University of Tartu
Institute of Computer Science

Computer Science Curriculum

Large RDF Graph Processing on Top of Spark

Supervisor
Dr. Riccardo Tommasini

Co-Supervisor
Mohammed Ragab

Author: Sadig Eyvazov
Personal Number: B87777

This dissertation is submitted for the degree of Master of Science

March 2021

Large RDF Graph Processing on Top of Spark

Abstract

In recent years, we have witnessed an uncontrollable growth of data generated by machines or humans. Big Data is a term used to indicate data-related challenges. Although several challenges have been identified for big data, main ones remain *volume*, *velocity*, and *variety*. Volume is related to the large quantity of data. Velocity is related to the high rates at which the data is generated and processed. Last but not least, the variety is related to the presence of multiple data formats. Although there are many solutions to handle the data variety issue, the most popular one is the RDF (Resource Description Framework) data model. RDF is a W3C standard for Semantic Web, and many web applications are built on top of the RDF data model using a SPARQL query language. Thus, RDF data's continuous growth leads to investigate how to handle large RDF datasets in a distributed environment. Apache Spark is a modern, high-performance big data engine for processing vast amounts of data in a distributed environment. Big data systems like Apache Spark are not tailored for dealing with RDF data models; however, they have an excellent performance for large-scale relational data processing. Therefore, we implement the SPARQL queries over RDF data using Spark-SQL.

In this thesis, we use existing relational approaches for storing RDF data in Spark DataFrame data abstraction. We present a systematic performance evaluation of the Spark-SQL engine for processing SPARQL queries on the SP^{2}Bench benchmark. In particular, we used three relevant relational schemes, two storage backends, and several file formats. We have also applied three different partitioning techniques to see how it affects the Spark-SQL query execution performance. Finally, a major contribution of this thesis is an advanced analysis of experimental results and a discussion about the impact of each dimension (i.e. relational schema, partitioning technique, storage backend) on the performance of the query execution process in the distributed environment of Spark.

Keywords: Large RDF Graphs, SPARQL, Spark-SQL, RDF Relational Schema
CERCS: P170 - Computer science, numerical analysis, systems, control

Suur RKK graafi töötlus Sparkil

Lühikokkuvõte

Viimaste aastate jooksul oleme täheldanud inimeste ja masinate poolt genereeritud andmemahu suur kasvu. Sellega kaasnevate probleemidele viitamiseks kasutatakse terminit suurandmed ("big data"). Suurandmetega kaasnevatest mitmetest probleemidest on põhilisteks jäänud maht, kiirus ja varieeruvus. Maht on seotud andmete suure kogusega, kiirus on seotud andmete tootmise ja töötlemise suure kiirusega ning varieeruvus on seotud andmete mitme erineva võimaliku formaadiga. Mitmetest olemasolevatest lahendustest mis tegelevad varieeruvusega, on kõige populaarsem RKK (Ressursside kirjeldamise karkass "resource description framework") andmemudel. RKK on W3C standard semantilisele veebile. Mitmed veebirakendused on ehitatud RKK andmemudelile kasutades SPARQL päringukeelt. Seega tekib RKK suureneva kasutuselevõtuga vajadus uurida suurte RKK andmekogumite kasutamist hajus keskkonnas. Apache Spark on modernne, suure jõudlusega suurandme mootor, mis on mõeldud töötlema tohutuid andmete koguseid hajus keskkonnas. Suurandme süsteemid nagu Apache Spark ei ole mõeldud töötlema RKK andmemudeleid, aga siiski on neil suurepärane jõudlus töötlemaks suurel skaalal relatsioonilisi andmeid. Seega implementeerime me SPARQL päringuid RKK andmete pihta kasutades Spark-SQL-i.

Selles dissertatsioonis hoiustame me RKK andmestiku Spark Dataframe abstraktsioonina kasutades olemasolevaid relatsioonilisi meetodeid. Me esitleme Spark-SQL mootori süsteemse jõudluse hinnangu SPARQL päringute täitmiseks, kasutades SP^{2}Bench mõõtlusalust. Täpsemalt kasutasime me kolme asjakohast relatsioonilist skeemi, kahte talletus taustaprogrammi ja mitut faili formaati. Lisaks rakendasime ma ka kolme erinevat partitsioneerimis tehnikat, et näha nende mõju Spark-SQL päringute täitmise jõudlusele.Viimaks, on selle dissertatsiooni suured panused eksperimentaalsete tulemuste põhjalik analüüs ja disskusioon üle erinevate dimensioonide (näiteks relatsiooniline skeem, partitsioneerimis tehnika, talletus taustprogramm) mõju hajus keskkonnas tehtud päringu täitmise jõudlusele.

Märksõnad: Suur RKK graafid, SPARQL, Spark-SQL, RKK relatsiooniline skeem

CERCS: P170 Arvutiteadus, arvanalüüs, süsteemid, kontroll

Contents

1 Introduction

Nowadays, modern technologies play an important role in our daily life. Consequently, it results in large data collected from different sources such as social media, emails, transactions, and many more. Efficient management of this large data is considered as the key to success for many companies. However, big data opens new challenges for data systems, making it hard to process the data using traditional methods. Although multiple challenges were identified, the most important and relevant ones are *volume*, *velocity*, and *variety* [1]. Volume refers to a large amount of data that needs to be processed. Velocity is associated with the speed at which the data is generated and processed. Variety is related to the heterogeneity of data that can be generated by humans or machines in different forms, such as structured, semi-structured, or unstructured.

In our research, we address the volume and variety issues of big data by investigating how to leverage big data processing engines like Apache Spark [2] to achieve better performance when processing large RDF (Resource Description Framework) datasets. We implement the SPARQL queries over RDF data using Spark-SQL.

RDF[1] is a simple and standard graph-like data model for data interchange on the web. This data model allows structured and semi-structured data to be combined and shared across different applications due to its flexibility. The SPARQL [2] is the only standard graph query language recommended by W3C Semantic Web for querying RDF data. Particularly, a SPARQL query Q specifies a graph pattern P that is matched against an RDF graph G.

Native triplestores like RDF-3X [3], RDF4j [13], Apache Jena [14], gStore [4] and Sesame [5] performs well on a single machine for RDF datasets. However, a sharp continuous growth of RDF datasets makes it infeasible to store and process the entire datasets at reasonable performance on a single machine. Thus, it caused the investigation of new clustered RDF database systems to handle the large RDF datasets. Standalone distributed RDF stores like Virtuoso Cluster [6] and YARS2 [7]

[1] `https://www.w3.org/RDF/`

[2] `https://www.w3.org/TR/rdf-sparql-query/`

which are designed from the ground for RDF management, can be used to manage RDF data. However, the data can be accessed only via specific interfaces or query endpoints in these RDF data stores. Therefore, it restrains the interoperability and causes high integration costs in these systems.

In this thesis, we endorse the usage of modern big data engines such as Spark-SQL [8] and HDFS (Hadoop Distributed File System) [9] for distributed processing of large RDF data. Apache Spark-SQL is used to execute SQL queries on Spark DataFrame data abstraction. Spark-SQL provides its optimizer, Catalyst, which speeds up the query execution process on DataFrames. HDFS is a unified data storage that can be shared among various applications. Thus, HDFS allows you to access the same data without duplication or movement for different purposes in multiple systems. Although big data platforms are not originally designed for RDF processing, they were used successfully to build engines for large-scale relational data processing. There exist several approaches for representing the RDF data as relations [10, 11, 12], and we perform a systematic comparison of the most three common RDF relational schemes, namely, Single Statement Table (three columns containing one row for each RDF statement), Vertically Partitioned Tables (two-column table for every RDF predicate) and Property Tables (n-ary predicate-clustered tables). Last but not least, big data processing frameworks require data partitioning, which is the main challenge towards distributed RDF processing. Because the data partitioning changes the data locality, that can result in increasing shuffling cost impacting the query execution performance. Therefore, we also applied three various partitioning techniques (i.e. Subject-based, Predicate-based, Horizontal partitioning) [15] on RDF tables to show the impact of each partitioning technique on the querying RDF data. As a storage layer, we decided to use two storage backends, i.e., Hive and HDFS. For the latter one, we use four various data formats such as CSV, ORC, Avro, and Parquet.

We conduct our experiments using SP^{2}Bench [16] benchmark and analyze the results showing interesting insights about how the relational schema, data partitioning, and storage backends affect the performance of the query execution process in the distributed environment of Spark. Additionally, we make an advanced prescriptive analysis of Spark-SQL performance. We use the existing techniques for ranking the

experimental results, which allows us to provide a framework for deciding the best performing configuration combination of relational schemes, storage backends, and partitioning techniques. In the evaluation and discussion part, we also present how we combine the ranking criteria (i.e., relational schemas, partitioning techniques, and storage backends) to investigate the trade-offs between the experimental dimensions.

The remainder of this thesis is organized as follows: Chapter 2 provides background knowledge to understand the content of this thesis. In Chapter 3, we define the research questions and formulate the problem that we addressed in this thesis. Chapter 4 shows how we designed and implemented our experiments. In Chapter 4, we also provide the analysis methodology used to analyze the results of experiments. In Chapter 5, we discuss the evaluation process, experimental results, and various insights regarding them. Finally, we conclude the paper in Chapter 6.

2 Background

In this chapter, we present background knowledge that helps to understand the content of this thesis. In particular, Section 2.1 describes the RDF data model and its graph query language, SPARQL. Section 2.2 provides information about Apache Spark and Spark-SQL. We describe the three main design dimensions of our experiments, i.e., relational schemes, partitioning techniques, and data storage backends, in sections 2.3, 2.4, and 2.5, respectively. In section 2.6, we present the relational algebra providing example trees that is used to translate SPARQL queries into SQL. Finally, section 2.7 presents several RDF graph serializations.

2.1 RDF and SPARQL

Resource Description Framework (RDF) is a graph-based data model which is a core component of W3C Semantic Web. It is used to annotate resources in a triple form of <*subject, predicate, object*> on the web. In a triple, *predicate (P)* expresses a relationship between *subject (S)* and *object (O)*. RDF models the statement that can be interpreted as an edge from S to O labeled with P, $S \xrightarrow{P} O$. The relationship is always directional; it goes from subject to object. However, the same resource can occur in different positions in multiple triples. In particular, a certain triple subject can be an object of another triple and vice versa. Thus, it enables multiple connections between triples, creating a connected graph of data [17]. The nodes of such graphs can be represented by subject or object of triples, and the edges stand for predicates that show the relationship between corresponding nodes. Figure 1 depicts a simple example of an RDF graph from the publication-network DBLP[3] computer science scenario. Elements of the RDF triple can be one of the followings [18]:

- *IRI* (International Resource Identifier) provides a global identifier for a resource, thus, it can be re-used by others to identify the same resource on the web. IRIs can occur at all three positions of a triple (subject, predicate, or object).

[3] `https://dblp.org/`

- *Literal* is used for basic values such as numbers, strings, and dates, which is not an IRI. It can appear only at the object of triple.

- *Blank Node* is used as a local and unique identifier within a specific RDF dataset, which indicates a resource without assigning a global identifier. Blank node can appear at the subject or object of a triple.

Therefore, assuming disjoint infinite sets of I (International Resource Identifiers), L (literals), and B (blank nodes), an RDF triple can be formulated as (s,p,o) $\in$ (I $\cup$ B) x I x (I $\cup$ B $\cup$ L) with s, p, and o respectively the subject, predicate and object.

SPARQL is the standard query language for RDF data, which is recommended by the W3C [19]. In its simplest form, a SPARQL query Q consists of a set of RDF triple patterns, which are modeled as a directed graph and called a basic graph pattern (BGP). The triple pattern is very similar to the RDF triple, except that elements of the triple pattern (s, p, o) can be replaced by variables that may appear in multiple patterns. Particularly, a SPARQL query Q specifies a graph pattern P that is matched against an RDF graph G. The query pattern matching is achieved by comparing the unbounded variables in P with the elements in graph G, such that the resulting graph is contained in G.

The syntax of SPARQL is closely similar to the one of SQL. The *SELECT* operator specifies the variables that will be appeared in the result set of the query, the *FROM* operator specifies the datasets (the RDF datasets, or named Graphs) to be used for matching the triple pattern variables, and the *WHERE* operator defines the triple pattern, and optional subqueries.

For instance, the Sparql query Q_1 shown in Figure 2(a) can be interpreted as *"Return the journal with the name* `Journal 1 (1940)`*"*. The query corresponds to the triple patterns and graph pattern in Figure 2(b) and Figure 2(c), respectively.

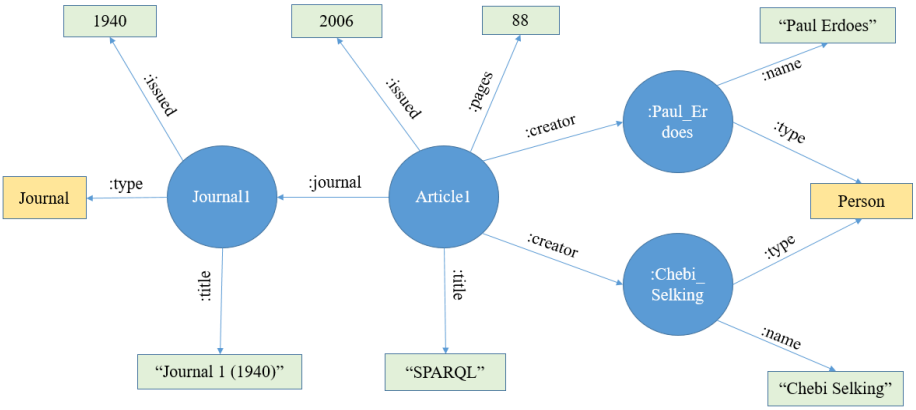


Figure 1: Visualization of an example RDF graph G_1. Each edge and its associated vertices correspond to an RDF triple, i.e., *<Journal1, issued, 1940>*.

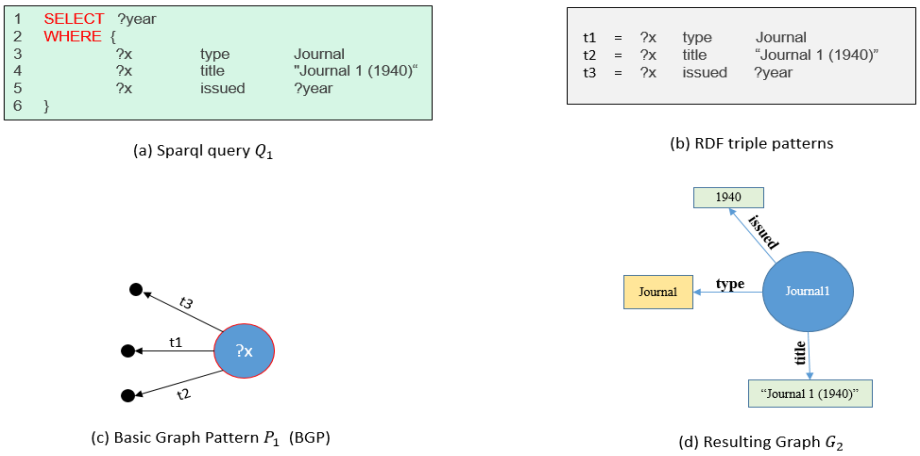


Figure 2: An example of SPARQL query (from the DBLP scenario)

Assuming data is stored in a table T(s, p, o), the Sparql query Q_1 can be answered by first decomposing it into three subqueries:

- $q_1 \equiv \sigma_{p=type \wedge o=Journal}(T)$
- $q_2 \equiv \sigma_{p=title \wedge o="Journal1(1940)"}(T)$
- $q_3 \equiv \sigma_{p=issued}(T)$

The subqueries are answered independently by matching variables on table T, and their intermediate results are joined on the subject attribute.

As a result of matching the variables of P_1 with the elements of G_1, we get the query graph G_2 *(Journal1* $\xrightarrow{type}$ *Journal, Journal1* $\xrightarrow{title}$ *Journal 1 (1940), Journal1* $\xrightarrow{issued}$ *1940)* which is illustrated in Figure 2(d).

2.2 Apache Spark and Spark-SQL

Apache Spark is the de-facto in-memory cluster computing system that can process big data workloads from any Hadoop data source [21]. Spark allows executing parallel computations in memory, which improves the speed of read and write operations as well as query execution performance. Consequently, compared to Hadoop, Spark is more efficient for applications that require frequent reuse of data sets across multiple parallel operations [20]. Spark mainly depends on two distributed main-memory data abstractions [22]: (i) Resilient Distributed Data sets (RDDs), distributed memory data abstraction which allows performing in-memory computations in a fault-tolerant manner and (ii) Data Frames (DFs), compressed and schema-enabled data abstraction. Unlike an RDD, data in DF is organized into a schema with named columns, like a table in a relational database. This nature of DF makes large data sets processing easier, allowing higher-level abstraction. Both abstractions facilitate the programming operations by supporting various relational operators such as join, selection, and projection that is not natively supported in Hadoop. Besides its efficient in-memory performance, Spark comes with high-level APIs for batch processing, graph and real-time stream processing, complex analytics as well as a relational interface called Spark-SQL.

Spark-SQL is a high-level library for structured data processing, which is used to execute SQL queries. The cornerstone of Spark-SQL is the DataFrame data abstraction since it allows for running queries on DataFrame by using its optimizer, *Catalyst* [8]. Catalyst makes logical and physical optimization on query plans and speeds up the query execution process on DataFrames. Whereas RDDs do not perform any join optimization. The reason behind that is that DataFrame provides Spark with more

information about the structure of data, which enables Catalyst to apply more tailored optimizations. The other benefit of using the DataFrame with Spark-SQL comes from the columnar, compressed in-memory representation of DF. Firstly, it allows for processing larger data sets compared with RDD. Secondly, DataFrame compression saves the data transfer cost between the nodes. Consequently, the best option for querying RDF data in Spark is to use Spark DataFrames API and Spark-SQL engine, which provide additional optimizations on the query plans.

Spark-SQL supports operating on various storage backends to read and write data through the DataFrame interface. In our experiments, we used two storage backends such as Apache Hive and HDFS. We used four different file formats for the latter one, i.e., CSV, Avro, ORC, and Parquet.

2.3 RDF relational schemes

Although triplestores can manage RDF data efficiently, it is a common approach to store the RDF data in a relational DBMS back-end for many of them [23]. There are several approaches to keep RDF in RDB. In the following, we show the three most common relational schemes that are suitable to represent RDF data. For each schema, we provide an example using the RDF graph G_1 and the respective SQL translation of SPARQL query Q_1.

Single Statement Table Single Statement Table (ST) stores the RDF triples in a single table with three columns *(s, p, o)*, containing one row for each RDF statement.

ST schema is used by many open-source triplestores such as RDF4J [13], Apache Jena [14], and Virtuoso [6], since it provides a simple and flexible schema for RDF data. Figure 3 illustrates the ST schema and SQL translation of SPARQL query Q_1 for the ST schema. Although the ST schema is the most straight-forward relational model for storing RDF data, it can not be considered as the best approach since it presents a number of disadvantages when it comes to query evaluation. A query with several predicates requires evaluating a set of self-joins on a large ST table, which has a negative effect on query evaluation. Therefore, this schema performs poorly for

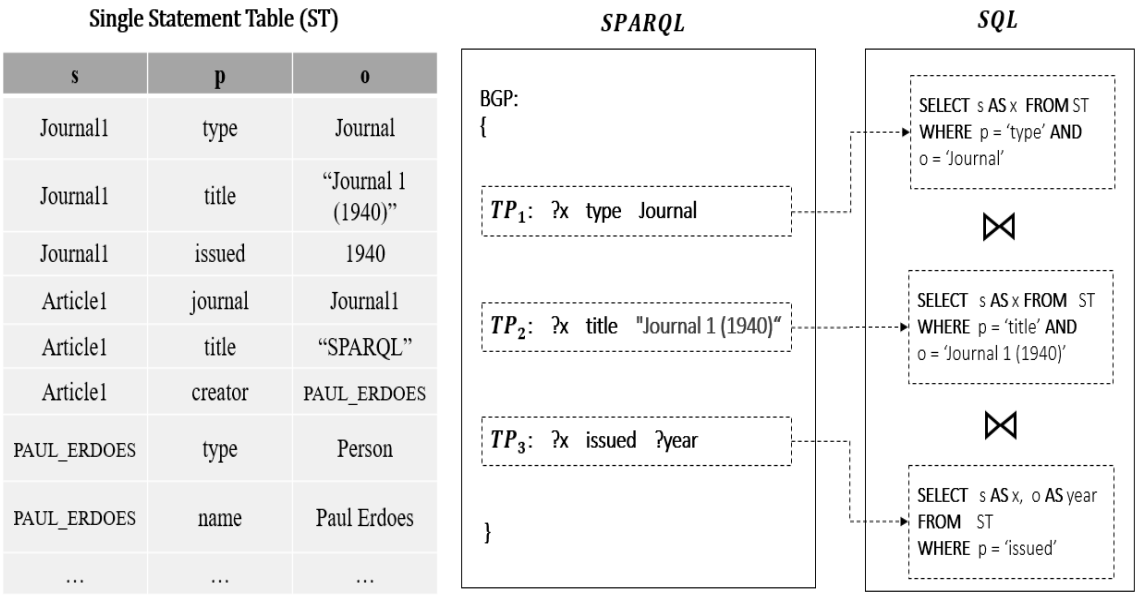


Figure 3: ST schema for RDF Graph G_1 and associated SQL translation of Q_1.

large-scale datasets due to expensive self-joins [24]. In Figure 3, it can be clearly seen that the table is referenced three times which requires several self-joins on this table.

Property Tables Other relational representations are used to reduce the number of joins needed for BGP evaluation, and one of them is called Property Tables (PT) [25]. In this schema, all similarly structured triples are clustered in the same table where column names are predicates, and each cell contains the object value of the corresponding subject and predicate. For instance, the *journal* table can have *title, issued,* and *type* columns that are predicates used to describe a journal in our example.

The grouping process of predicates can be either done using type definitions in the dataset itself or using clustering algorithms [26]. In principle, the main advantage of property tables compared to a single statement table is that they can reduce the number of subject-subject self-join operations. Particularly, the property tables approach improves the query evaluation performance by avoiding the high cost of many self-joins. In Figure 4, we see one of the property tables, *journal*, of graph G_1, and SQL translation of SPARQL Q_1 for the Journal table. For instance, all

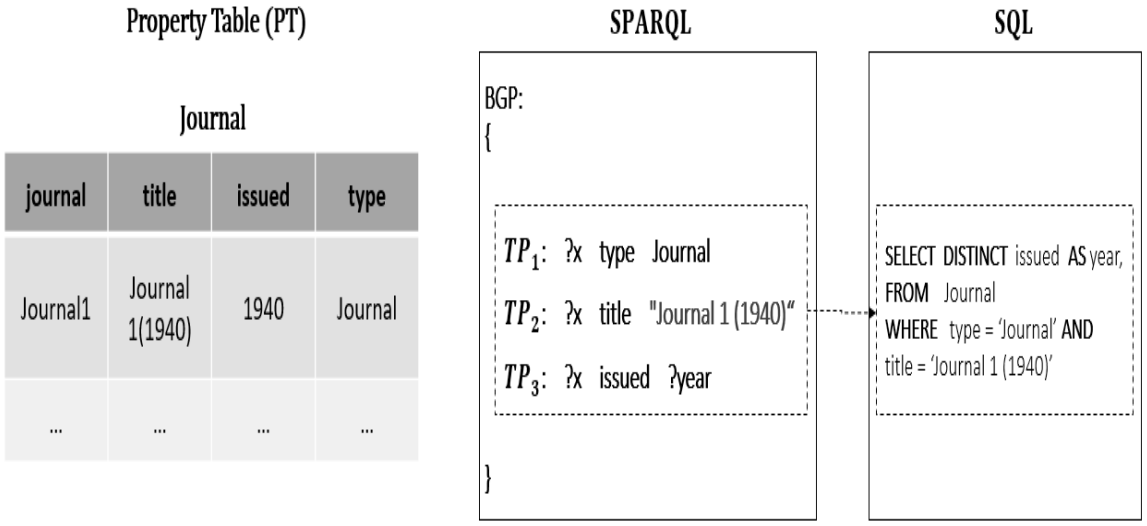


Figure 4: PT schema for RDF Graph G_1 and associated SQL translation of Q_1.

triple patterns in the SPARQL query Q_1 share the same variable for the subject position; thus, it can be answered without any join operation using the property table. Consequently, the PT schema for query Q_1 requires no joins instead of three compared to the ST schema.

However, the authors [27] discuss potential disadvantages of property tables. The main problem arises from the diverse structure of RDF, where not all subjects will use all properties. This makes the property tables very sparse containing many *NULL* values, which results in high storage overhead. The other drawback of PT schema is multi-valued predicates in RDF, e.g., an article has more than one creator in G_1. This issue can be solved by row duplication or using additional auxiliary tables [56]. Therefore, Property Tables are considered as an efficient approach for highly structured data, but not for poorly structured data [28].

Vertically Partitioned Table Vertically Partitioned (VT) schema is another representation for RDF data proposed by Abadi et al. [29]. This approach vertically partitions the single statement table into n tables, where n is the number of unique predicates in the dataset. A two-column (subject, object) table is generated for each

unique predicate, where a row corresponds to subject-object values connected through the predicate. VT schema does not store *NULL* values when a particular property does not describe the subject.

Figure 5 represents the vertically partitioned tables for RDF graph G_1 which are *title, type,* and *issued*. In Figure 5, we can also see the SQL translation of SPARQL query Q_1 which can easily select the corresponding VT table for each triple pattern ignoring additional selections.

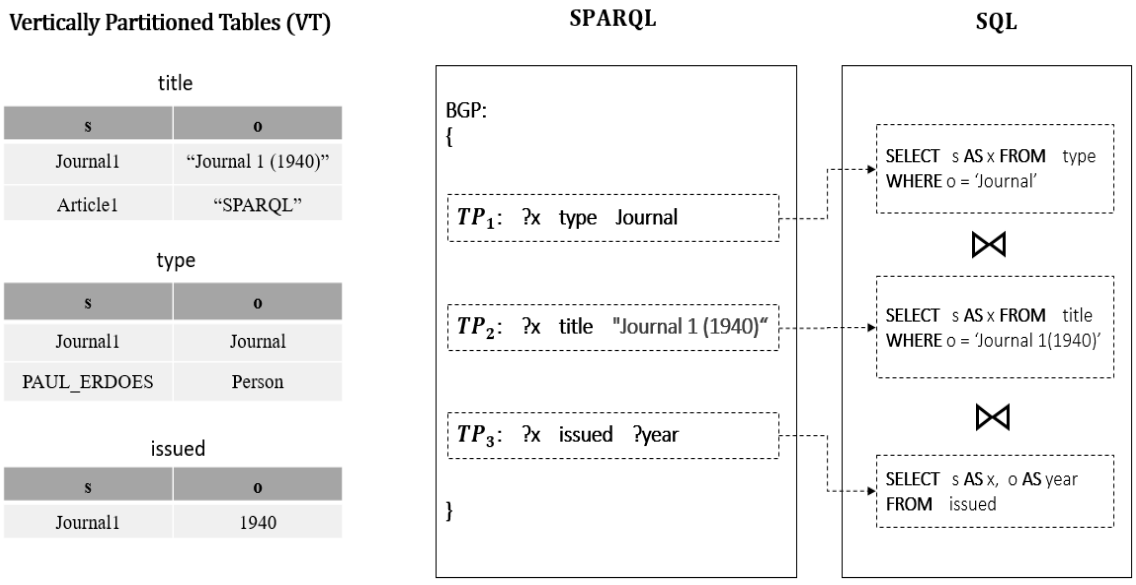


Figure 5: VT schema for RDF Graph G_1 and associated SQL translation of Q_1.

The main disadvantage of this approach is that some partitions can result in a massive portion of the entire graph and cause many I/O. Another shortage for the VT schema is that multiple tables are scanned to return information related to a single entity. On the other hand, it is also observed that the typical SPARQL query is predicate-bound; thus, VT schema enables you to scan only one VT table to find matching values for a certain triple pattern. In addition, particular subjects can be retrieved quickly since each of the n tables is indexed by subject. Last but not least, for the case of multi-valued attributes, this approach stores each distinct value in a successive row in the table for that property [30].

2.4 RDF data partitioning

Nowadays, as the size of RDF data grows, it becomes infeasible to handle large RDF datasets on a centralized system. This raises the need for distributed approaches. For this purpose, big data engines, i.e., Spark, are used to perform distributed parallel processing on a cluster of machines. However, big data engines require data partitioning, which is the main challenge towards distributed RDF processing. Because the data transfer required for distributed query execution, such as joins of intermediate query results from two or more partition nodes.

In particular, we store the RDF data in the relational tables, which are partitioned and distributed over the Spark cluster nodes. When a task is executed in Spark, data shuffling is happening, especially for data-joining operations. Spark shuffle is a costly operation since it moves the data between executors or nodes in a cluster. Therefore, we need to select the partitioning technique wisely as it impacts the query execution performance.

In our experiments, we use Spark-SQL, where join operations require the join keys to be in the same node; thus, we use the join key as a partitioning key in partitioning processes. Consequently, we implement three various partitioning techniques on RDF data, such as Horizontal Partitioning (HP), Subject-based Partitioning (SBP), and Predicate-based Partitioning (PBP) (see Figure 6).

- *Horizontal-Based Partitioning (HP)* requires partitioning the data horizontally into even size of chunks. The number of partitions, n, is an independent variable in the HP approach. It is the best practice to define the n based on the number of machines of the cluster, which optimizes the parallelization during the running process. For example, if you have a relational table with 100 rows and n is 5, as a result, you will have five partitions with 20 rows.

- *Subject-Based Partitioning (SBP)* partitions the data on the *subject* as a key. It applies hash-based locality based on the subject key. Consequently, all triples with the same subject are clustered on the same partition.

- *Predicate-Based Partitioning (PBP)*, similar to the SBP technique, applies par-

titioning based on a key. However, this approach uses *predicates* as a partitioning key. Thus, each triple of the table is distributed according to the hashed value of its predicate leading all the triples that have the same predicate to reside on the same partition.

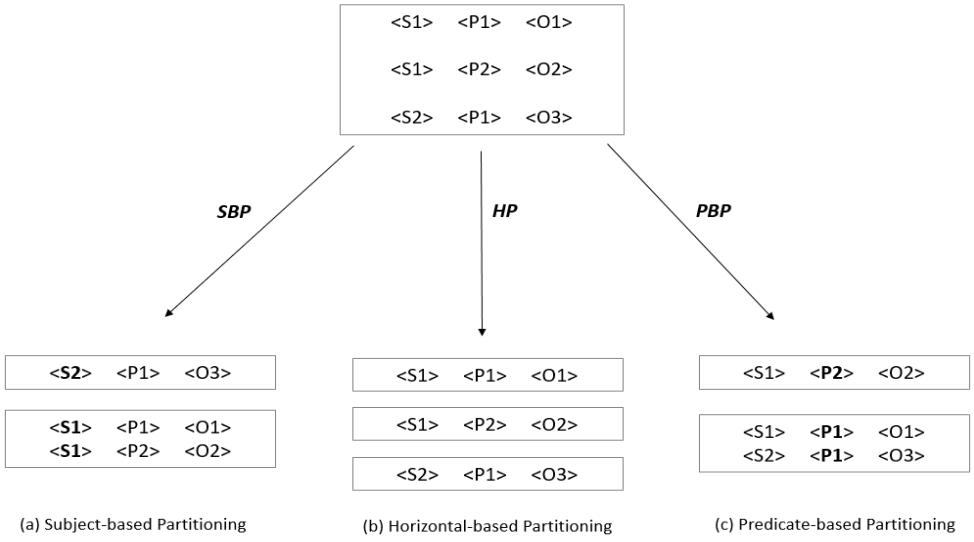


Figure 6: RDF Partitioning techniques.

It is worth mentioning that there are several other existing techniques for data partitioning. However, we selected the three most common methods which are suitable for our experiments to work within the Spark-SQL framework. Indeed, the other techniques like hierarchical partitioning [15] would require some redesign to fit our relational-based data processing scenario. Therefore, we have selected these three pure RDF-based partitioning methods among seven discussed in [15] and applied them on all three various relational schema tables.

2.5 Data Storages

Spark-SQL supports operating on various storage backends to read and write data through the DataFrame interface. In our experiments, we used two storage backends such as *Apache Hive* and *HDFS*. For the latter one, we use various file

formats such as *CSV (Comma-separated values), ORC (Optimized Row Columnar), Avro* and *Parquet*.

Hive [31] is an open-source distributed data warehousing database that was built on top of Hadoop. The Hive was built for reading, writing, querying, and analyzing large datasets residing on HDFS. Similar to RDBMS, Hive stores the data in the form of tables and makes operations using its SQL interface, HiveQL. The main advantage of Hive is that it scales horizontally, handling large volumes of data.

CSV file [32] is a widely supported data exchange format which uses a comma to separate values. Each line of a CSV file is a data record that consists of one or more fields, and each record has an identical list of fields which provides a straightforward information schema.

ORC file can store data in an optimized way reducing the original size of the data up to 75%. Thus, this also affects data processing speed resulting in better performance than Text, Sequence, and RC file formats [33]. ORC file stores statistics (Min, Max, Count, Sum) and has a lightweight index. Although the ORC file increases CPU overhead that increases the time to decompress the relational data, ORC format takes less time to access the data and improves performance, especially when Hive is processing the data.

Apache Avro is a language-neutral file format, namely, it supports various data formats that can be processed by multiple languages [33]. Avro enables you to store complex objects by encoding the schema of its contents, and it allows to modify (remove-add column) the data schema that is in JSON format. Avro does not require any prior knowledge about the schema in order to write any data into a file, and the schema of data is stored in a data file for further processing. In addition, as a result of data serialization, the resulting Avro data is lesser in size.

Parquet file format is a columnar format which is very useful when your data processing framework needs to read just a single column quickly without reading whole data [33]. Like ORC file, Parquet is also great for compression as it improves the query performance especially when querying data from a specific column. The main advantage of the ORC file is that it reduces a lot of I/O cost to make great read performance.

2.6 Relational algebra

SPARQL is the only standard graph query language used to extract information from RDF datasets. The SPARQL query consists of triple patterns which are matched against a graph G, and matching values are processed to give an answer. There are three processing phases of the SPARQL query. First, the *pattern matching* phase chooses the data source to be matched by a pattern and includes several pattern matching features such as *optional* parts, *union* of patterns and *filtering* of possible matched values. Second, the *solution modifiers* phase enables to modify the output of pattern by applying operators like *distinct, order, limit, offset, and projection.* Finally, the *output* can be different depending on type of queries: *SELECT, ASK (yes/no), CONSTRUCT*, and *DESCRIBE* queries [34].

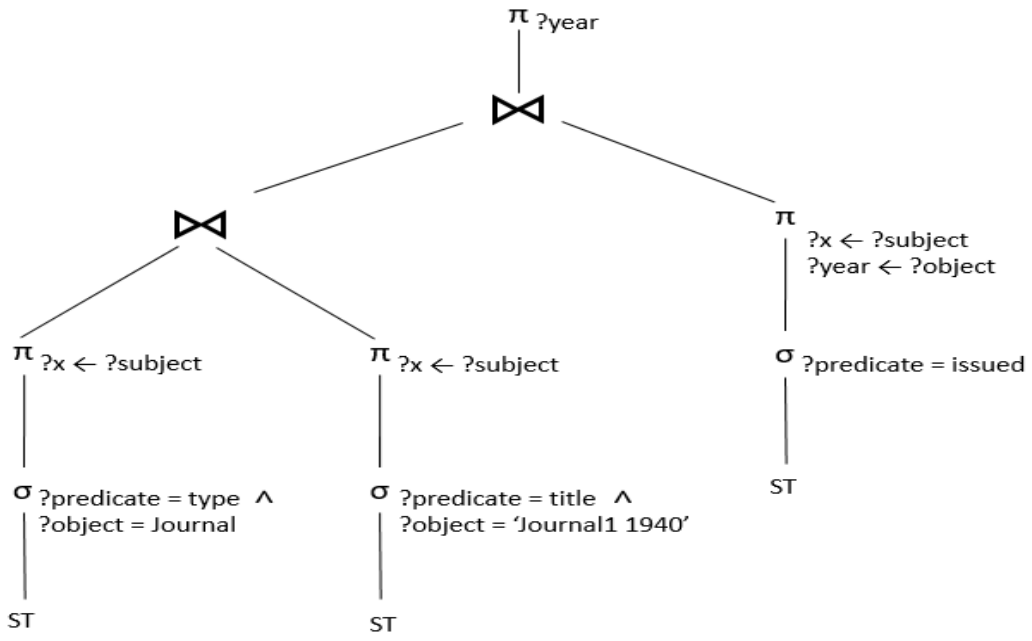


Figure 7: Relational tree of the SPARQL query Q_1 for ST schema

The SPARQL queries used in our experiments cover SPARQL operators such as *UNION, OPTIONAL, FILTER, AND (denoted as ".")* akin to relational *unions, left outer joins, selections,* and *inner joins,* respectively. Our queries also cover the

solution modifiers such as *LIMIT, OFFSET, ORDER BY,* and *DISTINCT*, which can impact the query execution plan. As the RDF data is stored in relational tables and Spark-SQL is used as a query engine in our experiments, we need to transform SPARQL queries into relational algebra that is used to generate SQL queries.

The *relational algebra* [35] is an intermediate language commonly used in the database field for the expression and interpretation of queries. The main benefit of using relational algebra is that it makes a large body of work on query planning and optimization available to SPARQL implementers. The term tuple is commonly used in relational algebra, which is a partial function to map query variables into RDF terms. For instance, Figure 7 represents the relational tree for ST schema, which we build transforming each tuple of SPARQL query Q_1 into relational algebra using operators like *selection (σ), projection/rename (π),* and *inner-join ($\bowtie$).* Relational trees for VT and PT schemes are shown in Figure 8 and 9, respectively.

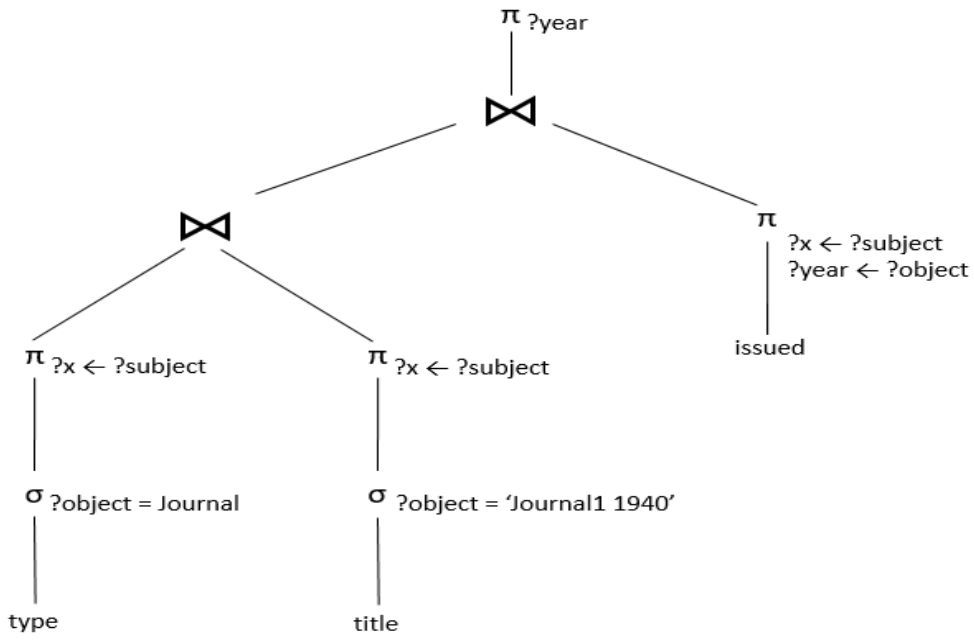


Figure 8: Relational tree of the SPARQL query Q_1 for VT schema

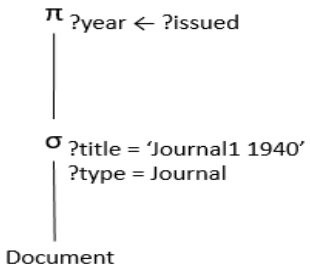


Figure 9: Relational tree of the SPARQL query Q_1 for PT schema

Projection and Rename (π). The projection operator restricts the selection of attributes into smaller subsets. The rename operator renames an attribute. To keep our relational tree smaller, we combine projection and rename operators into *projection/rename (π)* operator. The expression

$$\pi_{?\text{s}\leftarrow\text{subject}}\ (\text{R})$$

contains only the ?subject attribute of relation R, renamed to ?s.

For example, the expression

$$\pi_{\substack{?\text{x}\leftarrow\text{subject}\\ ?\text{year}\leftarrow\text{object}}}\ (\text{R})$$

contains the ?subject and ?object attributes of relation R, renamed to ?x and ?year, respectively. The expression will be translated to this

SELECT R.subject AS x, R.object AS year FROM (...) AS R;

And, the SQL translation of R will be in the brackets.

Selection (σ). Selection operator selects only those tuples for which a propositional formula holds.

For instance, the expression

$$\sigma_{?predicate=issued}\ (\text{R})$$

accepts all RDF tuples with a *issued* value on *predicate* attribute. This operator translates the expression to

*SELECT * from (...) AS R WHERE R.predicate = issued*

which will be used in the WHERE clause of a SELECT statement.

Inner Join ($\bowtie$). Join operator joins two tuples based on their shared attributes. The SQL translation of this expression:

$$\pi_{?x}\ (\mathrm{R}_1) \bowtie \pi_{?x,?year}\ (\mathrm{R}_2)$$

looks like this:

```
SELECT R1.x, R2.year
FROM (...) AS R1
NATURAL JOIN (...) AS R2;
```

The SQL translations of R_1 and R_2 go into the brackets. This SQL translation works only if the shared variable is always bound on both sides. Because a *NULL* in an SQL join causes the row to be rejected, but an unbound variable in a SPARQL join does not [35]. A SPARQL join rejects only rows where the variable is bound to different values on both sides. Therefore, for instance, if the variable might be unbound on the left side, then the SQL translation would be like this:

```
SELECT R2.x, R2.year
FROM (...) AS R1
JOIN (...) AS R2
ON (R1.x=R2.x OR R1.x IS NULL);
```

Left Outer Join ($⟕$). Additionally, the result of a left outer join contains all tuples from the first relation that have no matching in the second relation. The translation of the left outer join is analogous to the inner join.

$$\pi_{?x}\ (\mathrm{R}_1) ⟕ \pi_{?x,?year}\ (\mathrm{R}_2)$$

If ?x is always bound on both relations:

```
SELECT R1.x, R2.year
FROM (...) AS R1
NATURAL LEFT JOIN (...) AS R2;
```

If ?x may be unbound on the left:

```
SELECT COALESCE(R1.x, R2.x) AS x, R2.year
FROM (...) AS R1
LEFT JOIN (...) AS R2
ON (R1.x=R2.x OR R1.x IS NULL);
```

2.7 RDF Graph Serializations

RDF Graphs can be serialized in many ways. The default RDF serialization from W3C is RDF/XML[4]. However, it supports other serialization formats, e.g., Turtle[5] (TTL), N-Triples[6] (NT), Notation3[7], and JSON for Linked Data (JSON-LD)[8]. Despite the popularity of RDF/XML and JSON-LD, in this thesis, we adopt N-Triples as the default serialization because it makes serialization and parsing highly performant.

RDF/XML is the first RDF serialization format and seemed like a logical default format since many systems were able to handle XML when RDF was invented. In particular, RDF/XML is the combination of different concepts: a tree-like document and a triple-based graph. This makes the RDF/XML conceptually verbose and causes confusion. Listing 1 shows an example of RDF/XML format.

Unlike RDF/XML, the *Notation3* closely resembles the subject, predicate, object model of RDF. N3 uses *@prefixes* which makes it easier to understand how RDF works. However, N3 is relatively costly to serialize, which could hinder performance.

[4]https://www.w3.org/TR/rdf-syntax-grammar/
[5]`https://www.w3.org/TR/turtle/`
[6]`https://www.w3.org/TR/n-triples/`
[7]https://www.w3.org/TeamSubmission/n3/
[8]`https://www.w3.org/TR/json-ld11/`

The simpler subset of N3 is *Turtle*. Turtle syntax allows grouping triples with the same subject separating different predicates using a semicolon and . Turtle syntax allows separating alternative objects for the same predicate using a comma. Compared to N3, Turtle is more popular, and it is easier to find libraries for it. However, it is still costly to parse Turtle compared to N-Triples. Turtle is recommended if you need to edit RDF by hand since it is highly human-readable. Listing 2 represents an example of Turtle serialization.

Listing 1: Example illustrating RDF/XML Syntax for RDF triples

```
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:dcterms="http://purl.org/dc/terms/">

<rdf:Description rdf:about="http://localhost/publications/jo-
urnals/Journal1/1940/">
  <dcterms:issued>1940</dcterms:issued>
  <dcterms:title> Journal1(1940)</dcterms:title>
</rdf:Description>

</rdf:RDF>
```

Listing 2: Example illustrating Turtle Syntax for RDF triples

```
@prefix tim: <http://localhost/publications/journals/Jour-
nal1/1940/>.
@prefix dcterms: <http://purl.org/dc/terms/>.

<tim> dcterms:issued "1940"^^<http://www.w3.org/2001/XMLSc
hema#date>.
<tim> dcterms:title "Journal1(1940)"^^<http://www.w3.org/20
01/XMLSchema#string>.
```

The more simpler subset of Turtle is *N-Triples*. N-Triples does not support *@prefixes*, which makes it trivial to parse or serialize. Consequently, it makes parsing and serialization highly performant. In addition, Turtle/N3 parsers know how to deal with N-Triples since N-Triple is a subset of Turtle and N3.

Obviously, *JSON-LD* is the most popular way to serialize data in web applications. JSON-LD is an extension of JSON, which enables turning regular old JSON data into RDF by simply adding *@context*. JSON-LD contexts allow developers to use simple and locally-meaningful names while maintaining the meaning of the data across all users and communities. JSON-LD is still valid JSON and easy to read for those new to RDF. JSON-LD supports well both RDF and JSON; however, it is still difficult and costly to parse if you need the RDF data instead of the JSON object. Listing 3 and 4 demonstrates the examples of N-Triples and JSON-LD serializations for RDF triples, respectively.

Listing 3: Example illustrating N-Triples Syntax for RDF triples

```
<http://localhost/publications/journals/Journal1/1940/>
<http://purl.org/dc/terms/issued>
"1940"^^<http://www.w3.org/2001/XMLSchema#date>.

<http://localhost/publications/journals/Journal1/1940/>
<http://purl.org/dc/terms/title>
"Journal1(1940)"^^<http://www.w3.org/2001/XMLSchema#string>.
```

Listing 4: Example illustrating JSON-LD Syntax for RDF triples

```
{
 "@context": {
    "dcterms": "http://purl.org/dc/terms/"
 },
 "@id": "http://localhost/publications/journals/Journal1/1940/",
 "dcterms:issued": "1940",
 "dcterms:title": "Journal1(1940)"
}
```

3 Problem Statement

In this chapter, we formulate the problem that we addressed in this thesis (in Section 3.1) and define research questions for Macro, Meso, and Micro levels of the framework (in Section 3.2).

3.1 Problem Context

The amount of data generated every day opens new challenges for data systems. Big data, as identified by Laney [1], is a term used to identify three challenges: volume, variety, and velocity.

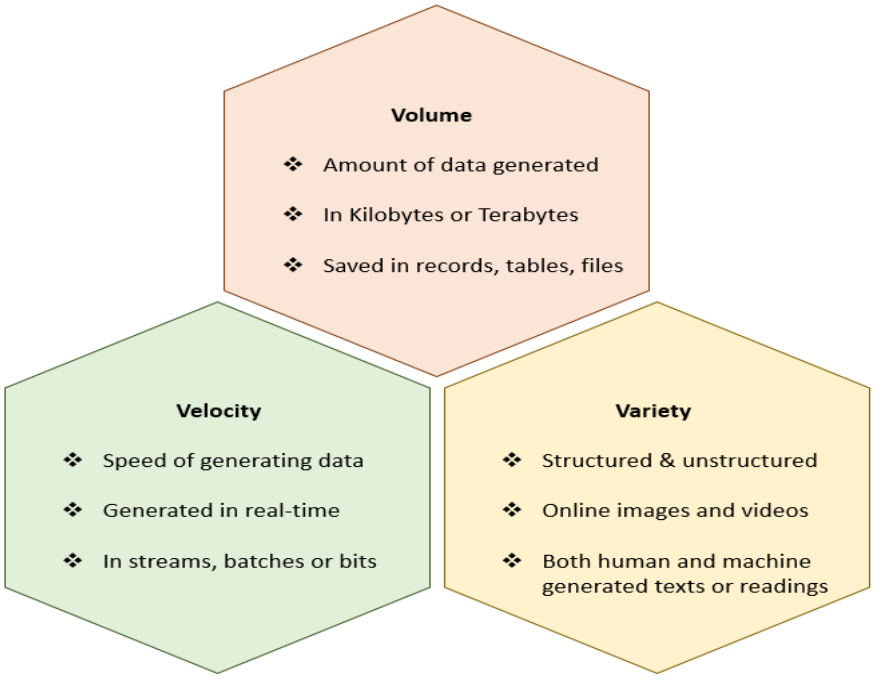


Figure 10: Big Data challenges.

Volume is the size of data that is produced. Since the amount of data generated every day increases, it creates problems for data systems to process and integrate the overwhelming volumes of data using traditional approaches. To process entire

volumes of the generated data requires big storage, which makes it crucial for data systems to scale in terms of storage and memory. For instance, relational database management systems (RDBMS) scale up with the expensive hardware. However, RDBMSs cannot scale out with commodity hardware in parallel, which is insufficient for handling the ever-growing data volume [36]. Therefore, we consider horizontally scalable big data engines like Hadoop [9], and Apache Spark [10].

Velocity is related to the speed with which data is being generated and processed. While accepting the incoming flow of data generated, it is also necessary to process the incoming data quickly. Previously, companies were using batch processing to analyze the data. Considering the latency, the result of that approach might be useful if the incoming data rate is slower than the batch processing rate [37]. However, the data is now being streamed in real-time through the server and the outcome is beneficial if the delay is very short. For example, fast data analysis will easily alert companies to stocking problems; thus, the issue can be fixed before it gets worse. Namely, data systems should provide high throughput. Therefore, stream processing systems (such as Apache Flink [38], Apache Storm [39], Apache Spark) are used to resolve the problem of data velocity by conducting continuous computation of incoming data.

Variety refers to heterogeneity of data. A few decades ago, data was collected in one format and stored in a structured database in a simple text file. However, data now can be generated by humans or machines in different forms like structured, semi-structured, or unstructured. For instance, companies consider email messages, photos, videos, audio recordings, documents, books, tweets, customer comments, or any other kind of data on social media that can help them to understand consumer sentiment better. Therefore, data can be extremely diverse, and it requires more work to decipher it to get insights. Consequently, data systems should perform data integration and universally accepted formats, such as XML [11] and RDF, are used to handle data variety issue.

In this thesis, we address the volume and variety dimensions of big data. RDF is a

[9] `https://hadoop.apache.org/`

[10] `https://spark.apache.org/`

[11] `https://www.w3.org/XML/`

schema-free data model that W3C recommends to handle a variety of big data. RDF allows combining structured and semi-structured data and sharing across different applications. There are several native triplestores (such as RDF-3X [3], RDF4j [13], Apache Jena [14], gStore [4], Sesame [5]) for handling RDF data sets. However, these RDF systems do not scale horizontally and can not handle large data sets as required by many applications, i.e., Yago [12], Wikidata [13]. Therefore, we need an alternative solution to handle both variety and volume of data.

Although big data platforms are not originally designed for RDF processing, they were used successfully to run online analytical workloads for large-scale relational data processing. Thus, we consider using a relational schema to store and process RDF data sets on Spark, a de facto standard big data framework.

3.2 Research Questions

To formalize our research questions, we use the *Macro-Meso-Micro* framework [40], which allows formulating research questions at three levels of analysis.

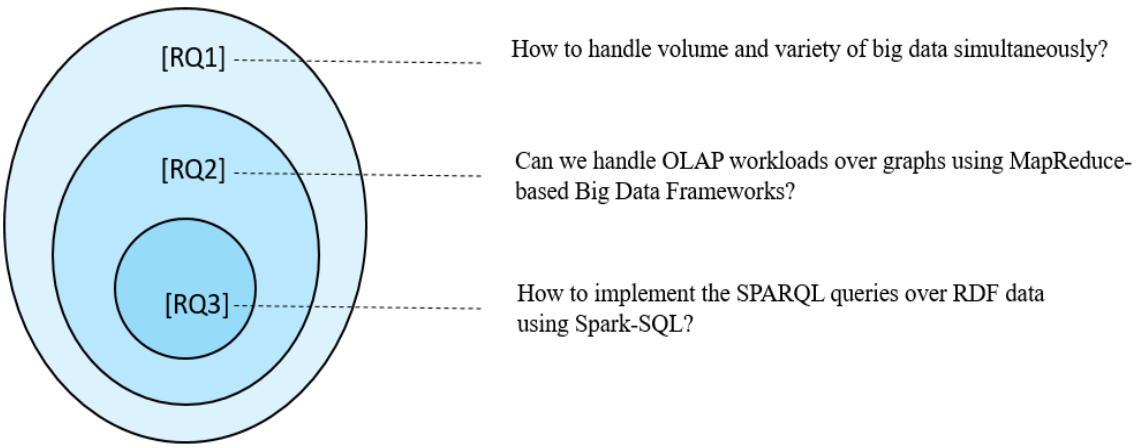


Figure 11: Macro - Meso - Micro framework.

[12]https://yago-knowledge.org/
[13]https://www.wikidata.org/wiki/Wikidata:Main_Page

The **Macro** level is the biggest unit of analysis and generally related to complex and broad questions that cannot be solved but capture a vision. At this level, we formulate the research question, which indicates that we will contribute to the volume and variety issues of big data.

RQ1: *How to handle volume and variety of big data simultaneously?*

Since the Macro level question is too broad to be answered, we list some problem-related requirements which help us to narrow down the problem to the Meso level, which is more specific (in Section 3.2.1).

The **Meso** level reveals connections between the macro and micro levels, and it refers to medium scale analysis.

RQ2: *Can we handle OLAP workloads over graphs using MapReduce-based Big Data Frameworks?*

The research question we formulate in Meso level is more specific since we made requirements specification in the Macro Level. However, the question at this level is still very complex to answer. Therefore, we analyze alternative solutions for each requirement, which leads us to the Micro level where we can answer the question (in Section 3.2.2).

The **Micro** level is the smallest unit of analysis where the actual research questions are defined and feasible problems are presented.

RQ3: *How to implement the SPARQL queries over RDF data using Spark-SQL?*

In this level, we explain why we selected specific solutions to answer the research question. Finally, this level of analysis allows us to design our experiments which will contribute to the main challenges of big data such as volume and variety (in Section 3.2.2).

3.2.1 From Macro to Meso

RQ1 is a Macro level question that is too broad to be answered in the scope of a thesis. However, we can use the question to narrow the problem by providing requirements specifications. In this section, we present the requirements that lead to the Meso question, *RQ2*.

R1 *Flexibility*

The solution should provide a flexible data model for handling both structured and unstructured data integration.

R2 *Semantic Interoperability*

The solution should provide a data model that includes semantic information to the data and the relationships that lie between them.

R3 *Online analytical workloads*

The solution should handle online analytical workloads.

R4 *Fault-tolerance*

In case of any node failure, the system should be able to recreate the data and continue operating tasks without interruption.

R5 *Scalability*

The solution should provide a horizontally scalable big data engine for large-scale data processing.

Specifically, **R1** and **R2** are related to data model. To address the data variety challenge, we should select a flexible data model that can integrate data in any format such as structured, semi-structured, or unstructured. The data model should also provide semantic meaning to the data, which enables the transmission of the meaning with the data. Generally, this is done by adding metadata and linking each data element to a controlled, shared vocabulary, thus, you can uniquely identify things that are distributed on the web. We can address these requirements by considering the graph data models like RDF and Property Graphs.

R3 requires the solution to handle OLAP workloads. Big data analytics is a key factor for successful businesses. Online analytical processing (OLAP) is designed to get usable information from data by allowing data aggregation, which helps organizations move smarter and increase profit. On the other hand, OLAP is valuable because it is flexible and allows to analyze the data based on various dimensions and facts. Therefore, we consider using big data frameworks since they support OLAP.

R4 and **R5** are system-related requirements. The system should be able to handle the scalability of both data and workload. Big data frameworks with *Shared-Nothing architecture* (interconnecting several independent processors) is the most suitable solution for scalability requirements of both data and workload. Last but not least, the system should also be fault-tolerant against any node failure. If any failure happens during the data processing, the system should recreate the data and continue operating tasks without interruption. These requirements prompt us to focus on horizontally scalable, MapReduce-based big data frameworks that run in a distributed environment in parallel.

3.2.2 From Meso to Micro

In the previous section, we have stated the requirements that allow reformulating the question as RQ2. The Meso level question, RQ2, is still broad to be answered, therefore, we can make an analysis of alternative options for each requirement in order to choose the best solution. Firstly, we need to choose a wise data model according to requirements *R1* and *R2*. Therefore, we will make a comparison between *XML*, *Property Graphs*, and *Resource Description Framework (RDF)*.

XML is one of the popular web standards for data exchange on the web. It deals with hierarchical, self-contained documents, and provides a serialization format to encode the information, thus the data can be parsed easily in the destination machine [41]. Although XML allows users to define their own structures, it does not provide any standard mechanism to interpret data in a formal and structured way [42]. Therefore, XML does not meet the requirement *R2*.

Property Graph is a reliable model performing scalable graph analytic tasks. It

provides efficient storage and flexibility to model various real-world domains. However, we can not neglect *RDF* graphs as a very popular and usable model for representing tons of data on the web. RDF is a *flexible* and W3C standard data model for data exchange on the web. RDF focuses on the meaning of data [43]. RDF makes use of URI (Universal Resource Identifier), which allows you to uniquely identify things that are distributed on the web. RDF makes it possible to integrate data from different sources without any custom programming [44], which indicates that RDF also meets the requirement *R2*. Last but not least, the RDF data model can also be represented in various serializations such as Turtle, N3, RDFa (RDF embedded in HTML), RDF/XML, JSON-LD. As a result, elicited requirements prompt us to focus our research on the RDF data model.

Regarding *R3*, we consider Graph Queries and Graph Algorithms. Graph queries define OLAP workloads in which queries touch on a large part of the graph to collect information and also perform complex aggregations on the data [45]. We can write OLAP workloads using SPARQL queries. SPARQL is a query language recommended by W3c to retrieve and manipulate the RDF data.

Graph Algorithms perform computations on the graph to measure certain aspects of a graph such as Shortest Path, Connected Components, PageRank, and many more. These workloads consider the connections or transitive paths in the entire graph. Thus, OLAP Algorithms are more expensive to run than OLAP queries and are often evaluated in batches [45].

We perform analysis of requirements *R4* and *R5* on massive parallel processing frameworks such as *Hive* and *Spark-SQL*.

Apache Hive is a data warehousing tool built on top of Apache Hadoop. Hive stores the data on Hadoop Distributed File System (HDFS) and provides an SQL-like interface - HiveQL, to read, write, and manage large datasets residing in distributed storage [31]. HDFS is a Filesystem of Hadoop designed to store large files on a cluster of machines. It replicates the partitions of data, making Hadoop fault-tolerant against any node failure. HiveQL queries are mapped into MapReduce jobs executed on Hadoop. MapReduce [46] is a processing model to execute parallel and distributed analytical tasks. However, MapReduce performs read and write operations only on a

Table 1: Table representing how the requirements are mapped to alternatives options.

	R1	**R2**	**R3**	**R4**	**R5**
XML	✓	✗	✓	✓	✓
Property Graph	✓	✓	✓	✓	✓
RDF	✓	✓	✓	✓	✓
OLAP Algorithms	✓	✓	✓	✗	✗
OLAP Queries	✓	✓	✓	✓	✓
HIVE	✓	✓	✓	✓	✓
Spark-SQL	✓	✓	✓	✓	✓

hard drive.

Spark is the de facto standard big data engine, which is used to handle vast amount of data in a distrubuted environment. Similar to Hive, Spark can run on a cluster of nodes processing the data in parallel. This nature of the Spark allows it to present high robustness against excessive data loads. Spark also has more user-friendly APIs that can be written in Scala, R, Java, and Python. While Hive stores data on hard drives, Spark allows executing parallel computations in memory [22]. In-memory computations improve the speed of read and write operations as well as query execution performance. Compared to Hive, Spark is more efficient for applications that require frequent reuse of data sets across multiple parallel operations. Spark mainly depends on two distributed main-memory data abstractions: (i) Resilient Distributed Data sets (RDD), a distributed memory data abstraction which allows performing in-memory computations in a fault-tolerant manner and (ii) Data Frames (DF), a compressed and schema-enabled data abstraction. Unlike an RDD, data in DF is organized into named columns, like a table in a relational database. This nature of DF makes large data sets processing easier, allowing higher-level abstraction. Both abstractions facilitate the programming operations by supporting various relational operators such as *join, selection,* and *project* that are not natively supported in Hadoop. These operators enable the translation and processing of high-level query expressions (e.g., SQL, SPARQL) using any programming language supported by

Spark, e.g., Java, Scala, or Python.

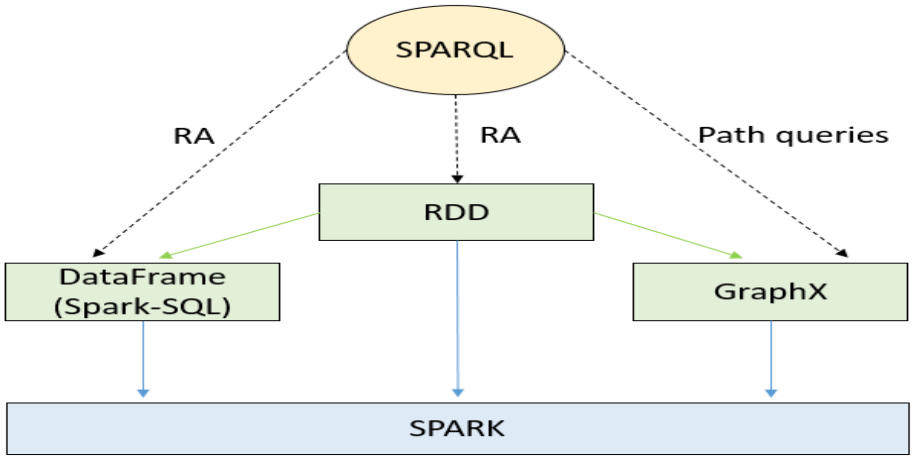


Figure 12: Spark's Abstracts

Furthermore, Spark proposes two higher-level APIs, GraphX and Spark SQL (see Figure 13). Spark GraphX API [47] enables graph-parallel computations through an extension of Spark's RDD. Like Pregel [48], GraphX follows a vertex-centric computation model which is dedicated to performing graph algorithms such as triangle counting and Pagerank. In addition, the authors [49] show that the RDF data processing on GraphX results in poor performance due to a large number of intermediate results. Spark-SQL allows for running queries on DataFrame by using its optimizer, Catalyst [8]. Catalyst makes logical and physical optimization on query plans and speeds up the query execution process on DataFrames, while RDDs does not perform any join optimization. The other benefit of using the DataFrame with Spark-SQL comes from the columnar, compressed in-memory representation of DF. Firstly, it allows for processing larger data sets compared with RDD. Secondly, DataFrame compression saves the data transfer cost between the nodes. Consequently, the best option for querying RDF data in Spark is to use Spark DataFrames API and Spark-SQL engine, which provide additional optimizations on the query plans.

Given that all the elicited requirements are complied with, we focus our analysis on RDF and Spark-SQL, formulating the Micro level research question, RQ3.

To handle OLAP workloads, we opt for Graph Query Workloads, since most pat-

tern matching operators can be expressed in relational algebra. Therefore, using operators such as selection, aggregation, and joins, we can translate SPARQL queries into the relational tree, which can be translated into SQL queries afterward. Furthermore, we need to choose a distributed scalable system (such as Hive and Spark-SQL), which allows us to execute queries in a distributed parallel manner. Spark-SQL has a better performance in latency than Hive because Spark is an in-memory processing big data engine. Namely, Spark keeps the data in random access memory(RAM), and the data is processed in parallel. Therefore, writing SPARQL Query Workloads on the RDF data model using Spark-SQL is the best solution to run our experiments.

4 Experiment Design and Analysis

In this chapter, we explain how we designed our experiments, and we describe the analysis methodology that we follow to analyze the findings of our experiments.

Particularly, in Section 4.1, we present the Sp^{2}Bench Benchmark dataset, queries, and how we designed the various experiments. Section 4.2 presents the approach for advanced analysis of experiments.

4.1 Design and Implementation

In the previous chapter, we defined research questions where the Micro-level question prompted us to implement SPARQL queries over RDF data using Spark-SQL. To answer the Micro-level question, we design our experiments addressing its challenges.

We use Sp^{2}Bench SPARQL Performance Benchmark to compare efficient storage techniques for RDF and evaluation strategies for SPARQL queries. In particular, to run SPARQL queries with Spark-SQL, we adopt SQL translations [14] of SPARQL queries provided by Sp^{2}Bench Benchmark according to our experiments. Spark-SQL works with Spark's DataFrame API, which is similar to relational tables in RDBMs. Therefore, we store RDF data in different relational tables, since there exist several relational RDF schemes, i.e., Single Statement Table (ST), Vertically Partitioned Tables (VT), and Property Tables (PT) (cf. RDF relational schemes 2.3).

Like other big data processing frameworks, Spark also requires data partitioning, which is the main challenge towards distributed RDF processing. Besides using relational tables, we also need to select a wise partitioning technique as it impacts the query execution performance. In Spark-SQL, join operations require the join keys to be in the same node, and we use the join key as a partitioning key in partitioning processes. Consequently, we implement various partitioning techniques on RDF tables, such as Horizontal Partitioning (HP), Subject-based Partitioning (SBP), and Predicate-based Partitioning (PBP) (cf. RDF partitioning 2.4).

Last but not least, since Spark supports many data formats, we also consider

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

storing RDF data in various storage backends and data formats (i.e., Hive, CSV, Avro, ORC, Parquet) for making an efficient comparison (cf. Storage backends 2.5).

In summary, to answer the Micro level question, we implement different experiments on Spark-SQL variating:

- Relational schemes (ST, VT, PT)
- Partitioning techniques (HP, SBP, PBP)
- Storage backends and data formats (Hive, CSV, Avro, ORC, Parquet).

In the remainder of this chapter, Section 4.1.1 and 4.1.2, respectively, introduce the Sp^{2}Bench Benchmark Dataset and how we applied different partitioning techniques on the RDF tables. Section 4.1.3 presents how the generated RDF data are converted from N-triples format into various Relational RDF tables. Section 4.1.4 shows the SPARQL and SQL queries that are used to conduct experiments and the complexity level of them. Finally, Section 4.1.5 describes how we converted and stored the RDF tables in different data storage backends and file formats.

4.1.1 SP^{2}Bench Benchmark Datasets

In our experiments, we used SP^{2}Bench [16], a publicly available SPARQL performance benchmark. Its RDF data is settled around the Digital Bibliography & Library Project (DBLP) and includes information about publications and their authors. SP^{2}Bench meets the key requirements postulated in the Benchmark Handbook [50], namely, it is Scalable, Portable, Relevant, and Simple. In particular, SP^{2}Bench is *scalable* because it has a data generator that allows creating arbitrary large RDF datasets. It is *portable* since its data generator is platform-independent. It provides a set of SPARQL queries and SQL translation of the SPARQL queries for each of the relational schemas we selected. These queries have different complexities and implement realistic requests by combining operators such as Optional, and Filter [51]. Thus, SP^{2}Bench is also *relevant*. Last but not least, SP^{2}Bench contains bibliographic

information about Computer Science, and its queries are compliant to diverse optimization strategies, which makes it a *simple* and understandable benchmark for us, which can be considered domain experts. Moreover, SP^{2}Bench has a similar nature to real-world RDF datasets since it has a low-level structuredness [51].

Using SP^{2}Bench data generator [15], we generated three data sets of different sizes having 100M, 250M, and 500M triples in n-triple format. We conducted our experiments on all three data sets to observe the linearity of results conformance. We analyze the experimental results in Chapter 5.

4.1.2 Data Partitioning

We run our experiments in a distributed environment, and the data is partitioned and distributed over the Spark cluster nodes. When a task is executed in Spark, data shuffling is happening, especially for data-joining operations.

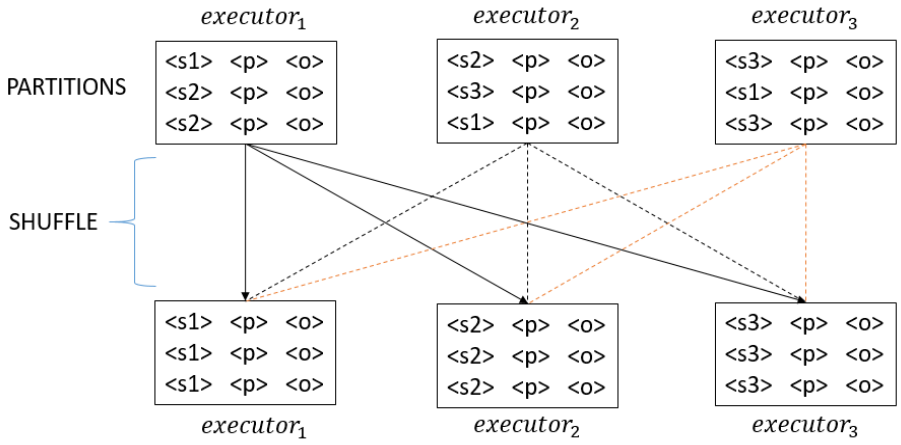


Figure 13: Spark Shuffle

Particularly, a shuffle occurs when a transformation requires information from other partitions, i.e., projection of all triples with the same subject value. Spark shuffle is a costly operation since it moves the data between executors or nodes in a

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

cluster [52]. Since data partitioning impacts the query execution performance [53], it is crucial to select the partitioning technique wisely.

In section 2.4, we have explained the partitioning techniques that we use in the experiments, i.e., Horizontal Partitioning, Subject-based Partitioning, Predicate-based Partitioning. These techniques were designed for RDF data, and we applied them for the relational RDF schemes. Given the number of partitions, n, the Horizontal-based Partitioning technique requires partitioning the data into the same size of n chunks. The Subject-based Partitioning technique makes partitioning based on the *subject* of RDF triple and clusters all the triples with the same subject on the same node. Similarly, the Predicate-based Partitioning technique makes partitioning based on the *predicate* of RDF triple, which leads all triples with the same predicate to reside on the same node.

Data preparation occurred in two phases. First, we used custom Spark partitioning techniques for creating partitioned DataFrames. The partitioning key was selected subject, predicate, or horizontal depending on the choice of partitioning technique, SBP, PBP, or HP, respectively. Then, the partitioned DataFrames were persisted on HDFS. It worth noticing that each partitioned block was replicated on nodes according to the default HDFS replication factor (RF = 3).

4.1.3 Relational schemes

Spark-SQL queries the data on the relational backend, thus, we use RDF relational schemes as the main scenario in our experiments. As we have described in Section 2.3, the three most common relational schemes were selected to store the RDF data, which are Single Statement Table (ST), Vertically Partitioned Tables (VT), and Property Tables (PT). RDF data generated by the SP^{2}Bench generator was in N-triples format. We used Apache Jena [54] for converting the data from N-triples files into various relational tables. Apache Jena takes N-triples data as a source, and a SPARQL query will be executed on the source data and outputs the table for the input SPARQL query (in CSV tables). For that purpose, we defined SPARQL queries based on the schema design of each relational table.

Listing 5: SPARQL Query Q_2 for generating ST table

```
1 SELECT ?Subject ?Predicate ?Object
2 WHERE {
3         ?Subject ?Predicate ?Object.
4 }
```

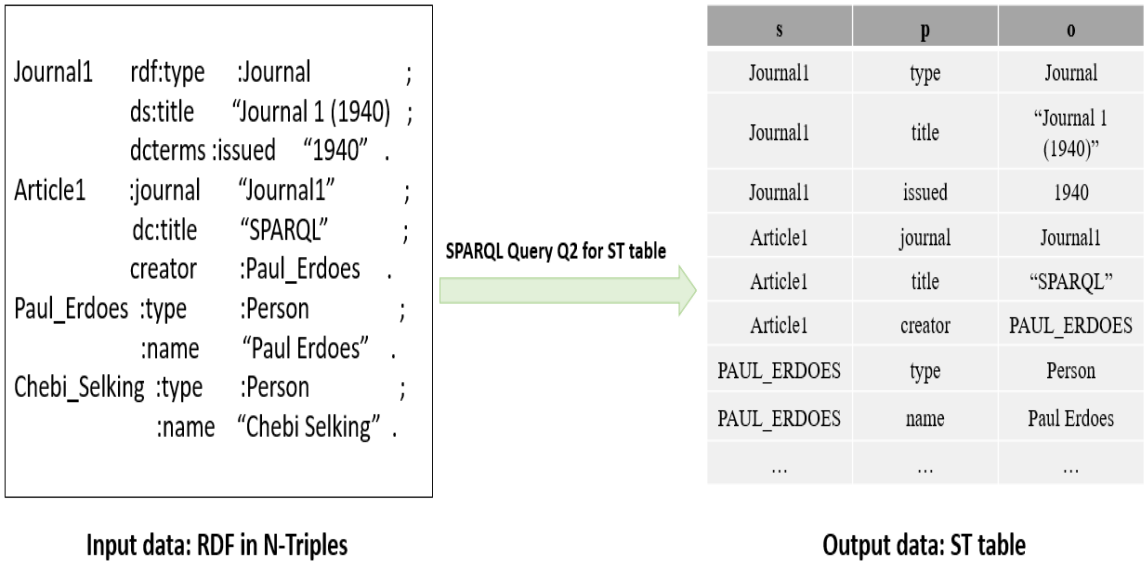


Figure 14: Convert RDF data from N-Triples into ST table. Prefixes are omitted.

For example, Listing 5 represents the SPARQL query Q_2 that is used for generating a Single Statement table. Figure 14 shows that we run the Sparql query Q_2 on N-Triples Serialization of Graph G_1 (see Figure 1), and we get Single Statement Table with three columns, Subject, Predicate, and Object.

Similarly, we use the same approach to generate other VT and PT tables. However, we use more than 1 query for generating VT and PT tables since we have to convert tables through various properties and individuals.

In the experiments, we generated 15 VT tables according to the distinct number of properties that we had in the generated Sp^{2}Bench data. In Figure 15, we show the VT table generation process only for a few of them. The SPARQL queries used to generate Issued, Title, and Type tables are shown in Listing 6, 7, and 8, respectively.

Listing 6: SPARQL Query Q_3 for generating VT table *Issued*

```
SELECT DISTINCT ?subject ?object
WHERE {
      ?subject dcterms:issued ?object.
 }
```

Listing 7: SPARQL Query Q_4 for generating VT table *Title*

```
SELECT DISTINCT ?subject ?object
WHERE {
      ?subject ds:title ?object.
}
```

Listing 8: SPARQL Query Q_5 for generating VT table *Type*

```
SELECT DISTINCT ?subject ?object
WHERE {
      ?subject rdf:type ?object.
}
```

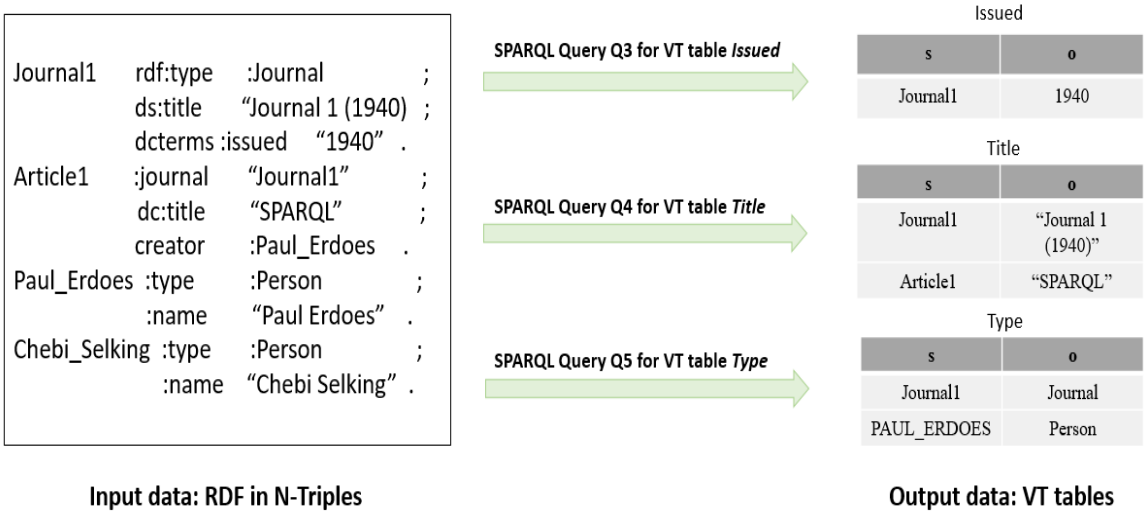


Figure 15: Convert RDF data from N-Triples into VT tables. Prefixes are omitted.

Listing 9: SPARQL Query Q_6 for creating PT table *Document*

```
SELECT  ?journal ?title ?issued ?type
WHERE
{
 { ?journal rdf:type bench:Journal.}

 OPTIONAL {?journal  dc:title ?title}
 OPTIONAL {?journal  dcterms:issued ?issued}
 OPTIONAL {?journal  swrc:type ?type}
}
```

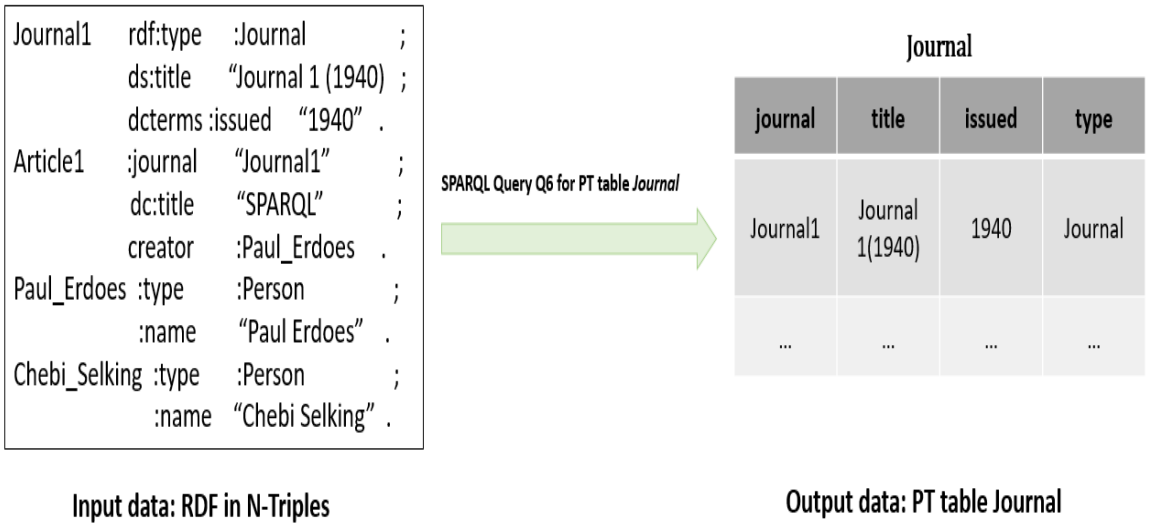


Figure 16: Convert RDF data from N-Triples into PT table Document. Prefixes are omitted.

For PT schema, we generated 13 different tables such as Journal, Author, References, and many more. Listing 9 shows the SPARQL query that is used to generate PT table Journal. We store all triples (where the subject value is a journal) in the Journal table along with their predicates. In the SPARQL query Q_6, *Optional* operator means that the value of predicate can be null.

Figure 16 represents the input RDF data and output PT table Journal that is

generated by SPARQL query Q_6.

Worth noting that the SPARQL queries for generating VT and PT tables are available in our Github repository. [16]

4.1.4 Queries

SP^{2}Bench queries contain diverse SPARQL operators and RDF access patterns. To run the query on Spark, we used SQL translation of the SPARQL queries that are provided for Single Statement Table, Vertically Partitioned Tables, and Property Tables. We evaluated all 11 queries by ignoring query Q_9 for Property tables which is not applicable for PT schema. To compare the complexity of queries, we considered the number of filters, joins, and projected variables for each query. Table 2 summarizes these complexity measures for Sp^{2}Bench SPARQL queries and for SQL translations related to each relational schema (ST, VT, and PT).

Table 2: SP^{2}Bench 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).

	SPARQL			**ST-SQL**		**VT-SQL**		**PT-SQL**	
	#Joins	**#Filters**	**#Projections**	**#Joins**	**#Selections**	**#Joins**	**#Selections**	**#Joins**	**#Selections**
Q_1	2	0	1	2	6	2	2	2	2
Q_2	9	0	10	9	9	9	0	8	5
Q_3	1	1	1	1	2	1	2	2	2
Q_4	7	1	2	7	16	7	3	8	3
Q_5	5	1	2	5	11	5	3	7	3
Q_6	8	3	2	9	13	9	4	6	3
Q_7	12	2	1	12	15	16	5	2	1
Q_8	10	2	1	8	13	8	7	8	4
Q_9	3	0	1	2	4	2	3	n/a	n/a
Q_{10}	0	0	2	0	1	1	2	6	2
Q_{11}	0	0	1	0	1	0	0	2	0

We consider *Join* and *Selection* operations as a complexity measure since they have

[16] `https://github.com/DataSystemsGroupUT/SPARKSQLRDFBenchmarking/tree/master/Datasets`

significant impact on query execution performance. We run large RDF tables on Spark where the data shuffling is happening, especially for data-joining operations. Data shuffling is a very expensive operation as it exchanges data between executors in a cluster. In relational algebra, *selection*, sometimes called a restriction, means filtering rows by eliminating or selecting the rows that will be returned. For instance, the number of variable projections in the SQL statements helps us compare data formats of the storage backends in terms of being row-oriented (e.g., Avro) or columnar-oriented (e.g., Parquet or ORC).

Listing 10 shows the one out of 11 SP^{2}Bench SPARQL queries that we run in our experiments. This query can be interpreted as "Select all articles with property swrc:pages".

Listing 10: SPARQL query Q_7

```
1 SELECT ?article
2 WHERE {
3         ?article rdf:type bench:Article .
4         ?article ?property ?value
5         FILTER (?property=swrc:pages)
6        }
```

To run the SPARQL query Q_7 on the RDF relational tables using Spark-SQL, we used SQL translation of Q_7 for each relational scheme, i.e., ST, VT, PT. Listing 11 shows the SQL translation of Q_7 for Single Statement Table.

Listing 11: SQL translation of SPARQL query Q_7 for ST

```
1 SELECT DISTINCT ST1.subject AS article
2 FROM SingleStmtTable ST1
3 JOIN SingleStmtTable ST2 ON ST2.subject=ST1.subject
4 AND ST2.predicate= 'http://swrc.ontoware.org/ontology#pages'
5
6 WHERE ST1.object='http://localhost/vocabulary/bench/Article'
7 AND  ST2.object IS NOT NULL
```

Listing 12 and 13 represent the SQL translations of Q_7 for VT and PT tables, respectively.

Listing 12: SQL translation of SPARQL query Q_7 for VT

```
1 SELECT DISTINCT T.Subject AS article FROM Type T
2 JOIN Pages P ON T.Subject=P.Subject
3
4 WHERE  T.Object='http://localhost/vocabulary/bench/Article'
5 AND    P.Object IS NOT NULL
```

Listing 13: SQL translation of SPARQL query Q_7 for PT

```
1 SELECT DISTINCT D.document AS article FROM Publication P
2 JOIN Document D          ON D.document=P.publication
3 JOIN PublicationType PT ON P.publication=PT.publication
4
5 WHERE  PT.type='http://localhost/vocabulary/bench/Article'
6 AND    P.pages IS NOT NULL
```

All SPARQL queries and SQL query translations for ST, VT, and PT are available in our github repository.

4.1.5 Data Storages

In our experiments, we used two storage backends, i.e., *Hive and HDFS*. For the latter one, we use various file formats such as *CSV, ORC, Avro,* and *Parquet*. Hive is an open-source distributed data warehousing database, which was built on top of Hadoop. The Hive was built for reading, writing, querying, and analyzing large datasets residing on HDFS [31]. Similar to RDBMS, Hive stores the data in the form of tables and makes operations using its SQL interface, HiveQL. To store the data in different storage backends, firstly, we converted the data from N-triples format into

CSV files using Apache Jena. Afterward, the Apache Spark framework was used to convert data from CSV file format into Avro, ORC, and Parquet file formats. The reason for using Spark is that it is able to handle large files in memory and supports reading and writing various file formats on HDFS. For storing data in Apache Hive data warehouse, we created three databases for each dataset size, 100M, 250M, and 500M. Then, we loaded CSV files into these Hive databases.

4.2 Analysis Methodology

In this section, we describe the approach that we follow to analyze the findings of our experiments. Instead of making only a descriptive analysis (what observed), we prefer to do advanced analysis, which consists of various analysis levels that are described in Figure 17[17]. These essential levels play an important role in defining the cost of the decision.

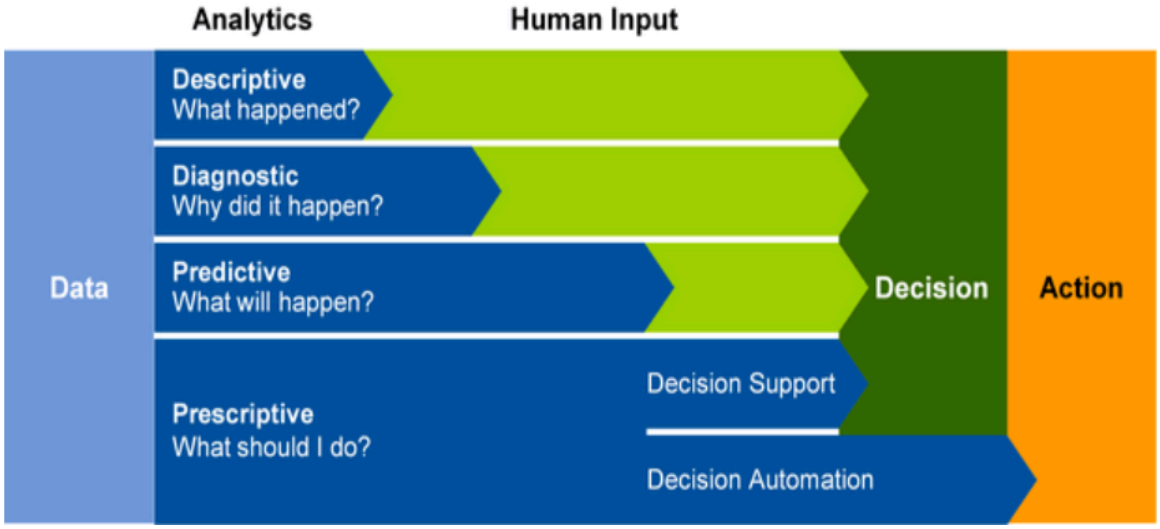


Figure 17: Analysis methodology framework

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

In our experiments, the solution area includes various variables and unknown trade-offs, which is a standard case for big data frameworks during performance benchmarking. Therefore, we advocate the necessity of the decision-making framework to analyze the results of the experiments in an unbiased and efficient manner. It is worth mentioning that the *Predictive* analysis level tries to provide an assessment model to answer the questions about what will happen in the future by knowing the historical data. Thus, we don't consider this level since it is out of scope in our investigation.

4.2.1 Descriptive Analysis

The *descriptive* analysis helps to describe and summarize data in a meaningful way which provides practitioners a simpler interpretation of the data; otherwise, it would be tough to understand what the data was showing if we simply presented our data in a row format. In practice, we use descriptive analysis to see which query is a short-running or long-running query based on their average run times. Moreover, we also get insights about which *storage backend, relational schema,* and *partitioning technique* performs best or worst for each query. However, the descriptive analysis does not allow us to make a conclusion regarding any hypotheses we might have made. Indeed, we are not able to say which configuration combination (storage backend, relational schema, and partitioning technique) performs best. In addition, some results make a contradiction in this level of analysis. For example, we will see that VT performs best for some queries, while for the same queries, the PT performs better when another partitioning technique is used.

4.2.2 Diagnostic Analysis

The *diagnostic* analysis examines the data to answer the question, *"Why did it happen?"*. It helps practitioners to understand why some result is given by taking the descriptive analysis a step further. With this level of advanced analysis, we can make a deeper analysis combining factual knowledge with contextual information about the query complexity and the configuration. Nevertheless, it is still hard to explore

the trade-offs in terms of dimensions that affect the performance of all queries. For this reason, we need a more advanced analysis level to investigate the impact of each dimension.

4.2.3 Prescriptive Analysis

At the bottom level of the framework, we can see the prescriptive analysis. As can be seen from its name, it suggests various courses of action to help analysts answer the question, *"What should you do?"*. In this level of analysis, we get insights for each dimension (relational schema, partitioning technique, and storage backend) in our experiments as well as considering the trade-offs between them, which helps to choose an optimal combination of these dimensions for the best query performance. For all of these reasons, we decided to use a formula of ranking criteria proposed [15] for partitioning techniques. This formula provides a high-level view of performance for a certain dimension across all queries. Therefore, we applied the ranking formula for relational schemas and storage backends as well. The below-mentioned equation is a generalized ranking formula that is used for ranking our partitioning techniques, relational schemas, and storage backends.

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

In formula 1, RS_D is the Rank Score of the dimension that the formula is applied to. As a rule, t shows the total number of the ranked dimension, and t=*3* when the formula is applied for ranking relational schema or partitioning technique since we have three different schemas and three different partitioning techniques. However, t = *5* when we apply the same formula for ranking storage backends as we have five different storage backends (or storage formats) in total. The variable b in the formula represents the number of executions, and it will be 11 in all cases because we have 11 queries in our SP^{2}Bench. Finally, the variable $O_D(r)$ represents how many times a certain dimension D occurs in the rank r.

Obviously, measuring the ranking degree for each dimension separately provides us a better analysis of results. However, we have observed that each of these three

calculated rank scores shows different values for a certain combination. As a result, we were not able to choose the best performing configuration combination since the trade-offs between these dimensions were not investigated yet. Hence, we tested three additional alternative approaches that give one unified ranking criterion by combining the ranking dimensions.

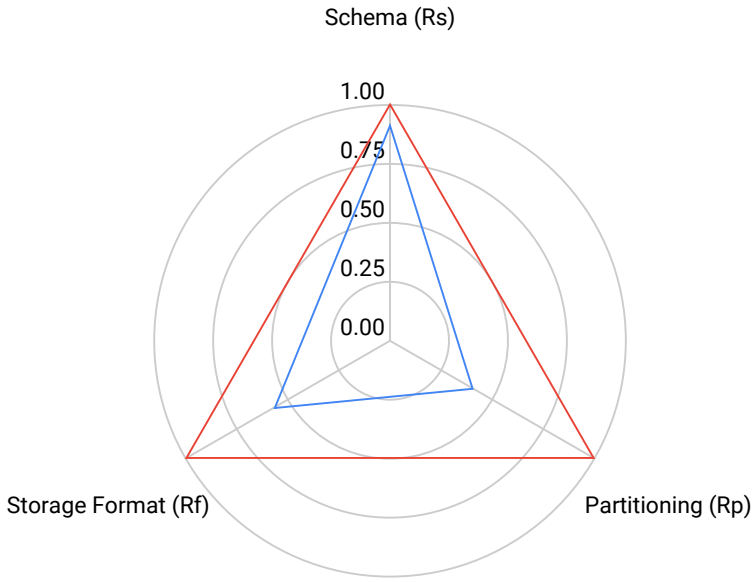


Figure 18: Triangle Area (Rta) combined Ranking criterion

- The ***Average (AVG)*** criterion simply averages the three dimensions rankings, Rf, Rp, and Rs, that are the ranking score of storage formats, partitioning techniques, and relational schemas, respectively. Equation 2 shows the formula for AVG.

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

- The ***Weighted Average (WAvg)*** extends the equation 2 by multiplying each ranking with its weight which is defined according to the number of storage backends,

the number of partitionings, and the number of schemas. Equation 3 shows the formula for WAvg.

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

- The ***Ranking Triangle Area (Rta)*** criterion attempts to leverage the trade-offs on a geometric interpretation of the three dimensions. It looks at the triangle where each side of the triangle represents the trade-offs of the ranking dimensions (Rf, Rp, Rs) and aims at maximizing the area of the triangle. The bigger area of the triangle, the better performance of all three dimensions together. The perfect case is to have a ranking score of *1 (max value)* on all sides, which is represented by the red triangle in Figure 18.

5 Evaluation and Discussion

In this chapter, we present the evaluation and discussion of our experimental results. Particularly, section 5.1 shows the architecture of experiments and describes hardware and software configuration for the deployment. In Section 5.2, we discuss the findings of experiments by making advanced analysis in different levels, i.e., descriptive analysis, diagnostic analysis, prescriptive analysis.

5.1 Architecture and deployment

The experiments have been conducted on a cluster of four machines on which a CentOS-Linux V7 OS was installed. Each node has 32-AMD cores, 128GB of memory, and a high-speed 2 TB SSD drive as a data drive. We also installed Spark V2.4 on each node to fully support Spark-SQL capabilities. For processing the data on Apache Hive data warehouse, we installed the Hive V3.2.1. In particular, our Spark cluster consists of three worker nodes and one master node. As a resource manager, we used YARN which uses 330 GB memory and 84 virtual processing cores in total.

We evaluated the Spark-SQL query performance for all combinations of relational schemas, partitioning techniques, and storage backends.

Figure 19 describes the architecture of experiments guiding the reader through the naming process that is used in results tables, i.e.,

$$\{Schema\}.\{Partitioning\_Technique\}.\{Storage\_Backend\}$$

For instance, (a.iii.4) corresponds to the Single Statement Table schema, PBP partitioning, and Parquet backend.

We run the experiment 5 times for each configuration. We excluded the first cold-start run time results during the analysis of results to avoid the warm-up bias and averaged the other four run time results. For getting query run time metric, we used the function *Spark.time*, and as a parameter we passed *spark.sql(query)* execution function. The output of this function is the running time of SQL query using Spark-SQL engine and SparkSession interface in the Spark environment.

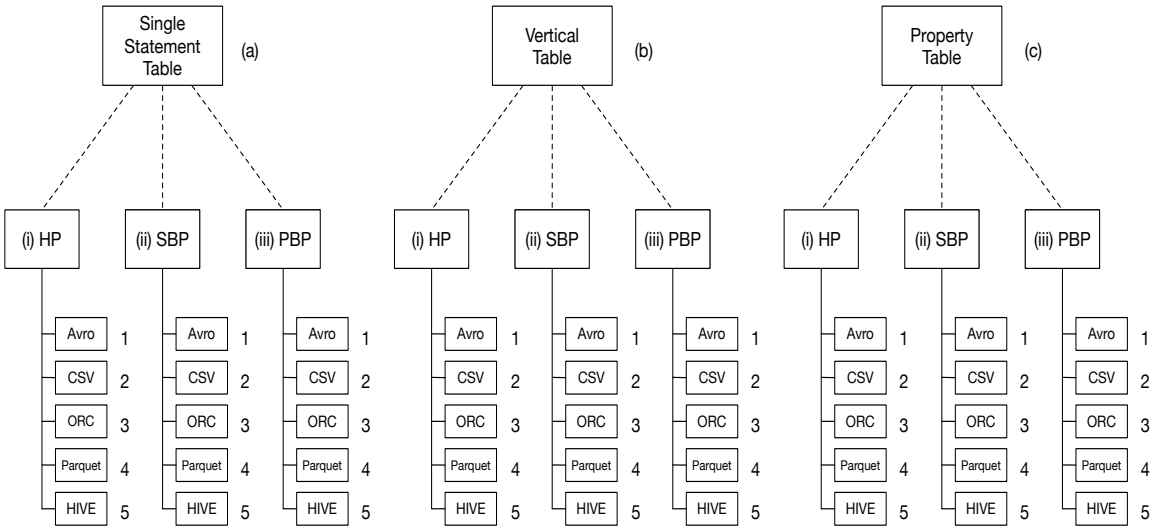


Figure 19: Experiments Architecture

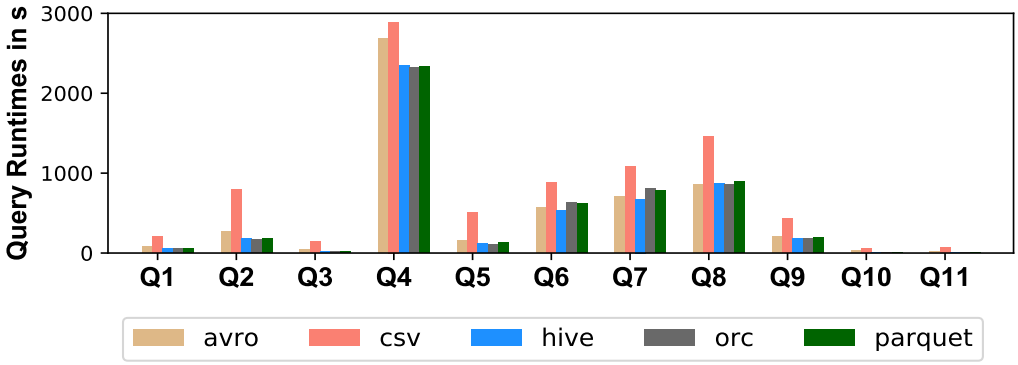
5.2 Experimental results

In this section, we analyze and discuss our experiments' results in several levels of advanced analysis, namely, descriptive, diagnostic, and prescriptive analysis (cf. Analysis Methodology 4.2). We unveil interesting insights about the performance of each criterion, e.g., relational schemas, partitioning techniques, and storage backends, on the Spark-SQL query engine. It worth mentioning that we analyze the results of Sp^{2}Bench Dataset with 500M triples in this thesis since it is the largest dataset in our experiments. However, experimental results and figures for 100M and 250M datasets are provided in our GitHub repository [18].

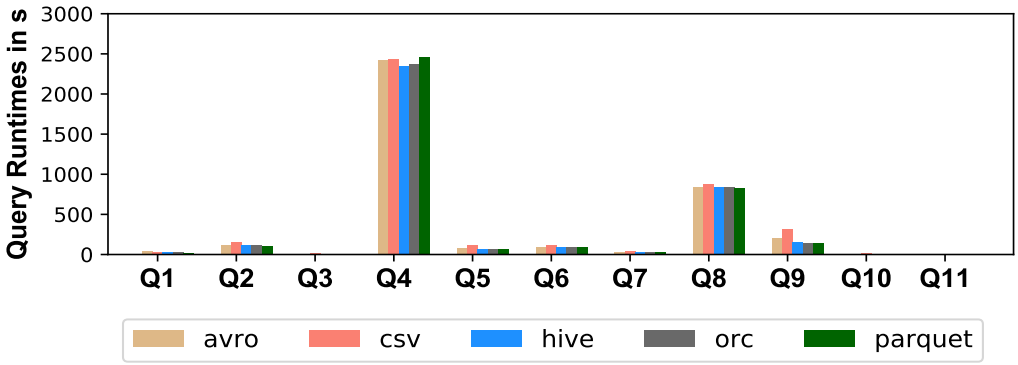
5.2.1 Descriptive analysis

We start with the descriptive analysis in which we provide the average query runtime figures of the benchmark queries by making a query performance analysis. Based on the execution time, we group the queries as a *short*-running, *medium*-running, and *long*-running queries, accordingly.

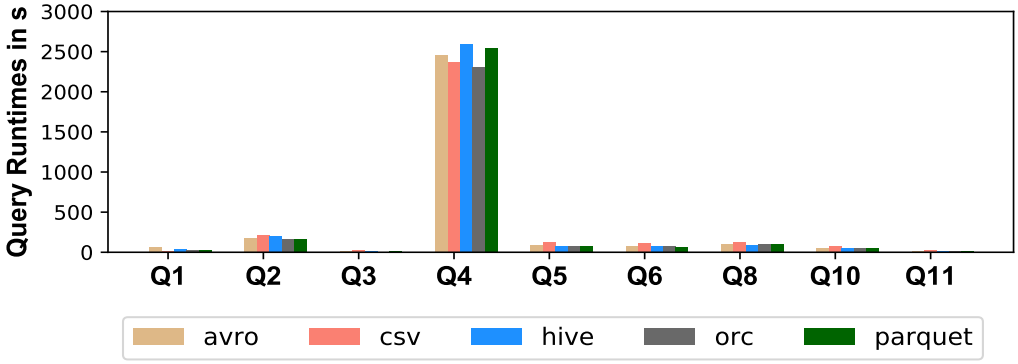
[18] `https://datasystemsgrouput.github.io/SPARKSQLRDFBenchmarking/Results`



(a) ST schema, Partitioned Horizontally

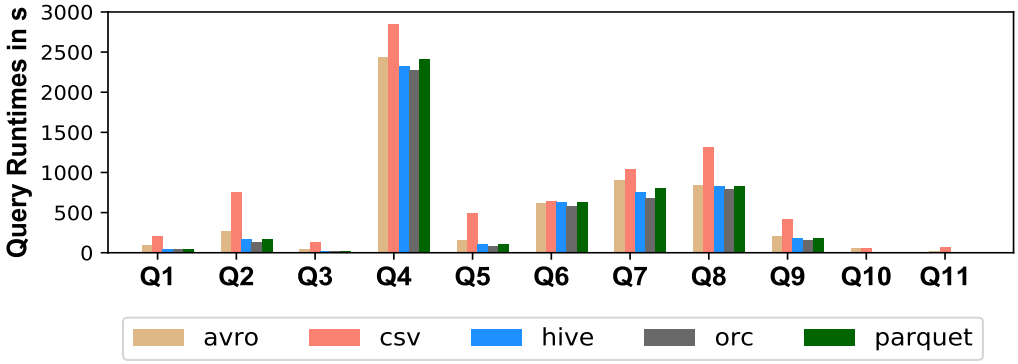


(b) VT schema, Partitioned Horizontally

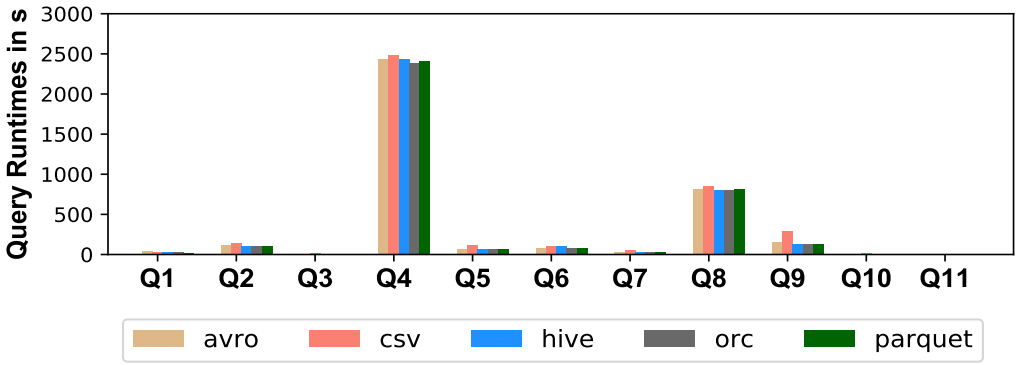


(c) PT schema, Partitioned Horizontally

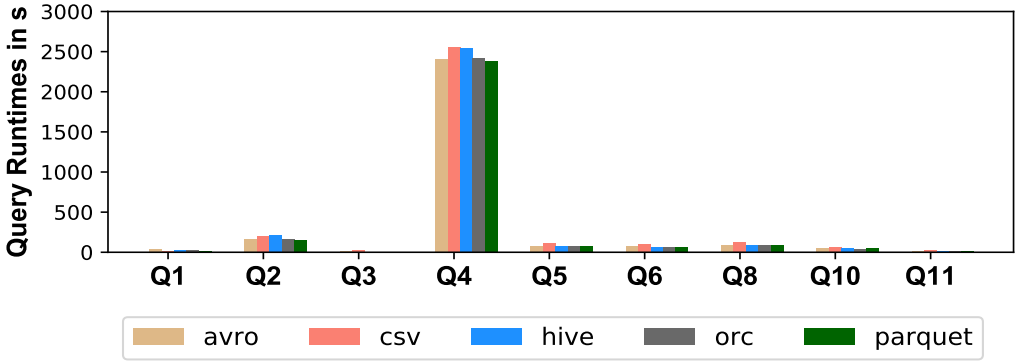
Figure 20: Average query execution time (in seconds) for 500M Sp^{2}Bench Dataset Partitioned Horizontally



(a) ST schema, Partitioned Based on Subject

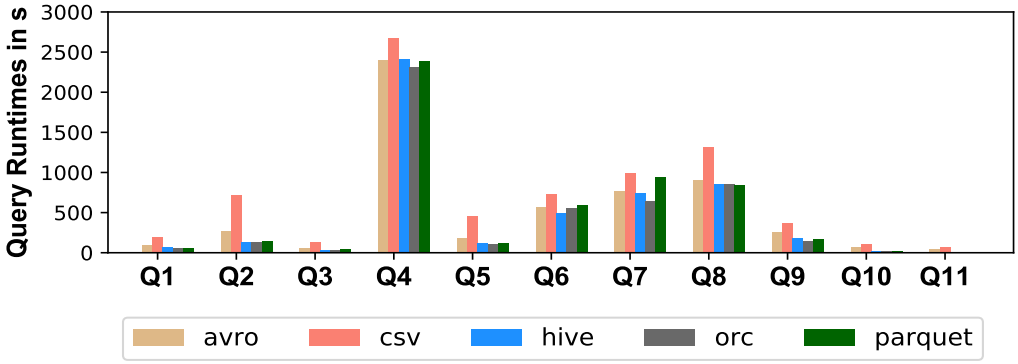


(b) VT schema, Partitioned Based on Subject

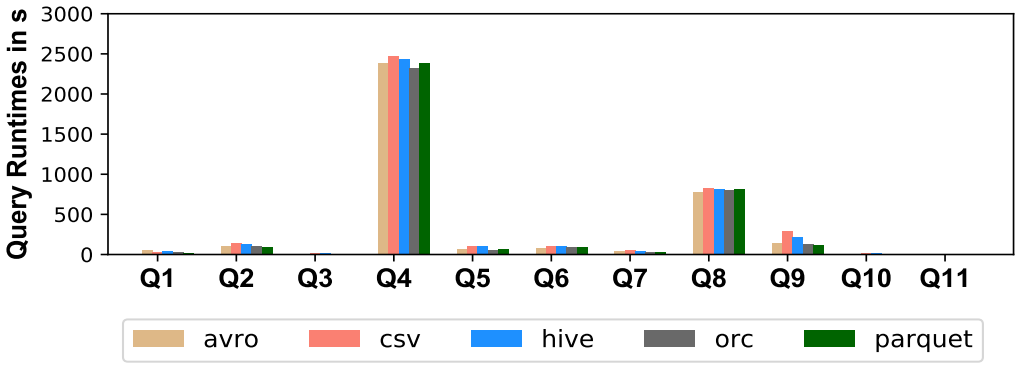


(c) PT schema, Partitioned Based on Subject

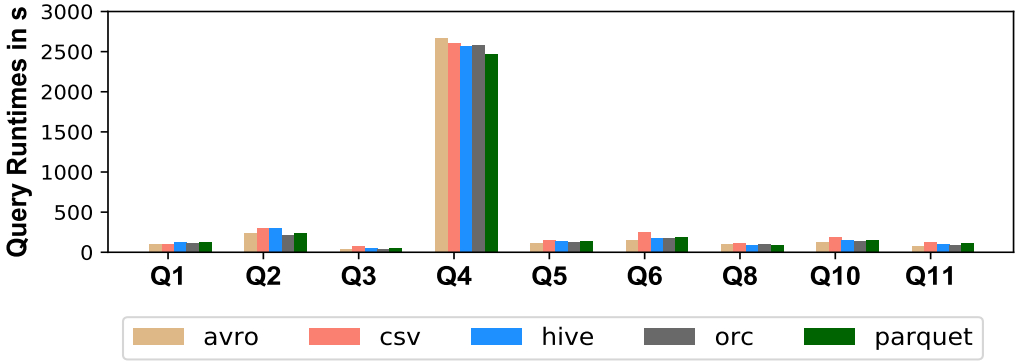
Figure 21: Average query execution time (in seconds) for 500M Sp^{2}Bench Dataset Partitioned Based on Subject



(a) ST schema, Partitioned Based on Predicate



(b) VT schema, Partitioned Based on Predicate



(c) PT schema, Partitioned Based on Predicate

Figure 22: Average query execution time (in seconds) for 500M Sp^{2}Bench Dataset Partitioned Based on Predicate

The average execution times of Sp2Bench queries are represented for *500M* dataset in Figures 20, 21, and 22. These figures show that the queries `Q1, Q3`, `Q10`, and `Q11` are short-running and the least impactful queries as they have the lowest running times. The queries `Q5`, `Q6`, `Q7`, and `Q9` are medium-running queries, while the queries `Q2`, `Q4` and `Q8` are long-running queries because they have the longest runtimes.

In this part of the analysis, we mainly focus on long-running queries since they might have interesting insights about approach limitations.

We observed that the query `Q2` has the lowest execution time with 100% when using VT schema. `Q2` has the highest execution time with the ST schema in the 60% of the results. However, when the PBP technique is used, `Q2` with the PT schema has a longer running time than ST schema, especially with ORC, Parquet, and Hive storage backends. This observation allows us to conclude that the VT is best-performing and ST is the worst-performing schema for Q2. Query `Q4` is the longest-running query for all relational schemes, partitioning techniques, and storage backends. Query `Q8` is the second longest-running query in all cases. Similarly, the VT schema for `Q4` performs better than other schemas. However, for query `Q8`, we have observed an interesting insight that the PT schema performs best by showing a significant enhancement in 100% of results. Consequently, we can say that the VT schema performs best in most cases, followed by the PT and ST schema.

Regarding the partitioning techniques, it is observed that the SBP approach outperforms other partitioning techniques for `Q4` and `Q8`. However, PBP method performs best in 60% of the `Q2` results, since the `Q2` is highly *predicate-oriented* query. For storage backends, we observed that the columnar file formats such as ORC and Parquet are the best performing file formats, followed by Hive. In comparison, Avro and CSV are mostly the worst-performing file formats.

Nevertheless, we can not claim that the VT schema is always the best performing schema because we have seen the PT schema is outperforming both ST and VT schemes in query `Q8`. Similarly, we can not conclude that the Avro is the worst performing file format since it performed best for `Q8` when HP applied on ST schema and PBP applied on VT schema. Therefore, a more detailed analysis of each criterion will be presented using ranking figures in the next sections.

5.2.2 Diagnostic analysis

In the diagnostic analysis, we expand our explanation on the previous descriptive analysis of results by providing more information about the query complexity of SP^{2}bench queries. Instead of focusing on each single query performance, we analyze the results for experimental dimensions such as relational schemes, partitioning techniques, and storage backends.

Regarding the relational schema comparison, we observed that the ST schema mostly performed worst in our experiments. The reason behind that is the size of the ST table is very large, and it requires the maximum number of self-joins. The PT schema performs better than ST schema since it requires the minimum number of joins when using SQL translation of SPARQL queries. Indeed, PT outperformed other relational schemes in the majority of query executions when the experiments were run in a single centralized machine with smaller size datasets [55]. However, scaling up the dataset sizes, PT schema resulted in larger intermediate results increasing the shuffling cost, which worsened its performance. Furthermore, PT schema with PBP and HP partitioning techniques did not give the desired result, especially with SP^{2}Bench queries, which are highly *'subject'-oriented*. Mostly, the VT schema performed best among other relational schemes. It is because the VT tables tend to be smaller, leading to smaller query inputs for Spark-SQL queries, which reduces the shuffling cost for join operations in Spark. Especially, VT schema becomes more efficient with queries that have a small number of joins.

During the experimental comparison of partitioning techniques, we have observed that Subject-based partitioning outperforms other approaches in general. The main reason behind that is the shape of SP^{2}Bench queries is 'Star' or 'Snowflake' in which the subject column is required to be a joining key. In fact, the SBP technique partitions the data by subject key, placing all rows with the same key in the same node, which significantly reduces the shuffling cost and maximizes the parallelization in the Spark cluster. This optimization is satisfied neither in Horizontal-based partitioning nor Predicate-based partitioning methods. Whereas, Horizontal-based partitioning randomly splits the data into the same size of chunks by keeping the balance be-

tween partitioned table sizes. The Predicate-based partitioning technique splits the data based on their predicate values which result in the highest degree of shuffling for most of the SP^{2}Bench queries when running with Spark-SQL. Consequently, the PBP approach does not perform well when evaluating *'subject'-oriented* queries. In addition, PBP is considered as the most unbalanced load partitioning technique since some of the RDF predicates in our SP^{2}Bench dataset have a small number of triples, while others have many. This unbalanced nature of the PBP technique leads to an inefficient join implementation in Spark-SQL. Therefore, we conclude that the SBP technique performs best, followed by PBP and HP techniques.

Considering the storage backends comparison, we observed that mostly ORC is the best performing file format among other file formats that we tested in our experiments. The second best-performing storage format is Parquet which is followed by Hive. Whereas, CSV and Avro file formats of HDFS showed the worst results. Exceptionally and interestingly, the Avro file format outperforms other storage formats when the PBP technique is used during experiments. In other words, we can say that the columnar storage backends performed best, while uncompressed textual CSV and the row-oriented Avro file formats performed worst, respectively. The main reason behind that is the most of SP^{2}Bench queries have a small number of projections, as presented in Table 1. Thus it is more efficient to query the data on columnar storage formats, which can easily scan only a certain set of columns, ignoring other unnecessary columns for query execution.

5.2.3 Prescriptive analysis

In the prescriptive level of analysis, we attempt to identify what is the best optimal configuration for the SP^{2}Bench benchmarking scenario. For this purpose, we show the result of the ranking criteria formula (cf. Formula 1) for each dimension and discuss the result of combined ranking criteria. Finally, we provide a table that shows the best and the worst configuration for each query, which helps to see which combined ranking criterion is the most relevant to choose the optimal performing configuration.

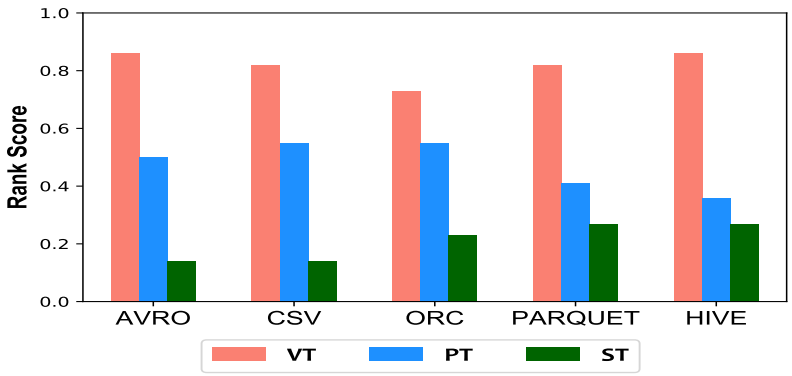
Relational Schema Ranking Analysis. Figure 23 shows the results of ranking scores for relational schemes (i.e., ST, VT, PT) by indicating how many times a particular schema performs best or worst, considering results of 500M dataset experiments. In Figure 23, we provide relational schema performance graphs separately for each storage backend and partitioning technique. Note that, the *higher* ranking score, the *better* performance.

We observed that the VT schema is always the best-performing schema of the relational schemes. The ST schema has the lowest ranking scores with 100% when HP and SBP techniques are applied, while it is the second best-performing schema with 60% when the PBP technique is used. On the other side, the PT schema falls between the ST schema and VT schema in 80% of query results.

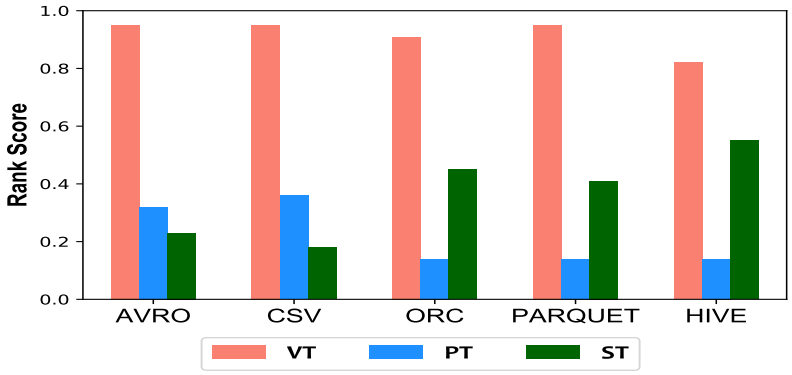
Partitioning Technique Ranking Analysis. Figure 24 represents the results of ranking scores for partitioning techniques (i.e., HP, SBP, PBP) and shows how many times a particular partitioning technique performs best or worst, considering the results of 500M datasets. We provided partitioning technique performance graphs for each schema separately for better analysis. The higher the ranking score, the better performance for these graphs as well. It worth mentioning that when a partitioning technique column in the graph is missing, this means that it always comes at the last rank(3rd rank in our case), and its rank score is zero.

Graphs in Figure 24 demonstrate that the Subject-based partitioning technique performs best in more than 73% of results. Although the Predicate-based partitioning technique performs worst in the more than 46% of results, it outperforms other techniques achieving the highest ranking score for ST schema with CSV file format and for VT schema with CSV and Avro file formats. On the other hand, the Horizontal Partitioning technique performs worst in more than 53% of the results.

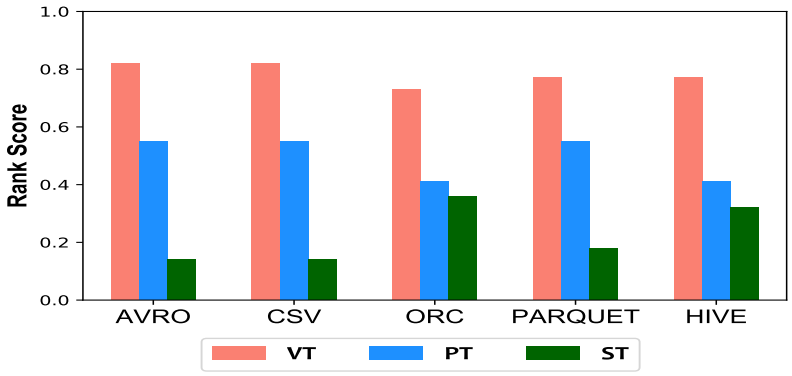
Storage Backends Ranking Analysis. Last but not least, we make an experimental comparison of storage backends as well to see how they impact the query execution performance in our experiments. Figure 25 represents ranking score graphs of storage backends for relational schemes and partitioning techniques separately.



(a) Relational Schema Ranking Scores with HP technique

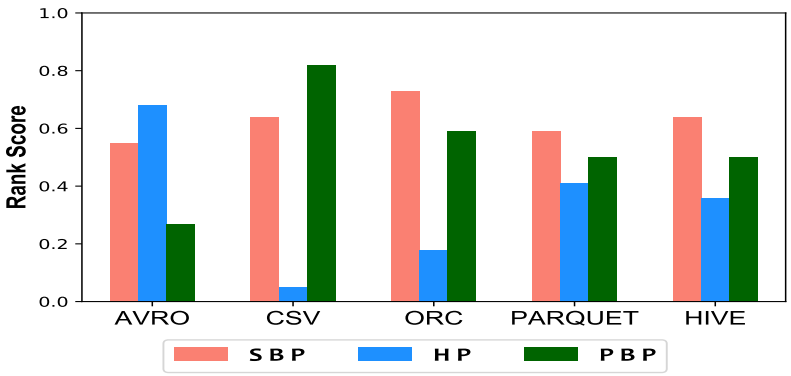


(b) Relational Schema Ranking Scores with PBP technique

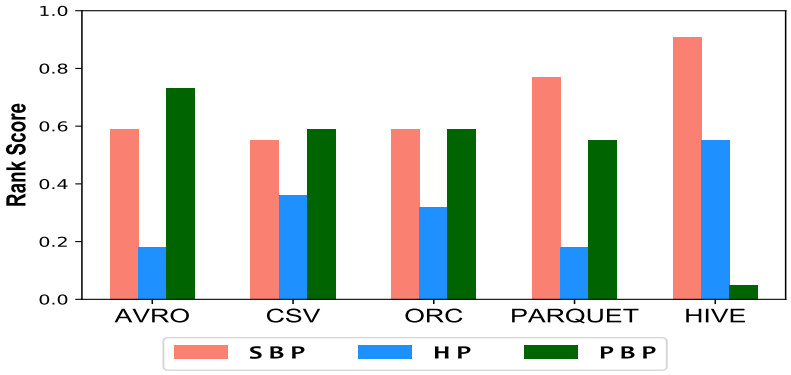


(c) Relational Schema Ranking Scores with SBP technique

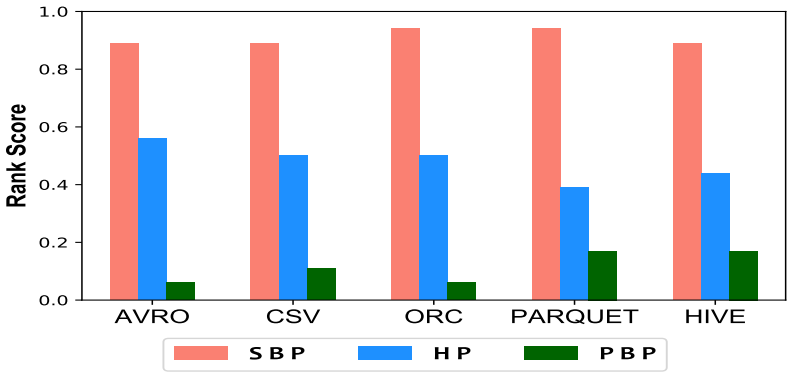
Figure 23: Relational Schema Ranking Scores for 500M Triples datasets (**Reading Key**: the higher is the better).



(a) Partitioning Technique Ranking Scores with ST schema

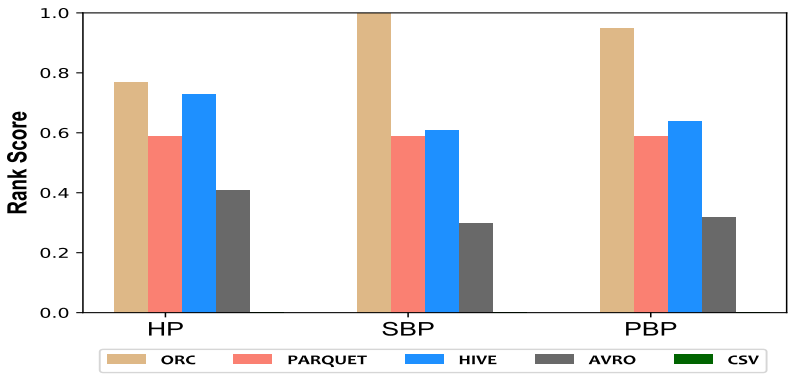


(b) Partitioning Technique Ranking Scores with VT schema

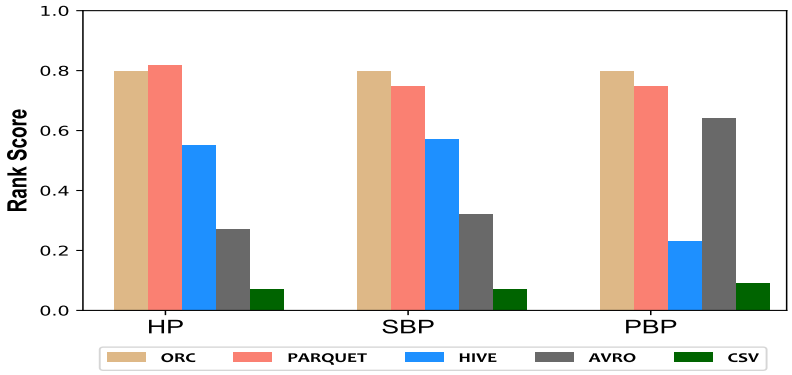


(c) Partitioning Technique Ranking Scores with PT schema

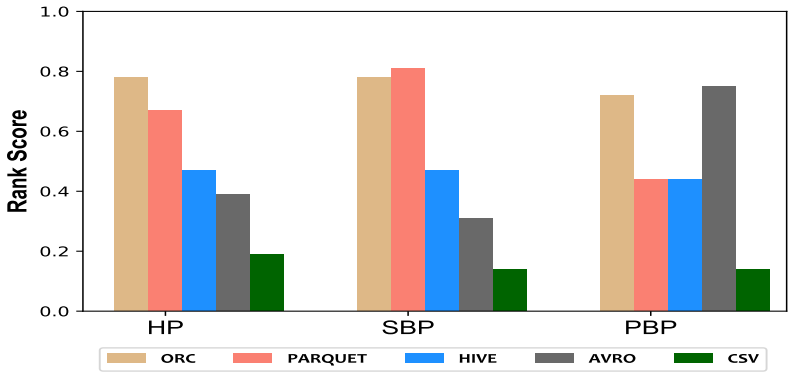
Figure 24: Partitioning Technique Ranking Scores for 500M Triples datasets (**Reading Key**: the higher is the better).



(a) Storage Backend Ranking Scores with ST schema



(b) Storage Backend Ranking Scores with VT schema



(c) Storage Backend Ranking Scores with PT schema

Figure 25: Storage Backend Ranking Scores for 500M Triples datasets (**Reading Key**: the higher is the better).

We observe that HDFS ORC is the best-performing file format for ST schema with all partitioning techniques, which is followed by Hive and Parquet file formats. At the same time, HDFS CSV and Avro are the worst-performing file formats for the mentioned cases.

For the VT schema, the HDFS Parquet file format outperforms the Hive storage backend, while the ORC is still the best-performing file format in all cases. HDFS CSV and Avro are the worst-performing file formats, respectively, having the lowest rank scores with 100% of these mentioned cases. As an exception, the Avro file format outperforms Hive and CSV file formats for the PBP technique with the VT schema.

Looking at the results of PT relational schema with HP and SBP techniques, we can observe that HDFS ORC and Parquet file formats are the best performing file formats as they share the highest ranking score with 50% for each of them. While Hive has a 3rd best rank in 100% of PT results. Again, CSV and Avro are performing worst with 100% for the mentioned ranking group, respectively. However, the Avro file format performs best, followed by ORC when the PBP technique is applied for PT schema.

Combined Ranking Analysis. Table 3 represents all possible various configuration combinations (cf. Figure 19) in our experiments. For instance, the configuration combination *(a.i.2)* indicates that the data is stored in *HDFS CSV* file format with *ST* schema, and *Horizontally Partitioned*. In table 3, we can see that we have 45 configuration combinations in total. The first three columns, i.e., *Rf, Rp,* and *Rs*, show the corresponding rank score of configuration which are calculated for storage formats, partitioning techniques, and relational schemes, respectively. The colored cells indicate the top 3 best-performing configuration for each column.

Looking at the Table 3, we observe that each dimension, i.e., *Rf, Rp,* and *Rs*, has different set of top 3 best-performing configurations. Thus, considering only one of these dimensions and ignoring others ends up with selecting a different configuration.

We decided to use a combined ranking criterion based on rank scores of these three dimensions since focusing on one ranking dimension leads to contradicting results as shown in Table 4. The combined ranking criterion, such as *Rta, WAvg,* and *AVG*,

Table 3: Configuration Ranking Criteria for 500M dataset

500M	Rf	Rp	Rs	Rta	WAvg	AVG
a.i.1	0.41	0.68	0.14	0.14	1.50	0.41
a.i.2	0.00	0.05	0.14	0.00	0.19	0.06
a.i.3	0.77	0.18	0.23	0.12	1.69	0.39
a.i.4	0.59	0.41	0.27	0.17	1.66	0.42
a.i.5	0.73	0.36	0.27	0.19	1.85	0.45
a.ii.1	0.30	0.55	0.14	0.09	1.19	0.33
a.ii.2	0.00	0.64	0.14	0.03	0.78	0.26
a.ii.3	1.00	0.73	0.36	0.45	2.76	0.70
a.ii.4	0.59	0.59	0.18	0.19	1.75	0.45
a.ii.5	0.61	0.64	0.32	0.26	1.98	0.52
a.iii.1	0.32	0.27	0.23	0.07	1.03	0.27
a.iii.2	0.00	0.82	0.18	0.05	1.00	0.33
a.iii.3	0.95	0.59	0.45	0.42	2.62	0.66
a.iii.4	0.59	0.50	0.41	0.25	1.89	0.50
a.iii.5	0.64	0.50	0.55	0.32	2.12	0.56
b.i.1	0.27	0.18	0.86	0.15	1.49	0.44
b.i.2	0.07	0.36	0.82	0.13	1.30	0.42
b.i.3	0.80	0.32	0.73	0.36	2.38	0.62
b.i.4	0.82	0.18	0.82	0.32	2.37	0.61
b.i.5	0.55	0.55	0.86	0.42	2.33	0.65
b.ii.1	0.32	0.59	0.82	0.31	1.94	0.58
b.ii.2	0.07	0.55	0.82	0.18	1.49	0.48
b.ii.3	0.80	0.59	0.73	0.50	2.65	0.71
b.ii.4	0.75	0.77	0.77	0.58	2.79	0.76
b.ii.5	0.57	0.91	0.77	0.55	2.63	0.75
b.iii.1	0.64	0.73	0.95	0.59	2.75	0.77
b.iii.2	0.09	0.59	0.95	0.23	1.69	0.54
b.iii.3	0.80	0.59	0.91	0.58	2.83	0.77
b.iii.4	0.75	0.55	0.95	0.55	2.75	0.75
b.iii.5	0.23	0.05	0.82	0.08	1.25	0.37
c.i.1	0.39	0.56	0.50	0.23	1.70	0.48
c.i.2	0.19	0.50	0.55	0.16	1.37	0.41
c.i.3	0.78	0.50	0.55	0.36	2.35	0.61
c.i.4	0.67	0.39	0.41	0.23	1.91	0.49
c.i.5	0.47	0.44	0.36	0.18	1.59	0.43
c.ii.1	0.31	0.89	0.55	0.31	1.95	0.58
c.ii.2	0.14	0.89	0.55	0.23	1.67	0.53
c.ii.3	0.78	0.94	0.41	0.48	2.65	0.71
c.ii.4	0.81	0.94	0.55	0.57	2.84	0.77
c.ii.5	0.47	0.89	0.41	0.33	2.09	0.59
c.iii.1	0.75	0.06	0.32	0.10	1.63	0.38
c.iii.2	0.14	0.11	0.36	0.04	0.70	0.20
c.iii.3	0.72	0.06	0.14	0.05	1.40	0.31
c.iii.4	0.44	0.17	0.14	0.05	1.05	0.25
c.iii.5	0.44	0.17	0.14	0.05	1.05	0.25

are explained in Section 4.2.3.

Table 4: Non-overlapping top 3 best-performing configuration combinations for the Rf, Rp, and Rs ranking criteria for 500M dataset

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

Particularly, we use the Table 5 to assess the correctness of each criterion (i.e., Rf, Rp, Rs, Rta, Avg, and WAvg) and to see which criteria is the most accurate. Firstly, we pick up the top 3 configurations (colored in Table 3) with the highest rank score for each criterion. Afterward, we arrange the ranks of these configurations for each query from Q1 to Q11 by calculating the rank position of the configuration for a certain query. For example, the configuration combination *(a.ii.3)* has the *23rd* rank position for Q1, *15th* rank position for Q2, and etc.

The colored cells in Table 5 indicate that the query is in the top 15 out of 45 ranking positions for the selected configurations. On the other hand, we also calculated how many times a certain configuration comes in the top 15 and averaged the result. For instance, the ranking criteria Rf with the configuration (a.ii.3) comes in the top 15 ranking position by three times, i.e., Q4 in 1st position, Q9 in 10th position, and Q10 in 12th position. While the configuration (a.iii.3) occurs to be in the top 15 ranks by four times for queries Q2, Q4, Q9, and Q11. The third selected configuration for Rf, *b.i.4*, achieves that goal by nine times for queries Q1, Q2, Q3, Q5, Q6, Q7, Q9, Q10, and Q11. Therefore, the rounded average top rank score for Rf criterion is 5 as shown in the table. The rest of other criteria are calculated in a similar way.

Finally, to determine which criteria is the most relevant to use for choosing the best optimal configuration, we calculate the accuracy of each criteria using the Formula 4.

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

Table 5: 500M Triples Dataset Ranking Criteria Comparison

Rf	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**<15**	**AVG**
a.ii.3	23	15	21	1	20	36	18	17	10	12	16	3	
a.iii.3	31	12	32	3	22	33	16	34	9	19	12	4	5
b.i.4	7	7	10	30	10	13	3	26	8	7	3	9	
Rp	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**<15**	**AVG**
c.ii.4	3	19	3	15	8	2	31	3	31	28	28	5	
c.ii.3	12	21	1	23	9	1	31	4	31	26	25	5	6
b.ii.5	10	6	8	25	4	20	7	19	5	5	2	8	
Rs	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**<15**	**AVG**
b.iii.1	27	3	16	13	5	10	12	16	7	8	9	8	
b.iii.2	18	16	31	31	23	21	14	28	25	22	18	1	6
b.iii.4	6	1	12	14	6	18	2	23	1	2	11	9	
AVG	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**<15**	**AVG**
b.iii.1	27	3	16	13	5	10	12	16	7	8	9	8	
b.iii.3	17	4	13	4	1	15	4	20	2	1	10	8	7
c.ii.4	3	19	3	15	8	2	31	3	31	28	28	5	
WAvg	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**<15**	**AVG**
c.ii.4	3	19	3	15	8	2	31	3	31	28	28	5	
b.iii.3	17	4	13	4	1	15	4	20	2	1	10	8	7
b.ii.4	4	2	7	19	3	11	1	22	4	6	4	9	
Rta	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**<15**	**AVG**
b.iii.1	27	3	16	13	5	10	12	16	7	8	9	8	
b.ii.4	4	2	7	19	3	11	1	22	4	6	4	9	8
b.iii.3	17	4	13	4	1	15	4	20	2	1	10	8	

In the formula 4, i is a certain configuration and *N(i)* is the number of times the configuration occurred to be in top 15 rank position for each ranking criteria *(cr)*. Applying this formula for results of 500M dataset, we observed that the accuracy for Rf, Rp, Rs, AVG, WAvg and Rta are 48%, 55%, 55%, 64%, 67%, 76%, respectively.

Table 6: The Best and Worst configurations for queries of Sp^{2}Bench Benchmark Dataset

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

In conclusion, it appears that the Rta (triangle area) criterion is the most accurate criteria for measuring the configuration performance in our SP^{2}Bench query workloads scenario. The AVG (average) and WAvg (weighted average) are the second accurate criteria for measuring the performance of configuration. Finally, Table 6 summarizes the analysis of the experiments, highlighting the best and worst combinations of schema, partitioning techniques, and storage backends for each query in the SP2Bench workload.

Notably, for the sake of conciseness, we used only the 500M dataset table to describe how the ranking calculations are done. However, all data and analysis can be found on the GitHub repository web page.

6 Conclusion

In this thesis work we presented our solution for handling volume and variety of big data simultaneously.

In Chapter 3, we presented the main challenges of Big Data, i.e., volume, variety, and velocity. Using the *Macro-Meso-Micro* framework, we formalized the Macro-level research question *How to handle volume and variety of big data simultaneously?* This question helps us to formulate the problem that we address in this thesis. However, we need to narrow the scope of the problem to make it more specific towards a solution. Therefore, we provided requirements specifications which led us to Meso-level question *Can we handle OLAP workloads over graphs using MapReduce-based Big Data Frameworks?* Although the Meso-level question is more specific, it is still very complex to answer. Thus, we made an analysis of alternative solutions for each requirement to identify a feasible problem to solve. Finally, we formulate the Micro-level question *How to implement the SPARQL queries over RDF data using Spark-SQL?* This question allows us to design our experiments, providing a solution for the problem.

In Chapter 4, we presented how we designed and implemented different experiments on Spark-SQL variating:

- Relational schemes (ST, VT, PT)
- Partitioning techniques (HP, SBP, PBP)
- Storage backends and data formats (Hive, CSV, Avro, ORC, Parquet)

In particular, we generated three different sizes of RDF datasets from the publication-network DBLP computer science scenario, i.e., 100M, 250M, and 500M triples datasets. We translated the SPARQL benchmark queries into SQL queries using relational algebra. We compared the trade-offs of relational schemes by storing the RDF data in Single Statement Table (ST), Vertically Partitioned Tables (VT), and Property Tables (PT). As the partitioning significantly affects the query execution performance

in a distributed environment, we also applied various partitioning techniques on RDF tables, namely, Horizontal Partitioning (HP), Subject-Based Partitioning (SBP), and Predicate-Based Partitioning (PBP). Furthermore, we evaluated the performance of the Spark-SQL engine using two different storage backends, i.e., Hive and HDFS. For the latter one, we compared four various file formats such as CSV, Avro, ORC, and Parquet.

In Chapter 5, we performed a systematic experimental evaluation of the Spark-SQL query engine performance for answering SPARQL queries over RDF datasets. Particularly, we analyzed and discussed the impact of each *relational schema*, *partitioning technique*, and s*torage backend* on the query execution performance of Spark-SQL engine in our experiments. We made advanced analysis for the experimental results, which covers *descriptive*, *diagnostic*, and *predictive* analysis levels.

We observed that the ST schema mostly performed worst in our experiments since it is very large and requires the maximum number of self-joins. PT schema with PBP and HP partitioning techniques did not give the desired result, especially with SP2Bench queries, which are highly 'subject'-oriented. Mostly, the VT schema performed best among other relational schemes. It is because the VT tables tend to be smaller, leading to smaller query inputs for Spark-SQL queries, which reduces the shuffling cost for join operations in Spark. Regarding partitioning techniques, Subject-based partitioning outperformed other approaches. Because the shape of SP2Bench queries is 'Star' or 'Snowflake' in which the subject column is required to be a joining key. SBP technique places all rows with the same subject in the same node, which significantly reduces the shuffling cost in a Spark cluster. However, the PBP technique did not perform well with subject-oriented queries since it splits the data based on their predicates and results with the highest degree of shuffling for most of the Sp^{2}Bench queries. For the storage backends, we observed that the columnar storage backends performed best, while uncompressed textual CSV and the row-oriented Avro file formats performed worst, respectively. The main reason behind that is the most of SP^{2}Bench queries have a small number of projections; thus it is more efficient to query the data on columnar storage formats, which can easily scan

only a certain set of columns ignoring other unnecessary columns for query execution.

Although the *descriptive* and *diagnostic* levels of analysis provided us valuable insights about each dimension, we were not able to say which configuration combination (relational schema, partitioning technique, and storage backend) performs best. Because some results made a contradiction in these levels of analysis. For instance, we observed that the VT schema performs best for some queries, while the PT schema performs better for the same queries when another partitioning technique is used.

Consequently, in the prescriptive level, we calculated rank scores (cf. Formula 1), i.e., *Rs*, *Rp*, and *Rf*, for each relational schema, partitioning technique, and storage backend, respectively. However, these three dimensions provided us with different sets of the top three best-performing configurations. And, considering only one of these dimensions and ignoring others, ended up selecting different configurations. Therefore, we used combined ranking criterion, i.e., *Rta*, *WAvg*, and *AVG*, based on rank scores of these three dimensions. Finally, we assessed the accuracy of each criterion (i.e., *Rf, Rp, Rs, Rta, Avg,* and *WAvg*) and concluded that the*Rta* is the most accurate criteria for measuring the configuration performance in our SP^{2}Bench query workloads scenario.

In conclusion, we summarized the analysis of the experiments in Table 6 highlighting the best and worst combinations of relational schema, partitioning techniques, and storage backends for each query in the Sp^{2}Bench workload.

As a future extension of this work, other benchmark datasets (i.e., WatDiv, LUBM) and real datasets (i.e., DBpedia, YAGO) with different levels of structuredness can be studied. We can also broaden the scope of our analysis by implementing different types of query shapes with different complexities.

References

[1] Laney, Doug. *"3D data management: Controlling data volume, velocity and variety."* META group research note 6.70 (2001): 1.

[2] 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.

[3] Neumann, Thomas & Weikum, Gerhard. (2008). *RDF-3X: a RISC-style Engine for RDF.* Proceedings of the VLDB Endowment, v.1, 647-659 (2008). 1. 10.1145/1453856.1453927.

[4] L. Zou, J. Mo, L. Chen, M.T. Ozsu and D. Zhao. *gStore: Answering SPARQL Queries via Subgraph Matching.* PVLDB, 4(8):482–493, 2011.

[5] https://sesame.readthedocs.io/en/latest/

[6] P. A. Boncz, O. Erling, and M.-D. Pham. *Experiences with Virtuoso Cluster RDF Column Store.* In Linked Data Management, pages 239–259. Chapman and Hall/CRC, 2014

[7] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: *A Federated Repository for Querying Graph Structured Data from the Web.* In The Semantic Web, number 4825 in Lecture Notes in Computer Science, pages 211–224. Springer Berlin Heidelberg, 2007.

[8] 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

[9] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.htmlIntroduction

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

[11] 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.

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

[13] https://rdf4j.org/documentation/

[14] https://jena.apache.org/

[15] 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.

[16] M. Schmidt, T. Hornung, G. Lausen and C. Pinkel, *SP^{2}Bench: 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.

[17] Bauer, Florian, and Martin Kaltenböck. "Linked open data: The essentials." Edition mono/monochrom, Vienna 710 (2011).

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

[19] E. Prud'hommeaux and A. Seaborne. *SPARQL Query Language for RDF.* http://www.w3.org/TR/rdf-sparql-query/, 2008.

[20] N. Hubert, C. Olivier, A. Bernd, *SPARQL query processing with Apache Spark, CoRR*, 2016, http://arxiv.org/abs/1604.08903

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Fast and Interactive Analytics Over Hadoop Data with Spark. USENIX ;login:, 34(4):45–51, 2012.

[22] Holden Karau and Rachel Warren. 2017. ¡i¿High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark¡/i¿ (1st. ed.). O'Reilly Media, Inc., pages 13-14

[23] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. *Efficient RDF Storage and Retrieval in Jena2.* In Proc. 1st International Workshop on Semantic Web and Databases (SWDB), pages 131–150, 2003.

[24] D. Faye, O. Cure, and D. Blin. *A Survey of RDF Storage Approaches.* ARIMA Journal, 15:11–35, 2012

[25] K. Wilkinson. *Jena Property Table Implementation.* In SSWS, pages 35–46, 2006.

[26] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. *Efficient RDF Storage and Retrieval in Jena2.* In Proc. of SWDB, pages 131–150, 2003

[27] Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: *Scalable Semantic Web Data Management Using Vertical Partitioning.* In: VLDB, pp. 411–422 (2007)

[28] 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

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

[30] Wylot, Marcin, et al. "RDF data storage and query processing schemes: A survey." ACM Computing Surveys (CSUR) 51.4 (2018): 1-36.

[31] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. ¡i¿Proc. VLDB Endow.¡/i¿ 2, 2 (August 2009), 1626–1629. DOI:https://doi.org/10.14778/1687553.1687609

[32] https://en.wikipedia.org/wiki/Comma-separated_values

[33] https://nxtgen.com/hadoop-file-formats-when-and-what-to-use

[34] Pérez, Jorge, Marcelo Arenas, and Claudio Gutierrez. *"Semantics and complexity of SPARQL."* ACM Transactions on Database Systems (TODS) 34.3 (2009): 1-45.

[35] Cyganiak, Richard. *"A relational algebra for SPARQL."* Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170 35 (2005): 9.

[36] H. Hu, Y. Wen, T. -S. Chua and X. Li, "Toward Scalable Systems for Big Data Analytics: A Technology Tutorial," in IEEE Access, vol. 2, pp. 652-687, 2014, doi: 10.1109/ACCESS.2014.2332453.

[37] https://demand-planning.com/2019/12/17/overcoming-the-challenges-of-big-data/

[38] https://flink.apache.org/

[39] https://storm.apache.org/

[40] S.Langhi, E.D. Valle, R. Tommasini, textitTowards extream processing with KEPLr, 29 Apr 2020, https://www.politesi.polimi.it/handle/10589/153935

[41] https://www.cambridgesemantics.com/blog/semantic-university/learn-rdf/rdf-vs-xml/

[42] T. Kudrass, "Coping with semantics in XML document management", in Proceedings of the Ninth OOPSLA Workshop on Behavioral Semantics, Northeastern University, 2001, pp. 150-161.

[43] M. Klein, "XML, RDF, and relatives", IEEE Intelligent Systems, vol.16, no. 2, pp. 26-28, 2001.

[44] https://www.cambridgesemantics.com/blog/semantic-university/learn-rdf/

[45] Gábor Szárnyas. Query, Analysis, and Benchmarking Techniques for Evolving Property Graphs of Software Systems. PhD thesis. Budapest University of Technology and Economics, 2019

[46] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107–113. DOI:https://doi.org/10.1145/1327452.1327492

[47] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In Symposium on Operating Systems Design and Implementation (OSDI), pages 599–613, 2014.

[48] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages 135–146. ACM, 2010.

[49] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. Proc. VLDB Endow. 10, 13 (September 2017), 2049–2060. DOI:https://doi.org/10.14778/3151106.3151109

[50] J. Gray, *The Benchmark Handbook for Database and Transaction Systems.* Morgan Kaufmann, 1993.

[51] 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

[52] https://sparkbyexamples.com/spark/spark-shuffle-partitions/

[53] https://blog.scottlogic.com/2018/03/22/apache-spark-performance.html

[54] https://jena.apache.org/documentation/tdb/

[55] M. Ragab, R. Tommasini and S. Sakr, *Benchmarking SparkSQL under Alliterative RDF Relational Storage Backends* (2019).

[56] A. Schatzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen. Sempala: *Interactive SPARQL Query Processing on Hadoop.* In ISWC, volume 8796 of LNCS, pages 164–179, 2014.

License

Non-exclusive license to reproduce thesis and make thesis public

I, **Sadig Eyvazov**

1. herewith grant the University of Tartu a free permit (non-exclusive license) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Large RDF Graph Processing on Top of Spark

supervised by Riccardo Tommasini, Mohammed Ragab.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons license CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Sadig Eyvazov

11/03/2021