Efficient SPARQL Query Evaluation In a Database Cluster

Fang Du[1,2], Haoqiong Bian[1], Yueguo Chen[1], Xiaoyong Du[1]
[1]*School of Information and DEKE lab, Renmin University of China, Beijing, China*
[2]*School of Mathematics and Computer Science, Ningxia University, Yinchuan, China*
{*dfang,bianhq,chenyueguo,duyong*}*@ruc.edu.cn*

***Abstract*—Efficient SPARQL query evaluation is a significant challenge when the database contains billions of RDF triples, which is very common for many existing Web-scale RDF data sources. We address this challenge by 1) effectively partitioning the whole RDF dataset into small partitions according to the schemas of the RDF subjects, and 2) elaborately placing the partitions within clusters so that, on each local partition, we can make the most advantage of the state-of-the-art SPARQL query processing engine, and across the partitions, we can exploit the power of parallel databases for achieving scalable query evaluation of massive RDF data. This paper introduces the data partitioning and placement strategies, as well as the SPARQL query evaluation and optimization techniques in a cluster environment. Experiments are conducted over a synthesized dataset and a real dataset containing billions of triples. The results demonstrate that better query evaluation performance over the baseline can be achieved.**

***Keywords*-SPARQL query; RDF; data partitioning; parallel database;**

I. Introduction

With the rapid growth of large RDF datasets such as Linked Open Data [1], Freebase [2] and DBPedia [3], efficient query evaluation over the huge RDF stores becomes more and more challenging. The state-of-the-art SPARQL query evaluation technique, RDF-3X [4], is designed for efficient SPARQL query processing in a single node, taking the advantage that most RDF data are statical so that excessive indexes can be built to guarantee the performance of query evaluation. However, the performance of RDF-3X is limited by the size of the dataset because large indexes degrade the performance significantly when processing billions of triples.

Alternatively, distributed SPARQL query processing systems such as YARS2 [5], SHARD [6], [7] and [8] have been recently proposed for addressing the challenge of efficient and scalable SPARQL query evaluation for massive RDF data. In these systems, the whole RDF triples are partitioned across multiple nodes (machines), and the query is conducted in parallel to achieve efficiency. Many of them [6], [7], [8] apply the MapReduce framework to achieve scalability for query evaluation. YARS2 [5] and SHARD [6] simply apply hash partitioning to distribute triples across nodes. The authors of [8] improve the partitioning strategy by applying graph partitioning techniques so that triples close to each other are likely to be partitioned into the same node. This saves the communication cost by reducing the intermediate results that need to be shipped across nodes.

Many high performance distributed SPARQL query evaluation systems [8] treat the nodes in the cluster equally, i.e., the same query evaluation process will be in parallel conducted across all the nodes. The difference between two nodes is only in the data processed by them. As data are not logically partitioned, it will generate a huge communication cost to synchronize the intermediate results across nodes for complex queries. As a result, for a complex query that requires long execution time, the system can be easily blocked by such expensive queries. In brief, these systems achieve scalability by partitioning over the data, but not the computational logics according to the underlying partitions.

In this paper, we propose an RDF data partitioning solution that is able to logically partition the RDF triples based on the schemas they can be fixed on. With these logical partitions, we are able to assign different query evaluation tasks to different logical partitions. This allows us to maximize the filtering power of local nodes, and therefore reduce the communication cost of the overall query evaluation accordingly. As queries can be vertically partitioned, depending on the queries, not all the nodes are necessary to be involved in a query evaluation process. Many simple star-join queries that are relevant to small logical partitions can be easily addressed by only one or a very small number of nodes. This actually provides mechanisms for achieving good throughput of query evaluation.

The contribution of the paper can be summarized as follows:

- We propose an RDF data partitioning solution to efficiently and effectively partition the RDF triples according to the schemas that the subjects of RDF triples have. We further propose a data placement strategy to elaborately distribute the logical partitions according to their size. Load balancing is carefully considered during the data placement process.
- Based on the partitioning and placement strategies, we propose a very efficient query evaluation solution that is able to rewrite the queries into small tasks, and run them over different logical partitions in parallel.
- We conduct extensive experiments over a synthesized dataset and a real dataset containing billions of triples. The results demonstrate that better query evaluation

performance can be achieved by our solution.

The paper is organized as follows. Section II gives the related work study. Section III shows the solutions for schema detection and data placement. The strategies of query evaluation and optimization are given in Section IV. Experimental studies are given in Section V, followed by the conclusions in Section VI.

II. Related Work

Query processing on RDF data has received much attention recently. However, most of them are based on centralized computing environment. Some well known early studies include 3-store [9], Sesame [10], Jena [11] et al., which store RDF data in relational database and query data by using relational query processing techniques. The state-of-the-art SPARQL query engine, RDF-3X [4], builds excessive indexes of all six SPO permutations, all triples are redundantly stored in six B+ trees on single node. Such data organization and storage fashion degrades the performance significantly when processing billions of triples. Some researchers began to study on distributed SPARQL query processing [7], [8], [12], [13], [14], [15]. Systems such as YARS2 [5], SHARD [6], [7] and [8] have been recently proposed for addressing the challenge of efficient and scalable SPARQL query evaluation for massive RDF data.

Query processing on distributed computing environment includes two main tasks: data distribution and query decomposition. In [8], Huang et al. partition RDF graph based on vertex, triples close to each other are likely to be partitioned into the same node. When processing, queries are decomposed into PWOC (parallelizable without communication) sub-queries according to the predicates, and are submitted to corresponding nodes to search for results. Each node in the cloud platform is equipped with RDF-3X to improve the query evaluation efficiency. The solution of [12] partition triples by predicates, which is called vertical partitioning. In order to avoid huge partitions, the data are re-partitioned into segments based on the objects. Authors of [15] propose a solution to distribute triples over different nodes in cloud platform according to the RDF graph structure. Triples are distributed based on key-value pairs in [7]. On query decomposition, a SPARQL query is decomposed into a number of star-join sub-queries, which are further processed on different nodes.

By using MapReduce and HDFS, the above approaches can achieve good scalability for query evaluation compared to traditional centralized query processing solutions. However, MapReduce has limitations for processing SPARQL queries. Firstly, it is designed for analysis tasks whose intermediate results are typically large. Therefore, they often generate huge communication cost; Secondly, too many MapReduce iterations are often necessary when processing complex queries, which cause a significant latency. In order to avoid such defects, the solution in [8] executes queries in a distributed system as much as possible. It only submits queries to the MapReduce platform when executing the expensive join operations.

III. Schema Detection and Data Placement

In this section, we first propose a solution to logically partition the whole RDF triples into small partitions. Then, we discuss how to distribute the partitions across the nodes of a cluster. The data partitioning solution is extended from the schema detection techniques proposed in [16].

A. Schema Detection

An RDF triple is simply represented as $t = < s, p, o >$, where $s = S(t)$, $p = P(t)$ and $o = O(t)$. A subject s in an RDF triple is an URI that uniquely identifies an entity in the whole RDF dataset. As such, we simply refer an entity whose subject is s as the entity s. All RDF triples of an entity s are denoted as a set $T(s) = \{t|S(t) = s\}$. The predicate set of an entity s is denoted as $P(s) = \{P(t)|S(t) = s\}$, which contains all predicates that the entity s has. We also call $P(s)$ as the schema of the entity s. Let the domain of all predicates in an RDF dataset be P. We therefore have $P(s) \subseteq P$. For an RDF store containing a large number of various types of entities, we have $|P(s)| \ll |P|$.

Based on the definition of entity schema, we are able to partition the whole RDF triples into small partitions, with each partition having entities of the same schema. However, this will generate a large number of partitions because RDF datasets are typically formed without any schema constraints in advance. Even for entities of the same type, their schemas can be varied from each other. We therefore need to further clustering the schemas into partitions so that each partition contains entities of similar schemas. After the clustering process, we actually partition the RDF triples according the schemas of entities. Within each derived partition, it is guaranteed that the schemas of different entities are similar enough to each other. The schema detection task is therefore the clustering process of entity schemas.

We treat a partition of RDF triples as a logical partition R contains entities whose predicate sets are not necessarily the same. The schema of a partition is formulated by merging the schemas of all entities it contains, i.e., $P(R) = \{p|p \in P(s), s \in R\}$. Such a partitioning mechanism actually allows us to efficiently filter irrelevant partitions simply based on their schemas.

Definition 1 (schema-valid partition): For a star-join query q, let $P(q)$ be the set of predicates specified in q, a partition R is a *schema-valid* partition to q only if $P(q) \subseteq P(R)$.

Definition 2 (schema-valid entity): For a star-join query q whose predicate set is $P(q)$, an entity s is a *schema-valid* entity to q only if $P(q) \subseteq P(s)$.

It is guaranteed that if a partition R is not *schema-valid* to a star-join query q, there will be no results in R satisfying the query q. The reason is quite obvious because there will be a predicate of q that is not a predicate of $P(R)$, i.e., no entities in R have the predicate that is specified by q. However, a *schema-valid* partition does not mean that all its entities are *schema-valid*. Fig. 1 shows an example of a *schema-valid* partition, and a non *schema-valid* entity. The entity s is not *schema-valid* because it does not have a triple with the predicate $awarded$ specified by the query.

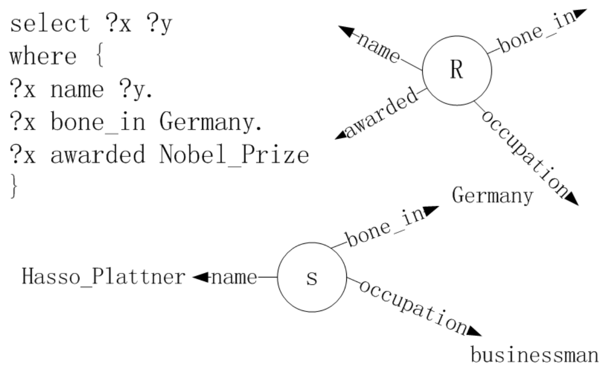


Figure 1. An example of a star-join query, a *schema-valid* partition, and a non *schema-valid* entity

B. Cost Model of Schema Clustering

The schema of a partition R is a superset of the schemas of entities it contains. As a result, the more entities a partition R has, the more predicates it probably contains. In one extreme case, if all entities in the RDF dataset are maintained by only one partition R, it will be a wide table where $P(R) = P$. The wide table R will be *schema-valid* to any query q (assuming only predicates in P are used by the query q). In the other extreme case, if only entities of the same schema share a partition R, any entity s in the partition R will have values on any predicate of $P(R)$ because $P(s) = P(R)$. However, as stated, a huge number of partitions will be generated in this case. As such, a reasonable partitioning solution of allocating entities should be in-between the above two extreme cases.

The goal of effective RDF triple partitioning is to speed up the query evaluation of a star-join query. For a complex query, it can be transformed into a number of star-join queries. Details for evaluating complex queries will be discussed in the next section. Let $R(q)$ be the set of *schema-valid* partitions to a star-join query q. The overall evaluation cost of q can be modeled as:

$$c(q) = \sum_{R \in R(q)} (a + k \cdot S(R)) \tag{1}$$

where a is a startup cost (e.g. query planning) of evaluating the query q in a partition R, which is irrelevant to the size of data in R. $S(R)$ is the size of the partition R, and k is a factor to the size of a partition. We assume that the query execution cost is linear to the size of the partition because a star-join query is evaluated based on the merge of indexes. The above query evaluation cost can be rewritten as:

$$\begin{aligned} c(q) &= a \cdot |R(q)| + k \cdot S(R(q)) \\ &= a \cdot |R(q)| + k \cdot \hat{S}(R(q)) + k \cdot \check{S}(R(q)) \end{aligned}$$

where $\hat{S}(R(q))$ is the number of all *schema-valid* entities in the partitions of $R(q)$, and $\check{S}(R(q))$ is the number of all non-*schema-valid* entities in the partitions of $R(q)$. For a given dataset, all *schema-valid* entities of q can only appear in *schema-valid* partitions of q. Therefore, $\hat{S}(R(q))$ will be a fixed number regardless of how the RDF dataset is partitioned. In contrast, the number of non-*schema-valid* entities, $\check{S}(R(q))$, is dependent on how data are partitioned. An interesting phenomenon of the above formula is that $|R(q)|$ and $\check{S}(R(q))$ are typically negative correlated. When data are partitioned in finer-granularity, the schema of a partition will be more compact, which means that the number $\check{S}(R(q))$ drops. However, the enlargement of the number of partitions will cause $|R(q)|$ to be enlarged. Therefore, an elaborate solution of partitioning RDF triples is desired to achieve a good trade-off between $\check{S}(R(q))$ and $|R(q)|$. As a clustering problem, find the optimal partitioning solution is known to be NP-hard, thus we need to find an approximate solution of schema clustering.

C. Efficient Schema Clustering

We observe that many entities of similar types have very similar schemas in existing RDF datasets. For examples, movie entities extracted from IMDB, paper entities extracted from DBLP. Those well-formed entities usually hold the majority in the RDF dataset. Based on this observation, we extract partitions from an RDF dataset by clustering those entities of similar schemas together. Considering that some entities are “outliers” in terms of the schemas they have, they are maintained by a separated partition $\bar{R}$, called the fusion partition. The system therefore maintains a number of partitions $R_1, R_2, \ldots, R_n$, and the fusion partition $\bar{R}$. Each entity in the RDF dataset must be in one of the partitions.

The distance of two schemas is measured based on the Jaccard distance. Given two schemas P_1 and P_2, the Jaccard distance is defined as the $d(P_1, P_2) = \frac{|P_1 \cup P_2| - |P_1 \cap P_2|}{|P_1 \cup P_2|}$. Taking the idea of Dirichlet process [17], an incremental clustering algorithm which can efficiently cluster schemas simply by scanning the whole dataset for once, is proposed in [16]. In this clustering algorithm, all distinct schemas of entities are first discovered. All the schemas are ranked based on the numbers of entities they have. Then, a one-pass scanning process is conducted over the ranked schema list. Schemas with large population are processed first. The first schema forms a partition directly. For each following newly scanned schema $P(R')$, it will be compared against all the existing partitions. Schema $P(R')$ can be allocated to an existing partition R only if 1) $P(R') \subseteq P(R)$; 2) $d(P(R'), P(R))$ is no more than the distances of $P(R')$ to any the other partition; 3) $d(P(R'), P(R))$ is no more than some given threshold ε. If the schema $P(R')$ cannot be allocated to any existing partition, a new partition will be created to hold it. In this way, all schemas are allocated into a large number of partitions. Finally, tailed partitions that do not have enough number of entities will be merged together as a fusion partition $\bar{R}$.

The above proposed clustering algorithm [16] may not achieve optimal partitioning strategy that can be achieved by the k-medoids algorithm. It however does not need to specify the parameter k beforehand. The quality of partitions is mainly controlled by the parameter ε, which determines the similarities between the subsequent schemas and the origin schema. The smaller the ε, the more similar of different schemas in each cluster, the larger number of clusters will be generated.

D. Data Placement

After generating the logical partitions of RDF schemas, we need physically assign the partitions to the nodes in a database cluster. Two factors need to be considered for the data placement problem: 1) the workloads of nodes need to be balanced; 2) large logical partitions need to be further partitioned, so that the workloads on such partitions can be alleviated by multiple nodes in parallel. We apply a simple re-partitioning strategy that divides large partitions into small segments by hashing over the subjects of RDF triples. The number of segments to be generated is determined by the size of a partition, and it guarantees that the size of a segment is within a certain range so that a segment can be efficiently processed by a single node. Moreover, triples of the same entity will be assigned to the same segment, which guarantees that star-join queries can be efficiently processed locally.

The placement of partitions (or segments) is a bit straightforward. To achieve good parallelism, segments of the same partition will be assigned to different nodes unless there are not enough nodes to hold them. In this paper, we do not consider the fault tolerance problem that can be addressed through the duplications of segments across multiple nodes. The workload balance across nodes can be easily achieved by a round-robin way of segment assignment. Fig. 2 uses an example to illustrate the partitioning and placement strategies of our solution.

IV. Query Evaluation and Optimization

In our approach, the basic logical query-processing unit (BLQU) is star-join queries. The basic idea of our query process engine is to first decompose a SPARQL query into a number of independent star-join queries. Those star-join queries can be efficiently processed by local nodes

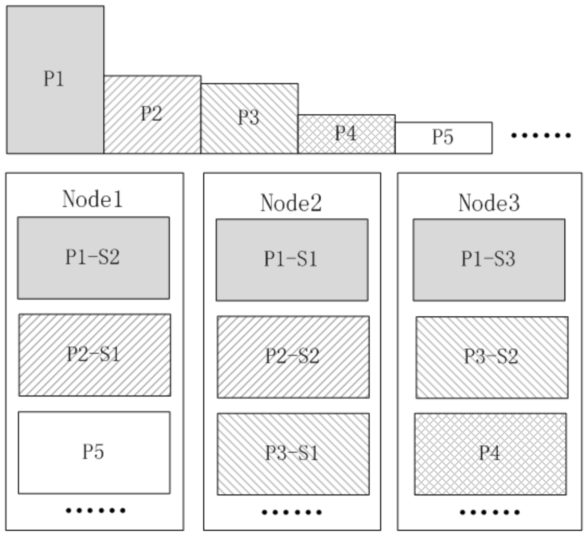


Figure 2. An illustration of data partitioning and placement

without incurring communication across nodes. The results of star-join queries are stored locally for further chain-join processing. Figure 3 shows the framework of the query evaluation and optimization engine.

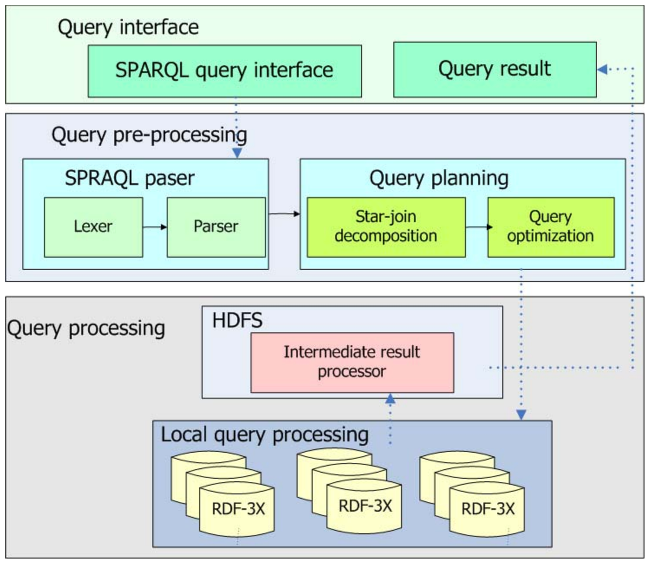


Figure 3. The framework of the query engine

First, a complex chain-join SPARQL query will be decomposed into multiple BLQUs through query parsing and lexical analysis. After that, there will be only *s-o* (or *o-s*) joining relationships between some BLQUs (note that o-o join will be decoupled into two BLQUs). A query optimizer will further analyze the star-join queries to generate an optimized join order. Each star-join query is submitted to the corresponding nodes that store *schema-valid* partitions of the query. Each node in the system is equipped with an RDF-3X

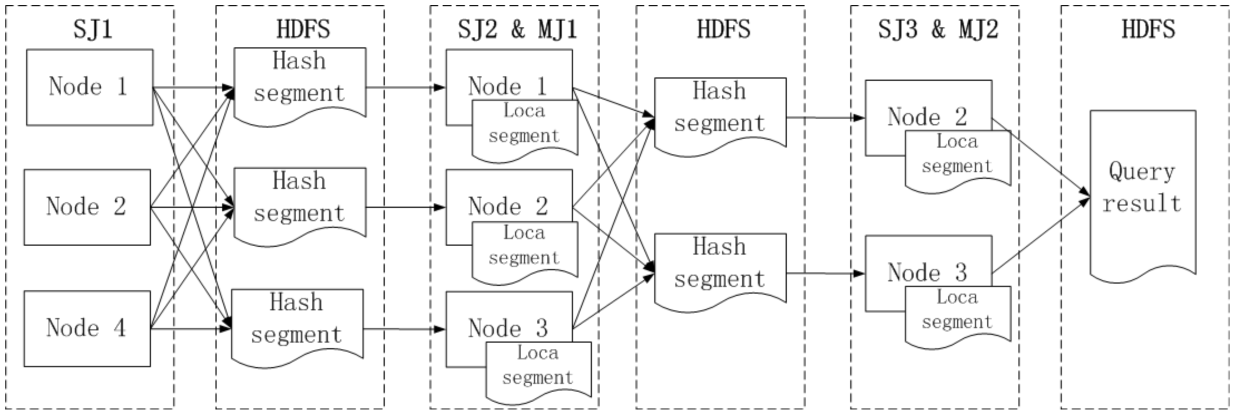


Figure 4. An illustration of an overall query evaluation process

engine, based on which results of a star-join query is locally generated. Then a chain join process is triggered in an order generated by the query optimizer. The intermediate results of chain join are stored on HDFS for two reasons: 1) the intermediate results may be too large to be held in memory; 2) it is for fault tolerance in case of node breakdown. The final results of the query will be generated after a series of chain join operations.

A. Query Decomposition and Star-Join Query Processing

After parsing and lexical analysis, a chain-join SPARQL query will be partitioned into star-join queries based on the join variables in the subjects of basic graph patterns (BGP) specified in the query. BGPs of the same subject form a star join query whose join attribute is the subject. After the query planning over the decomposed star-join queries, they will be forwarded to nodes for further processing.

A node locally processes star-join queries delivered to it based on the RDF-3X query engine. Different star-join queries will be processed in the order they are assigned by the query optimizer. To find proper nodes (containing the *schema-valid* partitions of the query) to process a star-join query, we apply the SFES algorithm that is proposed in [16]. In this algorithm, the system creates inverted lists of predicates for all partitions. Each predicate $p \in P$ has an inverted list l_p recording all the partitions that having p as one predicate in their schema. Given a star-join query q whose predicate set is $P(q)$, all *schema-valid* partitions to q can be efficiently discovered by scanning the inverted lists of all predicates of $P(q)$. A sort merge process will be applied to efficiently retrieve those *schema-valid* partitions from the relevant inverted lists.

Finally, results of star-join queries are produced based on a query evaluation process using the RDF-3X engine over those nodes that store *schema-valid* partitions of the query. Note that, for a logical partition with multiple segments, the same star-join queries will be processed in parallel on those nodes storing the segments of the *schema-valid* partition.

B. Query Optimization and Chain-Join Query Processing

The priority orders of star-join queries are determined by the collisions among chain-join variables. If there is no collision between two chain-join variables, the two star-join queries can be executed in parallel. Otherwise, they will be processed according to the order determined by the collision (i.e., the query in the *o* side of an *s-o* join has a higher priority than the one in the *s* side). With the join order, star-join queries in a local node will be processed in order.

Fig. 4 illustrates an overall query evaluation process. There are three ordered star-join queries: SJ1, SJ2 and SJ3. They will generate two merge-join processes: MJ1 and MJ2. Basically, the intermediate results of the LHS (*o* side) partition of a merge-join are hashed into hash segments based on the same hash function applied to the RHS (*s* side) partition. Those segments are stored in HDFS, from which the nodes storing the RHS partition read the corresponding segments, and conduct a merge-join with its intermediate results of star-join queries locally.

V. Experimental Study

A. Experimental Settings

Our experiments are conducted in a cluster of 8 nodes, with each node configured as an Intel(R) Core(TM)2 6300 @ 1.86GHz, and 3GB DDR2 RAM, running CentOS 6.3 x64. The applied softwares are RDF-3X 0.3.7 and Hadoop 1.0.4. A gigabit Ethernet is applied.

Our experiments are designed for three goals:

- We compare the performance of partitioning methods.
- We compare the query evaluation efficiency of our solution with a relevant solution [7] (shorted as MR-RDF) on both the LUBM synthesized dataset [18] and the BTC real dataset [19]. 15 queries of different join types and selectivities are used for testing.
- We examine the scalability of our solution by adjusting the number of nodes in the cluster.

The BTC (Billion Triple Challenge) [19] is a dataset with more than 3.2 billion triples. It contains triples from 12 sources such as YAGO, DBPedia, Freebase. We choose the BTC2010 as a testing set so that our solution can be evaluated on large scale of real-life RDF data. The dataset has 95,898 distinct predicates. The original data format in the dataset is the NQuads format which includes a subject, a predicate, an object and a context for each triple. We omit some noisy data and the context, thus keep 1,026,823,962 triples (after de-duplication) for evaluation.

LUBM (Lehigh University Benchmark) [18] is a synthesized dataset based on an ontology of the university domain. The size of the LUBM dataset can be set by parameters provided by users. It is now one of the most popular RDF benchmark. The generated LUBM dataset in our experiment includes information of 197,986 universities, 18 predicates, 217 million entities and about 1.1 billions of triples.

B. *Effects of Data Partitioning*

First, we analyze the population distribution of schemas in the BTC dataset. The results are shown in Fig. 5. We can see that among the top 50 schemas, the top 2 schemas take 30% of the total amount of entities. In contrast, the schemas behind the 15th altogether take no more than 1% of the population. Obviously, it is a long tailed distribution. Alternatively, if the dataset is partitioned by predicates, the most popular predicate *type* takes 13% of the whole dataset.

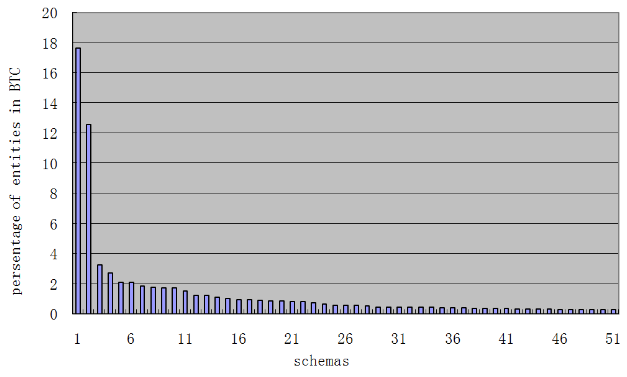


Figure 5. Entities distribution of BTC dataset(top 50 popular entities)

The above results show that the balancing problem of data distribution is a typical problem faced by the data partitioning strategies. We compare the data distribution of our partitioning solution with an alternative where no further partitioning over the clustering results is applied. The results are shown in Fig. 6. Obviously, the partitions generated from schema partitioning are skewly distributed. When we repartition the partitions into segments guaranteeing that no segments are larger than 256MB, the data will be almost evenly distributed.

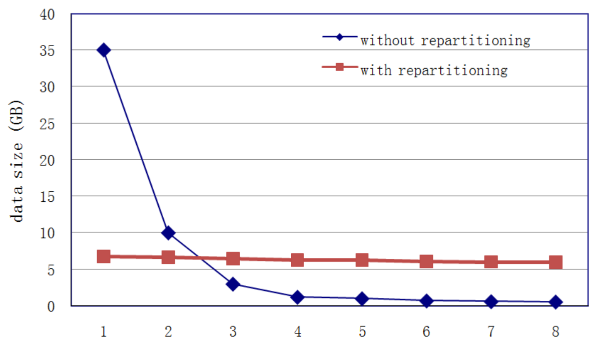
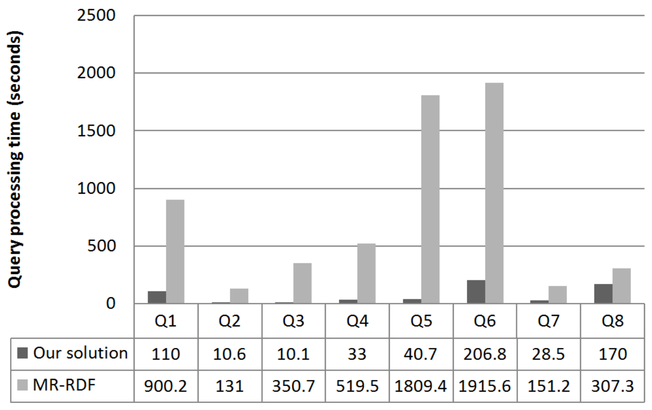


Figure 6. The data distribution across nodes

C. *Query Processing*

We compare the query evaluation time of our solution with that of the MR-RDF [7] on both the LUBM and the BTC datasets. The results of the LUBM dataset are shown in Fig. 7. The queries can be divided into 3 groups. Queries Q1, Q2 and Q6 consist of multiple *s-o* chain-join queries. Queries Q4, Q5 and Q7 consist of only one *s-o* chain-join query. Queries Q3 and Q8 consist of single star query.

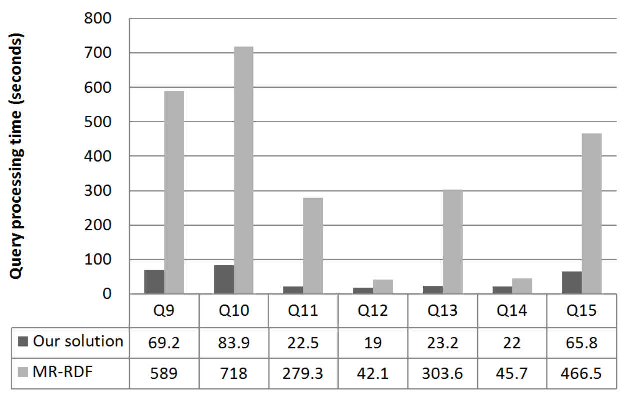


	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
■ Our solution	110	10.6	10.1	33	40.7	206.8	28.5	170
■ MR-RDF	900.2	131	350.7	519.5	1809.4	1915.6	151.2	307.3

Figure 7. Query elapsing time on the LUBM dataset

The experiment results (shown in Fig. 7) indicate that the performance of our solution is much better than that of the MR-RDF. Especially, the query processing speed of Q2-Q5 and Q7 is 10-30 times faster than that of MR-RDF. The reasons are: 1) these queries are distributed to more than 4 nodes. The query workloads on the these nodes are balanced; 2) the intermediate results of these queries are very small (no more than 1000 triples), so the cross node communication cost is very small; 3) these queries consist of common predicates and known objects, so that most irrelevant data segments can be filtered during the query planning process. Queries Q1 and Q6 consist of complex join relations, e.g. there are pair-wise joins among ?X, ?Y and ?Z. More than one MapReduce jobs are required in the MR-RDF solution for this kind of queries. Our solution does not need to pay time cost in starting and scheduling the MapReduce jobs. Therefore, our solution does much better job on this kind of queries. Query Q8 is a simple star-join query. The results of this query are very large. Both solutions need to write the large results into HDFS, so the time cost are very close.

Queries for experiments on the BTC dataset can also be divided into 3 groups. Q9, Q10 and Q15 consist of multiple *s-o* chain-join sub-queries. Q12 and Q14 consist of one *s-o* chain-join sub-query. Q11 and Q14 consist of unknown predicates. The processing time of queries on this dataset is shown in Fig. 8.



	Q9	Q10	Q11	Q12	Q13	Q14	Q15
■ Our solution	69.2	83.9	22.5	19	23.2	22	65.8
■ MR-RDF	589	718	279.3	42.1	303.6	45.7	466.5

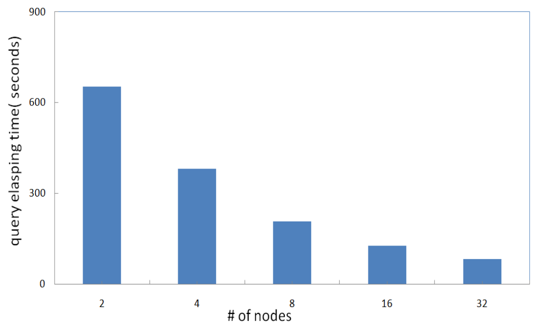
Figure 8. Query elapsing time on the BTC dataset

The BTC dataset contains much more predicates than the LUBM dataset. As a result, the intermediate results of queries are much smaller, and therefore query processing is much faster. Similar to the LUBM dataset, the queries with smaller intermediate results (e.g. , Q11 and Q13) are processed much faster by our solution than by the MR-RDF. In general, it can be observed that the query evaluation efficiency of our solution is significantly impacted by the intermediate results. Large intermediate results incur a lot of I/Os and cross-node communication.

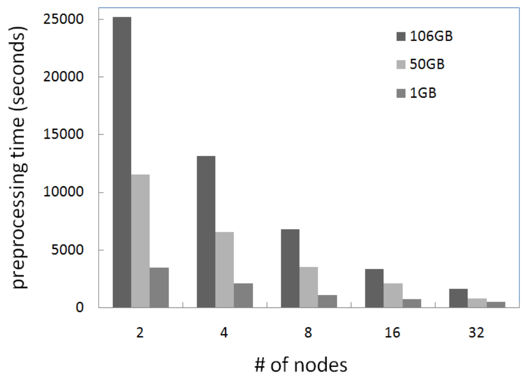
D. Scalability Test

To test the scalability of our method, we adjust the size of the LUBM dataset and the number of nodes in the cluster. Fig. 9(a) shows the scalability of query processing by using a complex query Q6. Fig. 9(b) shows the scalability of data preprocessing (data partitioning and placement).

Because the query Q6 consists of multiple *s-o* chain-join operators which is more complex than the others, we use it to test the scalability of our solution. Note that the average computational cost of queries is meaningless because the cost of different queries varies significantly. In Fig. 9(a), it can be seen that when the number of nodes increases from 2 to 32, the query efficiency improves a lot. The reason is that in a 2-node cluster, there are too many data segments on each node. The query workload on each node is very high. In Fig. 9(b), the data preprocessing (partitioning and loading segments into RDF-3X) time decreases when we increase the number of nodes and reduce the data size. It confirms that our partitioning method is scalable.



(a) For query processing



(b) For data preparing

Figure 9. Scalability test

VI. Conclusion

In this paper, we propose an effective data partitioning and placement solution that is able to evenly distribute RDF triples and workloads across the nodes. By using the SFES algorithm, *schema-valid* partitions can be efficiently identified. Through the query decomposition, star-join queries can be locally processed in parallel. The repartitioning mechanism supports the execution of merge-join in parallel, which further speeds up the chain-join query processing. The experiments over huge RDF datasets demonstrate that our solution is an efficient and scalable SPARQL query evaluation solution in a cluster environment.

Acknowledgment

This work was partially supported by NSFC under the grant No. 61170010 and 61003085. It was partially done when the authors worked in SA Center for Big Data Research in Renmin University of China. This Center is funded by a Chinese National 111 Project Attracting International Talents in Data Engineering Research.

References

[1] *LinkedOpenData: http://linkeddata.org/?*

[2] *Freebase: http://www.freebase.com/.*

[3] *DBPedia: http://dbpedia.org/.*

[4] T. Neumann and G. Weikum, "Rdf-3x: a risc-style engine for rdf," *PVLDB*, vol. 1, no. 1, pp. 647–659, 2008.

[5] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "Yars2: A federated repository for querying graph structured data from the web," in *ISWC/ASWC*, 2007, pp. 211–224.

[6] K. Rohloff and R. E. Schantz, "High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store," in *PSI EtA*, 2010, p. 4.

[7] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham, "Heuristics-based query processing for large rdf graphs using cloud computing," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 9, pp. 1312–1327, 2011.

[8] J. Huang, D. J. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *PVLDB*, vol. 4, no. 11, pp. 1123–1134, 2011.

[9] S. Harris and N. Gibbins, "3store: Efficient bulk rdf storage," in *PSSS*, 2003.

[10] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A generic architecture for storing and querying rdf and rdf schema," in *International Semantic Web Conference*, 2002, pp. 54–68.

[11] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds, "Efficient rdf storage and retrieval in jena2," in *SWDB*, 2003, pp. 131–150.

[12] T. Tran, H. Wang, and P. Haase, "Hermes: Data web search on a pay-as-you-go integration infrastructure," *J. Web Sem.*, vol. 7, no. 3, pp. 189–203, 2009.

[13] C. Guéret, S. Kotoulas, and P. T. Groth, "Triplecloud: An infrastructure for exploratory querying over web-scale rdf data," in *Web Intelligence/IAT Workshops*, 2011, pp. 245–248.

[14] H. Kim, P. Ravindra, and K. Anyanwu, "From sparql to mapreduce: The journey using a nested triplegroup algebra," *PVLDB*, vol. 4, no. 12, pp. 1426–1429, 2011.

[15] J. Myung, J. Yeon, and S.-g. Lee, "Sparql basic graph pattern processing with iterative mapreduce," in *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, ser. MDAC, 2010, pp. 1–6.

[16] F. Du, Y. Chen, and X. Du, "Partitioned indexes for entity search over rdf knowledge bases," in *DASFAA (1)*, 2012, pp. 141–155.

[17] Z. Zhang, G. Dai, and M. I. Jordan, "Matrix-variate dirichlet process mixture models," *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 980–987, 2010.

[18] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *J. Web Sem.*, vol. 3, no. 2-3, pp. 158–182, 2005.

[19] *BTC: http://challenge.semanticweb.org/.*

Appendix

Prefix list:rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
ub: <http://www.lehigh.edu/zhp2/2004/0401/univ-bench .owl#>
dbpedia: <http://dbpeida.org/property/>
dbpedias: <http://dbpeida.org/resource/>
geo: <http://www.geonames.org/>
pos: <http://www.w3.org/2003/01/geo/wgs84_pos#/>
owl: <http://www.w3.org/2002/07/owl#>

Q1 : select ?X, ?Y, ?Z where
{?X rdf:type ub:GraduateStuden . ?Y rdf:type ub:University . ?Z rdf:type ub:Department . ?X ub:memberOf ?Z . ?Z ub:subOrganizationOf ?Y . ?X ub:undergraduateDegreeFrom ?Y}

Q2 : select ?X where {?X rdf:type ub:GraduateStudent.?X ub:takesCourse <http://www.Department0.University0.edu/ GraduateCourse0>}

Q3: select ?X where
{?X rdf:type ub:Publication . ?X ub:publicationAuthor <http://www.Department0.University0. edu/AssistantProfessor0>}

Q4 : select ?X ?Y where {?X rdf:type ub:Student. ?Y rdf:type ub:Course. ?X ub:takesCourse ?Y. }

Q5 : select ?X ?Y ?Z where {?X rdf:type ub:Student. ?Y rdf:type ub:Department. ?X ub:memberOf ?Y.
?Y ub:subOrganizationOf
<http://www.University0.edu>. ?X ub:emailAddress ?Z}

Q6 : select ?X ?Y ?Z where {?X rdf:type ub:Student. ?Y rdf:type ub:Faculty. ?Z rdf:type ub:Course. ?X ub:advisor ?Y. ?Y ub:teacherOf ?Z. ?X ub:takesCourse ?Z}

Q7 : select ?X ?Y where {?X rdf:type ub:Chair. ?Y rdf:type ub:Department. ?X ub:worksFor ?Y. ?Y ub:subOrganizationOf <http://www.University0.edu>}

Q8 : SELECT ?X where {?X rdf:type ub:UndergraduateStudent}

Q9: select ?gn ?fn where { ?gn ¡givenNameOf¿ ?p. ?fn <familyNameOf> ?p. ?p <type> "scientist"; <bornInLocation> ?city; <hasDoctoralAdvisor> ?a. ?a <bornInLo- cation> ?city2. ?city <locatedIn> "Switzerland". ?city2 <locatedIn> "Germany". }

Q10: select ?n where { ?a <isCalled> ?n; <type> "actor"; <livesIn> ?city; <actedIn> ?m1; <directed> ?m2. ?city <locatedIn> ?s. ?s <locatedIn> "United States". ?m1 <type> "movie"; <producedInCountry> "Germany". ?m2 <type> "movie"; <producedInCountry> "Canada". }

Q11: select distinct ?n where { ?p <isCalled> ?n; [] ?c1. [] ?c2. ?c1 <type> <village>; <isCalled> "London". }

Q12: select ?n1 ?n2 where {?p1 <isCalled> ?n1; <bornInLocation> ?city; <isMarriedTo> ?p2. ?p2 <isCalled> ?n2; <bornInLocation> ?city. }

Q13: select distinct ?l ?long ?lat where { ?a [] "Barack Obama". ?a dbpedia:placeOfBirth ?l. ?l pos:lat ?lat. ?l pos:long ?long. }

Q14:select distinct ?d where {?a dbpedia:senators ?c.?a dbpedia:name ?d.?c dbpedia:profession dbpediares:Veterinarian.?a owl:sameAs ?b.?b geo:ontology#inCountry geo:countries#US.}

Q15: select distinct ?a ?b ?lat ?long where { ?a dbpedia:spouse ?b. ?a dbpedia:wikilink dbpediares:actor. ?b dbpedia:wikilink dbpediares:actor. ?a dbpedia:placeOfBirth ?c. ?b dbpedia:placeOfBirth ?c. ?c owl:sameAs ?c2. ?c2 pos:lat ?lat. ?c2 pos:long ?long. }