S3QLRDF: Distributed SPARQL Query Processing Using Apache Spark - A Comparative Performance Study

Mahmudul Hassan (phassan@asu.edu)
Arizona State University
Srividya Bansal
Arizona State University

Research Article

Keywords: RDF , SPARQL , Data Partitioning , Spark

Posted Date: June 1st, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1677298/v1

S3QLRDF: Distributed SPARQL Query Processing Using Apache Spark - A Comparative Performance Study

Mahmudul Hassan[1] · Srividya Bansal[1]

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. More precisely, 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 time. In this context, we propose a new relational partitioning schema called Property Table Partitioning (PTP) for RDF data, that further partitions existing Property Table into multiple tables based on distinct properties (comprising of all subjects with non-null values for those distinct properties) in order to minimize input size and number of *join* operations of a query. This paper proposed a distributed RDF data management system called S3QLRDF, which is built on top of Spark and utilizes SQL to execute SPARQL queries over PTP schema. The experimental analysis with respect to preprocessing costs and query performance, using synthetic and real datasets shows that S3QLRDF outperforms state-of-the-art distributed RDF management systems.

Keywords RDF · SPARQL · Data Partitioning · Spark

✉ Mahmudul Hassan
phassan@asu.edu

Srividya Bansal
srividya.bansal@asu.edu

[1] Arizona State University, Tempe, Arizona, USA

1 Introduction

The Semantic Web refers to a Web of Data that associates the semantics of information and services on the web to provide machine-readable and processable data. The RDF[1] is a data model proposed by W3C to represent metadata about Web resources and facilitates the search engine to precisely locate and extract information on the Semantic Web. Recently, RDF has gained 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. There are a growing number of organizations, institutions, and companies adopting Semantic Web technologies to represent data in a semantically structured way and thereby contributing to the Web of Data. Top search engine providers, Google, Bing, Yahoo!, and Yandex, have agreed to create a protocol (Schema.org[2]) for a structured data vocabulary in order to define entities, actions, and relationships through the internet, which helps search engines figure out the meanings on web pages more effectively and serve relevant results based on search queries of the internet users. To improve the accuracy of recommendations, recommender companies are increasingly using semantics and semantic tagging. DBpedia [1], YAGO [2], Bio2RDF [3], Google's Knowledge Vault [4], Probase [5], PubChemRDF [6], and Universal Protein Resource (UniProtKB) [7] consist of billions of facts that are represented as RDF data contained in the Linked Open Data (LOD) [8] cloud. Therefore, we can expect the Semantic Web to grow steadily at web-scale and produce a large amount of RDF data. This steady growth of RDF data necessitates an efficient RDF management solution for storing and querying these very large RDF graphs. Over the last decade, many RDF data management systems have been designed to provide scalable, highly available, and fault-tolerant RDF stores with efficient SPARQL[3] query processing for distributed environments (e.g., Partout [9], DREAM [10]). In the last few years, many distributed RDF management systems are built on Big Data technologies like Hadoop (e.g., Rya [11], H2RDF+ [12], SHARD [13], CliqueSquare [14], PigSPARQL [15], Sempala [16], S2RDF [17], SPARQLGX [18], PRoST [19]). These RDF data processing systems rely on cluster computing engines based on MapReduce [20] as an execution layer or in-memory frameworks such as Spark[4] and Impala [21]. In most cases, these systems are optimized for specific query patterns. Some of these existing systems often give up their data preprocessing time for better querying performance. Therefore, it is necessary to implement a distributed RDF management system for efficient query performance on a wide range of query patterns with minimized preprocessing overhead, and that is the goal of this work.

To achieve the above goal, RDF data partitioning strategy using existing approaches, Property Table [22] and Vertical Partitioning (VP) [23], to form SPT + VP [24] storage layout has been proposed. The combined SPT + VP RDF management solution outperforms state-of-the-art systems for all types of query

[1] https://www.w3.org/TR/rdf-concepts/
[2] https://schema.org/
[3] https://www.w3.org/TR/rdf-sparql-query/
[4] https://spark.apache.org/

patterns except a few complex-shaped queries. To overcome the query performance issue on complex query patterns, we further partition the Property Table into multiple tables based on distinct properties to propose a new storage schema called Property Table Partitioning (PTP). For storing and querying RDF data, we use HDFS[5] (Hadoop Distributed File System) and an in-memory cluster computing framework Spark, one of the most important and popular Hadoop ecosystem components.

This paper is an extension of the publication [25], where an initial version of the S3QLRDF system had been presented, by adding the following novel contributions:

- An experimental evaluation between Big Data file formats, Parquet and ORC, using the S3QLRDF system with the PTP schema that demonstrates the impact of using different file formats for storing RDF data.
- An empirical comparison of open-source Spark-based state-of-the-art systems: S3QLRDF, S2RDF, SPARQLGX, and PRoST based on real datasets, YAGO and DBLP, confirms the effectiveness and applicability of our approach.

Giving this outlook, rest of this paper is presented as follows: We introduce the background with preliminary definitions used in section 2. Section 3 studies the related work. A detailed overview of S3QLRDF (SPARQL on Spark SQL for RDF) system is given in Section. 4. Experimental evaluation of S3QLRDF with other state of the art Hadoop-based SPARQL query engines is presented in Section. 5. Section 6 illustrates the impact of using different Big Data columnar file formats on the S3QLRDF system for RDF data management. The comparative performance evaluation of the S3QLRDF system with other state-of-the-art Spark-based SPARQL processors using real datasets is demonstrated in Section 7. Section 8 concludes the paper.

2 Background

In this section, we briefly introduce background information on the RDF data and SPARQL query model followed by the Big Data technologies used.

2.1 RDF

RDF is a schema-free data model recommended by W3C to describe information about any resource 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), subject denotes the entity or a class of resources; predicate denotes the attribute or aspect and relationship (aka property) between entities or classes; and object denotes an entity, class, or 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 properties. Each node can be an Internationalized Resource Identifier (IRI), a literal or blank node.

[5] https://hadoop.apache.org/

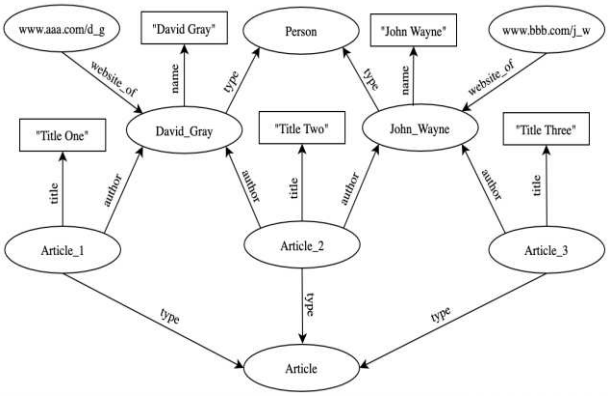

Fig. 1 An example RDF graph

Fig. 1 simulates an RDF graph with 16 edges of a simple publication network of an RDF dataset that consists of 16 triples, where ellipse nodes represent resources, directed edges represent properties, and rectangular nodes represent literal values. An RDF graph could also have blank nodes that represent resources without URI or literal assignment. Let I, B, and L be infinite sets of IRIs, blank nodes, and literals respectively which are pairwise disjoint. All RDF valid terms are the union of (I ∪ B ∪ L) and denoted by T.

RDF Triple A ternary tuple (s, p, o) ∈ (I ∪ B) ⨯ I ⨯ (I ∪ B ∪ L) is called an RDF triple where s, p, and o denote subject, predicate, and object respectively.

RDF Graph An RDF graph G = {t_1, …. t_n} is a finite set of RDF triples t_i where 1 ≤ i ≤ n.

RDF Dataset An RDF dataset is a collection of RDF graphs D = {G_0, (i_1, G_1), …. (i_n, G_n)} with i_1, . . . , i_n ∈ I. The pairs (i_i, G_i) are named graphs identified by IRI and the default graph is G_0 that does not have a name.

2.2 SPARQL

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 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 that the query is being executed on. A SPARQL query can be one of 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 Constraints* (FGP), *Optional Graph Pattern* (OGP), *Union Graph Pattern* (UGP) or *Alternative Graph Pattern* (AGP), and *Group Graph Pattern* (GGP). BGP, FGP, and OGP consist 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 elements of the RDF graph. A 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). SPARQL BGPs fall in one of the four following categories:

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.

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

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

Complex Structure is the compositions of the above-mentioned query patterns.

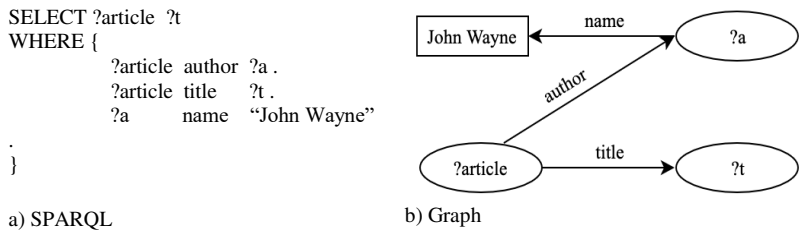

Fig. 2 A SPARQL query that finds the articles with their corresponding title written by John Wayne

Fig. 2a represents a SPARQL query that returns the title of articles written by John Wayne. The corresponding graph pattern of the SPARQL query is shown in Fig. 2b. The result is the set of ordered bindings of (?article, ?t) that render the query graph isomorphic to subgraphs in the data. Assuming data are stored in a table D(s, p, o), the query can be answered by first decomposing it into three subqueries: $q_1 \equiv \sigma_{p = name \wedge o = John\ Wayne}(D)$, $q_2 \equiv \sigma_{p = author}(D)$, and $q_3 \equiv \sigma_{p = title}(D)$. The subqueries are answered independently by scanning table D; then, their intermediate results are joined on the subject and object attribute: $q_1 \bowtie_{q1.s = q2.o} q_2 \bowtie_{q2.s = q3.s} q_3$. By applying the query on the data in Figure 1, we get (?article, ?t) $\in$ {(Article_2, Title Two), (Article_3, Title Three)}.

2.3 Hadoop & Spark

Hadoop is an open-source framework for distributed storage and processing of large datasets based on the HDFS and MapReduce paradigm. HDFS is a 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 (e.g., Spark).

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[6] 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.

2.4 Hadoop & Spark

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

Parquet[7] 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.

```
4–byte magic number "PAR1"
<Column 1 Chunk 1 + Column Metadata>
<Column 2 Chunk 1 + Column Metadata>
...
<Column N Chunk 1 + Column Metadata>
<Column 1 Chunk 2 + Column Metadata>
<Column 2 Chunk 2 + Column Metadata>
...
<Column N Chunk 2 + Column Metadata>
...
<Column 1 Chunk M + Column Metadata>
<Column 2 Chunk M + Column Metadata>
...
<Column N Chunk M + Column Metadata>
File Metadata
4–byte length in bytes of file metadata
4–byte magic number "PAR1"
```

Fig. 3 The Parquet File Format[8]

Parquet stores data organized by horizontal partitions called row groups. For each row group, the data values are organized by column chunk. Each column chunk corresponds to a column in the data set. A column chunk consists of multiple pages where each page contains values for a particular column. Parquet stores metadata at all the levels in the hierarchy (i.e., file, column chunk, and page). A sample parquet file format is shown in Fig. 3. This data format supports additional optimizations include encodings (bit packing, run length, and dictionary encoding) as well as compression algorithms like Snappy[9], GZip[10], LZO[11], 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[12] (Optimized Row Columnar) is a columnar file format that provides a

[6] https://spark.apache.org/sql/

[7] https://parquet.apache.org/

[8] https://parquet.apache.org/documentation/latest/

[9] http://google.github.io/snappy/

[10] https://www.gnu.org/software/gzip/

[11] http://www.oberhumer.com/opensource/lzo/

[12] https://orc.apache.org/

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 they are used in table scans. The stripe footer contains a directory of stream locations. Fig. 4 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.

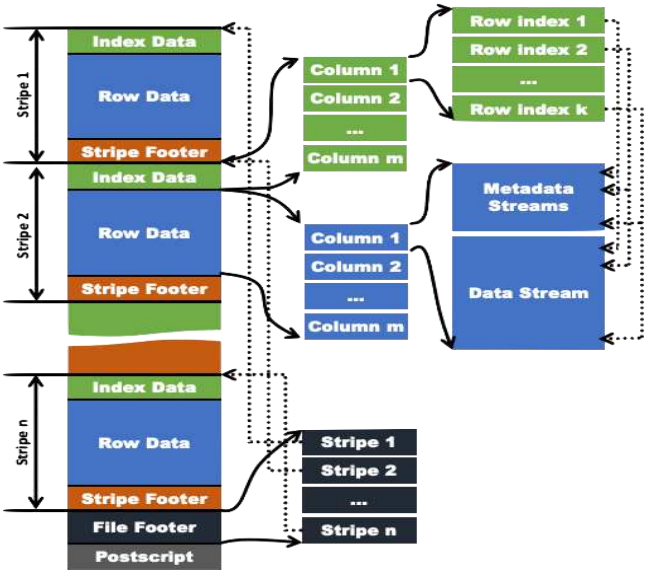

Fig. 4 The ORC File Format [26]

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[13], 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.

3 Related work

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. These systems use various indexing and partitioning strategies on RDF elements to develop RDF storage layouts. In this section, we discuss the state-of-the-art distributed RDF management systems that are relevant to this work.

Rya [11] has been implemented on top of a key-value store Accumulo[14] 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.

CliqueSquare [14] 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.

S2RDF [17] has been built on top of Spark that uses a relational partitioning technique called Extended Vertical Partitioning (ExtVP) which is an extension of Vertical Partitioning (VP) [27] approach to store RDF data on the HDFS using Parquet 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.

SPARQLGX [18] also built on top of Spark uses Vertically Partitioned approach proposed in [27] 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 [19] is a Spark based distributed system for RDF storage and SPARQL querying that stores data twice using Vertical Partitioning and Property Table. PRoST

[13] https://zlib.net/
[14] https://accumulo.apache.org/

translates SPARQL queries into the Join Tree format where every node represents either the Vertical Partitioning table or Property Table. The triple patterns with 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 1 Summary of Distributed RDF Systems

System	Storage Strategy	Storage Backend	Execution Framework
Rya	3 Indices (SPO, POS, OSP)	Key-Value Store	OpenRDF Sesame Framework
CliqueSquare	Hash and Vertical Partitioning	Distributed File System	MapReduce
S2RDF	Vertical Partitioning and Extended Vertical Partitioning	Distributed File System	SPARQL to SQL
SPARQLGX	Vertical Partitioning	Distributed File System	SPARQL to Scala Code
PRoST	Vertical Partitioning	Distributed File System	SPARQL to SQL

4 S3QLRDF architecture

In this section, we present the overall architecture of S3QLRDF[15] system. It consists of three main components: Data Loader – RDF data ingestion and partitioning using PTP schema, Query Translator – Spark SQL query generator from the SPARQL query, and Query Evaluator – Spark SQL query evaluated directly into the Spark SQL engine (Fig. 5).

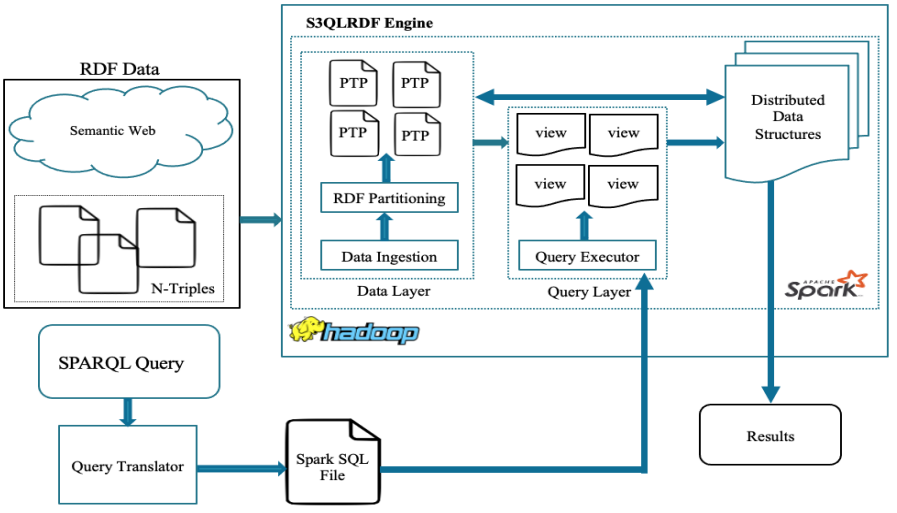

Fig. 5 S3QLRDF Architecture Overview

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

Data Loader S3QLRDF comes with a novel RDF data partitioning strategy called PTP schema. RDF data is first loaded into HDFS, and then Spark read and partition the data using the PTP schema that is a modified and enhanced version of the well-known PT schema introduced by Wilkinson et al. [22].

Table 2 Summary Modified Property Table for RDF Graph of Figure 1 (empty cells = NULL)

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

We introduced the *Modified Property Table* in [24] which is a modified version of the traditional PT where multi-valued properties are stored in a single cell using a nested data structure (e.g., Array). We briefly present the *Modified Property Table* schema followed by our proposed PTP schema; an extension of the *Modified Property Table* approach. 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:

$$PT(subject, property_1, ..., property_n)$$

where n is the total number of distinct properties present in a particular RDF dataset. Here, each RDF subject is stored in the subject column and their object values reside in their corresponding property columns.

Next, we partition the *Modified Property Table* into multiple tables based on distinct properties present in the RDF dataset to devise our proposed PTP schema. 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 shows the proposed RDF data layout that is obtained from partitioning the whole *Modified Property Table* (Table 2).

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*.

Table 3 Property Table Partitioning Schema for RDF Graph shown in Fig. 1

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]

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]

name

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

website_of

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

Query Translator The query translator generates the equivalent Spark SQL expressions from SPARQL query based on PTP schema using the statistics file that is generated during the PTP tables creation process. 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:

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

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). 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.subjectAS 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 SPARQL query is fed to the Spark engine as an equivalent Spark SQL query.

Query Executor In this process, the Spark SQL query created by the query translator is directly evaluated into the Spark SQL engine.

5 Evaluation

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.

5.1 Benchmark Queries

For the performance evaluation of our RDF management solutions, we utilize two synthetic and one real dataset, as shown in Table 4. 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 4 Experimental Setup - Dataset Scale

Dataset		Number of Triples (million)
	Number of Universities	
LUBM	1000	138
	5000	691
	10000	1381
	Scale Factor	
WatDiv	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.

5.2 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 @ 2.30GHz processor, 120 GB of memory, and 1 TB 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.

5.3 Empirical Comparison

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 5. 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.

Table 5 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**

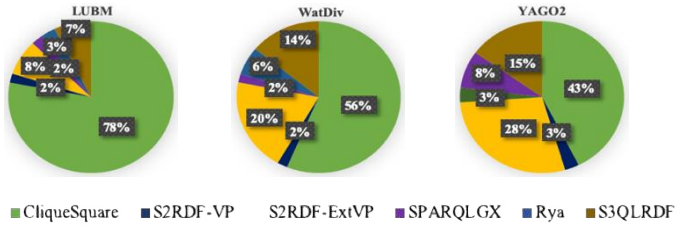

Fig. 6 Storage Space Distributions with Datasets

Fig. 6 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 5, 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.

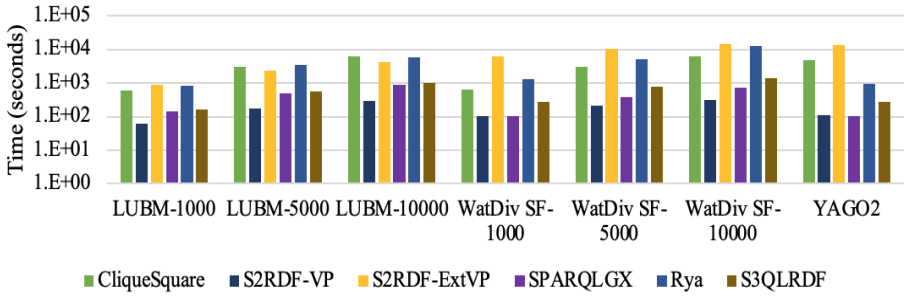

Fig. 7 Time Distributions with Datasets (log scale)

From Fig. 7, 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.

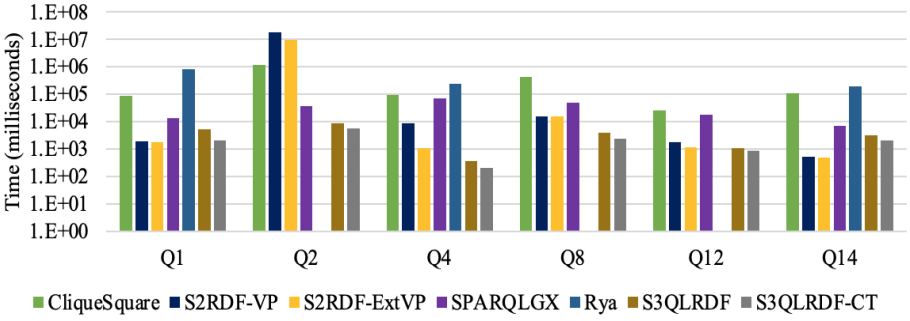

Fig. 8 Performance Comparison for LUBM 10000 (log scale)

The performance comparison for LUBM 10000 is illustrated in Fig. 8 on a log scale while absolute runtimes are given in Table 6. 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.

Table 6 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**

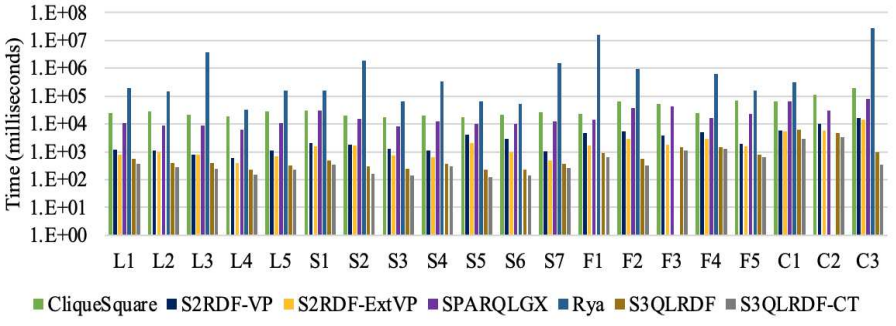

Fig. 9 Performance Comparison for WatDiv SF10000 (log scale)

Fig. 9 compares the different systems on the largest dataset (SF10000) of WatDiv, corresponding AM runtimes are listed in Table 7. For WatDiv, S3QLRDF and S3QLRDF-CT show a competitive runtime performance for all query categories when increasing the size of the dataset. In Table 7, 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.

Table 7 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**

Table 7 continued

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**

Table 7 continued

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**

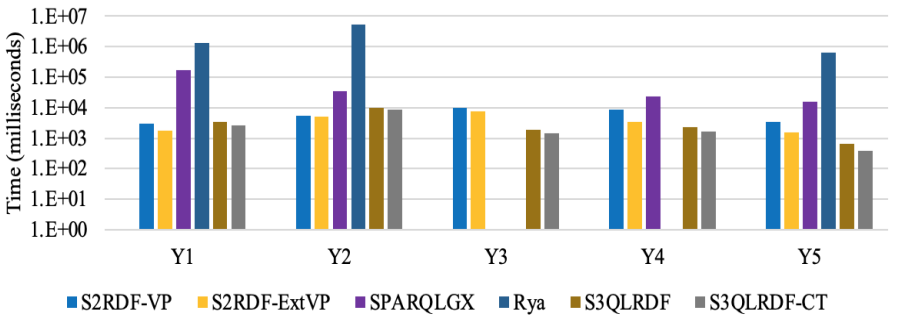

Fig. 10 Performance Comparison for YAGO2 (log scale)

Table 8 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**

Fig. 10 illustrates the execution times for YAGO2 queries of all compared systems while absolute runtimes, and Query/hr are given in Table 8. 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.

In this section, we conduct a comparative performance evaluation of the SQL 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. Our proposed S3QLRDF system outperforms state-of-the-art distributed SPARQL query processors by an order of magnitude on average for all query shapes.

6 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.

6.1 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.

6.2 Empirical Comparison

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

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 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 109 million 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.

Table 9 WatDiv and YAGO Loading Times and HDFS Sizes

Dataset	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

We report datasets loading times and HDFS sizes for PTP schema based on Parquet and ORC file formats in Table 9. From Table 9, we can see 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.

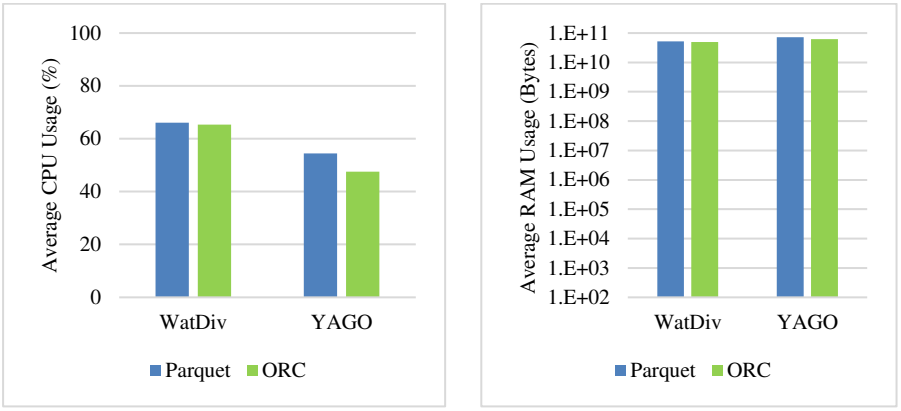

Fig. 11 CPU and RAM Consumptions During Data Loading Phase

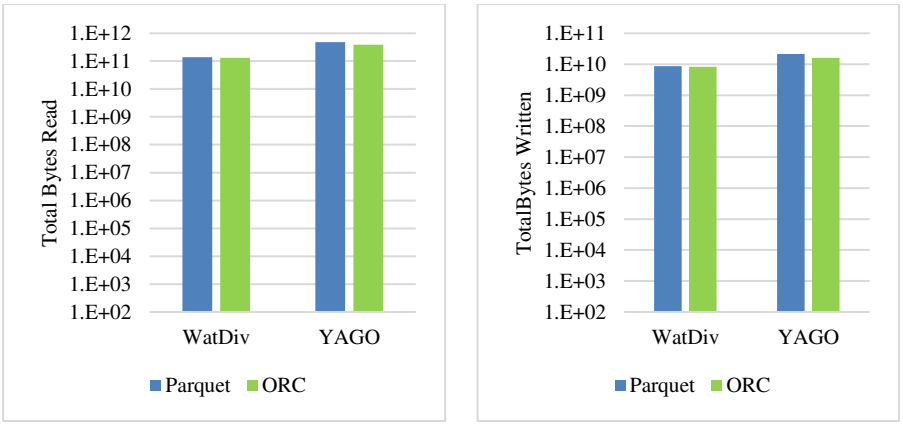

Fig. 12 Total HDFS Bytes Read/Written During Data Loading Phase

Fig. 11 and 12 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 10 WatDiv Basic Testing (milliseconds)

WatDiv-C	Parquet	ORC	WatDiv-F	Parquet	ORC
W/O-CT	18787	26521		30606	49703
CT	9932	8765		14596	13081
T-CT	34196	45296		52561	72198
WatDiv-S			WatDiv-L		
W/O-CT	31561	49325		22743	35096
CT	8079	7011		5454	4906
T-CT	47936	67483		35906	51103

Table 11 YAGO Query Run Times (milliseconds)

YAGO-C	Parquet	ORC	YAGO-F	Parquet	ORC
W/O-CT	146028	152791		5228	8321
CT	120097	122743		1704	1593
T-CT	136335	143867		8819	12435
YAGO-S			YAGO-L		
W/O-CT	59349	78218		4974	8232
CT	58876	68082		1541	1394
T-CT	64776	76602		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 10 and 11 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 section 7, 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 5.3.

7 Empirical Evaluation of Spark-based RDF Management Systems

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.

7.1 Benchmarked SPARQL Evaluators

In this section, we present a brief overview on Spark-based RDF management systems, namely S3QLRDF, S2RDF, SPARQLGX, and PRoST. Table 12 shows the RDF data partitioning techniques used in the state-of-the-art Spark-based systems.

Table 12 Partitioning Strategies of Spark-based RDF Management Solutions

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

Spark-based systems listed in Table 12 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) [19] for their storage layout. Table 13 represents the RDF query processing methods used in Spark-based systems based on Spark data abstraction.

Table 13 Data Access Model of Spark-based RDF Management Solutions

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

Table 14 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

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 14. 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[16] 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. We keep the same cluster configuration mentioned in the section 6.2.

7.2 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 15. From Table 15, 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 15 Loading Times and HDFS Sizes

		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

The following Fig. 13 and 14 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

[16] https://dblp.org/

discussion, we can conclude that S2RDF is the costliest system for the cluster because of the highest data loading times and RAM usages.

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.

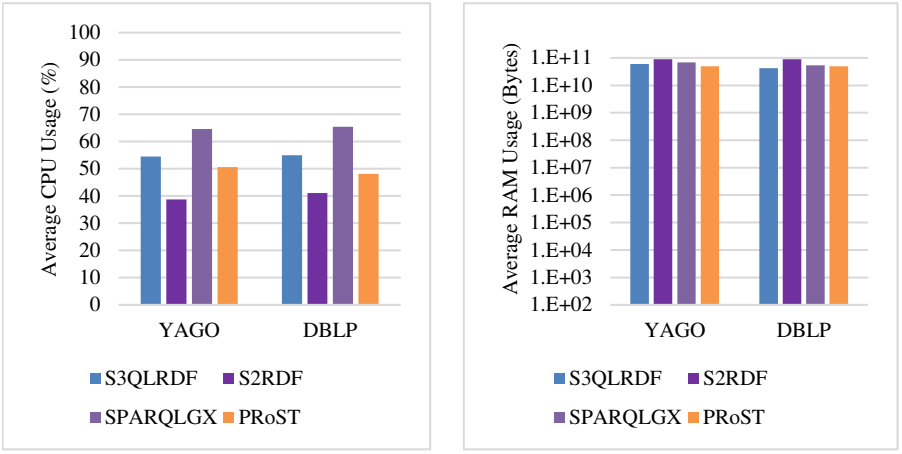

Fig. 13 CPU and RAM Consumptions During Data Loading Phase

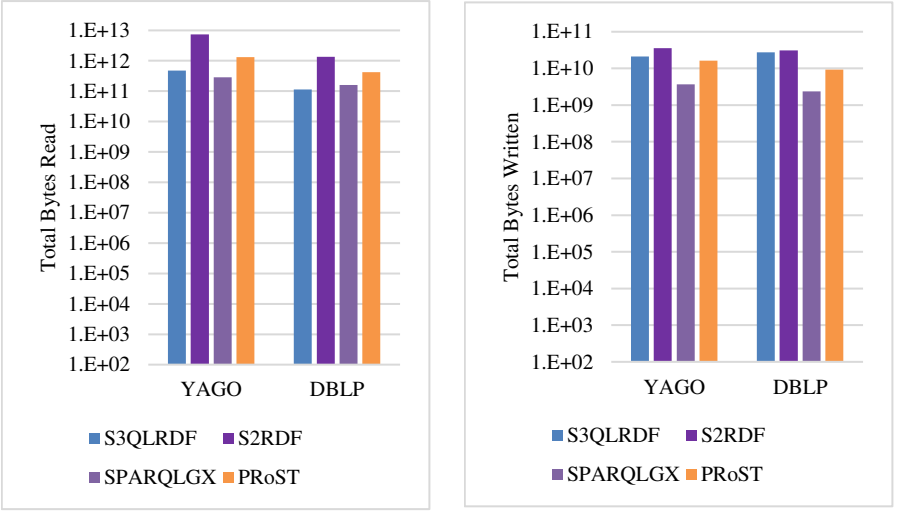

Fig. 14 Total HDFS Bytes Read/Written During Data Loading Phase (log scale)

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. The following Fig. 15 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.

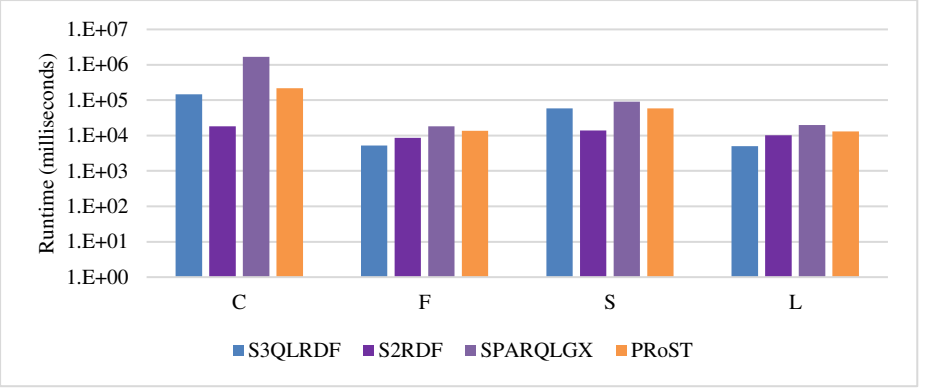

Fig. 15 YAGO Query Run Times (log scale)

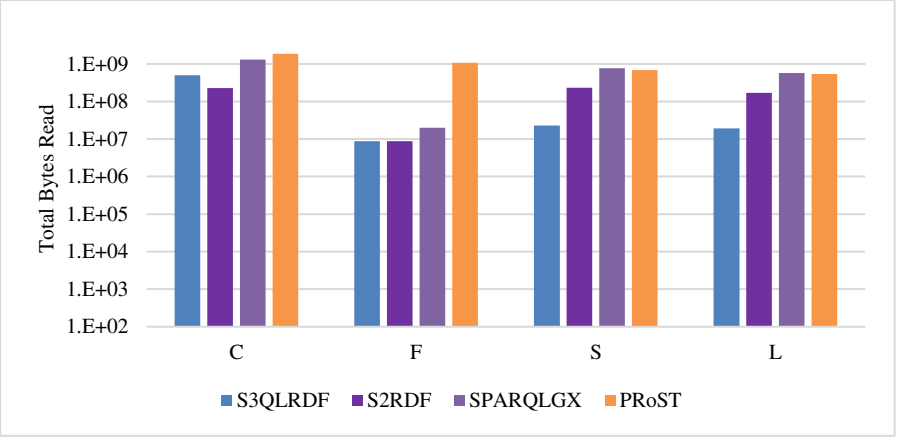

Fig. 16 Total HDFS Bytes Read During YAGO Query Phase (log scale)

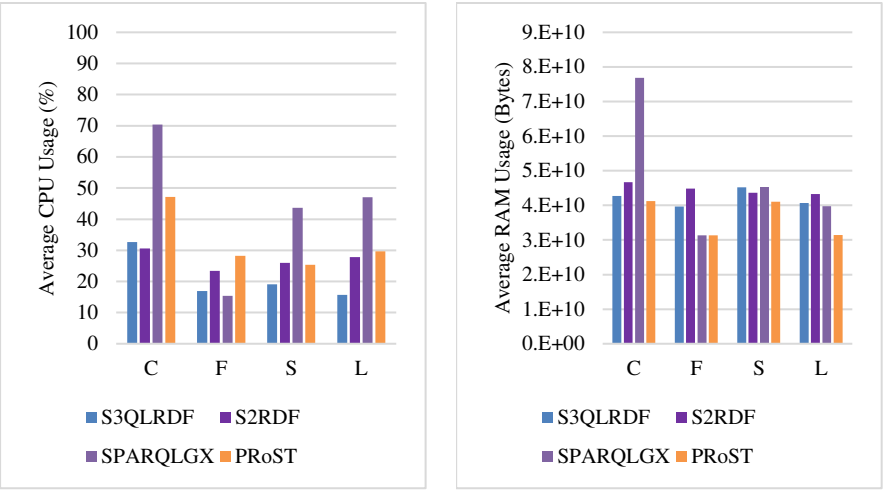

Fig. 17 CPU and RAM Consumptions During YAGO Query Phase

S2RDF trades off the query performances with disk space and loading time. SPARQLGX has poor runtimes for all queries among all systems. From Fig. 16 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 Fig. 17 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.

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

Fig. 18 DBLP Query Run Times (log scale)

Fig. 19 Total HDFS Bytes Read During DBLP Query Phase (log scale)

Fig. 18 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. We can also observe from Fig. 19 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 (Fig. 20).

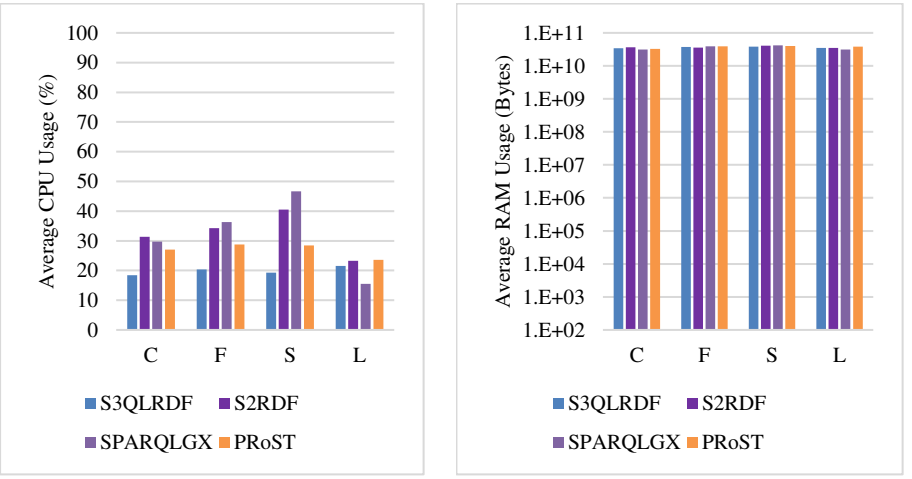

Fig. 20 CPU and RAM Consumptions During DBLP Query Phase

In this section, 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.

8 Conclusion

In this paper, we focus on two key elements in the distributed system for efficient SPARQL query processing; data parallelization and data partitioning. We propose a novel RDF data partitioning schema called Property Table Partitioning; and we use Spark 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. We presented 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.

Acknowledgements Authors gratefully acknowledge the support for this project under Google Cloud Platform and AWS for Research sponsorship to run the experiments using their Cloud Computing services.

References

1. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, “DBpedia: A nucleus for a Web of open data,” in The semantic web, Springer, pp. 722–735, 2007.
2. J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum, “YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia,” Artif. Intell., vol. 194, pp. 28–61, 2013.
3. A. Callahan, J. Cruz-Toledo, P. Ansell, and M. Dumontier, “Bio2RDF Release 2: Improved Coverage, Interoperability and Provenance of Life Science Linked Data,” in Proc. 10th Int. Conf. The Semantic Web: Semantics Big Data, pp. 200–212, 2013.
4. X. Dong et al., “Knowledge Vault: A Web-Scale Approach to Probabilistic Knowledge Fusion,” in Proc. 20th ACM SIGKDD Int. Conf. Knowl. Disc. Data Mining, pp. 601–610, 2014.
5. W. Wu, H. Li, H. Wang, and K. Q. Zhu, “Probase: A Probabilistic Taxonomy for Text Understanding” in Proc. ACM SIGMOD Int. Conf. Management Data, Scottsdale, AZ, pp. 481–492, 2012.
6. G. Fu, C. Batchelor, M. Dumontier, J. Hastings, E. Willighagen, and E. Bolton, “PubChemRDF: Towards the semantic annotation of PubChem compound and substance databases,” J. Cheminform., vol. 7, no. 1, pp. 1–15, 2015.
7. R. Apweiler et al., “Activities at the Universal Protein Resource (UniProt),” Nucleic Acids Res., vol. 42, no. D1, pp. 191–198, 2014.

8. C. Bizer, T. Heath, and T. Berners-Lee, “Linked Data - The Story So Far,” Int. J. Semant. Web Inf. Syst., vol. 5, no. 3, pp. 1–22, 2009.
9. L. Galárraga and R. Schenkel, “Partout : A Distributed Engine for Efficient RDF Processing,” in World Wide Web, pp. 267–268, 2014.
10. M. Hammoud et al., “DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication,” in Proceedings of the VLDB Endowment, vol. 8, no. 6, pp. 654–665, 2015.
11. R. Punnoose, A. Crainiceanu, and D. Rapp, “Rya: A Scalable RDF Triple Store for the Clouds,” in Proceedings of the ACM 1st International Workshop on Cloud Intelligence, ACM., p. 4, 2012.
12. N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris, “H2RDF+: High-performance distributed joins over large-scale RDF graphs,” in Proc. IEEE International Conference on Big Data, pp. 255–263, 2013.
13. K. Rohloff and R. E. Schantz, “Clause-iteration with MapReduce to Scalably Query Datagraphs in the SHARD Graph-store,” in Proc. 4th Int. Workshop Data-Intensive Distrib. Comput., pp. 35–44, 2011.
14. Z. Kaoudi, I. Manolescu, and S. Zampetakis, “CliqueSquare : Flat Plans for Massively Parallel RDF Queries,” in Proc. IEEE 31st Int. Conf. Data Eng., pp. 771–782, 2015.
15. A. Schätzle, M. Przyjaciel-Zablocki, T. Hornung, and G. Lausen, “PigSPARQL: A SPARQL Query Processing Baseline for Big Data,” in Proc. 12th Int. Semantic Web Conf. (Posters Demonstrations Track), pp. 241–244, 2013.
16. A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen, “Sempala: Interactive SPARQL Query Processing on Hadoop,” in Proc. 13th Int. Semantic Web Conf., pp. 164–179, 2014.
17. A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, “S2RDF: RDF Querying with SPARQL on Spark,” in Proc. of VLDB Endowment, vol. 9, no. 10, pp. 804–815, 2016.
18. D. Graux, L. Jachiet, P. Genevès, and N. Layaïda, “SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark,” Springer, Cham, 2016, pp. 80–87.
19. M. Cossu, M. Färber, and G. Lausen, “PRoST: Distributed execution of SPARQL queries using mixed partitioning strategies,” in Proc. of the 21th International Conference on Extending Database Technology, pp. 469–472, 2018.
20. J. Dean, S. Ghemawat, and I. Google, “MapReduce: Simplified Data Processing on Large Clusters,” Sixth Symposium on Operating System Design and Implementation, vol. 51, no. 1, pp. 1–13, 2008.
21. M. Kornacker et al., “Impala: A Modern, Open-Source SQL Engine for Hadoop,” in Proceedings of the Conference on Innovative Data Systems Research (CIDR’15), 2015.
22. K. Wilkinson, “Jena Property Table Implementation,” in Proc. Int. Workshop Scalable Semantic Web Knowl. Base Syst., pp. 35–46, 2006.
23. D. J. Abadi, S. R. Madden, and K. Hollenbach, “Scalable Semantic Web Data Management Using Vertical Partitioning,” in Proc. 33rd Int. Conf. Very Large Data Bases, pp. 411–422, 2007.
24. M. Hassan and S. K. Bansal, “Data Partitioning Scheme for Efficient Distributed RDF Querying Using Apache Spark,” in Proceedings of the13th IEEE International Conference on Semantic Computing, pp. 24–31, 2019.
25. M. Hassan and S. K. Bansal, “S3QLRDF: Property Table Partitioning Scheme for Distributed SPARQL Querying of large-scale RDF data,” in 2020 IEEE International Conference on Smart Data Services (SMDS), 2020, pp. 133–140.
26. Y. Huai et al., “Major technical advancements in Apache Hive,” in Proc. ACM SIGMOD Int. Conf. Manag. Data, pp. 1235–1246, 2014.
27. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, “SW-Store: A vertically partitioned DBMS for semantic web data management,” VLDB J., vol. 18, no. 2, pp. 385–406, 2009.

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 .
}
```

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> .
}
```

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 .
}
```

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 .
}
```