Distributed RDF Storage and Querying Using In-Memory Processing Engine

by

P M Mahmudul Hassan

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2021 by the
Graduate Supervisory Committee:

Srividya Bansal, Chair
Ajay Bansal
Hasan Davulcu
Mohamed Sarwat Abdelghany Aly Elsayed

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

The proliferation of semantic data in the form of RDF (Resource Description Framework) triples demands an efficient, scalable, and distributed storage along with a highly available and fault-tolerant parallel processing strategy. There are three open issues with distributed RDF data management systems that are not well addressed altogether in existing work. First is the querying efficiency, second is that solutions are optimized for certain types of query patterns and don't necessarily work well for all types, and third is concerned with reducing pre-processing cost. Therefore, the rapid growth of RDF data raises the need for an efficient partitioning strategy over distributed data management systems to improve SPARQL (SPARQL Protocol and RDF Query Language) query performance regardless of its pattern shape with minimized pre-processing overhead.

In this context, the first contribution of this work is a distributed RDF data partitioning schema called 3CStore that extends the existing VP (Vertical Partitioning) approach by using a subset of triples from the VP tables based on different join correlations. This approach speeds up queries at the cost of additional pre-processing overhead. To solve this, a relational partitioning schema called VPExp was developed by splitting predicates based on explicit type information of objects. This approach gains a significant query performance only for the specific type of query where the object is bound to a value for a particular predicate. To get efficient query performance on a wide range of query patterns, an improved solution is proposed by extending the existing Property Table approach to Subset-Property Table and combined with the VP approach. Further investigation on distributed RDF processing and querying systems based on

typical use cases led to a novel relational partitioning schema called PTP (Property Table Partitioning) that further partitions the whole Property Table into the number of unique properties to minimize query input size and join operations during query evaluation. Finally, an RDF data management system based on the SPARQL-over-SQL approach called S3QLRDF is developed that generates the optimal query execution plan using statistics of PTP tables to provide efficient SPARQL query processing on a distributed system.

DEDICATION

To my family for their love and support.

ACKNOWLEDGMENTS

I am extremely grateful to my advisor Dr. Srividya Bansal for bringing me into this challenging but rewarding journey as a Ph.D. student and helping me with valuable advice in research and support in life. Dr. Srividya Bansal's objective and comprehensive way of doing research deeply influenced and taught me how to participate in critical thinking when confronted with real-world problems. I would also like to mention my gratitude to my committee member, Dr. Ajay Bansal, Dr. Hasan Davulcu, and Dr. Mohamed Sarwat for their constructive criticism, helpful suggestions, and support in the process of my Ph.D. dissertation defense. I am also grateful to Microsoft Azure and Google Cloud Platform for Research sponsorship to run my experiments using their Cloud services.

TABLE OF CONTENTS

Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER 1

INTRODUCTION

The Semantic Web introduced by Tim Berners-Lee in 2001 with the goal to associate meaning with the data on the Web to automate the consumption by machines for exploiting the wealth of data on the Web through meaningful processing. It uses a graph data model called Resource Description Framework (RDF[1]) for data interchange on the Web and Simple Protocol and RDF Query Language (SPARQL[2]) for searching data defined in the RDF format. Every single year RDF data is growing on a web scale, posing great challenges for efficient storage and query processing that can scale well with the volume of data.

1.1 Motivation

The Web of linked data, Semantic Web, has evolved from a "Web of Documents" to an open inter-linked "Web of Data", provides machine-processable data for the consumption of software agents to understand the semantics presented by web documents. The Semantic Web is expressed in the form of RDF data model proposed by W3C is the standard to represent metadata about Web resources. Recently RDF has gained the popularity for its flexible data model, which is used for publishing data on the Web through a number of applications and use cases in many areas such as social networks, commercial search engines, public knowledge bases, and databases. Top search engine providers – Google, Bing, Yahoo!, and Yandex have agreed to create a protocol (*schema.org*) for a structure data vocabulary in order to define entities, actions, and relationships through the internet which makes search engines figure out the meanings on

[1] https://www.w3.org/TR/rdf11-concepts/

[2] https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/

web pages more effectively and serve relevant results based on search queries of the internet users. The Semantic Web is growing steadily and producing a large amount of RDF data. The number of Web pages that have markup conforming to the *schema.org* format is more than 2.5 billion and is still increasing. To improve the accuracy of recommendations, recommender companies are increasingly using semantics and semantic tagging. DBpedia (Auer et al., 2007), YAGO (Hoffart et al., 2013), Bio2RDF (Callahan et al., 2013), Google's Knowledge Vault (Dong et al., 2014) , Probase (Wu et al., 2012), PubChemRDF (Fu et al., 2015), and Universal Protein Resource (UniProtKB) (Apweiler et al., 2014) consist of billions of facts that are represented as RDF data contained in the Linked Open Data (LOD) (Bizer et al., 2009) cloud and are queried through a declarative query language SPARQL (Hartig et al., 2009) recommended by W3C.

The RDF data model is a directed, labeled, and interlinked graph consisting of a set of triples (subject, predicate, object) that can be interpreted as a directed edge from subject (s) to object (o) labeled by predicate (p). Thus, a predicate represents the relationship between a subject and an object where subjects and objects are the arbitrary resources.

For example, the statement:

Spielberg was born in 1946.

can be represented as a triple of RDF data:

(s, p, o) = (Spielberg, wasBorn, 1946)

And, also can be represented as the labeled directed graph (DG):

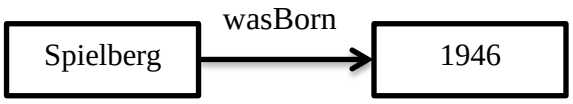

Figure 1.1: Labeled Directed Graph Representation of a Triple

A finite set of RDF triples constitutes an RDF graph. On the other hand, SPARQL queries have a set of triple patterns similar to RDF triples except that each of the subject, predicate, and object can be a variable (Pérez et al., 2009). The most basic SPARQL query contains a group of Basic Graph Pattern (BGP) queries having a SELECT clause that identifies *variables* that appear in the result set, and a WHERE clause of a graph pattern to match against the RDF graph. RDF and SPARQL constitute core layers of Semantic Web stack, in the next chapter a detailed description about RDF and SPARQL is presented.

A SPARQL query that returns the year Spielberg was born can be written as:

```
SELECT  ?year  WHERE {
            "Spielberg"  wasBorn  ?year .
}
```

Q1: An Example SPARQL Query

The SPARQL query graph of Q1 is shown in Figure 1.2.

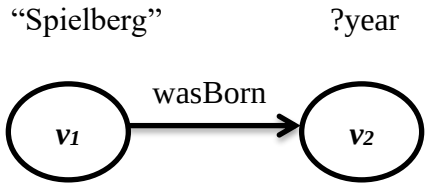

Figure 1.2: A SPARQL Query Graph of Q1

The Semantic Web keeps growing rapidly and continues to generate a high volume of RDF data. Therefore, the sizes of RDF datasets will continue to increase from hundreds of millions to several billions of triples, which represent large-scale graphs with

millions to billions of edges, making RDF data management very difficult. Providing scalable, highly available and fault-tolerant RDF store with efficient SPARQL query processing has become the major challenge in RDF data management systems.

Researchers proposed many centralized RDF data management systems during the past decades to store RDF data and execute SPARQL query. Such system includes: Jena (McBride, 2001), Sesame (Broekstra et al., 2002), RDF-3X (Neumann & Weikum, 2008), chameleon-db (Ozsu et al., 2013), HexaStore (Weiss et al., 2008), SW-Store (Abadi et al., 2009), TripleBit (Yuan et al., 2013), BitMat (Atre et al., 2009), and gStore (Zou et al., 2011). To handle the large-scale RDF data and answer complex queries efficiently these systems need to increase their resources like storage space, processing power, and memory, etc. The weaknesses of this approach are the lake of scalability and vulnerable to hardware failure. To overcome the limitations of centralized single machine systems researchers have moved towards distributed RDF data management systems, like 4store (Harris et al., 2009), YARS2 (Harth et al., 2007), Virtuoso Cluster (Boncz et al., 2014). These systems are extended from the centralized system to the nodes of the cluster for distributed RDF processing. Other distributed systems such as HadoopRDF (Du et al., 2012) use the HDFS[3] (Hadoop Distributed File System) and MapReduce paradigm (Dean et al., 2008) for storing and processing RDF data. H_2RDF+ (Papailiou et al., 2013) approach is based on HBase[4], column-oriented NoSQL key-value store on top of HDFS. With this approach, queries can be executed in a single machine as well as on a computing cluster via MapReduce. Apart from that, a number of NoSQL triple stores

[3] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[4] https://hbase.apache.org/

based on the Hadoop [5] ecosystem have been proposed. For instance: Jena-HBase (Khadilkar et al., 2012), Hive+HBase (Cudr´e-Mauroux1 et al., 2013), CumulusRDF (Ladwig & Harth, 2011), Couchbase(Cudr´e-Mauroux1 et al., 2013), and so on. Jena-HBase uses HBase as an RDF triple storage and Jena framework for the SPARQL query engine. Hive+HBase is also an HBase backed triple store and Hive[6] has SQL like query language (HiveQL) that allows querying using MapReduce. CumulusRDF has Apache Cassandra[7] as storage back-end and Sesame query processor for executing SPARQL query. Couchbase stores RDF triple as JSON document and it provides API for SPARQL query. The systems that rely on MapReduce framework suffer from the high latency of startup and I/O costs for its underlying batch-oriented nature and cannot provide interactive query runtimes, therefore, researchers are now moving towards in-memory frameworks like Impala (Kornacker et al., 2015), Spark (Zaharia et al., 2012) for distributed RDF management system. Distributed in-memory RDF management systems include: Sempala (Schätzle et al., 2014) a SPARQL-over-SQL approach based on Hadoop, uses a single Unified Property Table using Parquet columnar storage format that resides on HDFS, and a Massive Parallel Processing (MPP) SQL query engine Impala. The advantage of using a Unified Property Table compared to a Triples Table or VP approach is that it can reduce the number of subject-subject *self-joins* for star-shaped query pattern but there is no real benefit for linear-shaped patterns. S2RDF (Schätzle et al., 2016) is another in-memory SPARQL query processing system that uses the ExtVP (Extended Vertical Partitioning) scheme for data storage layout and relies on a translation

[5] https://hadoop.apache.org/
[6] https://hive.apache.org/
[7] https://cassandra.apache.org/

of SPARQL queries to SQL for being executed using an in-memory SQL framework called Spark SQL. ExtVP uses a *semi-join* based preprocessing approach to compute the possible *join* relations between partitions of VP tables in order to minimize the input size for the query. This approach suffers from high preprocessing cost.

1.2 Research Objectives

In distributed RDF systems, both the data and the query processing are highly distributed. On the other hand, SPARQL workloads are dynamic and structurally diverse (Ozsu & Daudjee, 2014). Each workload can have queries with different structures with different degrees of complexity (Ozsu & Daudjee, 2014). A complex SPARQL query over a large RDF graph in distributed systems requires combining a lot of distributed pieces of data through *join* operations. In a distributed system, the query engine decomposes a SPARQL query into multiple sub-queries, each of which are evaluated by individual computational nodes independently, and all participating nodes may need to exchange results during query evaluation. Consequently, a query having large intermediate results incurs high communication overhead as well as requires expensive *join* operations for intermediate results (Huang et al., 2011), which leads to poor performance of the query. Therefore, designing an efficient data-partitioning scheme and *join* strategy to minimize data transfer is the fundamental challenge in distributed RDF data management systems. The goal of this thesis is to provide a distributed RDF management system for SPARQL query based on in-memory cluster computing framework that overcomes the existing limitations.

CHAPTER 2

LITERATURE REVIEW

The greatest repository of information, the World Wide Web (aka the Web), is based mainly on documents that can be seen as a “Web of Documents”. These “Web of Documents” are primarily designed for human consumption and the structure of the data in the Web is not in machine-readable format, making the software agents almost impossible to access and process without human intervention. To overcome this limitation Tim Berners-Lee et al. proposed their vision of Semantic Web (Berners-Lee et al., 2001) to turn the Web into a global repository of information from “Web of Documents” to “Web of Data” by giving the data precise semantics, or well-defined meaning, in machine-processable format in-order to automate the processing of information for software agents in the web. Therefore, Semantic Web is viewed as an evolving extension of the current World Wide Web. The infrastructure of Semantic Web is a layered architecture consisting of a set of standards and technologies led by the World Wide Web Consortium (W3C).

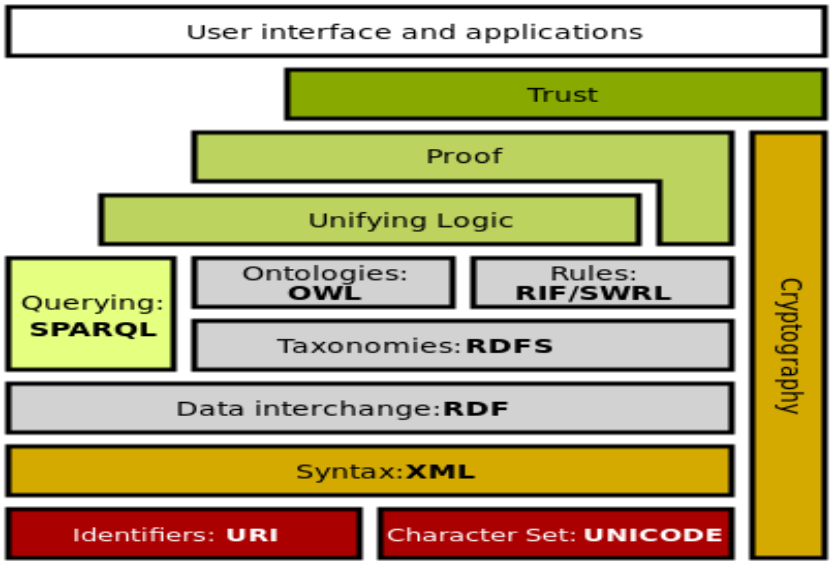

Figure 2.1: Semantic Web Stack (Gezer & Bergweiler, 2016)

Table 2.1: Sample RDF Data

1.	<http://localhost/publications/articles/Article_1>	<http://www.w3.org/1999/02/22−rdf−syntax−ns#type>	<http://localhost/vocabulary/bench/Article>
2.	<http://localhost/publications/articles/Article_1>	<http://example.org/property/title>	“Title One”
3.	<http://localhost/publications/articles/Article_1>	<http://example.org/property/author>	<http://localhost/persons/David_Gary>
4.	<http://example.com/person/David_Gary>	<http://www.w3.org/1999/02/22−rdf−syntax−ns#type>	<http://xmlns.com/foaf/0.1/Person>
5.	<http://localhost/persons/David_Gary>	<http://example.org/property/name>	“David Gary”
6.	<www.aaa.com/d_g>	<http://example.org/property/website_of>	<http://localhost/persons/David_Gary>
7.	<http://localhost/publications/articles/Article_2>	<http://www.w3.org/1999/02/22−rdf−syntax−ns#type>	<http://localhost/vocabulary/bench/Article>
8.	<http://localhost/publications/articles/Article_2>	<http://example.org/property/title>	“Title Two”
9.	<http://localhost/publications/articles/Article_2>	<http://example.org/property/author>	<http://localhost/persons/David_Gary>
10.	<http://localhost/publications/articles/Article_2>	<http://example.org/property/author>	<http://localhost/persons/John_Wayne>
11.	<http://localhost/persons/John_Wayne>	<http://www.w3.org/1999/02/22−rdf−syntax−ns#type>	<http://xmlns.com/foaf/0.1/Person>
12.	<http://localhost/persons/John_Wayne>	<http://example.org/property/name>	“John Wayne”
13.	<www.bbb.com/j_w>	<http://example.org/property/website_of>	<http://localhost/persons/John_Wayne>
14.	<http://localhost/publications/articles/Article_3>	<http://www.w3.org/1999/02/22−rdf−syntax−ns#type>	<http://localhost/vocabulary/bench/Article>
15.	<http://localhost/publications/articles/Article_3>	<http://example.org/property/title>	“Title Three”
16.	<http://localhost/publications/articles/Article_3>	<http://example.org/property/author>	<http://localhost/persons/John_Wayne>

Figure 2.1 illustrates the different layers of Semantic Web technology stack according to W3C to create a Web of linked

\ data. The fundamental technologies of Semantic Web are RDF, a data model to encode machine-readable data and SPARQL, a W3C recommended language to query RDF data. In the following sections, we describe in more detail about RDF and SPARQL.

2.1 RDF – The Resource Description Framework

RDF is a schema-free data model recommended by W3C to describe the information about arbitrary resources on the Web. An RDF dataset consists of a collection of triples (subject, predicate, object), abbreviated as (s, p, o). In an RDF triple (aka RDF statement) a subject denotes the entity or a class of resources, a predicate denotes an attribute or aspect and relationship (aka property) between entities or classes, and an object denotes an entity, a class, or a literal value. The RDF dataset represents triples as a directed graph with annotations called RDF graph. Nodes of an RDF graph represent either subject or object and edges represent the predicate. Each node can be an Internationalized Resource Identifier (IRI), literal or blank node.

An example of an RDF graph is given in Figure 2.2 of a simple RDF dataset (in N–triples[8] format) of Table 2.1. Table 2.1 is a collection of triples of the form (subject, predicate, object), where triple components: subject, predicate, and objects are separated by whitespace and each triple shown in each line of the table is terminated by a dot (.). Figure 2.2 simulates an RDF graph with 16 edges of a simple publication network of RDF dataset consists of 16 triples listed in Table 2.1, where eclipse nodes represent

[8] https://www.w3.org/TR/n-triples/

resources, directed edges represent properties, and rectangular nodes represent literal values.

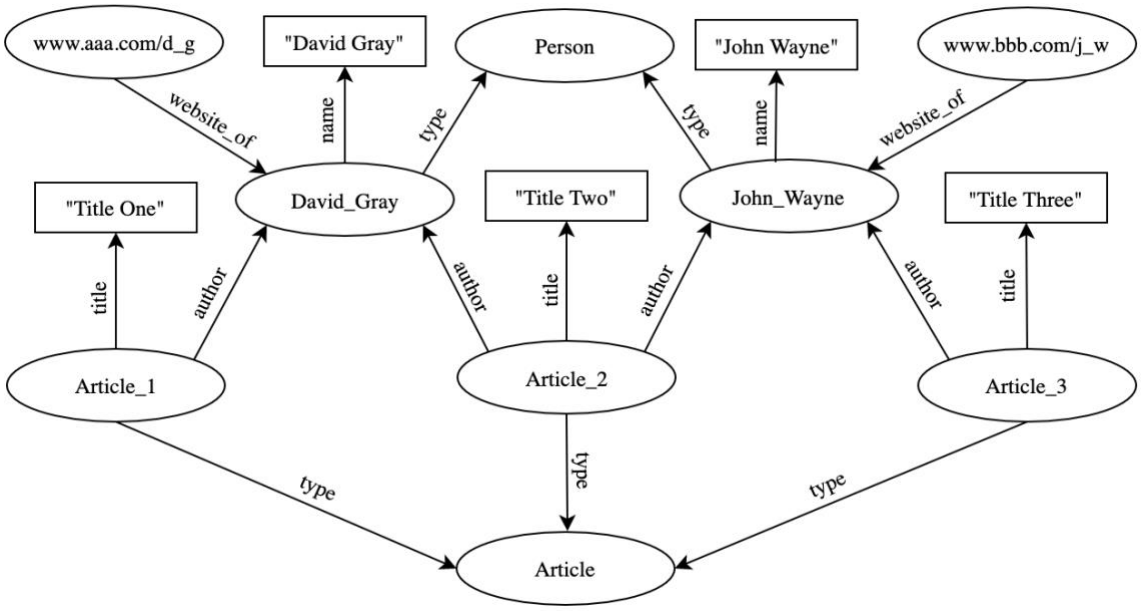

Figure 2.2: An Example RDF Graph

Let *I*, *B*, and *L* be infinite sets of *IRIs*, *blank nodes*, and *literal* respectively which are pairwise disjoint. And suppose V be an infinite set of variables disjoint with *I*, *B*, and *L*. Assuming furthermore all RDF valid terms are the union of $(I \cup B \cup L)$ and denoted by *T*.

Definition 2.1 (RDF Triple). A ternary tuple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an RDF triple where s, p, and o denote subject, predicate, and object respectively.

Definition 2.2 (RDF Graph). An RDF graph $G = \{t_1, \ldots. t_n\}$ is a finite set of RDF triples t_i where $1 \leq i \leq n$.

Definition 2.3 (RDF Dataset). An RDF dataset is a collection of RDF graphs $D = \{G_0, (i_1, G_1), \ldots. (i_n, G_n)\}$ with $i_1, \ldots, i_n \in I$ where $1 \leq i \leq n$. (i_i, G_i) are named graphs

identified by an IRI and default graph G_0 which does not have a name. D(i) denote the RDF graph in D identified by i.

2.2 SPARQL- SPARQL Protocol and RDF Query Language

SPARQL is the standard query language recommended by W3C for RDF data. A basic SPARQL query consists of a SELECT clause followed by query variables represented by the bound variables (variable with specified value) that appear in the result set and a WHERE clause followed by graph patterns that match against the RDF graph the query is being run on. A SPARQL query can be one of the four types including SELECT, ASK, DESCRIBE, and CONSTRUCT. On the other hand, a graph pattern that defines the query semantics can be one of the following types: Basic Graph Pattern (BGP), Basic Graph Pattern with Filter Constraint (FGP), Optional Graph Pattern (OGP), Union Graph Pattern (UGP) or Alternative Graph Pattern (AGP), and Group Graph Pattern (GGP). Anyone of BGP, FGP, and OGP consists of one or multiple triple patterns, while a GGP or UGP (aka AGP) consists of one or multiple BGPs, FGPs or OGPs. Each part of a triple pattern: subject, predicate, and object can be either a bound or unbound variable. Basically, the result of a SPARQL query is obtained by replacing the variables of the query graph patterns with the elements of an RDF graph. SPARQL query has solution modifiers: ORDER BY (sort by defined order), DISTINCT (remove all duplicates), REDUCED (remove some duplicates), OFFSET (skip the first specified number of solutions), and LIMIT (upper bound on the number of solutions).

A SPARQL query that returns the title of articles John Wayne wrote can be written as:

```
SELECT ?title WHERE {
?article <http://example.org/property/author> <http://localhost/persons/John_Wayne> .
?article <http://example.org/property/title> ?title .
<http://localhost/persons/John_Wayne> <http://example.org/property/name> "John Wayne" .
}
```

Figure 2.3: An Example SPARQL Query for RDF Dataset of Table 2.1

This query returns the results:

"Title Two"

"Title Three"

The above query can also be expressed using prefixes (Figure 2.4):

```
PREFIX  :   <http://example.org/property/>
PREFIX  pn:   <http://localhost/persons/>

SELECT ?title WHERE {
        ?article        :author     pn:John_Wayne .
        ?article        :title      ?title .
        pn:John_Wayne   :name       "John Wayne".
}
```

Figure 2.4: SPARQL Query with Prefixes

In practice, URIs (Uniform Resource Identifier) are used to uniquely identify resources, where each URI consists of a preceding *namespace* and a trailing *identifier*. Consider a URI for John Wayne in Table 2.1 (**http://localhost/persons/John_Wayne**) that consists of the namespace **http://localhost/persons/** and the identifier **John_Wayne** with angle brackets. In the above SPARQL query, prefix declarations are used for abbreviating URIs. Here, we have defined the prefix of resources of Table 2.1 as **PREFIX pn: <http://localhost/persons/>**, therefore, we can write the same URI as **pn: John_Wayne**.

2.3 Query Shapes

A BGP represents the core of the SPARQL query. SPARQL BGPs can have one of the four following shapes depending on the position of variables in the triple patterns which can have severe impacts on query performance (Aluç et al., 2014).

1. **Chain Shaped Pattern** consists of a set of triple patterns that are linked together as subject-object *joins* via different unique *join* variables at the subject or object positions.

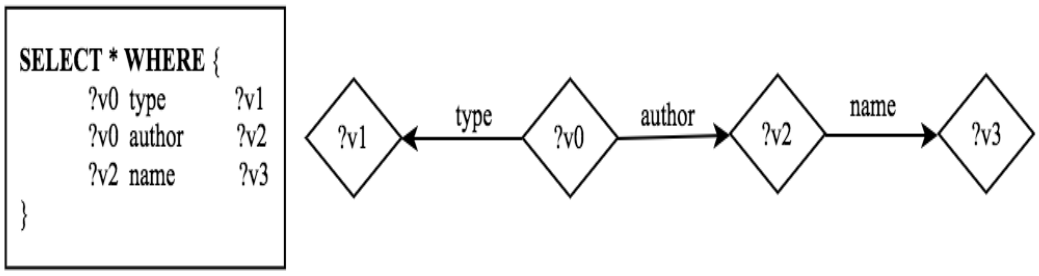

Figure 2.5: An Example Chain Shaped Query

2. **Star Shaped Pattern** consists of a set of triple patterns that are linked together via a single *join* variable at the subject or the object position.

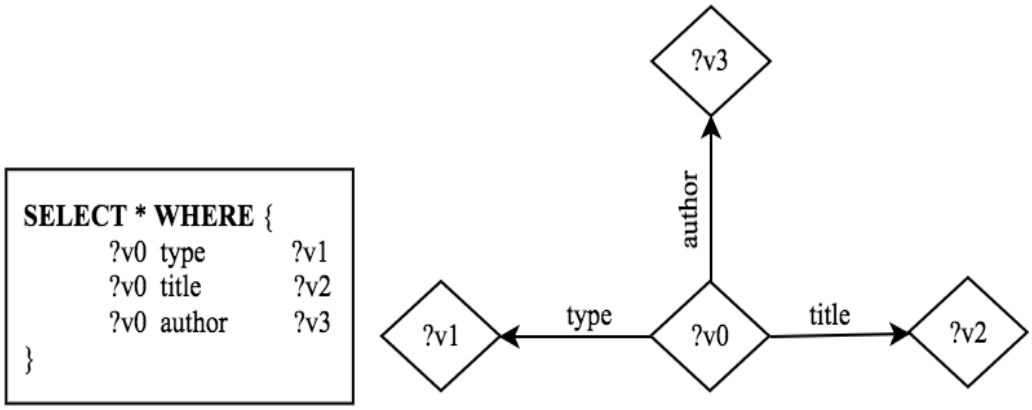

Figure 2.6: An Example Star Shaped Query

3. **Snowflake Shaped Pattern** consists of several star shapes via different *join* variables at the subject or the object positions in the triple pattern.

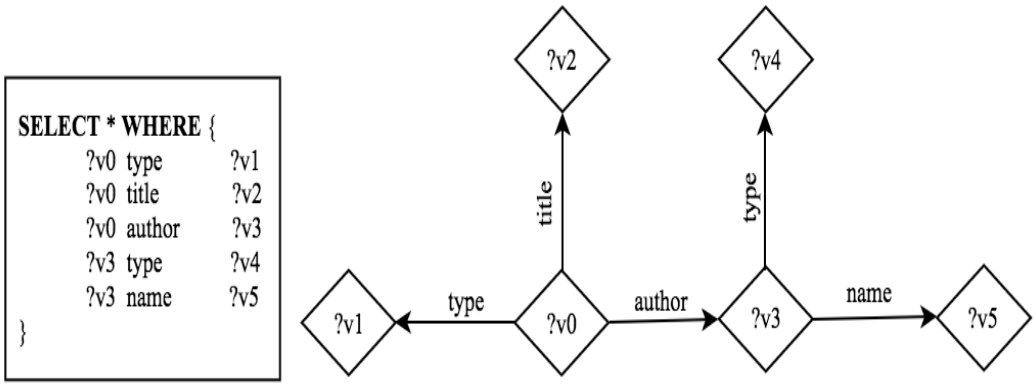

Figure 2.7: An Example Snowflake Shaped Query

4. **Complex query** structures are the compositions of the above-mentioned fundamental query patterns.

Definition 2.4 (Query Variables). A query variable is a member of an infinite set V where V is disjoint from RDF term T ($I \cup L \cup B$) where I, B, and L are *IRIs*, *blank nodes*, and *literal* respectively.

Definition 2.5 (Triple Pattern). A ternary tuple $tp \in (I \cup V) \times (I \cup V) \times (I \cup V \cup L)$ is called a SPARQL triple pattern.

Definition 2.6 (Basic Graph Pattern). A finite set of triple patterns is called a basic graph pattern (BGP). BGPs are the building block in SPARQL. The empty graph pattern is a basic graph pattern, which is an empty set.

Definition 2.7 (SPARQL Filter Condition). Let o denote one of the following comparison operators ($<, \leq, !=, =, \geq, >$) and *func* be a SPARQL built-in boolean function. Let ?x, ?y $\in V$ be variables and c $\in I \cup L$. A SPARQL filter condition can be recursively defined as follows:

- The expressions (?x o c) and (?x o ?y) are filter conditions.
- The expression *func*(?x) is a filter condition.

- If *F1* and *F2* are filter conditions, then $\neg F1$, $(F1 \wedge F2)$ and $(F1 \vee F2)$ are filter conditions.

Definition 2.8 (SPARQL Graph Pattern). A SPARQL graph pattern can be recursively defined as follows:

- A basic graph pattern is a graph pattern.
- If P_1 and P_2 are graph patterns, then expressions $(P_1 \text{ . } P_2)$, $(P_1$ **UNION** $P_2)$, $(P_1$ **OPTIONAL** $P_2)$ are graph patterns (conjunction graph pattern, optional graph pattern, and union graph pattern, respectively).
- If *P* is a graph pattern and $x \in I \cup V$, then (**GRAPH** *x P*) is a graph pattern.
- If *P* is a graph pattern and *F* is a filter condition to restrict the solutions of a triple pattern, then (*P* **FILTER** *F*) is a graph pattern.

Definition 2.9 (Solution Mapping). A solution mapping μ is a partial function that maps from a set of variables to a set of RDF terms represented by $\mu : V \rightarrow T$. The domain of μ, dom(μ), is the subset of V where μ is defined.

2.4 Distributed RDF Processing

Over the past decade, many RDF data management systems have been built based on distributed storage systems to provide efficient, scalable, highly available and fault tolerance services. Based on indexing and partitioning schemes on recent distributed RDF systems relevant to this research work distinguishing into two categories: RDF Storage with Indexing, and RDF Storage with Partitioning.

2.4.1 RDF Storage with Indexing

Existing works use various indexing strategies on RDF elements to develop RDF storage layouts.

HadoopRDF (Du et al., 2012) uses HDFS files to partition the input RDF data into multiple smaller files based on two steps. First, the input RDF file is partitioned and named after predicate values (creating pos index) into multiple smaller files similar to Vertical Partitioning presented in SWStore (Abadi et al., 2009). The second step works on the explicit type information of object in the *rdf:type* file. It splits the *rdf:type* file into multiple smaller files based on the number of distinct objects present in the *rdf:type* predicate. The pos index created in Hadoop RDF retrieves subject-object combinations for a given predicate. HadoopRDF uses MapReduce (Dean et al., 2008) jobs to perform SPARQL query.

H2RDF (Papailiou et al., 2012) is built on top of HBase uses a three-index scheme (spo, pso, and osp) for input RDF data over the HBase store. H2RDF collects statistics during data loading and uses Partial Input Join algorithm, which utilizes the statistics and HBase indexing to find the *join* containing small input patterns that need to be executed.

H2RDF+ (Papailiou et al., 2013) is another distributed RDF engine based on MapReduce (Dean et al., 2008) framework and HBase, which is an extension of H2RDF maintains all permutations of RDF indexing (SPO, PSO, POS, OPS, OSP, SOP). Additionally, it also maintains aggregated index statistics to estimate triple pattern selectivity, *join* output size and *join* cost. The indexing scheme is used to answer efficiently all SPARQL triple patterns using a single index scan on the corresponding index and every *join* between triple patterns are done using merge *joins* that can effectively exploit the pre-computed orderings in these indices. H2RDF+ can adaptively decide whether to execute the query in a centralized system over a single machine or a

distributed mode using MapReduce Jobs based on the query complexity determined from the above-mentioned estimations.

CumulusRDF proposed in (Ladwig & Harth, 2011) has been built on top of a nested key-value store Cassandra (Laksham Avinash & Prashant Malik, 2010) that uses two different indexing strategies, hierarchical layout and flat layout for storing RDF triples. The indexing scheme of the hierarchical layout uses super columns to build indices (SPO, POS, OSP) in-order to answer all eight possible RDF triple patterns. The second indexing strategy called flat layout uses the standard key-value model of Cassandra to store the SPO, POS and OSP indices, where keys are sorted according to their natural order. This secondary index refers to CSPO in CumulusRDF. To evaluate the SPARQL query, the indices to use are identified according to the fixed parts of the triple pattern.

Rya (Punnoose et al., 2012) has been implemented on top of a key-value store Accumulo[9] stores RDF triple in the Row ID part of the Accumulo tables and indexes the triples across three separate tables (spo, pos, and osp) by maintaining the different ordering of the subject, predicate, object for each table. These three permutations (spo, pos, and osp) of triple components are sufficient to answer all possible triple patterns by using range scan on the appropriate index.

AMADA (Aranda-Andújar et al., 2012) uses a distributed file system called Amazon's Simple Storage Service (S3) to store RDF datasets and builds its own data indexes by mapping RDF elements to the RDF datasets containing them using

[9] https://accumulo.apache.org/

SimpleDB[10]. In order to answer a query, the query processor module of AMADA first parse the query and then perform a look up to the indexes in SimpleDB to find out the relevant indexes for the query.

2.4.2 RDF Storage with Partitioning

This section reviews distributed RDF systems that utilize hashing or graph partitioning strategies.

CliqueSquare (Kaoudi et al., 2015) uses built-in data replication mechanism of HDFS to partition the RDF dataset by hashing on all three columns of triples based on their subject, predicate and object values and creates three replicas by default. The first replica holds the partitions of triples based on their subject, predicate, and object values. Second replica stores all subject, predicate, and object partitions of the same value within the same node. For the third replica, CliqueSquare groups all the subject partitions within a node by the value of the predicate in their triples. It also groups all object partitions based on their predicate values. CliqueSquare uses a clique-based algorithm to select the partitions in such a way that can reduce as much as possible data exchange in the shuffle phases and minimize the number of MapReduce stages.

PigSPARQL (Schätzle et al., 2011, 2013) stores RDF data using a Vertical Partitioning schema proposed in (Abadi et al., 2007) and uses an intermediate layer called Pig[11] that translates each SPARQL query into a PigLatin (Schätzle et al., 2011) program that is executed using MapReduce.

S2X (Schätzle et al., 2016) is a SPARQL query engine that exploits the graph

[10] https://aws.amazon.com/simpledb/
[11] https://pig.apache.org/

parallel abstraction, GraphX[12], along with the data-parallel computation of Spark to evaluate SPARQL queries over RDF data on Hadoop. S2X devises a property graph model for RDF data and uses GraphX 2D hashing to partition the input graph. It applies a parallel vertex-centric model for basic graph pattern matching of SPARQL.

S2RDF (Schätzle et al., 2016) has been built on top of Spark[13] that uses a relational partitioning technique called Extended Vertical Partitioning (ExtVP) which is an extension of Vertical Partitioning (VP) approach used by HadoopRDF (Du et al., 2012) to store RDF data on the HDFS using Parquet[14] columnar storage format. The goal of ExtVP approach is to minimize the input size for the query by using a *semi-join* based preprocessing approach to compute the possible *join* relations between partitions of VP tables. S2RDF executes SPARQL queries by translating them into SQL queries, which are then evaluated using Spark SQL[15].

SPARQLGX (Graux et al., 2016) also built on top of Spark uses Vertically Partitioned approach proposed in (Abadi et al., 2009) to store the RDF dataset into HDFS and compiles the SPARQL queries into Scala code in order to execute directly into Spark operations. The system uses its own statistics to optimize the computation with less intermediate results.

PRoST (Cossu et al., 2018) is a Spark based distributed system for RDF storage and SPARQL querying that stores data twice using Vertical Partitioning and Property Table. PRoST translates SPARQL queries into the *Join Tree* format where every node represents either the Vertical Partitioning table or Property Table. The triple patterns with

[12] https://spark.apache.org/graphx/
[13] https://spark.apache.org/
[14] https://parquet.apache.org/
[15] https://spark.apache.org/docs/latest/sql-programming-guide.html

the same subject in a unique basic graph pattern are grouped to form a single node where the Property Table is used. All the other groups with a single triple pattern are translated to nodes that use the Vertical Partitioning tables.

Table 2.2: Summary of Distributed RDF Systems

System	Storage Strategy	Storage Backend	Execution Framework
HadoopRDF (Du et al., 2012)	Vertical Partitioning and Property-based Files	Distributed File System	MapReduce
H2RDF (Papailiou et al., 2012)	3 Indices (SPO, POS, OSP)	Key-Value Store	MapReduce
H2RDF+ (Papailiou et al., 2013)	6 Indices (SPO, PSO, POS, OPS, OSP, SOP)	Key-Value Store	MapReduce
CumulusRDF (Ladwig & Harth, 2011)	3 Indices (SPO, POS, OSP)	Key-Value Store	Sesame Query Processor
Rya (Punnoose et al., 2012)	3 Indices (SPO, POS, OSP)	Key-Value Store	OpenRDF Sesame Framework
AMADA (Aranda-Andújar et al., 2012)	3 Indices (SPO, POS, OSP)	Key-Value Store	SimpleDB Query Processor
CliqueSquare (Kaoudi et al., 2015)	Hash and Vertical Partitioning	Distributed File System	MapReduce
PigSPARQL (Schätzle et al., 2011, 2013)	Vertical Partitioning	Distributed File System	SPARQL to PigLatin
S2X (Schätzle et al., 2016)	Graph-based Partitioning	Distributed File System	Vertex-Centric BGP Matching
S2RDF (Schätzle et al., 2016)	Vertical Partitioning and Extended Vertical Partitioning	Distributed File System	SPARQL to SQL
SPARQLGX (Graux et al., 2016)	Vertical Partitioning	Distributed File System	SPARQL to Scala Code
PRoST (Cossu et al., 2018)	Vertical Partitioning	Distributed File System	SPARQL to SQL

Distributed systems typically index and partition RDF data using various indexing and partitioning strategies. Table 2.2 summarizes the storage layout schemes used by the

existing distributed RDF systems.

2.5 RDF Benchmarks

RDF benchmarks are used to identify the strengths and weakness of SPARQL evaluators. These benchmarks are usually made of two parts: the first one is the dataset generator, and the second part is the set of standard queries that should be evaluated on those datasets to assess the performance of RDF systems. It is hard to choose a right benchmark that will cover all the use cases of RDF systems. Following are the most commonly used benchmarks to tune the performance of RDF systems.

- **Lehigh University Benchmark** (LUBM) (Guo et al., 2005) was proposed in 2005 that features an ontology for the university domain, synthetic OWL data scalable to an arbitrary size, fourteen standard queries representing a variety of properties, and several performance metrics. This benchmark was originally designed to test the inference capabilities of Semantic Web repositories.
- **SPARQL Performance Benchmark** (SP^{2}Bench) uses DBLP[16] as its domain and generates the synthetic dataset mimicking the original DBLP data in RDF format. SP^{2}Bench was presented in (Schmidt et al., 2009) with a set of benchmark queries to tests the various SPARQL features including FILTER, OPTIONAL, UNION, solution modifiers and ASK queries.
- **Berlin SPARQL Benchmark** (BSBM) (Bizer & Schultz, 2009) was developed for comparing the performance between native RDF stores and systems featuring SPARQL-to-SQL rewriters. BSBM has adopted an e-commerce application as their case study and mainly addressed the dataset generation process. This

[16] https://dblp.org/

benchmark defines parametric query templates, which are used to create concrete, randomized benchmark queries by sampling template parameters over the corresponding input data. It was proposed with a "query mix" to test various SPARQL features similar to SP^2Bench excepted the ASK but with additional DESCRIBE and CONSTRUCT queries.

- **Waterloo SPARQL Diversity Test Suite** (WatDiv) (Aluç et al., 2014), introduced in 2014 by the University of Waterloo, is a synthetic dataset based on the e-commerce use case scenario. WatDiv has a data generator as well as query generator and was designed to cover both structural and data-driven features of four different shapes, namely, linear, star, snowflake, and complex SPARQL queries.
- **YAGO** (Yet Another Great Ontology), which is a semantic knowledge base developed at the Max Planck Institute for Computer Science, derived from Wikipedia, WordNet, and GeoNames. **YAGO2** (Hoffart et al., 2013) is an extension of the **YAGO** knowledge base in which entities, facts, and events are anchored in both time and space. This dataset represents a prime example of a large real-world dataset.
- **DBLP** Computer Science Bibliography provides bibliographic information on computer science journals and proceedings. It is a real-world dataset.

2.6 Distributed Data Management Tools

Hadoop is an open-source framework for distributed storage and processing of large datasets based on the HDFS and MapReduce paradigm (Dean et al., 2008). HDFS is a very popular distributed file system due to its replication capability to provide data

redundancy where MapReduce can be I/O intensive and not suitable for interactive queries. To overcome this issue, a number of distributed computation engines based on in-memory processing strategy have been introduced.

Spark is an in-memory cluster-computing framework like MapReduce, which utilizes in-memory caching and advanced directed acyclic graph (DAG) execution engine to create efficient query plans for data transformations. Spark runs programs up to 100 times faster in-memory processing mode and 10 times faster in disk processing mode than Hadoop MapReduce. Spark has a SQL like module called Spark SQL that is used for structured data processing and allows running SQL like queries on Spark data. Spark SQL includes a cost-based optimizer that enables control code generation to make queries faster.

Drill[17] is an MPP-based schema-free distributed SQL query engine that executes SQL queries on a number of NoSQL databases and different file formats. Drill can operate on more than one record at a time with vectorization and can also optimize queries and has ability to generate code on the fly for better performance.

2.7 Big Data File Formats

In this section, we discuss the state-of-the-art big data file formats called Parquet and ORC, which are relevant to this research work.

Parquet is a column-oriented data storage format of the Apache Hadoop ecosystem. It stores data in a column-oriented way, where the values of each column are organized consecutively on a disk that enables better compression. This data format supports additional optimizations include encodings (bit packing, run length, and

[17] https://drill.apache.org/

dictionary encoding) as well as compression algorithms like Snappy[18], GZip[19], LZO[20], and so on. Parquet supports both flat and nested data. Parquet has a filter pushdown option that prunes extraneous data to reduce the number of data scans and reads when a query contains a filter expression. Pruning data reduces the I/O, CPU, and network overhead to optimize query performance. Another advantage is that NULL values are not stored explicitly in Parquet, therefore, sparse columns cause little to no storage overhead.

ORC[21] (Optimized Row Columnar) is a columnar file format that provides a highly efficient way to store relational data. It stores collections of rows in one file, and within the collection, the row data is stored in a columnar format. This allows parallel processing of row collections across a cluster. Each file with the columnar layout is optimized for compression and skipping of data/columns reduces read and decompression load. Its file structure consists of three parts: Stripe, Footer, and Postscript. It breaks the source file into a set of rows called a Stripe. The default stripe size is 250 MB. This large stripe size enables an efficient read of columns from HDFS. The file footer contains a list of stripes in the file, the number of rows per stripe, and each column's data type. It also contains column-level aggregate count, min, max, and sum. Postscript contains compression parameter and size of the compressed footer. Each stripe in an ORC File has three parts: Index data, Row data, and Stripe footer. Index data include min and max values for each column and the row positions within each column. Row index entries provide offsets that enable seeking the right compression block and byte within a decompressed block. The Row data are composed of multiple streams per column, and

[18] http://google.github.io/snappy/
[19] https://www.gnu.org/software/gzip/
[20] http://www.oberhumer.com/opensource/lzo/
[21] https://orc.apache.org/

they are used in table scans. The stripe footer contains a directory of stream locations. Figure 2.8 illustrates the layout of the ORC File structure. The columns in an ORC File separate the stripes or sections of the file. An internal index is used to track a section of the data within each column. This organization allows readers to efficiently omit the columns that are not required. Only required column values on each query are scanned and transferred on query execution. The ORC File supports sparse indexes that are data statistics and position pointers. The data statistics are used in query optimization, and they are also used to answer simple aggregation queries. The ORC reader uses these statistics to avoid unnecessary data read from HDFS. The position pointers are used to locate the index groups and stripes.

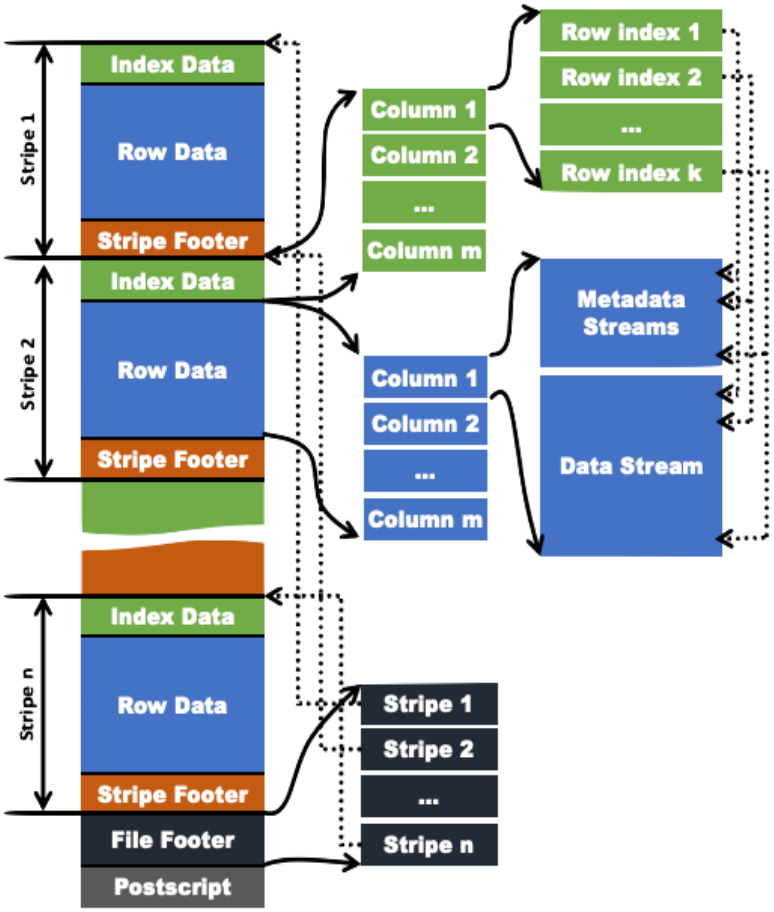

Figure 2.8: The ORC File Format (Huai et al., 2014)

The ORC File uses a two-level compression scheme. Each column can apply one of the four types of encoding schemes based on its data type: 1) a sequence of bytes, 2) a run-length encoded sequence of bytes, 3) a run-length and delta encoded sequence of integers, and 4) a bit vector. Users can further ask the writer of an ORC File to compress streams of data with a general-purpose codec among ZLIB[22], Snappy, and LZO. Metadata about the ORC data, such as the schema and compression format, are serialized into the file and are made available to the readers. The operator translates the ORC File schema into appropriate data flow types when possible.

[22] https://zlib.net/

CHAPTER 3

DISTRIBUTED IN-MEMORY RDF MANAGEMENT

During recent years, researchers have made significant efforts for efficient storage and querying of big RDF data in the distributed computing environments. Many RDF management systems are employing NoSQL databases on their storage layers. Concurrently, a number of distributed RDF management systems use in-memory cluster computing engines like Spark to minimize data preprocessing cost and improve query performance by exploiting data parallelization. Another key element of efficient query processing is the data partitioning scheme that has a huge impact in query answering. An efficient data partitioning scheme can handle all query types efficiently to improve the overall the efficiency of the system.

3.1 RDF Management with NoSQL Databases

In last few years, NoSQL databases have been used along with MapReduce computation model for RDF data storage and query processing. In contrast, in this chapter, we propose RDF data management systems based on NoSQL databases with in-memory processing engine (e.g., Spark) that seem promising for hosting massive RDF datasets in the distributed environment as well as faster query responses.

3.1.1 Data Modeling

In this section, we present our methodology for RDF data modeling using NoSQL databases (HBase[23] and Cassandra[24]). Before loading RDF data to HBase and Cassandra, we parse the input dataset and apply various transformations. The transformations include replacing all URIs with their corresponding namespace prefix and removing data type

[23] https://hbase.apache.org/
[24] https://cassandra.apache.org/

information from RDF object to convert it to primitive type because Spark SQL supports complex and primitive data types. Data type information of predicates is maintained in order to load RDF data into HBase and Cassandra.

RDF Data Layout for HBase. Each RDF subject is compressed with its properties and corresponding values and mapped to HBase row key. Each column name represents a property and the value corresponds to the subject, that is the row key. The column value represents the object from the RDF dataset. All columns of a row key belong to a single HBase column family. It is recommended in HBase Reference Guide[25] to keep the number of column families in schema low, that's why we choose a single column family for our HBase storage schema. In our approach, we created a single DataFrame for the HBase table and cached that table in memory.

Table 3.1: An Example of RDF Triples

<s_1>	<p_1>	<o_1>
<s_1>	<p_2>	"ABC"
<s_1>	<p_3>	<o_2>
<s_2>	<p_1>	<o_3>
<s_2>	<p_2>	"DEF"
<s_2>	<p_3>	<o_2>
<s_2>	<p_3>	<o_4>
<s_2>	<p_4>	"GHI"

Table 3.2: Sample Instance in HBase Storage Schema

rowkey	p:p_1	p:p_2	p:p_3	p:p_4
s_1	{o_1}	{"ABC"}	{o_2}	null
s_2	{o_3}	{"DEF"}	{o_2, o_4}	"GHI"

Table 3.2 represents our proposed HBase storage schema of sample RDF triples of Table 3.1. In Table 3.2, s_1 and s_2 denote row keys; p_1, p_2, p_3, and p_4 are column qualifiers, which belong to the same column family p; and {} denote set of cell values

[25] http://hbase.apache.org/book.html#_preface

with timestamps excluded. Here namespace IRIs of all of the subject, property, and object are replaced with their corresponding prefixes.

RDF Data Layout for Cassandra. The data layout to map RDF data to Cassandra involves each RDF subject and object (s, o) combined to form a partition key and each property forms the clustering column in a Cassandra table. Initially, a single DataFrame with subject, property, and object has been created for the Cassandra table.

Table 3.3: Sample Instance in Cassandra Storage Schema

s	p	o
s_1	p_1	o_1
s_1	p_2	"ABC"
s_1	p_3	o_2
s_2	p_1	o_3
s_2	p_2	"DEF"
s_2	p_3	o_2
s_2	p_3	o_4
s_2	p_4	"GHI"

Table 3.3 represents the Cassandra RDF storage schema of sample RDF triples (Table. 3.1) where s, p, and o denote subject, property, and object columns and their corresponding values are presented in the table cells. All namespace IRIs are replaced by their corresponding prefixes. Next, we use vertical partitioning approach that involves grouping triples based on predicates. A DataFrame is created for each predicate and cached in memory.

Table 3.4 shows the temporary views named p_1, p_2, p_3, and p_4 that are created from the initial Cassandra table (Table 3.3) for properties p_1, p_2, p_3, and p_4.

Table 3.4: Temporary Views of Predicates p_1, p_2, p_3, and p_4

p_1	
s	o
s_1	o_1
s_2	o_3

p_2	
s	o
s_1	"ABC"
s_2	"DEF"

p_3	
s	o
s_1	o_2
s_2	o_2
s_2	o_4

p_4	
s	o
s_2	"GHI"

3.1.2 SPARQL Query Translation

In this section, we present SPARQL query translation to SPARK SQL for both HBase and Cassandra storage schemas. In order to rewrite SPARQL query to SPARK SQL, we developed a query compiler, that is implemented in Flex – a lexical analyzer creator, Bison – a parser generator and C++11.

Algorithm 1: Block Query Generation and Triple Rewriting

Input: *sparqlQuery*

Output: *blocks: vector<block>, op: vector<Operator>, proj: vector<Projection>*

1: *{selectClause, whereClause}* ← *sparqlQuery*
2: Add each variable of *selectClause* to *proj* list
3: *{blocks, op}* ← *whereClause*
4: *blocks* → *{block₁, block₂,……., blockₙ}*
5: *op* → *{op₁, op₂,……,opₙ₋₁}*
6: *blocks* ← *tripleRewriting(blocks)*
7: return *blocks, op, proj*

SPARQL Parsing. Initially, the compiler validates the input SPARQL query and builds a query parse tree from SPARQL grammar rules. Then, Algorithm 1 transforms the input query into two clauses, SELECT and WHERE. The projected variables are listed in projection list *proj* from *selectClause*. The *whereClause*, basically a group graph

pattern, consists of n number of blocks of BGP where n ≥ 1 and are connected through n – 1 operators. The block connecting operator, *op*, is OPTIONAL or UNION. Each block, B_k contains a single or multiple sets of triple pattern *tp*, filter pattern *f* and block ID *k* as follows: $\{(tp_1, f_1, k), (tp_2, f_2, k) \ldots, (tp_i, f_i, k)\}$, followed by next block B_{k+1} and so on.

The list of *op* is based on the operator present in WHERE clause. If the operator is OPTIONAL then *op* will be LEFT OUTER JOIN, and if the operator is UNION then *op* will be UNION ALL. If *op* does not present in the SPARQL query, then all triple patterns will be treated as a single block. SPARQL queries can have the property part of the triple as a variable. The *tripleRewriting* method replaces property variable with a bound value that is found in a filter pattern for that variable and discards the FILTER from that block.

```
SELECT ?article WHERE {
                    ?article    rdf:type    bench:Article .
                    ?article    ?property   ?value .
                    FILTER (?property = swrc:pages)
}
```

Figure 3.1: SP^{2}Bench Query 3a

```
tp1: <?article, rdf:type, bench:Article>
tp2: <?article, swrc:pages, ?value>
```

Figure 3.2: Rewritten SP^{2}Bench Query Triples 3a

Figure 3.2 shows the triple patterns in tuple format for the SPARQL query of Figure 3.1. Finally, the algorithm produces blocks, op, and proj list.

SPARQL Translation for HBase Data Layout. The translation of SPARQL query to Spark SQL statement consists of the following steps: Initially, the compiler identifies unique predicates from query's BGP to create a view over HBase table where each row key represents unique subject and columns represent predicates of the SPARQL query. Each row in HBase table contains object value of corresponding predicate column.

If a subject does not have a value of a particular predicate, then we put object value as null for that predicate column. The compiler also lists predicates for each subject present in query's BGP. From the HBase table, we create DataFrame for each subject by adding NOT NULL to the subject's properties in the WHERE clause and then create temporary views from that DataFrame.

Table 3.5: Temporary Views of Subject s_1 and s_2

S1

p:p_1	p:p_2	p:p_3
{o_1}	{"ABC"}	{o_2}

S2

p:p_1	p:p_2	p:p_3	p:p_4
{o_3}	{"DEF"}	{o_2, o_4}	{o_3}

Table 3.5 represents the temporary views s_1, and s_2 of the corresponding subjects s_1 *and* s_2. These temporary views are NOT materialized to disk or even to the memory and lifetime of these views are dependent on the SparkSession. The *join* conditions are identified from any two triple patterns in the WHERE clause having the form (S_1 P_1 O_1) and (S_2 P_2 O_2), where $O_1 = S_2$ or $S_1 = O_2$ and $S_1 \neq S_2$ and requires two temporary views of subject to be joined. Finally, the compiler translates the SPARQL query to Spark SQL statement based on the subject-predicate mapping that is identified from query's BGP.

Algorithm 2: Generate Temporary Views

Input: *blocks, op*

Output: *predicates: vector<p>, tempViews: vector<string>*

1: *{subjects, predicates} ← getSubjectPredicate(blocks)*

2: *predicates → {p_1, p_2, p_3, ……..}*

subjects → {s_1, s_2, s_3, ……..}

// list of unique predicates and subjects used to create DataFrame for HBase table

3: *subToPred ← getSubPredMapping(blocks, op, subjects)*

```
4:    subToPred → vector<subject, vector<predicates>>
      // return subject with their corresponding predicates
5: tempViews ← getTempViews(blocks, op, subToPred)
   // generate temp views for each subject with NOT NULL condition.
6: return predicates, tempViews
```

Algorithm 2 takes *blocks* and *op* as input and produces a list of *predicates* and *tempViews*. The *getSubjectPredicate* method takes *blocks* as input and, *triples* in blocks are processed in an iterative manner to produce two lists of unique *predicates* and *subjects*. The *getSubPredMapping* method takes *blocks*, *op,* and *subjects* as input and produces *subToPred*, a list of subjects with their corresponding predicates list. The final method in algorithm 2, *getTempViews*, generates temporary views of each subject with the NOT NULL condition for their corresponding predicates. Finally, it produces a list of predicates and a list of the string representation of *tempViews* for each subject. Overall translation process from SPARQL to Spark SQL is shown in Figure 3.4 using BSBM query 10 (Figure 3.3). It is important to mention that we replaced colon (:) with an underscore (_) because Spark SQL statements do not support colon except in String values.

```
PREFIX  bsbm-inst:  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX  bsbm:       <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX  xsd:        <http://www.w3.org/2001/XMLSchema#>
PREFIX  dc:         <http://purl.org/dc/elements/1.1/>

SELECT DISTINCT ?offer ?price WHERE {
  ?offer    bsbm:product        bsbm-inst:dataFromProducer7/Product323.
  ?offer    bsbm:vendor         ?vendor .
  ?offer    dc:publisher        ?vendor .
  ?vendor   bsbm:country        <http://downlode.org/rdf/iso-3166/countries#US> .
  ?offer    bsbm:deliveryDays   ?deliveryDays .
  FILTER (?deliveryDays <= 3)
```

```
  ?offer     bsbm:price            ?price .
  ?offer     bsbm:validTo          ?date .
  FILTER (?date > "2008-07-08" ^^xsd:date)
}
ORDER BY xsd:double(str(?price))
LIMIT 10
```

Figure 3.3: BSBM Query 10

```
Temporary View offerTable:
SELECT rowKey AS offer, bsbm_product, bsbm_vendor, dc_publisher, bsbm_deliveryDays,
bsbm_price AS price, bsbm_validTo  FROM hbaseTable WHERE bsbm_product IS NOT NULL,
bsbm_vendor IS NOT NULL, dc_publisher IS NOT NULL, bsbm_deliveryDays IS NOT NULL,
bsbm_price IS NOT NULL, bsbm_validTo IS NOT NULL
Temporary View vendorTable:
SELECT rowKey AS vendor, bsbm_country FROM hbaseTable WHERE bsbm_country IS NOT NULL

Final Spark SQL statement:
SELECT DISTINCT offerTable.offer AS offer, offerTable.price AS price FROM offerTable JOIN
vendorTable ON (vendorTable.vendor = offerTable.vendor) WHERE offerTable.bsbm_product =
'bsbm-inst:dataFromProducer7/Product323'
AND vendorTable.bsbm_country = 'http://downlode.org/rdf/iso-3166/countries#US'
AND offerTable.bsbm_deliveryDays <= 3 AND offerTable.bsbm_validTo > 20080708
ORDER BY price
LIMIT 10
```

Figure 3.4: Compiler Translation of BSBM Query 10

This query translation process is slightly different for the query that has an OPTIONAL clause. If the SPARQL query contains OPTIONAL clauses, then the compiler first makes a list of all properties for each subject that will be used for projection along with subject (row key). If a subject or object is given, then it is added to the WHERE clause, however, only those predicates with the NOT NULL condition are added to the WHERE clause of Spark SQL statement. Predicates that appear before the OPTIONAL operator and predicates of the same subject that appear for the first time

after an OPTIONAL operator will be used for the NOT NULL condition in WHERE clause. Predicates are simply discarded from WHERE clause if they appear subsequent times after OPTIONAL operator for the same subject. If the same subject within a block exists after OPTIONAL operator in query's BGP, the compiler does not add that subject with LEFT OUTER JOIN in Spark SQL statement unless there is a join condition or multi-valued attribute with given object value in that block.

```
PREFIX  bsbm-inst:  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX  bsbm:       <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX  xsd:        <http://www.w3.org/2001/XMLSchema#>
PREFIX  dc:         <http://purl.org/dc/elements/1.1/>

SELECT ?label ?comment ?producer ?productFeature ?propertyTextual1 ?propertyTextual2
?propertyTextual3 ?propertyNumeric1 ?propertyNumeric2 ?propertyTextual4 ?propertyTextual5
?propertyNumeric4 WHERE {
  bsbm-inst:dataFromProducer8/Product360   rdfs:label                 ?label .
  bsbm-inst:dataFromProducer8/Product360   rdfs:comment               ?comment .
  bsbm-inst:dataFromProducer8/Product360   bsbm:producer              ?p .
  ?p                                       rdfs:label                 ?producer .
  bsbm-inst:dataFromProducer8/Product360   dc:publisher               ?p .
  bsbm-inst:dataFromProducer8/Product360   bsbm:productFeature        ?f .
  ?f                                       rdfs:label                 ?productFeature .
  bsbm-inst:dataFromProducer8/Product360   bsbm:productPropertyTextual1  ?propertyTextual1 .
  bsbm-inst:dataFromProducer8/Product360   bsbm:productPropertyTextual2  ?propertyTextual2 .
  bsbm-inst:dataFromProducer8/Product360   bsbm:productPropertyTextual3  ?propertyTextual3 .
  bsbm-inst:dataFromProducer8/Product360   bsbm:productPropertyNumeric1  ?propertyNumeric1 .
  bsbm-inst:dataFromProducer8/Product360   bsbm:productPropertyNumeric2  ?propertyNumeric2 .
OPTIONAL
    { bsbm-inst:dataFromProducer8/Product360   bsbm:productPropertyTextual4   ?propertyTextual4 }
 OPTIONAL
    { bsbm-inst:dataFromProducer8/Product360   bsbm:productPropertyTextual5   ?propertyTextual5 }
 OPTIONAL
    { bsbm-inst:dataFromProducer8/Product360   bsbm:productPropertyNumeric4  ?propertyNumeric4 }
}
```

Figure 3.5: BSBM Query 2

We use BSBM query 2 (Figure 3.5) to demonstrate how our query compiler works with OPTIONAL clause (Figure 3.6).

Temporary View bsbm_inst_dataFromProducer8_Product360Table:
SELECT rowKey **AS** product, rdfs_label **AS label**, rdfs_comment **AS comment**, bsbm_producer **AS** producer, bsbm_productFeature **AS** feature, bsbm_productPropertyTextual1 **AS** propertyTextual1, bsbm_productPropertyTextual2 **AS** property Textual2, bsbm_productPropertyTextual3 **AS** property Textual3, bsbm_productPropertyTextual4 **AS** propertyTextual4, bsbm_productPropertyTextual5 **AS** propertyTextual5, bsbm_productPropertyNumeric1 **AS** propertyNumeric1, bsbm_productPropertyNumeric2 **AS** propertyNumeric2, bsbm_productPropertyNumeric4 **AS** propertyNumeric4
FROM hbaseTable **WHERE** rdfs_label **IS NOT NULL AND** rdfs_comment **IS NOT NULL AND** bsbm_producer **IS NOT NULL AND** bsbm_productPropertyTextual1 **IS NOT NULL AND** bsbm_productPropertyTextual2 **IS NOT NULL AND** bsbm_productPropertyTextual3 **IS NOT NULL AND** bsbm_productPropertyNumeric1 **IS NOT NULL AND** bsbm_productPropertyNumeric2 **IS NOT NULL**

Temporary View pTable:
SELECT rowKey **AS** p, rdfs_label **AS label FROM** hbaseTable **WHERE** rdfs_label **IS NOT NULL**

Temporary View fTable:
SELECT rowKey **AS** f, rdfs_label **AS label FROM** hbaseTable **WHERE** rdfs_label **IS NOT NULL**

Final Spark **SQL** statement:
SELECT productTable.**label AS label**, productTable.**comment AS comment**, pTable.**label AS** producer, fTable.**label AS** feature, productTable.propertyTextual1 **AS** propertyTextual1, productTable.propertyTextual2 **AS** propertyTextual2, productTable.propertyTextual3 **AS** propertyTextual3, productTable.propertyNumeric1 **AS** propertyNumeric1, productTable.propertyNumeric2 **AS** propertyNumeric2, productTable.propertyTextual4 **AS** propertyTextual4, productTable.propertyTextual5 **AS** propertyTextual5, productTable.propertyNumeric4 **AS** propertyNumeric4
FROM productTable **JOIN** pTable **ON** (productTable.producer = pTable.p) **JOIN** fTable **ON** (ARRAY_CONTAINS(productTable.feature, fTable.f))

Figure 3.6: Compiler Translation of BSBM Query 2

HBase supports multi-valued attributes by using their corresponding timestamp. We use array data type to access multi-valued attributes.

```
PREFIX  bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX  bsbm:      <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX  rdfs:      <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  rdf:       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?product ?label WHERE {
  ?product     rdfs:label                    ?label .
  ?product     a                             bsbm-inst:ProductType1 .
  ?product     bsbm:productFeature           bsbm-inst:ProductFeature1 .
  ?product     bsbm:productFeature           bsbm-inst:ProductFeature2 .
  ?product     bsbm:productPropertyNumeric1  ?value1 .
  FILTER (?value1 > 450)
}
ORDER BY ?label
LIMIT 10
```

Figure 3.7: BSBM Query 1

```
Temporary View productTable:
SELECT rowKey AS product, rdfs_label AS label, rdf_type, bsbm_productFeature,
bsbm_productPropertyNumeric1
FROM hbaseTable WHERE rdfs_label IS NOT NULL, rdf_type IS NOT NULL, bsbm_productFeature IS
NOT NULL, bsbm_productPropertyNumeric1 IS NOT NULL

Final Spark SQL statement:
SELECT DISTINCT productTable.product AS product, productTable.label AS label
FROM productTable WHERE ARRAY_CONTAINS(productTable.rdf_type, 'bsbm-inst:ProductType1')
AND  ARRAY_CONTAINS(productTable.bsbm_productFeature, 'bsbm-inst:ProductFeature1')  AND
ARRAY_CONTAINS(productTable.bsbm_productFeature,  'bsbm-inst:ProductFeature2') AND
productTable.bsbm_productPropertyNumeric1 > 450
ORDER BY label
LIMIT 10
```

Figure 3.8: Compiler Translation of BSBM Query 1

We use BSBM query 1 (Figure 3.7) to demonstrate how our query compiler handles multi-valued attributes. Before the query translation process, we have to provide the multi-valued attributes as input to the compiler. In this case, rdf:type and bsbm:productFeature are the multi-valued attributes. First, we have to use row object to create temporary view of HBase table similar to query 10. Then the query compiler translates the SPARQL query to Spark SQL as shown in Figure 3.8.

SPARQL Translation for Cassandra data layout. To define a mapping from SPARQL to Spark SQL based on Cassandra data layout, algorithm 3 is used. It takes *blocks*, *op*, and *proj* as input, translates and processes every triple pattern (tp_i) and filter pattern (f_i) in each block in an iterative manner. First, it starts processing triple pattern (tp_i) by translating it to a tuple of <*s, p, o*> and then iterate through all the tuples to find *join* condition among predicates and conditions for *whereClause.* After processing all tuples of that block, it starts processing filter constraints (f_i) for that block and then adds the filter constraints to the conditions to produce a *whereClause*. Next, *fromClause* is combined with *join_conditions* to produce the final *fromClause* for that block. The *getBlockJoin* method is used to produce LEFT OUTER JOIN condition between two blocks. The *getSelectClauseLeftJoin* and *getSelectClauseUnion* methods are used to produce final select clauses for LEFT OUTER JOIN and UNION respectively. Finally, the algorithm generates the SQL statement for the input SPARQL Query. BSBM query 7 (Figure 3.9) is used to demonstrate SPARQL to Spark SQL translation for Cassandra RDF data. The compiler maps SPARQL to Spark SQL statement as shown in Figure 3.10.

Algorithm 3: Generate VP SQL

Input: *blocks, op, proj*

Output: *query*

1: *query = “”*
2: *aliaseMap = map<String, String>*
3: *blockQuery* $\leftarrow \phi$
4: *m* $\leftarrow$ *getTripleCount(blocks)*
5: **if** *n* > *1*
6: *buildJoin* $\leftarrow$ *true*
7: **end if**
8: **for** *k = 1 to k = n do*
9: *projections, conditions* $\leftarrow \phi$
10: *fromClause* $\leftarrow \phi$
11: *whereClause* $\leftarrow \phi$
12: **for** *i = 1 to m do*
13: **if** *b == k* // b is the block number
14: *join_conditions* $\leftarrow \phi$
15: **if** *isVar(tp_i.Subject) then*
16: *projections* $\leftarrow$ *projections* $\cup$ *(tp_i.Subject* $\rightarrow$ *T_i.Subject)*
17: *aliaseMap[T_i.Subject] = tp_i.Subject*
18: **else**
19: *conditions* $\leftarrow$ *conditions* $\cup$ *(T_i.Subject = tp_i.Subject)*
20: **end if**
21: **if** *isVar(tp_i.Object) then*
22: *projections* $\leftarrow$ *projections* $\cup$ *(tp_i.Object* $\rightarrow$ *T_i.Object)*
23: *aliaseMap[T_i.Object] = tp_i.Object*
24: **else**
25: *conditions* $\leftarrow$ *conditions* $\cup$ *(T_i.Object = tp_i.Object)*
26: **end if**
27: **for** *j = m to j = 1 do*
28: **if** *b == k*
29: **if** *$tp_{i == 1}$*
30: *fromClause* $\leftarrow$ *predicate(i) at tp_i assign to aliases T_i*
31: **if** *isPresent(f_i) then*
32: *conditions* $\leftarrow$ *conditions* $\cup$ *f_i*
33: **end if**
34: **else**

```
35:                     fromClause ← concate_str(fromClause, "JOIN", predicate(i) at tp_i assign to
        aliases T_i)
36:                 if isVar(tp_i.Subject)
37:                     if tp_i.Subject == tp_j.Subject
38:                         join_conditions ← join_conditions ∪ (T_i.Subject == T_j.Subject)
39:                     end if
40:                     if tp_i.Subject == tp_j.Object
41:                         join_conditions ← join_conditions ∪ (T_i.Subject == T_j.Object)
42:                     end if
43:                 end if
44:                 if isVar(tp_i.Object)
45:                     if tp_i.Object == tp_j.Subject
46:                         join_conditions ← join_conditions ∪ (T_i. Object == T_j.Subject)
47:                     end if
48:                     if tp_i.Object == tp_j.Object
49:                         join_conditions ← join_conditions ∪ (T_i. Object == T_j.Object)
50:                     end if
51:                 end if
52:             end if
53:         end if
54:     end for
55:     if isPresent(f_i) then
56:         if isContains(f_i, "!bound")
57:             op[i] = "EXCEPT"
58:         else
59:             whereClause ← conditions ∪ f_i
60:         end if
61:     end if
62:     fromClause ← fromClause ∪ join_conditions
63: end if
64: end for
65: if (buildJoin = true) then
66:     tempQuery ← getTempQuery(projections, fromClause, whereClause)
67:     Add tempQuery to vector blockQuery.
68: else
69:     selectClause ← getProjections(proj, isDistinct)
```

```
tempQuery ← getTempQuery(selectClause, fromClause, whereClause)
        query = tempQuery // query is the final result if SPARQL Query does not have operator
    OPTIONAL or UNION.
  end if
end for
if n > 1 then
   blockJoinCond ← getBlockJoin(proj, aliaseMap, blocks)
       blockJoinCond → {cond1, cond2, ..... condn-1}
   selectClauseLeftJoin ← getSelectClauseLeftJoin(proj, aliaseMap, blocks)
       selectClauseLeftJoin → {XK. projK}  // k ≤ n
   selectClauseUnion ← getSelectClauseUnion(proj)
    // selectClauseUnion is the string representation of the projected elements present in proj vector.
    for j = 1 to j = n – 1 do
       if op[j] == "LEFT OUTER JOIN" || op[j] == "EXCEPT"
           if j == 1 then
               query += concat_str(selectClauseLeftJoin, blockQuery[j-1], op[j], blockQuery[j],
         cond[j-1])
           else
               query += concat_strQuery(op[j], blockQuery[j], cond[j-1])
           end if
       end if
       if op[j] == 'UNION ALL' then
           if j == 1 then
               query += concat_str(selectClauseUnion, blockQuery[j-1], op[j], blockQuery[j])
           else
               query += concat_strQuery(op[j], blockQuery[j], "")
           end if
       end if
    end for
end if
return query
```

```
PREFIX  bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX  rdfs:      <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  rev:       <http://purl.org/stuff/rev#>
PREFIX  foaf:      <http://xmlns.com/foaf/0.1/>
PREFIX  bsbm:      <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX  dc:        <http://purl.org/dc/elements/1.1/>

SELECT ?productLabel ?offer ?price ?vendor ?vendorTitle ?review ?revTitle  ?reviewer ?revName
?rating1 ?rating2 WHERE {
  bsbm-inst:dataFromProducer8/Product360     rdfs:label     ?productLabel .
   OPTIONAL {
       ?offer        bsbm:product            bsbm-inst:dataFromProducer8/Product360 .
       ?offer        bsbm:price              ?price .
       ?offer        bsbm:vendor             ?vendor .
       ?vendor       rdfs:label              ?vendorTitle .
       ?vendor       bsbm:country            <http://downlode.org/rdf/iso-3166/countries#DE> .
       ?offer        dc:publisher            ?vendor .
       ?offer        bsbm:validTo            ?date .
       FILTER (?date > "2008-07-08" ^^xsd:date)
 }
   OPTIONAL {
       ?review       bsbm:reviewFor          bsbm-inst:dataFromProducer8/Product360 .
       ?review       rev:reviewer            ?reviewer .
       ?reviewer     foaf:name               ?revName .
       ?review       dc:title                ?revTitle .
       OPTIONAL { ?review     bsbm:rating1     ?rating1 . }
       OPTIONAL { ?review     bsbm:rating2     ?rating2 . }
    }
}
```

Figure 3.9: BSBM Query 9

```
SELECT X0.productLabel, X1.offer, X1.price, X1.vendor, X1.vendorTitle, X4.review, X2.reviewer,
X2.revName, X2.revTitle, X3.rating1, X4.rating2
FROM  (SELECT T0.Subject AS Product, T0.Object AS productLabel FROM rdfs_label T0 WHERE
T0.Subject = 'bsbm-inst:dataFromProducer8/Product360') AS X0
LEFT OUTER JOIN (SELECT T1.Object AS Product, T7.Object AS date, T7.Subject AS offer,
T2.Object AS price, T6.Object AS vendor, T4.Object AS vendorTitle FROM bsbm_product T1 JOIN
bsbm_price T2 ON (T2.Subject = T1.Subject)  JOIN bsbm_vendor T3 ON (T3.Subject = T2.Subject)
JOIN rdfs_label T4  ON (T4.Subject = T3.Object) JOIN bsbm_country T5 ON  (T5.Subject =
T4.Subject) JOIN dc_publisher T6 ON  (T6.Object = T5.Subject AND T6.Subject = T3.Subject) JOIN
bsbm_validTo T7 ON (T7.Subject = T6.Subject) WHERE T1.Object = 'bsbm-
inst:dataFromProducer8/Product360' AND T5.Object =
'http://downlode.org/rdf/iso3166/countries#DE'  AND T7.Object > 20080708) AS X1 ON
(X0.Product = X1.Product)
LEFT OUTER JOIN (SELECT T8. Object AS Product, T10. Object AS revName,  T11.Object AS
revTitle, T11.Subject AS review, T10.Subject AS reviewer
FROM bsbm_reviewFor T8 JOIN rev_reviewer T9 ON (T9.Subject = T8.Subject) JOIN foaf_name T10
ON (T10.Subject = T9.Object) JOIN dc_title T11 ON  (T11.Subject = T9.Subject) WHERE T8.Object =
'bsbm-inst:dataFromProducer8/Product360') AS X2  ON (X0.Product = X2. Product) LEFT OUTER
JOIN  (SELECT T12.Object AS rating1, T12.Subject AS review FROM bsbm_rating1 T12) AS X3 ON
(X2.review = X3.review) LEFT OUTER JOIN (SELECT T13.Object AS rating2, T13.Subject AS review
FROM bsbm_rating2 T13) AS X4 ON (X2.review = X4.review)
```

Figure 3.10: Compiler Translation of BSBM Query 7

The query compiler supports basic SPARQL queries. The Optional graph pattern and Alternate graph pattern are mapped to LEFT OUTER JOIN and UNION ALL in Spark SQL respectively. It supports FILTERS, ORDER BY, DISTINCT, LIMIT modifiers. This compiler does not support DESCRIBE and CONSTRUCT clauses. So, we perform a manual translation for query performance evaluation. There is a difference between HBase and Cassandra RDF data layout for mapping SPARQL function !Bound(Object) to Spark SQL statement. In this case, HBase uses `Object IS NULL` whereas Cassandra uses EXCEPT clause. To demonstrate the translation process of SPARQL function !Bound(Object) to Spark SQL statement using EXCEPT clause, we

used BSBM query 3 (Figure 3.11) to demonstrate the translated SQL statements (Figure 3.12).

```
PREFIX  bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX  bsbm:      <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX  rdfs:      <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  rdf:       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?product ?label WHERE {
  ?product    rdfs:label                    ?label .
  ?product    a                             bsbm-inst:ProductType1 .
  ?product    bsbm:productFeature           bsbm-inst:ProductFeature100 .
  ?product    bsbm:productPropertyNumeric1  ?p1 .
  FILTER ( ?p1 > 730 )
  ?product    bsbm:productPropertyNumeric3  ?p3 .
  FILTER (?p3 < 380 )
  OPTIONAL {
     ?product    bsbm:productFeature        bsbm-inst:ProductFeature200 .
     ?product    rdfs:label                 ?testVar }
     FILTER (!bound(?testVar))
}
ORDER BY ?label
LIMIT 10
```

Figure 3.11: BSBM Query 3

```
SELECT T0.Subject AS product, T0.Object AS label
FROM rdfs_label AS T0 JOIN rdf_type AS T1 ON T0.Subject = T1.Subject JOIN bsbm_productFeature
AS T2 ON T0.Subject = T2.Subject JOIN bsbm_productPropertyNumeric1 T3 ON T0.Subject =
T3.Subject JOIN bsbm_productPropertyNumeric3 AS T4 ON T0.Subject = T4.Subject WHERE
T1.Object = 'bsbm-inst:ProductType1' AND  T2.Object = 'bsbm-inst:ProductFeature100' AND T3.
Object > 730 AND T4. Object < 380 EXCEPT SELECT T5.Subject AS product, T6. Object AS label
FROM bsbm_productFeature AS T5 JOIN rdfs_label AS T6  ON T5.Subject = T6.Subject WHERE
T5.Object = 'bsbm-inst:ProductFeature200'
ORDER BY label
LIMIT 10
```

Figure 3.12: Compiler Translation of BSBM Query 3

3.1.3 Experimental Setup

In this section, we present a comparative performance evaluation of HBase and Cassandra systems conducted to benchmark with Spark SQL. The experimental setup followed by a discussion of the results is presented.

Benchmark Queries. For the performance evaluation of two systems, we utilize two datasets. The Berlin SPARQL Benchmark (Bizer & Schultz, 2009) is built around e-commerce use-case where a number of vendors offer a set of products to the consumers and the consumers post their reviews about the products. The BSBM queries are based on the real-world use cases that simulate the search patterns of a consumer looking for a given product. We have selected all BSBM benchmark queries for performance evaluation. Characteristics of the BSBM benchmark queries are presented in (Bizer & Schultz, 2009). The SP^{2}Bench (Schmidt et al., 2009) is a SPARQL performance benchmark suite designed to cover the most important SPARQL constructs and operator constellations using a range of RDF data access pattern as well as SPARQL-to-SQL re-write systems. The data model of SP^{2}Bench is based on DBLP (Digital Bibliography & Library Project), a computer science bibliography website started in 1993 and currently hosts more than 2.3 million journal articles. From SP^{2}Bench we have selected 1, 2, 3a, 5a, 6, 8 and 11. Query 1 is simple with only one joining variable. Queries 2, 6 and 8 have OPTIONAL blocks. Queries 3a and 5a both have one FILTER operator and query 11 has LIMIT, OFFSET and ORDER BY modifiers. Both BSBM and SP^{2}Bench have data generator that can generate any number of triples based on user specification.

Cluster Configuration. To conduct the comparative analysis between two systems, we constructed seven node clusters on Microsoft Windows Azure Platform.

Each node in the cluster had 2 vCPUs Intel(R) Xeon(R) CPU E5-2673 v3 @ 2.40 GHz processor, 14 GB of memory, 6400 Max IOPS, 4 data disks and total 512 GB of hard disk space running Ubuntu 16.04.3 LTS OS. Hadoop 2.7.4 and Spark 2.2.0 were configured on all nodes where each Spark executor was given 8 GB of memory. For HBase system settings, we used HBase 1.3.1 in the cluster configuration. For distributed applications, HBase relies on a high-performance coordination service called ZooKeeper. In our system configuration, we used built-in Zookeeper of HBase that is dedicated to the master. For Cassandra system settings, we used Apache Cassandra 3.11 with a uniform cluster name and IP configuration set to all nodes. Each Cassandra instance can equally hold a maximum of 256 index tokens.

3.1.4 Evaluation

Initially, the RDF dataset was copied into HDFS; a Spark job was used by RDF-loader to parse and convert raw RDF data in order to load into HBase and Cassandra storage schemas. The following result demonstrates the performance of RDF Loader in parsing and loading 10 and 20 million triples for HBase and Cassandra storage strategies (Figure 3.13 and 3.14). After loading triples into HBase and Cassandra storage systems, we ran benchmark queries described in section 3.1.3 on a 7-Node cluster.

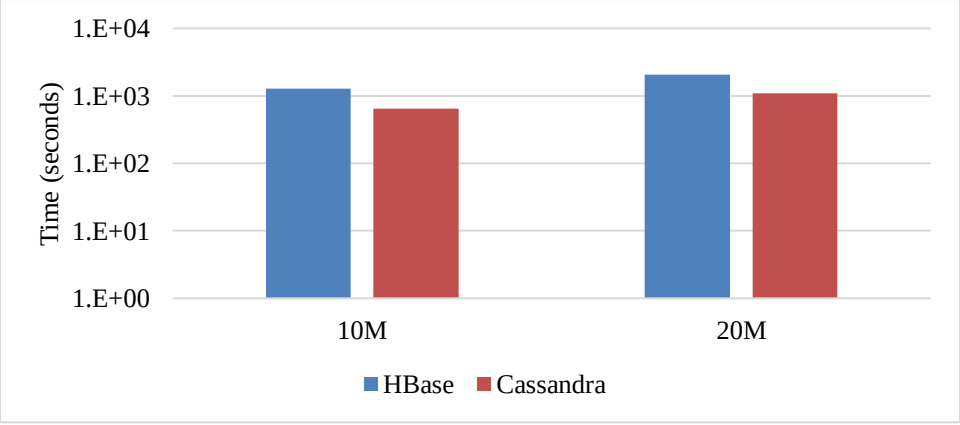

Figure 3.13: BSBM RDF Dataset Loading Time for 10 and 20 Million Triples (log scale)

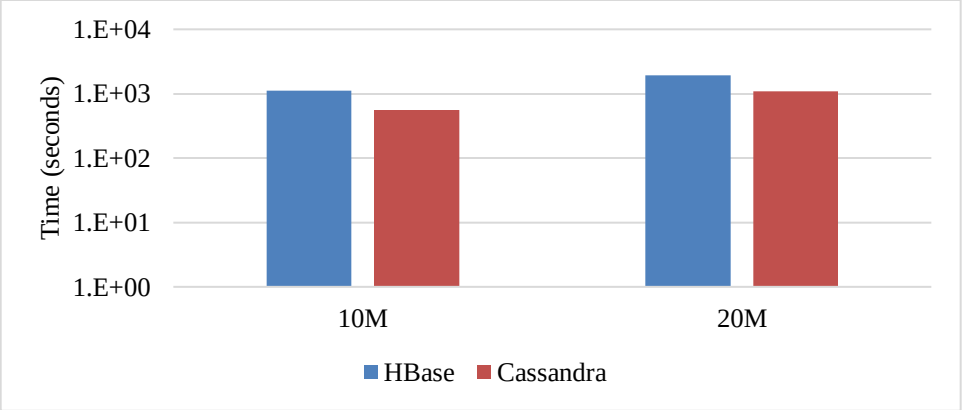

Figure 3.14: SP^{2}Bench RDF Dataset Loading Time for 10 and 20 Million Triples (log scale)

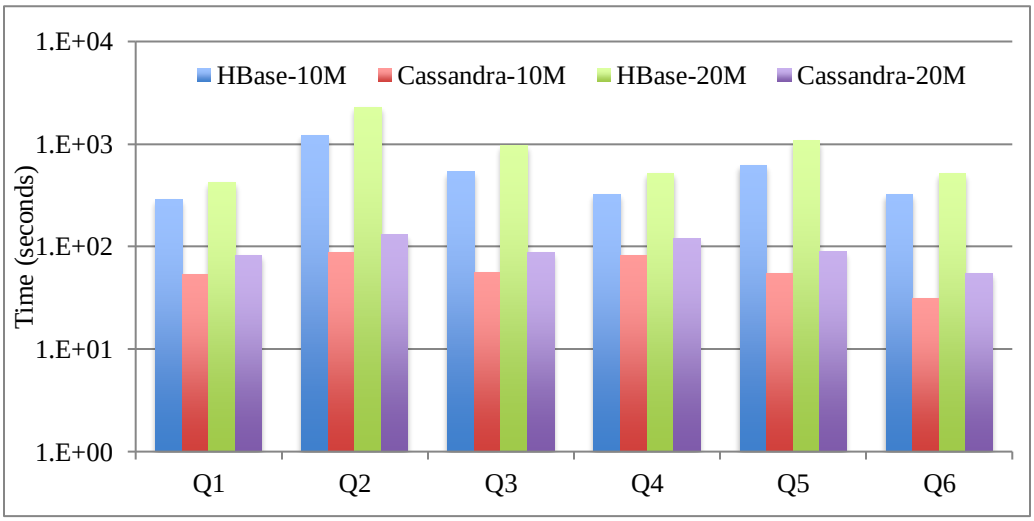

Figure 3.15: Query Runtimes - BSBM Queries [Q1 – Q6] (log scale)

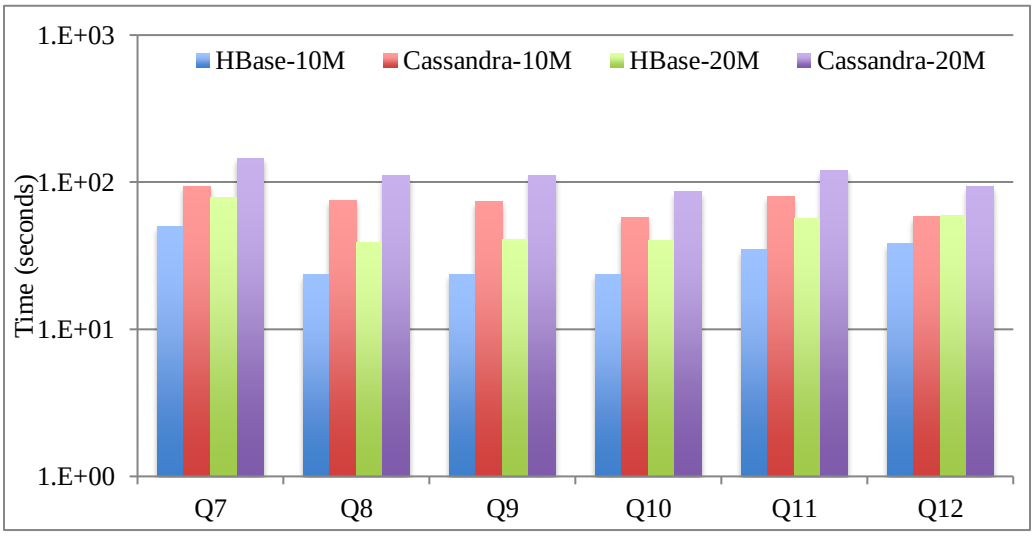

Figure 3.16: Query Runtimes - BSBM Queries [Q7 – Q12] (log scale)

Performance comparison between HBase and Cassandra storage schemas for BSBM queries on 7-node cluster is shown in Figure 3.15 [Q1 – Q6] and 3.16 [Q7 – Q12] of 10 million and 20 million triples. We ran each query 3 times and took the average response time. Cassandra outperforms HBase in query response time of queries from Q1 to Q6. However, for queries Q7 through Q12, HBase is faster than Cassandra. Query response time depends on the size (number of rows) of each temporary view. Besides, HBase has to handle multi-valued attributes via array data type and performing operations on array data type were time-consuming. Most of the queries from Q1 to Q6 involved array data type in HBase and the size of the subject temporary views are larger than the size of property temporary views in Cassandra; therefore, Cassandra outperforms HBase. However, from query Q7 to Q12, HBase outperforms Cassandra, as it requires less number of *joins* with subject views. On the other hand, for all of the *join* conditions in a query BGP, all property table views have to participate in Cassandra storage scheme.

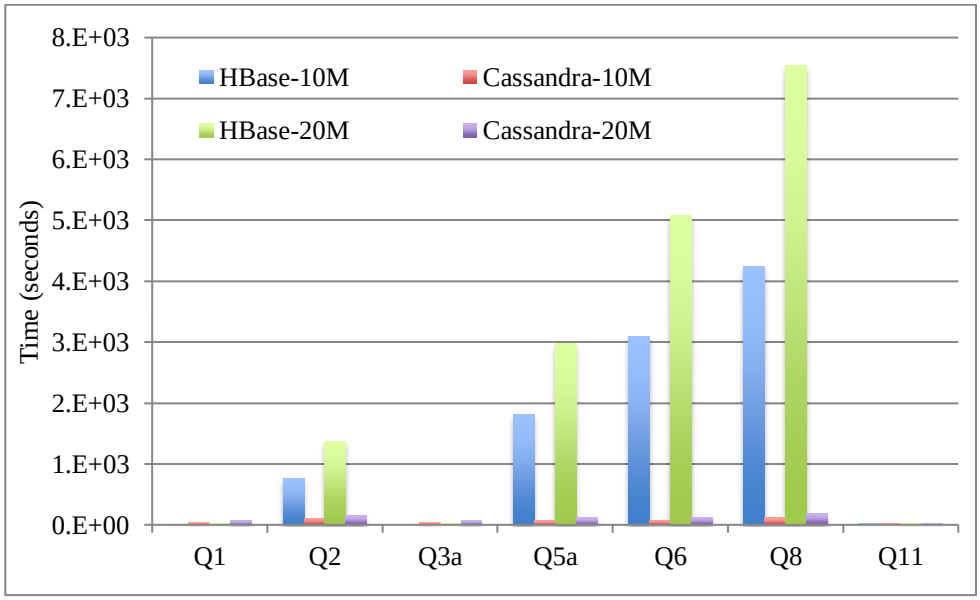

Figure 3.17: Query Runtimes - SP^{2}Bench Queries (log scale)

Figure 3.17 shows the SP^{2}Bench query response time of 10 million and 20 million triples using HBase and Cassandra storage schemas. Response time for query 1, 3a, and 11 are

almost same for both HBase and Cassandra system - a few seconds, however, query response time for HBase storage schema is noticeably higher on queries 2, 5a, 6 and 8. Queries 2 and 6 have OPTIONAL blocks. Query 2 has ten triple patterns with one unique subject. For HBase storage schema, the generated Spark SQL statement for query 2 gets rid of the complexity of the *joins* from the original query by not including LEFT OUTER JOIN from the final query statement. Although we have eliminated *join*, the size of the subject temporary view is still very large that requires many rows to be scanned to get final result as compared to Cassandra storage schema with property table *joins*. Query 5a has six triple patterns and one FILTER. Queries 6 and 8 have more triple patterns and filters than query 5a. Query response time of query 8 that involves multiple *joins*, and a UNION is very high on HBase storage schema as compared to Cassandra because the size of subject temporary views is greater than the size of property temporary views.

3.1.5 Conclusion

We present new data models for storing RDF data in HBase and Cassandra that are different from the existing NoSQL based RDF management systems. Further optimization of indexing schemes and tuning for efficient retrieval capabilities on both databases can be explored. A query compiler that translates SPARQL queries to Spark SQL statements using RDF data layouts for the two systems has been presented. Comparison of query performance of HBase and Cassandra systems is presented with a discussion of the results. Results show that HBase outperforms Cassandra for queries involving one subject (star-shaped query). On the other hand, Cassandra performs better on queries with multiple subjects because of vertical partitioning data storage schema.

3.2 RDF Management with VPExp and 3CStore Data Layouts

In an RDF dataset, the number of distinct predicates is often relatively few as compared to the number of distinct subjects or objects. Based on this observation, Abadi, et al. introduced Vertically Partitioned (VP) architecture in (Abadi et al., 2007) by storing the tripe in files named after predicates whose content keeps only subject and object entries. To provide more efficient relational data layouts by taking advantage of VP schema, we propose two relational data layouts for RDF graphs called VPExp and 3CStore. The first approach, VPExp, is based on the concept of splitting predicates using explicit type information of the object introduced in (Husain et al., 2011). The second approach, 3CStore, is primarily designed to minimize the *join* operations and data communication costs for SPARQL queries over distributed systems by creating sub-tables from VP table based on different *join* correlations between triple patterns.

3.2.1 Data Modeling

Our first RDF data partitioning approach VPExp is based on the concept of minimizing input data size for a particular predicate *rdf:type* during query evaluation. The *rdf:type* predicate is very common in RDF datasets and usually contains a lot of rows while doing Vertical Partitioning. Therefore, we decided to further split that predicate into a number of distinct objects the predicate has. Suppose the *rdf:type* predicate has o_1, o_2,…..,o_n number of distinct objects then VPExp will create rdf_type_o_1, rdf_type_o_2,……, rdf_type_o_n tables along with the original rdf_type table. We use the following query to determine unique objects of *rdf:type* predicate.

SELECT DISTINCT o **FROM** TripleTable **WHERE** p = 'rdf:type'

For each *rdf:type* object (say, obj) we use the following query to create VPExp tables.

SELECT DISTINCT s **FROM** TripleTable **WHERE** p = 'rdf:type' **AND** o = obj

We name the table as *rdf_type_obj*. This approach takes advantage by minimizing the input data size of *rdf:type* predicate table when the triple pattern of a SPARQL query contains *rdf:type* in predicate position and the corresponding object is not a variable. The second approach 3CStore is a three-column data layout. The goal of this storage model is to minimize the input data size as well as *join* operations during query evaluation. Generally, in VP approach the *join* variable that occurs in subject or object position between two triple patterns in SPARQL query determines the columns on which the corresponding VP tables have to be joined. Four possible correlations (Schätzle et al., 2015) exist for position of the *join* variable that can occur between two triple patterns based on their corresponding subject and object positions. If both triple patterns have the *join* variable in their subject position, then it is subject-subject correlation (SS). Similarly, other three correlations are subject-object (SO), object-subject (OS), and object-object (OO). Based on these correlations we pre-compute a subset of the VP table with all other VP tables using *inner join* and create three-column stores for all correlations where the second column (column2) contains the correlation value, the first column (column1) contains subject or object values that are not common between VPs and from the first VP table $VP_{P1,}$ and the third column (column3) contains the remaining subject or object values from the second VP table VP_{P2}. The file is named as (correlation name)_p_1_p_2. Suppose VP_{P1} has (s_1, o_1) and VP_{P2} has (s_2, o_2) and if we are computing SS correlation then s_1 must be equal to s_2. In this case, we call that common value as ss and store in 3CStore approach as (o_1, ss, o_2) and materialized that table named as ss_p_1_p_2 unless the table is empty. For example, we use the following query to determine the SS correlation

of VP_{P1} with VP_{P2}.

SELECT t1.o **AS** column1, t1.s **AS** column2, t2.o **AS** column3 **FROM** VP_{P1} t1 **JOIN** VP_{P2} t2 **ON** t1.s = t2.s

Similarly, we can pre-compute all other correlations and materialized as 3CStore tables in HDFS.

Table 3.6: 3CStore Table Construction Using Correlations Between Triple Patterns

Correlation	Join Condition (name of common value)	3CStore (column1, column2, column3)	Table Name
SS	$s_1 = s_2$ (ss)	(o_1, ss, o_2)	ss_p_1_p_2
SO	$s_1 = o_2$ (so)	(o_1, so, s_2)	so_p_1_p_2
OS	$o_1 = s_2$ (os)	(s_1, os, o_2)	os_p_1_p_2
OO	$o_1 = o_2$ (oo)	(s_1, oo, s_2)	oo_p_1_p_2

We did not pre-compute 3CStore tables for OO correlations because the number of distinct objects in RDF dataset can be huge and create a lot of 3CStore files in HDFS, which will cost a significant amount of storage space. If there is no storage space issue, then one can pre-compute 3CStore tables for OO correlation also. For SPARQL translation with 3CStore approach VP tables can be utilized if there is no 3CStore table in HDFS.

3.2.2 Data Loading and Query Translation

We use Spark as the RDF loader and, for query processing, we use both Spark and Drill. We describe the data loading approach and SPARQL query translation to SQL for both Spark and Drill based on the three storage models, VP, VPExp, and 3CStore. In order to rewrite the SPARQL query to SQL, we developed a query compiler, which is implemented in Flex – a lexical analyzer creator, Bison – a parser generator and C++11.

RDF Loader. The goal of the RDF loader is to load, parse, and store RDF data into three different RDF storage schemas in HDFS. Before storing RDF data to HDFS, we parse the input dataset and apply various transformations. The transformation includes replacing all URIs with their corresponding namespace prefix and remove data type information from RDF object to convert that object into primitive type because Spark SQL and Drill both support complex and primitive data types. We use Parquet storage format to store RDF data into HDFS.

Query Translation. The compiler first validates the input query and builds a query parse tree from SPARQL grammar rules. Algorithm 4 shows how the translation process works. The compiler then translates the input query into two clauses, SELECT and WHERE clause. The projected variables are listed in the projection list from SELECT Clause. The WHERE Clause, which is basically a group graph pattern, consists of n number of BGP where $n \geq 1$ and are connected through $n - 1$ operators. The BGP connecting operator *op* is OPTIONAL, UNION, and DOT where DOT operator indicates an implicit conjunction between two BGPs. We can simply write the WHERE clause as $\{BGP_1, op_1, BGP_2, op_2, \ldots op_{n-1}, BGP_n\}$. Each BGP consists of triple pattern tp_i and filter pattern *fi* as follows $BGP_1 = \{(tp_1, f_1), (tp_2, f_2), \ldots\ldots\}$. The BGP connecting operators OPTIONAL, UNION, and DOT are replaced by LEFT OUTER JOIN, UNION ALL, and JOIN respectively when SPARQL query is translated into SQL statement. SPARQL queries can have predicate as a variable. The *tripleRewriting* method replaces the predicate variable with a bound value that is found in a filter pattern for that variable and then discards the FILTER from that BGP. The *tripleRearranging* method rearranges each triple pattern in the BGP in such a way that *join* condition between two triple patterns can

be found in successive order to avoid further cross *join* from SQL statement.

Algorithm 4: SPARQL Query Translation

Input: *sparqlQuery*

Output: *bgp: vector<BGP>, op: vector<Operator>, projection: vector<Projection>*

1: *{selectClause, whereClause}* ← *sparqlQuery*
2: Add each variable of *selectClause* to project list
3: *{bgp, op}* ← *whereClause*
4: *bgp* → *{BGP$_1$, BGP$_2$, ……., BGP$_n$}*
5: *op* → *{op$_1$, op$_2$, ……,op$_{n-1}$}*
6: *bgp* ← *tripleRewriting(bgp)*
7: *bgp* ← *tripleRearranging(bgp)*
8: return *bgp, op, projection*

Algorithm 5: BGP Translation for VP and VPExp Schema

Input: *BGP$_j$: vector<Triple Pattern: tp: (s, p, o)>*
stg: Name of the schema (*VP* or *VPExp*)

Output: *projection: Projection, from: vector<Table Name, Table alias>, condition: Where*

1: *projection* ← *ϕ; from* ← *ϕ; condition* ← *ϕ*
2: **for** *each triple pattern tp$_i$* ∈ *BGP$_j$ do*
3: **if** *tp$_i$.p = 'rdf:type'* ∧ *!isVar(tp$_i$.o)* ∧ *stg = VPExp* then
4: *from* ← *from* ∪ *(concat_str(tp$_i$.p, 'rdf:type', ''), T$_t$)* // T$_t$ is the unique alias of tp$_i$
5: **if** *isVar(tp$_i$.s)* then
6: *projection* ← *projection* ∪ *(tp$_i$.s* → *T$_t$.s)*
7: **else**
8: *condition* ← *condition* ∪ *(T$_t$.s = tp$_i$.s)*
9: **end if**
10: **else**
11: *from* ← *from* ∪ *(tp$_i$.p, T$_i$)*
12: **if** *isVar(tp$_i$.s)* then
13: *projection* ← *projection* ∪ *(tp$_i$.s* → *T$_t$.s)*
14: **else**
15: *condition* ← *condition* ∪ *(T$_t$.s = tp$_i$.s)*
16: **end if**

```
17:        if isVar(tp_i.o) then
18:            projection ← projection ∪ (tp_i.o → T_t.o)
19:        else
20:            condition ← condition ∪ (T_t.o = tp_i.o)
21:        end if
22:    end if
23: end for
24: return projection, from, condition
```

For VP and VPExp approaches, tables are selected from the predicate position of each triple pattern. In 3CStore approach, tables are selected from the combination of two triple patterns if there is any existing correlation, otherwise, the table is selected just like in VP approach. The BGP translation of VP and VPExp is shown in algorithm 5, and algorithm 6 shows BGP translation of 3CStore storage layouts. Each algorithm takes a BGP as input and generates a list of projections, conditions, and from; from is the table names with their corresponding unique aliases, which are used to generate subqueries for triple patterns. These subqueries are combined together to compute the SQL statement for each BGP. These SQL statements are then combined with the generated projection(s) and op (operators) from algorithm 1 to generate the final SQL statement. For all three storage approaches, if the predicate position remains a variable and no filter pattern presents in the BGP to replace that variable, in that case, the base triple table will be selected. For the VPExp schema with predicate position as *'rdf:type'* and the object is a bound value, all bound values from triple patterns and filter constraints are used as conditions in WHERE clause. The table columns are renamed by their corresponding variable name in the SELECT clause of SQL statement.

Algorithm 6: BGP Translation for 3CStore Schema

Input: *BGP_j : vector<tp: (s, p, o)>*

Output: *projection: Projection, from: vector<Table Name, Table alias>, condition: Where*

1: $n \leftarrow getTripleCount(BGP_j)$
2: $m, l = 0$
3: **if** $n > 1$
4: **for** $m = 0$ to $n/2 - 1$
5: $l = m + 1$
6: **if** $tp_m.s = tp_l.s$ then
7: $table \leftarrow table \cup (concat\_str('ss\_', tp_m.s, tp_l.s), T_t)$ // T_t is the unique alias of tp_i
8: **if** $isVar(tp_m.o)$ then
9: $projection \leftarrow projection \cup (tp_m.o \rightarrow T_t.column1)$
10: **else**
11: $condition \leftarrow condition \cup (T_t.column1 = tp_m.o)$
12: **end if**
13: **if** $isVar(tp_l.o)$ then
14: $projection \leftarrow projection \cup (tp_l.o \rightarrow T_t.column3)$
15: **else**
16: $condition \leftarrow condition \cup (T_t.column3 = tp_l.o)$
17: **end if**
18: **end if**
19: **if** $tp_m.s = tp_l.o$ then
20: $table \leftarrow table \cup (concat\_str('so\_', tp_m.s, tp_l.o), T_t)$
21: **if** $isVar(tp_m.o)$ then
22: $projection \leftarrow projection \cup (tp_m.o \rightarrow T_t.column1)$
23: **else**
24: $condition \leftarrow condition \cup (T_t.column1 = tp_m.o)$
25: **end if**
26: **if** $isVar(tp_l.s)$ then
27: $projection \leftarrow projection \cup (tp_l.s \rightarrow T_t.column3)$
28: **else**
29: $condition \leftarrow condition \cup (T_t.column3 = tp_l.s)$
30: **end if**
31: **end if**
32: **if** $tp_m.o = tp_l.s$ then

33: *table* ← *table* ∪ *(concat_str('os_',tp_m.o, tp_l.s), T_t)*

34: **if** *isVar(tp_m.s)* then

35: *projection* ← *projection* ∪ *(tp_m.s → T_t.column1)*

36: **else**

37: *condition* ← *condition* ∪ *(T_t.column1 = tp_m.s)*

38: **end if**

39: **if** *isVar(tp_l.o)* then

40: *projection* ← *projection* ∪ *(tp_l.o → T_t.column3)*

41: **else**

42: *condition* ← *condition* ∪ *(T_t.column3 = tp_l.o)*

43: **end if**

44: **end if**

45: *m = m + 2*

46: **end for**

47: **end if**

48: **if** *n%2 != 0 || n = 1* then

// find the last triple pattern if the total count of triple pattern is odd or there is only one triple pattern in the bgp

49: *table* ← *table* ∪ *(tp_{n-1}.p, T_t)*

50: **if** *isVar(tp_{n-1}.s)* then

51: *projection* ← *projection* ∪ *(tp_{n-1}.s → T_t.s)*

52: **else**

53: *condition* ← *condition* ∪ *(T_t.s = tp_{n-1}.s)*

54: **end if**

55: **if** *isVar(tp_{n-1}.o)* then

56: *projection* ← *projection* ∪ *(tp_{n-1}.o → T_t.o)*

57: **else**

58: *condition* ← *condition* ∪ *(T_t.o = tp_{n-1}.o)*

59: **end if**

60: **end if**

61: return *projection, from, condition*

The translation process of the three data models is given below using query 4[26] from LUBM in the following example. It is important to mention here that we replace

[26] http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt

colon (:) with underscore (_) because SQL statements do not support colon except in String values. We append the HDFS path of tables with the file extension (parquet) for Drill SQL statements.

VP (Spark SQL):

SELECT T0.**s** **AS** X, T2.o **AS** Y1, T3.o **AS** Y2, T4.o **AS** Y3 **FROM** rdf_type T0 **JOIN** ub_worksFor T1 **ON** (T1.**s** = T0.**s**) **JOIN** ub_name T2 **ON** (T2.**s** = T0.**s**) **JOIN** ub_emailAddress T3 **ON** (T3.**s** = T0.**s**) **JOIN** ub_telephone T4 **ON** (T4.**s** = T0.**s**) **WHERE** T1.o = '<http://www.Department0.University0.edu>' **AND** T0.o = '<ub:FullProfessor>')

VP (Drill):

SELECT T0.**s** **AS** X, T2.o **AS** Y1, T3.o **AS** Y2, T4.o **AS** Y3 **FROM** hdfs.vp.`rdf_type.parquet` T0 **JOIN** hdfs.vp.`ub_worksFor.parquet` T1 **ON** (T1.**s** = T0.**s**) **JOIN** hdfs.vp.`ub_name.parquet` T2 **ON** (T2.**s** = T0.**s**) **JOIN** hdfs.vp.`ub_emailAddress.parquet` T3 **ON** (T3.**s** = T0.**s**) **JOIN** hdfs.vp.`ub_telephone.parquet` T4 **ON** (T4.**s** = T0.**s**) **WHERE** T1.o = '<http://www.Department0.University0.edu>' **AND** T0.o = '<ub:FullProfessor>')

VPExp (Spark SQL):

SELECT T0.**s** **AS** X, T2.o **AS** Y1, T3.o **AS** Y2, T4.o **AS** Y3 **FROM** rdf_type_ub_FullProfessor T0 **JOIN** ub_worksFor T1 **ON** (T1.**s** = T0.**s**) **JOIN** ub_name T2 **ON** (T2.**s** = T0.**s** **AND** T2.**s** = T1.**s**) **JOIN** ub_emailAddress T3 **ON** (T3.**s** = T0.**s** **AND** T3.**s** = T1.**s** **AND** T3.**s** = T2.**s**) **JOIN** ub_telephone T4 **ON** (T4.**s** = T0.**s** **AND** T4.**s** = T1.**s** **AND** T4.**s** = T2.**s** **AND** T4.**s** = T3.**s**)
WHERE T1.o = '<http://www.Department0.University0.edu>'

VPExp (Drill):

SELECT T0.**s** **AS** X, T2.o **AS** Y1, T3.o **AS** Y2, T4.o **AS** Y3 **FROM** hdfs.vpexp.`rdf_type_ub_FullProfessor.parquet` T0 **JOIN** hdfs.vpexp.`ub_worksFor.parquet` T1 **ON** (T1.**s** = T0.**s**) **JOIN** hdfs.vpexp.`ub_name.parquet` T2 **ON** (T2.**s** = T0.**s** **AND** T2.**s** = T1.**s**) **JOIN** hdfs.vpexp.`ub_emailAddress.parquet` T3 **ON** (T3.**s** = T0.**s** **AND** T3.**s** = T1.**s** **AND** T3.**s** = T2.**s**) **JOIN** hdfs.vpexp.`ub_telephone.parquet` T4 **ON** (T4.**s** = T0.**s** **AND** T4.**s** = T1.**s** **AND** T4.**s** = T2.**s** **AND** T4.**s** = T3.**s**)
WHERE T1.o = '<http://www.Department0.University0.edu>'

3CStore (Spark SQL):

SELECT T0.column2 **AS** X, T1.column1 **AS** Y1, T1.column3 **AS** Y2, T2.column3 **AS** Y3 **FROM** ss_rdf_type_ub_worksFor T0 **JOIN** ss_ub_name_ub_emailAddress T1 **ON** (T0.column2 = T1.column2)

```
JOIN ub_telephone T2 ON (T0.column2 = T2.s) WHERE T0.column1 = '<ub:FullProfessor>' AND
T0.column3 = '<http://www.Department0.University0.edu>'
```

3CStore (Drill):

```
SELECT T0.column2 AS X, T1.column1 AS Y1, T1.column3 AS Y2, T2.column3 AS Y3 FROM
hdfs.threecstore.`ss_rdf_type_ub_worksFor .parquet` T0 JOIN
hdfs.threecstore.`ss_ub_name_ub_emailAddress.parquet` T1 ON (T0.column2 = T1.column2) JOIN
hdfs.threecstore.`ub_telephone.parquet` T2 ON (T0.column2 = T2.s) WHERE T0.column1 =
'<ub:FullProfessor>' AND T0.column3 = '<http://www.Department0.University0.edu>'
```

3.2.3 Experimental Setup

We present a comparative performance evaluation of our data management solutions VPExp, and 3CStore conducted to benchmark with Spark and Drill against traditional VP data layout and S2RDF. The experimental setup and a discussion of results are presented.

Benchmark Queries. For the performance evaluation of our data management solutions, we utilize two datasets, LUBM with the number of universities set to 1000 and 2000 and WatDiv with scale factor 1000 and 5000. LUBM was proposed in 2005 with fourteen standard queries. This benchmark was originally designed to test the inference capabilities of Semantic Web repositories. The University of Waterloo introduced WatDiv in 2014. WatDiv has data generator as well as query generator and was designed to cover both structural and data-driven features of four different types of query shapes, namely, linear, star, snowflake, and complex SPARQL queries.

Cluster Configuration. To conduct the comparative analysis of our data management solutions, we constructed five node clusters on Microsoft Windows Azure Platform. Each node in the cluster has a 4 vCPUs Intel(R) Xeon(R) CPU E5-2673 v3 @ 2.40 GHz processor, 28 GB of memory, 16000 Max IOPS, 8 Data disks and total 4 TB of

hard disk space running Ubuntu 16.04.3 LTS OS. Hadoop 2.7.4, Spark 2.2.0, ZooKeeper 3.4.10 and Drill 1.11 are configured on all nodes where each spark executor and drillbit is given 16 GB of memory. We also configure Zeppelin[27] 0.7.3 for query execution of our systems.

3.2.4 Evaluation

This section presents an empirical comparison of our approaches with an open source HDFS-based RDF management system S2RDF based on a data partitioning schema called ExtVP to load RDF data in HDFS and uses SparkSQL for query processing. Initially, the RDF dataset is copied into HDFS; RDF-loader uses a Spark job to parse, convert, and store the raw RDF data using Parquet file format in to three different storage schemas, namely, VP, VPExp, and 3CStore.

Table 3.7: Experimental Setup - Dataset Scale

Dataset		Number of Triples (million)
LUBM	Number of Universities	
	1000	138
	2000	276
WatDiv	Scale Factor	
	1000	109
	5000	549

[27] https://zeppelin.apache.org/

Table 3.8 Load Times

Dataset	Load Time (minutes)				
	VP	VPExp	3CStore	S2RDF SF = 0.5	S2RDF SF = 1
LUBM 1000	6.6	8.8	118.58	105.2	111.9
LUBM 2000	9.9	12.6	198.52	172.8	179.8
WatDiv 1000	5.1	6.7	248.4	159.8	214.9
WatDiv 5000	17.5	21.5	908.33	596.0	703.8

Table 3.7 shows the size of the datasets that we used in this experiment. Table 3.8 presents results that demonstrate the performance of RDF Loader in parsing and loading triples into three storage strategies (VP, VPExp, and 3CStore) on a 5-Node cluster. S2RDF has three components: DataSetCreator, QueryTranslator, and QueryExecutor. Before using DataSetCreator to load data into HDFS, we used our parser to replace all URIs with their corresponding namespace prefix and remove data type information from RDF objects to convert those objects into primitive type that we did in our proposed systems. We use the selectivity threshold (SF TH) of 0.5 and 1 for S2RDF to load data in HDFS. After that, we run selected benchmark queries from LUBM and WatDiv datasets. The data loading times of S2RDF are also reported in Table 3.8. 3CStore has the slowest data load time as compared to other storage layouts because it creates three additional tables for each VP table and also materializes them. For S2RDF, when SF value is set to 1, it materializes all ExtVP tables for query execution, when SF is set to 0.5, it materializes only those ExtVP tables whose entries do not exceed half of the corresponding VP entries, therefore, S2RDF with SF = 0.5 has faster data load time as compared to SF = 1.

Table 3.9: Store Sizes

Dataset	RDF File Size on HDFS (GB)					
	Original	VP	VPExp	3CStore	S2RDF SF = 1	S2RDF SF = 0.5
LUBM 1000	23.0	1.8	2.0	34.8	8.6	4.3
LUBM 2000	46.2	3.6	4.0	70.3	11.4	8.3
WatDiv 1000	14.5	1	1	58.5	12.1	5.4
WatDiv 5000	74.0	7.5	7.5	242.3	64.5	30.9

VP and VPExp do not have any additional computation like 3CStore and S2RDF; therefore, VP and VPExp have the fastest data load time compared to other storage systems. Table 3.9 demonstrates the HDFS sizes for the four datasets in the above-mentioned storage systems. 3CStore requires more storage space than other data storage layouts because we have not done any optimization to reduce the storage space like S2RDF system did. In addition to the load time and storage space, we are also concern about the query response time of the above-mentioned RDF systems. To conduct performance evaluation of the RDF storage management solutions using two different cluster computing engines we selected 4 representative queries from each benchmark query set. We selected Q1, Q2, Q8, and Q14 from LUBM and L1, S1, F2, and C3 from WatDiv query set. Q1 has a star-shaped pattern with high selectivity and carries large input; Q2 has a complex pattern with large intermediate results, Q8 is the most complex snowflake query of the LUBM benchmark, and Q14 is the most unselective query that returns all undergraduate students. The WatDiv representative queries fall into one of the four categories: L1 is a linear query, S1 is a star query, F2 is a snowflake, and C3 a complex one. The RDF-loader creates parquet files with table names for VP, VPExp, and 3CStore data layouts. These parquet files are used to create DataFrames, which are then

used to register Spark SQL temporary views. In Zeppelin, it takes a few seconds to a few minutes for the Spark to create temporary views for storage layouts depending on the number of tables. The compiler also generates the name of required tables for a particular SPARQL query while generating SQL statement. It is recommended to create temporary views from compiler generated table names if the cluster does not have enough memory and the number of tables is huge (especially for 3CStore data layout). Table 3.10 shows the query response time for the selected queries of the VP, VPExp, 3CStore, and ExtVP of S2RDF system.

Table 3.10: Query Runtimes (seconds) S = Spark, D = Drill

Query		VP		VPExp		3CStore		S2RDF SF = 1	S2RDF SF = 0.5
		S	D	S	D	S	D		
LUBM 1000	Q1	4	3	2	2	2	2	2	3
	Q2	11	9	8	9	4	5	3588	3629
	Q8	8	7	7	5	4	3	6	6
	Q14	1	1	1	1	1	1	1	1
LUBM 2000	Q1	6	4	3	3	2	2	2	5
	Q2	17	13	15	13	5	6	8244	8623
	Q8	14	12	14	11	7	6	7	12
	Q14	1	1	1	1	1	1	1	1
WatDiv 1000	L1	3	1	3	1	2	1	1	1
	S1	4	2	4	2	3	2	2	2
	F2	4	1	4	1	2	1	2	2
	C3	3	2	3	2	2	1	4	4
WatDiv 5000	L1	3	2	3	2	2	2	2	2
	S1	7	6	7	6	3	2	2	2
	F2	5	4	5	4	2	2	2	2
	C3	9	8	9	8	5	2	10	11

From Table 3.10 we can see that 3CStore data layout outperforms all other systems although it costs a huge amount of storage space and requires a significant amount of loading time. The reason is that a smaller number of *join* operations are involved with 3CStore data layout. S2RDF has better query response time than VP and VPExp systems except with queries involving complex shape (C3) and has large intermediate results (Q2). On the other hand, VPExp outperforms VP where *rdf:type* is in predicate position with the bound object (e.g. Q1). Table 3.10 also shows a comparative performance evaluation between Spark and Drill for VP, VPExp, 3CStore schemas along with S2RDF system. In most of the cases Drill outperforms Spark. One important thing is that Drill does not need to create temporary views like Spark, all we need to append the HDFS path (using file system storage plugin) and file format along the table names. We ran all queries for VP, VPExp, and 3CStore systems using Zeppelin.

Our proposed 3CStore outperforms all other data layouts and S2RDF system but costs a significant amount of storage space with huge data loading time. The VPExp layout only outperforms traditional VP where the object of predicates '*rdf:type'* is bound to a value in a triple pattern. The compiler can reorder triple patterns in each BGP to avoid cross product (*Cartesian join*) while generating SQL statement.

3.2.5 Conclusion

We introduce VPExp and 3CStore data layouts and present a comparative performance evaluation against traditional VP and S2RDF systems over distributed cluster computing engines Spark and Drill using different query shapes and datasets. The 3CStore accelerates the query processing but costs a significant increase in storage consumption. In the following sections, we describe several Spark-based RDF

management systems that are developed within the course of this thesis over the last years.

3.3 Mixed RDF Partitioning Strategies

The Property Table (PT) is an RDF data storage schema like a traditional relational table where each row contains a distinct subject, and its object values are stored in the corresponding columns. Each table is identified by a predicate. This approach was first introduced in Jena (Wilkinson, 2006) wherein two types of property tables were proposed. They are property class table and clustered property table. The clustered table contains sets of properties that tend to be defined together. On the other hand, the property-class table uses the "rdf:type" property of subjects to cluster similar sets of subjects together in the same table. In both implementations, a leftover table with three columns (s, p, o) is created for storing the triples not belonging to any other table. The PT approach has some drawbacks. Firstly, it generates many NULL values for a given cluster since not all properties will be defined for all subjects because of the fact that RDF data may not be very structured. Secondly, PTs cannot handle multi-valued attributes. Another variant of PT is the Unified Property Table which was proposed in Sempala (Schätzle et al., 2014) where all RDF properties of a dataset are used to form a single Property Table. In the Unified Property Table, a duplicate row will be created for each value of a multi-valued property that leads to a huge table and poses a significant overhead. Despite these limitations, the Property Table approach has an advantage that it can reduce subject-subject *self-joins* required by a query. To address the above-mentioned issues, we devise a modified version of a Property Table that avoids storing NULL values explicitly thereby preventing additional storage overhead. Multi-valued

properties are stored in a single cell using nested data structures to prevent creation of duplicate rows.

3.3.1 Modified Property Table

We use an approach to create the Property Table where all predicates (or properties) along with the subject define the name of columns in a single table. We use RDF in N-Triples format for the data storage layout. Initially, we create a TT (Triple Table) with three columns where each row comprises an RDF statement, i.e., triples (subject, property, object). Then we create PT (Property Table) with the following schema:

$$PT(subject, property_1, \dots, property_n)$$

where n is the total number of distinct properties present in a particular dataset. Here, each RDF subject is stored in the subject column and their object values reside in their corresponding property columns. We overcome the drawback that is the presence of multi-valued properties by storing the multi-valued properties in a single cell using a nested data structure (e.g., Array). Neither the clustering algorithm nor class of the subject is required to create the schema for the modified property table. Triple table is maintained along with the modified property table to answer SPARQL queries with unbound property triple patterns (e.g. {s ?p o}). Queries involving multi-valued properties require un-nesting those properties that introduces a small overhead.

Table 3.11: Modified Property Table for RDF Graph in Figure 2.2

subject	type	title	author	name	website_of
Article_1	Article	“Title One”	[David_Gary]		
Article_2	Article	“Title Two”	[David_Gary, John_Wayne]		
David_Gary	Person			“David Gary”	
www.aaa.com/d_g					David_Gary
Article_3	Article	“Title Three”	[John_Wayne]		
John_Wayne	Person			“John Wayne”	
www.bbb.com/j_w					John_Wayne

3.3.2 Subset Property Table

The Modified Property Table approach is optimized for star pattern queries. This approach allows star-pattern queries to be answered entirely without requiring a *join*, which is the most expensive operation in SPARQL. However, this data layout is not suitable for answering SPARQL queries with triple patterns having different subjects. The number of times the modified property table needs to be joined is equal to the number of different subjects present in the BGP minus one. For example, if a BGP has three different subjects, then there will be two *join* operations of the whole modified property table which degrades querying efficiency. To address this issue, we further split the whole modified property table into a number of tables with a subset of properties to reduce the input table sizes for the SPARQL query. Further splitting into subsets of properties is done as follows:

(i) use each value of *rdf:type* predicate to find distinct set of properties associated with it;

(ii) use properties that do not belong to any value of *rdf:type* predicate to form another subset. We use this subset to find distinct supersets of properties that are in turn used to create individual subset property tables.

Hence, we call this partitioning schema Subset Property Table (SPT) approach. In this data storage schema, a few numbers of subjects can reside in multiple subset property tables and causes a small amount of storage overhead that is negligible.

From the Property Table presented in Table 3.11, we can see that there are two unique values (*Person* and *Article*) of the property *rdf:type* (type). For each value of *rdf:type* property we find two subsets of properties: sp_1 = {*type, title, author*} and sp_2 = {*type, name*}. We find another subset of properties sp_3 = {*website_of*} where the property *website_of* does not belong to any value of *rdf:type* predicate. Then we take property names from each *sp* along with subject to create schema for the subset property tables and save them as parquet file. We also keep a statistics file to keep the table names along with their list of properties and row count that are used in SPARQL query translation process. Some subset property tables may have data duplication, but it improves query performance by decreasing the number of rows in input tables. These tables are smaller than the original modified property table and therefore, in many cases, it could become possible to keep the entire table in memory during query execution and the reduction of rows consequently decreases the complexity of *joins*. Example of SPTs are shown in Table 3.12.

Table 3.12: Subset Property Tables for RDF Graph in Figure 2.2

spt_1

subject	type	title	author
Article_1	Article	"Title One"	[David_Gary]
Article_2	Article	"Title Two"	[David_Gary, John_Wayne]
Article_3	Article	"Title Three"	[John_Wayne]

spt_2

subject	type	name
John_Wayne	Person	"John Wayne"
David_Gary	Person	"David Gary"

spt_3

subject	website_of
www.aaa.com/d_g	David_Gary
www.bbb.com/j_w	John_Wayne

Table 3.13: Statistics of Subset Property Tables

set_of_properties	table	size
(type, title, author)	spt_1	3 tuples
(type, name)	spt_2	2 tuples
(website_of)	spt_3	2 tuples

3.3.3 Combined Property Table & Vertical Partitioning (PT + VP)

In this RDF storage model we use our proposed Modified Property Table approach with Vertical Partitioning (VP) approach proposed in SW-Store (Abadi et al., 2009). VP has some advantages over PT approach. VP approach does not need a nested data structure to store multi-valued attribute. It creates unique rows in the table for every value of that particular property. The approach skips storing those properties for a particular subject that are not defined. Therefore, it avoids the explicit storage of NULL

data, although we overcame this limitation in the modified property table by saving the table in Parquet format.

Table 3.14: Vertical Partitioning for RDF Graph in Figure 2.2

type

subject	object
Article_1	Article
Article_2	Article
Article_3	Article
David_Gary	Person
John_Wayne	Person

title

subject	object
Article_1	"Title One"
Article_2	"Title Two"
Article_3	"Title Three"

author

subject	object
Article_1	David_Gary
Article_2	David_Gary
Article_2	John_Wayne
Article_3	John_Wayne

name

subject	object
David_Gary	"David Gary"
John_Wayne	"John Wayne"

website_of

subject	object
www.aaa.com/d_g	David_Gary
www.bbb.com/j_w	John_Wayne

This VP approach is not free from drawbacks. This approach involves more *joins* than the PT approach, technically, for n number of triple patterns in a SPARQL BGP, it needs n – 1 number of *join* operations to answer the particular query. VP tables are normally very narrow and often small; therefore, it becomes possible to keep the entire table in memory during query execution. Generally, PT is wider and bigger than each VP table, therefore, we could take advantage by using VP approach for those subjects that have a single property in SPARQL query's BGP. For subjects with multiple properties, we could use PT approach to avoid more than one *join* for star-pattern queries. In order to

get benefits from both PT and VP approaches, we decided to store RDF datasets using both these approaches.

3.3.4 Combined Subset Property Table & Vertical Partitioning (SPT + VP)

As SPTs are smaller than the original Property Table, therefore, we decided to combine the VP and SPT approaches in order to store an entire RDF dataset. As we know that the same subject can reside in multiple SPTs, therefore, we used table statistics to take the smallest table in terms of the number of rows during SPARQL processing.

3.3.5 SPARQL to Spark SQL

In order to rewrite the SPARQL query to Spark SQL, we developed a query compiler, that is implemented in Flex – a lexical analyzer creator, Bison – a parser generator and C++14. We know that every BGP consists of a set of triple patterns and *join* is the most expensive operation in Spark, therefore, we have to do the ordering of *joins* by reordering the triple patterns in the BGP using the number of bound values in the triple patterns and the statistics of the input RDF dataset. We have given the highest priority to the triple patterns that contain bound values because it limits the number of resulting tuples. Then we give priority to those triple patterns for which the selected input tables' sizes in terms of the number of tuples will be small because we want to *join* the smallest tables first, and this reduces the intermediate result size thereby minimizes the amount of data to be shuffled across the network and saves I/O and CPU. It is important to mention in here that for VP approach, tables will be selected from the name of the property of the corresponding triple patterns, and for the subset property tables, tables will be selected from the group of property names of a particular subject for which the triple patterns are grouped together. In SPT there is a possibility of getting multiple tables

for the same group of properties. In that case, we select the smallest one among those tables. For the combined approaches, VP tables will be selected only for those triple patterns whose subject is unique and no other triple pattern has the same subject in the BGP. PT or SPTs will be selected in the combined approach where triple patterns can be grouped with the same subject, that means there should be at least two triple patterns having the same subject in a BGP. The input SPARQL query can be translated to an equivalent Spark SQL query by mapping its operators to the equivalent Spark SQL keywords. A FILTER expression in SPARQL can be mapped to the equivalent conditions in Spark SQL by adapting the SPARQL syntax to the syntax of SQL and then these conditions can be added to the WHERE clause of the corresponding (sub)query in Spark SQL statement. The OPTIONAL pattern is mapped to a left outer *join*, and UNION, OFFSET, LIMIT, ORDER BY and DISTINCT can be mapped using their equivalent clauses in the SQL dialect of Spark. Finally, a translated SPARQL query is executed by Spark using a single equivalent Spark SQL query.

```
BASE            <http://example.org/property/>
PREFIX  rdf:    <http://www.w3.org/1999/02/22−rdf−syntax−ns#>

SELECT * WHERE {
        ?x    rdf:type      ?a .
        ?x    title         ?b .
        ?x    author        ?y .
        ?y    rdf:type      ?c .
        ?y    name          “John Wayne” .
        ?z    website_of    ?y }
```

Q2. An Example SPARQL Query

Figure 3.18 shows the SPARQL to SQL translation process for SPT approach. At the beginning of the translation process the compiler groups the properties for each subject and then use statistics table to select SPTs.

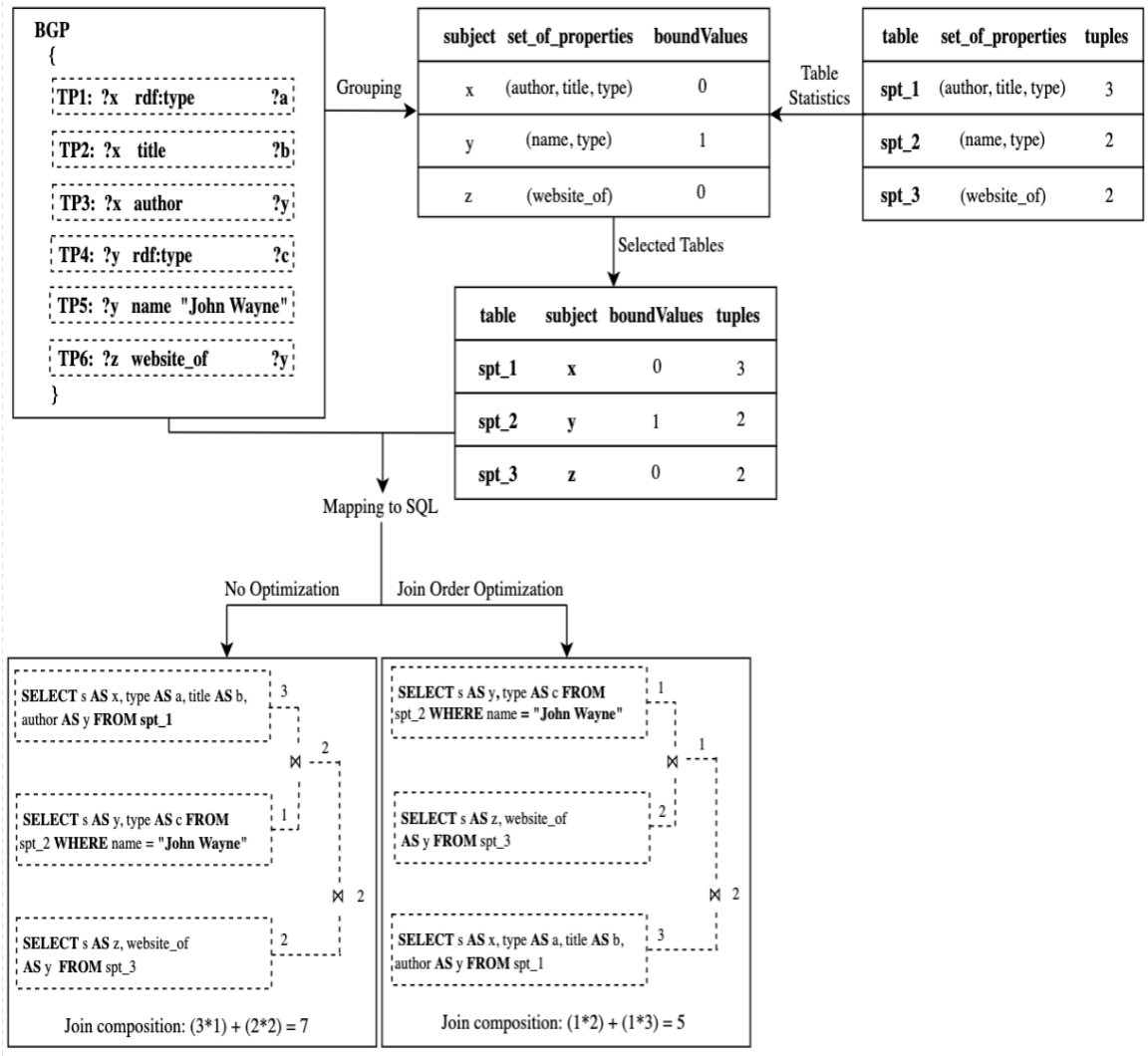

Figure 3.18: SPARQL to SQL Translation Process for SPT Approach of Q2

In this example, we showed the Join-Order Optimization technique that we used for this implementation by showing the comparison between the optimized and unoptimized version of translation for the SPARQL query. We used *join* order optimization technique where triple patterns with more bound values are given top priority because they (e.g., *spt_2*) have the best selectivity. After that, we give priority to

those selected tables who has the smallest size and then add a *join* operator for that table to generate the SQL query. For the SPT + VP approach, the table *website_of* will be selected instead of *spt_3* because the subject z has only one property, and the statistics file will also have the statistics for VP tables. PT approach has the same translation process as SPT where all selected table will have the same name *pt* with the same size and PT + VP approach follows the same translation process of SPT + VP approach.

3.3.6 Experimental Setup

We present comparative performance evaluation of our data management solutions along with other state-of-the-art systems, name S2RDF (Schätzle et al., 2015), SPARQLGX (Graux et al., 2016), and S2X (Schätzle et al., 2016). The reason for choosing those state-of-the-art systems is that all systems use the most popular general-purpose in-memory cluster computing system Spark and its components for answering SPARQL queries like we are doing for our RDF management solutions.

Benchmark Queries. We utilize two datasets for the performance evaluation of our data management solutions, LUBM with the number of universities set to 1000 and 2000, and WatDiv with scale factor 1000 and 5000. LUBM was proposed in 2005 with fourteen standard queries. This benchmark was originally designed to test the inference capabilities of Semantic Web repositories. The University of Waterloo introduced WatDiv in 2014. WatDiv has data generator as well as query generator and was designed to cover both structural and data-driven features of four different types of query shapes, namely, linear, star, snowflake, and complex SPARQL queries. We selected Q1, Q2, Q8, and Q14 from LUBM test query set. Q1 has a star-shaped pattern with high selectivity and carries large input; Q2 has a complex pattern with large intermediate results, Q8 is

the most complex snowflake query of the LUBM benchmark, and Q14 is the most unselective query that returns all undergraduate students. The WatDiv basic query set contains queries of varying shape and selectivity in order to model different scenarios. The queries are grouped into the following subsets:

- L (L1, L2, L3, L4, L5): Linear shaped queries.
- S (S1, S2, S3, S4, S5, S6, S7): Star shaped queries.
- F (F1, F2, F3, F4, F5): Snowflake shaped queries.
- C (C1, C2, C3): Complex shaped queries.

Cluster Configuration. To conduct the comparative analysis of our data management solutions, we constructed five node clusters on Microsoft Windows Azure Platform. Each node in the cluster has 4 vCPUs Intel(R) Xeon(R) CPU E5-2673 v3 @ 2.40 GHz processor, 28 GB of memory, 16000 Max IOPS, 8 Data disks and total 4 TB of hard disk space running Ubuntu 16.04.3 LTS OS. Hadoop 2.7.4 and Spark 2.2.0 are configured on all nodes where each spark executor is given 16 GB of memory.

3.3.7 Evaluation

We present an empirical comparison of our approaches with the open source HDFS and Spark based RDF management systems S2RDF, SPARQLGX, and S2X. Table 3.15 shows the size of datasets that we used in our experiment. Table 3.16 presents the storage sizes of each RDF data layout on HDFS. We use RDF dataset in N-Triples format to our proposed storage schemes to store in HDFS as parquet file like S2RDF system. SPARQLGX also uses N-Triples format for input RDF graph and stores data using text files. S2X uses Notation3 format for input RDF graph. The size of the RDF graph in Notation3 format is reported as required storage space for S2X. From the table,

we can see that PT and SPT approaches have low storage overhead, and S2RDF and S2X have high storage overhead.

Table 3.15: Experimental Setup - Dataset Scale

Dataset		Number of Triples (million)
LUBM	**Number of Universities**	
	1000	138
	2000	276
WatDiv	**Scale Factor**	
	1000	109
	5000	549

Table 3.16: Store Sizes in GB

Dataset Data Layout	LUBM 1000	LUBM 2000	WatDiv 1000	WatDiv 5000
Original	23	46.2	14.5	74
PT	0.69	1.4	0.98	5.1
VP	1.8	3.6	1	7.5
PT + VP	2.2	4.4	2	10.8
SPT	1	2	1.4	7.2
SPT + VP	2.1	4.2	2.4	12.9
S2RDF	4.88	9.6	11.3	58.9
SPARQLGX	1.1	2.3	0.899	4.8
S2X	13.6	27.3	6.1	33.4

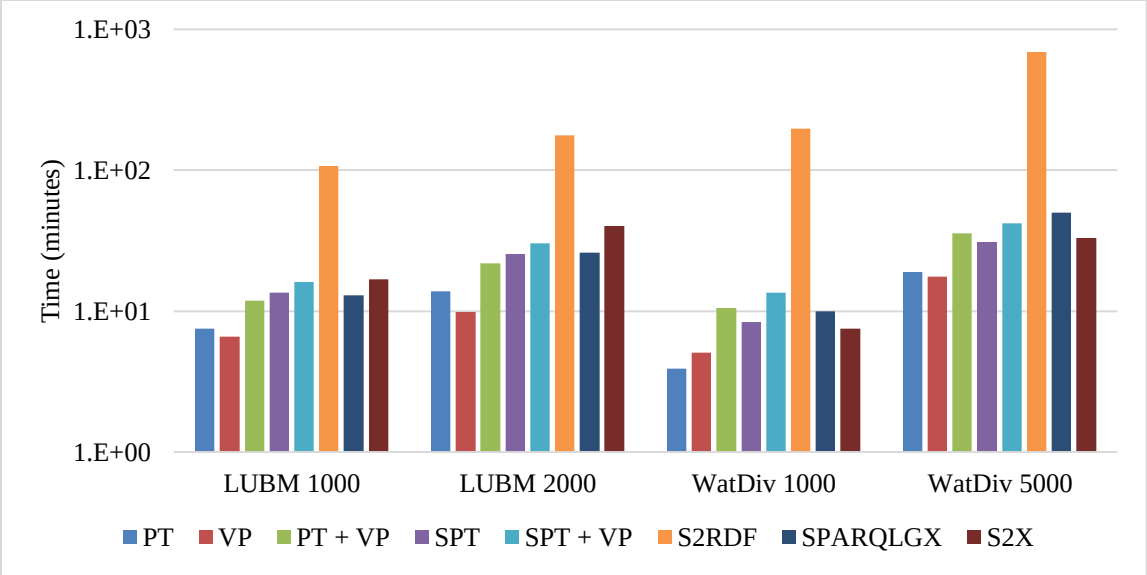

Figure 3.19: RDF Data Load Times (log scale)

3.3.8 Comparison of Storage Strategies

From the above Figure 3.19, we can see that PT and VP approaches have fast data load time compared to other storage layouts. On the other hand, S2RDF has the highest data preprocessing cost. Our proposed SPT and combined storage strategy (SPT + VP) have the moderate data loading overhead as compared to other storage systems.

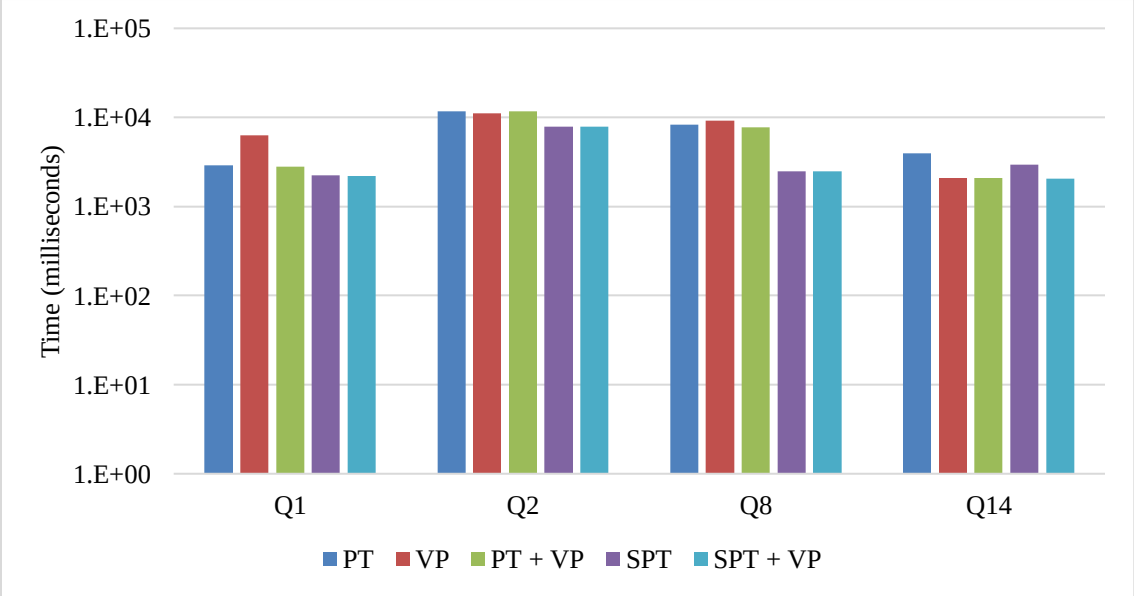

Figure 3.20: Query Runtimes - LUBM 1000 (log scale)

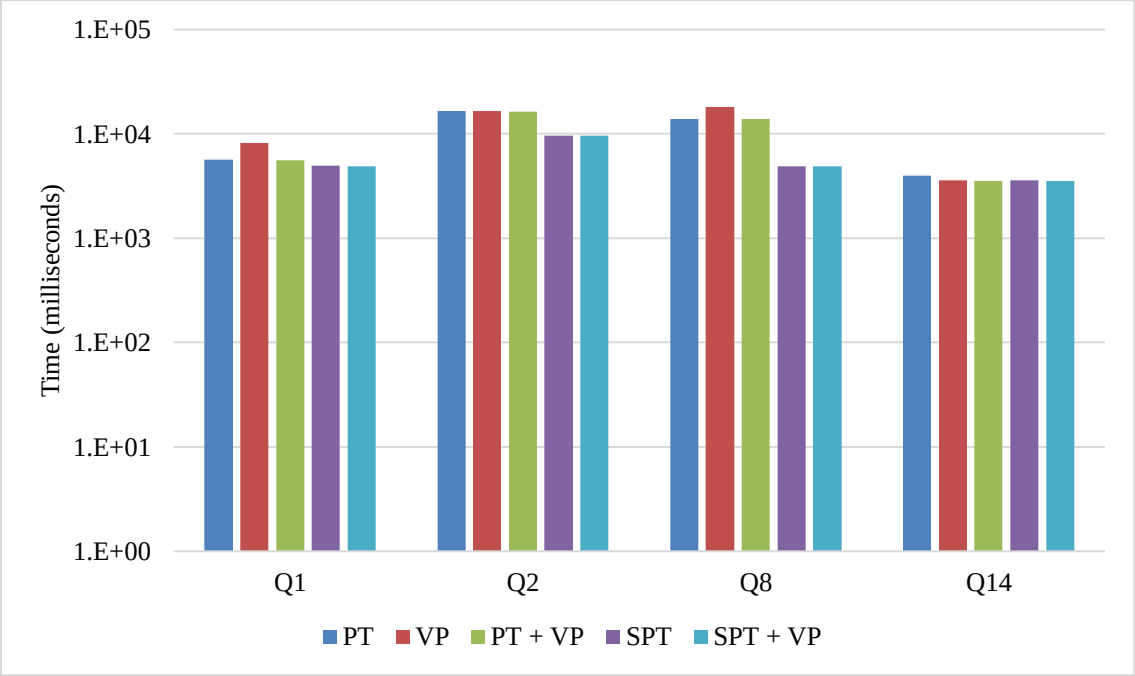

Figure 3.21: Query Runtimes - LUBM 2000 (log scale)

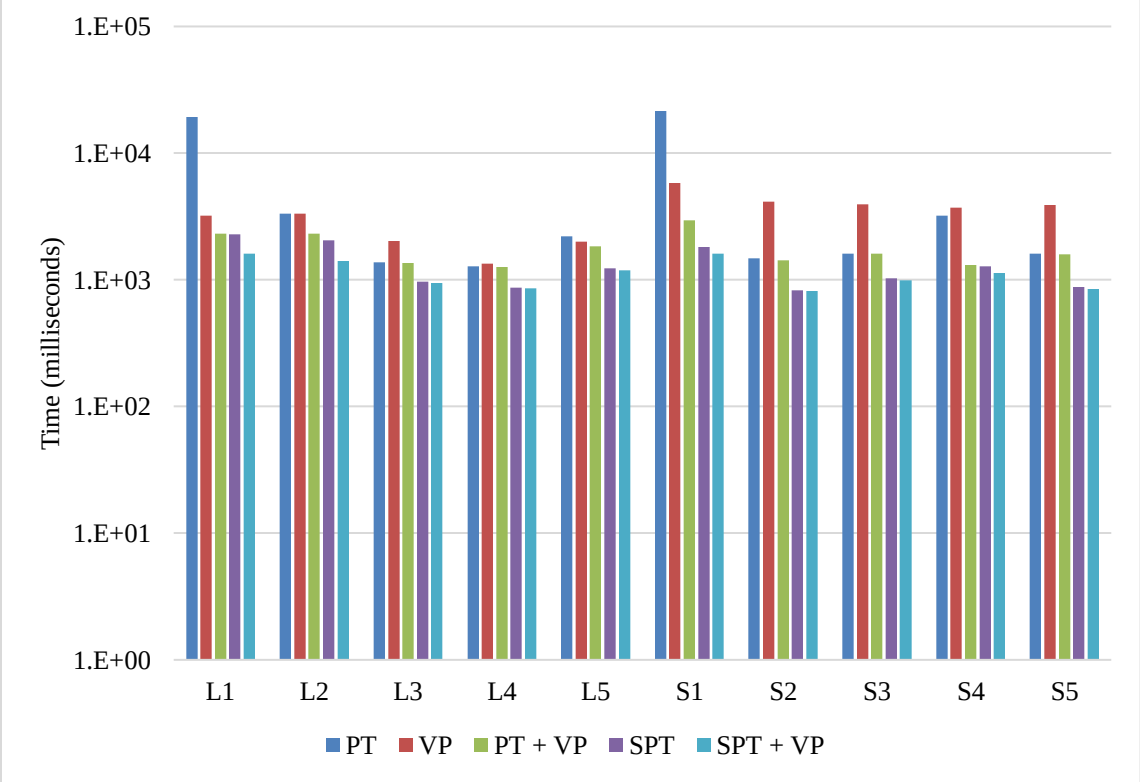

Figure 3.22a: Query Runtimes - WatDiv 1000 (log scale)

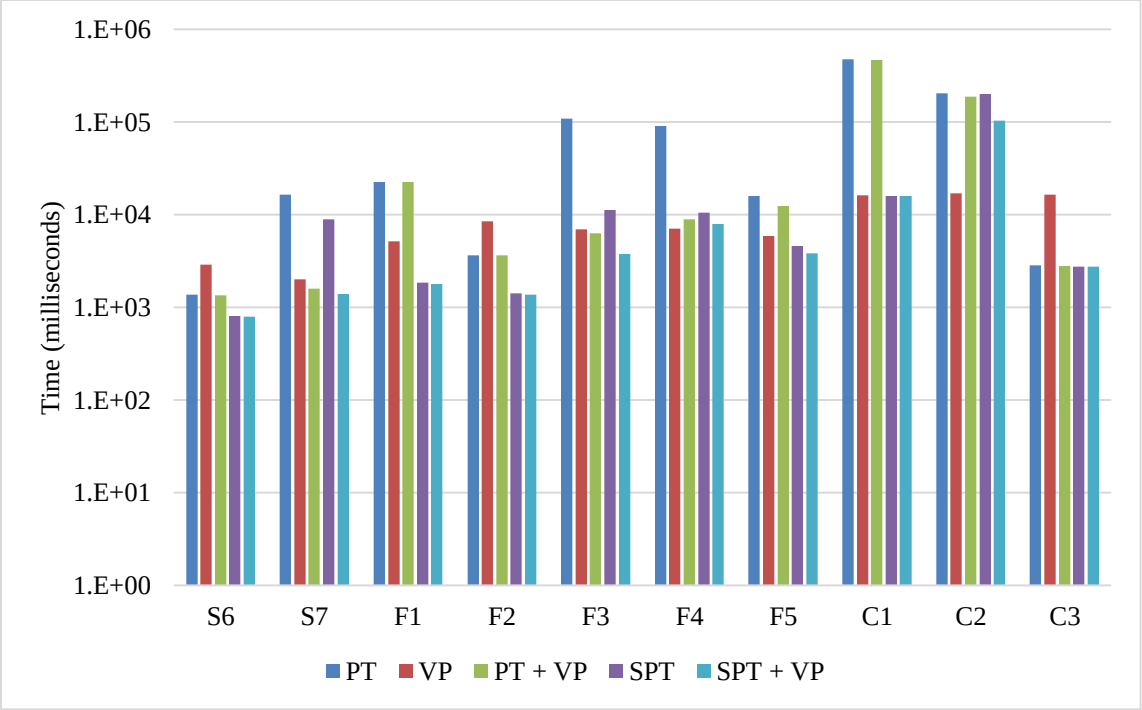

Figure 3.22b: Query Runtimes - WatDiv 1000 (log scale)

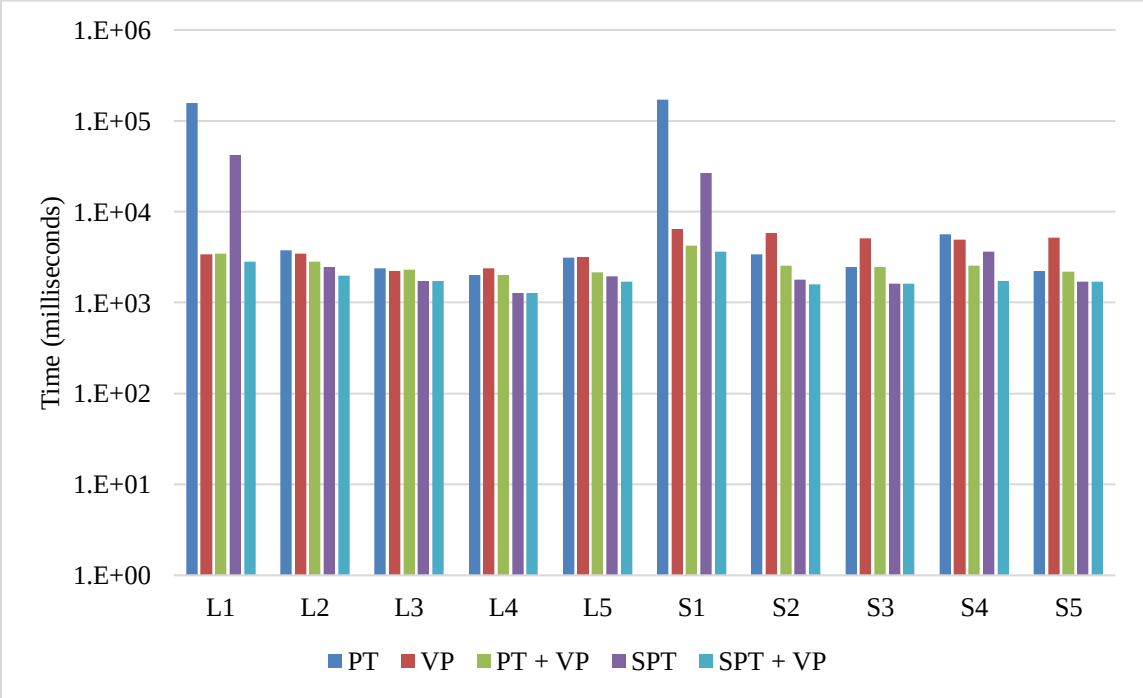

Figure 3.23a: Query Runtimes - WatDiv 5000 (log scale)

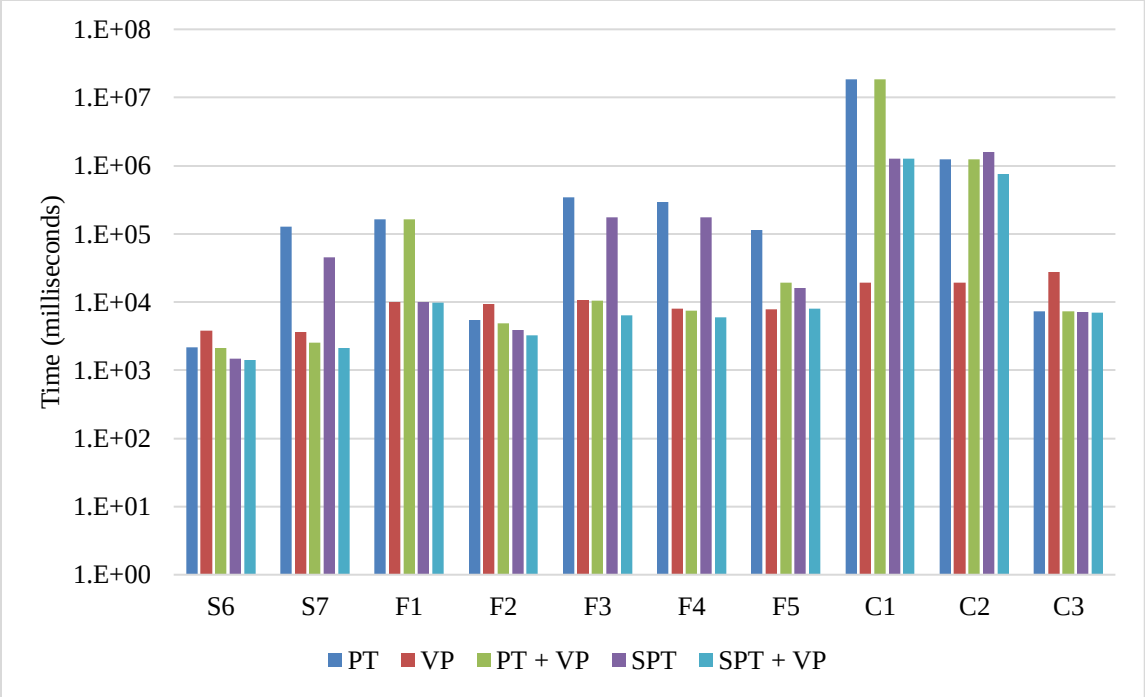

Figure 3.23b: Query Runtimes - WatDiv 5000 (log scale)

In Figure 3.20 to 3.23, we show the times in logarithmic scale to compare the query sets from LUBM and WatDiv datasets of Property Table, Vertical Partitioning, Subset Property Table, and their combined strategies. We can see that SPT + VP outperforms all other data layouts for all types of queries except query C1 and C2 from WatDiv where VP approach dominates because these queries contain mostly triples with distinct subject variables and each having more than two properties, therefore, the number of tuples in selected subset property tables is high and also quite larger than VP tables.

3.3.9 Comparison of SPT + VP Approach with Related Systems

Since the SPT + VP approach dominates in most of the cases, we decide to compare SPT + VP approach with other state-of-the-art solutions: S2RDF, SPARQLGX, and S2X in the same cluster setup.

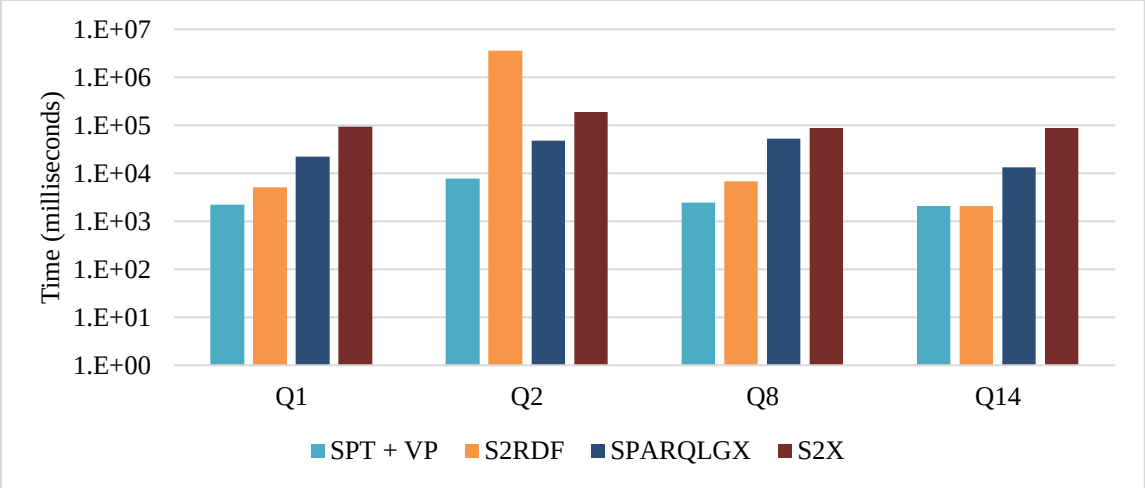

Figure 3.24: Query Runtimes - LUBM 1000 (log scale)

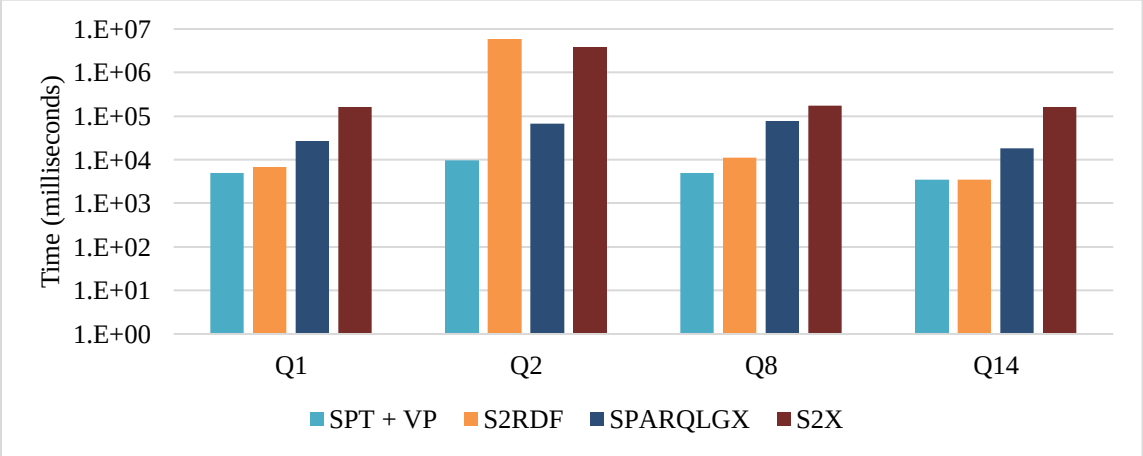

Figure 3.25: Query Runtimes - LUBM 2000 (log scale)

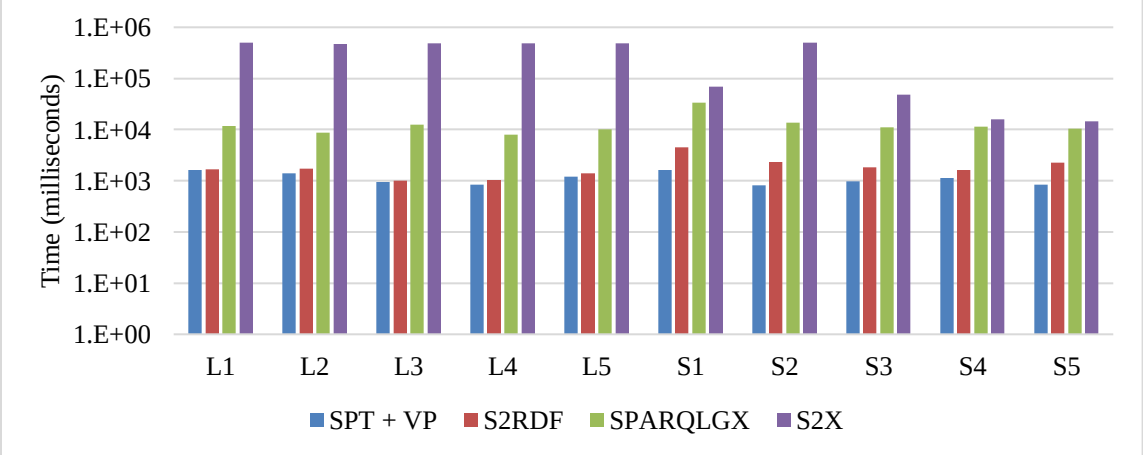

Figure 3.26a: Query Runtimes - WatDiv 1000 (log scale)

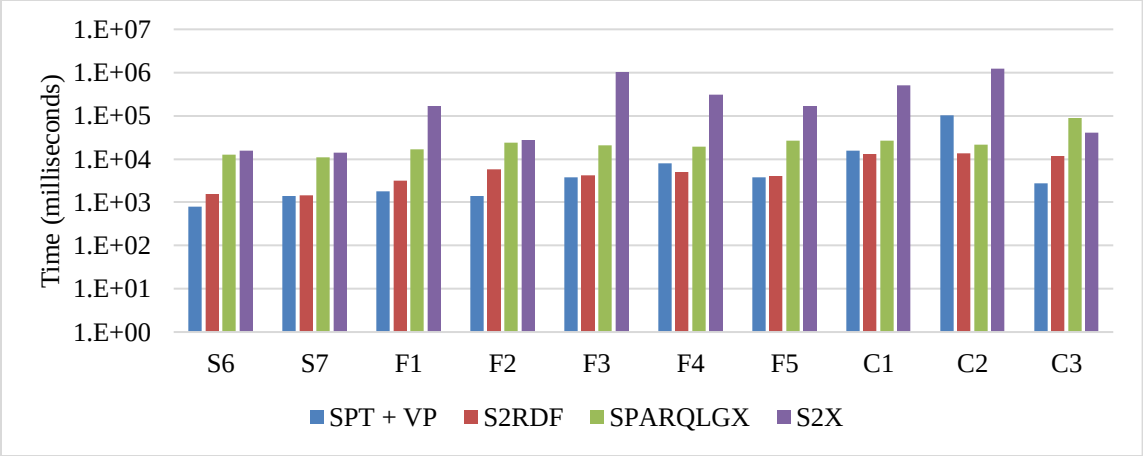

Figure 3.26b: Query Runtimes - WatDiv 1000 (log scale)

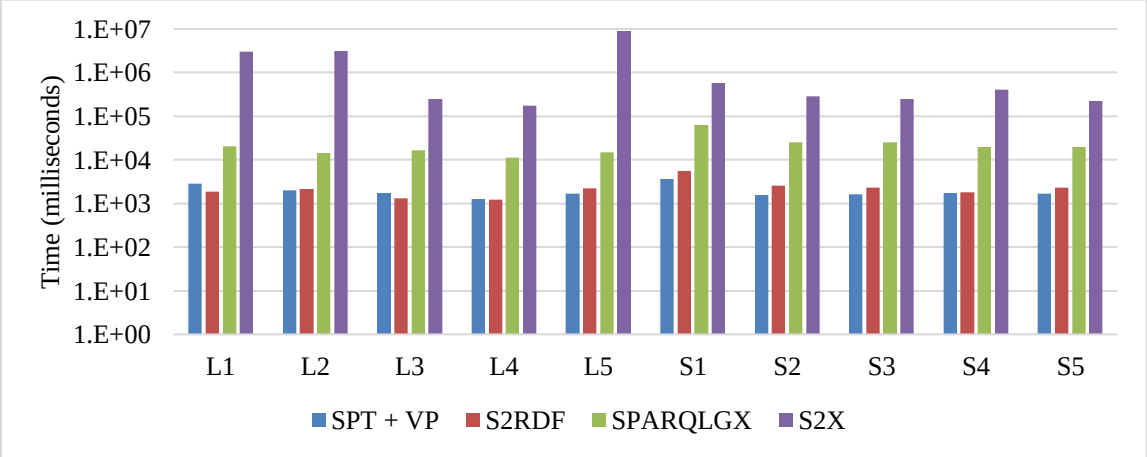

Figure 3.27a: Query Runtimes - WatDiv 5000 (log scale)

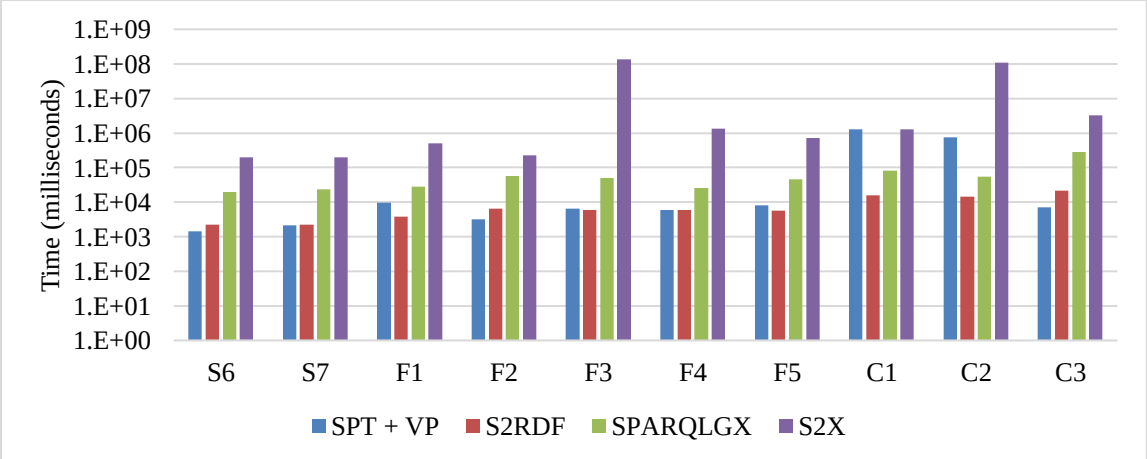

Figure 3.27b: Query Runtimes - WatDiv 5000 (log scale)

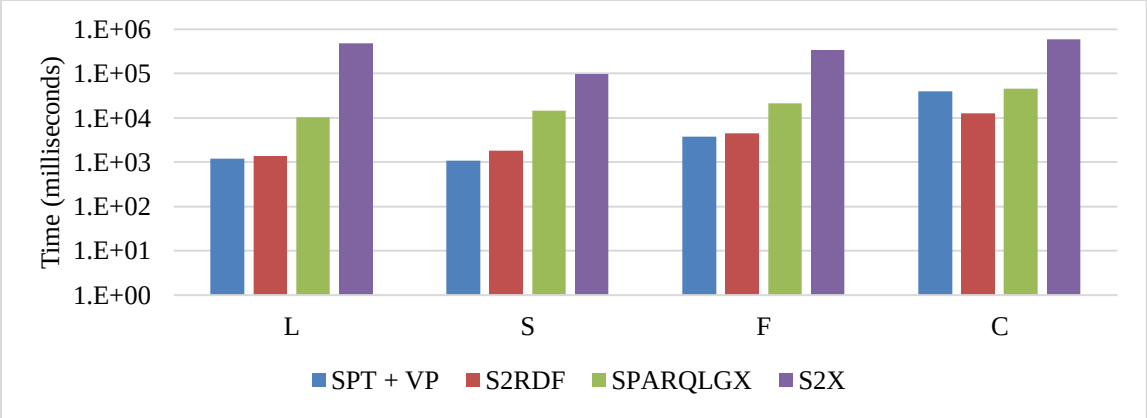

Figure 3.28a: Average Querying Time Grouped by Query Type - WatDiv 1000 (log scale)

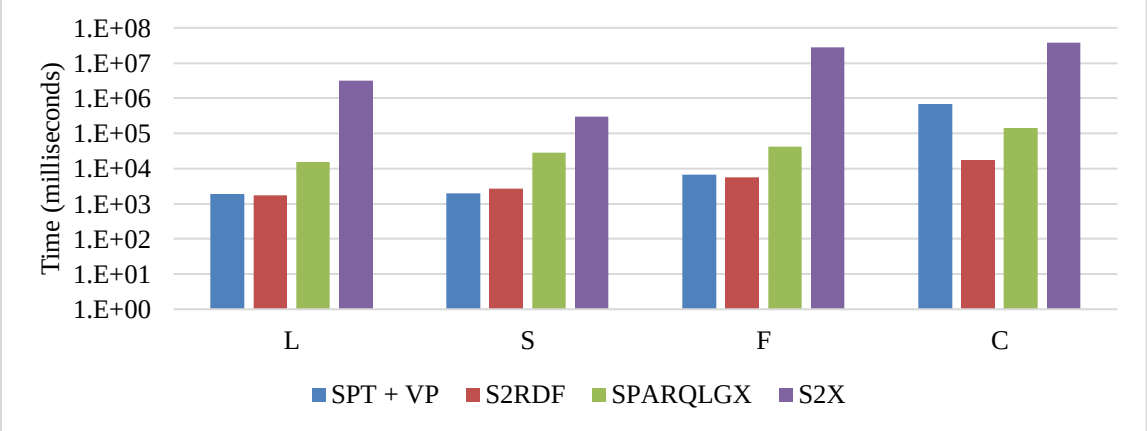

Figure 3.28b: Average Querying Time Grouped by Query Type - WatDiv 5000 (log scale)

From Figure 3.24 and 3.25, we can see that SPT + VP outperforms all other systems for the selected LUBM queries. The query performance of S2RDF is very poor with query having large intermediate result (Q2). From Figure 3.26 and 3.27, we can see that SPT + VP has a significant querying performance for star-shaped pattern (S1 to S7). Compared to S2RDF, for WatDiv 1000, SPT + VP approach is faster for queries L1 to L5, F1, F2, F3, F5, and C3 and when the data size increases to 5000, SPT + VP becomes slower for queries L1, L3, L4, F1, F3, F5 and it is still slower for C1 and C2 by a considerable margin. This is because of the extensive precomputations of S2RDF which

heavily decrease the processing time for *joins* between VP tables. Note that, S2RDF achieves this querying performance because of extensive precomputations with high loading time, therefore, this system is not suitable for some datasets having a large number of properties. On the other hand, SPT + VP has fast data loading time and does not depend on number of predicates of a particular input graph. SPARQLGX outperforms SPT + VP in queries C1 and C2 but in rest of the queries SPT + VP outperforms SPARQLGX. S2X constantly performing poor for all types of query patterns. Therefore, S2X has the worst average query response time among all other systems, as shown in Figure 3.28a and 3.28b. SPT + VP outperforms S2X in every case, mostly by a significant order of magnitude.

3.3.10 Conclusion

We presented data partitioning strategies for distributed RDF data storage and SPARQL querying built on top of Spark. The evaluation results show that our proposed SPT + VP approach outperforms all other storage approaches. We also conducted a comparative performance evaluation of SPT + VP approach on a Hadoop cluster with other state-of-the-art systems: S2RDF, SPARQLGX, and S2X using different query patterns and datasets. Our proposed Spark-based SPT + VP RDF management solution outperforms for all query types except few complex queries where S2RDF outperforms because of its materialized semi-join reduction ExtVP tables that come with an expensive preprocessing phase.

3.4 S3QLRDF with Property Table Partitioning Strategy

In this section, we describe S3QLRDF[28] (Hassan & Bansal, 2020) (SPARQL to

[28] https://github.com/sbansallab/S3QLRDF

Spark SQL for RDF), a distributed Hadoop-based SPARQL query processor for large-scale RDF data implemented on top of Spark. It uses the relational interface of Spark for query execution by compiling SPARQL to SQL and comes with a new partitioning schema for RDF data called PTP that is a modified and enhanced version of the well-known PT schema (Wilkinson, 2006).

3.4.1 Property Table Partitioning

The Modified Property Table (Hassan & Bansal, 2019) introduced in section 3.3.1 is a modified version of the traditional Property Table where multi-valued properties are stored in a single cell using a nested data structure (e.g. Array). We further partition the Modified Property Table into multiple tables based on distinct properties present in the RDF dataset to devise our proposed Property Table Partitioning (PTP) schema.

Table 3.17: PTP Schema for RDF Graph Shown in Figure 2.2

type

subject	type	title	author	name
Article_1	Article	"Title One"	[David_Gary]	
Article_2	Article	"Title Two"	[David_Gary, John_Wayne]	
David_Gary	Person			"David Gary"
Article_3	Article	"Title Three"	[John_Wayne]	
John_Wayne	Person			"John Wayne"

title

subject	type	title	author
Article_1	Article	"Title One"	[David_Gary]
Article_2	Article	"Title Two"	[David_Gary, John_Wayne]
Article_3	Article	"Title Three"	[John_Wayne]

name

subject	type	name
David_Gary	Person	"David Gary"
John_Wayne	Person	"John Wayne"

author

subject	type	title	author
Article_1	Article	"Title One"	[David_Gary]
Article_2	Article	"Title Two"	[David_Gary, John_Wayne]
Article_3	Article	"Title Three"	[John_Wayne]

website_of

subject	website_of
www.aaa.com/d_g	David_Gary
www.bbb.com/j_w	John_Wayne

Each of the PTP tables contains only those subjects that have a value for the particular property on which that partition is based, and we use the name of that particular property as the partitioned table name. Table 3.17 shows the proposed RDF data layout that is obtained from partitioning the whole Modified Property Table (Table 3.11).

An RDF dataset can have many properties, and most subjects will only use a small subset of these properties, therefore, these tables will be sparse containing NULL values. We decide to use the general-purpose Parquet columnar storage format to materialize those PTP tables in HDFS because Parquet does not store NULL values explicitly, thus sparse columns cause little to no storage overhead. We also keep a statistics file to store the actual sizes (number of tuples) of each PTP table along with the name of multi-valued attributes, such that these statistics can be used for query generation.

The goal of PTP approach is to reduce the number of tuples to scan and the amount of I/O required for a query. Since each table of the PTP is the fragment of the Property Table, it is possible to minimize unnecessary I/O and comparisons during join execution to reduce in-memory consumption. Spark is an in-memory system, and memory is typically much more limited than HDFS disk space, thus saving this resource is important for scalability. Another advantage of the PTP approach is that star patterns can be answered entirely without the need for a *join*.

3.4.2 SPARQL to Spark SQL

In this section, we describe the SPARQL query processing of S3QLRDF based on PTP schema. To generate the equivalent Spark SQL expressions from SPARQL query, we develop a query compiler that is implemented in *Flex* – a lexical analyzer creator,

Bison – a parser generator and C++14. An input SPARQL query gets mapped by the compiler to a single Spark SQL query based on PTP schema that is then executed by Spark.

Every SPARQL query defines a graph pattern to be matched against an RDF graph. A triple pattern is the basic building block of a SPARQL query, and a Basic Graph Pattern (BGP) is simply the concatenation of a set of triple patterns using AND (.). Since a BGP represents the core of the SPARQL query, we will mainly focus on the BGP fragment. A *triple group* (tg) consists of a set of triple patterns having the same subject in a BGP. So, a BGP (bgp) can have more than one distinct triple group.

Consider the following BGP (Figure 3.29).

```
bgp = {   ?x   type     ?p .
          ?x   name     "John Wayne" .
          ?y   type     "Article" .
          ?y   author   ?x .
          ?y   title    ?t .
          ?z   website_of   ?x  }
```

Figure 3.29: An Example BGP of a SPARQL Query

At first, we group the triple patterns having the same subject. The above mentioned bgp consists of three distinct triple groups, tg_1 = *{ ?x type ?p . ?x name "John Wayne" }*, tg_2 = *{ ?y type "Article" . ?y author ?x . ?y title ?t }*, and tg_3 = *{ ?z website_of ?x }*. Then we count the bound (fixed) values for each triple group. The number of bound values for the bgp is *($tg_1 \rightarrow 1$, $tg_2 \rightarrow 1$, $tg_3 \rightarrow 0$).* Here, the basic concept is that each triple group can be answered by a subquery without a join where variables occurring in a triple group define the columns to be selected and fixed values are used as conditions in the WHERE clause. Variables are mapped by subject and

property based on their position in the triple pattern. A subject variable is mapped to *subject* column and the object(s) variable is mapped to its corresponding property (multi-valued property is labeled with a special extension) column. It is worth mentioning here that Spark uses the LATERAL VIEW EXPLODE function to flatten a complex column (multi-valued property). This variable mapping is used to name the output columns such that an outer query can easily refer to it. The table for a triple group is selected from the properties which belong to that triple group. We also add a test for NOT NULL to the property (multi-valued property with a special extension) in the WHERE clause if the corresponding object is a variable in the triple pattern. This is not necessary for variables on the subject position as the subject column does not contain NULL values. Because the system is aware of the size of the PTP tables and each table is named after the property, it can select the table for a triple group that has the lowest number of tuples identified from the statistics file. For example, tg_1 has two distinct properties, *type* and *name*, so two candidate tables are available. From the statistics file, the number of tuples for the two distinct tables are *type* → *5 and name* → *2* (refer to Table 3.17). Since table *name* has fewer number of tuples compared to *type*, the table *name* will be selected for tg_1. Similarly, table *title* and *website_of* will be selected for tg_2 and tg_3 respectively. Note that, *title* and *author* have the same number of tuples; therefore, a random table will be selected between them for the tg_2. It then arranges the triple groups based on the number of bound values and the size of the selected PTP tables for the triple groups. The triple group with the highest number of bound values is given the top rank to execute first during the query execution. A triple group having the smallest number of tuples will be given the higher rank among the triple groups if they have the same number of bound

values. For example, tg_1 and tg_2 both have the highest number of bound values among the triple groups, but the selected table of tg_1 has a smaller number of tuples compared to tg_2, so tg_1 will be given the highest rank during the query execution to execute first. Now, out of the remaining two triple groups, tg_2 and tg_3, tg_3 has a lower number of tuples compared to tg_2, but the number of bound values of tg_2 is higher than tg_3. Since we are giving higher priority to the number of bound values than number of tuples of the selected table, tg_2 will be given a higher rank than tg_3. Finally, the triple groups are arranged in such a way that there must be at least one common variable between a triple group and any of its higher ranked triple group(s) to avoid cross joins when processing them in that order. So, the final ordering (ranking) among the three triple groups will be $tg_1 \rightarrow tg_2 \rightarrow tg_3$.

Overall SPARQL translation process can be described as follows:

The subquery sq_1 for tg_1 is

SELECT subject, type **FROM** name **WHERE** type **IS NOT NULL AND**

name = 'John Wayne'

The *author* is a multi-valued property that is identified from the statistics file. Thus, the *author* column is flattened by the LATERAL VIEW EXPLODE function, and we rename that column with an extension *_lev*.

The second subquery sq_2 for tg_2 is

SELECT subject, title, author_lve **FROM** title **LATERAL VIEW EXPLODE(**author**)**

EXPLODED_NAMES AS author_lve **WHERE** type = "Article" **AND**

title **IS NOT NULL AND** author_lve **IS NOT NULL**

And the third subquery sq_3 for tg_3 is

SELECT subject, website_of **FROM** website_of **WHERE** website_of **IS NOT NULL**

After applying the final ordering of triple groups ($tg_2 \rightarrow tg_1 \rightarrow tg_3$) and variable mapping for each triple group, we get the final SQL query for the bgp, that is

SELECT table_1.subject AS x, t1.type AS p, t2.subject AS y, t2.title AS t, t3.subject AS z

FROM (sq1) table_1 **JOIN** (sq2) table_2 **ON** (table_1.subject = table2.author_lve)

JOIN (sq3) table_3 **ON** (table_1.subject = table3.website_of **AND**

table2.author_lve = table3.website_of)

Therefore, the input SPARQL query can be translated to an equivalent Spark SQL query by mapping its operators to the equivalent Spark SQL keywords. A FILTER expression in SPARQL can be mapped to the equivalent conditions in Spark SQL by adapting the SPARQL syntax to the syntax of SQL, and then these conditions can be added to the WHERE clause of the corresponding (sub)query in Spark SQL statement. The OPTIONAL pattern can be mapped to a LEFT OUTER JOIN, and UNION, LIMIT, ORDER BY, and DISTINCT can be mapped directly using their equivalent clauses in the SQL dialect of Spark. Finally, a translated SPARQL query is executed by Spark using a single equivalent Spark SQL query.

3.4.3 Experimental Setup

In this section, we present a comparative performance evaluation of our RDF management system S3QLRDF along with other state-of-the-art Hadoop-based RDF querying approaches, namely CliqueSquare, S2RDF, SPARQLGX, and Rya as they are the most similar to our system. The experimental setup and a discussion of results are presented.

Benchmark Queries. For the performance evaluation of our RDF management solutions, we utilize two synthetic and one real dataset, as shown in Table 3.18. The

synthetic datasets are LUBM with the number of universities set to 1000, 5000, and 10000, and WatDiv with scale factor of 1000, 5000, and 10000.

Table 3.18: Experimental Setup - Dataset Scale

Dataset		Number of Triples (million)
LUBM	Number of Universities	
	1000	138
	5000	691
	10000	1381
WatDiv	Scale Factor	
	1000	109
	5000	549
	10000	1098
YAGO2		72

LUBM was proposed in 2005 with a data generator and was originally designed to test the inference capabilities of Semantic Web repositories. LUBM provides 14 predefined test queries, but many of these queries have simple structures and are quite similar to each other. Therefore, we selected Q1, Q2, Q4, Q8, Q12, and Q14 from the LUBM test query set based on their structure and selectivity. Q1 has a star-shaped pattern with high selectivity, and it carries large input; Q2 has a complex pattern with large intermediate results; Q4 is a simple highly selective star query with a small size of result set; Q8 is the most complex snowflake query of the LUBM benchmark; Q12 is a simple selective query, which has a constant number of solutions similar to Q1, Q4, and Q8 regardless of the dataset size; and Q14 is the most unselective query, which has a large size of results set. Q2 and Q14 have increasing numbers of solutions proportional to the dataset size. The University of Waterloo introduced WatDiv in 2014. WatDiv has a data

generator as well as a query generator, and it was designed to cover both structural and data-driven features of four different types of query shapes, namely, linear, star, snowflake, and complex SPARQL queries. The WatDiv basic query set contains queries of varying shape and selectivity to model different scenarios. The queries are grouped into the following subsets:

- L (L1, L2, L3, L4, L5): Linear shaped queries.
- S (S1, S2, S3, S4, S5, S6, S7): Star shaped queries.
- F (F1, F2, F3, F4, F5): Snowflake shaped queries.
- C (C1, C2, C3): Complex shaped queries.

The real-life dataset is the YAGO2, which is a semantic knowledge base, derived from Wikipedia, WordNet, and GeoNames. YAGO2 does not provide benchmark queries; we have created a set of representative test queries (Y1 – Y5) with different structures and complexities relative to LUBM and WatDiv query sets. Regarding LUBM queries, we modified some of the original queries because executing those original queries without the inferred triples returns an empty result set. All YAGO2 and modified LUBM queries are listed in appendix A and B respectively.

Cluster Configuration. To conduct the comparative analysis of distributed RDF data management solutions, we constructed seven node clusters (1 master and 6 workers) on the Google Cloud Platform. Each node in the cluster has a 32 vCPUs Intel(R) Xeon(R) CPU @ 2.30 GHz processor, 120 GB of memory, and 1TB of hard disk space running Ubuntu 16.04.3 LTS OS. Hadoop 2.7.7 and Spark 2.4.4 are configured on all nodes where each spark worker is given 100 GB of memory and 30 cores. In addition, Parquet filter pushdown is enabled and broadcast joins in Spark SQL are disabled.

3.4.4 Evaluation

We present an empirical comparison of our prototype S3QLRDF system with four other open-source Hadoop based state-of-the-art systems: CliqueSquare, S2RDF, SPARQLGX, and Rya. The store sizes and data loading times are listed in Table 3.19. During data loading phase, we parse data to replace all URIs with their corresponding namespace prefix and remove data type information from RDF objects to convert them into primitive types. We do not consider the data import on the HDFS as part of the preprocessing phase. We conduct a performance evaluation of S3QLRDF with other competitor systems based on three metrics: preprocessing (loading) times, store sizes, and query execution times. All measurements are averaged over four runs. S3QLRDF has two data loading options: 1. Drop all columns whose entries are all empty (NULL), and 2. Keep all columns even if all entries are empty (NULL), which we call *light-load*. The light-load requires much less time compared to the first loading option to store RDF data in PTP schema. We notice that using the first data loading option cannot reduce noticeable storage space consumption and also query execution times compared to the light-load in our cluster configuration. Therefore, we discuss results with the light-load preprocessing option for S3QLRDF. S3QLRDF has a two-step data loading process. The first step is creating the Property Table, and the second step is to create PTP tables. We do not report about the Property Table in the results of query run time because it does not participate in query evaluation. Since Spark SQL has the *cacheTable* functionality to cache table in memory, we report query execution times for both caching and without caching PTP table along with the average mean runtimes (AM). S2RDF has two preprocessing modes: VP and ExtVP, so we keep both of them in our results. We indicate

"**TimeOut**" whenever the query processing does not complete within a certain amount of time (8 hours) and "**Fail**" whenever the query is not supported by the system or the system crashes before the timeout delay.

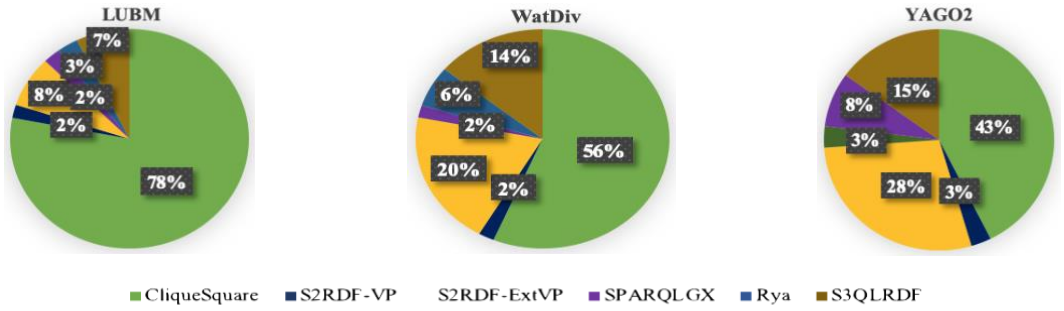

Figure 3.30: Storage Space Distributions with Datasets

Table 3.19: Loading Times and HDFS Sizes of S3QLRDF and Competitors

	Dataset	LUBM-1000	LUBM-5000	LUBM-10000	WatDiv SF-1000	WatDiv SF-5000	WatDiv SF-10000	YAGO2
HDFS Size (GB)	Original	24	116	232	15	74	149	11
	CliqueSquare	39.7	201	402	30	153	308	15
	S2RDF-VP	0.98	5	10	1	5.5	11.1	1
	S2RDF-ExtVP	3.9	19.2	38.9	10.4	53.7	108.5	10
	SPARQLGX	1.2	5.9	12.1	0.88	4.8	9.8	1.1
	Rya	1.4	7.3	14.9	2.9	17.2	32.3	2.8
	S3QLRDF	**3.7**	**18.7**	**37.4**	**7.6**	**38.3**	**76.6**	**5.3**
Loading Time (seconds)	CliqueSquare	611	3027	6149	645	2983	6237	4876
	S2RDF-VP	63	173	289	104	219	325	114
	S2RDF-ExtVP	898	2293	4112	6082	10261	14606	13899
	SPARQLGX	143	508	908	106	380	749	105
	Rya	854	3476	5735	1277	5084	12509	977
	S3QLRDF	**163**	**556**	**1009**	**279**	**766**	**1419**	**271**

Figure 3.30 indicates the storage space distribution of LUBM (avg. of 1000, 5000, and 10000), WatDiv (avg. of SF 1000, 5000, and 10000), and YAGO2 datasets. From Table 3.19, we can see that S2RDF-VP and SPARQLGX have low space overhead; on

the other hand, CliqueSquare and S2RDF-ExtVP need more storage space due to their underlying data storage layouts.

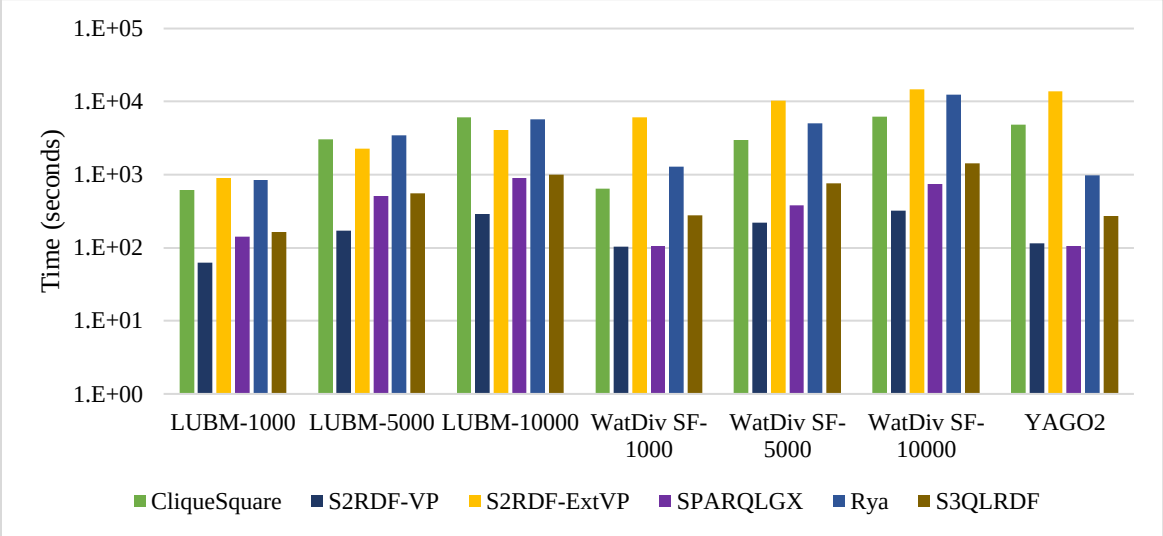

Figure 3.31: Time Distributions with Datasets (log scale)

From Figure 3.31, we notice that CliqueSquare, S2RDF-ExtVP, and Rya need more time to load data compare to S2RDF-VP and SPARQLGX because of their preprocessing methods. The lack of in-memory data processing framework in CliqueSquare and Rya causes high overhead. S2RDF-ExtVP incurs significantly higher overhead compared to S2RDF-VP because of additional pre-computation phases. Although YAGO2 is the smallest dataset, S2RDF-ExtVP needs more preprocessing time with YAGO2 due to its large number of predicates. We observe that the data loading time of S2RDF-ExtVP depends not only on the size of the dataset but also on the number of predicates. S3QLRDF has a moderate overhead in terms of data loading time and storage space as compared to other systems.

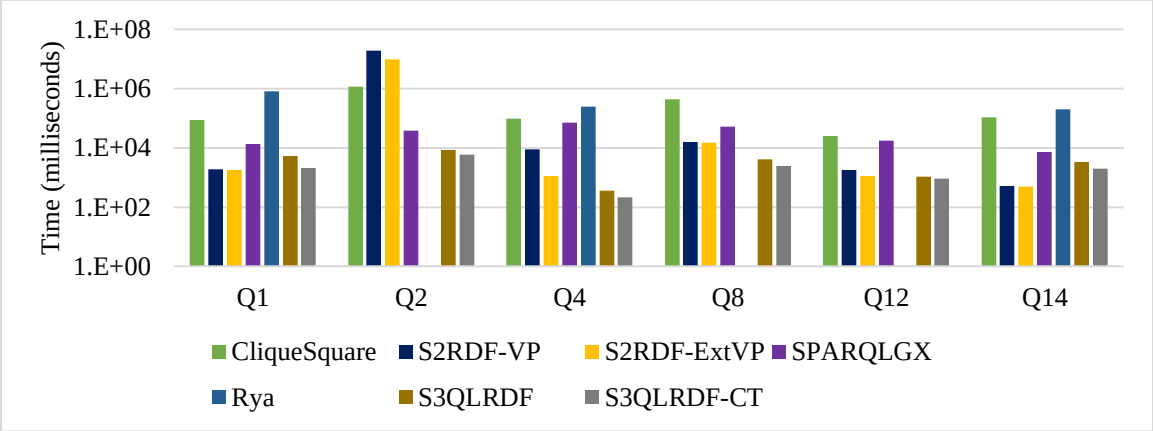

Figure 3.32: Performance Comparison for LUBM 10000 (log scale)

Table 3.20: LUBM Query Runtimes (milliseconds), AM: Arithmetic Mean

	Query	Q1	Q2	Q4	Q8	Q12	Q14	AM	Query/ hr
1000	CliqueSquare	23004	131023	24005	55008	17003	25004	45841	78
	S2RDF-VP	737	1447923	1417	3346	1291	249	242493	14
	S2RDF-ExtVP	626	436253	773	2473	816	202	73523	48
	SPARQLGX	7435	16159	15676	15320	9528	4654	11462	314
	Rya	82519	TimeOut	24306	TimeOut	TimeOut	19467	-	-
	S3QLRDF	**1289**	**4275**	**318**	**875**	**809**	**839**	**1400**	**2569**
	S3QLRDF-CT	**753**	**2708**	**162**	**579**	**529**	**468**	**866**	**4154**
5000	CliqueSquare	51008	547086	58008	221037	23004	61012	160192	22
	S2RDF-VP	1170	7535191	4220	6630	1588	424	1258203	2
	S2RDF-ExtVP	1045	2534103	811	5308	1012	364	423773	8
	SPARQLGX	10820	24649	36834	28121	11966	5328	19619	183
	Rya	393219	TimeOut	93028	TimeOut	TimeOut	103257	-	-
	S3QLRDF	**3672**	**6445**	**331**	**2045**	**984**	**1822**	**2549**	**1411**
	S3QLRDF-CT	**1387**	**4430**	**187**	**1584**	**720**	**1013**	**1553**	**2317**
10000	CliqueSquare	85014	1149205	97020	429089	25005	109019	315725	11
	S2RDF-VP	1899	18737030	8751	15377	1818	512	3127564	1
	S2RDF-ExtVP	1813	9909611	1105	15261	1126	492	1654901	2
	SPARQLGX	13780	36944	69986	51158	17697	7233	32799	109
	Rya	820376	TimeOut	250340	TimeOut	TimeOut	198825	-	-
	S3QLRDF	**5193**	**8565**	**359**	**4005**	**1069**	**3298**	**3748**	**960**
	S3QLRDF-CT	**2132**	**5841**	**209**	**2388**	**887**	**2016**	**2245**	**1603**

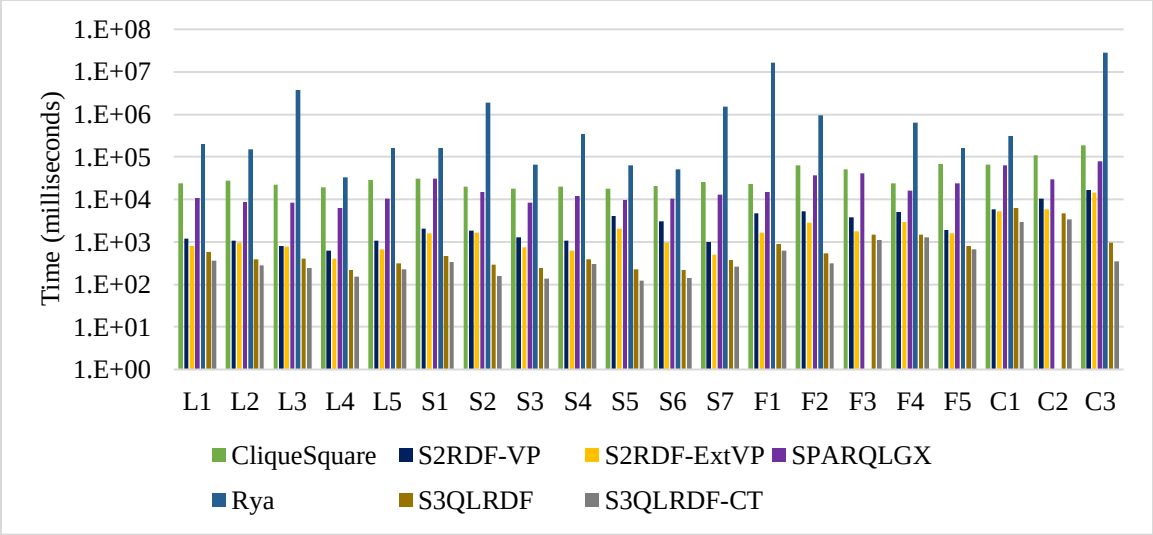

Figure 3.33: Performance Comparison for WatDiv SF10000 (log scale)

Table 3.21: WatDiv Query Runtimes (milliseconds), AM: Arithmetic Mean

	Query	L1	L2	L3	L4	L5	AM-L	Query/hr
1000	CliqueSquare	17004	17003	17003	16003	16002	16603	216
	S2RDF-VP	1057	833	728	383	655	731	4923
	S2RDF-ExtVP	693	668	483	203	345	478	7525
	SPARQLGX	7499	6056	6266	5164	6513	6299	571
	Rya	11553	13986	179566	2503	7850	43091	83
	S3QLRDF	**372**	**361**	**243**	**194**	**301**	**294**	**12236**
	S3QLRDF-CT	**271**	**241**	**154**	**107**	**209**	**196**	**18329**
5000	CliqueSquare	21004	23005	20004	18004	23005	21004	171
	S2RDF-VP	1193	864	788	476	817	827	4349
	S2RDF-ExtVP	753	740	556	364	493	581	6194
	SPARQLGX	9332	7233	7550	5295	7678	7417	485
	Rya	93100	139321	2425292	16366	73631	549542	6
	S3QLRDF	**417**	**402**	**321**	**206**	**316**	**332**	**10830**
	S3QLRDF-CT	**324**	**258**	**225**	**119**	**219**	**229**	**15720**
10000	CliqueSquare	24004	28004	22004	19004	29007	24404	147
	S2RDF-VP	1214	1082	802	612	1079	957	3758
	S2RDF-ExtVP	804	972	781	409	669	727	4951
	SPARQLGX	10803	8740	8535	6330	10579	8997	400
	Rya	201572	150843	3773827	32482	163556	864456	4
	S3QLRDF	**577**	**389**	**405**	**221**	**319**	**382**	**9419**
	S3QLRDF-CT	**364**	**279**	**243**	**153**	**226**	**253**	**14229**

	Query	S1	S2	S3	S4	S5	S6	S7	AM-S	Query/ hr
1000	CliqueSquare	18003	17003	17003	17003	17003	18003	17003	17288	208
	S2RDF-VP	1403	1351	802	993	2893	1998	974	1487	2419
	S2RDF-ExtVP	1156	1015	381	468	1220	535	403	739	4866
	SPARQLGX	17207	8156	6499	8221	5944	6999	7655	8668	415
	Rya	14013	104851	2930	30746	4713	2020	129859	41304	87
	S3QLRDF	**347**	**218**	**211**	**339**	**160**	**178**	**308**	**251**	**14310**
	S3QLRDF-CT	**242**	**115**	**120**	**210**	**117**	**112**	**227**	**163**	**22047**
5000	CliqueSquare	23006	18003	17003	18004	17003	20005	21003	19146	188
	S2RDF-VP	1947	1618	1005	1063	3276	1863	1004	1682	2139
	S2RDF-ExtVP	1224	1064	505	586	1890	689	454	916	3930
	SPARQLGX	22275	15479	7560	11251	8751	8541	8845	11814	304
	Rya	81997	976214	28658	167601	33253	33400	715782	290986	12
	S3QLRDF	**454**	**283**	**228**	**366**	**196**	**210**	**371**	**301**	**11954**
	S3QLRDF-CT	**321**	**144**	**125**	**254**	**126**	**128**	**238**	**190**	**18862**
10000	CliqueSquare	31005	20003	18003	20005	18004	21004	26005	22004	163
	S2RDF-VP	2071	1810	1276	1089	4049	3015	1012	2046	1759
	S2RDF-ExtVP	1588	1665	738	627	2054	964	498	1162	3098
	SPARQLGX	30205	15140	8251	12190	9846	10440	12707	14111	255
	Rya	160363	1914860	66350	339725	64166	51112	1544922	591642	6
	S3QLRDF	**472**	**294**	**242**	**383**	**226**	**222**	**378**	**316**	**11366**
	S3QLRDF-CT	**338**	**159**	**137**	**301**	**121**	**142**	**261**	**208**	**17272**

	Query	F1	F2	F3	F4	F5	AM-F	Query/hr
1000	CliqueSquare	17003	34005	17003	17004	23004	21603	166
	S2RDF-VP	3213	3299	2806	3100	1200	2723	1321
	S2RDF-ExtVP	1195	1762	1590	1695	1020	1452	2478
	SPARQLGX	9303	14175	12139	12256	16317	12838	280
	Rya	118584	58966	3028489	36392	13775	651241	5
	S3QLRDF	**498**	**410**	**813**	**902**	**750**	**674**	**5336**
	S3QLRDF-CT	**394**	**263**	**570**	**614**	**428**	**453**	**7933**
5000	CliqueSquare	22004	52010	29004	18004	45009	33206	108
	S2RDF-VP	4015	4174	3186	4415	1804	3518	1023
	S2RDF-ExtVP	1418	2393	1611	1996	1415	1766	2037
	SPARQLGX	12077	26228	24835	14840	20742	19744	182
	Rya	2935654	502117	TimeOut	244267	87633	-	-
	S3QLRDF	**621**	**460**	**1343**	**1152**	**765**	**868**	**4146**
	S3QLRDF-CT	**484**	**282**	**891**	**764**	**529**	**590**	**6101**
10000	CliqueSquare	23004	64009	51009	24005	69015	46208	77
	S2RDF-VP	4707	5249	3743	5052	1899	4130	871
	S2RDF-ExtVP	1666	2859	1759	2967	1586	2167	1660
	SPARQLGX	14727	36746	41766	15964	23861	26612	135
	Rya	16566663	955236	TimeOut	641823	161901	-	-
	S3QLRDF	**903**	**543**	**1488**	**1502**	**802**	**1047**	**3436**
	S3QLRDF-CT	**630**	**316**	**1093**	**1278**	**662**	**795**	**4523**

	Query	C1	C2	C3	AM-C	Query/hr
1000	CliqueSquare	33005	37006	30005	33338	107
	S2RDF-VP	3427	5250	5852	4843	743
	S2RDF-ExtVP	3251	3189	5275	3905	921
	SPARQLGX	19854	15152	21817	18941	190
	Rya	15444	2992945	2173732	1727373	2
	S3QLRDF	**3854**	**2615**	**387**	**2285**	**1575**
	S3QLRDF-CT	**1597**	**1686**	**212**	**1165**	**3090**
5000	CliqueSquare	49010	71018	92018	70682	50
	S2RDF-VP	4625	8970	10709	8101	444
	S2RDF-ExtVP	4092	4892	8705	5896	610
	SPARQLGX	34894	32621	48768	38761	92
	Rya	130440	TimeOut	13385691	-	-
	S3QLRDF	**5199**	**2844**	**664**	**2902**	**1240**
	S3QLRDF-CT	**2152**	**2332**	**302**	**1595**	**2256**
10000	CliqueSquare	65014	109017	190041	121357	29
	S2RDF-VP	5880	10361	16488	10909	329
	S2RDF-ExtVP	5292	5783	14382	8485	424
	SPARQLGX	64319	29652	78596	57522	62
	Rya	310289	TimeOut	28712939	-	-
	S3QLRDF	**6370**	**4702**	**977**	**4016**	**896**
	S3QLRDF-CT	**2968**	**3449**	**351**	**2256**	**1595**

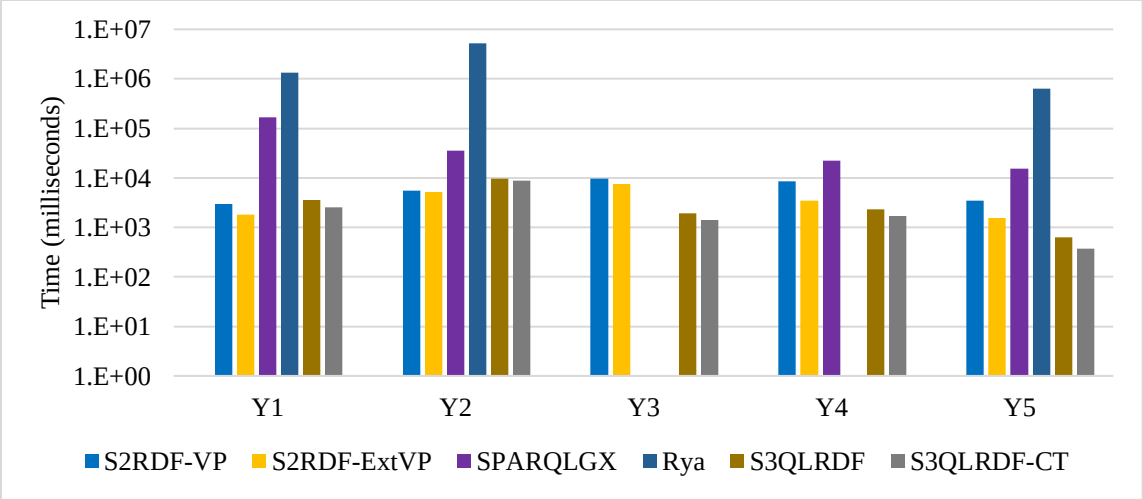

Figure 3.34: Performance Comparison for YAGO2 (log scale)

Table 3.22: YAGO2 Query Runtimes (milliseconds), AM: Arithmetic Mean

Query	Y1	Y2	Y3	Y4	Y5	AM	Query/hr
S2RDF-VP	2923	5585	9754	8620	3469	6070	593
S2RDF-ExtVP	1811	5188	7507	3445	1566	3903	922
SPARQLGX	169546	35260	Fail	22542	15141	-	-
Rya	1329020	5288669	Fail	TimeOut	632515	-	-
S3QLRDF	**3525**	**9610**	**1921**	**2284**	**632**	**3594**	**1001**
S3QLRDF-CT	**2544**	**8853**	**1407**	**1685**	**376**	**2973**	**1210**

The performance comparison for LUBM 10000 is illustrated in Figure. 3.32 on a log scale while absolute runtimes are given in Table 3.20. We can observe that S3QLRDF outperforms all other systems by up to an order of magnitude on average (arithmetic mean). Q1 and Q4 are the most selective queries, returning only a few results and can be answered by S3QLRDF within 5200 milliseconds or less. These queries define a star-shaped pattern, which can be answered very efficiently with the PTP table of S3QLRDF. For the most unselective query, Q14, S3QLRDF outperforms all other systems. Q2, Q8, and Q12 define the complex patterns where Q8 and Q12 produce

results of constant size as the size of the dataset increases. On the other hand, the intermediate result set of Q2 increases when the input dataset increases. Also, for these queries, runtimes of S3QLRDF are significantly faster than for all other systems, which is below 9000 milliseconds. If we use the *cacheTable* functionality of Spark SQL to cache PTP tables in memory, which we call S3QLRDF-CT, then we achieve an order of magnitude faster response time despite that the caching table incurs a little overhead due to caching time. We also report the number of query executions per hour (Query/hr) where S3QLRDF and S3QLRDF-CT outperform all other systems. Figure 3.33 compares the different systems on the largest dataset (SF10000) of WatDiv, corresponding AM runtimes are listed in Table 3.21. For WatDiv, S3QLRDF and S3QLRDF-CT show a competitive runtime performance for all query categories when increasing the size of the dataset. In Table 3.21, we report the number of queries to execute per hour (Query/hr) under all query categories for all competitors. Again, S3QLRDF and S3QLRDF-CT outperform all of its competitors by an order of magnitude in terms of Query/hr. Figure 3.34 illustrates the execution times for YAGO2 queries of all compared systems while absolute runtimes, and Query/hr are given in Table 3.22. CliqueSquare fails to execute YAGO2 queries; therefore, we did not include CliqueSquare in the YAGO2 query evaluation. We can observe that S3QLRDF and S3QLRDF-CT outperform SPARQLGX and Rya by an order of magnitude on runtime in all queries. S2RDF has faster query response times for Y1 and Y2 compared to S3QLRDF because of the materialized join reduction tables of ExtVP and because S3QLRDF incurs a little overhead while flattening a complex column. Since a number of complex columns are required to be flattened in Y1 and Y2, S3QLRDF is slower in response time compared to S2RDF, but in terms of

average runtime and Query/hr, S3QLRDF outperforms all of its competitors, including S2RDF.

3.4.5 Conclusion

We propose a distributed RDF storage and SPARQL querying system, S3QLRDF, based on the PTP schema built on top of Spark. S3QLRDF uses the SQL interface of Spark for query execution by compiling SPARQL to SQL. A complex join query over large tables is expensive in a distributed setting because of large amounts of data reading and shuffling across the network. The concept of this work is to reduce the amount of data that must be accessed in a distributed environment. We conduct a comparative performance evaluation of the S3QLRDF system on a Hadoop cluster with the state-of-the-art systems CliqueSquare, S2RDF, SPARQLGX, and Rya, using different query shapes, complexities with three different datasets up to 1.4 billion triples. Overall, the evaluation demonstrates that S3QLRDF can be an efficient solution for querying semantic data.

CHAPTER 4

BENCHMARKING S3QLRDF UNDER COLUMNAR FILE FORMATS

Columnar file formats have well known advantages that can improve the storage efficiency by effective data compression, as well as helping to achieve significant performance gains by moving only relevant portions of data into memory during query processing. Columnar storage formats have been available for storing data in HDFS for over a decade. Currently, Parquet and ORC formats are two of the most popular ones for HDFS.

4.1 Relational Data Management Using Parquet and ORC

Relational data management including analysis is one of the most popular data processing paradigms. Modern cloud-based relational data processing systems typically do not manage their storage. They leverage a variety of external file formats to store and access data. Over the last decade, a variety of external file formats such as Parquet, ORC, etc., have been developed to store large volumes of relational data in the cloud. High-performance networking and storage devices are used pervasively to process this massive amount of data in Big Data frameworks like Spark and Hadoop. The performance of a file format in terms of storage efficiency and data access rate plays an important role in data management.

Parquet and ORC are columnar data storage in the Hadoop ecosystem. They offer features that store data by employing different encoding, column-wise compression, compression based on data type, and predicate pushdown. Typically, enhanced compression ratios, or skipping blocks of data, involves reading fewer bytes from HDFS, resulting in enhanced query performance. We use Parquet and ORC file formats as the

storage backend for our S3QLRDF system to run the experiments in order to measure the RDF data storage efficiency, loading, and query execution performance.

4.2 Evaluation

We present an empirical comparison between Parquet and ORC file formats while using S3QLRDF system with the PTP schema.

4.2.1 Experimental Setup

We performed our evaluation on a small cluster of 6 machines (1 master and 5 workers) using AWS EC2 instances. Each machine is equipped with 64GB of memory, 1 TB of disk space and with an 8 Core Intel Xeon Platinum 8175M CPU @ 2.50 GHz. The cluster runs with Hadoop 2.7.7, Hive 2.3.6, and Spark 2.4.4 on Ubuntu 16.04 LTS. The resource manager, Yarn, uses 240 GB of memory and 40 virtual cores. In our cluster configuration, a Spark partition size is equal to the default size of an HDFS block (128 MB). We kept the default settings for both Parquet and ORC file formats with filter pushdown enabled.

The experiments are conducted on a synthetic dataset, WatDiv, with around 109M triples and 86 predicates, and a real-world dataset, a dump of YAGO (Yago2s 2.5.3), with a total size of 245 million triples and 104 predicates. The PT (Property Table) creation is the prerequisite to create the PTP tables, therefore, we report total time to create PT and PTP as data loading time. Both Parquet and ORC are efficient formats in terms of storage size due to their use of columnar storage and built-in compression. For this performance comparison, we use their default compression codec when writing Parquet/ORC files using Spark 2.4.4.

4.2.2 Analysis of Results

We report datasets loading times and HDFS sizes for PTP schema based on Parquet and ORC file formats in Table 4.1.

Table 4.1: WatDiv and YAGO Loading Times and HDFS Sizes

	File Format	Load Time	HDFS Size
WatDiv	Parquet	796 s	7.1 GB
	ORC	768 s	6.6 GB
YAGO	Parquet	5621 s	16.7 GB
	ORC	4871 s	12.1 GB

Table 4.1 shows that ORC outperforms Parquet in terms of storage space and data loading time. These two formats physically organize the data in different manners, which is why they differ from one another in terms of their total size. Table 4.2 indicates the total number of completed stages and tasks during the loading phase, where Parquet takes more stages and tasks than ORC.

Table 4.2: Stages and Tasks During Dataset Loading Phase

	File Format	Total Number of Stages	Total Number of Tasks
WatDiv	Parquet	267	4679
	ORC	266	4673
YAGO	Parquet	321	8830
	ORC	320	7789

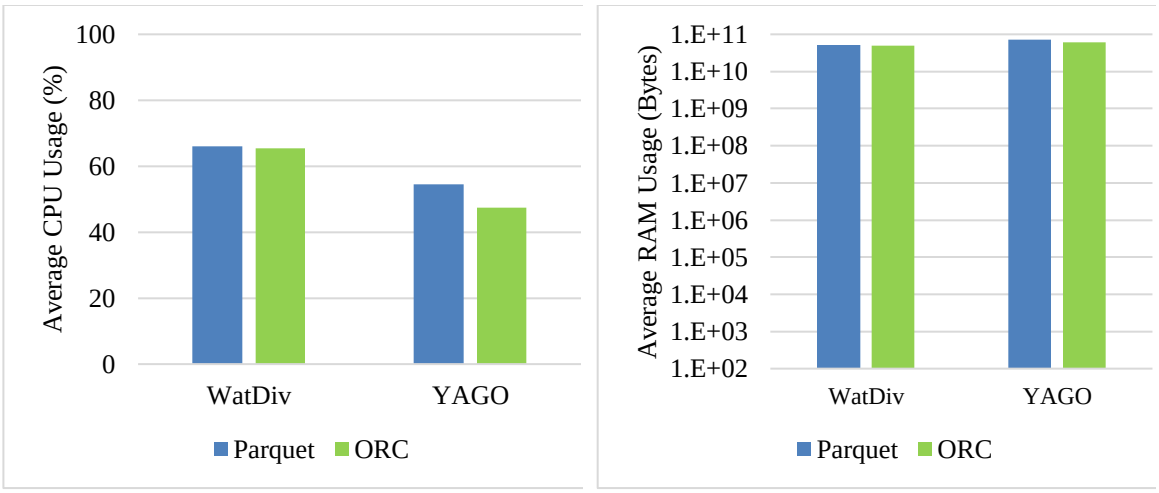

Figure 4.1: CPU and RAM Consumptions During Data Loading Phase

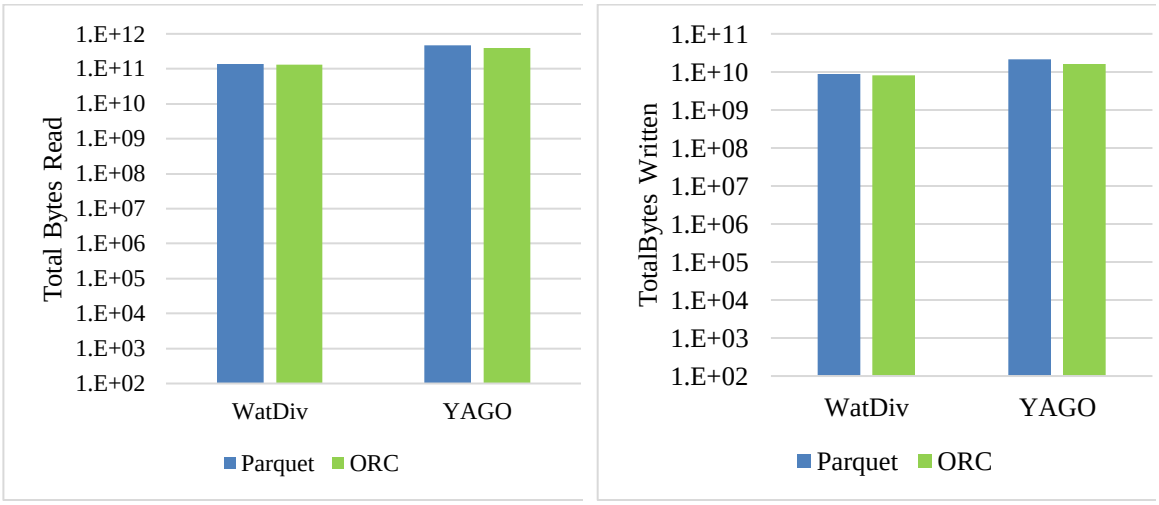

Figure 4.2: Total HDFS Bytes Read/Written During Data Loading Phase

Figures 4.1 and 4.2 present resource usages (CPU and RAM) and the total amount of bytes read from and written on the HDFS during the data loading process. The percent of CPU and the amount of RAM usage are slightly less in ORC than Parquet. Similarly, S3QLRDF reads and saves less amount of data while working with ORC than Parquet.

WatDiv comes with a set of 20 predefined query templates called Basic Testing Use Case that can be grouped in four categories according to their shape: complex (C), snowflake (F), star (S), and linear (L). Each of the queries from the basic query set is evaluated four times to get the average run time. Finally, the query run times are

aggregated by the query shapes. YAGO does not provide benchmark queries; we have created four representative test queries (C, F, S, and L) based on the categories of WatDiv basic query set where C, F, S, and L represent complex, snowflake, star, and linear-shaped query. We submitted each query at a time as a single Spark Application in the cold-start scenario when memory was free. The run times reported for each query are the average of 4 execution times. Since Spark SQL has the *cacheTable* functionality to cache tables in memory before execution, we report average query execution times for both caching (CT) and without caching (W/O-CT) PTP tables. We also report the query run times (T-CT) including caching times to investigate how the caching table affects the overall query runtimes.

Table 4.3: WatDiv Basic Testing (milliseconds)

WatDiv-C	Parquet	ORC
W/O-CT	18787	26521
CT	9932	8765
T-CT	34196	45296

WatDiv-F	Parquet	ORC
W/O-CT	30606	49703
CT	14596	13081
T-CT	52561	72198

WatDiv-S	Parquet	ORC
W/O-CT	31561	49325
CT	8079	7011
T-CT	47936	67483

WatDiv-L	Parquet	ORC
W/O-CT	22743	35096
CT	5454	4906
T-CT	35906	51103

Table 4.4: YAGO Query Run Times (milliseconds)

YAGO-C	Parquet	ORC
W/O-CT	146028	152791
CT	120097	122743
T-CT	136335	143867

YAGO-F	Parquet	ORC
W/O-CT	5228	8321
CT	1704	1593
T-CT	8819	12435

YAGO-S	Parquet	ORC
W/O-CT	59349	78218
CT	58876	68082
T-CT	64776	76602

YAGO-L	Parquet	ORC
W/O-CT	4974	8232
CT	1541	1394
T-CT	8302	13064

The performance comparison between Parquet and ORC storage formats based on PTP schema in terms of the query execution times for WatDiv and YAGO are shown in Tables 4.3 and 4.4 respectively. The first observation was that ORC with CT, compared to that of other options, had the best query performance for all WatDiv query types. For YAGO, ORC with CT shows the best performance except for the C and S query types, although it is not significantly worse. We did not consider caching times of PTP table in memory for CT, but if we report caching times along with query runtimes (T-CT) then ORC has slightly worse performance for the majority of query types. We also observe that Parquet without *cacheTable* method (W/O-CT) shows reasonably better performance for all query types. For future experiments in chapter 5, we will be using Parquet without *cacheTable* method to measure query runtimes.

From the above discussion, we can conclude that the caching table in memory adds some overhead to the total query runtimes; therefore, the *cacheTable* method is recommended only for batch execution of queries. We demonstrate query performance while using *cacheTable* method for batch execution of queries in section 3.4.4.

4.3 Conclusion

Spark is a prominent Big Data framework that offers a high-level SQL interface (Spark-SQL) optimized by means of the Catalyst query optimizer. We conducted a systematic evaluation for the performance of the Spark-SQL query engine for answering SPARQL queries over the PTP schema for RDF datasets using two columnar file formats, Parquet and ORC. The experimental results show that ORC has better storage space efficiency, but in most of the cases Parquet was able to achieve better performance in terms of query execution times.

CHAPTER 5

ASSESSMENT ON SPARK-BASED RDF MANAGEMENT SYSTEMS

Apache Spark is one of the most widely used tools in the Big Data platform for efficient query answering over a large volume of data. This cluster computing framework uses in-memory data structures that can be used to efficiently store data and enables distributed query answering. Over the last few years, several systems have been designed to exploit the Spark framework for building scalable RDF processing engines like S3QLRDF, S2RDF, SPARQLGX, and PRoST. These systems load data as triples, and a simple partitioning technique, like vertical partitioning or property table partitioning, is applied to their raw form for further processing. In such systems, the RDD API, or Spark SQL, is used to answer the SPARQL query.

5.1 Benchmarked SPARQL Evaluators

In this section, we present a brief overview on Spark-based RDF management systems, namely S3QLRDF, S2RDF, SPARQLGX, and PRoST. Table 5.1 shows the RDF data partitioning techniques used in the state-of-the-art Spark-based systems.

Table 5.1: Partitioning Strategies of Spark-based RDF Management Solutions

Storage Schema	VP	WPT	PTP	ExtVP
S3QLRDF			X	
S2RDF	X			X
SPARQLGX	X			
PRoST	X	X		

Spark-based systems listed in Table 5.1 use one or a combination of relational partitioning techniques. S3QLRDF uses PTP schema to devise the RDF data storage layout, S2RDF makes use of both VP and ExtVP approaches, SPARQLGX uses only the

VP approach, and PRoST combines the VP with the Wide Property Table (WPT) (Cossu et al., 2018) for their storage layout. Table 5.2 represents the RDF query processing methods used in Spark-based systems based on Spark data abstraction.

Table 5.2: Data Access Model of Spark-based RDF Management Solutions

Data Access Model	RDD API	DataFrame/Dataset (Spark SQL)
S3QLRDF		X
S2RDF		X
SPARQLGX	X	
PRoST		X

For the performance evaluation of Spark-based RDF management solutions, we utilize two real datasets YAGO (Yago2s 2.5.3) and DBLP as shown in Table 5.3. The YAGO is a semantic knowledge base, derived from Wikipedia, WordNet, and GeoNames. Meanwhile, the DBLP Computer Science Bibliography provides bibliographic information on computer science journals and proceedings. Both YAGO and DBLP do not provide benchmark queries. Thus, we have created four representative test queries C, F, S, and L for each dataset based on varying shape; like complex, snowflake, star, and linear to model different scenarios respectively. These query patterns actually affect the overall query performance. All YAGO and DBLP queries are listed in appendix C and D respectively.

Table 5.3: Experimental Setup - Dataset Statistics

Dataset	Number of Triples (million)	Number of Predicates	HDFS Size (GB)
YAGO	245	104	35.5
DBLP	129	27	19.3

5.2 Evaluation

Cluster Configuration. We performed our evaluation on a small cluster of 6 machines (1 master and 5 workers) using AWS EC2 instances. Each machine is equipped with 64 GB of memory, 1 TB of disk space, and an 8 Core Intel Xeon Platinum 8175M CPU @ 2.50 GHz. The cluster runs with Hadoop 2.7.7, Hive 2.3.6, and Spark 2.4.4 on Ubuntu 16.04 LTS. Yarn is the resource manager, which in total uses 240 GB memory and 40 virtual cores. In our cluster configuration, we keep the default size of the HDFS block (128 MB) with parquet filter pushdown enabled.

Table 5.4: Loading Times and HDFS Sizes

	Dataset	YAGO	DBLP
HDFS Size (GB)	**S3QLRDF**	16.7	23.8
	S2RDF	32.8	29.1
	SPARQLGX	3.4	2.2
	PRoST	15.3	8.7
Loading Time (seconds)	**S3QLRDF**	5621	486
	S2RDF	10999	2385
	SPARQLGX	751	417
	PRoST	1695	723

Experimental Results. We present an empirical comparison of 4 open-source Spark-based state-of-the-art systems: S3QLRDF, S2RDF, SPARQLGX, and PRoST based on real datasets, YAGO and DBLP. The store sizes and data loading times are listed in Table 5.4. From Table 5.4, we can see that SPARQLGX has low space overhead; on the other hand, S2RDF needs more storage space due to their underlying data layouts. SPARQLGX also has low preprocessing overhead compared to other systems. S2RDF needs more preprocessing time with YAGO due to its large number of

predicates. We observe that the data loading time of S2RDF depends not only on the size of the dataset but also on the number of predicates which involve extensive precomputations with high loading time; therefore, this system is not suitable for some datasets having a large number of properties. S3QLRDF has a moderate overhead in terms of data loading time when compared to other systems. Table 5.5 indicates the total number of completed stages and tasks during the loading phase. S2RDF is highly expensive in terms of computation among all other systems.

Table 5.5: Stages and Tasks During Dataset Loading Phase

	System	Total Number of Stages	Total Number of Tasks
YAGO	**S3QLRDF**	321	8830
	S2RDF	33568	1412013
	SPARQLGX	25	7552
	PRoST	1227	67446
DBLP	**S3QLRDF**	90	2374
	S2RDF	4304	275578
	SPARQLGX	25	4135
	PRoST	378	23098

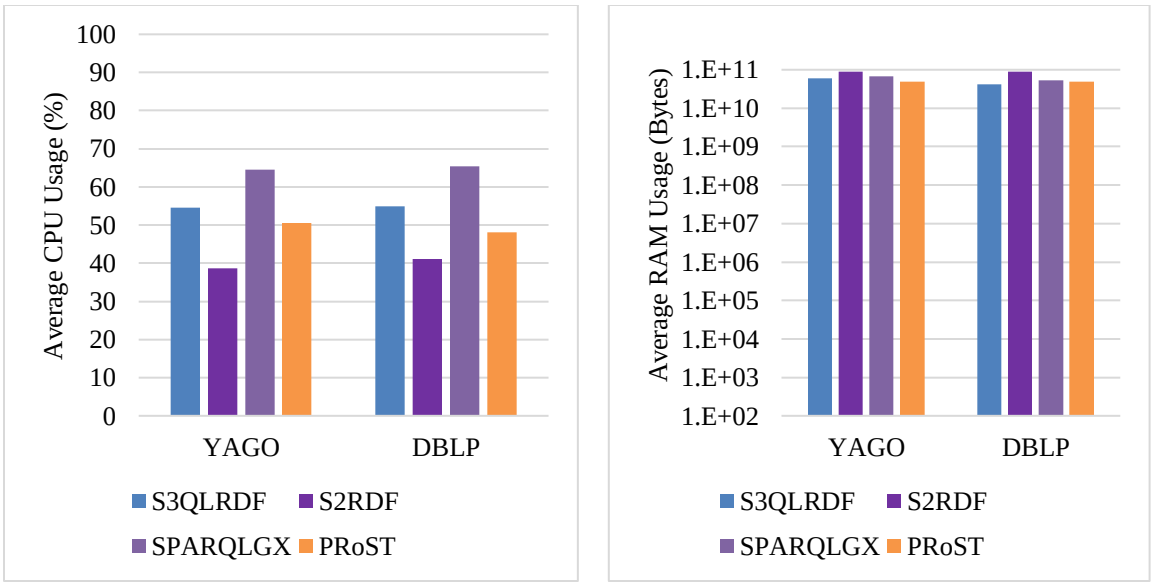

Figure 5.1: CPU and RAM Consumptions During Data Loading Phase

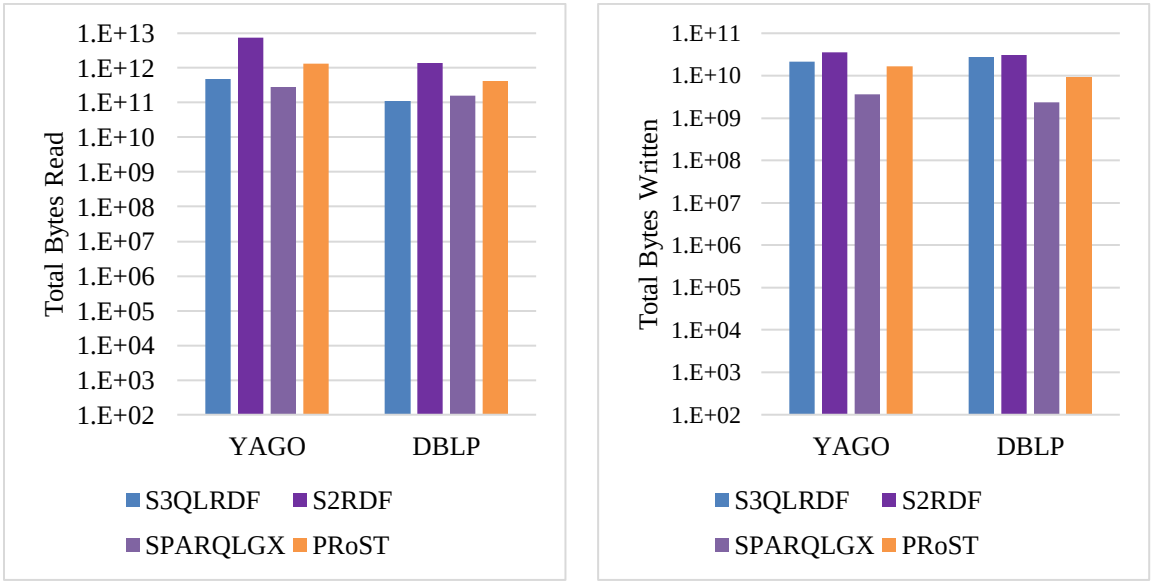

Figure 5.2: Total HDFS Bytes Read/Written During Data Loading Phase (log scale)

Figures 5.1 and 5.2 present resource usages (CPU and RAM) and the total amount of bytes read from and written on the HDFS during the data loading phase. SPARQLGX has highest CPU utilization while reading and saving less amount of data for both YAGO and DBLP datasets. On the other hand, S2RDF has the highest amount of RAM usage compared to other systems. From the above discussion, we can conclude that S2RDF is

the costliest system for the cluster because of the highest data loading times and RAM usages.

Query Performance. We conduct a query performance evaluation of Spark-based RDF management systems based on query execution times and cluster resource utilization. We report the query run times including caching times for those systems that use *cacheTable* functionality to cache table in memory. Not all systems offer to execute a set of queries in the same Spark application to take advantage of in-memory data left by a previously executed query. Thus, we submitted each query at a time as a single Spark application to make a fair comparison among all systems. All measurements are averaged over four runs.

YAGO Dataset. YAGO does not provide benchmark queries. Therefore, we use the YAGO test queries C, F, S, and L listed in appendix C to benchmark the performance of different Spark-based systems.

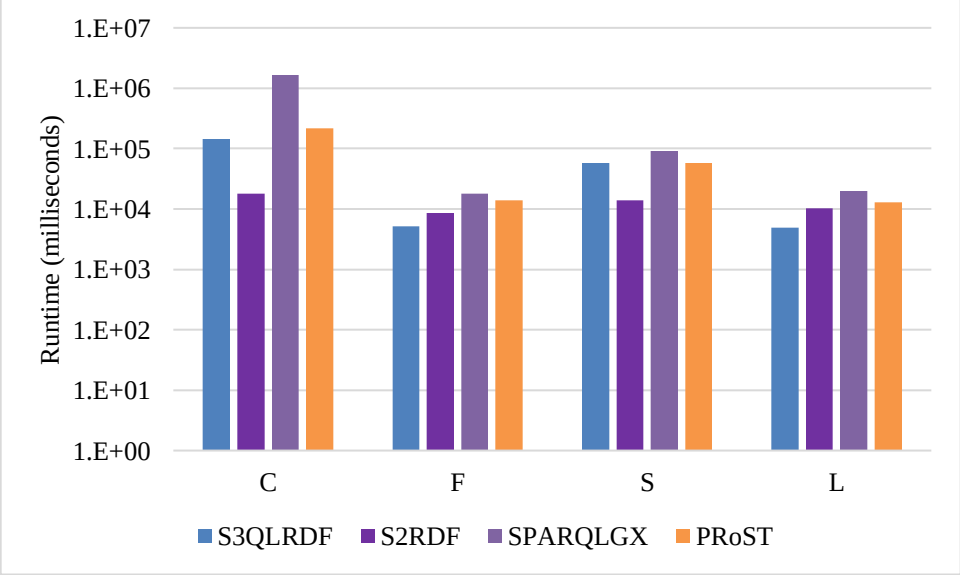

Figure 5.3: YAGO Query Run Times (log scale)

Table 5.6: Stages and Tasks During YAGO Query Phase

	Query	C	F	S	L
Total Number of Stages	**S3QLRDF**	12	5	3	5
	S2RDF	59	36	27	22
	SPARQLGX	18	9	7	7
	PRoST	12	6	2	6
Total Number of Tasks	**S3QLRDF**	502	37	19	37
	S2RDF	1858	809	800	658
	SPARQLGX	831	9	228	403
	PRoST	1151	482	35	482

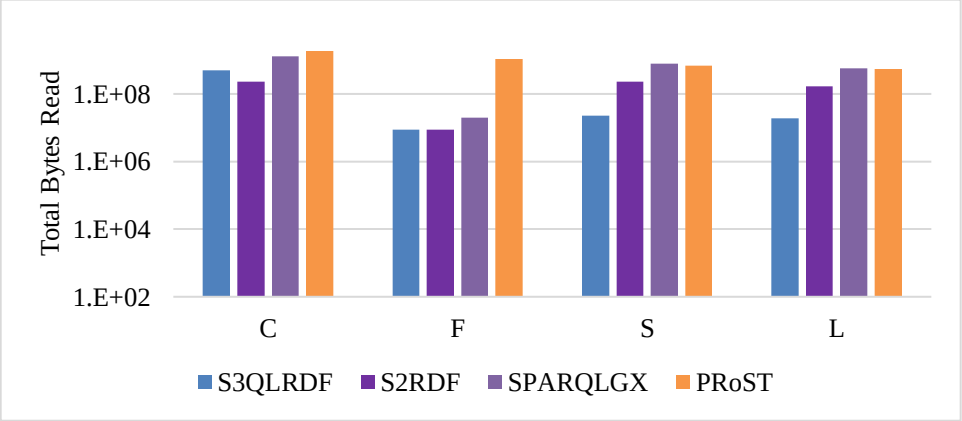

Figure 5.4: Total HDFS Bytes Read During YAGO Query Phase (log scale)

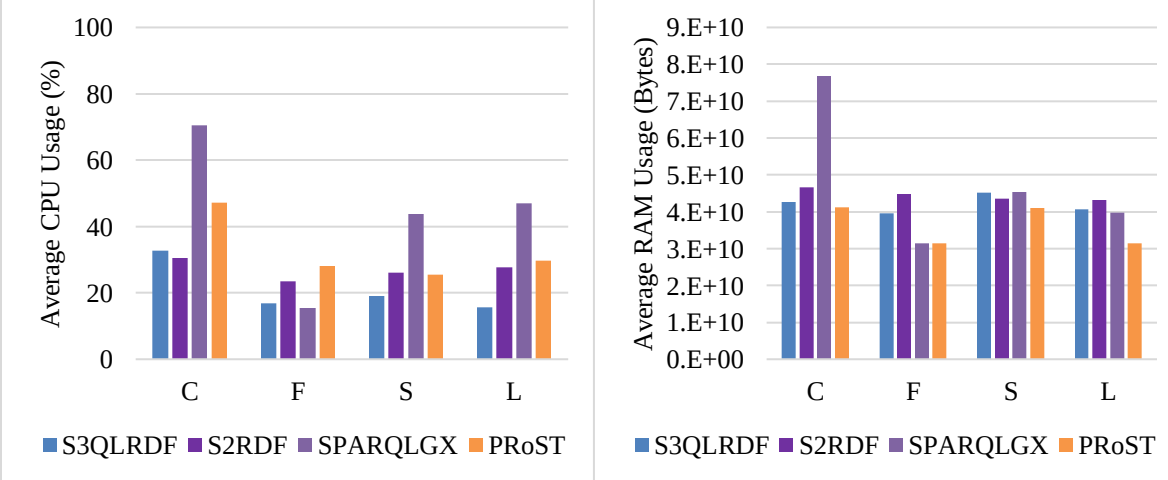

Figure 5.5: CPU and RAM Consumptions During YAGO Query Phase

Figure. 5.3. illustrates the performance comparison for YAGO. S3QLRDF shows the best performance, except for query C and S, although it is not significantly worse. S3QLRDF incurs a little overhead while flattening a complex column. Since a number of complex columns are required to be flattened in C and S, S3QLRDF is slower in response time compared to S2RDF, which has the fastest query response times for C and S compared to all other systems due to the materialized join reduction of ExtVP tables. S2RDF trades off the query performances with disk space and loading time. SPARQLGX has poor runtimes for all queries among all systems. The total number of completed stages and tasks during the YAGO query phase are listed in Table 5.6. From Figure 5.4 we can see that the number of bytes required to read during query evaluation is less in S3QLRDF for all of the queries, except C. We also figure out from Figure 5.5 that the system SPARQLGX, which is inexpensive in terms of data loading time, become costly in cluster resource utilization (CPU and RAM) for evaluating most of the queries, except query F.

DBLP Dataset. Like YAGO, DBLP does not have benchmark queries; therefore, we use the DBLP test queries C, F, S, and L listed in appendix D.

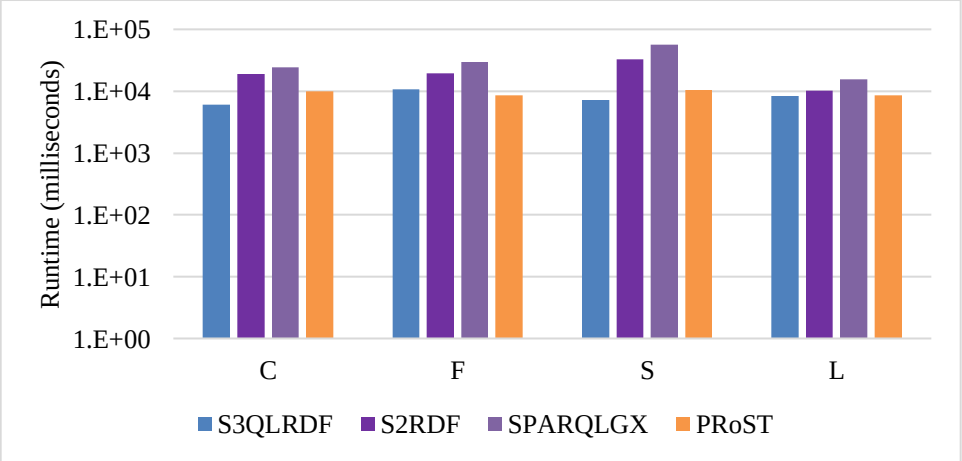

Figure 5.6: DBLP Query Run Times (log scale)

Table 5.7: Stages and Tasks During DBLP Query Phase

	Query	C	F	S	L
Total Number of Stages	**S3QLRDF**	7	5	10	8
	S2RDF	52	43	82	17
	SPARQLGX	13	10	17	5
	PRoST	6	4	2	4
Total Number of Tasks	**S3QLRDF**	64	43	285	456
	S2RDF	1385	1377	2153	563
	SPARQLGX	73	76	136	29
	PRoST	461	241	21	241

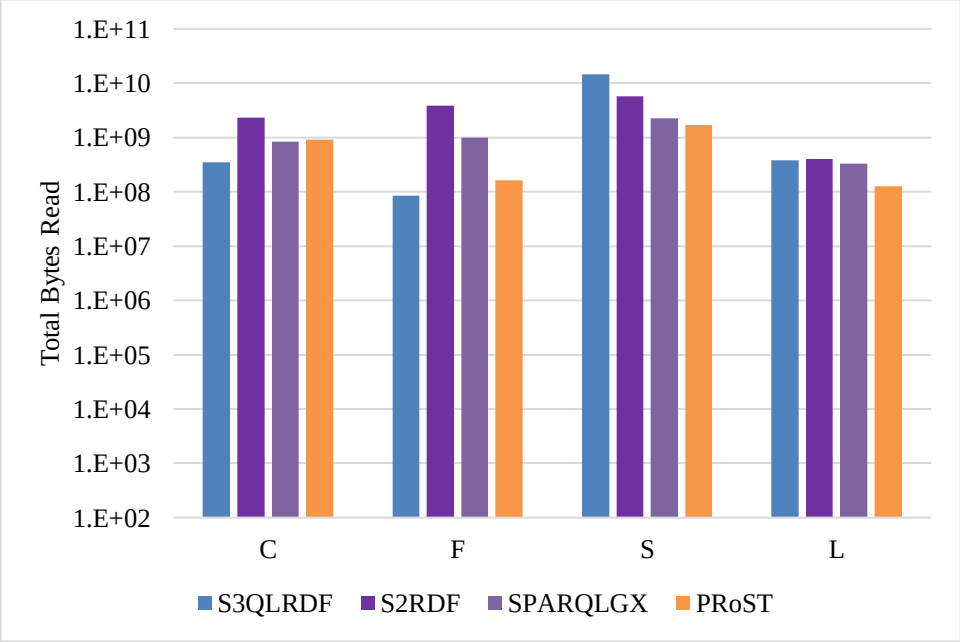

Figure 5.7: Total HDFS Bytes Read During DBLP Query Phase (log scale)

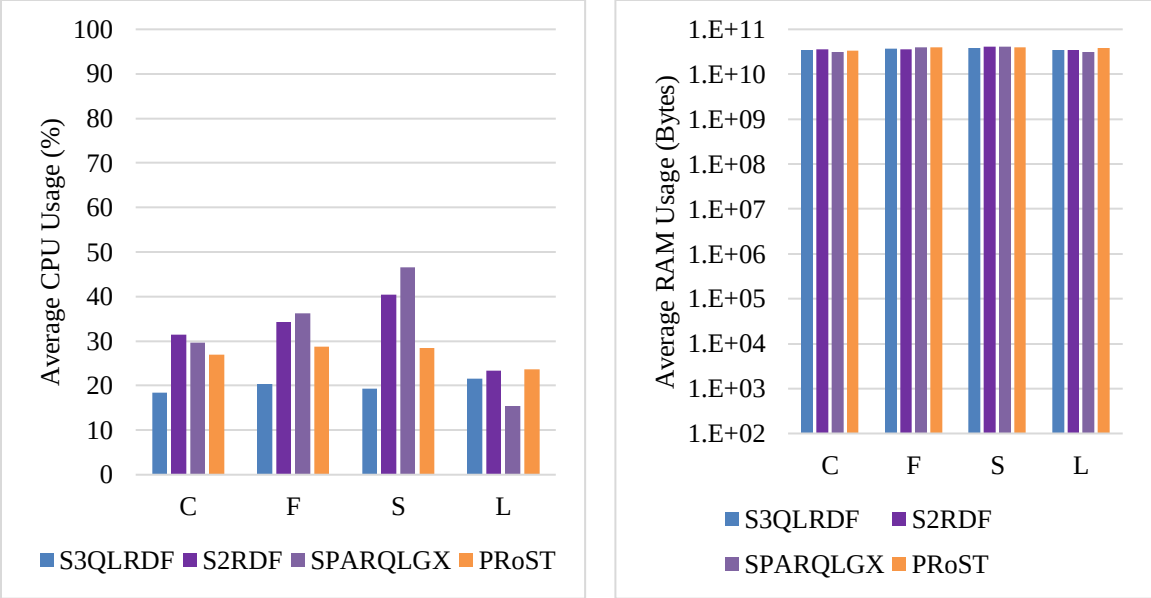

Figure 5.8: CPU and RAM Consumptions During DBLP Query Phase

Figure 5.6 illustrates the execution times for DBLP queries of all compared systems. We can observe that S3QLRDF outperforms its competitors on runtime in most of the queries, except F, where PRoST shows the best performance. Like YAGO, SPARQLGX again shows poor query performance among all systems. The total number of completed stages and tasks during the DBLP query phase are listed in Table 5.7. We can also observe from Figure 5.7 that S3QLRDF reads relatively a less number of bytes to answer queries C and F; on the other hand, PRoST requires less number of bytes to read during query S and L evaluation. The average cluster CPU usage percent is high in S2RDF and SPARQLGX while the average RAM usage is almost similar for all systems (Figure 5.8).

5.3 Conclusion

In this chapter, we conduct an empirical evaluation of 4 state-of-the-art Spark-based RDF management solutions based on common criteria: preprocessing (loading) times, store sizes, query execution times, and cluster resource utilization. All of these

systems use different data partitioning techniques to devise their relational storage schemas for RDF triplestore on top of Hadoop. The aim of using Spark with Hadoop is to provide efficient RDF management systems to improve query performance by exploiting data parallelization. Moreover, data partitioning also plays a vital role in efficient query processing which has a huge impact on query performance.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The proliferation of the Semantic Web in the form of RDF demands an efficient, scalable, and distributed storage along with a highly available and fault-tolerant parallel processing strategy. More precisely, the rapid growth of RDF data raises the need for an efficient partitioning strategy over distributed data management systems to improve SPARQL query performance regardless of its pattern shape with minimized pre-processing time. To this direction, a number of distributed big data processing tools, like Hadoop, Hive, HBase and Impala, are exploited progressively due to their ability to effectively handle large amounts of data. Spark is one of the most prominent Big Data frameworks that offers a high-level SQL interface, being Spark-SQL, which is optimized by means of the Catalyst query optimizer. This framework uses in-memory data structures that can be used to store RDF data, offering increasing efficiency, and enabling effective distributed query answering.

In this work, we focus on two key elements in the distributed system for efficient SPARQL query processing; data parallelization and data partitioning. We propose several RDF data partitioning schemas, like VPExp, 3CStore, Modified Property Table, Subset Property Table, and Property Table Partitioning; and we use Spark and Drill to exploit data parallelization for the distributed RDF management system. We also demonstrate how columnar storage formats, like Parquet and ORC, can affect the overall performance of the distributed RDF storage and SPARQL querying system.

Finally, we propose S3QLRDF, a distributed RDF management solution based on Property Table Partitioning schema built on top of Spark. Based on our extensive

evaluation of S3QLRDF with other open-source state-of-the-art systems using real and synthetic RDF datasets, we conclude that S3QLRDF system improves the efficiency of SPARQL query processing.

For future work, we consider further improvements of S3QLRDF system in terms of querying performance, especially for the query that involves flattening a number of complex columns. We aim at generating a better query plan with complex properties for less expensive retrieval. In addition to that, we plan to further extend the query translator to support more SPARQL fragments and adding statistics to the query evaluator while evaluating queries. For timely process and derive valuable insights from data produced in the Semantic Web, we aim at integrating a real-time data pipeline (e.g., Kafka[29] and Spark Streaming pipeline) to our system for RDF stream processing.

[29] https://kafka.apache.org/

PUBLISHED WORKS

Hassan, M., & Bansal, S. K. (2020). S3QLRDF: Property Table Partitioning Scheme for Distributed SPARQL Querying of large-scale RDF data. In *Proceedings of IEEE International Conference on Smart Data Services (SMDS)* (pp. 133-140).

Hassan, M., & Bansal, S. K. (2019). Data Partitioning Scheme for Efficient Distributed RDF Querying Using Apache Spark. In *Proceedings of IEEE International Conference on Semantic Computing (ICSC)* (pp. 24-31).

Hassan, M., & Bansal, S. K. (2018). RDF Data Storage Techniques for Efficient SPARQL Query Processing Using Distributed Computation Engines. In *Proceedings of IEEE International Conference on Information Reuse and Integration (IRI)* (pp. 323-330).

Hassan, M., & Bansal, S. K. (2018). Semantic Data Querying over NoSQL Databases with Apache Spark. In *Proceedings of IEEE International Conference on Information Reuse and Integration (IRI)* (pp. 364-371).

Mammo, M., Hassan, M., & Bansal, S. K. (2015). Distributed SPARQL Querying over Big RDF Data Using PRESTO-RDF. *International Journal of Big Data*, vol 2, no. 3.

REFERENCES

Abadi, D. J., Marcus, A., Madden, S. R., & Hollenbach, K. (2007). Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Databases* (pp. 411-422).

Abadi, D. J., Marcus, A., Madden, S. R., & Hollenbach, K. (2009). SW-Store: a vertically partitioned DBMS for Semantic Web data management. *The VLDB Journal*, 18(2), 385-406.

Aluç, G., Hartig, O., Özsu, M. T., & Daudjee, K. (2014). Diversified Stress Testing of RDF Data Management Systems. In *International Semantic Web Conference* (pp. 197-212). Springer, Cham.

UniProt Consortium. (2014). Activities at the Universal Protein Resource (UniProt). *Nucleic acids research*, 42(D1), D191-D198.

Aranda-Andújar, A., Bugiotti, F., Camacho-Rodríguez, J., Colazzo, D., Goasdoué, F., Kaoudi, Z., & Manolescu, I. (2012). AMADA: Web Data Repositories in the Amazon Cloud. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management* (pp. 2749-2751).

Atre, M., Srinivasan, J., & Hendler, J. A. (2009). BitMat: A Main Memory RDF Triple Store. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowladge Base System* (p. 33).

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., & Ives, Z. (2007). DBpedia: A Nucleus for a Web of Open Data. In *Proceedings of the 6th International Semantic Web Conference* (pp. 722-735). Springer.

Berners-Lee, T., Hendler, J. & Lassila, O. (2001). Scientific American: Feature Article: The Semantic Web.

Bizer, C., Heath, T., & Berners-Lee, T. (2011). Linked Data - The Story So Far. In *Semantic Services, Interoperability and Web Applications: Emerging Concepts* (pp. 205-227). IGI global.

Bizer, C., & Schultz, A. (2009). The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2), 1-24.

Boncz, P. A., Erling, O., & Minh Duc, P. (2013). Experiences with Virtuoso Cluster RDF Column Store. In *Linked Data Management* (pp. 239–259). Chapman and Hall/CRC, 2014.

Broekstra, J., Kampman, A., & Van Harmelen, F. (2002). Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *International*

Semantic Web Conference (pp. 54-68). Springer, Berlin, Heidelberg.

Callahan, A., Cruz-Toledo, J., Ansell, P., & Dumontier, M. (2013). Bio2RDF Release 2: Improved Coverage, Interoperability and Provenance of Life Science Linked Data. In *Extended Semantic Web Conference* (pp. 200-212). Springer.

Cossu, M., Färber, M., & Lausen, G. (2018). PRoST: Distributed Execution of SPARQL Queries Using Mixed Partitioning Strategies. In *21st International Conference on Extending Database Technology* (pp. 469–472).

Cudré-Mauroux, P., Enchev, I., Fundatureanu, S., Groth, P., Haque, A., Harth, A., & Wylot, M. (2013). NoSQL Databases for RDF: An Empirical Evaluation. In *International Semantic Web Conference* (pp. 310-325). Springer.

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *OSDI* (pp. 137-150).

Dong, X., Gabrilovich, E., Heitz, G., Horn, W., Lao, N., Murphy, K., & Zhang, W. (2014). Knowledge Vault: A Web-Scale Approach to Probabilistic Knowledge Fusion. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 601-610).

Du, J. H., Wang, H. F., Ni, Y., & Yu, Y. (2012). HadoopRDF: A Scalable Semantic Data Analytical Engine. In *International Conference on Intelligent Computing* (pp. 633-641). Springer.

Fu, G., Batchelor, C., Dumontier, M., Hastings, J., Willighagen, E., & Bolton, E. (2015). PubChemRDF: Towards the semantic annotation of PubChem compound and substance databases. *Journal of Cheminformatics*, 7(1), 1-15.

Gezer, V., & Bergweiler, S. (2016). Service and Workflow Engineering based on Semantic Web Technologies. In *Tenth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2016), International Academy, Research, and Industry Association (IARIA). IARIA* (Vol. 10, pp. 152-157).

Graux, D., Jachiet, L., Genevès, P., & Layaïda, N. (2016). SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark. In *International Semantic Web Conference* (pp. 80-87). Springer, Cham.

Guo, Y., Pan, Z., & Heflin, J. (2005). LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics*, 3(2-3), 158-182.

Harris, S., Lamb, N., & Shadbolt, N. (2009). 4store: The design and implementation of a clustered RDF store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)* (pp. 94-109).

Harth, A., Umbrich, J., Hogan, A., & Decker, S. (2007). YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *ISWC/ASWC* (pp. 211-224). Springer.

Hartig, O., Bizer, C., & Freytag, J. C. (2009). Executing SPARQL Queries over the Web of Linked Data. In *International Semantic Web Conference* (pp. 293-309). Springer.

Hassan, M., & Bansal, S. K. (2019). Data Partitioning Scheme for Efficient Distributed RDF Querying Using Apache Spark. In *Proceedings of IEEE International Conference on Semantic Computing (ICSC)* (pp. 24-31).

Hassan, M., & Bansal, S. K. (2020). S3QLRDF: Property Table Partitioning Scheme for Distributed SPARQL Querying of large-scale RDF data. In *Proceedings of IEEE International Conference on Smart Data Services (SMDS)* (pp. 133-140).

Hoffart, J., Suchanek, F. M., Berberich, K., & Weikum, G. (2013). YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence*, 194, 28-61.

Huai, Y., Chauhan, A., Gates, A., Hagleitner, G., Hanson, E. N., O'Malley, O., & Zhang, X. (2014, June). Major technical advancements in Apache Hive. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (pp. 1235-1246).

Huang, J., Abadi, D. J., & Ren, K. (2011). Scalable SPARQL Querying of Large RDF Graphs. In *Proceedings of the VLDB Endowment*, 4(11), 1123-1134.

Husain, M. F., McGlothlin, J., Khan, L., & Thuraisingham, B. (2011). Scalable complex query processing over large semantic web data using cloud. In *IEEE 4th International Conference on Cloud Computing* (pp. 187-194).

Goasdoué, F., Kaoudi, Z., Manolescu, I., Quiané-Ruiz, J. A., & Zampetakis, S. (2015). Cliquesquare: Flat plans for massively parallel RDF queries. In *IEEE 31st International Conference on Data Engineering* (pp. 771-782).

Khadilkar, V., Kantarcioglu, M., Thuraisingham, B., & Castagna, P. (2012). Jena-HBase: A distributed, scalable and efficient RDF triple store. In *Proceedings of the 11th International Semantic Web Conference Posters & Demonstrations Track, ISWC-PD* (Vol. 12, pp. 85-88).

Kornacker, M., Behm, A., Bittorf, V., Bobrovytsky, T., Ching, C., Choi, A., & Yoder, M. (2015). Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proceedings of the Conference on Innovative Data Systems Research* (Vol. 1, p. 9).

Ladwig, G., & Harth, A. (2011). CumulusRDF: linked data management on nested key-

value stores. In *the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)* (Vol. 30).

Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2), 35-40.

McBride, B. (2001). Jena: Implementing the RDF Model and Syntax Specification. In *2nd Int'l Semantic Web Workshop* (Vol. 40, pp. 23-28).

Neumann, T., & Weikum, G. (2008). RDF-3X: a RISC-style engine for RDF. In *Proceedings of the VLDB Endowment*, 1(1), 647-659.

Aluç, G., Özsu, M. T., & Daudjee, K. (2014). Workload matters: Why RDF databases need a new design. In *Proceedings of the VLDB Endowment*, 7(10), 837-840.

Aluç, G., Ozsu, M. T., Daudjee, K., & Hartig, O. (2013). chameleon-db: a Workload-Aware Robust RDF Data Management System. University of Waterloo, Tech. Rep. CS-2013-10.

Papailiou, N., Konstantinou, I., Tsoumakos, D., Karras, P., & Koziris, N. (2013, October). H2RDF+: High-performance distributed joins over large-scale RDF graphs. In *2013 IEEE International Conference on Big Data* (pp. 255-263).

Papailiou, N., Konstantinou, I., Tsoumakos, D., & Koziris, N. (2012). H2RDF: Adaptive Query Processing on RDF Data in the Cloud. In *Proceedings of the 21st International Conference on World Wide Web* (pp. 397-400).

Pérez, J., Arenas, M., & Gutierrez, C. (2009). Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3), 1-45.

Punnoose, R., Crainiceanu, A., & Rapp, D. (2012). Rya: A Scalable RDF Triple Store for the Clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence* (pp. 1-8).

Schätzle, A., Przyjaciel-Zablocki, M., Berberich, T., & Lausen, G. (2015). S2X: Graph-Parallel Querying of RDF with GraphX. In *Biomedical Data Management and Graph Online Querying* (pp. 155-168). Springer, Cham.

Schätzle, A., Przyjaciel-Zablocki, M., & Lausen, G. (2011). PigSPARQL: Mapping SPARQL to Pig Latin. In *Proceedings of the International Workshop on Semantic Web Information Management* (pp. 1-8).

Schätzle, A., Przyjaciel-Zablocki, M., Hornung, T., & Lausen, G. (2013). PigSPARQL: A SPARQL Query Processing Baseline for Big Data. In *International semantic web conference (posters & demos)* (Vol. 1035, pp. 241-244).

Schätzle, A., Przyjaciel-Zablocki, M., Neu, A., & Lausen, G. (2014). Sempala: Interactive SPARQL Query Processing on Hadoop. In *International Semantic Web Conference* (pp. 164-179). Springer, Cham.

Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., & Lausen, G. (2016). S2RDF: RDF Querying with SPARQL on Spark. In *Proceedings of the VLDB Endowment,* vol. 9, no. 10, pp. 804–815.

Schmidt, M., Hornung, T., Lausen, G., & Pinkel, C. (2009). SP2Bench: A SPARQL Performance Benchmark. In *2009 IEEE 25th International Conference on Data Engineering* (pp. 222-233). IEEE.

Weiss, C., Karras, P., & Bernstein, A. (2008). Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Proceedings of the VLDB Endowment*, 1(1), 1008-1019.

Wilkinson, K., & Wilkinson, K. (2006). Jena Property Table Implementation. In *Proceedings of the Second International Workshop on Scalable Semantic Web Knowledge Base Systems*, pages 54–68.

Wu, W., Li, H., Wang, H., & Zhu, K. Q. (2012). Probase: A Probabilistic Taxonomy for Text Understanding. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (pp. 481-492).

Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., & Liu, L. (2013). TripleBit: a Fast and Compact System for Large Scale RDF Data. In *Proceedings of the VLDB Endowment*, 6(7), 517-528.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., Mccauley, M., & Stoica, I. (2012). Fast and Interactive Analytics over Hadoop Data with Spark. *Usenix Login*, 37(4), 45-51.

Zou, L., Mo, J., Chen, L., Özsu, M. T., & Zhao, D. (2011). gStore: Answering SPARQL Queries via Subgraph Matching. In *Proceedings of the VLDB Endowment*, 4(8), 482-493.

APPENDIX A

YAGO2 QUERIES

```
BASE <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

Y1: SELECT ?GivenName ?FamilyName WHERE {
        ?p          hasGivenName        ?GivenName .
        ?p          hasFamilyName       ?FamilyName .
        ?p          rdf:type            ?scientist .
        ?scientist  rdfs:label          "scientist" .
        ?p          wasBornIn           ?city1 .
        ?city1      isLocatedIn         "France" .
        ?p          hasAcademicAdvisor  ?a .
        ?a          wasBornIn           ?city2 .
        ?city2      isLocatedIn         "United_States" .
}

Y2: SELECT ?name WHERE {
        ?a      isCalled    ?name .
        ?a      rdf:type    ?actor .
        ?actor  rdfs:label  "actor" .
        ?a      actedIn     ?m1 .
        ?a      directed    ?m2 .
        ?m1     rdf:type    ?movie .
        ?movie  rdfs:label  "movie" .
        ?m1     isLocatedIn "Portugal" .
        ?m2     rdf:type    ?movie .
        ?m2     isLocatedIn "Spain" .
}

Y3: SELECT DISTINCT ?name1 ?name2 WHERE {
        ?p1         hasFamilyName   ?name1 .
        ?p2         hasFamilyName   ?name2 .
        ?p1         rdf:type        ?scientist .
        ?p2         rdf:type        ?scientist .
        ?scientist  rdfs:label      "scientist" .
        ?p1         hasWonPrize     ?award .
        ?p2         hasWonPrize     ?award .
        ?p1         wasBornIn       ?city .
        ?p2         wasBornIn       ?city .
        FILTER (?p1 != ?p2)
}

Y4: SELECT DISTINCT ?name1 ?name2 WHERE {
        ?p1  isCalled   ?name1 .
        ?p1  wasBornIn  ?city1 .
        ?p1  actedIn    ?movie .
        ?p2  isCalled   ?name2 .
        ?p2  wasBornIn  ?city2 .
        ?p2  actedIn    ?movie .
```

```
        ?city1   isLocatedIn   “United_States” .
        ?city2   isLocatedIn   “United_States” .
        FILTER (?p1 != ?p2)
}

Y5:  SELECT  ?name1  ?name2  WHERE {
        ?p1   isCalled         ?name1 .
        ?p1   wasBornIn      ?city .
        ?p1   isMarriedTo   ?p2 .
        ?p2   isCalled         ?name2 .
        ?p2   wasBornIn      ?city .
}
```

APPENDIX B

LUBM QUERIES

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>

Q4:  SELECT ?X ?Y1 ?Y2 ?Y3 WHERE {
        ?X  rdf:type          ub:FullProfessor .
        ?X  ub:worksFor       <http://www.Department0.University0.edu> .
        ?X  ub:name           ?Y1 .
        ?X  ub:emailAddress   ?Y2 .
        ?X  ub:telephone      ?Y3 .
}

Q8:  SELECT ?X ?Y ?Z WHERE {
        ?X  rdf:type                ub:UndergraduateStudent .
        ?Y  rdf:type                ub:Department .
        ?X  ub:memberOf             ?Y .
        ?Y  ub:subOrganizationOf    <http://www.University0.edu> .
        ?X  ub:emailAddress         ?Z .
}

Q12: SELECT ?X ?Y WHERE {
        ?X  rdf:type                ub:FullProfessor .
        ?Y  rdf:type                ub:Department .
        ?X  ub:worksFor             ?Y .
        ?Y  ub:subOrganizationOf    <http://www.University0.edu> .
}
```

APPENDIX C

YAGO QUERIES

```
BASE  <http://yago-knowledge.org/resource/>

C:  SELECT  ?country ?capital ?lang ?geo ?lon ?lat ?area ?population ?inst ?player ?city1 ?city2
    WHERE {
        ?geo      hasLongitude          ?lon .
        ?geo      hasLatitude           ?lat .
        ?geo      hasArea               ?area .
        ?geo      linksTo               ?lang .
        ?country  hasOfficialLanguage   ?lang .
        ?country  hasNumberOfPeople     ?population .
        ?country  hasCapital            ?capital .
        ?capital  linksTo               ?inst .
        ?player   playsFor              ?inst .
        ?player   wasBornIn             ?city1 .
        ?player   diedIn                ?city2  .
}

F:  SELECT  ?gname1  ?gname2  ?fname1  ?fname2  ?city1  ?city2  WHERE {
        ?p1  hasGivenName     ?gname1 .
        ?p2  hasGivenName     ?gname2 .
        ?p1  hasFamilyName    ?fname1 .
        ?p2  hasFamilyName    ?fname2 .
        ?p1  isMarriedTo      ?p2 .
        ?p1  wasBornIn        ?city1 .
        ?p2  wasBornIn        ?city2 .
 }

S:  SELECT  ?geo  ?lon  ?lat  ?area  ?wiki  ?lang  WHERE {
        ?geo  hasLongitude       ?lon .
        ?geo  hasLatitude        ?lat .
        ?geo  hasArea            ?area .
        ?geo  hasWikipediaUrl    ?wiki .
        ?geo  linksTo            ?lang .
 }

L:  SELECT  ?country  ?capital  ?lang  ?geo  ?area  WHERE {
        ?geo      hasArea               ?area .
        ?geo      linksTo               ?lang .
        ?country  hasOfficialLanguage   ?lang .
        ?country  hasCapital            ?capital .
 }
```

APPENDIX D

DBLP QUERIES

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

C: **SELECT** ?v0 ?homepage ?name ?v1 ?year ?isbn ?publisher ?v2 ?title ?creator **WHERE** {
?v0 foaf:homepage ?homepage .
?v0 foaf:name ?name .
?v1 swrc:editor ?name .
?v1 dcterms:issued ?year .
?v1 swrc:isbn ?isbn .
?v1 dc:publisher ?publisher .
?v2 dcterms:partOf ?v1 .
?v2 dc:title ?title .
?v2 swrc:series <http://dblp.l3s.de/d2r/resource/collections/crypt> .
?v2 dc:creator ?creator .
}

F: **SELECT** ?v0 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 **WHERE** {
?v0 swrc:series <http://dblp.l3s.de/d2r/resource/conferences/genetic> .
?v0 foaf:homepage ?v2 .
?v0 dcterms:bibliographicCitation ?v3 .
?v0 dcterms:issued ?v4 .
?v0 dc:title ?v5 .
?v0 dc:creator ?v6 .
?v6 foaf:name ?v7 .
?v6 rdf:type ?v8 .
}

S: **SELECT** ?v0 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9 ?v10 ?v11 ?v12 ?v13 ?v14 ?v15 ?v16
WHERE {
?v0 swrc:journal <http://dblp.l3s.de/d2r/resource/journals/vldb> .
?v0 foaf:homepage ?v2 .
?v0 dc:creator ?v3 .
?v0 foaf:maker ?v4 .
?v0 rdfs:seeAlso ?v5 .
?v0 dc:identifier ?v6 .
?v0 dc:title ?v7 .
?v0 dc:type ?v8 .
?v0 dcterms:bibliographicCitation ?v9 .
?v0 dcterms:issued ?v10 .
?v0 swrc:number ?v11 .
?v0 swrc:pages ?v12 .
?v0 swrc:volume ?v13 .

```
        ?v0   rdf:type                  ?v14 .
        ?v0   rdfs:label                ?v15 .
        ?v0   owl:sameAs                ?v16 .
}

L:  SELECT  ?v0  ?v1  ?v2  WHERE {
        ?v0   dcterms:issued   "2017" .
        ?v0   swrc:journal     ?v1 .
        ?v1   rdfs:label       ?v2 .
}
```