Benchmarking Spark-SQL under Alliterative RDF Relational Storage Backends

Mohamed Ragab[1], Riccardo Tommasini[2], and Sherif Sakr[1]

[1] Tartu University, Data Systems Group, Tartu, Estonia
Firstname.Lastname@ut.ee
[2] Politecnico di Milano, DEIB, Milan, Italy
Firstname.Lastname@polimi.it

Abstract. Recently, a wide range of Web applications (e.g. *DBPedia*, *Uniprot*, and *Probase*) are built on top of vast RDF knowledge bases and using 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 (e.g, Hadoop, Spark) can handle vast relational repositories, however, their application in the Semantic Web context is still limited. One possible reason is that such frameworks rely on distributed systems, which are good for relational data, however, their performance on dealing with graph data models like RDF have not been well-studied yet. In this paper, we present a systematic comparison of there relevant RDF relational schemas, i.e., Single Statement Table, Property Tables or Vertically-Partitioned Tables queried using Apache Spark. We evaluate the performance of Spark SQL querying engine for processing SPARQL queries using three different storage back-ends, namely, PostgreSQL, Hive, and HDFS. For the latter one, we compare four different data formats (CSV, ORC, Avro, and Parquet). We drove our experiment using a representative query workloads from the `SP`^{2}`Bench` benchmark scenario. The results of our experiments show many interesting insights about the impact of the relational encoding scheme, storage backends and storage formats on the performance of the query execution process.

Keywords: Large RDF Graphs · Apache Spark · SPARQL · Spark-SQL· RDF Relational Schema .

1 Introduction

The Linked Data initiative is fostering increasing adoption of semantic technologies like never before [1, 2]. Vast RDF datasets (e.g. *DBPedia*, *Uniprot*, and *Probase*) are now publicly available and new challenges for storing, managing and querying such data remain unveil. Recently, the Semantic Web community started investigating on how to leverage big data processing frameworks (e.g., Hadoop, Spark) to process large amounts of RDF data [3, 4]. A number of systems were designed to handle a huge amount of RDF data [5]. In practice, big data processing frameworks demand data partitioning to exploit the

full power of a distributed solution. However, a main challenge towards scalable and distributed RDF query processing is data partitioning. In particular, efficient partitioning of RDF data remains an open challenge [6]. An alternative approach relies on storing RDF data using a relational schema. To this extent, the relational RDF storage were proposed [7, 8], e.g., (i) ***Single Statement Table Schema (ST)***: A schema in which all triples are stored in one single large table [9, 10]. (ii) ***Vertically Partitioned Tables Schema (VT)***: A schema in which a table for each property with only two columns (subject, object) is stored [11]. (iii) ***Property (n-ary) Table Schema (PT)***: A schema in which multiple RDF properties are grouped and stored as columns in one table for the same RDF subject [12]. In general, in relations-based processing of RDF queries, the design of the underlying relational schema can significantly impact the performance of query processing [8]. In principle, a systematic analysis for the performance of Big Data framework on answering queries over relational RDF storage is still missing. In this paper, we take the first step for filling this gap by presenting a systematic analysis of the performance of Spark-SQL query engine for answering SPARQL queries over RDF repositories. In particular, we experimentally evaluate the performance of various storage backends, namely, PostgreSQL, Hive, and HDFS with textual and binary formats (e.g. CSV, Avro, Parquet, ORC). Moreover, we evaluate Spark-SQL under the three aforementioned relational RDF schemas (Single Statement Table, Property Tables, Vertically-Partitioned Tables) using various sizes of RDF databases and different query workloads generated by the `SP`^{2}`Bench` benchmark [13].

The remainder of the paper is organized as follows: Section 2 presents an overview of the required background information for our study. Section 3 describes the benchmarking scenario of our study. Section 4 describes the experimental setup of our benchmark. Section 5 presents the results and discusses various insights. We discuss the related work in Section 6 before we conclude the paper in Section 7.

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 [14] is an in-memory distributed computing framework for large scale data processing. Its core abstractions are Resilient Distributed Datasets (RDDs) and DataFrames, both are an immutable distributed collection of data elements, but DataFrames are also organized according to a specific schema into named and data-typed columns like a table in relational databases. Spark-SQL [15] is a high-level library for processing structured data on top of DataFrames. In particular, SparkSQL allows the ability to query data stored in the DataFrames

abstraction using SQL-like languages, optimized using the `Catalyst` query optimizer[3]. Last but not least, Spark supports different storage backends for reading and writing data. In the following, we present those backends that are relevant for our performance evaluation:

Apache Hive: A data warehouse built on top of Apache Hadoop for providing data query and analysis [16].

PostgreSQL DB: A popular open-source relational database system which is designed to handle a various range of workloads, from single machines to data warehouses or web services that can handle many concurrent users.

HDFS: The Hadoop Distributed File System. HDFS supports the following file formats: (i) *Comma Separated Value (CSV)*, which is a readable and easy to debug file format that, however, does not support block compression; (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. It also supports schema evolution, files splitting, and data blocks compression. (iv) *Optimized Row Columnar (ORC*[6]*)*, which provides a highly efficient way to store and process Hive data. However, unlike Avro, ORC does not support the schema evolution.

2.2 Relational RDF Schemas

The study of efficient storage of RDF data that also supports efficient data access is still an important research problem. Although there have been some research proposals for storing and querying RDF data in a non-relational stores, relational Schemas remains an efficient and scalable solution [17]. In the following, we present three prominent relational RDF schemas. Moreover, we provide examples for each schema, using the RDF data in Listing 1.1, and we translate the SPARQL query in Listing 1.2 into the corresponding SQL query for that given schema.

```
: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 1.1: RDF example in N-Triples. Prefixes are omitted.

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

Listing 1.2: SPARQL Example against RDF graph in Listing 1.1. Prefixes are omitted.

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

Subject	Predicate	Object
Journal1	type	Journal
Journal1	title	'Journal 1 (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 T1.subject=T2.subject
AND T2.subject=T3.subject
AND
T1.object='http://localhost/vocabulary/bench/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.

title

Subject	Object
Journal1	'Journal 1 (1940)'
Article1	'richer dwelling scrapped'
...	...

type

Subject	Object
Journal1	Journal
Article1	Article
...	...

issued

Subject	Object
Journal1	1940
...	...

```
SELECT
T2.object AS Year
FROM Title T1, Issued T2, Type T3
WHERE T1.subject=T2.subject
AND T2.subject = T3.subject AND
T3.object='http://localhost/vocabulary/bench/Journal'
AND T1.object='Journal 1 (1940)'
```

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

Single Statement Table Schema is the approach that has been adopted by the majority of existing open-source RDF stores, e.g., Apache Jena, RDF4J and Virtuoso, as well as by several centralized RDF processing systems [11, 18]. This approach requires storing RDF datasets in a single triples table of three columns that represent the three components of the RDF triple, i.e., Subject, Predicate, and Object. Figure 1 shows the Single Statements Table representation schema of the Sample RDF graph shown in Listing 1.1, and the associated SQL translation for the query in Listing 1.2.

Vertically Partitioned Tables Schema is an alternative schema storage proposed by Abadi et.al. [11] to speed up the queries over RDF triple stores. In this schema, the RDF triples table is decomposed into a table of two columns (Subject, Object) for each unique property in the RDF dataset such that the first (subject) column contains all subject URIs of that unique property, and the second (object) contains all the object values (URIs and Literals) for those subjects. Figure 2 shows the vertically partitioned tables schema of the sample RDF graph shown in Listing 1.1, and the associated SQL translation for the query in Listing 1.2.

Property (n-ary) Tables Schema proposed to cluster multiple RDF properties as *n-ary* table columns for the same subject to group entities that are similar in structure. As one of the advantages of the Property Tables Schema is that it works perfectly with the highly structured RDF scenarios, but not for poorly

Journal

ID	title	issued	type
Journal1	'Journal 1 (1940)'	1940	Journal
...	...	...	...

Article

ID	title	journal	type
Article1	'richer dwelling scrapped'	Journal1	Article
...	...	...	...

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

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

structured ones [17]. Figure 3 shows the relational flattened property tables of the RDF graph in Listing 1.1 and the associated SQL translation for the query in Listing 1.2.

3 Benchmark Datasets & Queries

In our experimental evaluation, we have used one of the most popular RDF Benchmarks, `SP2Bench` [13].

3.1 SPARQL Performance Benchmark (SP²-Benchmark) Dataset

SP²Bench is a publicly available, language-specific SPARQL performance benchmark. We have chosen SP²Bench for our experimental evaluation, since it is one of the most *well-structured* synthetic benchmarks [19], something which perfectly fits the goals of our study. In particular, SP²Bench is centered around the Computer Science DBLP scenario, and it comprises both a data generator that enables the creation of arbitrarily large DBLP-like documents (in N-Triples format) in addition to a set of carefully designed benchmark SPARQL queries with a high diversity score of benchmark query features as stated by Saleem et.al. [19]. Moreover, these queries are covering most of the SPARQL key operators as well as various data access patterns. The generated RDF dataset simulates the real-world key characteristics of the academic social network distributions encountered from the original DBLP datasets[7].

We have reused a similar schema like the relational schema proposed by Schmidt et.al [17]. In their experiment, the SP²Bench 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[8] that is generated by SP²Bench generator.

[7] https://dblp.uni-trier.de/db/.

[8] DBLP-like RDF Data Produced by the SP²Benchhttp://dbis.informatik.uni-freiburg.de/forschung/projekte/SP2B/

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

Table 1: SP2Bench Query analysis in terms of complexity (number of joins), and number of projections for SPARQL query and our three considered RDF relational schemes (ST, VT, and PT).

3.2 Queries

The set of queries selected for our experiments are associated with the SP^{2}Bench scenario. These queries implement meaningful requests on top of the RDF data generated by the SP^{2}Bench generator, covering a variety of SPARQL operators as well as various RDF access patterns. This list of queries can be found, including a short textual description for each query in the benchmark project website[9]. Notably, $Q9$ is not applicable for the PT relational schema.

In our experiments, we focus on two query features that give an indication of the query complexity, namely, *number of joins*, 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. 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. In addition, we discuss how we configured our experimental hardware and software components. Furthermore, we describe how we prepared and stored the datasets. Finally, we provide the design details of our experiments.

Hardware and Software Configurations: Our experiments have been executed on a Desktop PC running a Cloudera Virtual Machine (VM) v.5.13 with Centos v7.3 Linux system, running on Intel(R) Core(TM) i5-8250U 1.60 GHz X64-based CPU and 24 GB DDR3 of physical memory. We also used a 64GB virtual hard drive for our VM. We used Spark V2.3 parcel on Cloudera VM to

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

fully support Spark-SQL capabilities. We used the already installed Hive service on Cloudera VM (version:hive-1.1.0+cdh5.16.1+1431). We have installed a relational DB PostgreSQL (V. 11.4).

Benchmark datasets: Using the SP^{2}Bench generator, we generated three *synthetic* RDF datasets with scaling sizes (100k, 1M, 10M triples) in `NTriples` format. For SP^{2}Bench, 11 SPARQL queries are provided with their relational schemas translation[10]. We have evaluated all of these 11 queries of type SELECT. In some experiments, the results of `Q7` and `Q9` are missing. In particular, the results of `Q7` is missing for the 10M triples dataset using the Property tables schemes as its execution time is very long (more than 30 minutes to complete) even after caching its join tables DataFrames, while the results of `Q9` are missing in all dataset sizes as it is not implemented in the third schema (property tables relational schema PT) according to [17], and as shown in the early mentioned SPARQL-SQL translations Webpage. We have used these translated queries from SPARQL into SQL to be compliant with the Spark-SQL framework in our experiment.

Data Storage: We have conducted our experiments using various data storage backends and data storage file formats. We have used the Spark framework to convert the data from the CSV format (generated from processing N-triples files) into the other HDFS file formats (Avro, Parquet, and ORC). For this step, we have used the Spark framework, because of its ability of fast handling for the conversion of large files. Moreover, Spark supports reading different file formats into and from HDFS. This approach has been also used to load the data into the tables of the Apache Hive data warehouse (DWH) using three created databases, one for each dataset size. Converting 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 Spark session configuration in addition to the warehouse location. Last but not least, we have also created three PostgreSQL databases, one for each dataset size, and created tables within them with the expected schema and data-types for each table according to the different RDF relational schemes (ST, VT and PT). Then, we have loaded the data into the PostgreSQL tables from CSV files using the PostgreSQL databases tables *'COPY'* command.

Experiments: The main goal of our benchmarking experiment is to evaluate and compare the *execution times* of the SQL translations of the SPARQL queries over the Spark-SQL framework using the three introduced relational schemes as well as on top of different storage backends. We have used the standard SP^{2}Bench SPARQL benchmark as one of the most popular and well-structured synthetic RDF benchmarks [19]. SP^{2}Bench comes with several SPARQL queries for evaluating the performance of different triple stores. In this experiment, we focused on the 'SELECT' queries of the benchmark. In particular, we selected 11 queries (Table 1) and used their SQL translations to conduct our experiment.

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

The descriptions for the translations from SPARQL to SQL for all schemes (ST, VT, and PT) are available in the translation page of SP^{2}Bench. We also made our used SQL translation for the SPARQL queries using the different relational schemes available in our project repository[11]

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 Spark session interface. All queries are evaluated for all schemas and on top of all the different storage backends Hive, PostgreSQL, and the HDFS file formats namely, CSV, Parquet, ORC, and Avro.

For each storage backend and a relational schema, we run the experiments for all queries five times (excluding the first *cold start* run time, to avoid the warm-up bias, and computed an average of the other 4 run times).

5 Experimental Results

In this section, we present the results of our experiments and discuss several interesting insights on the performance of the Spark-SQL query engine using the various relational RDF storage schemas and the various storage backends.

5.1 Query Performance Analysis

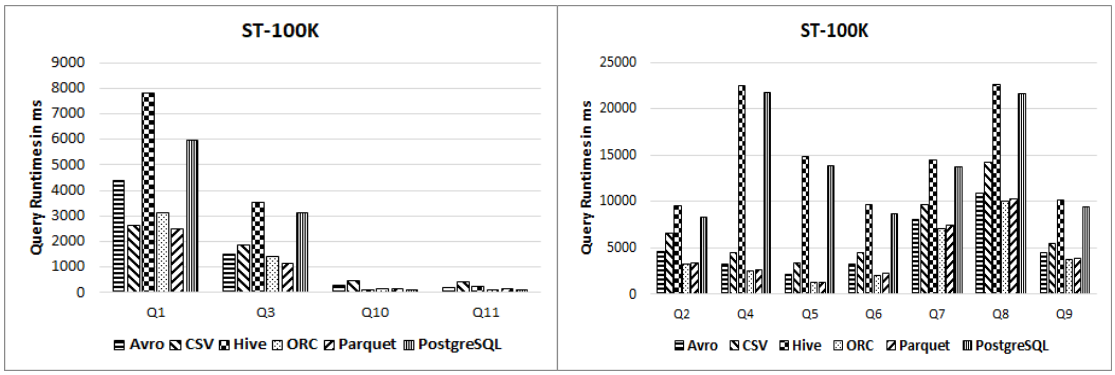
Figures 4, 5 and 6 show the average execution times for SP^{2}Bench queries for the 100K, 1M, 10M triples datasets, respectively. To foster readability, we organized the figures as follow: on the left side (sub-figures: a, c, and e), we present the results of *short-running* queries ($Q1$, $Q3$, $Q10$, $Q11$) while on the right side (sub-figures: b, d, and f), we present the result of *long-running* queries ($Q2$, $Q4$, $Q5$, $Q7$, $Q8$, $Q9$). Notably, according to Table 1, short-running queries are those presenting the least number of joins. For all the graphs, the reading key is the lower the better. Thus, we indicate that a particular configuration *outperforms* another one when it takes less time to compute the same query.

Figure 4 illustrates the results for the 100K triples dataset where we can observe that `PT` schema outperforms `VT` and `ST`, especially for simple queries like `Q1` and `Q11`, (cf Table 1). Indeed, the `PT` schema is the one that requires the minimum number of joins.

Scaling up the dataset size to 1M triples (Figure 5), we notice that for short-running queries (Figures 5 (a), (c), and (e)), the `PT` schema is the best performing, followed by `VT` schema, and finally the `ST` schema for all queries and for the majority of storage backends. The same observation can be seen for the long-running queries (Figures 5 (b), (d), and (f)) of all schemes. Notably, the `PT` schema provides a remarkable advantage over the `VT` and the `ST` ones for all queries and for the majority of storage backends, except for PostgreSQL.

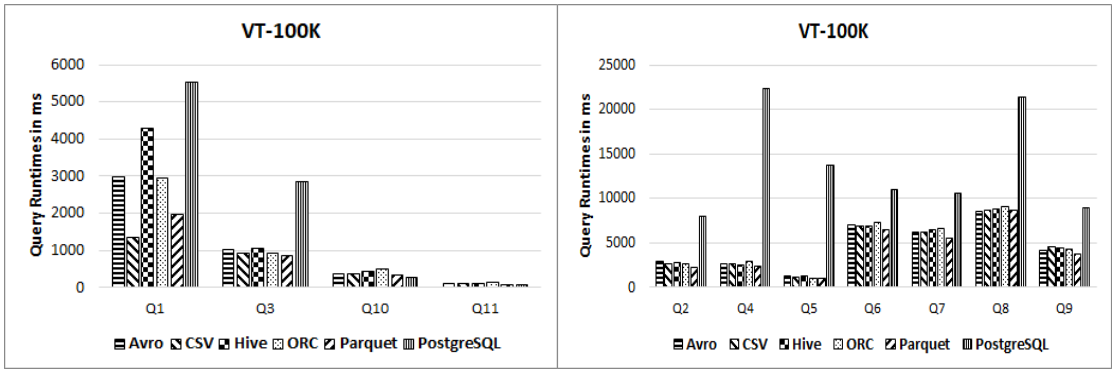
Finally, scaling up the dataset size to 10M triples (Figure 6), we observe that for short-running queries (Figures 6 (a), (c) and (e)), there is a huge performance

[11] https://github.com/DataSystemsGroupUT/SPARKSQLRDFBenchmarking



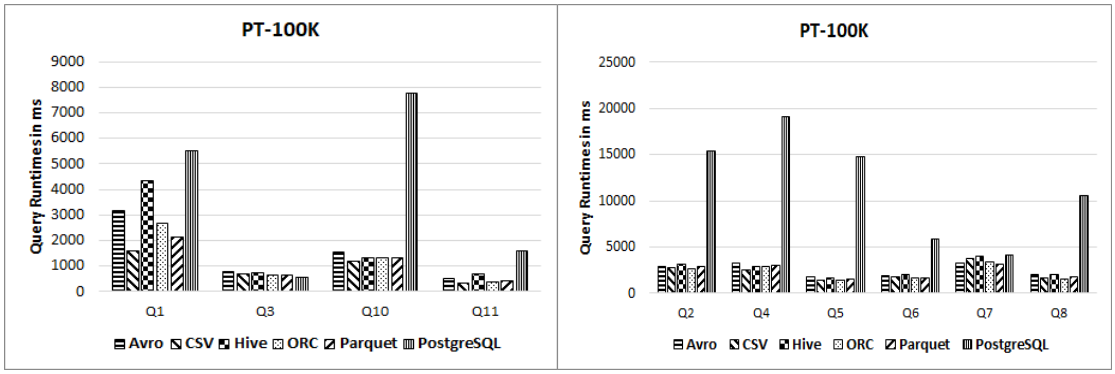
(a) Short running queries ST schema.

(b) Long running queries ST schema.



(c) Short running queries VT schema.

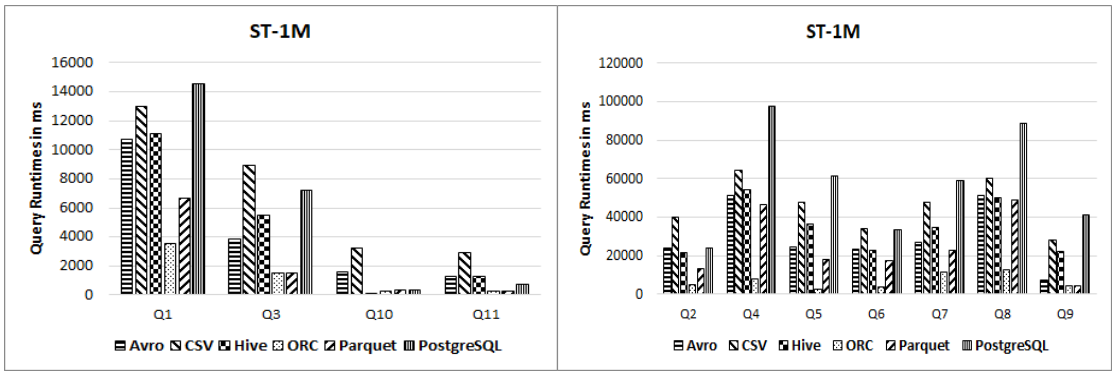
(d) Long running queries VT schema.



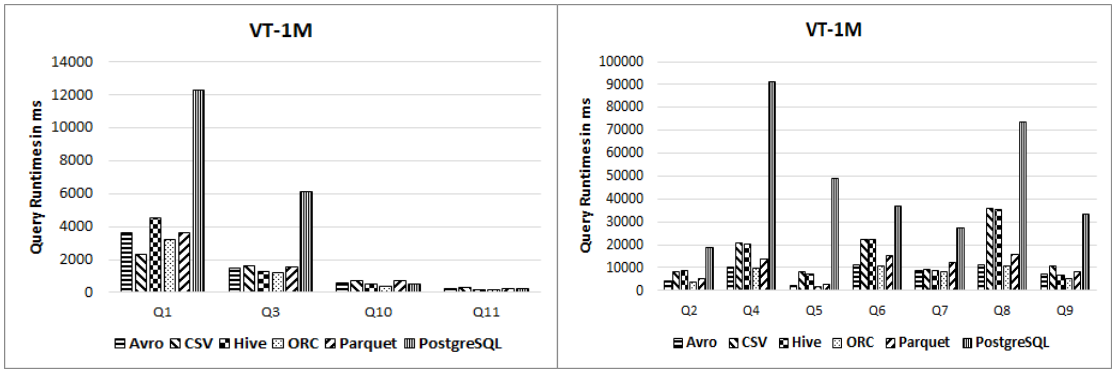
(e) Short running queries PT schema.

(f) Long running queries PT schema.

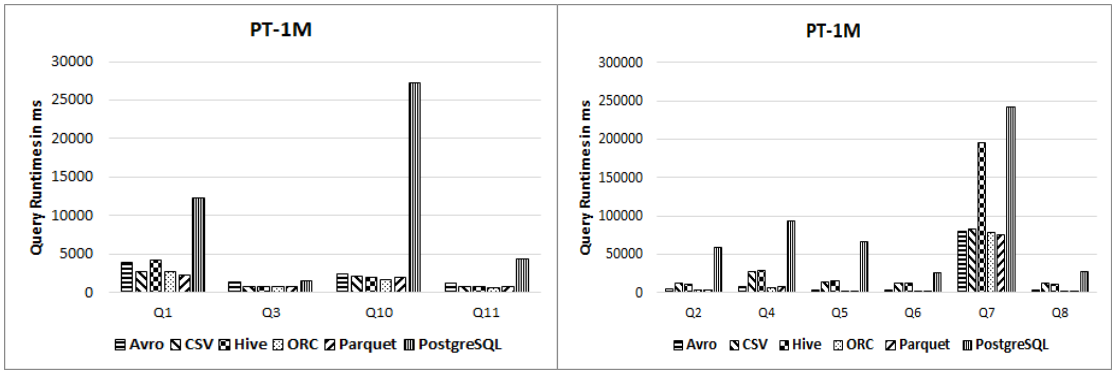
Fig. 4: Query Execution Times for **100K** Triples dataset.



(a) Short running queries ST schema. (b) Long running queries ST schema.

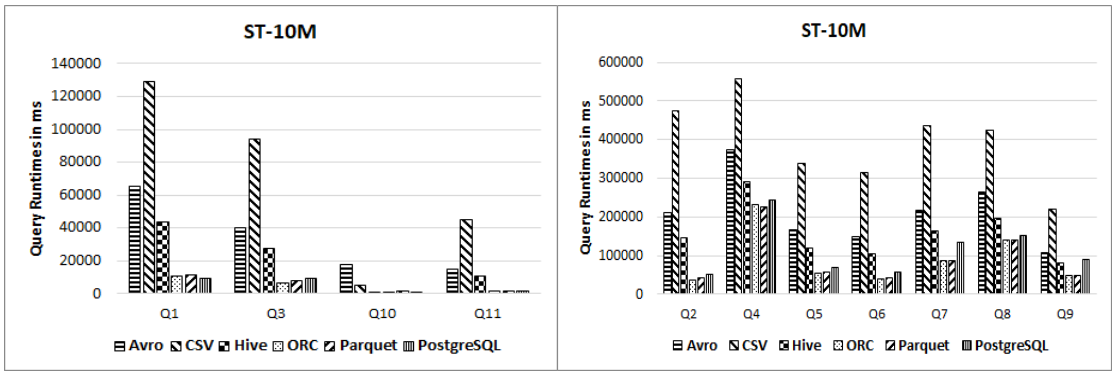


(c) Short running queries VT schema. (d) Long running queries VT schema.



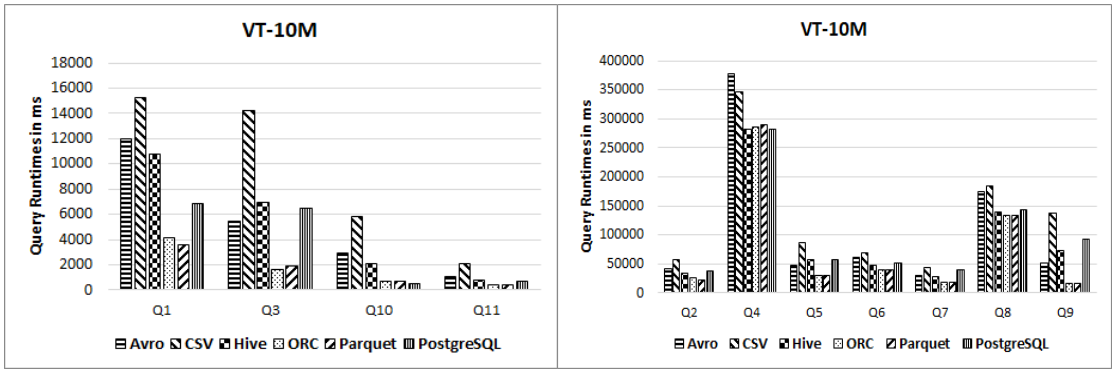
(e) Short running queries PT schema. (f) Long running queries PT schema.

Fig. 5: Query Execution Times for **1M** Triples dataset.



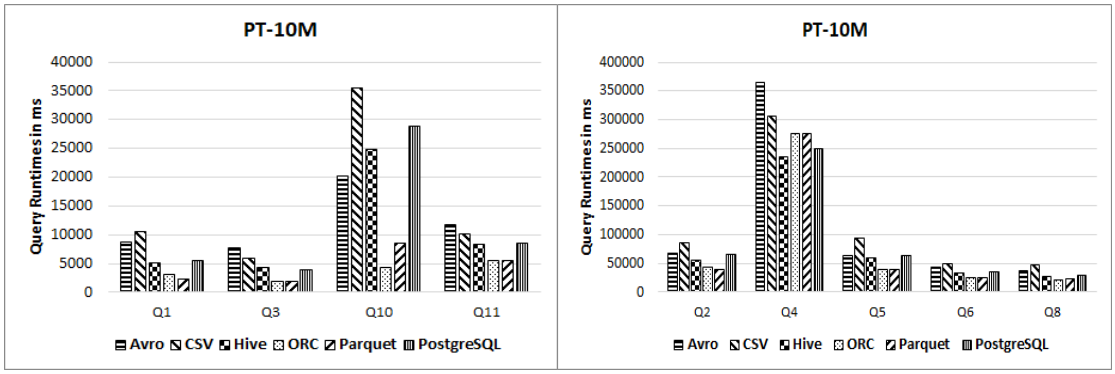
(a) Short running queries ST schema.

(b) Long running queries ST schema.



(c) Short running queries VT schema.

(d) Long running queries VT schema.



(e) Short running queries PT schema.

(f) Long running queries PT schema.

Fig. 6: Query Execution Times for **10M** Triples dataset.

	Avro	CSV	Hive	ORC	Parquet	PostgreSQL
ST100k	0.0%	0.0%	9.1%	54.5%	27.3%	9.1%
VT100K	9.1%	9.1%	0.0%	0.0%	63.6%	18.2%
PT100K	0.0%	40.0%	0.0%	40.0%	10.0%	10.0%
ST1M	0.0%	0.0%	9.1%	81.8%	9.1%	0.0%
VT1M	0.0%	9.1%	0.0%	90.9%	0.0%	0.0%
PT1M	0.0%	0.0%	0.0%	70.0%	30.0%	0.0%
ST10M	0.0%	0.0%	9.1%	63.6%	18.2%	9.1%
VT10M	0.0%	0.0%	0.0%	45.5%	36.4%	18.2%
PT10M	0.0%	0.0%	11.1%	44.4%	44.4%	0.0%

Table 2: How many times a given backend gives the best results?

	Avro	CSV	Hive	ORC	Parquet	PostgreSQL
ST100k	0.0%	18.2%	81.8%	0.0%	0.0%	0.0%
VT100K	0.0%	0.0%	0.0%	18.2%	0.0%	81.8%
PT100K	10.0%	0.0%	0.0%	0.0%	0.0%	90.0%
ST1M	0.0%	45.5%	0.0%	0.0%	0.0%	54.5%
VT1M	0.0%	18.2%	0.0%	0.0%	0.0%	81.8%
PT1M	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%
ST10M	9.1%	90.9%	0.0%	0.0%	0.0%	0.0%
VT10M	9.1%	90.9%	0.0%	0.0%	0.0%	0.0%
PT10M	33.3%	66.7%	0.0%	0.0%	0.0%	0.0%

Table 3: How many times a given backend gives the worst results?

increase on the average run times when using the `PT` schema, followed by the `VT` schema which is better than the ST schema for all queries as well as for the majority of the storage backends except for the PostgreSQL. The same observation can be seen in the long-running queries (Figures 6(b), (d), and (f)). In particular, the `PT` schema is greatly outperforming the `VT` and the `ST` schemes. This is due to the minimal number of joins required by the `PT` over `VT` then followed by `ST` schema.

5.2 Storage Backends Performance Analysis

Let us now investigate how different storage backends impact the performance in our experiments. Tables 2 and 3 report how many times a particular backend achieves the *best* or the *lowest* performance, respectively, considering the results of all experiments.

Considering the dataset with size of 100K triples, we observe that for the ST schema (Figures 4(a) and (b)), Hive is the lowest performing backend in 81.8% of the queries, for both short and long-running ones. Hadoop CSV immediately follows. In contrast, HDFS with ORC file format is the best performing storage backend in 54.5% of the queries, followed by HDFS Parquet. Notably, Hive and

PostgreSQL achieve the best for one query out of eleven, respectively Q10 and Q11.

For the VT schema (Figures 4 (c) and (d)), we notice that PostgreSQL is the lowest performing storage backend in 81.8% of the queries; HDFS with ORC file format has the lowest performance for 2 queries out of 11 queries. HDFS with Parquet file format is the best performing storage backend for 63.6% of the queries. PostgreSQL immediately follows by outperforming the other backends in 18.2% of the cases (Query evaluations).

Last but not least, for the PT schema (Figures 4(e) and (f)), PostgreSQL is the lowest performing storage backend in 90% of the queries, except for Q3 where Avro is the lowest performing one. Equally, HDFS with CSV and ORC file formats are the best performing backends in 40% of the queries, that we recall do not include Q9.

Considering the dataset of 1M triples size, for the ST schema (Figures 5 (a), and (b)), Postgres has the lowest performance in 54.5% of the queries, followed by CSV (45,5%). ORC is the best performing storage format in 81.8% of the queries, followed by Parquet and Hive (9.1%).

For the `VT` schema (Figures 5 (c) and (d)), Postgres is still the lowest performing backend in 81.8% of the queries, followed by CSV (18.2%). ORC is the best performing backend in almost all the queries (90.9%), except for $Q1$ where HDFS CSV has the highest performance.

For the PT schema (Figures 5 (e) and (f)), the performance dramatically dropped with almost the same outcomes of the `VT` schema. PostgreSQL is always the lowest performing backend. While ORC is the outperforming backend for 7 out of 10 queries ($Q9$ is not applicable here). That is, for queries $Q1$, $Q3$ and $Q7$, HDFS Parquet has the highest performance.

Regarding the dataset with 10M triple size, for the `ST` schema (Figures 6 (a) and (b)), CSV is the lowest performing storage backend with 90% of the queries, with the exception of Q10 where Avro has the lowest performance. queries. The best performing storage backend is ORC in 63.6% of the cases, followed by Parquet (18.2%).

For the `VT` schema (Figures 6 (c) and (d)), CSV is still the lowest performing storage backend in 90.9% of the cases, followed by Avro that has the lowest performance in $Q4$ this time. ORC has the best performance in 45.5% of the queries, followed by Parquet in 36.4%, and then PostgreSQL in 18.2%/

For the `PT` schema (Figures 6 (e) and (f)), that we recall for 10M triple dataset size do not include neither $Q7$ nor $Q9$, we observe that the CSV file format is the lowest performing storage backend in 66.7% of the cases. The best storage backends are HDFS with ORC and Parquet file formats both in 44.4% of the cases. Only for $Q4$, Hive shows the highest performance this time.

6 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, 17]. For example, Schmidt et.al. [17] performed an experimental comparison between existing RDF storage approaches using the SP^{2}Bench 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*[12]) that is relied on a native RDF store using SP^{2}Bench dataset, with a pure translation of the same SP^{2}Bench 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. [20] 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. [11], followed by Sidirourgos et.al. [21] who used the *Barton* library catalog data scenario[13] to evaluate a similar comparison between the Single Triples schema and the Vertical schema. On another side, Owens et.al [22] performed benchmarking experiments for comparing different RDF stores (eg. *Allegrograph*[14], *BigOWLIM*[15]) using different RDF benchmarks (e.g., LUBM[16]) and RDBMS benchmarks (e.g., The Transaction Processing Performing Council family (TPC-C) benchmark)[17]. 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 is the *first* that consider evaluating and comparing various relational-based schemes for processing RDF queries on top of the big data processing framework, Spark, and using different backend storage techniques.

7 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 conducted a systematic evaluation for the performance of the Spark-SQL query engine for answering SPARQL queries over different relational encoding for RDF datasets. In particular, we studied the performance of Spark-SQL using three different storage backends, namely, HDF, Hive and PostgreSQL. For HDFS we compared four different data formats, namely, CSV, OCR, Avro, and Parquet. We used SP^{2}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 ap-

[12] https://rdf4j.eclipse.org/
[13] http://simile.mit.edu/rdf-test-data/barton
[14] https://franz.com/agraph/allegrograph3.3/
[15] http://www.proxml.be/products/bigowlim.html
[16] http://swat.cse.lehigh.edu/projects/lubm/
[17] http://www.tpc.org/tpcc/

proaches for RDF relational storage, i.e., Single Triples Table Schema, Vertically Partitioned Tables schema, and Property Tables Schema.

The results of our experiments show that Property (n-ary) tables schema is able to achieve better performance in terms of query execution times. This is due to the extensive number of joins and self-joins required by Vertical Partitioned and Single Statement Table schemas. For the same reason, the Vertically-Partitioned schema works, in most of times, better than the Single Table schema. Regrading the supported Spark storage backend alternatives, the results have shown that using columnar HDFS file formats provide better performance for short running queries. For this, the main reason is that most of the queries of SP^2Bench are with a small number of projections. Thus, columnar file storage backends are able to perform better. On the other side, Postgres, CSV and Hive are shown to have the lowest performing storage options, respectively. Last but not least, scaling up the dataset sizes from 100K to 10 Million triples showed a dramatic performance enhancements for Property Tables and Vertical Partitioned Table schemas over the Single Statement Table schema. Moreover, with 10M triples dataset, the HDFS CSV file format has been shown to be the lowest performing storage backend followed by Avro.

As a natural extension of our benchmarking study, we aim to conduct our evaluations on a cluster deployments with varying node sizes, with more RDF benchmarks that have different types of queries and more scaling sizes of RDF datasets.

References

1. Ibrahim Abdelaziz, Razen Harbi, Semih Salihoglu, Panos Kalnis, and Nikos Mamoulis. Spartex: A vertex-centric framework for RDF data analytics. *PVLDB*, 8(12):1880–1883, 2015.
2. María Hallo, Sergio Luján-Mora, Alejandro Maté, and Juan Trujillo. Current state of linked data in digital libraries. *Journal of Information Science*, 42(2), 2016.
3. Giannis Agathangelos, Georgia Troullinou, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris 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*, pages 54–59, 2018.
4. Marcin Wylot and Sherif Sakr. Framework-based scale-out RDF systems. In *Encyclopedia of Big Data Technologies*. 2019.
5. Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. Rdf data storage and query processing schemes: A survey. *ACM Computing Surveys (CSUR)*, 51(4):84, 2018.
6. Adnan Akhter, Axel-Cyrille Ngonga Ngomo, and Muhammad Saleem. An empirical evaluation of RDF graph partitioning techniques. In *EKAW*, 2018.
7. Sherif Sakr. GraphREL: A Decomposition-Based and Selectivity-Aware Relational Framework for Processing Sub-graph Queries. In *DASFAA*, 2009.
8. Sherif Sakr and Ghazi Al-Naymat. Relational processing of rdf queries: a survey. *ACM SIGMOD Record*, 38(4):23–28, 2010.
9. Thomas Neumann and Gerhard Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

10. Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1), 2008.
11. Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.
12. Justin J Levandoski and Mohamed F Mokbel. Rdf data-centric storage. In *2009 IEEE International Conference on Web Services*, pages 911–918. IEEE, 2009.
13. Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph 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*, pages 222–233, 2009.
14. Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.
15. Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei 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*, pages 1383–1394, 2015.
16. Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther 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.*, pages 1773–1786, 2019.
17. Michael Schmidt, Thomas Hornung, Norbert Küchlin, Georg Lausen, and Christoph 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*, pages 82–97, 2008.
18. Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient sql-based rdf querying scheme. In *VLDB*, 2005.
19. Muhammad Saleem, Gábor Szárnyas, Felix Conrads, Syed Ahmad Chan Bukhari, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. How representative is a sparql benchmark? an analysis of rdf triplestore benchmarks? In *The World Wide Web Conference*, pages 1623–1633. ACM, 2019.
20. Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, and Dimitris 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*, pages 43–48, 2001.
21. Lefteris Sidirourgos, Romulo Goncalves, Martin L. Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.
22. Alisdair Owens, Nick Gibbins, et al. Effective benchmarking for rdf stores using synthetic data. 2008.