Systematic Performance Analysis of Distributed SPARQL Query Answering Using Spark-SQL

Mohamed Ragab, Sadiq Eyvazov, Riccardo Tommasini and Sherif Sakr
Data Systems Group, Tartu University, Tartu, Estonia
E-mail: firstname.lastname@ut.ee

Abstract. Recently, a wide range of Web applications utilize vast RDF knowledge bases (e.g. DBPedia, Uniprot, and Probase), and use the SPARQL query language. The continuous growth of these knowledge bases led to the investigation of new paradigms and technologies for storing, accessing, and querying RDF data. In practice, modern big data systems like Apache Spark can handle large data repositories. However, their application in the Semantic Web context is still limited. One possible reason is that such frameworks are not tailored for dealing with graph data models like RDF. In this paper, we present a systematic evaluation of the performance of SparkSQL engine for processing SPARQL queries. We configured the experiments using three relevant RDF relational schemas, and two different storage backends, namely, Hive, and HDFS. In addition, we show the impact of using three different RDF-based partitioning techniques with our relational scenario. Moreover, we discuss the results of our experiments showing interesting insights about the impact of different configuration combinations.

Keywords: Large RDF Graphs, SPARQL, Apache Spark, Spark-SQL, RDF Relational Schema, RDF Partitioning

1. Introduction

The Linked Data initiative is fostering the adoption of semantic technologies like never before [1, 2]. Vast Resource Description Framework (RDF) datasets (e.g. DBPedia, Uniprot, and Probase) are now publicly available, and the challenges of storing, managing, and querying large RDF datasets are getting popular.

In this regards, the scalability of native triplestores like Apache Jena, RDF4J, and RDF-3X is bound by a centralized architecture. Thus, the Semantic Web community is investigating how to leverage big data processing frameworks like Apache Spark [3] to achieve better performance when processing large RDF datasets [4, 5].

In fact, despite big data frameworks are not tailored to perform native RDF processing, they were successfully used to build engines for large-scale relational data processing and several approaches exist for representing the RDF data as relations [6–8].

To the best of our knowledge, a systematic analysis of the performance of Big Data frameworks when answering SPARQL Protocol and RDF Query Language (SPARQL) queries is still missing. Our research work focuses on filling this gap. In particular, we focus on Apache Spark that, with the Spark SQL engine, is the de-facto standards for processing large datasets.

In the first phase of our work [9], we presented a systematic analysis of the performance of Spark-SQL on a centralized single-machine. In particular, we measured the execution time required to answer SPARQL queries. In our evaluation we considered: (i) alternative relational schemas for RDF, i.e., Single Statement Tables (ST), Vertical Tables (VT), and Property Tables (PT); (ii) various storage backends, i.e., PostgreSQL, Hive, and HDFS, and (iii) different data formats (e.g. CSV, Avro, Parquet, ORC).

In this paper, we present the second phase of our investigation. Our experiments include larger dataset
than before, in a distributed environment in presence of
data partitioning. In particular, we evaluate the impact
of three different RDF-based partitioning techniques
(i.e., Subject-based, Predicate-based, and Horizontal
partitioning) on our relational data. An additional con-	ributions of the current paper is a deeper and prescriptive
analysis of Spark SQL performance. Hence, in-
spired by the work in [10], we analyze the experiments
results in detail and provide a framework for deciding
the best configurations combinations of schema, par-
titioning, and storage for seeking better performance.
In particular, this paper applies existing techniques for
ranking experimental results, and discusses their pros
and cons alongside with their limitations. Last but not
least, the paper shows how to combine ranking criteria
(i.e. relational schemas, partitioning techniques, and
storage backends) to better investigate the trade-offs
that occur across these experimental dimensions.

The remainder of the paper is organized as follows:
Section 2 presents an overview of the required knowl-
edge to understand the content of the paper. Section 3
describes the benchmarking scenario of our study. Section 4
describes the experimental setup of our bench-
mark. While, section 5 presents the analysis methodology
followed to analyze the results. Section 6 discusses
these results and various insights regarding them. We
discuss the related work in Section 7 before we con-
clude the paper in Section 8.

2. Background

In this section, we present the necessary background
to understand the content of the paper. We assume that
the reader is familiar with RDF data model and the
SPARQL query language.

2.1. Spark & Spark-SQL

Apache Spark [3] is an in-memory distributed com-
puting framework for large scale data processing.
At the Spark’s core there are Resilient Distributed
Datasets (RDDs, i.e., immutable) distributed collection
of data elements.

Spark supports different storage backends for read-
ing and writing data. Those that are relevant for our
performance evaluation are Apache Hive, i.e., a data
warehouse built on top of Apache Hadoop for pro-
viding data query and analysis [11]: the Hadoop Dis-
tributed File System (HDFS). In particular, HDFS
supports the following file formats: (i) Comma Sepa-
rated Values (CSV), which is a readable and easy to de-
bug file format; (ii) Parquet [4] which stores the data in
a nested data structure and a flat columnar format that
supports compression; (iii) Avro [5] which contains data
serialized in a compact binary format and schema in
JSON format. (iv) Optimized Row Columnar (ORC)[6],
which provides a highly efficient way to store and pro-
cess Hive data.

Last but not least, DataFrames are a convenient
programming abstraction that adds to the flexibil-
ity of RDDs a specific named schema with typed
columns like in relational databases. Spark-SQL [12]
is a high-level library for processing DataFrames in
a relational manner. In particular, it allows querying
DataFrames using an SQL-like language, and it relies
on the Catalyst query optimizer

2.2. Relational RDF Schemas

Although triplestores can be used to efficiently store
and manage RDF data [13], some research works sug-
gest how to manage RDF data in relational stores [14].

In the following, we present three relational schemas
that are suitable for representing RDF data. For each
schema we give an example of data using Listing 1,
and we provide the respective SQL translation of the
SPARQL query in Listing 2.

Listing 1: RDF example in N-Triples. Prefixes are omitted.

```
@JOURNAL rdf:type :Journal ;
  dc:title "Journal 1 (1940)" ;
  dcterms:issued "1940" .

@ARTICLE rdf:type :Article ;
  dc:title "richer dwelling scrapped" ;
  dcterms:issued "2019" ;
  journal :Journal .
```

Listing 2: SPARQL Example against RDF graph in
Listing 1. Prefixes are omitted.

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

Apart from the SQL translation, we also provide a
Catalyst query optimizer that can be used to
execute the query. The query optimizer
allows to optimize the query plan
and provides insights on how
the query is executed. The
query optimizer
is implemented as a part
of the Spark SQL engine
and is available as a
module that can be
Downloaded from https://parquet.apache.org/
Downloaded from https://avro.apache.org/
Downloaded from https://orc.apache.org/
Downloaded from https://databricks.com/glossary/catalyst-optimizer
adopted. For instance, the major open-source triplestores, i.e., Apache Jena, RDF4J and Virtuoso, use the ST schema for storing RDF data. Figure 1 shows the ST schema representation of the sample in Listing 1 and the associated SQL translation for the SPARQL query in Listing 2. Vertically Partitioned Tables Schema requires to store RDF triples into tables of two columns (subject, object) for each unique property in the RDF dataset. VT schema was proposed to speed up the queries over RDF triple stores. Figure 2 shows the VT representation of the sample RDF graph shown in Listing 1 and the associated SQL translation for the SPARQL query in Listing 2. Property Tables Schema requires to cluster multiple RDF properties as n-ary table columns for the same subject to group entities that are similar in structure. PT schema works perfectly with highly structured data, but not with the poorly structured datasets, due to the high number of null values that it might incur. Moreover, due to its sparse tables representation, PT suffers from high storage overheads when a large number of predicates is present in the RDF data model. Figure 3 shows the relational flattened property tables of the RDF graph in Listing 1 and the associated SQL translation for the SPARQL query in Listing 2.

It is worth mentioning that there are other variants of the relational schemas mentioned above. In particular, the Wide Property Tables (WPT) schema and Extended Vertical Partitioning (ExtVP) are assumed to reside on the same partition. In our scenario, we applied Spark partitioning using the subject as a key with our different relational schema tables/Dataframes. Subject-Based Partitioning (SBP) requires to distribute triples to the various partitions according to the hash value computed for the subjects. As a result, all the triples that have the same subject are assumed to reside on the same partition. In our scenario, we applied spark partitioning using the subject as a key with our different relational schema tables/Dataframes. Predicate-Based Partitioning (PB) requires, similar to the SBP, to distribute triples to the various partitions based on the hash value computed for the predicate. As a result, all the triples that have the same predicate are assumed to reside on the same partition. In our scenario, we applied Spark partitioning using the predicate as a key with our different relational schema tables/Dataframes.

Also for partitioning techniques, it is worth mentioning that other approaches exist in the literature. However, the partitioning techniques presented above are suitable to work within the Spark-SQL framework. Indeed, techniques like Hierarchical Partitioning rely one the URIs structure, or are based on the k-way multi-level RDF partitioning strategy. These approaches may require some re-design to fit with our relational-based data processing scenario. Thus, we have selected them among the seven pure RDF-based partitioning techniques discussed in.

3. Benchmark Datasets & Queries

According to Jim Gray, a domain-specific benchmark must be Relevant, Portable, Scalable, and Simple. In our evaluation, we used SP2Bench (SPARQL Performance Benchmark) because it meets these criteria. In fact, it is also one of the most popular RDF benchmarks.

SP2Bench is Simple, as it is centered around the Computer Science DBLP scenario which is easy for researchers to understand. It is Scalable, because it comprises a data generator that enables the creation of arbitrarily large DBLP-like documents (in Notation-3 format). It is Portable w.r.t. our scenario, as it provides a set of SPARQL queries with their translations into SQL for each of the relational schemas we selected. These queries have different complexities, and a high diversity of features. Thus, SP2Bench is also a Relevant benchmark. Moreover, it has a reasonable low score of Structuredness, making it closer to the structure of real-world RDF datasets.

Since the design of ST and VT schemas is independent from the meaning of RDF, we have reused a
Fig. 1. Single Statement Table Schema and an associated SQL query sample. Prefixes are omitted.

```
<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>journal1</td>
<td>type</td>
<td>Journal</td>
</tr>
<tr>
<td>journal1</td>
<td>title</td>
<td>'Journal(1940)'</td>
</tr>
<tr>
<td>journal1</td>
<td>issued</td>
<td>1940</td>
</tr>
<tr>
<td>journal1</td>
<td>editor</td>
<td>Sharise_Heagy</td>
</tr>
<tr>
<td>article1</td>
<td>type</td>
<td>Article</td>
</tr>
</tbody>
</table>

SELECT T3.object AS year
FROM SingleTable T1, SingleTable T2, SingleTable T3
```

Fig. 2. Vertical Partitioned Tables Schema and an associated SQL query sample. Prefixes Omitted.

```
<table>
<thead>
<tr>
<th>Subject</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>journal1</td>
<td>Journal</td>
</tr>
<tr>
<td>article1</td>
<td>Article</td>
</tr>
</tbody>
</table>

SELECT T2.object AS year
FROM Title T1, Issued T2, Type T3
WHERE T2.subject=T2.subject AND T2.subject=T3.subject AND T1.object=http://sp2bench/Journal" AND T1.object=Journal1 (1940)"
```

Fig. 3. Property Tables Schema and an associated SQL query sample. Prefixes are omitted.

```
<table>
<thead>
<tr>
<th>ID</th>
<th>title</th>
<th>issue</th>
<th>editor</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>journal1</td>
<td>'Journal(1940)'</td>
<td>1940</td>
<td>Sharise_Heagy</td>
<td>Journal</td>
</tr>
</tbody>
</table>

SELECT Issued AS year
FROM Journal J
WHERE J.title=Journal1 (1940)"
```

Fig. 4. RDF partitioning techniques, (a) Horizontal Partitioning, (b) Subject-based Partitioning, (c) Predicate-based partitioning
similar PT schema inspired by the relational schema proposed by Schmidt et al. [3]. In their experiments, the SP²Bench RDF dataset contains nine different relational entities namely, Journal, Article, Book, Person, InProceeding, Proceeding, InCollection, PhDThesis, MasterThesis, and WWW documents. This schema is inspired by the original DBLP schema[5] that is generated by SP²Bench generator.

3.1. Queries

SP²Bench queries[6] cover a variety of SPARQL operators as well as various RDF access patterns. In our experiments, to be compliant with the Spark-SQL, we use the SQL translation of these SPARQL queries, which are provided for relational schemas translation, i.e., ST, VT, and PT. We have evaluated all of these 11 queries of type SELECT, except Q3 which is not applicable (NA) for the PT relational schema.

To give an indication of the query complexity, we looked at the following query features, i.e., number of joins, number of filters, and the number of projected variables. Table 1 summarizes these complexity measures for SP²Bench queries in SPARQL, and for the SQL-translations that are related to each RDF relational schema. For instance, we use the number of variable projections in the SQL statements as an indicator for the performance comparison between the data formats of the storage backends in terms of being row-oriented (e.g., Avro) or columnar-oriented (e.g., Parquet or ORC).

4. Experimental Setup

In this section, we describe our experimental environment, i.e., (i) we discuss how we configured our experimental hardware and software components; (ii) we describe how we prepared, partitioned, and stored the datasets, and (iii) we present the details of the experiment design.

Hardware and Software Configurations. Our experiments have been executed on a bare metal cluster of four machines with a CentOS-Linux V7 OS, running on a 32-AMD cores per node processors, and 128 GB of memory per node, alongside with a high speed 2 TB SSD drive as the data drive on each node. We used Spark V2.4 to fully support Spark-SQL capabilities. We used Hive V3.2.1. In particular, our Spark cluster is consisted of one master node and three worker machines, while Yarn is used as the resource manager, which in total uses 330 GB and 84 virtual processing cores.

Benchmark Datasets. Three datasets were generated using SP²Bench 100M, 250M, 500M triples in Notation3 (.n3) format. We have tested our experiments on these datasets to check the linearity of our results conformance. For the sake conciseness, we show in the paper only results related to 100M and 500M datasets. Nevertheless, all the results, including those for the intermediary dataset of 250M, are available in our GitHub repository[7].

Data Partitioning. In Section 2, we describe the partitioning techniques we selected, i.e., HP, SBP, and PBP. Partitioning impacts data distribution and, thus, Spark-SQL performance is affected, specially reading from data backends and data-joining operations. Therefore, when partitioning is required, the goal is to minimize data shuffling. One should select the technique that best suits the workload, i.e., the queries to run. Our mentioned partitioning techniques were originally designed for RDF partitioning. Hence, we defined their equivalent version for tabular RDF representation.

In Spark-SQL, Join operations are equi-joins, i.e., they require the join key to be the partitioning key. That means that data must be on the same node. Thus, we prepared the data in two phases. First, we use custom Spark partitioners for creating DataFrames that fulfill a certain partitioning technique. Depending on the partitioning techniques of choice (i.e, SBP, PBP, or HP), we used as partitioning keys respectively subject or predicate, or we used the horizontal approach. Then, we persisted the DataFrames on HDFS. We fixed the data partition block size on HDFS as the default block size on Spark (128MB). HDFS manages also the replication of these partitioned blocks according to a configurable replication factor (RF) (i.e. we used the default RF = 3).

Data Storage. In our experiments, we use two storage backends, i.e., HDFS and Hive (see Section 2). Additionally, for HDFS we used multiples file formats. We used Spark to convert the data from the N3 format generated by SP²Bench, into Avro, Parquet, and


[6] [https://datasystemsgrouput.github.io/SPARKSQLRDFBenchmarking/Results]
We used the same approach to load the data into the tables of the Apache Hive data warehouse using three created databases, one for each dataset size (100M, 250M, and 500M). Loading the data of the CSV files into the Hive data warehouse has been done in a little bit different way. In particular, to store data into Hive tables, it is a must to enable the support for Hive in the Spark session configuration using the enableHiveSupport function. Moreover, it is also important to give the Hive metastore URI using the Thrift URI protocol, also specified in the SparkSession configuration in addition to the warehouse location.

**Experiments Design.** We evaluated all the SP²Bench queries for all the combinations of schemas, backends/formats and partitioning techniques. For each configuration, we run the experiment five times (excluding the first cold-start run time, to avoid the warm-up bias, and computed an average of the other four run times). Figure 5 summarizes the experiments configurations, guiding the reader through the naming process in our further analysis results and plots, i.e.,

\{(Schema).{Partitioning_Technique}.{Storage_Backend}\}

For instance, (a.ii.4) corresponds to Single ST schema, SBP partitioning, and Parquet backend.

We used the Spark.time function by passing the spark.sql(query) query execution function as a parameter. The output of this function is the running time of
evaluating the SQL query into the Spark environment using the SparkSession interface.

5. Analysis Methodology

![Analysis Methodology Diagram](https://www.gartner.com/en/newsroom/press-releases/)

Fig. 6. Analysis Methodology

In this section, we describe the methodology that we follow for analyzing our results. We structured our analysis according to Figure 6, which tries to quantify the cost of decision making starting from different levels of analysis.

We advocate the need of a decision-making framework for making sense of performance of big data systems. This gets more crucial, especially when the solution space includes several different variables and unknown trade-offs, which is the standard case for big-data frameworks performance benchmarking, e.g., Spark in our scenario.

Before explaining how we structure each level of analysis, it is worth noticing that, the predictive analysis is out of the scope of this work. Predictive analysis typically leverage statistical models in order to answer questions about the future. In our research, we focus on systematically applying a post-hoc evaluation of the performance.

5.1. Descriptive Analysis

On the top level, descriptive analysis allows to answer factual questions. We extrapolate fine-grain insights, e.g., what is happening in the query evaluation level. In particular, we use descriptive analysis to identify which queries are long running, medium running, or short running according to their average running times. In this phase of analysis, we will also be able to observe general performance dimensions. For each query, we can observe which schema, partitioning technique, and storage backend are performing the best or the worst. However in this level of analysis, we are unable to decide which configuration combination (i.e., schema, partitioning, and storage backend) shows the best performance. Moreover, some of the descriptive results are contradicting in this level of analysis. For instance, we will show that in some queries the VT schema is the best performing choice. Whereas, for the same query with another partitioning technique, the PT schema performs better.

5.2. Diagnostic Analysis

Right below the descriptive analysis, there is diagnostic analysis allows answering why questions. In this level, we combine factual knowledge from the observed data with knowledge about the world to make sense of the results. We can enrich the descriptive analyses mentioned above with contextual information about the query complexity and the configuration. However, we are unable to investigate the trade-offs in terms of the dimensions affecting the performance. Thus, we advocate the need of better indicators that help investigating the impact of each dimension across all the queries.

5.3. Prescriptive Analysis

Last but not least, at bottom level there is prescriptive analysis which allows providing actionable insights for the analyst to decide. In practice, this means systematically investigating the impact of each dimension of the experiment, i.e., schema, partitioning, and storage, while discussing the trade-offs across these different dimensions to the extent of identifying an optimal solution.

In this regards, ranking criteria, e.g., the one proposed in [10] for partitioning, help giving a high-level view of the performance of a certain dimension across queries. Thus, we have extended the proposed ranking techniques to schemas and storage. The following equation shows a generalized ranking formula for ranking our relational schemas, partitioning techniques, and storage backends.

\[
RS_D = \sum_{r=1}^{t} \frac{O_D(r) \times (t - r)}{b(t - 1)}, 0 < RS_D \leq 1 \tag{1}
\]

In Equation 1, \(RS_D\) defines the Rank Score (RS) of the ranked dimension \(D\) (relational schema, partitioning technique, and storage backend). Such that, \(t\) represents the total number of the ranked dimension.
Using Equation 1 for ranking the relational schemas poses \( t = 3 \), as we have three different schemas in our paper. It will be the same (i.e. \( t = 3 \)) while applying the equation for ranking partitioning techniques, as they are also three. Whereas, while applying the equation for ranking the storage backends we pose \( t = 5 \), as we have five different backends/file formats. While \( b \) in the formula, represents the total number of query executions, as we have 11 query executions in our SP²Bench benchmark (i.e. \( b = 11 \)). Finally, \( O_d(r) \) denotes the total number of occurrences of a particular dimension \( D \) to come in the rank \( r \).

Applying the ranking criteria independently for each dimension supports a better explanations of the results. Nevertheless, we observed that, we are still unable to identify which configuration combination is the best performing, since the trade-offs between those dimensions are still not investigated. Therefore, we advocate for combining rankings towards for choosing the best performing configuration combination. To this extent, we tested three alternative techniques that aim at combining the ranking dimensions into one unified ranking criterion.

- The **Average** (AVG) criterion aims at combining the three dimensions rankings (\( R_f, R_p \) and \( R_s \)) by averaging them (Cf. equation 2). Such that, \( R_f, R_p \), and \( R_s \) are the rankings of storage backend, partitioning, and storage backends respectively.

\[
AVG = \frac{1}{3}(R_f + R_p + R_s)
\]  

(2)

- The **Weighted Average** (WAVG) criterion extends the Average assigning weights to each individual rank according to its impact in the experiments, i.e., we have 5 different storage backends, 3 partitioning techniques, and 3 relational schemas). (Cf. equation 3).

\[
WAVG = \frac{1}{3}(R_f \times 5 + R_p \times 3 + R_s \times 3)
\]  

(3)

- The **Ranking Triangle Area** (RTA) criterion leverages on a geometric interpretation of the trade-off of our experiments three dimensions. It looks at the triangle subsumed by each ranking criterion (\( R_f, R_p \), and \( R_s \)). The trade-offs ranking dimensions are presented by the triangle sides. The criterion aims at maximizing the area of this triangle. In other words, the bigger the area of this triangle, the better the performance of the three ranking dimensions all together. The ideal case is represented by the red triangle in Figure 7 which has the maximum ranking score of 1 (as, \( 0 < R_{SD} \leq 1 \) cf. equation 1) in all the vertices.

![Fig. 7. Triangle Area (Rta) combined Ranking criterion](image)

6. Experimental Results

In this section, we discuss the results of our experiments at the different levels of our analysis. We present the insights which each analysis criterion unveiled about the performance of the Spark-SQL query engine, using various relational RDF storage schema, alongside with many partitioning techniques, and on top of various storage backends.

6.1. Descriptive and Diagnostic Analysis

We start by discussing the descriptive analysis by showing the average query runtimes figures for the benchmark queries. We further follow these descriptive results with respective diagnosis analysis for answering the ‘why’ question for those results.

6.1.1. Query Performance Analysis

Figures 8 and 9 show the average execution times for running the SP²Bench queries for the 100M and 500M datasets, respectively. We can immediately observe that queries Q1, Q3, Q10, and Q11 are the least impactful queries (have the lowest running times). Thus, we call these queries short-running queries. On the other hand, queries Q2, Q4, and Q8 have the longest runtimes. The remaining queries Q5, Q6, Q7, and Q9 are medium-running queries. In the following, we focus our analysis on the longest running queries, as they may hide interesting insights about the approach limitations.

Query Q2 shows a low average execution time when using the ST schema or the VT schema. However, for
Fig. 8.: 100M dataset Average Query Execution Run-times (full-size figures are on the project github repository)
Fig. 9: 500M dataset average query execution run-times (full-size figures are on the project github repository)
the PT schema, it has much higher runtimes (at some cases 2X of runtime). This observation is confirmed in the 100M, 250M, and 500M datasets, and despite the partitioning technique of choice. Therefore, we can conclude that PT schema for Query Q2 is not the best option to choose. The previous observation is only valid with neglecting the bad performance of the CSV and Avro in the ST schema. However, it gives a clear answer of the impact of storage impact on our observations. Query Q4 has the highest latency for all the relational schema, for our different partitioning techniques, and for the different storage backends. Query Q8 immediately follows, as the second longest running query for the different partitioning techniques, and for the different storage backends. Interestingly, Q8 shows a significant enhancement when using the PT schema. We can observe that in the 100M dataset Figures[9] and it is even clearer by scaling up to 250M (i.e. figures are kept in the github repository), and 500M dataset, Figures[9].

Finally, we can also notice that, the ST relational schema has the worst impact on the majority of the queries. While, VT schema is mostly the best performing one, directly followed by the PT schema. Regarding the storage backends, we generally observe that the columnar file formats of HDFS (ORC, and Parquet) are the best performing, followed by Hive. Whereas, the row-oriented Avro and the CSV textual file format of HDFS are mostly the worst performing backends. Regarding to the partitioning impact, the SBP approach tends to considerably outperform its other opponents. Particularly, it directly outperforms HP, leaving the PBP technique in the worst rank. The partitioning impact observations are shown clearer in the next section ranking figures.

However, we cannot straightforwardly state that the VT schema is outperforming the PT schema. As it has been shown in Q8, the PT schema is obviously outperforming the VT schema. Similarly, we cannot state that Avro file format is always the worst or the second worst performing storage backend (however it is in the majority of queries), as Avro has been the best performing storage backend several times.

6.1.2. Results Diagnostic Analysis

Moving to the diagnostic analysis, we try to explain the previous observations by analyzing the query complexity (cf. Table 1) and using our knowledge about Spark and the experimental dimensions, i.e. relational schema, partitioning techniques, storage backends. We try to provide diagnostic analysis concerning these dimensions rather than investigating each single query result.

Regarding the relational schema comparison, we could observe that the ST schema is mostly the worst performing schema. Indeed, the ST schema is the one that requires the maximum number of self-joins (cf. Table 1 ST-SQL column). Moreover, ST schema single table is the largest table even after partitioning. Whereas, VT is mostly the best performing schema, specially when we scale up to higher datasets. The reason behind this is that VT tables tend to be smaller than other relational schema tables. Thus, Spark query joins have smaller intermediate results in the shuffle operations. In addition, VT tends to be more and more efficient with queries with small number of joins (i.e. BG triple patterns). While, the PT schema is yet a strong competitor to the VT schema, since it is the schema that requires the minimum number of joins while translating SPARQL into SQL. Indeed, PT in the single machine experiments achieved the highest ranks in the majority of query executions [9]. However, scaling up the experiment sizes, PT schema starts to incur larger intermediate results with higher shuffling costs that degrade its performance. Moreover, partitioning PT schema over Predicate (PBP) or Horizontally (HP) gives a negative effect on the PT schema performance, especially with SP2Bench query set that is highly 'subject'-oriented. We specify this with the reasoning about the impact of partitioning in our experiments.

Regarding the partitioning techniques comparison, in general, the Subject-based partitioning approach tends to considerably outperform its other partitioning opponents. Particularly, it directly outperforms the HP approach, leaving the PBP technique in the last/worst rank. The reason behind this is that most of the queries in SP2Bench are on shape of 'Star' or 'Snowflake' which are mostly oriented to the RDF subject as the joining key. Indeed, partitioning by subject allocates the triples with the same subject on the same machine reducing data shuffling to the minimum, and maximizing the level of parallelism by all workers. Whereas, this is not satisfied in the Horizontal-based approach, as it randomly splits the tables and only cares about distributing them in a balanced way as much as possible regardless grouping of the rows of the same subject on the same machine. Finally, the predicate-based approach presents the highest degree of shuffling while joins are run by Spark-SQL in most of the SP2Bench.

Therefore, Predicate-based technique are not recommended when evaluating 'subject'-oriented queries.
Moreover, Predicate-based partitioning is the most unbalanced load partitioning technique with the highest data skewness [18]. Since our SP²Bench RDF dataset has some predicates with few triples/table entries (i.e. subClassOf), while others have the most portions of the RDF graph (i.e. creator, type, and home page). Hence, this unbalanced nature leads to stragglers and inefficient join implementations in Spark-SQL.

Last but not least, let us consider the storage backends comparison. What we observed is that, columnar file formats of HDFS storage backend are outperforming the others. In particular, ORC is the best performing storage format followed by Parquet. While Hive directly follows them. Whereas, Avro and CSV file formats of HDFS are the worst-performing backends. Interestingly, we can observe that Avro outperforms the other storage formats in the PBP partitioning technique. The reason behind these results is that, most of the SP²Bench queries are with a small number of projections as shown in Table 1. Thus, columnar file storage backends perform better since they have to scan only a subset of the columns with filtering out unnecessary columns for the query [23]. Hive is consistently following them. On the other side, the textual uncompressed CSV and the row-oriented Avro file formats are shown to have the lowest performing storage options, respectively.

6.2. Prescriptive Analysis

In the following, we attempt to make the performance analysis prescriptive, i.e., we aim at identifying what are the optimal configuration combinations to use for SP²Bench, as our RDF benchmarking scenario.

To this extent, we use alternative ranking criteria (see Section 5 cf. Equation 1). We start by showing separate ranking analysis results for the experiment dimensions. Then, we discuss the results of combined ranking criteria. We finally discuss which ranking criterion is the most relevant to choose the optimal performing configurations, along side with showing a table of the best and the worst configuration combinations for the queries.

Notably, we keep all the intermediate ranking results/tables from which we calculated these final scores of the relational schemas, partitioning techniques as well as the storage backends comparison in our mentioned GitHub repository of this project, and its web-page[^1].

6.2.1. Relational-Schemas Ranking Analysis

Figure [10] shows how many times a particular relational schema achieves the highest or the lowest ranking scores, respectively, considering the results of all experiments. Specifically, schema ranking scores for the 100M datasets (Figures [10] (a), (c), and (e)), and 500M triples dataset (Figures [10] (b), (d), and (f)) for different storage backends respectively, and for our three partitioning techniques (HP, SBP, and PBP). For these graphs, we indicate a particular relational schema outperforms others when it has a higher ranking score, i.e., the higher ranking score, the better performance.

For the 100M dataset figures, we observe that the ST schema is always the worst performing one with 100% when HP and SBP are chosen as partitioning techniques. However, for the PBP partitioning, the ST schema has the second highest scores after the VT schema with 60%. On the other side, the VT schema has the highest ranking scores by 100% in the ranking scores of the relational schemas. The PT schema is falling between the VT schema and the ST schema by more than 73%. Scaling up to the 250M, and 500M triples dataset, our observations are confirmed.

6.2.2. Partitioning Techniques Ranking Analysis

Figure [11] shows our different partitioning techniques (i.e. Horizontal, Subject-based, and Predicate-based Partitioning) ranking scores for the 100M (Figures [11] (a), (c), and (e)), 500M (Figures [11] (b), (d), and (f)) triples datasets for different relational schema ST, VT, and PT respectively. Also for these graphs the higher rank score the better it is. Thus, we indicate a particular partitioning technique outperforms other techniques when it achieves higher ranking score amongst them. Notably, when a partitioning technique score column is missing, this indicates that its rank score is zero (i.e. it always comes at the last rank [3rd rank in our case]).

In the 100M dataset figures, the Subject-based partitioning is the best performing approach. Indeed, it has the highest ranking scores with more than 93% of the ranking times. Whereas, Predicate-based partitioning performs as the worst technique with roughly the same ratio (i.e. 93%). On the other hand, the performance of Horizontal partitioning lies somewhere between the

[^1]: https://datasystemsgroup.github.io/
[^2]: SPARQLRDFBenchmarking
Fig. 10. Relational Schemas Ranking Scores for 100M and 500M Triples datasets (Reading Key: the higher is the better).
Fig. 11. Partitioning Techniques Ranking Scores for 100M and 500M Triples datasets (Reading Key: the higher is the better).
other two techniques. However, HP outperforms the SBP having a higher rank with the PT schema and for Avro file format.

Considering the 500M triples dataset, we still observe the same pattern of performance. That is, the Subject-based still outperforms the other techniques with more than 75% of the ranking times. The Predicate-based partitioning is still performing the worst in the majority of ranking times with more than 46%. However, it outperformed the other techniques achieving the highest rank by three times with the ST schema and for CSV file format, and with the VT schema for CSV and Avro file formats.

6.2.3. Storage Backends Ranking Analysis

Last but not least, we investigate how different storage backends impact the performance in our experiments.

Figure 12 shows how many times a particular storage backend achieves the best or the lowest performance. The figure presents the results of all experiments, for different relational schema ST, VT, and PT respectively, and for our three partitioning techniques (HP, SB, and PB). Specifically, ranking scores for the 100M datasets are on the left, i.e., sub-figures (a), (c), and (e); ranking scores for the 500M triples dataset are on the right, i.e., sub-figures (b), (d), and (f)).

The reading key is still the higher the rank the better it is. Thus, we indicate a particular storage backend outperforms others when it achieves higher ranking score amongst them. Similarly, when a storage format score column is missing, this indicates its rank score is zero (i.e., it always comes at the last rank [5th rank in our case]).

For both the 100M and 500M triples datasets with the ST relational schema and HP and SB partitioning techniques, we observe that HDFS ORC is the best performing backend with 100%. Hive is immediately following, and then Parquet by 100% of these ranking cases. On the other hand, HDFS CSV and Avro file formats are respectively the worst performing on HDFS with 100% of the mentioned cases. The 500M ST schema dataset scores in the PBP have the same previous ranking results for the storage backends. However, for the 100M ST schema, and in the PBP partitioning technique, Avro is the second best-performing storage format coming after ORC.

While for 100M and 500M triples datasets with the VT relational schema and HP and SBP partitioning techniques, we can still observe that ORC and Parquet are sharing the best performing backend rank with 50% for each of them. Hive directly follows them in the third best rank with 100%. HDFS CSV and Avro file formats respectively keep performing the worst, having the lowest rank scores with 100% of these mentioned cases. However, for the PBP partitioning in 100M with the VT schema, Avro is the best performing backend, followed by CSV. For the 500M with the VT schema, and for PBP technique, ORC and Parquet respectively still in the best ranks, but Avro, interestingly, outperforms Hive and CSV.

Looking at the results of the 100M and 500M triples datasets with the PT schema and for HP and SBP, we can find that, the HDFS ORC and Parquet are still sharing the best performing backend with 50% for each of them in this ranking group, immediately followed by Hive as in the third best rank with 100%. CSV and Avro are respectively the worst performing storage formats with 100% of this mentioned ranking group. However, for the PBP partitioning of both the 100M and 500M datasets, Avro is significantly shown to be the best performing backend followed by ORC with 100% of the cases.

6.2.4. Combined Ranking Analysis

Table 2 shows all our possible different configuration combinations as shown from Figure 3. For instance the (a.i.1) representing the ST schema, Horizontally partitioned, and stored in Avro storage backend. From the table, we can see that we have different 45 possible configuration combinations. The next three columns Rf, Rp, and Rs include ranking score values which are calculated for storage formats, partitioning techniques, and relational schemas, respectively. The colored cells indicate the top 3 best-performing configurations for each column.

Looking at Table 2, we observe that ranking over one of the dimensions and ignoring the others ends up with selecting different configuration. For instance, ranking over Rf, i.e., storage backend/format; Rp, i.e., partitioning technique, or Rs, i.e., the relational schema, end up selecting different combinations of Schema, Partitioning and Storage backends.

Since focusing on one ranking at time leads to contradicting results, as shown in Table 3 we opt for a combined ranking criterion. In Section 5, we presented three i.e., Rta, WAvg and AVG, that are shown in Table 4.

In particular, we use Table 4 to assess each criterion by checking its ranking results (i.e., top results/configurations) against the actual queries results with
Fig. 12. Storage Backends Ranking Scores for 100M and 500M Triples datasets (Reading Key: the higher is the better).
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Measure</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100M</td>
<td>Rf</td>
<td>0.27</td>
<td>0.23</td>
<td>0.27</td>
<td>0.73</td>
<td>0.09</td>
<td>0.23</td>
<td>0.21</td>
<td>0.45</td>
<td>0.24</td>
<td>0.64</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>Rp</td>
<td>0.45</td>
<td>0.28</td>
<td>0.32</td>
<td>0.77</td>
<td>0.09</td>
<td>0.32</td>
<td>0.36</td>
<td>0.45</td>
<td>0.24</td>
<td>0.64</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>Rs</td>
<td>0.23</td>
<td>0.10</td>
<td>0.27</td>
<td>0.82</td>
<td>0.01</td>
<td>0.32</td>
<td>0.30</td>
<td>0.55</td>
<td>0.24</td>
<td>0.65</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>Rta</td>
<td>1.13</td>
<td>0.45</td>
<td>0.19</td>
<td>0.57</td>
<td>0.11</td>
<td>0.71</td>
<td>0.64</td>
<td>1.94</td>
<td>0.94</td>
<td>2.32</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>WAvg</td>
<td>0.32</td>
<td>0.23</td>
<td>0.21</td>
<td>0.64</td>
<td>0.11</td>
<td>0.71</td>
<td>0.71</td>
<td>1.17</td>
<td>0.32</td>
<td>0.55</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>AVG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500M</td>
<td>Rf</td>
<td>0.41</td>
<td>0.32</td>
<td>0.09</td>
<td>1.75</td>
<td>0.01</td>
<td>0.45</td>
<td>0.24</td>
<td>0.59</td>
<td>0.27</td>
<td>0.94</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>Rp</td>
<td>0.68</td>
<td>0.11</td>
<td>0.14</td>
<td>0.77</td>
<td>0.21</td>
<td>0.14</td>
<td>0.47</td>
<td>0.77</td>
<td>0.27</td>
<td>0.94</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>Rs</td>
<td>0.14</td>
<td>0.36</td>
<td>0.36</td>
<td>0.82</td>
<td>0.11</td>
<td>0.47</td>
<td>0.77</td>
<td>1.17</td>
<td>0.36</td>
<td>0.59</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>Rta</td>
<td>1.50</td>
<td>0.45</td>
<td>0.01</td>
<td>1.69</td>
<td>0.01</td>
<td>0.47</td>
<td>0.77</td>
<td>1.66</td>
<td>0.36</td>
<td>0.59</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>WAvg</td>
<td>0.41</td>
<td>0.11</td>
<td>0.36</td>
<td>1.69</td>
<td>0.01</td>
<td>0.47</td>
<td>0.77</td>
<td>1.66</td>
<td>0.36</td>
<td>0.59</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>AVG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Configuration Ranking Criteria for 100M, and 500M triples datasets
these configurations. To achieve that, we pick up the top three configurations (coloured in Table 2) with the highest rank score of the different ranking criteria columns (i.e. Rf, Rp, Rs, Rta, Avg, and WAvg). Afterwards, we list the ranking of these configurations for each query (Q1,...Q11) by calculating the performance rank position of this configuration in this query. For example, the configuration combination (a.iii.3) has the 29th performance rank position for Q1, 20th for Q2,...

The coloured cells in Table 4 indicate the top 15 out of 45 ranking positions for the selected top configurations for the queries. In addition, we have calculated the average of the number of times a specific configuration combination to be in the top 15 ranking positions for each criteria. For instance, the Rf ranking criteria with the first top selected configuration (a.iii.3) occurred only once to be 11th ranking position (i.e. in the first 15 actual query ranking positions) only for one query (Q11). While, the second selected configuration (a.ii.3) achieves that goal (to be in the first 15 ranks) by 4 times for queries Q2, Q4, Q9, and Q10 respectively. While, the (c.iii.3) configuration achieves that goal by 5 times for queries Q3, Q4, Q5, Q6, Q8. Therefore, the average of the Rf criteria to be relevant for choosing the best configuration is 3 (i.e we neglect the fractions) as shown in the Table. The rest of ranking criteria, and combined ranking techniques are calculated in a similar way.

The next step is to calculate the accuracy of each criteria using the formula in equation 4 in order to see which ranking criteria should be used to determine the best configuration combination to choose.

\[ Acc(cr) = \sum_{i=1}^{3} \frac{N(i)}{33}, cri \in \{Rf, Rp, Rs, ...\} \]  (4)

In the formula 4, i indicates certain configuration and N(i) is the number of times a certain configuration occurred to be in a query ranking position less than the 15 rank, for each ranking criteria (cr). Applying this formula for each criteria, we observed that, for the 100M, results are as follows, 30%, 33%, 39%, 64%, 55% and 64% for Rf, Rp, Rs, AVG, WAvg and Rta ranking criteria, accordingly. While for 500M, the results of applying formula 4 are 48%, 55%, 55%, 64%, 67%, 76% for Rf, Rp, Rs, AVG, WAvg and Rta, accordingly.

From the 100M and 500M results, it appears that the triangle area (Rta) criterion is the most accurate for measuring the performance of configurations in our SP²Bench query workloads scenario. The average (AVG) and the weighted average (WAVG) follow right after. Notably, we used the 100M triples dataset table to describe how ranking calculations are done. We omit tables of the other datasets for sake of conciseness. However, data and analysis can be found in the repository web page.

Finally, Table 5 summarizes the experiments analysis, highlighting the best and worst combinations of schema, partitioning techniques, and storage backends for each query in the SP²Bench workload.

7. Related Work

Several related experimental evaluation and comparisons of the relational-based evaluation of SPARQL queries over RDF databases have been presented in the literature [8, 14]. For example, Schmidt et.al. [14] performed an experimental comparison between existing RDF storage approaches using the SP²Bench performance suite, and the pure relational models of RDF data implementations namely, Single Triples relation, Flattened Tables of clustered properties relation, and Vertical partitioning Relations. In particular, they compared the native RDF scenario using Sesame SPARQL engine (known currently as RDF4J) that is relied on a native RDF store using SP²Bench dataset, with a pure translation of the same SP²Bench scenario into pure relational database technologies. Another experimental comparison of the single triples table and vertically partitioned relational schemes was conducted by Alexaki et al. [24] in which the additional costs of predicate table unions in the vertical partitioned tables scenario are clearly shown. This experiment was also similar to the ones performed by Abadi et al. [15], followed by Sidirourgos et al. [25] who used the Barton library catalog data scenario to evaluate a similar comparison between the Single Triples schema and the Vertical schema. On another
side, Owens et al. performed benchmarking experiments for comparing different RDF stores (e.g., Allegrograph, BigOWLIM, and RDBMS benchmarks (e.g., The Transaction Processing Performing Council family (TPC-C) benchmark). This work is focused on a pure detailed RDF stores comparison using SPARQL beyond any relational schemes implementations or comparisons. To the best of our knowledge, our benchmarking study (that we are extending here in this paper but in a distributed deployment), was the first that considers evaluating and comparing various relational-based schemes for processing RDF queries on top of the big data processing framework, Spark, and evaluating different backend storage techniques.

Moreover, Owens et al. performed benchmarking experiments for comparing different RDF stores (e.g., Allegrograph, BigOWLIM, and RDBMS benchmarks (e.g., The Transaction Processing Performing Council family (TPC-C) benchmark). This work is focused on a pure detailed RDF stores comparison using SPARQL beyond any relational schemes implementations or comparisons. To the best of our knowledge, our benchmarking study (that we are extending here in this paper but in a distributed deployment), was the first that considers evaluating and comparing various relational-based schemes for processing RDF queries on top of the big data processing framework, Spark, and evaluating different backend storage techniques.

![Table 4](https://example.com/table4.png)

**Table 4**

100M Triples Dataset Ranking Criteria Comparison

<table>
<thead>
<tr>
<th>Rp</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>Q11</th>
<th>Ranking&lt;15</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i.3</td>
<td>29</td>
<td>20</td>
<td>32</td>
<td>26</td>
<td>27</td>
<td>30</td>
<td>18</td>
<td>41</td>
<td>20</td>
<td>22</td>
<td>11</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>a.i.3</td>
<td>26</td>
<td>14</td>
<td>18</td>
<td>1</td>
<td>17</td>
<td>25</td>
<td>17</td>
<td>17</td>
<td>5</td>
<td>9</td>
<td>16</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>c.i.3</td>
<td>15</td>
<td>15</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>31</td>
<td>3</td>
<td>31</td>
<td>27</td>
<td>26</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rf</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>Q11</th>
<th>Ranking&lt;15</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i.3</td>
<td>26</td>
<td>14</td>
<td>18</td>
<td>1</td>
<td>17</td>
<td>25</td>
<td>17</td>
<td>17</td>
<td>5</td>
<td>9</td>
<td>16</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>a.i.5</td>
<td>27</td>
<td>24</td>
<td>26</td>
<td>2</td>
<td>13</td>
<td>27</td>
<td>16</td>
<td>24</td>
<td>7</td>
<td>14</td>
<td>19</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>a.i.4</td>
<td>23</td>
<td>23</td>
<td>24</td>
<td>4</td>
<td>24</td>
<td>26</td>
<td>19</td>
<td>23</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rs</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>Q11</th>
<th>Ranking&lt;15</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.ii.2</td>
<td>5</td>
<td>14</td>
<td>29</td>
<td>32</td>
<td>24</td>
<td>19</td>
<td>11</td>
<td>19</td>
<td>5</td>
<td>25</td>
<td>17</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>b.iii.1</td>
<td>18</td>
<td>17</td>
<td>17</td>
<td>41</td>
<td>9</td>
<td>18</td>
<td>12</td>
<td>30</td>
<td>14</td>
<td>12</td>
<td>9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>b.iii.4</td>
<td>12</td>
<td>6</td>
<td>23</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>14</td>
<td>36</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AVG</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>Q11</th>
<th>Ranking&lt;15</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.i.4</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>27</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>18</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>a.i.3</td>
<td>26</td>
<td>14</td>
<td>18</td>
<td>1</td>
<td>17</td>
<td>25</td>
<td>17</td>
<td>17</td>
<td>5</td>
<td>9</td>
<td>16</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>b.i.5</td>
<td>16</td>
<td>2</td>
<td>10</td>
<td>28</td>
<td>21</td>
<td>14</td>
<td>7</td>
<td>16</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WAvg</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>Q11</th>
<th>Ranking&lt;15</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i.3</td>
<td>26</td>
<td>14</td>
<td>18</td>
<td>1</td>
<td>17</td>
<td>25</td>
<td>17</td>
<td>17</td>
<td>5</td>
<td>9</td>
<td>16</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>b.i.4</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>27</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>18</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>c.i.3</td>
<td>15</td>
<td>15</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>31</td>
<td>3</td>
<td>31</td>
<td>27</td>
<td>26</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rta</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>Q11</th>
<th>Ranking&lt;15</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.i.4</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>27</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>18</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>b.i.5</td>
<td>16</td>
<td>2</td>
<td>10</td>
<td>28</td>
<td>14</td>
<td>14</td>
<td>7</td>
<td>16</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>a.i.3</td>
<td>26</td>
<td>14</td>
<td>18</td>
<td>1</td>
<td>17</td>
<td>25</td>
<td>17</td>
<td>17</td>
<td>5</td>
<td>9</td>
<td>16</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Table 5

**Table 5**

Best and Worst Configurations for Running SP2Bench Queries

suggest, using different RDF benchmarks (e.g., LUBM and RDBMS benchmarks (e.g., The Transaction Processing Performing Council family (TPC-C) benchmark). This work is focused on a pure detailed RDF stores comparison using SPARQL beyond any relational schemes implementations or comparisons. To the best of our knowledge, our benchmarking study (that we are extending here in this paper but in a distributed deployment), was the first that considers evaluating and comparing various relational-based schemes for processing RDF queries on top of the big data processing framework, Spark, and evaluating different backend storage techniques.

13 [https://franz.com/agraph/allegrograph3.3/](https://franz.com/agraph/allegrograph3.3/)
A parallel and recent similar research to this work was conducted by Victor Anthony et al. [27]. Authors there performed similar experiments to evaluate also the performance of Spark-SQL engine querying four relational-based RDF schemes, namely a single Triples Table (TT), Vertical Partitioned Table (VP), Domain-dependent schema (DDS), and (differently from our paper) the Wide Property Tables schema (WPT). They have shown that the WPT relational schema is superior to the other relational opponent approaches. They have only evaluated the Hive with Parquet storage backend, while as we have mentioned in this paper, we are evaluating other storage alternative backends of Spark. Moreover, partitioning in that paper is made by Spark, partitioning on the Subject only, while in this paper, we evaluate three different partitioning strategies logically and physically (i.e. done also by Spark), namely, Subject-based, Predicate-based, and, Horizontal partitioning. In addition, this work evaluates only the micro-benchmarking level of Spark-SQL system.

8. Conclusion

Apache Spark is a prominent big data framework that offers a high-level SQL interface, Spark-SQL, optimized by means of the Catalyst query optimizer. In this paper, we perform a systematic evaluation for the performance of the Spark-SQL query engine for answering SPARQL queries over different relational encoding for RDF datasets on a distributed setup. In particular, we studied the performance of Spark-SQL using two different storage backends, namely, Hive and HDFS. For HDFS we compared four different data formats, namely, CSV, ORC, Avro, and Parquet. We used SP-Bench to generate our experimental RDF datasets. We translated the benchmark queries into SQL, storing the RDF data using Spark’s DataFrame abstraction. To this extent, we evaluated three different approaches for RDF relational storage, i.e., Single Triples Table schema, Vertically Partitioned Tables schema, and Property Tables schema. We show also the impact of partitioning the mentioned relational schemas with three different partitioning techniques, namely Horizontal-based, Subject-based, and Predicate-based partitioning, which applied on our five mentioned storage formats.

As a future extension of this work, we aim to conduct our benchmarking study with other benchmarking such as benchmarks of WatDiv, and the state-of-the-art LDBC with different types of query shapes and complexities.

References


