On the Evaluation of RDF Distribution Algorithms Implemented over Apache Spark

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. Querying very large RDF data sets in an efficient and scalable manner requires parallel query plans combined with appropriate data distribution strategies. Several innovative solutions have recently been proposed for optimizing data distribution with or without predefined query workloads. This paper presents an in-depth analysis and experimental comparison of five representative RDF data distribution approaches. For achieving fair experimental results, we are using Apache Spark as a common parallel computing framework by rewriting the concerned algorithms using the Spark API. Spark provides guarantees in terms of fault tolerance, high availability and scalability which are essential in such systems. Our different implementations aim to highlight the fundamental implementation-independent characteristics of each approach in terms of data preparation, load balancing, data replication and to some extent to query answering cost and performance. The presented measures are obtained by testing each system on one synthetic and one real-world data set over query workloads with differing characteristics and different partitioning constraints.

1 Introduction

During the last few years, an important number of papers have been published on data distribution issues in RDF database systems, [14], [8], [24], [10] and [11] to name a few. Repositories of hundreds of millions to billions of RDF triples are now more and more frequent and the main motivation of this research movement is the efficient management of ever growing size of produced RDF data sets. RDF being one of the prominent data models of the Big data ecosystem, RDF repositories have to cope with issues such as scalability, high availability, fault tolerance. Other systems addressing these issues like NoSQL systems [21], generally adopt a scale-out approach consisting of distributing both data storage and processing over a cluster of commodity hardware.

Depending on the data model, it is well-known that an optimal distribution in terms of data replication rate (characterizing the number of copies of a given information across the cluster), load balancing and/or query answering performance is hard to achieve. RDF data encode large graphs and obtaining a balanced partitioning into smaller components with specific properties is known to be an NP-hard problem in general. Hence, most distributed RDF systems are proposing heuristic-based approaches for producing optimal data distributions with respect to specific query processing environments and query workloads. In a distributed RDF data processing context, the total cost of

a distributed query evaluation process is often dominated by the data exchange cost produced by a large number of triple pattern joins corresponding to complex SPARQL graph pattern. Therefore, one of the supreme goals of all distributed RDF query processing solutions is to limit the amount of data exchanged over the cluster network through optimal data partitioning and replication strategies. Each such strategy also comes with a set of data transformation, storage and indexing steps that are more or less cost intensive.

The first systems considering distributed storage and query answering for RDF data appeared quite early in the history of RDF. Systems like Edutella [18] and RDFPeers [2] were already tackling partitioning issues in the early 2000s. More recent systems like YARS2 [12], HadoopRDF [15] and Virtuoso [5] are based on hash partitioning schemes for distributing RDF triple indexes on different cluster nodes. In 2011, [14], henceforth denoted *nHopDB*, presented the first attempt to apply graph partitioning on RDF data sets for distributed SPARQL query evaluation. In the following, the database research community has proposed a large number of RDF triple data partitioning and replication strategies for different distributed data and query processing environments. Recent systems are either extending the graph partitioning approach [13] or are complaining about their limitations [17].

As a consequence of the plethora of distribution strategies, it is not always easy to identify the most efficient solution for a given context. The first objective of this paper is to clarify this situation by conducting evaluations of prominent RDF triple distribution algorithms. A second goal is to consider Apache Spark as the parallel computing framework for hosting and comparing these implementations. This is particularly relevant in a context where a large portion of existing RDF distributed databases, $e.g.$ nHopDB [14], Semstore [24], SHAPE [17], SHARD [20], have been implemented using Apache Hadoop, an open source MapReduce [4] reference implementation. These implementations suffer from certain [22] limitations of MapReduce for processing large data sets, some of them being related to the high rate of disk reads and writes. We have chosen Spark since it is up to 100 times more efficient than Hadoop through Resilient Distributed Datasets (RDD) implementing a new distributed and fault tolerant memory abstraction.

Our experimentation is conducted over a reimplementation of five data distribution approaches where two of them are hash-based, two of them are based on graph partitioning with and without query workload awareness and one is hybrid combing hash-based partitioning and query workload awareness. Each system is evaluated over a synthetic and a real-world data-set with varying cluster settings and on a total of six queries which differ in terms of their shape, $e.g.$, star and property chains, and selectivity. We present and analyze experimentations conducted in terms of data preparation cost, distribution load balancing, data replication rate and query answering performance.

2 Background knowledge

2.1 RDF - SPARQL

RDF is a schema-free data model that permits to describe data on the Web. It is usually considered as the cornerstone of the Semantic Web and the Web of Data. Assuming

disjoint infinite sets U (RDF URI references), B (blank nodes) and L (literals), a triple (s,p,o) $\in$ (U $\cup$ B) x U x (U $\cup$ B $\cup$ L) is called an ***RDF triple*** with s, p and o respectively being the subject, predicate and object. Since subjects and objects can be shared among triples, a set of RDF triples generates an *RDF graph*.

SPARQL [1] is the standard query language for RDF graphs (triple collections) based on *graph patterns* for extracting information from RDF graphs. Let V be an infinite set of variables disjoint with U, B and L. Then, a triple $tp \in$ (U $\cup$ V) x (U $\cup$ V) x (U $\cup$ V $\cup$ L) followed by a dot '.' is a SPARQL triple pattern. The semantics of a triple pattern follows the standard *matching semantics* which consists in finding all mappings $\mu : V \rightarrow U \cup B \cup L$ such that $\mu(tp)$ is a triple in the input graphs. Graph patterns are defined recursively. A possibly empty set of triple patterns is a basic graph pattern. The semantics of a basic graph pattern gp is defined by the conjunctive extension of the triple matching semantics ($\mu(gp)$ is a connected or disconnected subgraph of the input graphs). If gp_1 and gp_2 are graph patterns, then $\{gp_1\}$ is a group pattern, gp_1 `OPTIONAL` $\{gp_2\}$ is an optional pattern, $\{gp_1\}$ `UNION` $\{gp_2\}$ is a pattern alternative. Finally, a graph pattern *gp* can contain any a constraint `FILTER` C where C is a built-in condition to restrict the solutions of a graph pattern match according to the expression C.

The complete SPARQL syntax follows the SELECT-FROM-WHERE syntax of SQL queries. The `SELECT` clause specifies the variables appearing in the query result set, the optional `FROM` clause specifies the input graphs (an input graph can be defined by default), the `WHERE` clause defines a graph pattern which is matched against the input RDF graphs.

2.2 Apache Spark

Apache Spark [26] is a cluster computing framework whose design and implementation started at UC Berkeley's AMPlab. Just like Apache Hadoop, Spark enables parallel computations on unreliable machines and automatically handles locality-aware scheduling, fault tolerance and load balancing tasks. While both systems are based on a data flow computation model, Spark is more efficient than Hadoop's MapReduce for applications requiring the reuse working data sets across multiple parallel operations. This efficiency is due to Spark's Resilient Distributed Dataset (RDD) [25], a distributed, lineage supported fault tolerant memory abstraction that enables in-memory computations more efficiently than Hadoop (which is mainly disk-based). The Spark API also simplifies data-centric programming by integrating set-oriented functions like $join$ and $filter$ which are not natively supported in Hadoop.

2.3 METIS graph partitioner

Finding a graph partitioning which is optimal with respect to certain constraints is an NP-hard problem which is practically solved by approximative algorithms like[7]. These algorithms are generally still not efficient for very large graphs hence motivating a multi-level propagation approach where the graph is coarsened until its size permits

[1] http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

to use one of the approximate solutions. The METIS system [16] follows this approach and is known to reach its limits for graphs of about half a billion triples. METIS takes as input an unlabeled, undirected graph and an integer value corresponding to the desired number of partitions. Its output provides a partition number for each node of the graph. As explained in the following section, nHopDB [14] and WARP [13] are two recent systems that are using METIS to partition RDF graphs.

3 RDF data partitioning methods

In this section, we present the main features and design principles of the RDF data partitioning methods we have chosen to compare with respect to their data preparation cost, storage load balancing, data replication and query processing costs. It is obvious that the performance results, and in particular the results concerning query processing performance, have to be considered with caution. Our goal is not to rank the different methods, but to analyze some general properties (including implementation effort) in the context of Apache Spark, which is a common modern scalable distributed data processing environment. More details about these implementations are described in Section 6.

As a starting point, we consider four different data partitioning approaches which can be characterized as hash and graph partitioning based. Each category is divided into two approaches which have been used in various systems and described in conference publications. Our fifth system corresponds to a new hybrid approach that mixes a hash-based approach with a replication strategy that enables to efficiently process long chain queries. Note that we do not consider range-based partitioning approaches since they are rarely used in existing systems due to their inefficiency.

3.1 Hash-based RDF data partitioning

The two approaches defined in this section correspond to families of RDF database systems rather than to specific systems (as in the next section). The basic of hash-based Data partitioning consists in applying to each RDF triple a hash function which returns for some triple-specific key value the node where the triple should be stored. One advantage of hash-based approaches is that they do not require any additional structure to locate the partition of a given triple except the hash function and the key value. Data replication can be achieved by defining several hash functions.

Random hashing: In a distributed random hash-based solution, the partitioning key does not correspond to a particular element of the data model like the graph, subject, property or object of a triple. For instance, the key can correspond to an internal triple identifier or to some other value obtained from the entire triple. The former solution is the one adopted by the Trinity.RDF system [27]. Some other forms of random partitioning exist and may require an additional structure for directed lookups to cluster nodes where triples are located, $e.g.$ round-robin approach. We do not consider such random partitioning approaches in our evaluation since they do not provide particularly useful data placement properties for any of the query shapes (star, property chains, tree, cycle or hybrid) used in our experiments (see Appendix A).

RDF triple element hashing: This approach has been adopted by systems like YARS2, Virtuoso, Jena ClusteredTDB and SHARD. In these systems, the hashing key provided to the hash function is composed of one or several RDF quadruple elements (graph, subject, property, object). Partitioning by subject provides the nice property of ensuring that star-shaped queries, $i.e.$ queries composed of a graph where one node has an out-degree greater than 1 and all other nodes are leaves, are performed locally on a given machine. Nevertheless they do not provide guarantees for queries composed of property chains or more complex query patterns. We will study the performance of a subject-hash based partitioning in Section 6.

3.2 Graph-based partitioning approaches

Hash-based data partitioning methods are likely to require a high data exchange rate over the network for more complex query patterns composed of longer property chains. One way to address this issue is to introduce data replication and/or to use more structured hashing functions adapted for a given query workload. Of course, these extensions come with at an additional processing cost which needs to be considered with attention. Systems corresponding to each of these approaches are considered next.

nHopDB: The data partitioning approach presented in [14] is composed of two steps. In a first stage, the RDF data set is transformed such that it can be sent to the METIS graph partitioner. This transformation removes properties and adds inverted subject-object edges to obtain an undirected graph. Then, the partitions obtained by METIS are translated into triple allocations over the cluster (all triples of the same partition are located on the same node). The partition state obtained at the end of this first stage is denoted as 1-hop. The second stage starts and corresponds to an overlap strategy which is performed using a so-called n-hop guarantee. Intuitively, for each partition, each leaf l is extended with triples whose subject correspond to l. This second replication stage can be performed several times on the successively generated partitions. Each execution increases the n-hop guarantee by a single unit.

[14] describes an architecture composed of a data partitioner and a set of local query engine workers implemented by RDF-3X [19] database instances. Some queries can be executed locally on a single node and thus enjoy all the optimization machinery of RDF-3X. For queries where the answer set spans multiple partitions, the Hadoop MapReduce system is used to supervise query processing.

WARP: The WARP system [13] has been influenced by nHopDB and the Partout system [8] (the two authors of WARP also worked on Partout). WARP borrows from nHopDB its graph partitioning approach and 2-hop guarantee. Like Partout, it then refines triple allocations by considering a given query workload of the most frequently performed queries over the given data set. The system considers that this query workload is provided in one way or another. More exactly, each of these queries is transformed into a set of query patterns (defining a class of equivalent queries) and WARP guarantees that frequent queries can be distributed over the cluster nodes and processed locally without exchanging data across machines (the final result is defined by the union

of locally obtained results). WARP proceeds as follows for partitioning and replicating a RDF triple collection:

1. A first step partitions the transformed unlabeled and undirected RDF graph using the METIS graph partitioner as described in nHopDB.
2. The RDF triples are fragmented according to the partitions of their subjects and loaded into the corresponding RDF-3X [19] database instance.
3. A replication strategy is applied to ensure a 2-hop guarantee.
4. Finally, WARP chooses for each query pattern qp a partition which will receive all triples necessary for evaluating pattern qp locally. For this, WARP decomposes each query pattern obtained from the query workload into a set of sub-queries which are potential starting points or *seed queries* for the evaluation of the entire query pattern. Then, WARP estimates for each seed query and partition the cost of transferring missing triples into the current partition and selects the seed query candidate that minimizes this cost. An example is presented in Section 4.3.

The WARP system implements its own distributed join operator to combine the local sub-queries. Locally, the queries are executed using RDF-3X. As our experiments confirm, most of the data preparation effort for WARP is spent in the graph partitioning stage.

3.3 Hybrid partitioning approach

The design of this original hybrid approach has been motivated by our analysis of the WARP system as well as some hash-based solutions. We have already highlighted (as confirmed in the next section) that the hash-based solutions require short data preparation times but come with poor query answering performance for complex query patterns. On the other hand, the WARP system proposes an interesting analysis of query workloads which is translated into an efficient data distribution. We will present in our experiments a hybrid solution which combines RDF triple element hashing using subjects as hash keys with query workload aware triple replication is described in the last step of WARP.

4 Spark system implementations

4.1 Data set loading and encoding

All data sets are first loaded on the cluster's Hadoop File System(HDFS). The loading rate in our cluster averages 520.000 triples per second which allows us to load large data sets like LUBM 2K or Wikidata in less than 10 minutes.

Like in most RDF stores, each data set is encoded by providing a distinct integer value to each node and edge of the graph (see Chapter 4 in [3] for a presentation of RDF triple encoding methods). The encoding is performed in parallel in one step using the Spark framework.[2] The encoded data sets, together with their dictionaries (one for the properties and another for subjects and objects) are also loaded into HDFS.

[2] More implementation details can be found at http://www-bd.lip6.fr/wiki/doku.php?id=site:recherche:logiciels:rdfdist.

4.2 Hash-based data distribution

The implementation of hash-based partitioning approaches in Spark is relatively straightforward since the Spark API directly provides hash-based data distribution functionalities. We achieve random-hash partitioning by using the whole RDF triple as hash key. Triple element hashing is obtained by using the triple subject URI. In our experiments, we do not provide replication by adding other hashing function. The query answering evaluation is performed forthrightly following a translation from SPARQL to Spark scripts requiring a mix of `map`, `filter`, `join` and `distinct` methods performed over RDDs.

4.3 Graph partitioning-based data distribution

The two approaches in this partitioning category, nHopDB and WARP, require three METIS related steps for the preparation, computation and transformation of the results. Since METIS only can deal with unlabeled and undirected graphs, we start by removing predicates from the data sets and appending the reversed subject/object edges to the graph. Using METIS also imposes limitations in terms of accepted graph size. Indeed, the largest graph that can be processed contains about half a billion nodes. Consequently, we limit our experimentations to data sets of at most 250 million RDF triples provided that their undirected transformation yields graphs of 500 million nodes. The output of METIS is a set of mapping assertions between nodes and partitions. Based on these mappings, we allocate a triple to the partition of its subject. In terms of data encoding, we extend triples with partition identifiers yielding quads. Note that at this stage, the partition identifier can be considered as 'logical' and not 'physical' since the data is not yet stored on a given cluster node. We would like to stress that the preparation and transformation phases described above are performed in parallel using Spark programs.

nHopDB: In the Spark implementation of nHopDB, the n-hop guarantee is computed over the RDD corresponding to the generated quads. This Spark program can be executed (n-1) times to obtain an n-hop guarantee.

WARP: Our implementation of WARP analyzes the query workload generalization using Spark built-in operators. For instance, consider the following graph pattern of a query denoted Q1:

```
?x advisor ?y . ?y worksFor ?z . ?z subOrganisation ?t
```

For processing this pattern, the system uses the `filter` operator to select all triples that match the `advisor`, `worksFor` and `subOrganization` properties. Then, the `join` operator is used to perform equality join predicates on variables `y` and `z`. The query result is a set of variable bindings. We extend the notion of variable bindings with the information regarding the partition identifier of each triple. For instance, an extract of a Q1's result (in an decoded readable form) is represented as {`(Bob,Alice,1), (Alice, DBteam,3),(DBteam, Univ1,1)`}. The result for pattern `?y worksFor ?z` contains the triple binding {`(Alice, DBteam,`

3)} which means that "Alice" and "DBTeam" are bound to variables ?x and ?y and the triple is located on partition 3. The two other triples for triple patterns ?x advisor ?y and ?z subOrganisation ?t are located on partition 1. It is easy to see that by choosing the seed query ?x advisor ?y or ?z subOrganisation ?t, we need to copy only triple (Alice, worksFor, DBteam) in partition 3 to partition 1 whereas by choosing pattern ?y worksFor ?z two triples have to be copied to partition 1. As specified earlier in Section 3.2, we consider all the candidate seeds to choose the seed that implies the minimal number of triples to replicate.

Finally, for querying purposes, each query is extended with a predicate enforcing local evaluation by joining triples with the same partition identifier.

4.4 Hybrid approach

This approach is mixing the subject-based hashing method with the WARP workload-aware processing. Hence, using our standard representations of triples and quads together with Spark's data transformation facilities made our coding effort for this experiment relatively low.

5 Experimental setting

5.1 Data sets and queries

In this evaluation, we are using one synthetic and one real world data set. The synthetic data set corresponds to the well-established LUBM [9]. We are using three instances of LUBM, denoted LUBM1K, LUBM2K and LUBM10K which are parameterized respectively with 1000, 2000 and 10000 universities. The real world data set consists in Wikidata [23], a free collaborative knowledge base which will replace Freebase [1] in 2015. Table 2 presents the number of triples as well as the size of each of these data sets.

Data set	#triples	nt File Size
LUBM 1K	133 M	22 GB
LUBM 2K	267 M	43 GB
LUBM 10K	1,334 M	213 GB
Wikidata	233 M	37 GB

Table 1. Data set statistics of our running examples

Concerning queries, we have selected three SPARQL queries from LUBM (namely queries #2, #9 and #12 respectively denoted Q2, Q3 and Q4) extended by a new query, denoted Q1, which requires a 3-hop guarantee to be performed locally on the nHopDB, WARP and hybrid implementations. To complement the query evaluation, we have created two queries for the Wikidata experiments, resp. Q5 and Q6. The first one takes

the form of a 3-hop property chain query that shows to be much more selective than the LUBM ones, the second one is shaped as a simple star and was motivated by the absence of such a form in our query set. All six queries are presented in Appendix A.

5.2 Computational environment

Our evaluation was deployed on a cluster consisting of 21 DELL PowerEdge R410 running a Debian distribution with a 3.16.0-4-amd64 kernel version. Each machine has 64GB of DDR3 RAM, two Intel Xeon E5645 processors each of which is equipped with 6 cores running at 2.40GHz and allowing to run two threads in parallel (hyperthreading). Hence, the number of virtual cores amounts to 24 but we used only 15 cores per machine. In terms of storage, each machine is equipped with a 900GB 7200rpm SATA disk. The machines are connected via a 1GB/s Ethernet Network adapter. We used Spark version 1.2.1 and implemented all experiments in Scala, using version 2.11.6. The Spark setting requires that the total number of cores of the cluster to be specified. Since in our experiments we considered clusters of 5, 10 and 20 machines respectively, we had to set the number of cores to 75, 150 and 300 cores respectively.

6 Experimentation

Since we could not get any query workloads for Wikidata, it was not possible to conduct any experimentation with WARP and the hybrid approach over this data sets. Moreover, since METIS is limited to data sets of half a million edges, it was not possible to handle nHopDB and WARP over LUBM10K. Given the fact that the hybrid system relies on subject hashing, and not METIS, it was possible to conduct this experimentation over LUBM10K for that system.

6.1 Data preparation costs

Figure 1 presents the data preparation processing times for the different systems. As one would expect, the hash-based approaches are between 6 and 30 times faster (depending on the number of partitions) than the graph partition-based approaches. This is mainly due to the fact that METIS runs on a single machine (we have not tested parMETIS, a parallelized version of METIS) while the hash operations are being performed in parallel on the Spark cluster. The evaluation also emphasizes that the hybrid approach presents an interesting compromise between these distribution method families. By evaluating the different processing steps in each of the solutions, we also could find out that, for hash-based approaches, around 15% of processing time is spent on loading the data sets whereas the remaining 85% of time is spent on partitioning the data. For the graph partitioning approaches, 85 to 90% corresponds to the time spent by METIS for creating the partitions; the durations increase with the larger data set sizes. This explains that the time spent by graph partitioning approaches are slightly increasing even when more machines are added. This does not apply for the other solutions where more machines lead to a reduction of the preparation processing time.

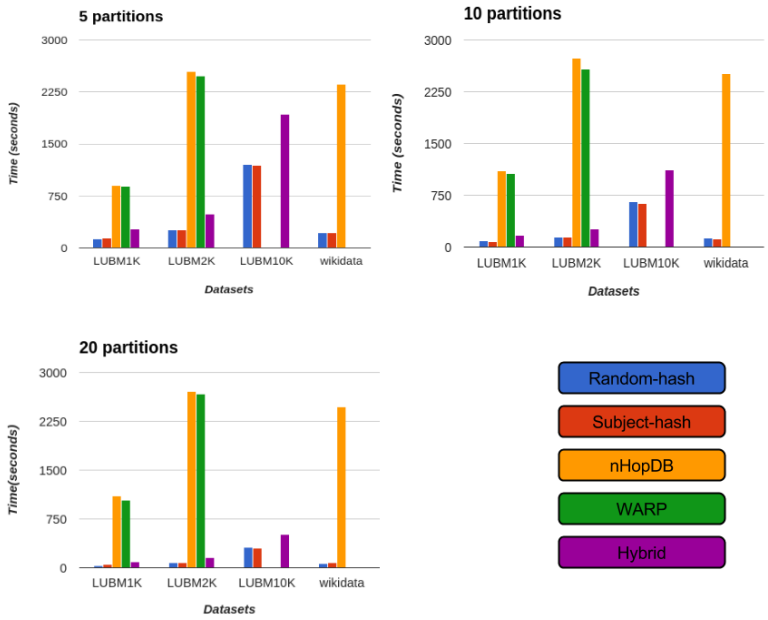

Fig. 1. Data preparation times

6.2 Storage load balancing

Load balancing is an important aspect when distributing data for storage and querying purposes. In Figure 2, we present the standard deviations over all partition sizes (in log scale) for the different implementation. For the graph partitioning-based and hybrid approaches, we only consider the standard deviation of the partition sizes at the end of the partitioning process, $i.e.$, METIS partitioning and n-hop guarantee application.

The two hash-based approaches and the hybrid approach are the best solutions and are close to each other. This is rather obvious since the hash partitioning approaches are concentrating on load balancing while a graph partitioning tries to reduce the number of edges cut during the fragmentation process. The hybrid approach is slightly less balanced due to the application of the WARP query workload-aware strategy. The random-based hashing has 5 to 12% less deviation than subject hashing. This is due to high degree nodes that may increase the size of some partitions. The nHopDB approach is the less efficient graph partitioning solution when considering load balancing. We believe that this is highly related to the structure and the number of queries one considers in the query workload. We consider that further analysis needs to be conducted on real world data sets and query workloads to confirm these nevertheless interesting conclusions.

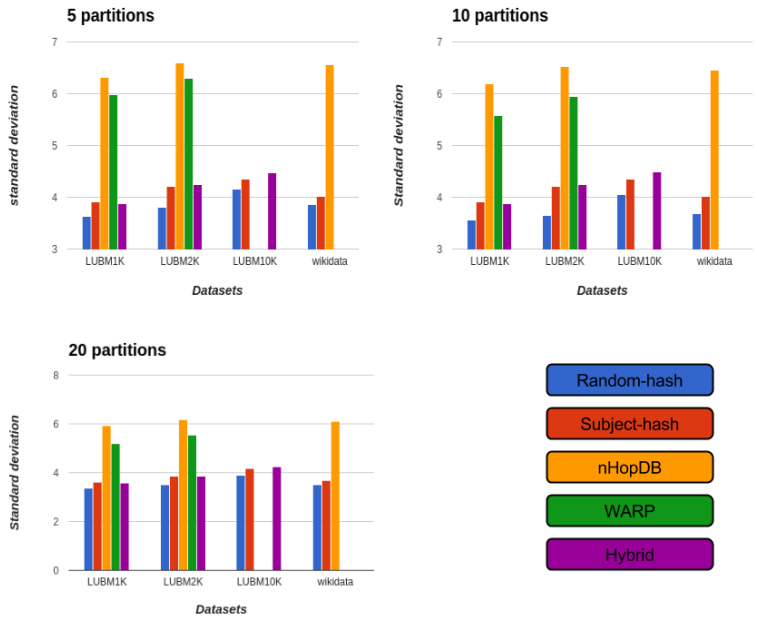

Fig. 2. Standard deviation

6.3 Data replication

Intrinsically, all solutions present some node replications since a given node can be an object in one partition and a subject in another one. This corresponds to the 1-hop guarantee that ensures validity of data, i.e., no triples are lost during the partitioning phase. In this section, we are only interested in triple replication. Only the nHopDB, WARP and hybrid solutions present such replications.

Table 2 provides the replication rates for each of these systems for the LUBM 1K and 2K data sets. Several conclusions can be drawn from this table. First, METIS-based approaches are more efficient than the subject-hashing of the hybrid system. Remember that by minimizing edge cut, a graph partitioner groups the nodes that are close to each other in the input graph. Secondly, the more partitions the cluster contains, the more overall replication one obtains. The n-hop guarantee replicates less than the query workload-aware method of WARP. Finally, we can stress that the replication of the hybrid approach can be considered quite acceptable given the data replication duration highlighted in Section 6.1.

6.4 Query processing

In order to efficiently process local queries and to fairly support performance comparison in a distributed setting, we must use the same computing resources for local and distributed runs. A parallel query runs locally when every machine only has to access its

Part. scheme	nHopDB			WARP			Hybrid		
Data set	5 part.	10 part.	20 part.	5 part.	10 part.	20 part.	5 part.	10 part.	20 part.
LUBM 1K	0.12	0.16	0.17	0.26	0.54	0.57	0.54	1.33	1.84
LUBM 2K	0.12	0.16	0.18	0.34	0.52	0.54	0.54	1.33	1.94

Table 2. Replication rate comparison for three partitioning schemes and three cluster sizes

own partition (inter-partition parallelism). To fully exploit the multi-core machines on which we perform our experiments, it would be interesting to consider not only inter-partition parallelism but intra-partition parallelism exploiting all cores as well. Unfortunately, intra-partition parallelism is not fully supported in Spark since a partition is the unit of data that one core is processing. Thus, to use 15 cores on a machine, we must split a partition into 15 sub partitions. Spark does not allow to specify that such sub-partitions must reside together on the same machine[3]. In the absence of any triple replication, the hash-based solutions are not impacted by this limitation. This is not the case for the systems using replication and where local queries might be evaluated on different partitions. For the two query workload-aware solutions (i.e., WARP and hybrid), we conducted our experiment using a workaround that forces Spark to use only one machine for processing one partition: for each local query, we run Spark with only one slave node. Then we load only the data of one partition and process the query using all the cores of the slave node. To be fair and take into account the possibility that the execution time of a local query might depend on the choice of the partition, we repeat the experiment for every partition and report the maximum response time. The case of nHopDB is more involved and requires to develop a special dedicated query processor, specialized for Spark, to fully benefit from the data fragmentation. In a nutshell, that system would have to combine intra and inter-partition query processors. The former would run for query subgraphs that can run locally and the second one would perform joins over all partitions with retrieved temporary results. Since the topic of this paper concerns the evaluation of distribution strategies, we do not detail the implementation of such a query processor in this work and hence we do not present any results for the nHopDB system.

Table 3 presents the query processing times for our data set. Due to space limitation, we only present the execution time obtained over the 20 partitions experiment. The web site companion (see [6]) highlights that the more partitions are used the more efficient is the query processing. The table clearly highlights that the WARP systems are more efficient than the hash-based solutions. Obviously, the simpler the query, $e.g.$ Q4 and Q6, run locally while the others require inter-partition communication. Spark version 1.2.1 `shuffle read` measure indicates the total information exchange (locally on a node and globally over the network) and we could not measure the inter node information communication cost.

[3] We expect that future version of Spark will allow such a control.

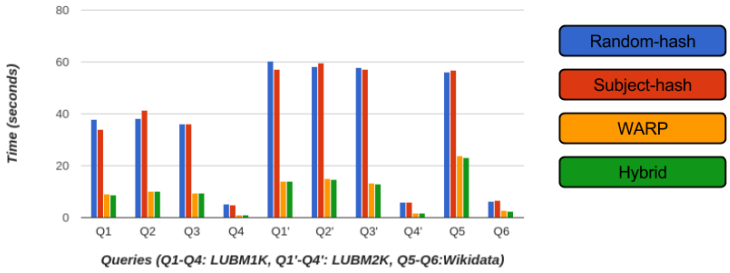

Fig. 3. Query Evaluation on 20 partitions

7 Other related work

Some other interesting works have recently been published on the distribution of RDF data. Systems such as Semstore [24] and SHAPE [17] take some original position.

Instead of using the common query workload, Semstore divides a complete RDF graph into a set of paths which cover all the original graph nodes, possibly with node overlapping between paths. These paths are denoted Rooted Sub Graph (RSG in short) since they are generated starting from nodes with a null in-degree, $i.e.$, roots, to all their possible leaves. A special workaround is used to handle cycles that may occur at the root position, $i.e.$, cycles that are not reachable from any root. The idea is then to regroup these RSG into different partitions. This is obviously a hard problem for which the authors propose an approximated solution. Their solution uses the K-means clustering approach which regroups RSG with common segments together in the same partition. A first limitation of this approach is the high dimensionality of the vectors handled by the K-means algorithm, $i.e.$, the size of any vector corresponds to the number of nodes in the graph. A second limitation is related to the lack of an efficient balancing of the number triples across the partitions. In fact, the system operates at the coarse-grained level of RSG and provides a balancing at this level only. Semstore is finally limited in terms of join patterns. It can efficiently handle S-O (subject-object) and S-S (subject-subject) join patterns but other patterns, such as O-O (object-object) may require inter node communication.

The motivation of the SHAPE system is that graph partitioning approaches do not scale. Just like in our hybrid solution, they propose to replace the graph partitioning step by a hash partitioning one. Then, just like in the nHopDB system, they replicate according to the n-hop guarantee. Hence, they do not consider any query workload and take the risk of inter-partition communication for long chain queries longer than their n-hop guarantee.

8 Conclusions and perspectives

This paper presents an evaluation of different RDF graph distribution methods which are ranging over two important partitioning categories: hashing and graph partitioning.

We have implemented five different approaches over the Apache Spark framework. Due to its efficient main memory usage, Spark is considered to provide better performances than Hadoop's MapReduce. While several RDF stores have been designed on top of Hadoop, we are not aware of any systems running on top of Spark. The main motivation of the experiments is that existing partitioning solutions do not scale gracefully to several billion triples. For instance, the METIS partitioner is limited to less than half a billion triples and SemStore (cf. related works section) relies on K-Means clustering of vectors whose dimension amount to the number of nodes of the data to be processed ($i.e.$, 32 millions in the case of LUBM1K). Computing a distance at such high dimension is currently not possible within Spark, even when using sparse vectors. Moreover, applying a dimension reduction algorithm to all the vectors is not tractable.

The conclusion of our experiment is that basic hash-based partitioning solutions are viable for scalable RDF management: they come at no preparation cost, $i.e.$ only require to load the triples into the right machine, and are fully supported by the underlying Spark system. As emphasized by our experimentation, Spark scales out to several billion triples by simply adding extra machines. Nevertheless, without any replication, these systems may hinder availability and reduce the parallelism of query processing. They also involve a lot of network communications for complex queries which require to retrieve data from many partitions. Nonetheless, by making intensive use of main memory, we believe that Spark provides a high potential for these systems. Clearly, with the measures we have obtained in this evaluation, we can stress that if one needs a fast access to large RDF data sets and is, to some extent, ready to sacrifice the performance of processing complex query patterns then the hash-based solution over Spark is a good compromise.

Concerning the nHopDB and WARP approaches, we consider that using a graph partitioning system like METIS has an important drawback due to the performance limitations. Based in these observations, we investigated the hybrid candidate solution which does not involve a heavy preparation step and retains the interesting query workload aware replication strategy. This approach may be particularly interesting for data warehouses where the most common queries (materialized views) are well identified. With this hybrid solution we may get the best of worlds, the experiments clearly emphasize that the replication overhead compared to the pure WARP approach is marginal but the gain in data preparation is quite important.

Concerning Spark, we highlighted that it can process distributed RDF queries efficiently. Moreover, the system can be used for the two main steps, data preparation and query processing, in an homogeneous way. Rewriting SPARQL queries into the Scala language (supported by Spark) is rather easy and we consider that there is still much room for optimization. The next versions of Spark which are supposed to provide more feedback on data exchange over the network should help fine-tune our experiments and design a complete production-ready system.

References

1. K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM*

SIGMOD International Conference on Management of Data, SIGMOD '08, pages 1247–1250, New York, NY, USA, 2008. ACM.
2. M. Cai and M. Frank. RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network. In *Proc. 13th International World Wide Web Conference*, New York City, NY, USA, May 2004.
3. O. Curé and G. Blin. *RDF Database Systems, 1st Edition*. Morgan Kaufmann, Nov. 2014.
4. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004.
5. O. Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.
6. http://webia.lip6.fr/∼baazizi/research/iswc2015eval/expe.html.
7. C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference, DAC '82, Las Vegas, Nevada, USA, June 14-16, 1982*, pages 175–181, 1982.
8. L. Galarraga, K. Hose, and R. Schenkel. Partout: A distributed engine for efficient RDF processing. *CoRR*, abs/1212.5636, 2012.
9. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
10. 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, USA, June 22-27, 2014*, pages 289–300, 2014.
11. 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(6):654–665, 2015.
12. A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. In *The Semantic Web, 6th International Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007.*, pages 211–224, 2007.
13. 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*, pages 1–6, 2013.
14. J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.
15. M. F. Husain, J. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Transactions on Knowledge and Data Engineering*, 23:1312–1327, 2011.
16. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.
17. K. Lee and L. Liu. Scaling queries over big RDF graphs with semantic hash partitioning. *PVLDB*, 6(14):1894–1905, 2013.
18. W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In *Proceedings of the Eleventh International World Wide Web Conference, WWW 2002, USA*, pages 604–615, 2002.
19. T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.
20. 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.

21. P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.
22. M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.
23. D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.
24. 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.
25. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, 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 Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.
26. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.
27. K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4):265–276, 2013.

A Appendix : Query workload

Q1: (property path)
```
SELECT ?x ?y ?z WHERE { ?x lubm:advisor ?y.
?y lubm:worksFor ?z. ?z lubm:subOrganisation ?t. }
```
Q2: (typed, star and property path)
```
SELECT ?x ?y ?z WHERE {
?x rdf:type lubm:GraduateStudent.
?y rdf:type lubm:University.
?z rdf:type lubm:Department. ?x lubm:memberOf ?z.
?z lubm:subOrganizationOf ?y.
?x lubm:undergraduateDegreeFrom ?y }
```
Q3: (typed, star and property path)
```
SELECT ?x ?y ?z WHERE { ?x rdf:type lubm:Student.
?y rdf:type lubm:Faculty. ?z rdf:type lubm:Course.
?x lubm:advisor ?y. ?y lubm:teacherOf ?z.
?x lubm:takesCourse ?z }
```
Q4: (typed, property path)
```
SELECT ?x ?y WHERE { ?x rdf:type lubm:Chair.
?y rdf:type lubm:Department. ?x lubm:worksFor ?y.
?y lubm:subOrganizationOf <http://www.University0.edu>
}
```
Q5: (property path)
```
SELECT ?x ?y ?z WHERE { ?x entity:P131s ?y.
?y entity:P961v ?z. ?z entity:P704s ?w }
```
Q6: (star)
```
SELECT ?x ?y ?z WHERE { ?x entity:P39v ?y.
?x entity:P580q ?z. ?x rdf:type ?w }
```