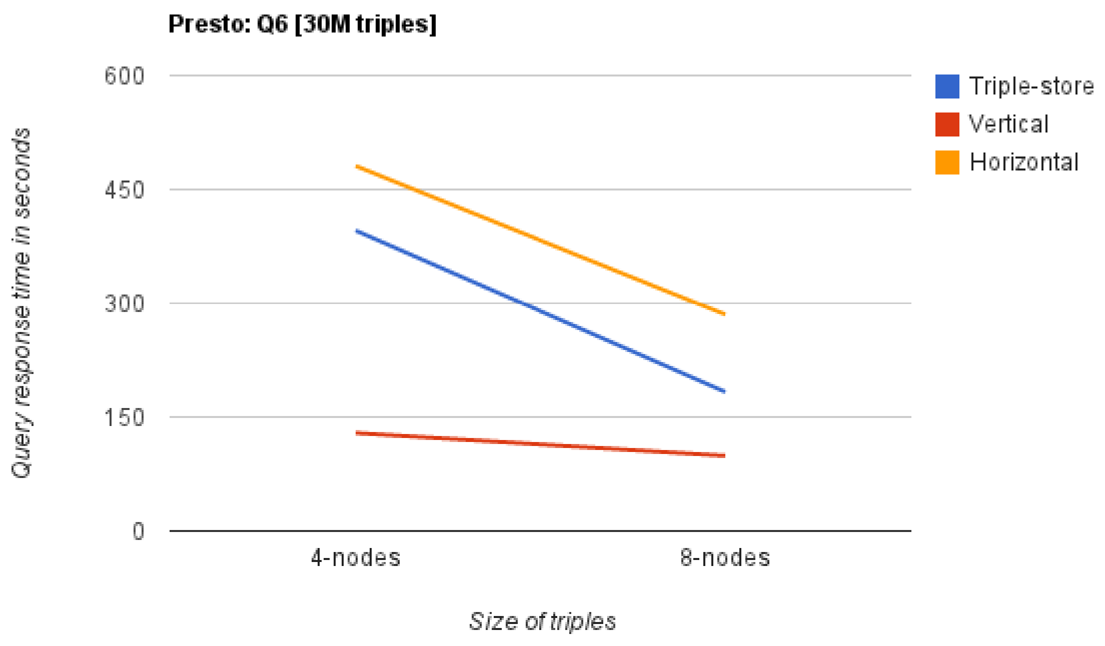

Figure 37 Effect of Node Increase on Presto for Q6 with 30M Triples

For Hive, unlike Presto, as the number of nodes was increased there was a drop in performance – which can be attributed to increase in replication across nodes and disk I/O operations – see figure 38 below.

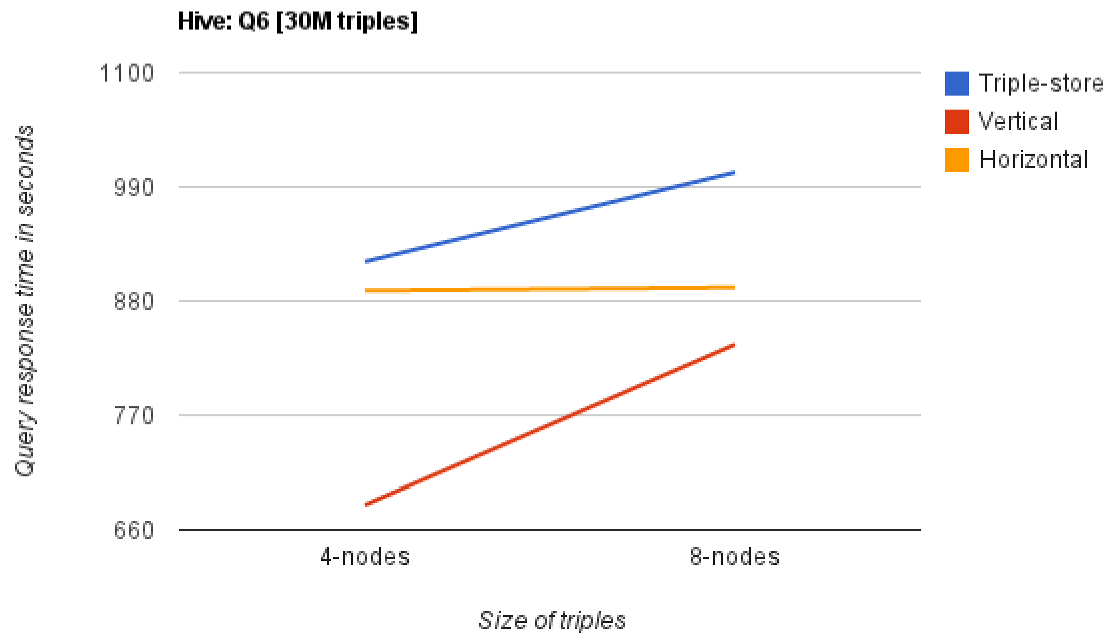

Figure 38 Effect of Node Increase on Hive for Q6 with 30M Triples

Presto vs. Hive: For Q6 as well, Presto has a much higher performance than Hive – see Figure 39 below.

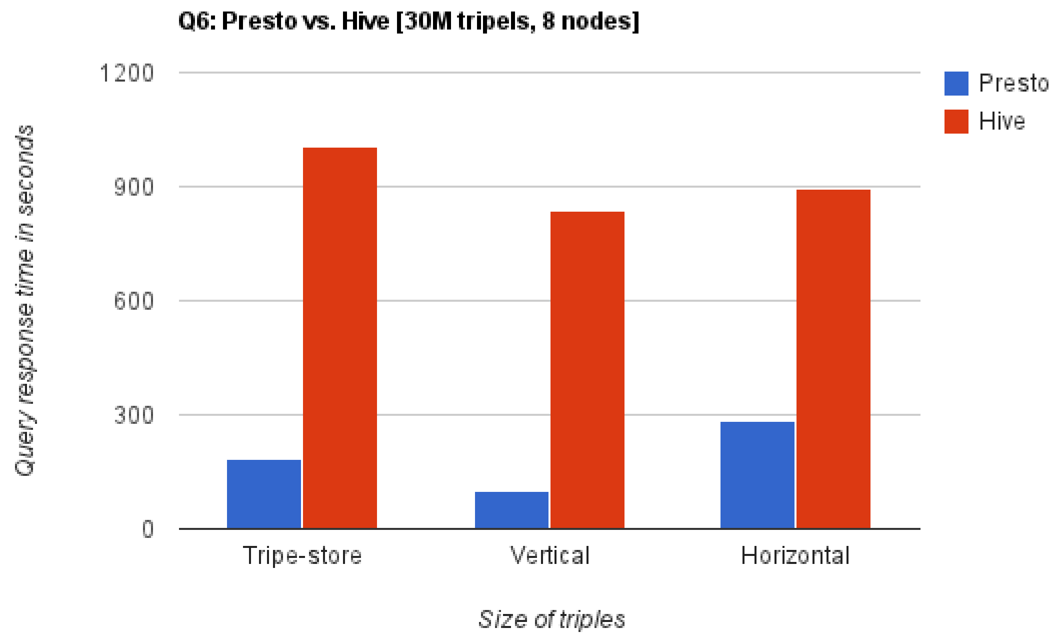

Figure 39 Presto vs. Hive for 30M Triples over 8-Node Cluster

10.5.4 Evaluation Result for Q8

The SQL translations for Q8 involve multiple JOINs (just as the case were in Q6) and a UNION – see Appendix A. The results have the same behavior as Q6 – the vertical store has a much better performance than the triple-store and horizontal stores, and Presto has a much higher performance than Hive.

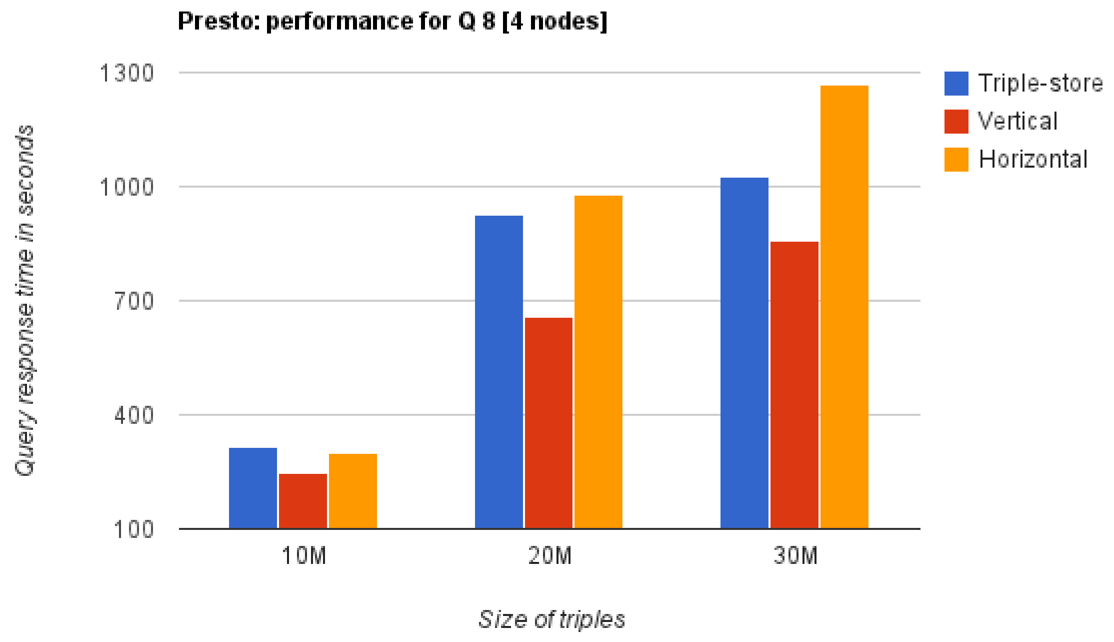

Figure 40 Q8 Performance, on Presto with 4 Nodes

10.5.5 Evaluation Result for Q11

The SQL translations for Q11 involve a simple select with an ORDER BY and LIMIT clauses.

Q11 Triple-store SQL:

```
SELECT
   T.Object AS ee
FROM
   Triples T
WHERE
   T.Predicate='<http://www.w3.org/2000/01/rdf-schema#seeAlso>'
ORDER BY ee
LIMIT 10
;
```

Q11 Vertical-store SQL:

```
SELECT
   T.Object AS ee
FROM
   http___www_w3_org_2000_01_rdf_schema_seeAlso T
ORDER BY ee
LIMIT 10
;
```

Q11 Horizontal-store SQL:

```
SELECT
   T1.rdfs_seeAlso AS ee
FROM
   HorizontalTable T1 WHERE T1.rdfs_seeAlso != 'null'
ORDER BY ee
LIMIT 10
;
```

Figure 41 below shows the results of running the above queries over 10, 20 and 30M triples.

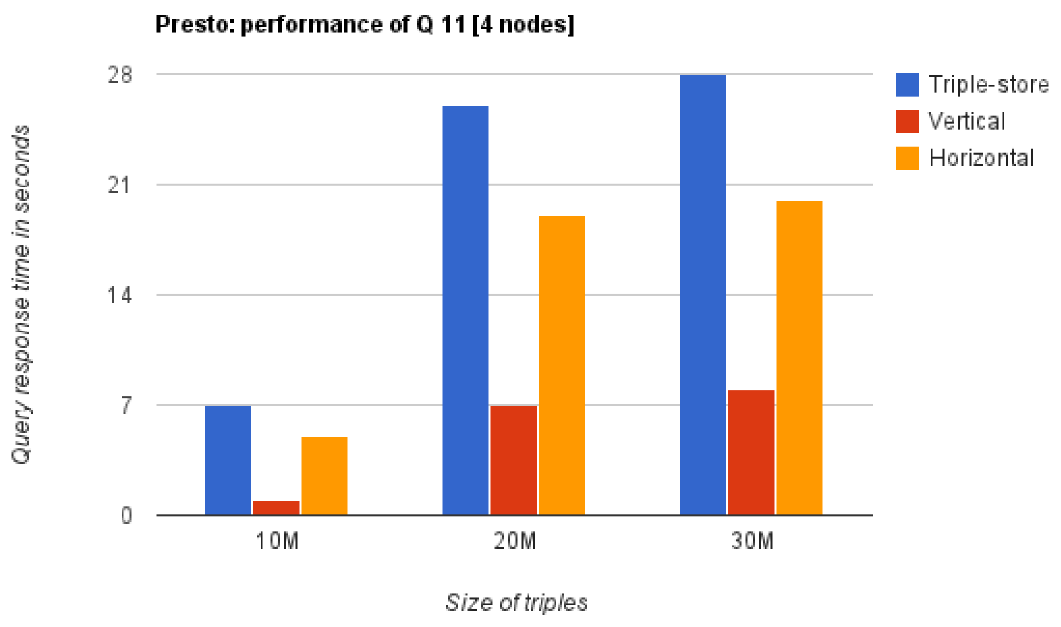

Figure 41 Q11 Performance, on Presto with 4 Nodes

Because Q11 involves just one table which has less number of rows for the vertical and horizontal storage schemes than the triple-store (which is one table), the results shown above are expected.

For 8 nodes, there is a performance improvement – see Figure 42 below.

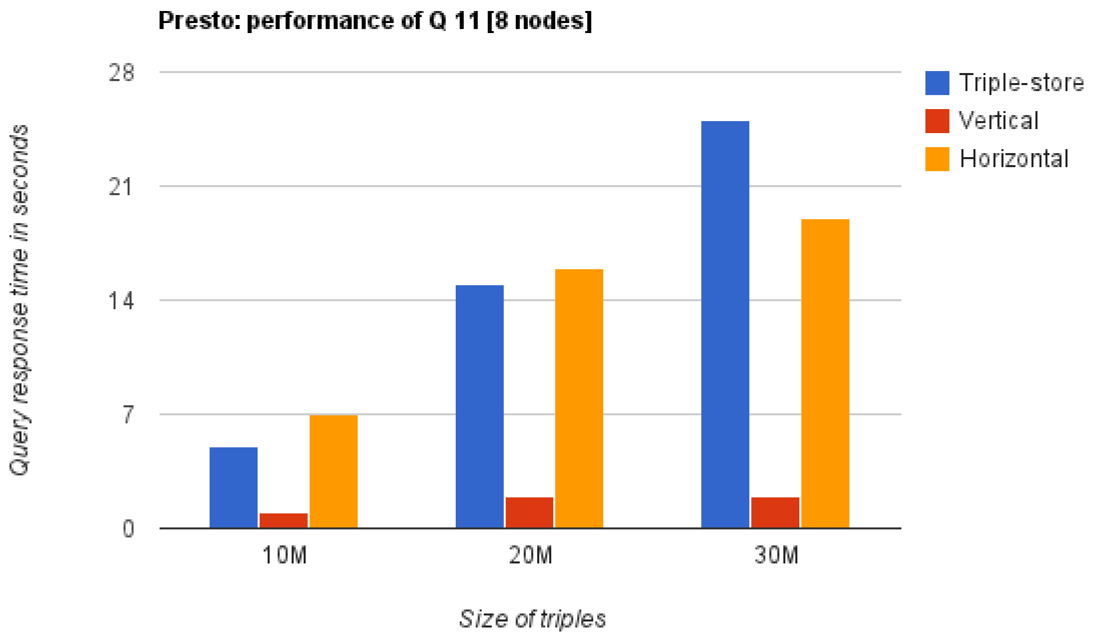

Figure 42 Q11 Performance, on Presto with 8 Nodes

Presto vs. Hive:

Compared to Hive, Presto again has a much higher performance – see Figure 43 below.

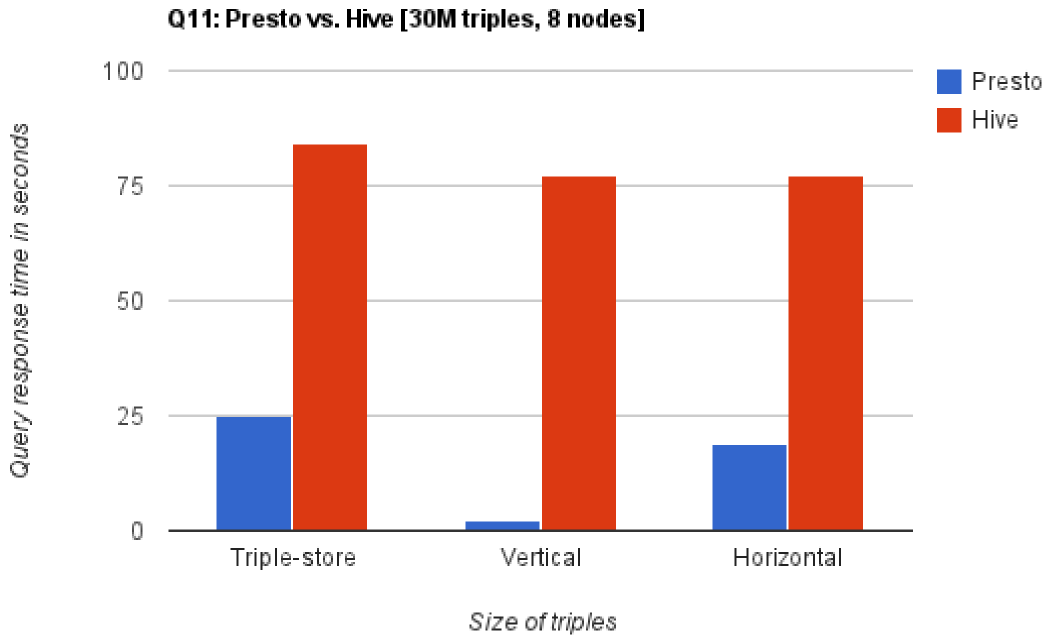

Figure 43 Presto vs. Hive, 30M Triples on 8-Node Cluster

CHAPTER 11

RELATED WORK

This chapter presents a review of selected research papers that propose and evaluate different distributed SPARQL query engines. It also presents a review of two systems, Apache Spark and Cloudera Impala, which are similar to Facebook Presto. Research papers that propose and evaluate different RDF storage schemes have already been discussed in chapters 4 and 5.

11.1 Distributed SPARQL

[56] proposes a distributed SPARQL query engine based on Jena ARQ [57]. The query engine extends Jena ARQ and makes it distributed across a cluster of machines. The extension involves re-designing some parts of Jena ARQ. Document indexing and pre-computation joins were also used to optimize the design. The results of the experiments that were conducted showed that the distributed query engine scaled well with the size of RDF data but its overall performance was very poor. The query engine, unlike Facebook Presto, uses MapReduce.

Marcello Leida et al. [58] propose a query processing architecture that can be used to efficiently process RDF graphs that are distributed over a local data grid. The architecture has no single point of failure and no specialized nodes – which is a different than Hadoop. The paper proposes a sophisticated non-memory query planning and execution algorithm based on streaming RDF triples. Presto uses a distributed in-memory query processing algorithm.

Xin Wang et al. [59] discuss how the performance of a distributed SPARQL query processing can be optimized by applying methods from graph theory. They propose a Minimum-Spanning-Tree-based (MST-based) algorithm for distributed SPARQL processing. The results of their experiment show that a distributed SPARQL processing engine based on MST-based algorithms performs much better than other non-graph traversal algorithms. Because this thesis translates a

SPARQL query into its equivalent SQL query, the query optimization that is done by Presto is for the SQL query and not for the SPARQL query.

[60] proposes a distributed RDF query processing engine based on a message passing. The engine uses in-memory data structures to store indices for data blocks and dictionaries. Just like Presto, the query processing engine avoids disk I/O operations. The authors experimented their design over several types of SPARQL queries and were able to get a significant performance gain (as compared to Hadoop).

11.2 Apache Spark and Cloudera Impala

Apache Spark [61] and Cloudera Impala [62] are two open-sources systems that are very similar to Facebook Presto. Both Apache Spark and Cloudera Impala offer in-memory processing of queries over a cluster of machines.

According to Apache, Apache Spark is a “fast and general engine for large-scale data processing”. Spark uses advanced Directed Acyclic Graph (DAG) execution engine with cyclic data flow and in-memory processing to run programs up to 100 (for in-memory processing mode) or 10 times faster (for disk processing mode) than Hadoop MapReduce [61]. Apache Spark became an Apache top-level project in February 2014.

Cloudera Impala is an open-source massively parallel processing (MPP) engine for data stored in HDFS. Cloudera Impala is based on Cloudera’s Distribution for Hadoop (CDH) and benefits from Hadoop’s key features – scalability, flexibility, and fault tolerance. Cloudera Impala, just like Presto, uses Hive Metastore to store the metadata information of directories and files in HDFS [62]. Cloudera Impala became available in May 2013.

CHAPTER 12

CONCLUSION AND FUTURE WORK

This thesis presented a comparative analysis of big RDF data using Presto, which uses in-memory query processing engine, and Hive, which uses MapReduce to evaluate SQL queries. The thesis also proposed a Presto-based architecture, Presto-RDF, that can be used to store and process big RDF data.

12.1 Conclusion

From the experiments conducted, the following conclusions can be drawn:

- 4store has a much higher performance than Presto and Hive for small data sets. For bigger data sets (10M, 20M and 30M triples), however, 4store was simply unable to process the data and crashed. This is true when Presto, Hive and 4store are all tested with single-node setups.
- For all queries, Presto has a much higher performance than Hive.
- The vertical storage scheme has a consistent performance advantage than both the triple-store or horizontal storage schemes.
- As the size of data increases, the horizontal storage scheme performed relatively better than the triple-store scheme. This is unlike the articles reviewed during this thesis, which ignore the horizontal scheme as being not efficient (because it has many null values).
- Increasing the number of nodes improved query performance in Presto but not in Hive. This can be explained by the fact that Hive replicates data across clusters and does IO operations – which increase as the size of nodes increase.

12.2 Hypotheses Revisited

Revisiting the hypotheses from Chapter 1, we can conclude:

- As the size of RDF data increases to big-data levels, RDF stores based on Hadoop outperform native RDF stores like 4store.

  This is a true hypothesis if qualified like the below:

  As the size of RDF data increases to big-data levels, RDF stores based on Hadoop outperform native RDF store, 4store, for a single-node setup.

- Distributed in-memory query processing engines deliver faster response time on big RDF datasets than query processing engines that rely on MapReduce.

  This hypothesis holds true.

- Vertical partitioning scheme for RDF data gives better performance than other RDF storage schemes on Hive.

  This hypothesis holds true.

- Increasing number of processing nodes dramatically improves query performance.

  This hypothesis, as stated, and according to the experiments conducted, is false. But it holds true for Presto if the qualification "dramatically" is removed.

12.3 Contributions

This thesis is unique in the following respects:

- It uses a distributed in-memory query execution model, based on Presto, to evaluate the performance of SPARQL queries over big RDF data.
- It demonstrates the use of a SPARQL to SQL compiler based on Flex and Bison. The compiler is also unique in that it generates SQL for the three storage schemes discussed in this thesis – triple-store, vertical and horizontal.

- Publishes the result of the query performance of a horizontal storage scheme, which had a better performance than the triple-store as the size of data increases. No published results were found on the horizontal storage scheme during the literature review.
- The RDF-Loader component of Presto-RDF uses map-reduce to load RDF data into the different storage structures and the implementation is available.

12.4 Future Work

There are a number of areas that a future researcher can work on to extend this thesis:

- This thesis used a single benchmark, SP^{2}Bench, which has a limitation of being "not realistic". Hence, one can extend the work by experimenting on different benchmarks based on the proposal made in Chapter 6, section 5.
- There are different optimization techniques that can be applied to the three storage schemas as well as to the RDF data directly. In this thesis, the RDF data is stored as a text file, which is not optimal. A researcher can test using RCFILE, ORC, and AVRO formats, which are better optimized than text file.
- Presto is an open-source project whose implementation can be download for free GitHub. https://github.com/facebook/presto. In this thesis, a SPARQL to SQL compiler was built because Presto does not support SPARQL. A project that extends Presto to have a direct support for SPARQL would be nice.

REFERENCES

[1] T. Berners-Lee, J. Hendler, and O. Lassila. “The Semantic Web”, May 2001

[2] Nigel Shadbolt, Wendy Hall, Tim Berners-Lee "The Semantic Web Revisited", 2007

[3] linkeddata.org, "Linked Data - Connect Distributed Data across the Web", 2014

[4] Dean Allemang, "Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL", May 2008

[5] Ora Lassila, Ralph R. Swick, "Resource Description Framework (RDF) Model and Syntax Specification No. REC-rdf-syntax-19990222.", 1999

[6] D. Brickley, R. V. Guha, "RDF Vocabulary Description Language 1.0: RDF Schema", February 2004

[7] J.A. Hendler, “Frequently Asked Questions on W3C’s Web Ontology Language (OWL),” W3C, 2004; www.w3.org/2003/08/owlfaq

[8] T. Berners-Lee, R.T.Fielding, and L.Masinter, "Uniform Resource Identifier (URI: Generic Syntax", IETF RFP 3986 (standards strack), Internet Eng. Task Force, Jan. 2005: www.ietf.org/rfc/rfc3986.txt

[9] World Wide Web Consortium (W3C), "Using Qualified Names (QNames) as Identifiers in XML Content", TAG Finding 17 March 2004, http://www.w3.org/2001/tag/doc/qnameids-2004-03-17

[10] World Wide Web Consortium (W3C), "RDF Schema 1.1, W3C Recommendation" 25 February 2014, http://www.w3.org/TR/rdf-schema/

[11] World Wide Web Consortium (W3C), "Media Types Issues for Text RDF Formats", January 2008, http://www.w3.org/2008/01/rdf-media-types

[12] World Wide Web Consortium (W3C), "RDF/XML Syntax Specification (Revised)", W3C Recommendation 10 February 2004, http://www.w3.org/TR/REC-rdf-syntax/

[13] World Wide Web Consortium (W3C), "Turtle – Terse RDF Triple Language". 10 July 2012. Retrieved 20 November 2012.

[14] World Wide Web Consortium (W3C), "SPARQL Query Language for RDF", 2006

[15] IBM, "What is big data?", http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html, 2014

[16] Microsoft, "The Big Bang: How the Big Data Explosion is Chaning the World", April 2013

[17] John Weathington, "Big Data defined", September 2012

[18] Edd Dumbill, "What is big data? An introduction to the big data landscape", January 2012

[19] Bernard Marr, "Big Data - The 5 Vs Everyone Must Know", Feb 28, 2014

[20] Apache Hadoop Project, "What is Apache Hadoop?" , hadoop.apache.org, 2014

[21] Apache Hadoop Project, "HDFS Architecture Guide", 2008

[22] Dhruba Borthakur, "The Hadoop Distributed File System: Architecture and Design", 2008

[23] Jeffery Dean, Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Communications of the ACM, 2008

[24] M.A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient rdf store over a relational database," in Proceedings of the 2013 international conference on Management of data. ACM, 2013, pp. 121–132.

[25] 4store.org, "4store, an efficient, scalable and stable rdf database," http://www.4store.org/, 2014

[26] K. Wilkinson, C. Sayers, H. A. Kuno, D. Reynolds et al., "Efficient rdf storage and retrieval in jena2." in SWDB, vol. 3, 2003, pp. 131–150

[27] Jeen Broekstra, Arjohn Kampman, Frank van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema", June 2002

[28] Stephen Harris, Nicholas Gibbins, "3store: Efficient Bulk RDF Storage", 2003

[29] Albert Haque, Lynette Perkins, "Distributed RDF Triple Store Using HBase and Hive", December 2012

[30] AllegroGraph, "Semantic Graph Technologies", http://franz.com/agraph/allegrograph, 2014

[31] Kiyoshi Nitaa, Iztok Savnik, "Survey of RDF storage Managers", 2010

[32] Y. Theoharis, V. Christophides, and G. Karvounarakis. "Benchmarking database representations of RDF/S stores”. In Proc. of ISWC, 2005

[33] David C. Faye, Olivier Cure, Guillaume Blin. "A survey of RDF storage approaches". February 2012

[34] Philippe Cudre-Mauroux, LLiya Enchev, Sever Fundatureanu, Paul Groth, Albert Haque, Andreas Harth, Felix Leif Keppmann, Daniel Miranker, Juan F. Sequeda, Marcin Wylot. "NoSQL Databases for RDF: An Empirical Evaluation", 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II

[35] Nancy Lynch and Seth Gilbert, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”, ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59.

[36] Gunter Ladwig and Andreas Harth. "CumulusRDF: Linked Data Management on Nested Key-Value Stores", 2011

[37] F. Chang, J. Dean, S. Ghemawat, WC Hsieh "Bigtable: A distributed system for structured data", ACM Transactions on Computer Systems archive Volume 26 Issue 2, June 2008

[38] A. hetrapal and V. Ganesh, “Hbase and hypertable for large scale distributed storage systems,” http://www.uavindia.com/ankur/downloads/HypertableHBaseEval2.pdf, 2006

[39] Ian Robinson and Jim Webber, "Graph Databases", June 20, 2013

[40] Andreas Harth, Katja Hose Ralf Schenkel, "Linked Data Management", 2014

[41] Sherif Sakr and Ghazi Al-Naymat, "Relational Processing of RDF Queries: A Survey", 2009

[42] Yongming Luo, Fran¸cois Picalausa, George H.L. Fletcher, Jan Hidders, and Stijn Vansummeren. Storing and indexing massive RDF datasets. Springer Berlin Heidelberg, 2012

[43] Daniel J. Abadi, Adam Marcus and Samuel R. Madden "Scalable Semantic Web Data Management Using Vertical Partitioning". 2007

[44] DBpedia, http://www.dbpedia.org, 2014

[45] Mohamed Morsey, Jens Lehmann, Soren Auer, and Axel-Cyrille Ngona Ngomo, "DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data", Springer, 2011

[46] Yuanbo Guo, Zhengxiang Pan, Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems", October 2005

[47] The Lehigh University Benchmark (LUBM), http://swat.cse.lehigh.edu/projects/lubm, 2014

[48] Christian Bizer and Andreas Schultz, "The Berlin SPARQL Benchmark", 2009

[49] The Berlin SPARQL Benchmark, http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/, 2014

[50] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. In Proceedings of the 25th International Conference on Data Engineering, pages 222–233, Shanghai, 2009.

[51] SP2Bench, http://dbis.informatik.uni-freiburg.de/forschung/projekte/SP2B/, 2014

[52] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. "Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets" 2011.

[53] Facebook, "Presto: Interacting with petabytes of data at Facebook", https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920, 2014

[54] Apache Hive Project, "Apache Hive", http://hive.apache.org, 2014

[55] Thusoo,A., Sarma, J.s., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. "Hive, a warehousing solution over a Map-Reduce framework", 2009

[56] Prasad Kulkarni “Distributed SPARQL query engine using MapReduce”, Univerisity of Edinburgh, 2010.

[57] ARQ - A SPRQL Processor for Jena, http://jena.apache.org/documentation/query/, 2014

[58] Marcello Leida, Andrej Chu, "Distributed SPARQL query answering over RDF data streams", IEEE International Congress on Big Data, 2013

[59] Xin Wang, Thanassis Tiropanis, and Hugh C. Davis, "Evaluating Graph Traversal Algorithms for Distributed SPARQL Query Optimization", Springer-Verlag Berlin Heidelberg 2012

[60] Arnab Kumar Dutta "A Distributed In-Memory SPARQL QUery Processor based on Message Passing", Universitat des Saarlandes Max-Planck-Institut fur Informatik, July 2012

[61] Apache Software Foundation, https://spark.apache.org/, 2014

[62] Cloudera, http://www.cloudera.com, 2014

APPENDIX A

SQL TRANSLATIONS OF THE BENCHMARK QUERIES

Q1 – Triple-store

```
SELECT
   T3.Object AS yr
FROM Triples T1
   JOIN Triples T2 ON T1.subject = T2.subject
   JOIN Triples T3 ON T1.subject = T3.subject
WHERE
   T1.Predicate = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'
   AND T2.Predicate = '<http://purl.org/dc/elements/1.1/title>'
   AND T3.Predicate = '<http://purl.org/dc/terms/issued>'
   AND T1.Object = '<http://localhost/vocabulary/bench/Journal>'
   AND T2.Object = '"Journal 1 (1940)"^^<http://www.w3.org/2001/XMLSchema#string>'
;
```

Q6 – Triple-store

```
SELECT
   X.yr AS yr,
   X.name AS name,
   X.document AS document
FROM (SELECT
      T1.subject AS class,
      T2.Subject AS document,
      T3.Object AS yr,
      T4.Object AS author,
      T5.Object AS name
   FROM Triples T1
      JOIN Triples T2 ON T1.subject = T2.object
      JOIN Triples T3 ON T3.subject = T2.subject
      JOIN Triples T4 ON T4.subject = T3.subject
      JOIN Triples T5 ON T5.subject = T4.object
   WHERE
      T1.Predicate = '<http://www.w3.org/2000/01/rdf-schema#subClassOf>'
      AND T2.Predicate = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'
      AND T3.Predicate = '<http://purl.org/dc/terms/issued>'
      AND T4.Predicate = '<http://purl.org/dc/elements/1.1/creator>'
      AND T5.Predicate = '<http://xmlns.com/foaf/0.1/name>'
      AND T1.Object = '<http://xmlns.com/foaf/0.1/Document>') X
   LEFT JOIN (SELECT
      T1.subject AS class,
      T2.Subject AS document,
      T3.Object AS yr,
      T4.Object AS author
   FROM Triples T1
```

```
        JOIN Triples T2 ON T1.subject = T2.object
        JOIN Triples T3 ON T3.subject = T2.subject
        JOIN Triples T4 ON T4.subject = T3.subject
    WHERE
        T1.Predicate = '<http://www.w3.org/2000/01/rdf-schema#subClassOf>'
        AND T2.Predicate = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'
        AND T3.Predicate = '<http://purl.org/dc/terms/issued>'
        AND T4.Predicate = '<http://purl.org/dc/elements/1.1/creator>'
        AND T1.Object = '<http://xmlns.com/foaf/0.1/Document>') Y ON X.author = Y.author
;
```

Q8 – Triple-store

```
SELECT DISTINCT
    name
FROM Triples T1
    JOIN Triples T2 ON T1.subject = T2.subject
    JOIN (SELECT
        name,
        erdoes
    FROM (SELECT
            T5.Object AS name,
            T3.object AS erdoes
        FROM Triples T3
            JOIN Triples T4 ON T3.subject = T4.subject
            JOIN Triples T5 ON T4.object = T5.subject
        WHERE
            T3.Predicate = '<http://purl.org/dc/elements/1.1/creator>'
            AND T4.Predicate = '<http://purl.org/dc/elements/1.1/creator>'
            AND T5.Predicate = '<http://xmlns.com/foaf/0.1/name>'
            AND NOT T3.object = T4.object) X
    UNION ALL
    SELECT
        T7.Object AS name,
        T3.object AS erdoes
    FROM Triples T3
        JOIN Triples T4 ON T3.subject = T4.subject
        JOIN Triples T5 ON T4.object = T5.object
        JOIN Triples T6 ON T5.subject = T6.subject
        JOIN Triples T7 ON T6.object = T7.subject
    WHERE
        T3.Predicate = '<http://purl.org/dc/elements/1.1/creator>'
        AND T4.Predicate = '<http://purl.org/dc/elements/1.1/creator>'
        AND T5.Predicate = '<http://purl.org/dc/elements/1.1/creator>'
        AND T6.Predicate = '<http://purl.org/dc/elements/1.1/creator>'
        AND T7.Predicate = '<http://xmlns.com/foaf/0.1/name>'
        AND NOT T4.object = T3.object
        AND NOT T5.subject = T3.subject
        AND NOT T6.object = T3.object
        AND NOT T4.object = T6.object) Y ON T2.subject = Y.erdoes
WHERE
    T1.Predicate = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'
    AND T2.Predicate = '<http://xmlns.com/foaf/0.1/name>'
    AND T1.Object = '<http://xmlns.com/foaf/0.1/Person>'
    AND T2.Object = '"Paul Erdoes"^^<http://www.w3.org/2001/XMLSchema#string>'
;
```

Q11 – Triple-store

```
SELECT
    T.Object AS ee
FROM
    Triples T
WHERE
    T.Predicate='<http://www.w3.org/2000/01/rdf-schema#seeAlso>'
ORDER BY ee
LIMIT 10
;
```

Q1 – Vertical Scheme

```
SELECT
    T3.Object AS yr
FROM http___www_w3_org_1999_02_22_rdf_syntax_ns_type T1
    JOIN http___purl_org_dc_elements_1_1_title T2 ON T1.subject = T2.subject
    JOIN http___purl_org_dc_terms_issued T3 ON T1.subject = T3.subject
WHERE
    T1.Object = '<http://localhost/vocabulary/bench/Journal>'
    AND T2.Object = '"Journal 1 (1940)"^^<http://www.w3.org/2001/XMLSchema#string>'
;
```

Q6 – Vertical Scheme

```
SELECT
    X.yr AS yr,
    X.name AS name,
    X.document AS document
FROM (SELECT
        T1.Subject AS class,
        T2.Subject AS document,
        T3.Object AS yr,
        T4.Object AS author,
        T5.Object AS name
    FROM http___www_w3_org_2000_01_rdf_schema_subClassOf T1
        JOIN http___www_w3_org_1999_02_22_rdf_syntax_ns_type T2 ON T1.subject =
T2.object
        JOIN http___purl_org_dc_terms_issued T3 ON T3.subject = T2.subject
        JOIN http___purl_org_dc_elements_1_1_creator T4 ON T4.subject = T3.subject
        JOIN http___xmlns_com_foaf_0_1_name T5 ON T4.object = T5.subject
    WHERE
        T1.Object = '<http://xmlns.com/foaf/0.1/Document>') AS X
    LEFT JOIN (SELECT
        T1.Subject AS class,
        T2.Subject AS document,
        T3.Object AS yr,
        T4.Object AS author
    FROM http___www_w3_org_2000_01_rdf_schema_subClassOf T1
        JOIN http___www_w3_org_1999_02_22_rdf_syntax_ns_type T2 ON T1.subject =
T2.object
        JOIN http___purl_org_dc_terms_issued T3 ON T3.subject = T2.subject
        JOIN http___purl_org_dc_elements_1_1_creator T4 ON T4.subject = T3.subject
    WHERE
        T1.Object = '<http://xmlns.com/foaf/0.1/Document>') AS Y ON X.author = Y.author
;
```

Q8 – Vertical Scheme

```
SELECT DISTINCT
    name
FROM http___www_w3_org_1999_02_22_rdf_syntax_ns_type T1
    JOIN http___xmlns_com_foaf_0_1_name T2 ON T1.subject = T2.subject
    JOIN (SELECT
        name,
        erdoes
    FROM (SELECT
            T3.Object AS name,
            T4.object AS erdoes
        FROM http___purl_org_dc_elements_1_1_creator T4
            JOIN http___purl_org_dc_elements_1_1_creator T5 ON T4.subject = T5.subject
            JOIN http___xmlns_com_foaf_0_1_name T3 ON T5.object = T3.subject
        WHERE
            NOT T4.object = T5.object) AS L
    UNION ALL
    SELECT
        T3.Object AS name,
        T4.object AS erdoes
    FROM http___purl_org_dc_elements_1_1_creator T4
        JOIN http___purl_org_dc_elements_1_1_creator T5 ON T4.subject = T5.subject
        JOIN http___purl_org_dc_elements_1_1_creator T6 ON T5.object = T6.object
        JOIN http___purl_org_dc_elements_1_1_creator T7 ON T6.subject = T7.subject
        JOIN http___xmlns_com_foaf_0_1_name T3 ON T7.object = T3.subject
    WHERE
        NOT T5.object = T4.object
        AND NOT T6.subject = T4.subject
        AND NOT T7.object = T4.object
        AND NOT T5.object = T7.object) AS R ON T2.subject = R.erdoes
WHERE
    T1.Object = '<http://xmlns.com/foaf/0.1/Person>'
    AND T2.Object = '"Paul Erdoes"^^<http://www.w3.org/2001/XMLSchema#string>'
;
```

Q11 – Vertical Scheme

```
SELECT
    T.Object AS ee
FROM
    http___www_w3_org_2000_01_rdf_schema_seeAlso T
ORDER BY ee
LIMIT 10
;
```

Q1 – Horizontal Scheme

```
SELECT
    T.dcterms_issued AS yr
FROM HorizontalTable T
WHERE
    T.rdf_type = '<http://localhost/vocabulary/bench/Journal>'
    AND T.dc_title = '"Journal 1 (1940)"^^<http://www.w3.org/2001/XMLSchema#string>'
;
```

Q6 – Horizontal Scheme

```
SELECT
    X.yr AS yr,
    X.name As name,
    X.document As document
```

```
FROM
   (
      SELECT T4.dc_creator As author, T3.dcterms_issued AS yr, T5.foaf_name AS name,
T1.rdfs_subClassOf As document
      FROM
         (SELECT Subject, rdfs_subClassOf FROM HorizontalTable WHERE rdfs_subClassOf =
'<http://xmlns.com/foaf/0.1/Document>') T1
         JOIN (SELECT Subject, rdf_type FROM HorizontalTable WHERE rdf_type != 'null') T2
ON T2.rdf_type = T1.Subject
         JOIN (SELECT Subject, dcterms_issued FROM HorizontalTable WHERE
dcterms_issued != 'null') T3 ON T3.Subject = T2.Subject
         JOIN (SELECT Subject, dc_creator FROM HorizontalTable WHERE dc_creator !=
'null') T4 ON T4.Subject = T3.Subject
         JOIN (SELECT Subject, foaf_name FROM HorizontalTable WHERE foaf_name !=
'null') T5 ON T5.Subject = T4.dc_creator
   ) X
   LEFT JOIN
   (
      SELECT T4.dc_creator AS author, T3.dcterms_issued AS yr
      FROM
         (SELECT Subject, rdfs_subClassOf FROM HorizontalTable WHERE rdfs_subClassOf =
'<http://xmlns.com/foaf/0.1/Document>') T1
         JOIN (SELECT Subject, rdf_type FROM HorizontalTable WHERE rdf_type != 'null') T2
ON T2.rdf_type = T1.Subject
         JOIN (SELECT Subject, dcterms_issued FROM HorizontalTable WHERE
dcterms_issued != 'null') T3 ON T3.Subject = T2.Subject
         JOIN (SELECT Subject, dc_creator FROM HorizontalTable WHERE dc_creator !=
'null') T4 ON T4.Subject = T3.Subject
   ) Y
   ON X.author = Y.author
;
```

Q8 – Horizontal Scheme

```
SELECT DISTINCT
   name
FROM
   (SELECT Subject, rdf_type FROM HorizontalTable WHERE rdf_type =
'<http://xmlns.com/foaf/0.1/Person>') T1
   JOIN (SELECT Subject, foaf_name FROM HorizontalTable WHERE foaf_name = '"Paul
Erdoes"^^<http://www.w3.org/2001/XMLSchema#string>') T2 ON T1.Subject = T2.Subject
   JOIN
   (
      SELECT
         name,
            erdoes
        FROM
        (
            SELECT
                T7.foaf_name AS name,
                T3.dc_creator AS erdoes
            FROM
                (SELECT Subject, dc_creator FROM HorizontalTable WHERE dc_creator != 'null')
T3
                JOIN (SELECT Subject, dc_creator FROM HorizontalTable WHERE dc_creator !=
'null') T4 ON T4.Subject = T3.Subject
                JOIN (SELECT Subject, dc_creator FROM HorizontalTable WHERE dc_creator !=
'null') T5 ON T5.dc_creator = T4.dc_creator
                JOIN (SELECT Subject, dc_creator FROM HorizontalTable WHERE dc_creator !=
'null') T6 ON T6.Subject = T5.Subject
                JOIN (SELECT Subject, foaf_name FROM HorizontalTable WHERE foaf_name !=
'null') T7 ON T7.Subject = T6.dc_creator
            WHERE
                T4.dc_creator != T3.dc_creator AND T5.Subject != T3.Subject AND T6.dc_creator !=
T3.dc_creator AND T6.dc_creator != T4.dc_creator
        ) X
        UNION ALL
            SELECT
                T10.foaf_name AS name,
                T8.dc_creator As erdoes
            FROM
                (SELECT Subject, dc_creator FROM HorizontalTable WHERE dc_creator != 'null')
T8
                JOIN (SELECT Subject, dc_creator FROM HorizontalTable WHERE dc_creator !=
'null') T9 ON T9.Subject = T8.Subject
                JOIN (SELECT Subject, foaf_name FROM HorizontalTable WHERE foaf_name !=
'null') T10 ON T10.Subject = T9.dc_creator
            WHERE
                T9.dc_creator != T8.dc_creator
    ) Y ON T2.Subject = Y.erdoes
;
```

Q11 – Horizontal Scheme

```
SELECT
    T1.rdfs_seeAlso AS ee
FROM
    HorizontalTable T1 WHERE T1.rdfs_seeAlso != 'null'
```

```
ORDER BY ee
LIMIT 10
;
```

APPENDIX B

LEX FILE FOR THE RQ2SQL COMPILER

```
/*****************************************************************
*
*       Tool:           Sparql to Sql Compiler
*       version:        1.0
*       Author:         Mulugeta Mammo
*       Date:           September 2014
*
*****************************************************************/
%{
#include <cstdio>
#include <iostream>
#include "rq2sql.tab.h"
%}

%option case-insensitive

%%

"SELECT"                        return T_SELECT;
"DISTINCT"                      return T_DISTINCT;
"WHERE"                         return T_WHERE;
"PREFIX"                        return T_PREFIX;
"FILTER"                        return T_FILTER;
"UNION"                         return T_UNION;
"OPTIONAL"                      return T_OPT;
"ORDER BY"                      {
                                        yylval.s = strdup(yytext);
                                        return T_ORDER_BY;
                                }
"ASC"                           {
                                        yylval.s = strdup(yytext);
                                        return T_ASC;
                                }
"DESC"                          {
                                        yylval.s = strdup(yytext);
                                        return T_DESC;
                                }
"LIMIT"                         {
                                        yylval.s = strdup(yytext);
                                        return T_LIMIT;
                                }
"@PREFIX"                       return T_AT_PREFIX;
"bound"                         return T_BOUND;
"{"                             return T_LB;
"}"                             return T_RB;
```

```
"("                                     return T_LP;
")"                                     return T_RP;
";"                                     return T_SEMICOLON;
"="                                     return T_EQU;
"!"                                     return T_NOT;
"!="                                    return T_NEQ;
"&&"                                    return T_AND;
"||"                                    return T_OR;
"<"                                     return T_LT;
"<="                                    return T_LTE;
">"                                     return T_GT;
">="                                    return T_GTE;
[1-9][0-9]*                             {
                                                yylval.i = atoi(yytext);
                                                return T_INT;
                                        }
[a-z][a-z0-9_]*                         {
                                                yylval.s = strdup(yytext);
                                                return T_IDT;
                                        };
\?[a-z][a-z0-9_]*                       {
                                                yylval.s = strdup(yytext);
                                                return T_VAR;
                                        };
[a-z][a-z0-9]*":"[a-z0-9_]+             {
                                                yylval.s = strdup(yytext);
                                                return T_SURI;
                                        }
[a-z][a-z0-9]*":"                       {
                                                yylval.s = strdup(yytext);
                                                        return T_SYM;
                                        }
\<http[^\>]*\>                          {
                                                yylval.s = strdup(yytext);
                                                return T_URI;
                                        };

\"(\\.|[^"])*\"\^\^[^ ]*                {
                                                yylval.s = strdup(yytext);
                                                return T_VALUE;
                                        };

\"(\\.|[^"])*\"                         {
                                                yylval.s = strdup(yytext);
                                                return T_CSTR;
```

```
                                    };
"\n"                                {
                                            yylineno++;
                                    };
\.                                  return T_DOT;
[ \t]+                              {
                                            /* ignore white space */
                                    };


"#".*                               {
                                            /* ignore comments */
                                    };
[ \t\v\f\r]+                        {
                                    };
                                    {
                                            std::cerr << "Lexical Error!\n";
                                    };
%%

int yywrap() {
        return 1;
}
```

APPENDIX C

GRAMMER FILE FOR THE RQ2SQL COMPILER

```
/*******************************************************************
*
*       Tool:           Sparql to SQL Compiler
*       version:        1.0
*       Author:         Mulugeta Mammo
*       Date:           September 2014
*
********************************************************************/

%{

#include <iostream>
#include <fstream>
#include <sstream>
#include <map>
#include <string>
#include <vector>
#include <stack>
#include <utility>
#include <algorithm>
#include <regex>
#include "definition.h"
#include "utility.h"
#include "vertical.h"
#include "horizontal.h"
#include "triples.h"

using namespace std;

int num_errors = 0;             // We report the # of syntax errors
const int T_TOK = 0; // Triple ID
const int F_TOK = -1;// Filter ID

// Bison's
extern FILE* yyin;
extern int yylineno;
extern int yylex();
extern int yyparse();
extern void yyerror(const char*);

%}

%error-verbose
%union {
	int i;
	char* s;
}

%token<s> T_PREFIX T_AT_PREFIX;
%token<s> T_IDT T_VAR T_URI T_CSTR T_INT T_SURI T_VALUE T_NOT;
%token<s> T_SELECT T_DISTINCT T_WHERE T_FILTER T_LIMIT T_BOUND T_SYM
T_OPT T_UNION;
%token<s> T_LP T_RP T_LB T_RB T_COLON T_SEMICOLON T_DOT;
%token<s> T_AND T_OR T_EQU T_NEQ T_LT T_LTE T_GT T_GTE;
%token<s> T_ORDER_BY T_ASC T_DESC;
%type<s> primary unary_expr or_expr and_expr equality_expr relational_expr stmt stmts;

%%

query:
	body
	| decl body
	;

decl:
	prefix
	| decl prefix
	;

prefix:
	prefix_keyword T_SYM T_URI line_end
	{
		string s($<s>2);
		prefixes[s.substr(0, s.size() - 1)] = $<s>3;
	}
	;

prefix_keyword:
	T_PREFIX
	| T_AT_PREFIX
	;

line_end:
	| T_DOT
	;

body:
	T_SELECT distinct select T_WHERE T_LB stmts T_RB order_by limit
	;
```

```
distinct:
	| T_DISTINCT
	{
		distinct_stmt = "DISTINCT ";
	}
	;


select:
	T_VAR
	{
		select_list.push_back($<s>1);
	}
	| select T_VAR
	{
		select_list.push_back($<s>2);
	}
	;

stmts:
	stmt
	| stmts stmt
	| stmts T_OPT T_LB stmts T_RB
	{
		cerr << "This production version of rq2sql does not support code generation for
optionals." << endl;
	}
	| stmts T_LB stmts T_RB T_UNION T_LB stmts T_RB
	{
		cerr << "This version of rq2sql does not support code generation for unions." <<
endl;
	}
	| T_LB stmts T_RB T_UNION T_LB stmts T_RB
	{
		cerr << "This version of rq2sql does not support code generation for unions." <<
endl;
	}
	;

stmt:
	subject predicate object line_end
	{
		query_index.push_back(make_pair(T_TOK, subjects.size() - 1));
	}
```

```
        | T_FILTER T_LP expr T_RP line_end
        {
                string e($<s>3);
                for (int i = 0; i != predicates.size(); ++i) {

                        if (subjects[i].second == "var") {
                                replace(e, subjects[i].first, rq_vars[subjects[i].first]);
                        }

                        if (objects[i].second == "var") {
                                replace(e, objects[i].first, rq_vars[objects[i].first]);
                        }

                }

                filters.push_back(e);
                query_index.push_back(make_pair(F_TOK, filters.size() - 1));
        }
        ;

expr:
        or_expr
        ;

or_expr:
        and_expr
        | or_expr T_OR and_expr
        {
                $$ = cat_str($<s>1, " || ", $<s>3);
        }
        ;

and_expr:
        equality_expr
        | and_expr T_AND equality_expr
        {
                $$ = cat_str($<s>1, " && ", $<s>3);
        }
        ;

equality_expr:
        relational_expr
        | equality_expr T_EQU relational_expr
        {
                $$ = cat_str($<s>1, " = ", $<s>3);
```

```
          }
          | equality_expr T_NEQ relational_expr
          {
                    $$ = cat_str($<s>1, " != ", $<s>3);
          }
          ;

relational_expr:
          unary_expr
          | relational_expr T_LT unary_expr
          {
                    $$ = cat_str($<s>1, " < ", $<s>3);
          }
          | relational_expr T_LTE unary_expr
          {
                    $$ = cat_str($<s>1, " <= ", $<s>3);
          }
          | relational_expr T_GT unary_expr
          {
                    $$ = cat_str($<s>1, " > ", $<s>3);
          }
          | relational_expr T_GTE unary_expr
          {
                    $$ = cat_str($<s>1, " >= ", $<s>3);
          }
          ;

unary_expr:
          primary
          | T_NOT primary
          ;

primary:
          T_VAR
          {
                    string s = rq_vars[$<s>1];
                    replace_quote(s);
                    char* str = (char*) malloc(strlen(s.c_str()) + 1);
                    strcpy(str, s.c_str());
                    $$ = str;
          }
          | T_VALUE
          | T_CSTR
          {
                    string s($<s>1);
```

```
                replace_quote(s);
                char* str = (char*) malloc(strlen(s.c_str()) + 1);
                strcpy(str, s.c_str());
                $$ = str;
        }
        | T_INT
        {
                string s(itos($<i>1));
                char* str = (char*) malloc(strlen(s.c_str()) + 1);
                strcpy(str, s.c_str());
                $$ = str;
        }
        | T_LP expr T_RP
        {
                cerr << "This version of rq2sql does not support nested filter expressions." <<
endl;
                exit(1);
        }
        | T_BOUND T_LP T_VAR T_RP
        {
                string s = " != null";
                char* str = (char*) malloc(strlen(s.c_str()) + strlen($<s>3) + 1);
                strcpy(str, $<s>3);
                strcat(str, s.c_str());
                $$ = str;
        }
        ;

subject:
        T_VAR
        {
                rq_vars[$<s>1] = "T" + itos(subjects.size()) + ".Subject";
                rq_vars_index[$<s>1] = subjects.size();
                subjects.push_back(make_pair($<s>1, "var"));
        }
        | T_URI
        {
                subjects.push_back(make_pair($<s>1, "uri"));
        }
        | T_SURI
        {
                string result = "";
                expand_uri(result, prefixes, $<s>1);
                subjects.push_back(make_pair(result, "suri"));
        }
```

```
	;

predicate:
	T_VAR
	{
		cerr << "This version of rq2sql does not support variables at the predicate
position." << endl;
		exit(1);
	}
	| T_URI
	{
		string p($<s>1);
		format_predicate(p);
		predicates.push_back(make_pair(p, "uri"));
	}
	| T_SURI
	{
		string result = "";
		expand_uri(result, prefixes,  $<s>1);
		format_predicate(result);
		predicates.push_back(make_pair(result, "suri"));
	}
	;

object:
	T_VAR
	{
		rq_vars[$<s>1] = "T" + itos(objects.size()) + ".Object";
		rq_vars_index[$<s>1] = objects.size();
		objects.push_back(make_pair($<s>1, "var"));
	}
	| T_URI
	{
		objects.push_back(make_pair($<s>1, "uri"));
	}
	| T_CSTR
	{
		string s($<s>1);
		replace_quote(s);
		objects.push_back(make_pair(s, "val"));
	}
	| T_VALUE
	{
		objects.push_back(make_pair($<s>1, "val"));
	}
```

```
        | T_SURI
        {
                string result = "";
                expand_uri(result, prefixes,  $<s>1);
                objects.push_back(make_pair(result, "suri"));
        }
        ;

sort_option:
        {
                char* str = (char*) malloc(strlen("ASC") + 1);
                strcpy(str, "ASC");
                $<s>$ = str;
        }
        | T_ASC
        | T_DESC
        ;

order_by:
        | T_ORDER_BY T_VAR sort_option
        {
                char* str = (char*) malloc(strlen($<s>1) + strlen($<s>2) + strlen($<s>3) + 1);
                strcpy(str, $<s>1);
                strcat(str, " ");
                string s($<s>2);
                strcat(str, rq_vars[s].c_str());
                string s2($<s>3);
                strcat(str, rq_vars[s2].c_str());

                order_by_stmt = str;
                $<s>$ = str;
        }
        ;

limit:
        | T_LIMIT T_INT
        {
                const char* istr = itos($<i>2).c_str();
                char* str = (char*) malloc(strlen($<s>1) + strlen(istr) + 1);
                strcpy(str, $<s>1);
                strcat(str, " ");
                strcat(str, istr);

                limit_stmt = str;
                $<s>$ = str;
        }
        ;

%%

int main(int argc, char** argv)
{

        if (argc < 4) {
                cerr << "Error: Invalid command. Usage:rq2sql --storage [triples | vertical |
horizontal] input.rq\n";
                return 1;
        }

        string arg1 = argv[1];
        if (arg1 != "--storage") {
                cerr << "Error: Invalid command. Usage: rq2sql --storage [triples | vertical |
horizontal] input.rq\n";
                return 1;
        }


        string storage = argv[2];
        if (storage != "triples" && storage != "vertical" && storage != "horizontal") {
                cerr << "Error: Invalid storage. Options: triples, vertical and horizontal.\n";
                return 1;
        }

        string input = argv[3];
        if (input.substr(input.length() - 2) != "rq") {
                cerr << "Error: " << argv[3] << " is not a valid rq file.\n";
                return 1;
        }

        yyin = fopen(input.c_str(), "r");
        if (yyin == NULL)
        {
                cerr << "Error: " << input << " does not exist.\n";
                return 1;
        }

        cout << "Compiling:\t" << input << "...\n";

        // parse sparql query
        //
        yyparse();

        // generate code based on parse result and storage type
        //
        string sql_str = "";
        if (storage == "vertical")
                sql_str = get_vertical_sql();
        else if (storage == "triples")
                sql_str = get_triples_sql();
        else if (storage == "horizontal")
                sql_str = get_horizontal_sql();

        string output = input;
        output = output.substr(0, output.length() - 3).append("_" + storage + ".sql");

        ofstream ofs(output);
        ofs << sql_str.c_str() << endl;

        cout << "Errors:\t\t" << num_errors << "\n";
        if (num_errors) {
                cout << "Generated:\tNothing\n";
                exit(1);
        }

        cout << "Generated:\t" << output << "\n";

        return 0;
}

void yyerror(const char* str)
{
        cerr << "Line " << yylineno << ": " << str << "\n";
        yyclearin;
        ++num_errors;
}
```