Towards A Scalable Semantic-based Distributed Approach for SPARQL query evaluation

Gezim Sejdiu[1], Damien Graux[2], Imran Khan[1], Ioanna Lytra[1], Hajira Jabeen[1], and Jens Lehmann[1,2]

[1] Smart Data Analytics, University of Bonn, Germany
sejdiu@cs.uni-bonn.de, s6imkhan@uni-bonn.de, lytra@cs.uni-bonn.de, jabeen@cs.uni-bonn.de, jens.lehmann@cs.uni-bonn.de
[2] Enterprise Information Systems, Fraunhofer IAIS, Germany
damien.graux@iais.fraunhofer.de, jens.lehmann@iais.fraunhofer.de

Abstract. Over the last two decades, the amount of data which has been created, published and managed using Semantic Web standards and especially via Resource Description Framework (RDF) has been increasing. As a result, efficient processing of such big RDF datasets has become challenging. Indeed, these processes require, both efficient storage strategies and query-processing engines, to be able to scale in terms of data size. In this study, we propose a scalable approach to evaluate SPARQL queries over distributed RDF datasets using a semantic-based partition and is implemented inside the state-of-the-art RDF processing framework: SANSA. An evaluation of the performance of our approach in processing large-scale RDF datasets is also presented. The preliminary results of the conducted experiments show that our approach can scale horizontally and perform well as compared with the previous Hadoop-based system. It is also comparable with the in-memory SPARQL query evaluators when there is less shuffling involved.

1 Introduction

Recently, significant amounts of data have been created, published and managed using the Semantic Web standards. Currently, the Linked Open Data (LOD) cloud comprises more than 10 000 datasets available online[1] using the Semantic Web standards. RDF is a standard that represents data linked as a graph of resources following the idea of the linking structure of the Web and using URIs for representation.

To facilitate better maintenance and faster access to this scale of data, efficient data partitioning is needed. One of such partitioned strategies is semantic-based partitioning. It groups the facts based on the subject and its associated triples. We want to explore and evaluate the effect of semantic-based partitioning on query performance when dealing with such a volume of RDF datasets.

SPARQL is a W3C standard query language for querying data modeled as RDF. Querying RDF data efficiently becomes challenging when the size of the

[1] http://lodstats.aksw.org/

data increases. This has motivated a considerable amount of work on designing distributed RDF systems able to efficiently evaluate SPARQL queries ([20,6,21]). Being able to query a large amount of data in an efficient and faster way is one of the key requirements for every SPARQL engine.

To address these challenges, in this paper, we propose a scalable semantic-based distributed approach[2] for efficient evaluation of SPARQL queries over distributed RDF datasets. The main component of the system is the data partitioning and query evaluation over this data representation.

Our contributions are:

- A scalable approach for semantic-based partitioning using the distributed computing framework, Apache Spark.
- A scalable semantic-based query engine (*SANSA.Semantic*) on top of Apache Spark (under the *Apache Licence 2.0*).
- Comparison with state-of-the-art engines and demonstrate the performance empirically.
- Integration with the SANSA [13][3] framework.

The rest of the paper is structured as follows: Our approach for data modeling, data partitioning, and query translation using a distributed framework are detailed in section 3 and evaluated in section 4. Related work on the SPARQL query engines is discussed in section 5. Finally, we conclude and suggest planned extensions of our approach in section 6.

2 Preliminaries

Here, we first introduce the basic notions used throughout the paper.

Apache Hadoop and MapReduce Apache Hadoop is a distributed framework that allows for the distributed processing of large data sets across a cluster of computers using the MapReduce paradigm. Beside its computing system, it contains a distributed file system: the Hadoop Distributed File System (HDFS), which is a popular file system capable of handling the distribution of the data across multiple nodes in the cluster.

Apache Spark Apache Spark is a fast and generic-purpose cluster computing engine which is built over the Hadoop ecosystem. Its core data structure is Resilient Distributed Dataset (RDD) [25] which are a fault-tolerant and immutable collections of records that can be operated in a parallel setting. Apache Spark provides a rich set of APIs for faster, in-memory processing of RDDs.

Data Partitioning Partitioning the RDF data is the process of dividing datasets in a specific logical and/or physical representation in order to ease faster

[2] `https://github.com/SANSA-Stack/SANSA-Query/tree/develop/sansa-query-spark/src/main/scala/net/sansa_stack/query/spark/semantic`

[3] `http://sansa-stack.net/`

access and better maintenance. Often, this process is performed for improving the system availability, load balancing and query processing time. There are many different data partitioning techniques proposed in the literature. We choose to investigate the so-called *semantic-based partitioning* behaviors when dealing with large-scale RDF datasets. This partitioned technique was proposed in the SHARD [17] system. We have implemented this technique using in-memory processing engine, Apache Spark for better performance. A semantically partitioned fact is a tuple (S, R) containing pieces of information $R \in (P, O)$ about the same S where S is a unique subject on the RDF graph and R represents all its associated facts i.e predicates P and objects O.

3 Approach

In this section, we present the system architecture of our proposed approach, the semantic-based partitioning, and mapping SPARQL to Spark Scala-compliant code.

3.1 System Architecture Overview

The system architecture overview is shown in the Figure 1.

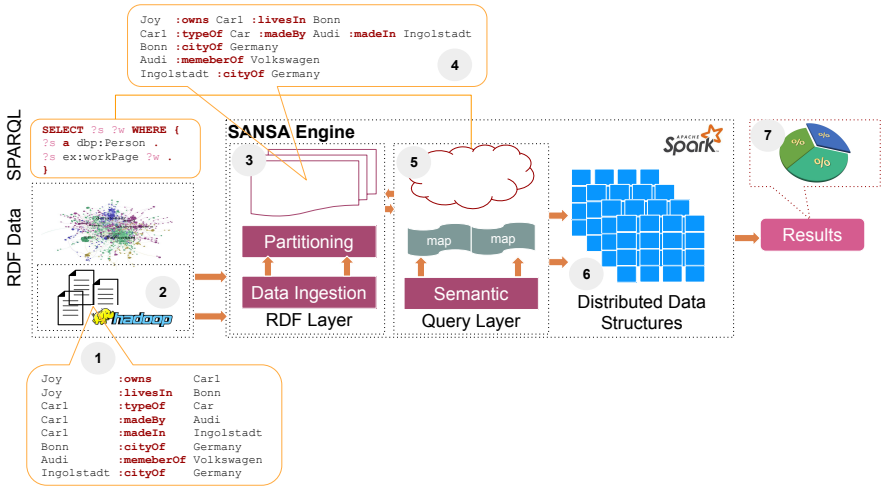

Fig. 1. System Architecture Overview.

It consists of three main facets: Data Storage Model, SPARQL Query Fragments Translator, and Query Evaluator. Below, each facet is discussed in more details.

Data Storage Model We model the RDF data following the concept of RDDs. RDDs are immutable collections of records, which represent the basic building blocks of the Spark framework. RDDs can be kept in-memory and are able to operate in parallel throughout the Spark cluster. We make use of SANSA [13]'s data representation and distribution layer for such representation.

First, the RDF data (see *Step 1* as an example) needs to be loaded into a large-scale distributed storage (*Step 2*). We use Hadoop Distributed File-System (HDFS)[4]. We choose HDFS as Spark is capable of performing operations based on data locality in order to choose the nearest data for faster and efficient computation over the cluster. Second, we partition (*Step 3*) the data using semantic-based partitioning (see *Step 4* as an example of such partition). Instead of working with table-wise representation where the triples are kept in the format of $RDD < Triple >$, data is partitioned into subject-based grouping (e.g. all entities which are associated with a unique subject). Consider the example in the Figure 1 (*Step 2*, first line), which represents two triples associated with the entity `Joy`:

```
Joy :owns Car1 :livesIn Bonn
```

This line represents that the entity `Joy` owns a car entity `Car1`, and that `Joy` lives in `Bonn`.

Often flattening data is considered immature with respect to other data representation, we want to explore and investigate if it improves the performance of the query evaluation. We choose this representation for the reason of easy-storage and reuse while designing a query engine. Although, it slightly degrades the performance when it comes to multiple scans over the table when there are multiple predicates involved in the query. However, this is minimal, as Spark uses in-memory, caching operations. We will discuss this on the section 4 into more detail.

SPARQL Query Fragments Translation This process generates the Scala code in the format of Spark RDD operations using the key-value pairs mechanism. With Spark pairRDD, one can manipulate the data by splitting it into key-value pairs and group all associated values with the same keys. It walks through the SPARQL query (*Step 4*) using the Jena ARQ[5] and iterate through clauses in the SPARQL query and bind the variables into the RDF data while fulfilling the clause conditions. Such iteration corresponds to a single clause with one of the Spark operations (e.g. *map*, *filter*, *reduce*). Often this operation needs to be materialized i.e the result set of the next iteration depends on the previous clauses and therefore a *join* operation is needed. This is a bottleneck since scanning and shuffling is required. In order to keep these joins as small as possible, we leverage the caching techniques of the Spark framework by keeping the intermediate results in-memory while the next iteration is performed. Finally, the Spark-Scala executable code is generated (*Step 5*) using the bindings

[4] `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`

[5] `https://jena.apache.org/documentation/query/`

Algorithm 1: Spark parallel semantic-based query engine.

input : q: a SPARQL query, *input*: an RDF dataset
output: *result* an RDD – list of result set

```
   /* Loading the graph                                                    */
1  graph = spark.rdf(lang)(input)
   /* Partitioning the graph. See algorithm 2 for more details.           */
2  partitionGraph ← graph.partitonAsSemanticGraph()
   /* Querying the graph. See algorithm 3 for more details.               */
3  result ← partitionGraph.sparql(q)
4  return result
```

corresponding the query. Besides simple BGP translation, our system supports **UNION**, **LIMIT** and **FILTER** clauses.

Query Evaluator The mappings created as shown in the previous section can now be evaluated directly into the Spark RDD executable code. The result set of these operations is distributed data structure of Spark (e.g. RDD)(*Step 6*). The result set can be used for further processing and visualization using the SANSA-Notebooks (*Step 7*) [5].

3.2 Distributed Algorithm Description

We implement our approach using the Apache Spark framework (see algorithm 1). It constructs the graph (*line 1*) while reading RDF data and converts it into an RDD of triples. Later, it partitions the data (*line 2*, for more details see algorithm 2) using the semantic-based partitioning strategy. Finally, the query evaluator is constructed (*line 3*) which is detailed in algorithm 3.

The partition algorithm (see algorithm 2) transforms the RDF graph into a convenient SP (*line 2*). For each unique triple in the graph in a distributed fashion, it does the following: It gets the values about subjects and objects (*line 3*) and local name of the predicate (*line 4*). It generates the key-value pairs of the subject and its associated triples with predicate and object separated with the space in between (*line 5*). After the mapping is done, the data is grouped by key (in our case *subject*) (*line 6*). Afterward, when this information is collected, the block is partitioned using the *map* transformation function of Spark to refactor the format of the lines based on the above information (*line 7*).

This SPARQL query rewriter includes multiple Spark operations. First, partitioned data is mapped to a list of variable bindings satisfying the first basic graph pattern (BGP) of the query (*line 2*). During this process, the duplicates are removed and the intermediate result is kept in-memory (RDD) with the variable bindings as a key. The consequent step is to iterate through other variables and bind them by processing the upcoming query clauses and/or filtering the other ones unseen on the new clause. These intermediate steps perform Spark operations over both, the partitioned data and the previously bound variables which were kept on Spark RDDs.

Algorithm 2: partitonAsSemanticGraph: Semantic-based partition algorithm.

```
   input  : graph: an RDD of triples
   output: partionedData: an RDD of partitions
1  partitonedData ← ∅
2  foreach ∀!triple ∈ graph  &&  triple.getSubject ≠ ∅ do
3  |  s ← triple.getSubject;  o ← triple.getObject
4  |  p ← triple.getPredicate.getLocalName
5  |  partitonedData += (s, p + " " + o + " ")
6  partitonedData.reduceByKey(_ + _)
7                .map(f → (f._1 + " " + f._2))
8  return partitonedData
```

Algorithm 3: sparql: Semantic-based query algorithm.

```
   input  : partitonedData: an RDD of partitions
   output: result an RDD of result set
1  foreach p ∈ partitionedData do
2  |  1stVariable ← assignVariablesFor1stClaues()
3  |  foreach i ∈ getClauses() do
4  |  |  iVariable ← assignVariablesForiClaues()
5  |  |  mapResult ← mapByKey(getCommonVariables())
6  |  |  joinResult ← join(mapResult)
7  |  joinResult.filter(getSelectVariables())
8  |  result ← result.join(joinResult)
9  return result
```

The ith step discovers all variables in the partitioned data which satisfy the ith clause appeared and keep this intermediate result in-memory with the key being any variable in the ith step which has been introduced on the previous step. During this iteration, the intermediate results are reconstructed in the way that the variables not seen in this iteration are mapped (*line 5*) with the variables of the previous clause and generate a key-value pair of variable bindings. Afterward, the *join* operation is performed over the intermediate results from the previous clause and the new ones with the same key. This process iterates until all clauses are seen and variables are assigned. Finally, the variable binding (*line 7*) to fulfill the *SELECT* clause of the SPARQL query happens and returns the result (*line 8*) of only those variables which are present in the SELECT clause.

4 Evaluation

In our evaluation, we observe the impact of semantic-based partitioning and analyze the scalability of our approach when the size of the dataset increases.

In the following subsections, we present the benchmarks used along with the server configuration setting, and finally, we discuss our findings.

4.1 Experimental Setup

We make use of two well-known SPARQL benchmarks for our experiments: the *Waterloo SPARQL Diversity Test Suite (WatDiv)* v0.6 [3] and *Lehigh University Benchmark (LUBM)* v3.1 [8]. The dataset characteristics of the considered benchmarks are given in Table 1.

WatDiv comes with a test suite with different query shapes which allows us to compare the performance of our approach and the other approaches. In particular, it comes with a predefined set of 20 query templates which are grouped into four categories, based on the query shape: *star-shaped* queries, *linear-shaped* queries, *snowflake-shaped* queries, and *complex-shaped* queries. We have used *WatDiv* datasets with 10M to 100M triples with scale factors 10 and 100, respectively. In addition, we have generated the SPARQL queries using *WatDiv Query Generator*.

LUBM comes with a *Data Generator (UBA)* which generates synthetic data over the *Univ-Bench* ontology in the unit of a university. *LUBM* provides Test Queries, more specifically 14 test queries. Our *LUBM* datasets consist of 1000, 2000, and 3000 universities. The number of triples varies from 138M for 1000 universities, to 414M triples for 3000 universities.

	LUBM			Watdiv	
	1K	2K	3K	10M	100M
#nr. of triples	138,280,374	276,349,040	414,493,296	10,916,457	108,997,714
size (GB)	24	49	70	1.5	15

Table 1. Dataset characteristics (nt format).

We implemented our approach using Spark-2.4.0, Scala 2.11.11, Java 8, and all the data were stored on the HDFS cluster using Hadoop 2.8.0. All experiments were carried out on a commodity cluster of 6 nodes (1 master, 5 workers): Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz (32 Cores), 128 GB RAM, 12 TB SATA RAID-5. We executed each experiment three times and the average query execution time has been reported.

4.2 Preliminary Results

We run experiments on the same cluster and evaluate our approach using the above benchmarks. In addition, we compare our proposed approach with selected state-of-the-art distributed SPARQL query evaluators. In particular, we compare our approach with SHARD [17] – the original approach implemented on Hadoop MapReduce, *SPARQLGX* [6]'s direct evaluator SDE, and Sparklify [21] and report the query execution time (cf. Table 2). We have selected these approaches as they do not include any pre-processing steps (e.g. statistics) while evaluating the SPARQL query, similar to our approach.

	Queries	Runtime (s) (mean) SHARD	SPARQLGX-SDE	SANSA.Sparklify	SANSA.Semantic
Watdiv-10M	$C3$	n/a	38.79	72.94	90.48
	$F3$	n/a	38.41	74.69	n/a
	$L3$	n/a	21.05	73.16	72.84
	$S3$	n/a	26.27	70.1	79.7
Watdiv-100M	$C3$	n/a	181.51	96.59	300.82
	$F3$	n/a	162.86	91.2	n/a
	$L3$	n/a	84.09	82.17	189.89
	$S3$	n/a	123.6	93.02	176.2
LUBM-1K	$Q1$	774.93	103.74	103.57	226.21
	$Q2$	fail	fail	3348.51	329.69
	$Q3$	772.55	126.31	107.25	235.31
	$Q4$	988.28	182.52	111.89	294.8
	$Q5$	771.69	101.05	100.37	226.21
	$Q6$	fail	73.05	100.72	207.06
	$Q7$	fail	160.94	113.03	277.08
	$Q8$	fail	179.56	114.83	309.39
	$Q9$	fail	204.62	114.25	326.29
	$Q10$	780.05	106.26	110.18	232.72
	$Q11$	783.2	112.23	105.13	231.36
	$Q12$	fail	159.65	105.86	283.53
	$Q13$	778.16	100.06	90.87	220.28
	$Q14$	688.44	74.64	100.58	204.43

Table 2. Performance analysis on large-scale RDF datasets.

Our evaluation results for performance analysis, sizeup analysis, node scalability, and breakdown analysis by SPARQL queries are shown in Table 2, Figure 2, Figure 3, and Figure 4 respectively. In Table 2 we use "fail" whenever the system fails to complete the task and "n/a" when the task could not be completed due to a parser error (e.g. not able to translate some of the basic patterns to RDDs operations).

In order to evaluate our approach with respect to the *speedup*, we analyze and compare it with other approaches. This set of experiments was run on three datasets, *Watdiv-10M*, *Watdiv-100M* and *LUBM-1K*.

Table 2 presents the performance analysis of the systems on three different datasets. We can see that our approach evaluates most of the queries as opposed to SHARD. SHARD system fails to evaluate most of the *LUBM* queries and its parser does not support *Watdiv* queries. On the other hand, SPARQLGX-SDE performs better than both Sparklify and our approach, when the size of the dataset is considerably small (e.g. less than 25GB). This behavior is due to the large partitioning overhead for Sparklify and our approach. However, Sparklify performs better compared to SPARQLGX-SDE when the size of the dataset

increases (see *Watdiv-100M* results in the Table 2) and the queries involve more joins (see *LUBM-1K* results in the Table 2). This is due to the Spark SQL optimizer and Sparqlify self-joins optimizers. Both SHARD and SPARQLGX-SDE fail to evaluate query *Q2* in the *LUBM-1K* dataset. Sparklify can evaluate the query but takes longer as compared to our approach. This is due to the fact that our approach uses Spark's lazy evaluation and join optimization by keeping the intermediate results in memory.

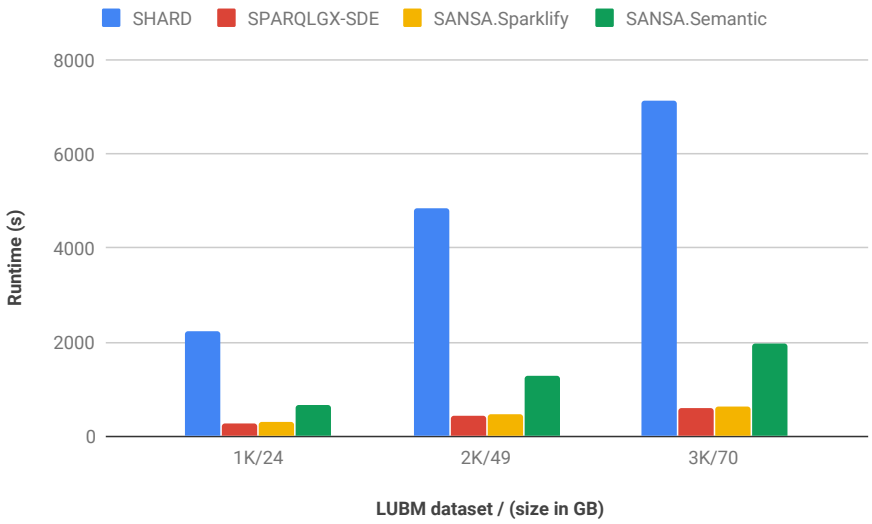

Fig. 2. Sizeup analysis (on LUBM dataset).

Scalability analysis In order to evaluate the scalability of our approach, we conducted two sets of experiments. First, we measure the data scalability (e.g. size-up) of our approach and position it with other approaches. As SHARD fails for most of the LUBM queries, we omit other queries on this set of experiments and choose only Q1, Q5, and Q14. Q1 has been chosen due to its complexity while bringing large inputs of the data and high selectivity, Q5 since it has considerably larger intermediate results due to the triangular pattern in the query, and Q14 mainly for its simplicity. We run experiments on three different sizes of *LUBM* (see Figure 2). We keep the number of nodes constant i.e. 5 worker nodes and increase the size of the datasets to measure whether our approach deals with larger datasets.

We see that the query execution time for our approach grows linearly when the size of the datasets increases. This shows the scalability of our approach as compared to SHARD, in context of the sizeup. SHARD suffers from the expen-

sive overhead of MapReduce joins which impact its performance, as a result, it is significantly worse than other systems.

Second, in order to measure the node scalability of our approach, we increase the number of worker nodes and keep the size of the dataset constant. We vary them from 1, 3 to 5 worker nodes.

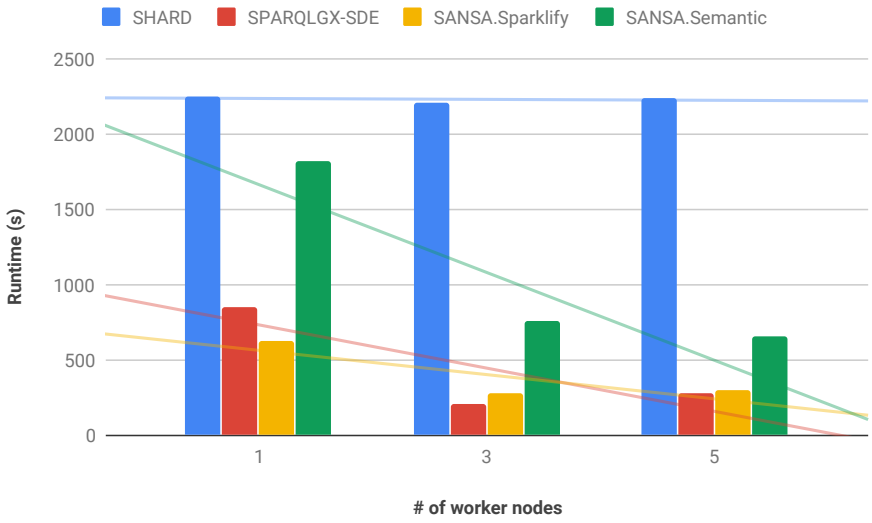

Fig. 3. Node scalability (on LUBM-1K).

Figure 3 shows the performance of systems on *LUBM-1K* dataset when the number of worker nodes varies. We see that as the number of nodes increases, the runtime cost of our query engine decreases linearly as compared with the SHARD, which keeps staying constant. SHARD performance stays constant (high) even when more worker nodes are added. This trend is due to the communication overhead SHARD needs to perform between map and reduce steps. The execution time of our approach decreases about 1.7 times (from 1,821.75 seconds down to 656.85 seconds) as the worker nodes increase from one to five nodes. SPARQLGX-SDE and Sparklify perform better when the number of nodes increases compared to our approach and SHARD.

Our main observation here is that our approach can achieve linear scalability in the performance.

Correctness In order to assess the correctness of the result set, we computed the count of the result set for the given queries and compare it with other approaches. As a result of it, we conclude that all approaches return exactly the same result set. This implies the correctness of the results.

Breakdown by SPARQL queries Here we analyze some of the LUBM queries (Q1, Q5, Q14) run on a *LUBM-1K* dataset in a cluster mode on all the systems.

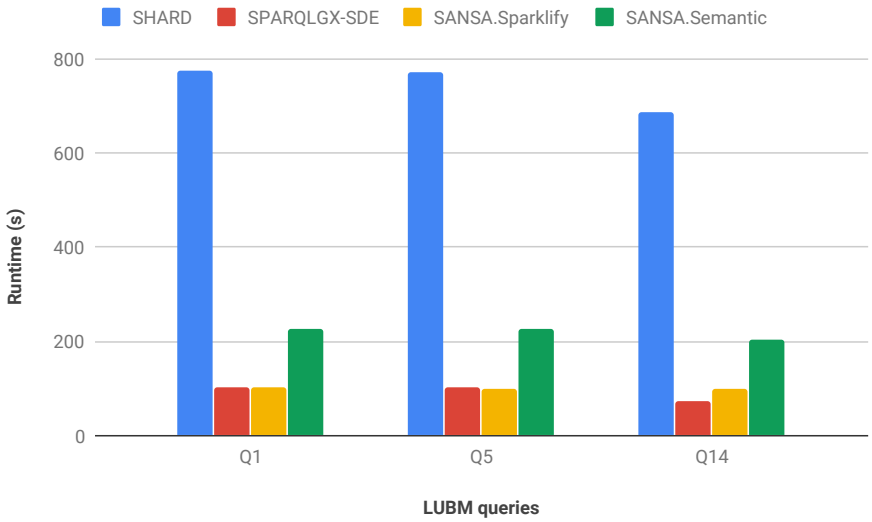

Fig. 4. Overall analysis of queries on LUBM-1K dataset (cluser mode).

We can see from Figure 4 that our approach performs better compared to Hadoop-based system, SHARD. This is due to the use of the Spark framework which leverages the in-memory computation for faster performance. However, the performance declines as compared to other approaches which use vertical partitioning (e.g., SPARQLGX-SDE on RDD and Sparklify on Spark SQL). This is due to the fact that our approach performs de-duplication of triples that involves shuffling and incurs network overhead. The results show that the performance of SPARQLGX-SDE decreases as the number of triple patterns involved in the query increases (see *Q5*) when compared to Sparklify. However, SPARQLGX-SDE performs better when there are simple queries (see *Q14*). This occurs because SPARQLGX-SDE must read the whole RDF graph each time when there is a triple pattern involved. In contrast to SPARQLGX-SDE, Sparklify performs better when there are more triple patterns involved (see *Q5*) but slightly worse when linear queries (see *Q14*) are evaluated.

Based on our findings and the evaluation study carried out in this paper, we show that our approach can scale up with the increasing size of the dataset.

5 Related Work

Partitioning of RDF Data Centralized RDF stores use relational (e.g., Sesame [4]), property (e.g., Jena [23]), or binary tables (e.g., SW-Store [1]) for storing RDF triples or maintain the graph structure of the RDF data (e.g., gStore [26]). For dealing with big RDF datasets, vertical partitioning and exhaustive indexing are commonly employed techniques. For instance, Abadi et al. [2] introduce a vertical partitioning approach in which each predicate is mapped to a two-column table containing the subject and object. This approach has been extended in Hexastore [22] to include all six permutations of subject, predicate, and object (s, p, o). To improve the efficiency of SPARQL queries RDF-3X [14] has adopted exhaustive indices not only for all (s, p, o) permutations but also for their binary and unary projections. While some of these techniques can be used in distributed configurations as well, storing and querying RDF datasets in distributed environments pose new challenges such as the scalability. In our approach, we tackle partitioning and querying of big RDF datasets in a distributed manner.

Partitioning-based approaches for distributed RDF systems propose to partition an RDF graph in fragments which are hosted in centralized RDF stores at different sites. Such approaches use either standard partitioning algorithms like METIS [9] or introduce their own partitioning strategies. For instance, Lee et al. [12] define a partition unit as a vertex with its closest neighbors based on heuristic rules while DiploCloud [24] and AdPart [10] use physiological RDF partitioning based on RDF molecules. In our proposal, we use a semantic-based partitioning approach.

Hadoop-based systems Cloud-based approaches for managing large-scale RDF mainly use NoSQL distributed data stores or employ various partitioning approaches on top of Hadoop infrastructure, i.e., the Hadoop Distributed File System (HDFS) and its MapReduce implementation, in order to leverage computational resources of multiple nodes. For instance, Sempala [19] is a Hadoop-based approach which serves as SPARQL-to-SQL approach on top of Hadoop. It uses Impala[6] as a distributed SQL processing engine. Sempala uses unified vertical partitioning based on a single property table to improve the runtime of the star-shaped queries by excluding the joins. The limitation of Sempala is that it was designed only for that particular shape of the queries. PigSPARQL [18] uses Hadoop based implementation of vertical partitioning for data representation. It translates SPARQL queries into Pig[7] LATIN queries and runs them using the Pig engine. A most recent approach based on MapReduce is RYA [16]. It is a Hadoop based scalable RDF store which uses Accumulo[8] as a distributed key-value store for indexing the RDF triples. One of RYA's advantages is the power of performing join reorder. The main drawback of RYA is that it relies on disk-based processing increasing query execution times. Other RDF systems like

[6] https://impala.apache.org/

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

[8] accumulo.apache.org

JenaHBase [11] and H2RDF+ [15] use the Hadoop database HBase for storing triple and property tables.

SHARD [17] is one approach which groups RDF data into a dedicated partition so-called semantic-based partition. It groups these RDF data by subject and implements a query engine which iterates through each of the clauses used on the query and performs a query processing. A MapReduce job is created while scanning each of the triple patterns and generates a single plan for each of the triple pattern which leads to a larger query plan, therefore, it contains too many Map and Reduces jobs. Our partitioning algorithm is based on SHARD, but instead of creating MapReduce jobs we employ the Spark framework in order to increase scalability.

In-Memory systems S2RDF [20] is a distributed query engine which translates SPARQL queries into SQL ones while running them on Spark-SQL. It introduces a data partitioning strategy that extends vertical partitioning with additional statistics, containing pre-computed semi-joins for query optimization. SPARQLGX [6] is similar to S2RDF, but instead of translating SPARQL to SQL, it maps SPARQL into direct Spark RDD operations. It is a scalable query engine which is capable of evaluating efficiently the SPARQL queries over distributed RDF datasets [7]. It uses a simplified VP approach, where each predicate is assigned to a specific parquet file. As an addition, it is able to assign RDF statistics for further query optimization while also providing the possibility of directly query files on the HDFS using SDE. Recently, Sparklify [21] – a scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets has been proposed. The approach uses Sparqify[9] as a SPARQL to SQL rewriter for translating SPARQL queries into Spark executable code. In our approach, intermediate results are kept in-memory in order to accelerate query execution over big RDF data.

6 Conclusions and Future Work

In this paper, we propose a scalable semantic-based query engine for efficient evaluation of SPARQL queries over distributed RDF datasets. It uses a semantic-based partitioning strategy as the data distribution and converts SPARQL to Spark executable code. By doing so, it leverages the advantages of the Spark framework's rich APIs. We have shown empirically that our approach can scale horizontally and perform well as compared with the previous Hadoop-based system: the SHARD triple store. It is also comparable with other in-memory SPARQL query evaluators when there is less shuffling involved i.e. less duplicate values.

Our next steps include expanding our parser to support more SPARQL fragments and adding statistics to the query engine while evaluating queries. We want to analyze the query performance in the large-scale RDF datasets and explore prospects for the improvement. For example, we intend to investigate the re-ordering of the BGPs and evaluate the effects on query execution time.

[9] `https://github.com/SmartDataAnalytics/Sparqlify`

Acknowledgment

This work was partly supported by the EU Horizon2020 projects Boost4.0 (GA no. 780732), BigDataOcean (GA no. 732310), SLIPO (GA no. 731581), and QROWD (GA no. 723088).

References

1. D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.*, 18(2):385–406, 2009.
2. D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 411–422, 2007.
3. G. Alu, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of rdf data management systems. In *International Semantic Web Conference*, 2014.
4. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, pages 54–68, 2002.
5. I. Ermilov, J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, S. Bin, N. Chakraborty, H. Petzka, M. Saleem, A.-C. N. Ngonga, and H. Jabeen. The Tale of Sansa Spark. In *16th International Semantic Web Conference, Poster & Demos*, 2017.
6. D. Graux, L. Jachiet, P. Genevès, and N. Layaïda. Sparqlgx: Efficient distributed evaluation of sparql with apache spark. In P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, and Y. Gil, editors, *The Semantic Web – ISWC 2016*, pages 80–87, Cham, 2016. Springer International Publishing.
7. D. Graux, L. Jachiet, P. Geneves, and N. Layaïda. A multi-criteria experimental ranking of distributed sparql evaluators. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 693–702. IEEE, 2018.
8. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Semant.*, 3:158–182, 2005.
9. S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing RDF engine based on asynchronous message passing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 289–300, 2014.
10. R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli. Accelerating sparql queries by exploiting hash-based locality and adaptive partitioning. *The VLDB JournalThe International Journal on Very Large Data Bases*, 25(3):355–380, 2016.
11. V. Khadilkar, M. Kantarcioglu, B. Thuraisingham, and P. Castagna. Jena-HBase: A Distributed, Scalable and Efficient RDF Triple Store. In *Proceedings of the 2012th International Conference on Posters &; Demonstrations Track - Volume 914*, ISWC-PD'12, pages 85–88, 2012.
12. K. Lee and L. Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *Proc. VLDB Endow.*, 6(14):1894–1905, Sept. 2013.

13. J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin, N. Chakraborty, M. Saleem, A.-C. N. Ngonga, and H. Jabeen. Distributed semantic analytics using the sansa stack. In *Proceedings of 16th International Semantic Web Conference - Resources Track (ISWC'2017)*, 2017.
14. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
15. N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF+: high-performance distributed joins over large-scale RDF graphs. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 255–263, 2013.
16. R. Punnoose, A. Crainiceanu, and D. Rapp. Rya: A scalable rdf triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, Cloud-I '12, pages 4:1–4:8, New York, NY, USA, 2012. ACM.
17. K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, PSI EtA '10, pages 4:1–4:5, New York, NY, USA, 2010. ACM.
18. A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. Pigsparql: Mapping sparql to pig latin. In *Proceedings of the International Workshop on Semantic Web Information Management*, SWIM '11, pages 4:1–4:8, New York, NY, USA, 2011. ACM.
19. A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen. Sempala: Interactive sparql query processing on hadoop. In P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, and C. Goble, editors, *The Semantic Web – ISWC 2014*, pages 164–179, Cham, 2014. Springer International Publishing.
20. A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2rdf: Rdf querying with sparql on spark. *Proc. VLDB Endow.*, 9(10):804–815, June 2016.
21. C. Stadler, G. Sejdiu, D. Graux, and J. Lehmann. Sparklify: A Scalable Software Component for Efficient evaluation of SPARQL queries over distributed RDF datasets. In *Proceedings of 18th International Semantic Web Conference*, 2019.
22. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.
23. K. Wilkinson. Jena Property Table Implementation. In *SSWS*, pages 35–46, Athens, Georgia, USA, 2006.
24. M. Wylot and P. Cudré-Mauroux. Diplocloud: Efficient and scalable management of RDF data in the cloud. *IEEE Trans. Knowl. Data Eng.*, 28(3):659–674, 2016.
25. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX, 2012.
26. L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: Answering SPARQL queries via subgraph matching. *PVLDB*, 4(8):482–493, 2011.