SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark

Damien Graux[312], Louis Jachiet[312], Pierre Genevès[213], and Nabil Layaïda[123]

[1] Inria, France
{damien.graux,louis.jachiet,nabil.layaida}@inria.fr
[2] CNRS, LIG, France
pierre.geneves@cnrs.fr
[3] Univ. Grenoble Alpes, France

Abstract. SPARQL is the W3C standard query language for querying data expressed in the Resource Description Framework (RDF). The increasing amounts of RDF data available raise a major need and research interest in building efficient and scalable distributed SPARQL query evaluators. In this context, we propose SPARQLGX: our implementation of a distributed RDF datastore based on Apache Spark. SPARQLGX is designed to leverage existing Hadoop infrastructures for evaluating SPARQL queries. SPARQLGX relies on a translation of SPARQL queries into executable Spark code that adopts evaluation strategies according to (1) the storage method used and (2) statistics on data. We show that SPARQLGX makes it possible to evaluate SPARQL queries on billions of triples distributed across multiple nodes, while providing attractive performance figures. We report on experiments which show how SPARQLGX compares to related state-of-the-art implementations and we show that our approach scales better than these systems in terms of supported dataset size. With its simple design, SPARQLGX represents an interesting alternative in several scenarios.

Keywords: RDF System, Distributed SPARQL Evaluation

1 Introduction

SPARQL is the standard query language for retrieving and manipulating data represented in the Resource Description Framework (RDF) [11]. SPARQL constitutes one key technology of the semantic web and has become very popular since it became an official W3C recommendation [1].

The construction of efficient SPARQL query evaluators faces several challenges. First, RDF datasets are increasingly large, with some already containing more than a billion triples. To handle efficiently this growing amount of data, we need systems to be distributed and to scale. Furthermore, semantic data often have the characteristic of being dynamic (frequently updated). Thus being able to answer quickly after a change in the input data constitutes a very desirable property for a SPARQL evaluator. In this context, we propose SPARQLGX: an engine designed

to evaluate SPARQL queries based on Apache Spark [21]: it relies on a compiler of SPARQL conjunctive queries which generates Scala code that is executed by the Spark infrastructure. The source code of our system is available online from the following URL: `https://github.com/tyrex-team/sparqlgx`.

The paper is organized as follows: we first introduce the technologies that we consider in Section 2. Then, in Section 3, we describe SPARQLGX and present additional available tools in Section 4. Section 5 reports on our experimental validation to compare our implementation with other open source HDFS-based RDF systems. Finally, we review related works in Section 6 and conclude in Section 7.

2 Background

The Resource Description Framework (RDF) is a language standardized by W3C to express structured information on the Web as graphs [11]. It models knowledge about arbitrary resources using Unique Resource Identifiers (URIs), Blank Nodes and Literals. RDF data is structured in sentences –or triples– written ($s\ p\ o$), each one having a subject s, a predicate p and an object o.

The SPARQL standard language has been studied under various forms and fragments. We focus on the problem of evaluating the Basic Graph Pattern (BGP) fragment over a dataset of RDF triples. The BGP fragment is composed of conjunctions of triple patterns (TPs). A candidate solution is a mapping from the variables of the query towards values, a candidate solution satisfies a TP when the replacement of the variables of the TP with their value corresponds to a triple that appears in the RDF data. A query solution is a candidate solution that satisfies all the TPs of the query.

Apache Hadoop[1] is a framework for distributed systems based on the Map-Reduce paradigm. The Hadoop Distributed File System (HDFS) is a popular distributed file system handling the distribution of data across a cluster and its replication [19].

Apache Spark [21] is a MapReduce-like data-parallel framework designed for large-scale data processing running on top of the JVM. Spark can be set up to use HDFS.

3 SPARQLGX: General Architecture

In this Section, we explain how we translate queries from our SPARQL fragment into lower-level Scala code [14] which is directly executable with the Spark API. To this end, after presenting the chosen data storage model, we give a translation into a sequence of Spark-compliant Scala-commands for each operator of the considered fragment.

[1] Apache Hadoop: http://hadoop.apache.org

3.1 Data Storage Model

In order to process RDF datasets with Apache Spark, we first have to adopt a convenient storage model on the HDFS. From a "raw" storage (*e.g.* a file in the N-Triple standard which is a simple list of all triples) to complex schemes (*e.g.* involving indexes or B-trees), there are many ways to store RDF data. Any storage choice is a compromise between **(1)** the time required for converting origin data into the target format, **(2)** the total disk-space needed, **(3)** the possible response time improvement induced.

RDF triples have very specific semantics. In a RDF triple (s p o), the predicate p represents the "semantic relationship" between the subject s and the object o. Thus, there are often relatively few distinct predicates compared to the number of distinct subjects or objects. The vertically partitioned architecture introduced by Abadi *et al.* in [2] takes advantage of this observation by storing the triple (s p o) in a file named p whose contents keeps only s and o entries.

(1) Converting RDF data into a vertically partitioned dataset does not involve complex computation: each triple is read once and the pair (subject, object) is appended to the predicate file.

(2) For large datasets with only a few predicates, two URIs are stored instead of three which reduce the memory footprint compared with the input dataset.

(3) Having vertically partitioned data reduces evaluation time of triple patterns whose predicate is a constant (*i.e.* not a variable): searches are limited to the relevant files. In practice, one can observe that most SPARQL queries have triple patterns with a constant predicate. [7] showed that graph patterns where all predicates are constant represent 77.81% of the queries asked to DBpedia and 98.08% of the ones asked to SWDF.

We believe that vertical partitionning is very well suited for RDF: it implies a pass over the data but with only simple computation, reduces the size of the dataset and provides an indexation.

3.2 SPARQL Fragment Translation

We compute the solution of a conjunction of TPs recursively. Given a conjunction of n TPs we recursively compute the set of solution for the $n-1$ first TPs and then we combine this set with the solutions of the last TP by joining them on their common variables.

To compute the solutions for a unique TP: when the predicate is a constant, we open the relevant HDFS file using `textFile`; otherwise, we have to open all predicate files. Then, using the constants of the TP, we use a `filter` to keep only the matching elements. Finally, we use the variables names appearing in the TP for variables. For instance, the following TP {?s age 21 .} matching people that are 21 years old is translated into:

```
val tp=sc.textFile("age.txt")
         .filter{case(s,obj)=>obj==21}
```

In order to translate a conjunction of TPs (*i.e.* a BGP), the TPs are joined. Two set of partial solutions are joined using their common variables as a key: `keyBy` in Spark. Joining TPs is then realized with `join` in Spark. For example the following TPs {?s age 21 . ?s gender ?g .} are translated into:

```
val tp1=sc.textFile("age.txt")
          .filter{case(s,obj)=>obj==21}
          .keyBy{case(s,obj)=>s}
val tp2=sc.textFile("gender.txt")
          .keyBy{case(s,g)=>s}
val bgp=tp2.join(tp1).values
```

A join with no common variables corresponds to a cross product (therefore a `cartesian` in Spark). For a conjunction of n TPs we perform $(n-1)$ joins.

The obtained translation (the Scala-code) thus depends on the initial order of TPs since the joins will be perfomed in the same order. This allows us to develop optimizations based on join commutativity such as the ones presented in Section 4.1.

3.3 SPARQL Fragment Extension

Once the TPs are translated, we use a `map` to retain only the desired fields (*i.e.* the distinguished variables) of the query. At that stage, we can also modify results according to the SPARQL solution modifiers [1] (*e.g.* removing duplicates with `distinct`, sorting with `sortByKey`, returning only few lines with `take`, etc.)

Furthermore, we also easily translate two additional SPARQL keywords: UNION and OPTIONAL, provided they are located at top-level in the WHERE clauses. Indeed, Spark allows to aggregate sets having similar structures with `union` and is also able to add data if possible with `leftOuterJoin`. Thus SPARQLGX natively supports a slight extension (UNIONs and OPTIONALs at top level) of the extensively studied SPARQL fragment made of conjunctions of triple patterns.

4 Additional Features

4.1 Optimized Join Order With Statistics

The evaluation process (using Spark) first evaluates TPs and then joins these subsets according to their common variables; thus, minimizing the intermediate set sizes involved in the join process reduces evaluation time (since communication between workers is then faster). Thereby, statistics on data and information on intermediate results sizes provide useful information that we exploit for optimisation purposes.

Given an RDF dataset $\mathcal{D}$ having T triples, and given a place in an RDF sentence $k \in \{subj, pred, obj\}$, we define the selectivity in $\mathcal{D}$ of an element e located at k as: (1) the occurrence number of e as k in $\mathcal{D}$ if e is a constant; (2) T if e is a variable. We note it $sel^k_{\mathcal{D}}(e)$. Similarly, we define the selectivity of a TP (a b c .) over an RDF dataset $\mathcal{D}$ as: $SEL_{\mathcal{D}}(a,b,c) = \min(sel^{subj}_{\mathcal{D}}(a)\,,\, sel^{pred}_{\mathcal{D}}(b)\,,\, sel^{obj}_{\mathcal{D}}(c))$.

Dataset	Number of Triples	Original File Size on HDFS
Watdiv-100M	109 million	46.8 GB
Lubm-1k	134 million	72.0 GB
Lubm-10k	1.38 billion	747 GB

Table 1: General Information about Used Datasets.

Thereby, to rank each TP, we compute statistics on datasets counting all the distinct subjects, predicates and objects. This is implemented in a compile-time module that sorts TPs in ascending order of their selectivities before they are translated.

Finally, we also want to avoid cartesian products. Given an ordered list l of TPs we compute a new list l' by repeating the following procedure: remove from l and append to l' the first TP that shares a variable with a TP of l'. If no such TP exists, we take the first.

4.2 SDE: SPARQLGX as a Direct Evaluator

Our tool evaluates SPARQL queries using Apache Spark after preprocessing RDF data. However, in certain situations, data might be dynamic (*e.g.* subject to updates) and/or users might only need to evaluate a single query (*e.g.* when evaluation is integrated into a pipeline of transformations). In such cases, it is interesting to limit as much as possible both the preprocessing time and the query evaluation time.

To take the original triple file as source, we only have to modify in our translation process the way we treat TPs to change our storage model. Instead of searching in predicate files, we directly use the initial file; and the rest of the translation process remains the same. We call this variant of our evaluator the "direct evaluator" or SDE.

5 Experimental Results

In this Section, we present an excerpt of our empirical comparison of our approach with other open source HDFS-based RDF systems. RYA [16] relies on key-value tables using Apache Accumulo[2]. CliqueSquare [8] converts queries in a Hadoop list of instructions. S2RDF [18] is a recent tool that allow to load RDF data according to a novel scheme called ExtVP and then to query the relational tables using Apache SparkSQL [4]. Finally, PigSPARQL [17] just translates SPARQL queries into an executable PigLatin [15] instruction sequence; and RDFHive[3] is a straightforward tool we made to evaluate SPARQL conjunctive queries directly on Apache Hive [20] after a naive translation of SPARQL into Hive-QL.

[2] Apache Accumulo: accumulo.apache.org
[3] RDFHive Sources: http://tyrex.inria.fr/rdfhive/home.html

All experiments are performed on a cluster of 10 Virtual Machines (VM) distributed on two physical machines (each one running 5 of them). The operating system is CentOS-X64 6.6-final. Each VM has 17 GB of memory and 4 cores at 2.1 GHz. We kept the default setting with which HDFS is resilient to the loss of two nodes and we do not consider the data import on the HDFS as part of the preprocessing phase.

We compare the presented systems using two popular benchmarks: LUBM [9] and Watdiv [3]. Table 1 presents characteristics of the considered datasets. We rely on three metrics to discuss results (Table 2): query execution times, preprocessing times (for systems that need to preprocess data), and disk footprints. For space reasons, Table 2 presents three Lubm queries: Q1 because it bears large input and high selectivity, Q2 since it has large intermediate results while involving a triangular pattern and Q14 for its simplicity. Moreover, we aggregate Watdiv queries by the categories proposed in the Watdiv paper [3]: 3 complex (QC), 5 snowflake-shaped (QF), 5 linear (QL) and 7 star queries (QS). In Table 2 we indicate "TIMEOUT" whenever the process did not complete within a certain amount of time[4]. We indicate "FAIL" whenever the system crashed before this timeout delay. This regroups several kinds of failure such as unability of evaluating queries and also unability of preprocessing the datasets. We indicate "N/A" whenever the task could not be accomplished because of a failure during the preprocessing phase.

Table 2 shows that SPARQLGX always answer all tested queries on all tested datasets whereas this is not the case with other conventional RDF datastores which either timeout or fail at some point.

In addition, SPARQLGX outperforms several implementations in many cases (also as shown on Table 2), while implementing a simple architecture exclusively built on top of open source and publicly available technologies. Furthermore, the SDE variant of our implementation, which does not require any preprocessing phase offers performances similar to the ones obtained with state-of-the-art implementations that require preprocessing.

6 Related Work

In recent years, many RDF systems capable of evaluating SPARQL queries have been developed [12]. These stores can be divided in two categories: centralized systems (*e.g.* RDF-3X [13] or Virtuoso [5]) and distributed ones, that we further review. Distributed RDF stores can in turn be divided into three categories. (1) The *ad-hoc* systems that are specially designed for RDF data and that distribute and store data across the nodes according to custom ad-hoc methods (*e.g.* 4store [10]). (2) Other systems use a communication layer between centralized systems deployed across the cluster and then evaluate sub-queries on each node such as Partout with RDF-3X [6]. (3) Lastly, some RDF systems [8, 16–18] are built on top of distributed Cloud platforms such as Apache Hadoop. One

[4] We set the timeout delay to 10 hours for the query evaluation stage and to 24 hours for the dataset preprocessing stage.

		Conventional RDF Datastores				Direct Evaluators		
		RYA	CliqueSquare	S2RDF	SPARQLGX	SDE	RDFHive	PigSPARQL
Watdiv-100M	Preprocessing (minutes)	35	288	718	24	0	0	0
	Footprint (GB)	11.0	30.2	15.2	23.6	46.8	46.8	46.8
	QC (seconds)	TIMEOUT	333	504	**118**	278	1174	6973
	QF (seconds)	12071	FAIL	771	**182**	355	1640	9904
	QL (seconds)	5895	**94**	490	119	199	1175	5670
	QS (seconds)	1892	FAIL	805	**210**	363	1053	2460
Lubm-1k	Preprocessing (minutes)	34	157	408	55	0	0	0
	Footprint (GB)	16.2	55.8	13.1	39.1	72.0	72.0	72.0
	Q1 (seconds)	192	461	118	**22**	96	281	226
	Q2 (seconds)	TIMEOUT	**105**	1599	320	8917	TIMEOUT	1239
	Q14 (seconds)	66	22	86	**9**	149	274	212
Lubm-10k	Preprocessing (minutes)	410	TIMEOUT	FAIL	672	0	0	0
	Footprint (GB)	177	403	N/A	407	747	747	747
	Q1 (seconds)	1799	524	N/A	**305**	904	1631	2272
	Q2 (seconds)	TIMEOUT	22093	N/A	**19158**	TIMEOUT	TIMEOUT	18029
	Q14 (seconds)	571	731	N/A	**541**	1500	2937	2525

Table 2: Compared System Performance.

major interest of such platforms relies on their common file systems (*e.g.*, HDFS): indeed various applications can access data at the same time and the distribution/replication issues are transparent. These systems [8, 16–18], then evaluate SPARQL conjunctive queries using various tools as presented in Section 5 (*e.g.* Accumulo, Hive, Spark, etc.). To set up appropriate tools for pipeline applications, we choose to distribute data with a Cloud platform (HDFS) and evaluate queries using Spark. We compared the performances of SPARQLGX with the most closely related implementations in Section 5.

Finally, it is worthwhile to notice that SPARQL is a very expressive language which offers a rich set of features and operators. Most evaluators based on Cloud platforms focus on the restricted SPARQL fragment composed of conjunctive queries. SPARQLGX also natively supports a slight extension of this fragment with UNION and OPTIONAL operators at top level.

7 Conclusion

We proposed SPARQLGX: a tool for the efficient evaluation of SPARQL queries on distributed RDF datasets. SPARQL queries are translated into Spark executable code, that attempts to leverage the advantages of the Spark platform in the specific setting of RDF data. SPARQLGX also comes with a direct evaluator based on the same SPARQL translation process and called SDE, for situations where preprocessing time matters as much as query evaluation time. We report on practical experiments with our systems that outperform several state-of-the-art Hadoop-reliant systems, while implementing a simple architecture that is easily deployable across a cluster.

References

1. SPARQL 1.1 overview (March 2013), http://www.w3.org/TR/sparql11-overview/
2. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: Proceedings of the 33rd international conference on Very large data bases. pp. 411–422. VLDB Endowment (2007)
3. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: ISWC. pp. 197–212. Springer (2014)
4. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., et al.: Spark SQL: Relational data processing in spark. In: SIGMOD. pp. 1383–1394. ACM (2015)
5. Erling, O., Mikhailov, I.: Virtuoso: RDF support in a native RDBMS. In: Semantic Web Information Management, pp. 501–519. Springer (2010)
6. Galarraga, L., Hose, K., Schenkel, R.: Partout: A distributed engine for efficient rdf processing. In: WWW Companion. pp. 267–268 (2014)
7. Gallego, M.A., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An empirical study of real-world SPARQL queries. In: 1st International Workshop on Usage Analysis and the Web of Data at the 20th International World Wide Web Conference (2011)
8. Goasdoué, F., Kaoudi, Z., Manolescu, I., Quiané-Ruiz, J.A., Zampetakis, S.: Cliquesquare: Flat plans for massively parallel RDF queries. In: ICDE. pp. 771–782. IEEE (2015)
9. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web 3(2), 158–182 (2005)
10. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered RDF store. SSWS (2009)
11. Hayes, P., McBride, B.: RDF semantics. W3C Rec. (2004)
12. Kaoudi, Z., Manolescu, I.: RDF in the clouds: A survey. The VLDB Journal 24(1), 67–91 (2015)
13. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment 1(1), 647–659 (2008)
14. Odersky, M.: The scala language specification v 2.9 (2014)
15. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD. pp. 1099–1110. ACM (2008)
16. Punnoose, R., Crainiceanu, A., Rapp, D.: Rya: a scalable RDF triple store for the clouds. In: International Workshop on Cloud Intelligence. p. 4. ACM (2012)
17. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: Mapping SPARQL to pig latin. In: Proceedings of the International Workshop on Semantic Web Information Management. p. 4. ACM (2011)
18. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on spark. VLDB pp. 804–815 (2016)
19. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Mass Storage Systems and Technologies (MSST). pp. 1–10. IEEE (2010)
20. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment 2(2), 1626–1629 (2009)
21. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: NSDI. pp. 2–2. USENIX Association (2012)