HAQWA: a Hash-based and Query Workload Aware Distributed RDF Store

Olivier Curé, Hubert Naacke, Mohamed-Amine Baazizi, Bernd Amann

Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris,
CNRS, UMR 7606, LIP6, F-75005, Paris, France
{firstName.lastname}@lip6.fr

Abstract. Like most data models encountered in the Big Data ecosystem, RDF stores are managing large data sets by partitioning triples across a cluster of machines. Nevertheless, the graphical nature of RDF data as well as its associated SPARQL query execution model makes the efficient data distribution more involved than in other data models, e.g., relational. In this paper, we propose a novel system that is characterized by a trade-off between complexity of data partitioning and efficiency of query answering in cases where a query workload is known. The prototype is implemented over the Apache Spark framework, ensuring high availability, fault tolerance and scalability. This short paper presents the main features of the system and highlights the omnipresence of parallel computation across data fragmentation and allocation, encoding and query processing tasks.

1 Introduction

The volume of produced RDF data keeps increasing, partly due to the popularity of projects such as Linked Open Data (LOD) and Schema.org. In order to fulfill the vision of the Web of data as well as of the Semantic Web, it is necessary to manage such data in an efficient manner. With RDF graphs composed of hundreds of millions to several billions of triples, a distributed approach is generally necessary to address issues such as fault tolerance, high availability and scalability. This analysis applies to all popular data models, e.g., relational, but the case of RDF is more involved due to its graphical nature and its navigational query characterization as well as its inherent need for reasoning services.

The graphical aspect is related to the manner in which one partitions a graph over a cluster. Usually one expects to obtain a properly balanced partitioning, i.e., the graph triples are evenly distributed over the cluster nodes. But for high availability, the system may also impose some data replication. This may impact the query processing performance which is fundamentally navigational in SPARQL. Intuitively, the system binds variables present in a query by matching its pattern to a given RDF graph. These variables generally span several triple patterns and thus provide a form of join. The number of joins generally exceeds what one is used to encounter in queries of other data models, e.g., SQL in the relational data model. Even, for a single machine RDF store [5], this aspect implies specialized query optimization techniques to ensure the identification of an efficient query plan. Several recent research works [4], [6] and [2] claim that these graphical and navigational aspects are not be considered independently. By considering them together, one wants to address load-balancing and replication on one side and to minimize exchanges across the network on the other side. For instance, the graph satisfying a SPARQL query may be partitioned across several machines thus requiring to transfer temporary results from one node to another to perform a join. It is frequent in real-world cases that the cost of network communication exceeds the cost of local query processing.

The inference aspect is related to the presence of an ontology, e.g., RDFS or OWL. It directly impacts data preparation and query processing. The two main solutions are: (i) materialization where all possible entailments are persisted in the store at data loading-time and hence allowing for a standard SPARQL query processor or (ii) query-reformulation where queries are rewritten at query run-time using a reasoner but where no extra data preparation is required.

In the HAQWA (Hash-based and Query Workload Aware) system, we propose a trade-off between data distribution complexity and query answering efficiency together with an encoding approach supporting RDFS entailments with a minimum of materialization and query reformulation. The performances of the system are partly due to the usage of the Apache Spark framework and to the best of our knowledge, our solution is the first implementation of an RDF store over this parallel computing system. In the following sections, we present three main components, namely data fragmentation/allocation, data encoding and query processing and highlight the impact of the Spark implementation.

2 System components

2.1 Fragmentation and Allocation

Our fragmentation and allocation strategies are complementary and aim to produce a compromise between query processing efficiency and data preparation duration. In a first step, a hash-based partitioning is performed using triples' subjects as keys. This fragmentation ensures that star-shaped queries are performed locally but they provide no guarantees on other query forms, e.g., property chains, tree, cycle or hybrid. This approach is much faster than other solutions, e.g., based on a graph partitioner such as Metis [3], [4] or some machine learning techniques [6], e.g., k-means.

The allocation approach is based on the analysis of frequent queries executed over the dataset and is inspired from the WARP algorithm [3]. Intuitively, the system decomposes the query triple pattern into a set of local sub-queries which are all evaluated locally. Each of those sub queries is a candidate to be the starting point (seed query) to evaluate the entire query pattern. To prevent any network communication across nodes, the missing triples are replicated into the partitions that contains the triples of the seed. To do so, for each candidate and partition, HAQWA computes the cost of transferring missing triples into the current partition.

These component's steps have been implemented as Spark programs using the Scala programming language, thus ensuring parallel executions of the different tasks, e.g., hash-based partitioning and query-aware distribution/replication.

2.2 Data set encoding

Once the data are allocated to cluster nodes, the system performs an encoding steps which has two benefits. First, it considerably reduces the volume of the data set by storing subjects, properties and objects as integer values and not as strings of characters. Moreover, operations over the integer-based triple representation is also much more efficient that the one on strings.

In the presence of an ontology, our encoding approach distinguishes between the ontology and the set of instances. First, a smart binary encoding assigns identifiers to concepts and properties of the ontology such that these identifiers represent the corresponding hierarchies. These identifiers are used during query processing to prevent query reformulation while ensuring answer completeness for the RDFS entailment regime. The encoding system also addresses *domain* and *range* RDFS properties through the creation of additional data structures. The facts present in the RDF dataset are encoded in a traditional way by producing two dictionaries (id to String and String to id) which enable to translate the triple patterns of a query and the computed result set efficiently.

The ontology encoding is performed on the Spark master node and thus runs on a single machine (the only non parallel task of our system). It uses the HermiT reasoner to compute concept classification of OWL ontologies. The instance dictionaries are computed in parallel using a chain of *map*, *reduce* and *zip* Spark operations reminiscent of functional programming. Compared to other approaches, e.g., based on MapReduce, our approach is much more efficient due to the intensive use of main-memory that Spark is doing, as opposed to disk accesses with MapReduce.

2.3 Query processing

Our fragmentation and allocation strategy ensures that queries satisfying (up to some generalization/specialization) any of the queries of the workload as well as star-shaped queries are performed locally. Other queries may require some network communication but that is a cost we are willing to pay for the efficiency of the most frequent queries as well as the minimization of the memory-footprint.

A SPARQL query sent to the HAQWA system is translated into a Spark program. The generation of such a program is simplified (compared to what one would have to implement over the Hadoop framework) due to the richness of the Spark API. Example 1 emphasizes the use of the *map*, *join* and $filter$ methods to translate the graph patterns of a simple query.

Example 1: In this example, we only concentrate on the triple pattern of the query since displaying the result set is straight forward.

```
SELECT ?x, ?y WHERE {?x rdf:type lubm:GraduateStudent.
?x lubm:name ?y. ?x lubm:advisor ?z.}
```

The elements of the graph patterns are translated using the dictionaries, e.g., the identifiers for $rdf : type$, $lubm : GraduateStudent$, $lubm : name$ are respectively 0 (from the ontology property dictionary), 956301312 (from the ontology concept dictionary) and 671088640 (from the instances dictionary). This permits to generate the following Scala command line where `triples` is a Spark Resilient Distributed Dataset (RDD), a data distributed data container.

```
tr.filter(case(s,p,o)=> p==0 && o==956301312).
  map(case(s,p,o)=>(s,(p,o))).
  join(tr.filter(case(s,p,o)=> p==671088640).
  map(case(s,p,o)=>(s,(p,o)))).map(case(s,(l1,l2))=>(s,null)).
  join(tr.filter(case(s,p,o)=> p==1233125376).
  map(case(s,p,o)=>(s,(p,o)))).map{case(s,(a,(p,o)))=>(o,s))}
```

3 Conclusion and future works

In this paper, we have presented a first implementation of an RDF store over Apache Spark, an evolution of the Hadoop framework. The first experiments [1] highlight encouraging performances which are mainly due to selection of particular strategies for the fragmentation and the allocation of the triples as well as to intensive usage of the cluster's main memory. Our solution also integrates an original encoding approach that supports the RDFS entailment regime without the usual materialization and query reformulation heavy machinery. In future works, we aim to implement a dedicated query optimizer for this Spark-based implementation and to extend the supported entailment regime to a more expressive ontology language, e.g. RDFS++.

References

1. O. Curé, H. Naacke, M. A. Baazizi, and B. Amann. On the evaluation of RDF distribution algorithms implemented over apache spark. *CoRR*, abs/1507.02321, 2015.
2. M. Hammoud, D. A. Rabbou, R. Nouri, S. Beheshti, and S. Sakr. DREAM: distributed RDF engine with adaptive query planner and minimal communication. *PVLDB*, 8.
3. K. Hose and R. Schenkel. WARP: workload-aware replication and partitioning for RDF. In *Workshops Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1–6, 2013.
4. J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.
5. T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.
6. B. Wu, Y. Zhou, P. Yuan, H. Jin, and L. Liu. Semstore: A semantic-preserving distributed rdf triple store. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 509–518, New York, NY, USA, 2014. ACM.