Sparklify: A Scalable Software Component for Efficient Evaluation of SPARQL Queries over Distributed RDF Datasets

Claus Stadler[1]([✉]), Gezim Sejdiu[2], Damien Graux[3,4], and Jens Lehmann[2,3]

[1] Institute for Applied Informatics (InfAI), University of Leipzig, Leipzig, Germany
`cstadler@informatik.uni-leipzig.de`
[2] Smart Data Analytics, University of Bonn, Bonn, Germany
`{sejdiu,jens.lehmann}@cs.uni-bonn.de`
[3] Enterprise Information Systems, Fraunhofer IAIS, Sankt Augustin, Germany
`{damien.graux,jens.lehmann}@iais.fraunhofer.de`
[4] ADAPT Centre, Trinity College of Dublin, Dublin, Ireland

Abstract. One of the key traits of Big Data is its complexity in terms of representation, structure, or formats. One existing way to deal with it is offered by Semantic Web standards. Among them, RDF – which proposes to model data with triples representing edges in a graph – has received a large success and the semantically annotated data has grown steadily towards a massive scale. Therefore, there is a need for scalable and efficient query engines capable of retrieving such information. In this paper, we propose *Sparklify*: a scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets. It uses Sparqlify as a SPARQL-to-SQL rewriter for translating SPARQL queries into Spark executable code. Our preliminary results demonstrate that our approach is more extensible, efficient, and scalable as compared to state-of-the-art approaches. Sparklify is integrated into a larger SANSA framework and it serves as a default query engine and has been used by at least three external use scenarios.

Resource type Software Framework
Website http://sansa-stack.net/sparklify/
Permanent URL https://doi.org/10.6084/m9.figshare.7963193

1 Introduction

In the recent years, our information society has reached the stage where it produces billions of data records, amounting to multiple quintillion of bytes[1], on a daily basis. Extraction, cleansing, enrichment and refinement of information are key to fuel value-adding processes, such as analytics as a premise for decision making. Devising appropriate (ideally uniform) representations and facilitating efficient querying of data, metadata and provenance arising from such

---

[1] https://www.domo.com/learn/data-never-sleeps-5.

phases constantly poses challenges, especially when data volumes are vast. The most prominent and promising effort is the W3C consortium with encouraging Resource Description Framework (RDF)[2] as a common data representation and vocabularies (e.g. RDFS, OWL) as a way to include meta-information about the data. These data and meta-data can be further processed and analyzed using the de-facto query language for RDF data, SPARQL[3].

SPARQL serves as a standard query language for manipulating and retrieving RDF data. Querying RDF data becomes challenging when the size of the data increases. Recently, many distributed RDF systems capable of evaluating SPARQL queries have been proposed and developed ([7,17]). Nevertheless, these engines lack one important information derived from the knowledge, *RDF terms*. RDF terms includes information about a statement such as *language*, *typed literals* and *blank nodes* which are omitted from most of the engines.

To cover this spectrum requires a specialized system which is capable of constructing an efficient SPARQL query engine. Doing so comes with several challenges. First and foremost, recently the RDF data is increasing drastically. Just as a record, today we count more than 10,0000 datasets[4] available online represented using the Semantic Web standards. This number is increasing daily including many other (e.g. Ethereum[5] dataset) datasets available at the organization premises. In addition, being able to query this large amount of data in an efficient and faster way is a requirement from most of the SPARQL evaluators.

To overcome these challenges, in this paper, we propose *Sparklify*[6]: a scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets. The conceptual foundation is the application of *ontology-based data access* (OBDA) tooling, specifically SPARQL-to-SQL rewriting, for translating SPARQL queries into Spark executable code. We demonstrate our approach using Sparqlify, which has been used in the LinkedGeoData[7] community project to serve more than 30 billion triples on-the-fly from a relational OpenStreetMap database. Our contributions are:

- We present a novel approach for vertical partitioning including RDF terms using the distributed computing framework, Apache Spark.
- We developed a scalable query engine using Sparqlify – a SPARQL-to-SQL rewriter on top of Apache Spark (under the *Apache Licence 2.0*).
- We evaluate our approach with state-of-the-art engines and demonstrate it empirically.
- We integrated the approach into the SANSA [11][8] larger framework. Sparklify serves as a default query engine in SANSA. SANSA is an active project and maintained, including issue tracker, mailing list, changelogs, website, etc.

---

[2] https://www.w3.org/TR/rdf11-primer/.
[3] https://www.w3.org/TR/sparql11-overview/.
[4] http://lodstats.aksw.org/.
[5] https://goo.gl/mJTkPp.
[6] https://github.com/SANSA-Stack/SANSA-Query/tree/develop/sansa-query-spark/src/main/scala/net/sansa_stack/query/spark/sparqlify.
[7] http://linkedgeodata.org.
[8] http://sansa-stack.net/.

The paper is structured as follows: Our approach for data modeling and query translation using a distributed framework is detailed in Sect. 3 and evaluated in Sect. 4. Related work on the SPARQL query engines is discussed in Sect. 6. Finally, we conclude and suggest planned extensions of our approach in Sect. 7.

2 Preliminaries

In this section, we first introduce the basic notions used in throughout the paper.

2.1 Sparqlify

Sparqlify[9] is a SPARQL-to-SQL rewriter that enables answering SPARQL queries on relational databases via a set of view definitions. R2RML[10] and the more intuitive *Sparqlification Mapping Language (SML)*[11] [18] are supported. In general, the rewriter compiles every SPARQL query into two related artifacts: A SQL query and set of SPARQL result variable definitions by means of expressions over the SQL query's result set. Sparqlify first converts the query into an algebra expression. Subsequently, algebraic optimizations and normalizations are applied, such as filter placement and constant folding. Given a query pattern, the view selection component identifies for every triple pattern the set of candidate view definitions together with the renaming of their variables to those of the requesting pattern. This is the base for obtaining the final algebra expression. In general, this involves a cartesian product between triple patterns and views definitions, which leads to a union of joins between the candidate views. Pruning is performed based on RDF term types and IRI prefixes: Choosing a view that binds variables to certain term types or prefixes will constrain subsequent loops only to those candidates with compatible bindings for these variables. Finally, this algebra expression are transformed into an SQL algebra expression using the general relational algebra for RDB-to-RDF mappings. The SQL query, which has been obtained, is used further (e.g. in our case for executing it over Spark SQL engine).

2.2 Apache Spark

Apache Spark is a fast and generic-purpose cluster computing engine which is built over Hadoop ecosystem. Its core data structure are Resilient Distributed Dataset (RDD) [19] which are a fault-tolerant and immutable collections of records that can be operated in a parallel setting. Spark also provides high-level APIs, and tools, including Spark SQL [2] for SQL and structured data processing which allows querying structured data inside Spark programs. In this work, we make use of the above libraries from the Apache Spark stack.

---

[9] https://github.com/SmartDataAnalytics/Sparqlify.
[10] https://www.w3.org/TR/r2rml/.
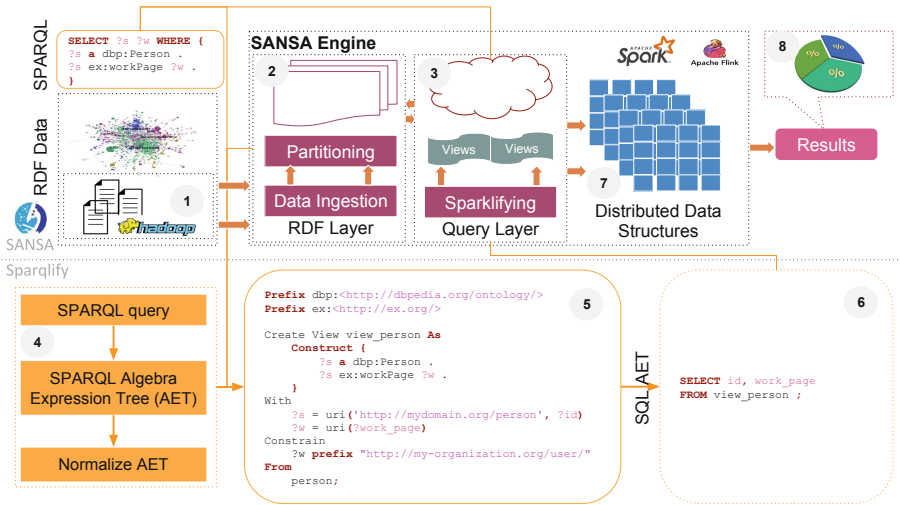[11] http://sml.aksw.org/.

Fig. 1. Sparklify architecture overview.

3   Sparklify

In this section, we present the overall architecture of our proposed approach, the SPARQL-to-SQL rewriter, and mapping to Spark Scala-compliant code.

3.1   System Architecture

The overall system architecture is shown in Fig. 1. It consists of four main components: Data Model, Mappings, Query Translator and Query Evaluator. In the following, each component is discussed in details.

Data Model. SANSA [11] comes with different data structures and different partitioning strategies. We model and store RDF graph following the concept of RDDs – a basic building blocks of the Spark Framework. RDDs are in-memory collections of records which are capable of operating in parallel overall larger cluster. Sparklify makes use of SANSA bottom layer which corresponds with the extended vertical partitioning (VP) including RDF terms. This partition model is the most convenient storage model for fast processing of RDF datasets on top of HDFS.

Data Ingestion (Step 1). RDF data first needs to be loaded into a large-scale storage that Spark can efficiently read from. We use Hadoop Distributed File-System (HDFS)[12]. Spark employ different data locality scheme in order to accomplish computations nearest to the desired data in HDFS, as a result avoiding i/o overhead.

---

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

Data Partition (Step 2). VP approach in SANSA is designed to support extensible partitioning of RDF data. Instead of dealing with a single three-column table (s, p, o), data is partitioned into multiple tables based on the used RDF predicates, RDF term types and literal datatypes. The first column of these tables is always a string representing the subject. The second column always represents the literal value as a Scala/Java datatype. Tables for storing literals with language tags have an additional third string column for the language tag.

Mappings/Views. After the RDF data has been partitioned using the extensible VP (as it has been described on *step 2*) the relational-to-RDF mapping is performed. Sparqlify supports both the W3C standard R2RML sparqlification [18].

The main entities defined with SML are *view definitions*. See *step 5* in the Fig. 1 as an example. The actual view definition is declared by the *Create View ... As* in the first line. The remainder of the view contains these parts: (1) the *From* directive defines the logical table based on the partitioned table (see *step 2*). (2) an RDF template is defined in the *Construct* block containing, URI, blank node or literals constants (e.g. *ex:worksAt*) and variables (e.g. *?emp, ?institute*). The *With* block defines the variables used in the template by means of RDF term constructor expressions whose arguments refer to columns of the logical table.

Query Translation. This process generates a SQL query from the SPARQL query using the bindings determined in the mapping/view construction phases. It walks through the SPARQL query (*step 4*) using Jena ARQ[13] and generates the SPARQL Algebra Expression Tree (AET). Essentially, rewriting SPARQL basic graph patterns and filters over views yields AETs that are UNIONS of JOINS. Further, these AETs are normalized and pruned in order to remove UNION members that are known to yield empty results, such as joins based on IRIs with disjoint sets of known namespaces, or joins between different RDF term types (e.g. literal and IRI). Finally, the SQL is generated (*step 6*) using the bindings corresponding to the views (*step 5*).

Query Evaluator. The SQL query created as described in the previous section can now be evaluated directly into the Spark SQL engine. The result set of this SQL query is distributed data structure of Spark (e.g. DataFrame) (*step 7*) which then is mapped into a SPARQL bindings. The result set can further used for analysis and visualization using the SANSA-Notebooks (*step 8*) [5].

3.2 Algorithm Description

The algorithm described in this paper has been implemented using the Apache Spark framework (see Algorithm 1). It constructs the graph (line 1) while reading RDF data and converts it into RDD of triples. After, it partitions the data (line

---

[13] https://jena.apache.org/documentation/query/.

Algorithm 1. Sparklify algorithm.

---

input : *q*: a SPARQL query, *input*: an RDF dataset
output: *df* list of result set
1 *graph* = *spark*.**rdf**(*lang*)(*input*)
2 *graph.persist*()
3 *partitionGraph* ← *graph*.**partitionGraph**()
4 *result* ← *partitionGraph*.**sparql**(*q*)
5 **return** *result*

Algorithm 2. PartitionGraph algorithm.

---

input : *graph*: an RDD[Triple] dataset
output: *views* a mapped views
1 **foreach** *triple* ∈ *graph* **do**
2   *s* ← *triple.getSubject*; *o* ← *triple.getObject*
3   *subjectType* ← *getRDFTermType*(*s*); *objectType* ← *getRDFTermType*(*o*)
4   *predicate* ← *triple.getPredicate.getURI*
5   **if** *o.isLiteral* **then**
6   **if** *isPlainLiteral(o)* **then**
7   *datatype* ← *XSD.xstring.getURI*
8   **else**
9   *datatype* ← *o.getLiteralDatatypeURI*
10   **else**
11   *datatype* ← *string.Empty*
12   *langTagPresent* ← *isPlainLiteral(o)*
13   *views.add*(*partitioner*(*subjectType*, *predicate*, *objectType*, *datatype*,
14   *langTagPresent*))
15 **return** *views*

3, for more details see Algorithm 2) using the vertical partitioning (VP) strategy. Finally, the query evaluator is constructed (line 4) which is described into more details in Algorithm 3 for consistency.

Partitioning the Graph. The partitioning algorithm (see Algorithm 2) transforms the RDF graph into a convenient VP including RDF terms (line 13). For each triple in the graph in a distributed fashion, it does the following: It gets the RDF terms about subjects and objects (line 3). In case of a literal it assigns the data type for a given column while partitioning the data to: *String* (line 7) when is plain literal, otherwise gets the data type of a given literal (e.g. *Integer*, *Double*) (line 9). The remaining block is the language tag (line 12) which is required for an extra column on the partitioned table containing the language tag value. After all this information is populated, the partitioned block is performed using the *map* transformation function of Spark splitting the tables based on the above information.

Algorithm 3. sparql algorithm.

---
input : views: a Map[partition, RDD[Row]] views, q: a SPARQL query
output: df a data frame with the rewritten SPARQL query's result set

1 $vds \leftarrow emptyList()$
2 **foreach** $(v, rdd) \in views$ **do**
3 $\quad$ $vd \leftarrow Sparqlify.createViewDefinition(v)$
4 $\quad$ $tableName \leftarrow vd.logicalTableName$
5 $\quad$ $scalaSchema \leftarrow v.layout.schema$
6 $\quad$ $sparkSchema \leftarrow ScalaReflection.schemaFor(scalaSchema).dataType$
7 $\quad$ $df \leftarrow spark.createDataFrame(rdd, sparkSchema)$
8 $\quad$ $df.createOrReplaceTempView(vd.logicalTableName)$
9 $\quad$ $vds.add(vd)$
10 $rewriter \leftarrow Sparkqlify.createDefaultSparqlSqlStringRewriter(vds)$
11 $rewrite \leftarrow rewriter.rewrite(q)$
12 $sqlQueryStr \leftarrow rewrite.sqlQueryString$
13 $df \leftarrow spark.sql(sqlQueryStr)$
14 **return** df

Querying the Graph. Given a SPARQL query and a set of partitions together with associated RDDs, Sparklify first has to create OBDA view definitions from the partitions (line 3) and register their corresponding RDDs with names that can be referenced from Spark SQL (line 8). Hence, the algorithm collects the schema (line 6) and constructs a logical table name (line 4) based on the partitions. The final step is to create a Spark data frame (line 13) from the SQL query that is part of the rewrite object generated by Sparqlify (line 12).

4 Evaluation

The goal of our evaluation is to observe the impact of the extensible VP as well as analyzing its scalability when the size of the datset increases. At the same time, we also want to measure the effect of using Sparqlify optimizer for improving the query performance. Especially, we want to verify and answer the following questions:

(Q1) : Is the runtime affected when more nodes are added in the cluster?
(Q2) : Does it scale to a larger dataset?
(Q3) : How does it scale when adding a larger number of datasets?

In the following, we present our experiments setting including the benchmarks used and server configurations. Afterword, we elaborate on our findings.

4.1   Experimental Setup

We used two well-known SPARQL benchmarks for our evaluation. The *Lehight University Benchmak (LUBM)* v3.1 [9] and *Waterloo SPARQL Diversity Test Suite (WatDiv)* v0.6 [1]. Characteristics of the considered datasets are given in Table 1.

   LUBM comes with a *Data Generator (UBA)* which generates synthetic data over the *Univ-Bench* ontology in the unit of a university. Our *LUBM* datasets consist of 1000, 5000, and 10000 universities. The number of triples varies from 138M for 1000 universities, to 1.4B triples for 10000 universities. *LUBM*'s test suite is comprised of 14 queries.

   We have used *WatDiv* datasets with approximate 10K to 1B triples with scale factors 10, 100 and 1000, respectively. *WatDiv* provides a test suite with different query shapes, therefore, it allows us to compare the performance of Sparklify and the other approach we compare with in a more compact way. We have generated these queries using the *WatDiv Query Generator* and report the average mean runtime in the overall results presented below. It comes with a set of 20 predefined query templates so-called *Basic Testing Use Case* which is grouped into four categories, based on the query shape: *star (QS), linear (QL), snowflake (QF)*, and *complex (QC)*.

Table 1. Summary information of used datasets (nt format).

$\longrightarrow$	LUBM			Watdiv		
	1K	5K	10K	10M	100M	1B
#nr. of triples	138,280,374	690,895,862	1,381,692,508	10,916,457	108,997,714	1,099,208,068
Size (GB)	24	116	232	1.5	15	149

   We implemented Sparklify using Spark-2.4.0, Scala 2.11.11, Java 8, and Sparqlify 0.8.3 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 7 nodes (1 master, 6 workers): Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10 GHz (32 Cores), 128 GB RAM, 12 TB SATA RAID-5, connected via a Gigabit network. The experiments have been executed three times and the average runtime has been reported into the results.

4.2   Results

We evaluate Sparklify using the above datasets and compare it with the chosen state-of-the-art distributed SPARQL query evaluator. Since our approach does not involve any pre-processing of the RDF data before being able to evaluate SPARQL queries on it, Sparklify is thereby closer to the so-called direct evaluators. Indeed, Sparklify only needs to virtually partition the data prior. As a consequence, we omit other distributed evaluators (such as e.g. S2RDF [17]) and

compare it with SPARQGX [7] as it outperforms other approaches as noted by Graux et al. [7]. We compare our approach with *SPARQLGX*'s direct evaluator named SDE and report the loading time for partitioning and query execution time, see Table 2. We specify "fail" whenever the system fails to complete the task and "n/a" when the task could not be completed due to a failure in one of the intermediate phase. In some cases e.g. in Table 2, *QC in Watdiv-1B* dataset, we define "partial fail" due to the failure of one of the queries, therefore the sum-up is not possible.

Findings of the experiments are depicted in Table 2, Figs. 2, 3, and 4.

To verify Q1, we analyze the *speedup* and compare it with SPARQLGX. We run the experiments on three datasets, *Watdiv-10M*, *Watdiv-1B* and *LUBM-10K*.

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

		Runtime (s) (mean)			
		SPARQLGX-SDE	Sparklify		
	$\longrightarrow$	a) total	b) paritioning	c) querying	d) total
Watdiv-10M	QC	103.24	134.81	61	195.84
	QF	157.8	241.24	107.33	349.51
	QL	102.51	236.06	134	370.3
	QS	131.16	237.12	108.56	346
Watdiv-1B	QC	partial fail	778.62	2043.66	2829.56
	QF	6734.68	1295.31	2576.52	3871.97
	QL	2575.72	1275.22	610.66	1886.73
	QS	4841.85	1290.72	1552.05	2845.3
LUBM-10K	$Q1$	1056.83	627.72	718.11	1346.8
	$Q2$	fail	595.76	fail	n/a
	$Q3$	1038.62	615.95	648.63	1267.37
	$Q4$	2761.11	632.93	1670.18	2303.18
	$Q5$	1026.94	641.53	564.13	1206.67
	$Q6$	537.65	695.74	267.48	963.62
	$Q7$	2080.67	630.44	1331.13	1967.25
	$Q8$	2636.12	639.93	1647.57	2288.48
	$Q9$	3124.52	583.86	2126.03	2711.24
	$Q10$	1002.56	593.68	693.73	1287.71
	$Q11$	1023.32	594.41	522.24	1118.58
	$Q12$	2027.59	576.31	1088.25	1665.87
	$Q13$	1007.39	626.57	6.66	633.26
	$Q14$	526.15	633.39	258.32	891.89

Table 2 shows the performance analysis of two approaches run on three different datasets. Column SPARQLGX-SDE[a] reports on the performance of

SPARQLGX-SDE considering the total runtime to evaluate the given queries. Column Sparklify[b] lists the times required for Sparklify to perform the VP and then the query execution time is reported on the Sparklify[c]. Total runtime for Sparklify is shown in the last column, Sparklify[d].

We observe that the execution of both approaches fails for the *Q2* in the *LUBM-10K* dataset while evaluating the query. We believe that it is due to the reason that *LUBM Q2* involves a triangular pattern which is often resource consuming. As a consequence, in both cases, Spark performs the shuffling (e.g. data scanning) while reducing the result set. It is interesting to note that for the *Watdiv-1B* dataset, SPARQLGX-SDE fails for the query *C3* when data scanning is performed. Sparklify is capable of evaluating it successfully. Due to the Spark SQL optimizer in conjunction with Sparqlify's approach of rewriting a SPARQL query typically into only a single SQL query – effectively offloading all query planning to Spark – Sparklify performs better than SPARQLGX-SDE when the size of the dataset increases (see *Watdiv-1B results* in the Table 2) and when there are more joins involved (see *Watdiv-1B* and *LUBM-10K* results in the Table 2). SPARQLGX-SDE evaluates the queries faster when the size of the datasets is smaller, but it degrades when the size of the dataset increases. The likely reason for Sparklify's worse performance on smaller datasets is its higher partitioning overhead. Figure 2 shows that Sparklify starts outperforming when the size of the datasets grows (e.g. *Watdiv-100M*).
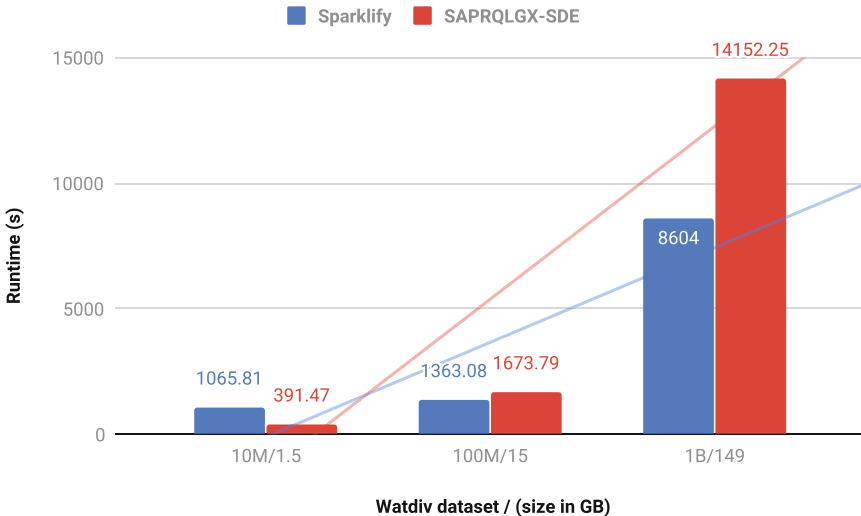

Fig. 2. Sizeup analysis (on Watdiv dataset).

Size-up Scalability Analysis. To measure the performance of the data scalability (e.g. size-up) of both approaches, we run experiments on three different sizes of *Watdiv* (see Fig. 2). We keep the number of nodes constant i.e 6 worker

nodes and grow the size of the datasets to measure whether both approaches can deal with larger datasets. We see that the execution time for Sparklify grows linearly compared with SPARQLGX-SDE, which keeps staying as near-linear when the size of the datasets increases. The results presented show scalability of Sparklify in context of the sizeup, which addresses the question Q2.

Node Scalability Analysis. To measure the node scalability of Sparklify, we vary the number of worker nodes. We vary them from 1, 3 to 6 worker nodes. Figure 3 depict the speedup performance of both approaches run on *Watdiv-100M* dataset when the number of worker nodes varies. We can see that as the number of nodes increases, the runtime cost for the Sparklify decrease linearly. The execution time for Sparklify decreases about 0.6 times (from 2547.26 s down to 1588.4 s) as worker nodes increase from one to three nodes. We see that the speedup stays constant when more worker nodes are added since the size of the data is not that large and the network overhead increases a little the runtime when it runs over six worker nodes. This imply that our approach is efficient up to three worker nodes for the *Watdiv-100M* (15GB) dataset. In another hand, SPARQLGX-SDE takes longer to evaluate the queries when running on one worker node but it improves when the number of worker nodes increases.

Result presented here shows that Sparklify can achieve linear scalability in the performance, which addresses Q3.
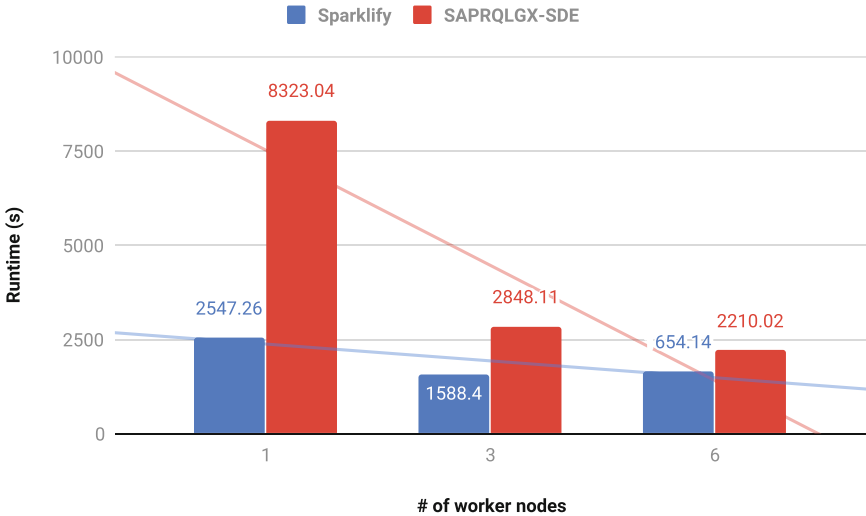

Fig. 3. Node scalability (on Watdiv-100M).

Correctness of the Result Set. 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 within both approaches. We conclude that both approaches return exactly the same result set which implies the correctness of the results.

Overall Analysis by SPARQL Queries. Here we analyze Watdiv queries
run on *Watdiv-100M* dataset in a cluster mode on both approaches.
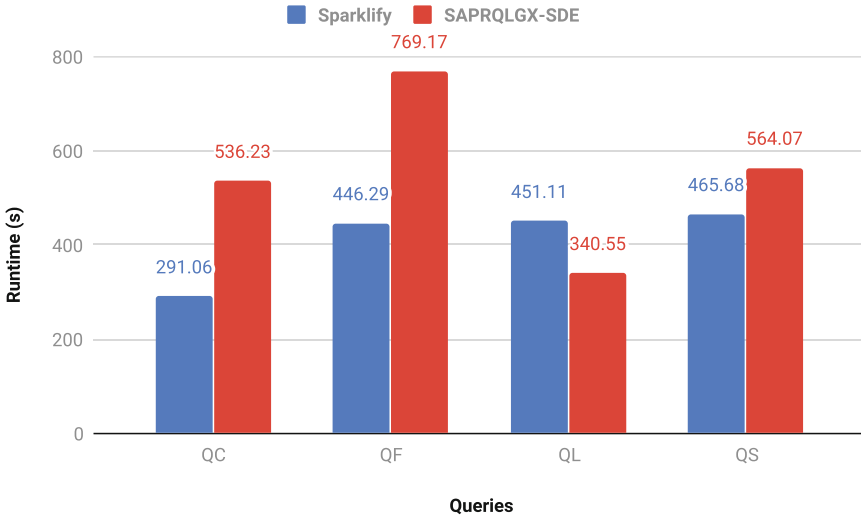

Fig. 4. Overall analysis of queries on Watdiv-100M dataset (cluster mode).

According to Fig. 4, SPARQLGX-SDE performance decreases as the number
of triple patterns involved in the query increase. This might be due to the fact
that SPARQLGX-SDE has to read the whole triple file each time. In contrast to
SPARQLGX-SDE, Sparklify seems to perform well when there are more triple
pattern involved (see queries *QC*, *QF* and *QS* in the Fig. 4) but slightly worst
when there are linear queries (see *QL*) evaluated. This may be due to the reason
that Sparqlify typically rewrites a SPARQL query into a single SQL query, thus
maximizing the opportunities given to the Spark SQL optimizer. Conversely,
SPARQLGX-SDE constructs the workflow by chaining Scala API calls, which
may restrict the possibilities e.g. in regard to join ordering. Based on our findings
and the evaluation study carried out in this paper, we show that Sparklify is
scalable and the execution time ends in a reasonable time given the size of the
dataset.

5   Use Cases

Sparklify, as a default query engine for SANSA has been used in different major
use cases. Below, we list some of them that we are aware of using Sparklify:
Blockchain – Alethio Use Case. Alethio[14] try to present the big picture of
the whole Ethereum ecosystem. It is a powerful blockchain data, analytics, and

---

[14] https://aleth.io/.

visualisation platform. It contains more than 18 Billion triples datasets "rdfized" using the structure of the Ethereum ontology[15]. They are taking advantage of the SANSA stack by querying this amount of data at scale e.g. analyzing the Hubs & Authorities in the Ethereum Transaction Network[16] and other analytics. ***SPECIAL – A Semantic Transparency and Compliance Use Case.*** SPECIAL[17] is a Scalable Policy-aware Linked Data platform for privacy, transparency, and compliance. Within the project, they introduce SPIRIT – a transparency and compliance checking implementation of the SANSA stack. SPECIAL uses SANSA engine in order to analyze the log information concerning personal data processing and sharing that as an output from line of business applications on a continuous basis, and to present the information to the user via the SPIRIT dashboard. The SPIRIT transaction log processing allows users to: (1) define the set of policies rules, (2) initialize the query engine with the log and schema/ontology data, here is where Sparklify is used in specific, (3) create a reasoner set reasoning profile, and (4) apply these rules to the given query in order be compliant with the policy rules.

SLIPO – Categorizing Areas of Interests (AOI) Use Case. SLIPO[18] take advantage of the Semantic Web Technologies for the scalable and efficient integration of Big Point of Interest (POI) datasets. In particular, the project focuses on designing efficient pipelines dealing with large semantic datasets of POIs: a wide range of features are available inter alia fusion & cleaning distinct datasets or detection of future "hot" AOIs where businesses should be created. In this project, Sparklify is used through the SANSA query layer to refine, filter and select the relevant POIs which are needed by the pipelines.

6   Related Work

As our main focus is on the area of distributed computing, we omit the centralized systems e.g. RDF-3X [12] or Virtuoso [4] (see [6] for a survey) and we review the distributed ones only (see [10] for a recent survey). Further, this set of tools is divided into: *MapReduce-based* systems and *In-Memory* systems e.g. on top of Apache Spark.

   MapReduce systems – SHARD [14] 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. PigSPARQL [15] is yet another approach which uses Hadoop based implementation of vertical partitioning for data representation. It translated the SPARQL queries into Pig[19] LATIN queries

---

and uses Pig as an intermediate engine. Another approach which is based on the MapReduce is Sempala [16] – as SPARQL-to-SQL approach on top of Hadoop. It uses Impala[20] as a distributed SQL processing engine. Sempala uses a so-called unified vertical partitioning (single property table) in order to boost the star-shaped queries by excluding the joins. Hence, its limitation is that it is designed only to that particular shape of the queries. RYA [13] is a Hadoop based scalable RDF store that uses Accumulo[21] as a distributed key-value store for indexing the RDF triples. RYA indexes triples into three tables and replicate them across the cluster for leveraging the indexes over all the possible records. It has the mechanism of performing join reorder, but it lacks of the in-memory computation, which makes it not comparable with other systems. While the MapReduce paradigm has been realized for disk-based as well as in-memory processing, the concept is not concerned with controlling aspects of general distributed workflows, such as which intermediate results to cache. As a consequence, high level frameworks were devised which may use MapReduce as a building block. Apache Spark is one of them [19]. Below, we will list some of the approaches which make use of the Apache Spark (in-memory computation) framework.

In-Memory systems – SPARQLGX [7] and S2RDF [17] approaches are considered the most recent distributed SPARQL evaluators over large-scale RDF datasets. SPARQLGX is a scalable query engine which is capable of evaluating efficiently the SPARQL queries over distributed RDF datasets [8]. It provides a mechanism for translating SPARQL queries into Spark executable code for better leveraging the advantage of the Spark framework. It uses a simplified VP approach, where each predicate is assigned with 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. S2RDF is similar to SPARQLGX, but instead of dealing with direct Spark code (aka RDDs), it translates SPARQL queries into SQL ones run by Spark-SQL. It introduces a data partitioning strategy that extends VP with additional statistics, containing pre-computed semi-joins for query optimization.

7   Conclusions and Future Work

Querying RDF data becomes challenging when the size of the data increases. Existing Spark-based SPARQL systems mostly do not retain all RDF term information consistently while transforming them to a dedicated storage model such as using vertical partitioning. Often, this process is both data and computing intensive and raises the need for a scalable, efficient and comprehensive query engine which can handle large scale RDF datasets.

In this paper, we propose *Sparklify*: a scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets. It uses Sparqify as a SPARQL-to-SQL rewriter for translating SPARQL queries into Spark executable code. By doing so, it leverages the advantages of the Spark framework.

---

[20] https://impala.apache.org/.
[21] https://accumulo.apache.org.

SANSA features methods to execute SPARQL queries directly as part of Spark workflows instead of writing the code corresponding to those queries (sorting, filtering, etc.). It also provides a command-line interface and a W3C standard compliant SPARQL endpoint for externally querying data that has been loaded using the SANSA framework. We have shown empirically that our approach can scale horizontally and perform well w.r.t to the state-of-the-art approaches.

With this work, we showed that the application of OBDA tooling to Big Data frameworks achieves promising results in terms of scalability. We present a working prototype implementation that can serve as a baseline for further research. Our next steps include evaluating other tools, such as Ontop [3], and analyze how their performance in the Big Data setting can be improved further. For example, we intend to investigate how OBDA tools can be combined with dictionary encoding of RDF terms as integers and evaluate the effects.

References

1. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: Mika, R., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 197–212. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_13
2. Armbrust, M., et al.: Spark SQL: relational data processing in Spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD 2015, pp. 1383–1394. ACM, New York (2015)
3. Calvanese, D., et al.: Ontop: answering SPARQL queries over relational databases. Semant. Web **8**, 471–487 (2017)
4. Erling, O., Mikhailov, I.: Virtuoso: RDF support in a native RDBMS. In: de Virgilio, R., Giunchiglia, F., Tanca, L. (eds.) Semantic Web Information Management, pp. 501–519. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-04329-1_21
5. Ermilov, I., et al.: The Tale of Sansa Spark. In 16th International Semantic Web Conference, Poster & Demos (2017)
6. Faye, D.C., Curé, O., Blin, G.: A survey of RDF storage approaches. Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées **15**, 11–35 (2012)
7. Graux, D., Jachiet, L., Genevès, P., Layaïda, N.: SPARQLGX: efficient distributed evaluation of SPARQL with Apache Spark. In: Groth, P., et al. (eds.) The Semantic Web - ISWC 2016. LNCS, pp. 80–87. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-46547-0_9
8. Graux, D., Jachiet, L., Geneves, P., Layaïda, N.: A multi-criteria experimental ranking of distributed SPARQL evaluators. In: 2018 IEEE International Conference on Big Data (Big Data), pp. 693–702. IEEE (2018)

9. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for owl knowledge base systems. J. Web Semant. **3**, 158–182 (2005)
10. Kaoudi, Z., Manolescu, I.: RDF in the clouds: a survey. VLDB J.-Int. J. Very Large Data Bases **24**(1), 67–91 (2015)
11. Lehmann, J., et al.: Distributed semantic analytics using the SANSA stack. In: d'Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10588, pp. 147–155. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_15
12. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proc. VLDB Endow. **1**(1), 647–659 (2008)
13. Punnoose, R., Crainiceanu, A., Rapp, D.: Rya: a scalable RDF triple store for the clouds. In: Proceedings of the 1st International Workshop on Cloud Intelligence, Cloud-I 2012, pp. 4:1–4:8. ACM, New York (2012)
14. Rohloff, K., Schantz, R.E.: 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 2010, pp. 4:1–4:5. ACM, New York (2010)
15. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: mapping SPARQL to Pig Latin. In: Proceedings of the International Workshop on Semantic Web Information Management, SWIM 2011, pp. 4:1–4:8. ACM, New York (2011)
16. Schätzle, A., Przyjaciel-Zablocki, M., Neu, A., Lausen, G.: Sempala: interactive SPARQL query processing on Hadoop. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 164–179. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_11
17. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on Spark. Proc. VLDB Endow. **9**(10), 804–815 (2016)
18. Stadler, C., Unbehauen, J., Westphal, P., Sherif, M.A., Lehmann, J.: Simplified RDB2RDF mapping. In: Proceedings of the 8th Workshop on Linked Data on the Web, LDOW 2015, Florence, Italy (2015)
19. Zaharia, M., et al.: 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)