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) and 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

*Corresponding author. E-mail: firstname.lastname@ut.ee.

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 contributions of the current paper is a deeper and prescriptive analysis of Spark SQL performance. Hence, inspired by the work in [10], we analyze the experiments results in detail and provide a framework for deciding the best configurations combinations of schema, partitioning, 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 knowledge to understand the content of the paper. Section 3 describes the benchmarking scenario of our study. Section 4 describes the experimental setup of our benchmark. 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 conclude 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 computing 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 reading 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 providing data query and analysis [11]; the **Hadoop Distributed File System** (HDFS). In particular, HDFS supports the following file formats: (i) *Comma Separated Values (CSV)*, which is a readable and easy to debug file format; (ii) *Parquet*[1], which stores the data in a nested data structure and a flat columnar format that supports compression; (iii) *Avro*[2], which contains data serialized in a compact binary format and schema in *JSON* format. (iv) *Optimized Row Columnar (ORC* [3]*)*, which provides a highly efficient way to store and process *Hive* data.

Last but not least, DataFrames are a convenient programming abstraction that adds to the flexibility 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[4].

2.2. Relational RDF Schemas

Although triplestores can be used to efficiently store and manage RDF data [13] , some research works suggest 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.

```
:Journal1   rdf:type :Journal ;
            dc:title "Journal 1 (1940)" ;
            dcterms:issued  "1940" .
:Article1   rdf:type :Article ;
            dc:title "richer dwelling scrapped" ;
            dcterms:issued  "2019" ;
            :journal  :Journal1 .
```

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

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

Listing 2: SPARQL Example against RDF graph in Listing 1.1. Prefixes are omitted.

Single Statement Table Schema requires to store RDF triples in a single table with three columns that represent the three components of the RDF triple, i.e., *subject*, *predicate*, and *object*. ST schema is widely

[1] https://parquet.apache.org/
[2] https://avro.apache.org/
[3] https://orc.apache.org/
[4] https://databricks.com/glossary/catalyst-optimizer

adopted [15, 16]. 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 [15]. 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 [14], 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 [7]. 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 [17] and *Extended Vertical Partitioning* (*ExtVP*) [18]. The former uses a unified table for the entire properties in the RDF graph; the latter is inspired by the *Semi-Join* reductions of the possible VP tables join correlations that can occur among the SPARQL query triple patterns. However, we opt to use the most *three* common RDF relational schemas (i.e. ST, VT, and PT).

2.3. RDF Data Partitioning

Last but not least, RDF data partitioning is another critical choice in our scenario. We have selected three partitioning techniques for RDF data that are suitable for our experiments on SparkSQL (cf. Figure 4).

Horizontal-Based Partitioning (HP) requires to partition the data evenly on the number of machines in the cluster. In particular, it divides the relational tables, i.e., in *n* equivalent chunks where *n* is number of machines in the cluster.

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 (PBP) 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 [7, 10]. 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 [19]. 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 [10].

3. Benchmark Datasets & Queries

According to *Jim Gray* [20], a domain-specific benchmark must be *Relevant*, *Portable*, *Scalable*, and *Simple*. In our evaluation, we used $\texttt{SP}^2\texttt{Bench}$ (SPARQL Performance Benchmark) [21] because it meets these criteria. In fact, it is also one of the most popular RDF benchmarks.

SP^{2}Bench 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 [22]. Thus, SP^{2}Bench 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 [22].

Since the design of ST and VT schemas is independent from the meaning of RDF, we have reused a

Subject	Predicate	Object
journal1	type	Journal
journal1	title	'Journal(1940)'
journal1	issued	1940
journal1	editor	Sharise_Heagy'
article1	type	Article
article1	title	richer dwelling scrapped'
…	…	…

```
SELECT T3.object AS year
FROM SingleTable T1, SingleTable T2, SingleTable T3
WHERE T2.subject=T2.subject AND
T2.subject=T3.subject AND
T1.object='http://.../sp2bench/Journal" AND
T2.predicate='http://purl.org/dc/elements/1.1/title' AND
T2.object='Journal 1 (1940)' AND
T3.predicate='http://purl.org/dc/terms/issued'
```

Fig. 1. Single Statement Table Schema and an associated SQL query sample. Prefixes are omitted.

type		title		issued	
Subject	Object	Subject	Object	Subject	Object
journal1	Journal	journal1	'Journal(1940)'	journal1	1940
article1	Article	article1	richer dwelling	…	…
…	…	…	…	…	…

```
SELECT T2.object AS year
FROM Title T1, Issued T2, Type T3
WHERE T2.subject=T2.subject AND
T2.subject=T3.subject AND
T3.object='http://.../sp2bench/Journal" AND
T1.object='Journal 1 (1940)'
```

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

Journal					Article			
ID	title	issue	editor	type	ID	title	jorunal	type
journal1	'Journal(1940)'	1940	Sharise_Heag	Journal	article1	'Journal(1940)'	journal1	Article
…	…	…	…	…	…	…	…	…

```
SELECT J.issued AS year
FROM Journal J
WHERE J.title='Journal 1 (1940)'
```

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

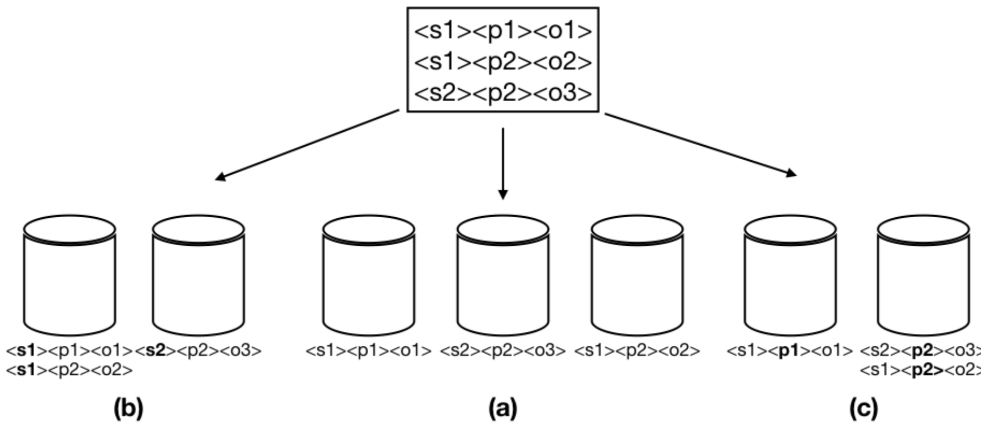


Fig. 4. RDF partitioning techniques, (a) Horizontal Partitioning, (b) Subject-based Partitioning, (c) Predicate-based partitioning

similar PT schema inspired by the relational schema proposed by Schmidt et al. [14]. In their experiments, the SP^{2}Bench RDF dataset contains *nine* different relational entities namely, *Journal*, *Article*, *Book*, *Person*, *InProceeding*, *Proceeding*, *InCollecion*, *PhDThesis*, *MasterThesis*, and *WWW* documents. This schema is inspired by the original *DBLP* schema[5] that is generated by SP^{2}Bench generator.

3.1. Queries

SP^{2}Bench 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[7]. We have evaluated all of these 11 queries of type *SELECT*, except $Q9$ 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 SP^{2}Bench 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 SP^{2}Bench $100M$, $250M$, $500M$ 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[8].

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 fulfil 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 SP^{2}Bench. into Avro, Parquet, and

[5]DBLP-like RDF data produced by the SP^{2}Bench http://dbis.informatik.uni-freiburg.de/forschung/projekte/SP2B/

[6]http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B/queries.php

[7]http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B/translations.html

[8]https://datasystemsgrouput.github.io/SPARKSQLRDFBenchmarking/Results

	SPARQL			ST-SQL		VT-SQL		PT-SQL	
	#Joins	Filters	#Proj	#Joins	#Sel	#Joins	#Sel	#Joins	#Sel
Q1	3	0	1	8	6	5	2	2	2
Q2	8	0	10	28	9	19	0	9	5
Q3	1	1	1	5	2	3	2	2	2
Q4	7	1	2	19	16	11	3	8	3
Q5	5	1	2	16	11	9	3	7	3
Q6	8	3	2	26	13	15	4	6	3
Q7	12	2	1	26	15	16	5	2	1
Q8	10	2	1	23	13	13	7	9	4
Q9	3	0	1	11	4	5	3	n/a	n/a
Q10	0	0	2	3	1	2	2	5	2
Q11	0	0	1	2	1	1	0	2	0

Table 1

SP^2Bench Queries Complexity Analysis, number of joins, and number of projections/selections for SPARQL query and our three considered RDF relational schemes (ST, VT, and PT).

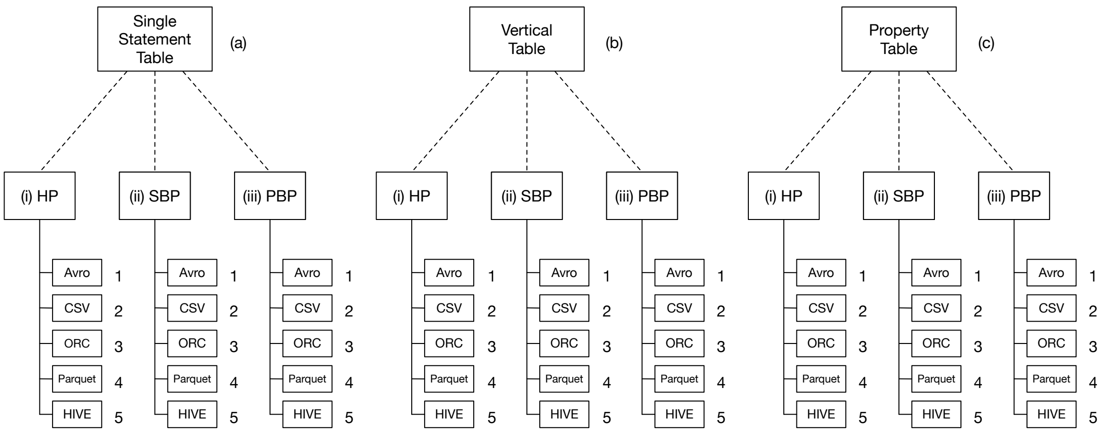


Fig. 5. Experiments Architecture.

ORC. We used Spark for this conversion because of its ability to efficiently handle large files in memory. This feature is required when converting graph data. Moreover, Spark supports reading and writing different file formats from and into HDFS.

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 SP^2Bench 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

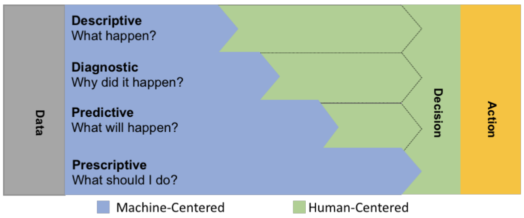


Fig. 6. Analysis Methodology

In this section, we describe the methodology that we follow for analyzing our results. We structured our analysis according to Figure 6[9], 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 level 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.

$$RS_D = \sum_{r=1}^{t} \frac{O_D(r) * (t-r)}{b(t-1)}, 0 < RS_D \leqslant 1 \qquad (1)$$

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.

[9]https://www.gartner.com/en/newsroom/press-releases/2014-10-21-gartner-says-advanced-analytics-is-a-top-business-priority

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 SP^{2}Bench 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 (Rf, Rp and Rs) by averaging them (Cf. equation 2). Such that, Rf, Rp, and Rs are the rankings of storage backend, partitioning, and storage backends respectively.

$$AVG = \frac{1}{3}(R_f + R_p + R_s) \qquad (2)$$

- 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).

$$WAvg = \frac{1}{3}(R_f * 5 + R_p * 3 + R_s * 3) \qquad (3)$$

- 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 (Rf, Rp, and Rs). 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 < RSD <=1 cf. equation 1) in all the vertices.

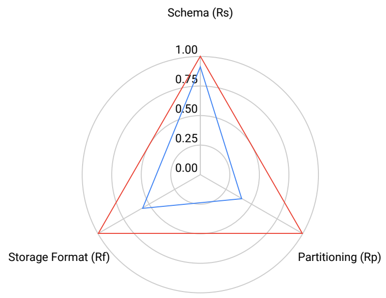


Fig. 7. Triangle Area (Rta) combined Ranking criterion

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.

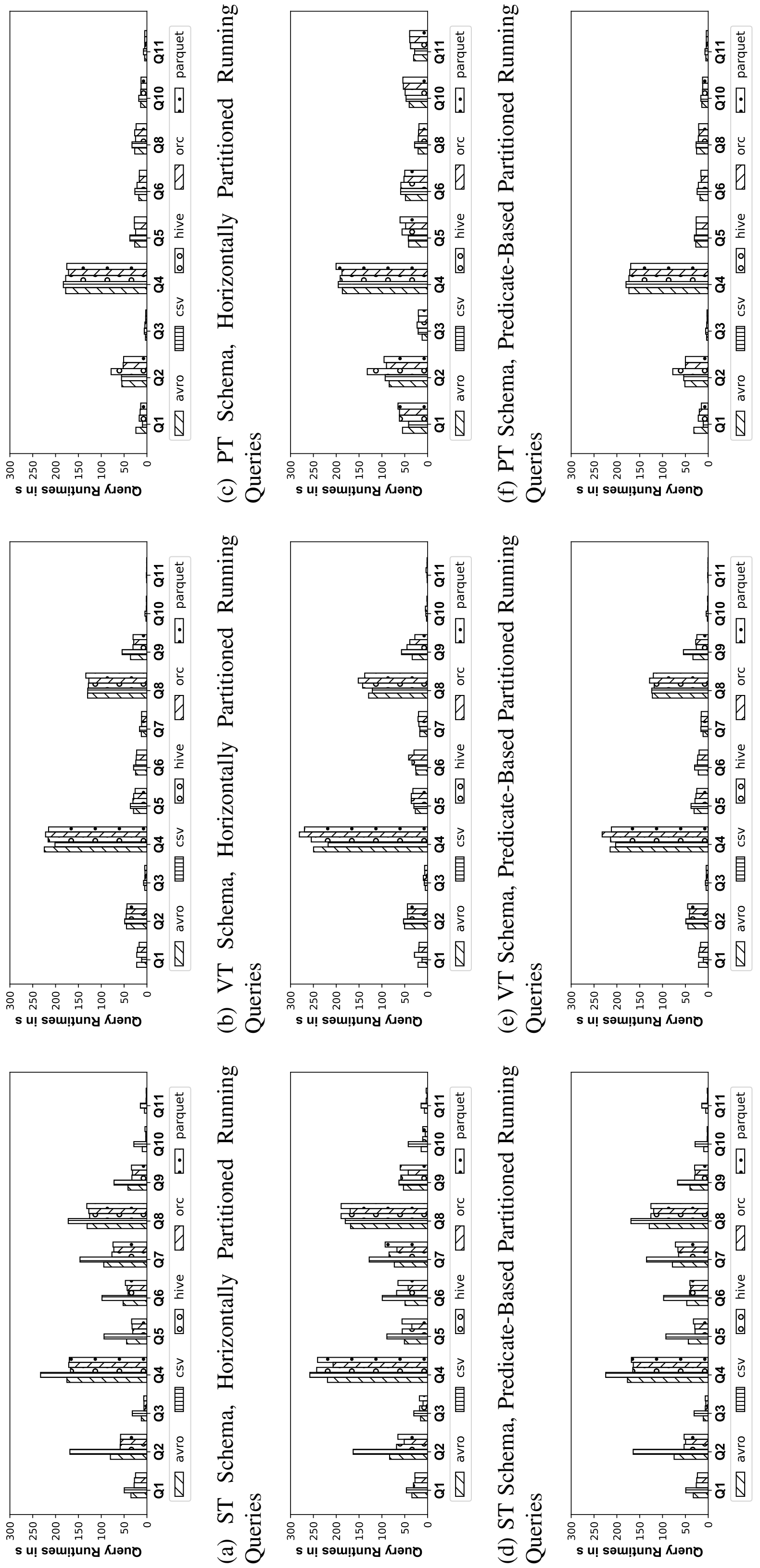
6.1. Descriptive and Diagnostic Analysis

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 SP^{2}Bench 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



(a) ST Schema, Horizontally Partitioned Running Queries

(b) VT Schema, Horizontally Partitioned Running Queries

(c) PT Schema, Horizontally Partitioned Running Queries

(d) ST Schema, Predicate-Based Partitioned Running Queries

(e) VT Schema, Predicate-Based Partitioned Running Queries

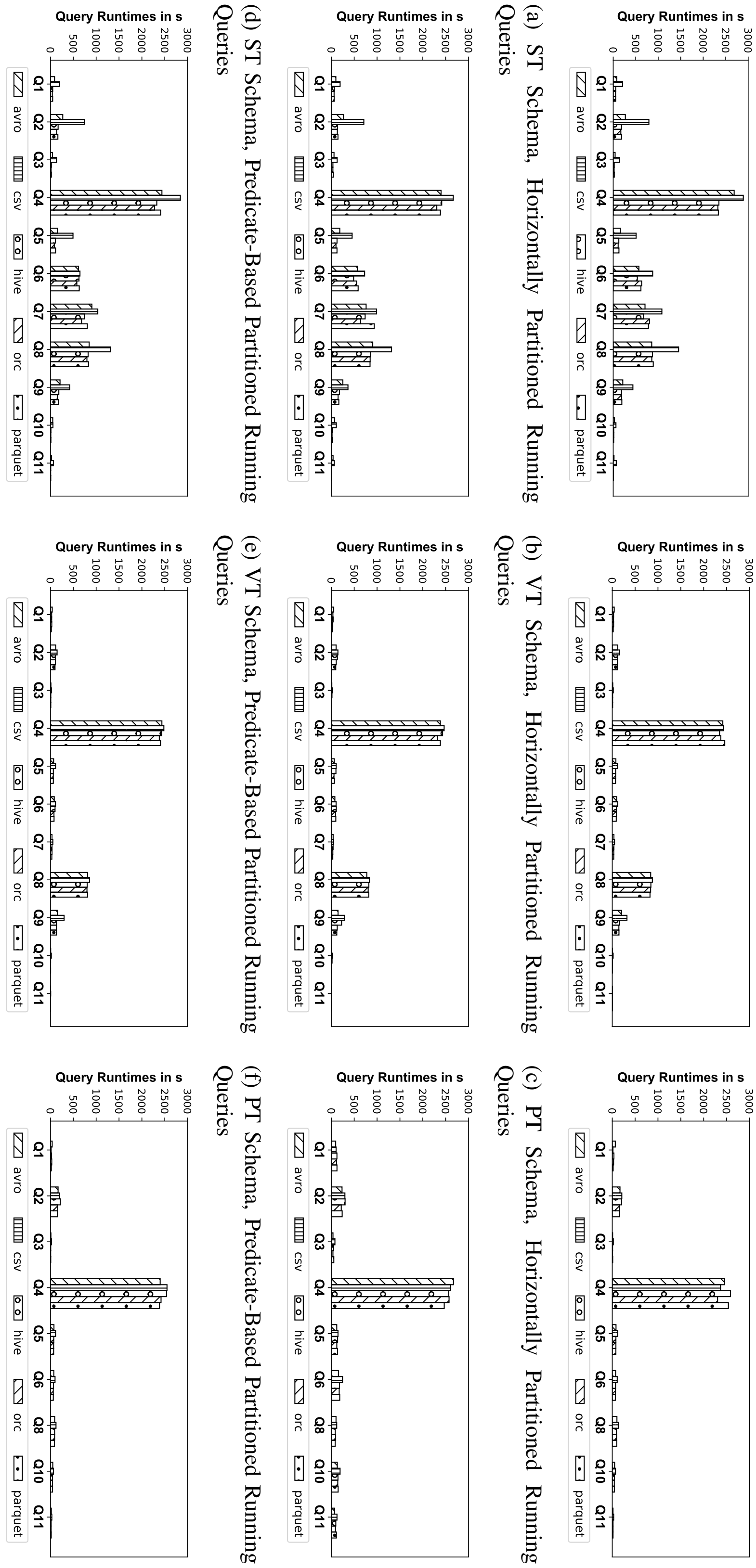
(f) PT Schema, Predicate-Based Partitioned Running Queries

(g) ST Schema, Subject-Based Partitioned Running Queries

(h) VT Schema, Subject-Based Partitioned Running Queries

(i) PT Schema, Subject-Based Partitioned Running Queries

Fig. 8.: 100M dataset Average Query Execution Run-times (full-size figures are on the project github repository)



(a) ST Schema, Horizontally Partitioned Running Queries

(b) VT Schema, Horizontally Partitioned Running Queries

(c) PT Schema, Horizontally Partitioned Running Queries

(d) ST Schema, Predicate-Based Partitioned Running Queries

(e) VT Schema, Predicate-Based Partitioned Running Queries

(f) PT Schema, Predicate-Based Partitioned Running Queries

(g) ST Schema, Subject-Based Partitioned Running Queries

(h) VT Schema, Subject-Based Partitioned Running Queries

(i) PT Schema, Subject-Based Partitioned Running Queries

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 8, 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. *BGB* 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 SP^{2}Bench 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 SP^{2}Bench 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 SP^{2}Bench.

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 SP^{2}Bench 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 *homepage*). 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 *SP^{2}Bench* 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 SP^{2}Bench, 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[10] .

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.

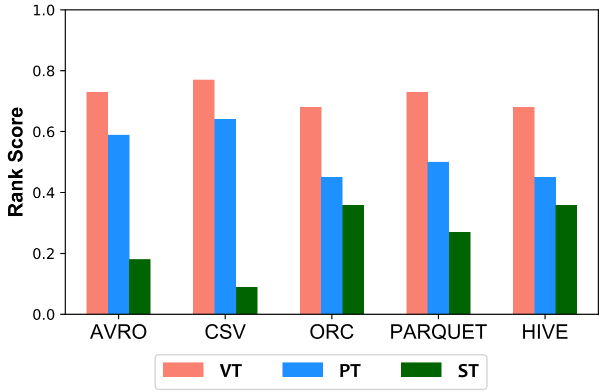
For the $100M$ dataset figures, we observe that the ST schema is always the worst performing one with 100% when HP and SBP are chosen as partitioning techniques. However, for the PBP partitioning, the ST schema has the second highest scores after the VT schema with 60%. On the other side, the VT schema 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*

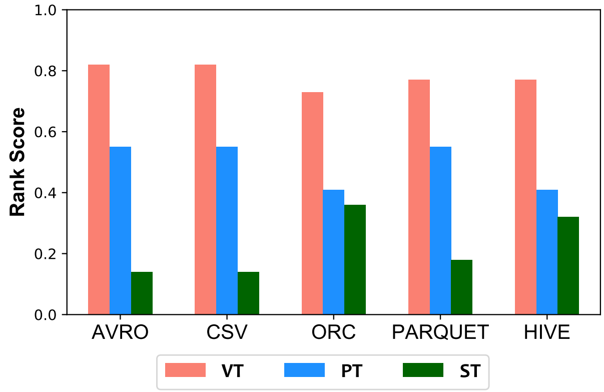
Figure 11 shows our different partitioning techniques (i.e. Horizontal, Subject-based, and Predicate-based Partitioning) ranking scores for the 100M (Figures 11 (*a*), (*c*), and (*e*)), 500M (Figures 11 (*b*), (*d*), and (*f*)) triples datasets for different relational schema ST, VT, and PT respectively. 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 [3^{rd} 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

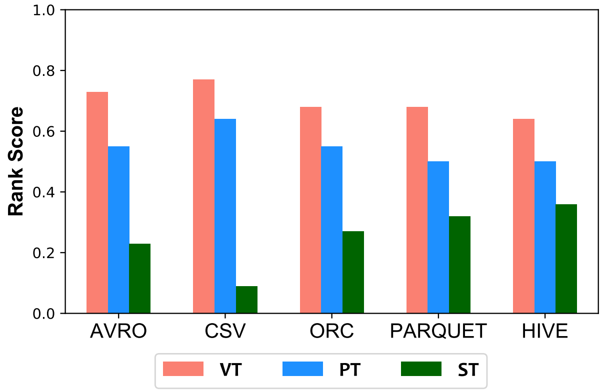
[10]https://datasystemsgrouput.github.io/SPARKSQLRDFBenchmarking/



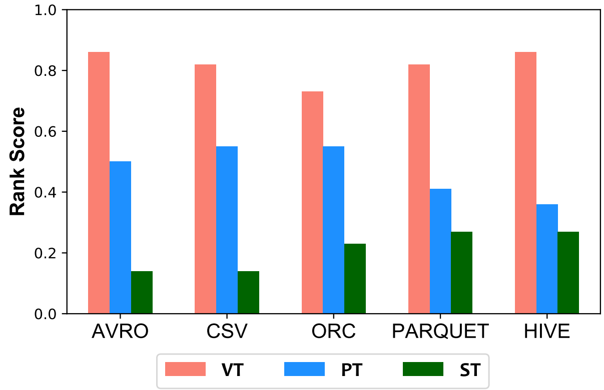
(a) 100M_HP partitioned Relational Schema Ranking Scores



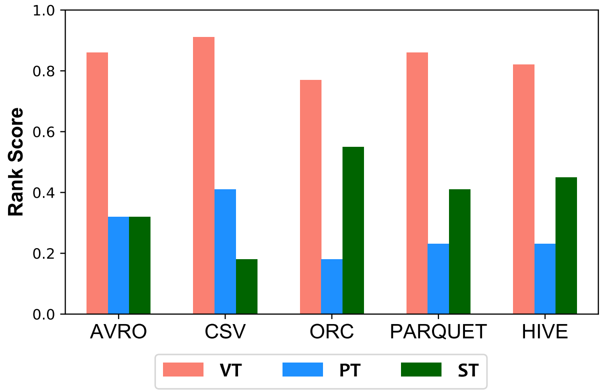
(b) 500M_HP partitioned Relational Schema Ranking Scores



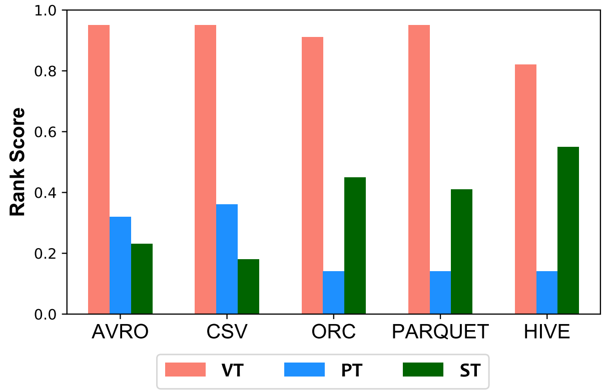
(c) 100M_SBP partitioned Relational Schema Ranking Scores



(d) 500M_SBP partitioned Relational Schema Ranking Scores

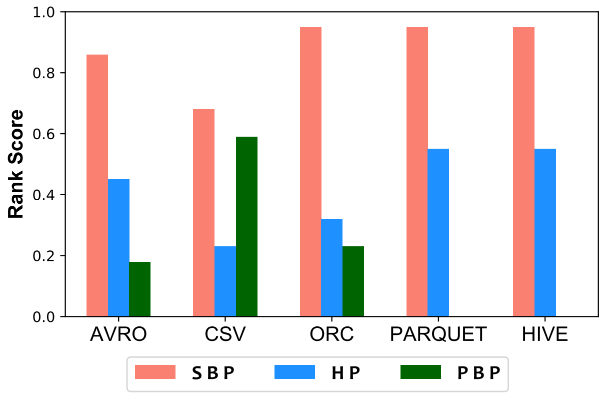


(e) 100M_PBP partitioned Relational Schema Ranking Scores

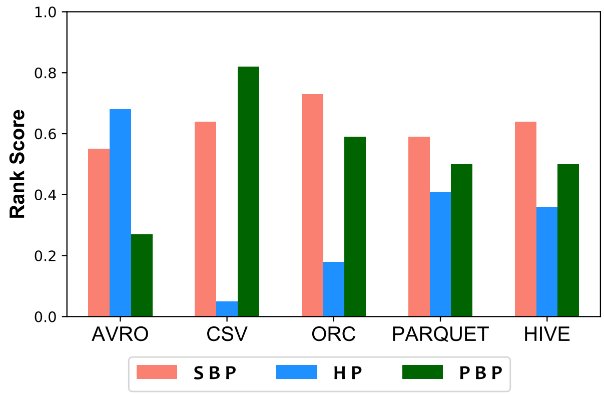


(f) 500M_PBP partitioned Relational Schema Ranking Scores

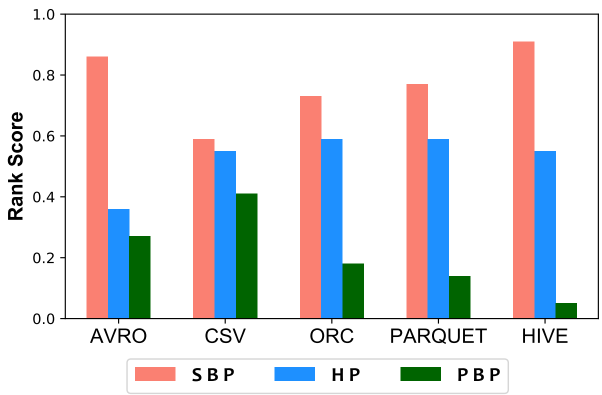
Fig. 10. Relational Schemas Ranking Scores for **100M** and **500M** Triples datasets (Reading Key: the higher is the better).



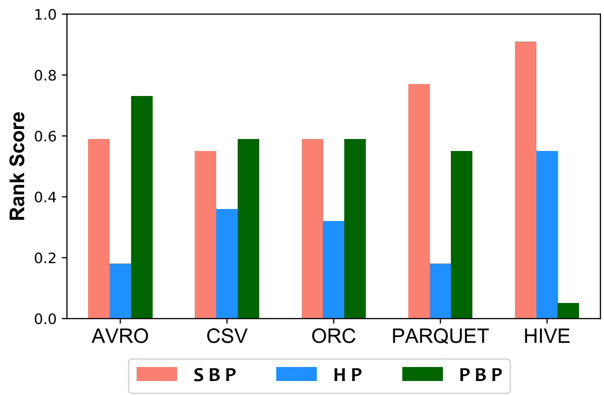
(a) 100M_ST Schema Partitioning Techniques Ranking Scores



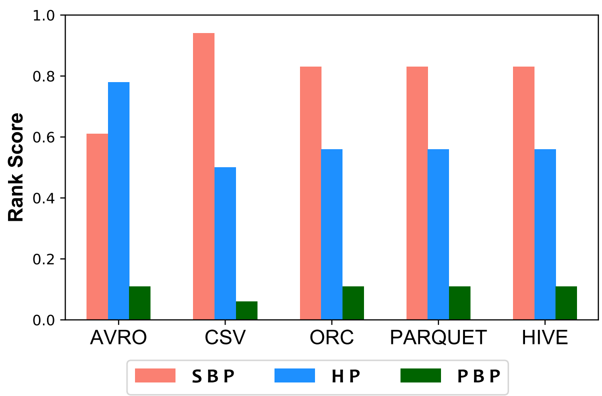
(b) 500M_ST Schema Partitioning Techniques Ranking Scores



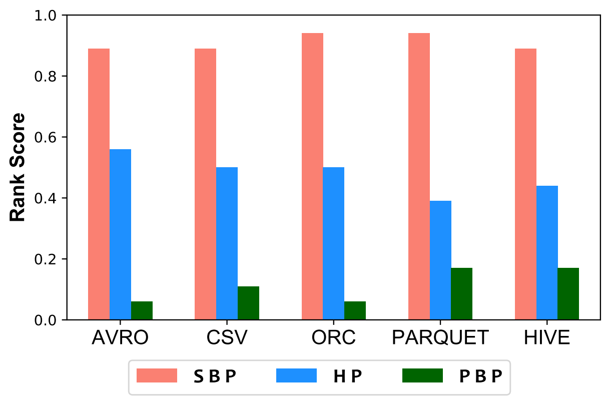
(c) 100M_VT Schema Partitioning Techniques Ranking Scores



(d) 500M_VT Schema Partitioning Techniques Ranking Scores



(e) 100M_PT Schema Partitioning Techniques Ranking Scores



(f) 500M_PT Schema Partitioning Techniques Ranking Scores

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 73% 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 11 (a), (c), and (e)); ranking scores for the $500M$ triples dataset are on the right, i.e., sub-figures 11 (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 [5^{th} 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 has 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.

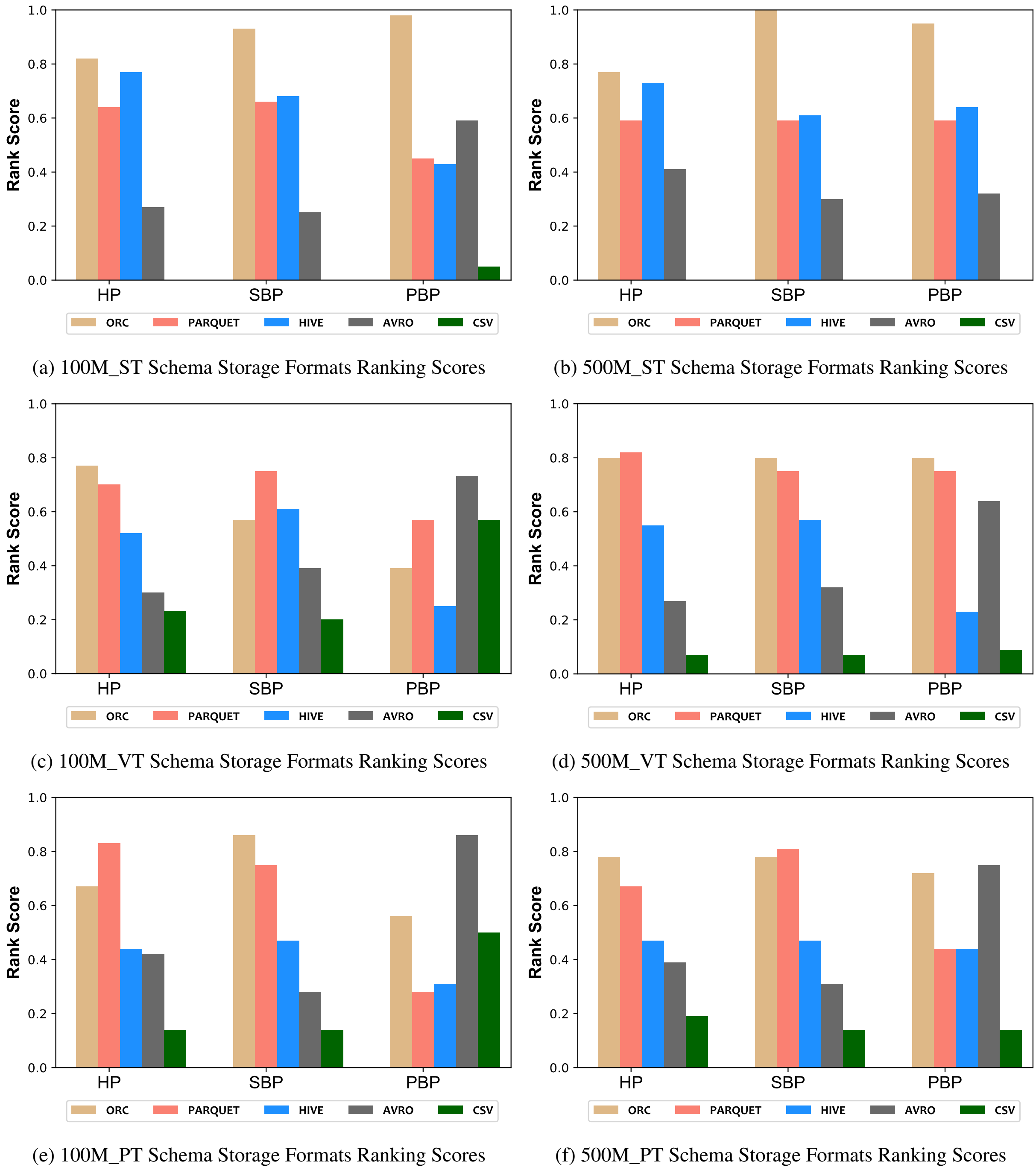
6.2.4. Combined Ranking Analysis

Table 2 shows all our possible different configuration combinations as shown from Figure 5. 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 raking 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



(a) 100M_ST Schema Storage Formats Ranking Scores
(b) 500M_ST Schema Storage Formats Ranking Scores
(c) 100M_VT Schema Storage Formats Ranking Scores
(d) 500M_VT Schema Storage Formats Ranking Scores
(e) 100M_PT Schema Storage Formats Ranking Scores
(f) 500M_PT Schema Storage Formats Ranking Scores

Fig. 12. Storage Backends Ranking Scores for **100M** and **500M** Triples datasets (Reading Key: the higher is the better).

100M	Rf	Rp	Rs	Rta	WAvg	AVG		500M	Rf	Rp	Rs	Rta	WAvg	AVG
a.i.1	0.27	0.45	0.23	0.10	1.13	0.32		**a.i.1**	0.41	0.68	0.14	0.14	1.50	0.41
a.i.2	0.00	0.23	0.09	0.01	0.32	0.11		**a.i.2**	0.00	0.05	0.14	0.00	0.19	0.06
a.i.3	0.82	0.32	0.27	0.19	1.96	0.47		**a.i.3**	0.77	0.18	0.23	0.12	1.69	0.39
a.i.4	0.64	0.55	0.32	0.24	1.94	0.50		**a.i.4**	0.59	0.41	0.27	0.17	1.66	0.42
a.i.5	0.77	0.55	0.36	0.30	2.19	0.56		**a.i.5**	0.73	0.36	0.27	0.19	1.85	0.45
a.ii.1	0.25	0.86	0.18	0.14	1.46	0.43		**a.ii.1**	0.30	0.55	0.14	0.09	1.19	0.33
a.ii.2	0.00	0.68	0.09	0.02	0.77	0.26		**a.ii.2**	0.00	0.64	0.14	0.03	0.78	0.26
a.ii.3	0.93	0.95	0.36	0.52	2.86	0.75		**a.ii.3**	1.00	0.73	0.36	0.45	2.76	0.70
a.ii.4	0.66	0.95	0.27	0.35	2.32	0.63		**a.ii.4**	0.59	0.59	0.18	0.19	1.75	0.45
a.ii.5	0.68	0.95	0.36	0.41	2.44	0.66		**a.ii.5**	0.61	0.64	0.32	0.26	1.98	0.52
a.iii.1	0.59	0.18	0.32	0.12	1.48	0.36		**a.iii.1**	0.32	0.27	0.23	0.07	1.03	0.27
a.iii.2	0.05	0.59	0.18	0.05	0.85	0.27		**a.iii.2**	0.00	0.82	0.18	0.05	1.00	0.33
a.iii.3	0.98	0.23	0.55	0.30	2.41	0.59		**a.iii.3**	0.95	0.59	0.45	0.42	2.62	0.66
a.iii.4	0.45	0.00	0.41	0.06	1.16	0.29		**a.iii.4**	0.59	0.50	0.41	0.25	1.89	0.50
a.iii.5	0.43	0.00	0.45	0.06	1.17	0.29		**a.iii.5**	0.64	0.50	0.55	0.32	2.12	0.56
b.i.1	0.30	0.36	0.73	0.20	1.59	0.46		**b.i.1**	0.27	0.18	0.86	0.15	1.49	0.44
b.i.2	0.23	0.55	0.77	0.24	1.70	0.52		**b.i.2**	0.07	0.36	0.82	0.13	1.30	0.42
b.i.3	0.77	0.59	0.68	0.46	2.55	0.68		**b.i.3**	0.80	0.32	0.73	0.36	2.38	0.62
b.i.4	0.70	0.59	0.68	0.43	2.44	0.66		**b.i.4**	0.82	0.18	0.82	0.32	2.37	0.61
b.i.5	0.52	0.55	0.64	0.32	2.06	0.57		**b.i.5**	0.55	0.55	0.86	0.42	2.33	0.65
b.ii.1	0.39	0.86	0.73	0.42	2.24	0.66		**b.ii.1**	0.32	0.59	0.82	0.31	1.94	0.58
b.ii.2	0.20	0.59	0.77	0.24	1.69	0.52		**b.ii.2**	0.07	0.55	0.82	0.18	1.49	0.48
b.ii.3	0.57	0.73	0.68	0.43	2.36	0.66		**b.ii.3**	0.80	0.59	0.73	0.50	2.65	0.71
b.ii.4	0.75	0.77	0.73	0.56	2.75	0.75		**b.ii.4**	0.75	0.77	0.77	0.58	2.79	0.76
b.ii.5	0.61	0.91	0.68	0.53	2.61	0.73		**b.ii.5**	0.57	0.91	0.77	0.55	2.63	0.75
b.iii.1	0.73	0.27	0.86	0.35	2.35	0.62		**b.iii.1**	0.64	0.73	0.95	0.59	2.75	0.77
b.iii.2	0.57	0.41	0.91	0.38	2.27	0.63		**b.iii.2**	0.09	0.59	0.95	0.23	1.69	0.54
b.iii.3	0.39	0.18	0.77	0.17	1.60	0.45		**b.iii.3**	0.80	0.59	0.91	0.58	2.83	0.77
b.iii.4	0.57	0.14	0.86	0.23	1.95	0.52		**b.iii.4**	0.75	0.55	0.95	0.55	2.75	0.75
b.iii.5	0.25	0.05	0.82	0.09	1.29	0.37		**b.iii.5**	0.23	0.05	0.82	0.08	1.25	0.37
c.i.1	0.42	0.78	0.55	0.33	2.02	0.58		**c.i.1**	0.39	0.56	0.50	0.23	1.70	0.48
c.i.2	0.14	0.50	0.64	0.16	1.37	0.43		**c.i.2**	0.19	0.50	0.55	0.16	1.37	0.41
c.i.3	0.67	0.56	0.55	0.35	2.22	0.59		**c.i.3**	0.78	0.50	0.55	0.36	2.35	0.61
c.i.4	0.83	0.56	0.50	0.39	2.44	0.63		**c.i.4**	0.67	0.39	0.41	0.23	1.91	0.49
c.i.5	0.44	0.56	0.50	0.25	1.80	0.50		**c.i.5**	0.47	0.44	0.36	0.18	1.59	0.43
c.ii.1	0.28	0.61	0.59	0.23	1.66	0.49		**c.ii.1**	0.31	0.89	0.55	0.31	1.95	0.58
c.ii.2	0.14	0.94	0.64	0.27	1.82	0.57		**c.ii.2**	0.14	0.89	0.55	0.23	1.67	0.53
c.ii.3	0.86	0.83	0.45	0.49	2.72	0.71		**c.ii.3**	0.78	0.94	0.41	0.48	2.65	0.71
c.ii.4	0.75	0.83	0.50	0.47	2.58	0.69		**c.ii.4**	0.81	0.94	0.55	0.57	2.84	0.77
c.ii.5	0.47	0.83	0.45	0.33	2.07	0.59		**c.ii.5**	0.47	0.89	0.41	0.33	2.09	0.59
c.iii.1	0.86	0.11	0.32	0.14	1.87	0.43		**c.iii.1**	0.75	0.06	0.32	0.10	1.63	0.38
c.iii.2	0.50	0.06	0.41	0.09	1.30	0.32		**c.iii.2**	0.14	0.11	0.36	0.04	0.70	0.20
c.iii.3	0.56	0.11	0.18	0.06	1.22	0.28		**c.iii.3**	0.72	0.06	0.14	0.05	1.40	0.31
c.iii.4	0.28	0.11	0.23	0.04	0.80	0.21		**c.iii.4**	0.44	0.17	0.14	0.05	1.05	0.25
c.iii.5	0.31	0.11	0.23	0.04	0.85	0.22		**c.iii.5**	0.44	0.17	0.14	0.05	1.05	0.25

Table 2

Configuration Ranking Criteria for 100M, and 500M triples datasets

	100M			500M		
	1st	*2nd*	*3rd*	*1st*	*2nd*	*3rd*
Rf	a.iii.3	a.ii.3	c.ii.3	a.ii.3	a.iii.3	b.i.4
Rp	a.ii.3	a.ii.4	a.ii.3	c.ii.3	c.ii.4	b.ii.5
Rs	b.iii.2	b.iii.1	b.iii.4	b.iii.1	b.iii.2	b.iii.4

Table 3

Non-overlapping top 3 best-performing configuration combinations for the Rf, Rp, and Rs ranking criteria

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 29^{th} performance rank position for `Q1`, 20^{th} for `Q2`,..etc.

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 11^{th} 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.ii.3*) configuration achieves that goal by 5 times for queries *Q3*, *Q4*, *Q5*, *Q6*, *Q8*. Therefore, the average of the Rf criterion 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.

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

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 SP^2Bench 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 SP^2Bench 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 SP^2Bench 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 *Seasme* SPARQL engine (known currently as *RDF4j*[11]) that is relied on a native RDF store using SP^2Bench dataset, with a pure translation of the same SP^2Bench 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[12] to evaluate a similar comparison between the Single Triples schema and the Vertical schema. On another

[11]https://rdf4j.eclipse.org/

[12]http://simile.mit.edu/rdf-test-data/barton

Rf	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**ranking<15**	**AVG**
a.iii.3	29	20	32	26	27	30	18	41	20	22	11	1	
a.ii.3	26	14	18	1	17	25	17	17	5	9	16	4	3
c.ii.3	15	15	1	9	5	1	31	3	31	27	26	5	
Rp	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**ranking<15**	**AVG**
a.ii.3	26	14	18	1	17	25	17	17	5	9	16	4	
a.ii.5	27	24	26	2	13	27	16	24	7	14	19	4	4
a.ii.4	23	23	24	4	24	26	19	23	8	13	21	3	
Rs	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**ranking<15**	**AVG**
b.iii.2	5	22	29	32	18	19	11	19	25	17	15	2	
b.iii.1	18	17	17	41	9	18	12	30	14	12	9	5	4
b.iii.4	12	6	23	44	23	22	14	36	4	7	10	6	
AVG	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**ranking<15**	**AVG**
b.ii.4	9	10	12	27	2	7	6	18	1	8	2	9	
a.ii.3	26	14	18	1	17	25	17	17	5	9	16	4	7
b.ii.5	16	2	10	28	14	14	7	16	2	6	4	8	
WAvg	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**ranking<15**	**AVG**
a.ii.3	26	14	18	1	17	25	17	17	5	9	16	4	
b.ii.4	9	10	12	27	2	7	6	18	1	8	2	9	6
c.ii.3	15	15	1	9	5	1	31	3	31	27	26	5	
Rta	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**ranking<15**	**AVG**
b.ii.4	9	10	12	27	2	7	6	18	1	8	2	9	
b.ii.5	16	2	10	28	14	14	7	16	2	6	4	8	7
a.ii.3	26	14	18	1	17	25	17	17	5	9	16	4	

Table 4

100M Triples Dataset Ranking Criteria Comparison

	100M		500M	
	BEST	**WORST**	**BEST**	**WORST**
Q1	c.i.2	c.iii.4	c.ii.2	a.i.2
Q2	b.ii.3	a.i.2	b.iii.4	a.i.2
Q3	c.ii.3	a.i.2	c.ii.3	a.i.2
Q4	a.ii.3	b.iii.3	a.ii.3	a.i.2
Q5	c.i.5	a.i.2	b.iii.3	a.i.2
Q6	c.ii.3	a.iii.2	c.ii.3	a.i.2
Q7	b.ii.1	a.i.2	b.ii.4	a.i.2
Q8	c.iii.4	a.iii.5	c.iii.4	a.i.2
Q9	b.ii.4	a.i.2	b.iii.4	a.i.2
Q10	b.iii.3	c.iii.4	b.iii.3	c.iii.2
Q11	b.i.3, b.ii.4	c.iii.4	b.i.3, b.ii.5	c.iii.2

Table 5

Best and Worst Configurations for Running SP2Bench Queries

side, Owens et.al [26] performed benchmarking experiments for comparing different RDF stores (eg. *Allegrograph*[13], *BigOWLIM*[14]) using different RDF benchmarks (e.g., LUBM[15]) and RDBMS benchmarks (e.g., The Transaction Processing Performing Council family (TPC-C) benchmark)[16]. 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 [9] (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/

[14] http://www.proxml.be/products/bigowlim.html

[15] http://swat.cse.lehigh.edu/projects/lubm/

[16] http://www.tpc.org/tpcc/

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-dependant 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^2Bench 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

[1] I. Abdelaziz, R. Harbi, S. Salihoglu, P. Kalnis and N. Mamoulis, SPARTex: A Vertex-Centric Framework for RDF Data Analytics, *PVLDB* **8**(12) (2015), 1880–1883. doi:10.14778/2824032.2824091. http://www.vldb.org/pvldb/vol8/p1880-abdelaziz.pdf.

[2] M. Hallo, S. Luján-Mora, A. Maté and J. Trujillo, Current state of Linked Data in digital libraries, *Journal of Information Science* **42**(2) (2016).

[3] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker and I. Stoica, Apache Spark: a unified engine for big data processing, *Commun. ACM* **59**(11) (2016), 56–65. doi:10.1145/2934664.

[4] G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis and D. Plexousakis, RDF Query Answering Using Apache Spark: Review and Assessment, in: *34th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2018, Paris, France, April 16-20, 2018*, 2018, pp. 54–59. doi:10.1109/ICDEW.2018.00016.

[5] M. Wylot and S. Sakr, Framework-Based Scale-Out RDF Systems, in: *Encyclopedia of Big Data Technologies.*, 2019. doi:10.1007/978-3-319-63962-8_225-1.

[6] S. Sakr, GraphREL: A Decomposition-Based and Selectivity-Aware Relational Framework for Processing Sub-graph Queries, in: *DASFAA*, 2009.

[7] I. Abdelaziz, R. Harbi, Z. Khayyat and P. Kalnis, A survey and experimental comparison of distributed SPARQL engines for very large RDF data, *Proceedings of the VLDB Endowment* **10**(13) (2017), 2049–2060.

[8] S. Sakr and G. Al-Naymat, Relational processing of RDF queries: a survey, *ACM SIGMOD Record* **38**(4) (2010), 23–28.

[9] M. Ragab, R. Tommasini and S. Sakr, Benchmarking Spark-SQL under Alliterative RDF Relational Storage Backends (2019).

[10] A. Akhter, A.-C.N. Ngonga and M. Saleem, An empirical evaluation of RDF graph partitioning techniques, in: *European Knowledge Acquisition Workshop*, Springer, 2018, pp. 3–18.

[11] J. Camacho-Rodríguez, A. Chauhan, A. Gates, E. Koifman, O. O'Malley, V. Garg, Z. Haindrich, S. Shelukhin, P. Jayachandran, S. Seth, D. Jaiswal, S. Bouguerra, N. Bangarwa, S. Hariappan, A. Agarwal, J. Dere, D. Dai, T. Nair, N. Dembla, G. Vijayaraghavan and G. Hagleitner, Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing, in: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, 2019, pp. 1773–1786. doi:10.1145/3299869.3314045.

[12] M. Armbrust, R.S. Xin, C. Lian, Y. Huai, D. Liu, J.K. Bradley, X. Meng, T. Kaftan, M.J. Franklin, A. Ghodsi and M. Zaharia, Spark SQL: Relational Data Processing in Spark, in: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, 2015, pp. 1383–1394. doi:10.1145/2723372.2742797.

[13] K. Alaoui, A categorization of RDF triplestores, in: *Proceedings of the 4th International Conference on Smart City Applications*, 2019, pp. 1–7.

[14] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen and C. Pinkel, An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario, in: *The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings*, 2008, pp. 82–97. doi:10.1007/978-3-540-88564-1_6.

[15] D.J. Abadi, A. Marcus, S.R. Madden and K. Hollenbach, Scalable semantic web data management using vertical partitioning, in: *VLDB*, 2007.

[16] E.I. Chong, S. Das, G. Eadon and J. Srinivasan, An efficient SQL-based RDF querying scheme, in: *VLDB*, 2005.

[17] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu and G. Lausen, Sempala: Interactive SPARQL query processing on hadoop, in: *International Semantic Web Conference*, Springer, 2014, pp. 164–179.

[18] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic and G. Lausen, S2RDF: RDF querying with SPARQL on spark, *Proceedings of the VLDB Endowment* **9**(10) (2016), 804–815.

[19] G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM Journal on scientific Computing* **20**(1) (1998), 359–392.

[20] J. Gray, Database and Transaction Processing Performance Handbook., 1993.

[21] M. Schmidt, T. Hornung, G. Lausen and C. Pinkel, SPˆ2Bench: A SPARQL Performance Benchmark, in: *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, 2009, pp. 222–233. doi:10.1109/ICDE.2009.28.

[22] M. Saleem, G. Szárnyas, F. Conrads, S.A.C. Bukhari, Q. Mehmood and A.-C. Ngonga Ngomo, How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks?, in: *The World Wide Web Conference*, ACM, 2019, pp. 1623–1633.

[23] T. Ivanov and M. Pergolesi, The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet, *Concurrency and Computation: Practice and Experience* (2019), e5523.

[24] S. Alexaki, V. Christophides, G. Karvounarakis and D. Plexousakis, On Storing Voluminous RDF Descriptions: The Case of Web Portal Catalogs, in: *Proceedings of the Fourth International Workshop on the Web and Databases, WebDB 2001, Santa Barbara, California, USA, May 24-25, 2001, in conjunction with ACM PODS/SIGMOD 2001. Informal proceedings*, 2001, pp. 43–48.

[25] L. Sidirourgos, R. Goncalves, M.L. Kersten, N. Nes and S. Manegold, Column-store support for RDF data management: not all swans are white, *PVLDB* **1**(2) (2008), 1553–1563. doi:10.14778/1454159.1454227. http://www.vldb.org/pvldb/1/1454227.pdf.

[26] A. Owens, N. Gibbins et al., Effective benchmarking for RDF stores using synthetic data (2008).

[27] V.A. Arrascue Ayala, P. Koleva, A. Alzogbi, M. Cossu, M. Färber, P. Philipp, G. Schievelbein, I. Taxidou and G. Lausen, Relational schemata for distributed SPARQL query processing, in: *Proceedings of the International Workshop on Semantic Big Data*, 2019, pp. 1–6.

[28] M. Wylot, M. Hauswirth, P. Cudré-Mauroux and S. Sakr, RDF data storage and query processing schemes: A survey, *ACM Computing Surveys (CSUR)* **51**(4) (2018), 84.

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

[30] C. Weiss, P. Karras and A. Bernstein, Hexastore: sextuple indexing for semantic web data management, *PVLDB* **1**(1) (2008).

[31] J.J. Levandoski and M.F. Mokbel, RDF data-centric storage, in: *2009 IEEE International Conference on Web Services*, IEEE, 2009, pp. 911–918.

[32] A. Akhter, A.N. Ngomo and M. Saleem, An Empirical Evaluation of RDF Graph Partitioning Techniques, in: *EKAW*, 2018. doi:10.1007/978-3-030-03667-6_1.