Efficient SPARQL Query Evaluation via Automatic Data Partitioning

Tao Yang[#], Jinchuan Chen[†], Xiaoyan Wang[#], Yueguo Chen[†], and Xiaoyong Du[#†]

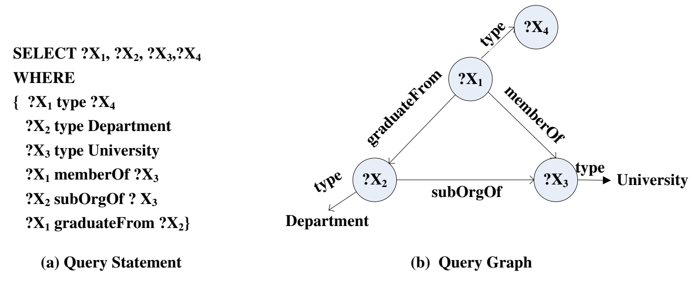
[#]School Of Information, Renmin University of China
[†]Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), MOE, China
{yangtao2007,jcchen,wxy,chenyueguo,duyong}@ruc.edu.cn

Abstract. The volume of RDF data increases very fast within the last five years, e.g. the Linked Open Data cloud grows from 2 billions to 50 billions of RDF triples. With its wonderful scalability, cloud computing platform like Hadoop is a good choice for processing queries over large data sets. Previous works on evaluating SPARQL queries with Hadoop mainly focus on reducing the number of joins through careful split of HDFS files and algorithms for generating Map/Reduce jobs. However, the way of partitioning RDF data could also affect the performance. Specifically, a good partitioning will greatly reduce or even totally avoid cross-node joins and significantly reduce the cost of query evaluation. Based on HadoopDB, this work processes SPARQL queries in a hybrid architecture where Map/Reduce takes charge of the computing tasks and an RDF query engine, RDF-3X, stores the data and evaluates join operations over local data. Based on analysis of query work-loads, we propose a novel algorithm for automatically partitioning RDF data. We also present an approximate solution to physically place the partitions in order to reduce data redundancy. All the proposed approaches are evaluated by extensive experiments over large RDF data sets.

1 Introduction

RDF, an abbreviation for *Resource Description Framework*, is a model recommended by W3C for data interchange on the Web. Basically, RDF represents each fact as a triple $< s, p, o >$. RDF dataset is essentially a graph with each vertex per entity and each edge per relationship between two entities. The SPARQL query is a widely accepted query language for accessing RDF triples. A SPARQL query contains a set of *triple patterns*, i.e. at least one element of *s, p*, and *o* is a variable. It can also be represented as a graph, with some vertexes or edge labels (predicates) as variables. The results of a SPARQL query are sub-graphs of the RDF graph. Hence a SPARQL query is basically a sub-graph pattern matching task. As a running example, Fig. 1 illustrates the statement and corresponding query graph of a SPARQL query, which tries to find all the persons who obtained his/her degree from the same university which he/she currently belongs to.

In recent years, with the quick proliferation of RDF data, it is often infeasible to store all RDF triples in a single node, which motivates the interests of processing SPARQL queries in a distributed environment, especially within the Hadoop platform [10, 12]. Benefiting from the Map/Reduce framework, these works obtain high scalability of evaluating SPARQL queries over billions of RDF triples. However, SPARQL queries usually contain multiple joins and these join operations may be conducted in multiple worker nodes, which is not favored by Map/Reduce because cross-node communications are not permitted in the map phase. Thus a SPARQL query may need multiple M/R jobs which is quite expensive since each such job requires several seconds to fire up, not to speak of the time cost of communication between multiple nodes.



```
SELECT ?X1, ?X2, ?X3,?X4
WHERE
{  ?X1 type ?X4
   ?X2 type Department
   ?X3 type University
   ?X1 memberOf ?X3
   ?X2 subOrgOf ? X3
   ?X1 graduateFrom ?X2}
```

Fig. 1. An example of a SPARQL Query

In the distributed database community, a classical technique to reduce cross-node communication is *data partitioning*. The basic idea is to put the tuples which may be involved in a join in the same worker node [4, 14, 6]. For this purpose, usually we need to analyze previous query workloads and identify which tuples or rows are probably appear in the same queries [14, 6]. Taking this idea, this work aims at facilitating scalable and efficient processing of SPARQL queries via automatic data partitioning. Our work is based on the HadoopDB project [3], which proposes a hybrid architecture by combining Map/Reduce and databases. The principle idea is to execute M/R jobs over a database cluster. In this way, the hybrid system could inherit the scalability and fault-tolerance from Map/Reduce framework while obtaining high efficiency from the powerful capability of processing complex operators like joins and aggregations in traditional databases. In our work, each worker node is equipped with a RDF-3X query engine [13], a state-of-art single-node system for processing SPARQL queries.

Our partitioning approach is inspired by the observation that in many applications there usually exist some frequent query patterns. A *query pattern* is a special SPARQL query, which essentially defines a code such that a group of similar SPARQL queries can be compiled according to it. For example, the query shown in Fig. 1 can be regarded as a query pattern. The variable X_4 can be replaced by different constants like *Student, Professor, and Staff* and correspondingly generate different queries. The general idea of our partitioning approach is to divide the RDF graph into twigs, or tiny sub-graphs, according to the frequent query patterns[1]. We can then ensure that no cross-node joins are needed when processing SPARQL queries complied with any query pattern. What should also be mentioned is that, thanks to the powerful capability of RDF-3X for processing triple joins, even for queries not in any identified patterns, the query performance of our system would also defeat those works based on Hadoop systems [10].

The work most similar to ours is [9], where the authors also propose to evaluate SPARQL queries over HadoopDB. In [9], the whole RDF graph is divided into several huge sub-graphs based on a graph partitioner `METIS` [2]. These sub-graphs would be stored at different worker-nodes, with triples near the division boundaries replicated to multiple nodes. Based on this partitioning, most queries could be answered based on the triples inside a single node.

Compared with [9], our solution has several significant advantages. First of all, [9] does not consider the dynamic properties in query workloads, and cannot guarantee that there are no cross-node joins for frequent query patterns. Suppose a query pattern happens to involve the triples on the partition boundaries, querying the queries compiled with this pattern have to coordinate triples in different nodes. Secondly, [9] may result in many duplicated triples and does not mention how to alleviate this redundancy. In our solution, the partitioning contains two steps. The first step is exactly a logical partitioning, i.e. it will divide the original dataset into many small parts but does not really move them. At the second step, we will place these partitions into different worker nodes. During this placement phase, we will try to reduce the data redundancy by putting partitions with large overlapping into the same worker node. Finally, the partitioning in [9] is based on graph partitioning, a known NP-complete problem [5], which will cost lots of computational efforts.

The contributions of our work are summarized as follows.

– We propose a query-driven data partitioning approach and based on it develop an efficient solution for processing SPARQL queries over large scale RDF data.
– We prove that the placement problem of reducing data redundancy is NP-hard.
– We present an approximate algorithm for reducing data redundancy, which is based on the LNS (Large Neighborhood Search) solution [14].
– We conduct extensive experiments over two large datasets, i.e. LUBM [8] and BTC [1], to evaluate the efficiency and effectiveness of our proposed approaches.

Next we will illustrate the architecture of our system in Sec. 2. We then discuss the partition and placement approaches in Sec.3. Sec. 4 will report our experimental results. We will discuss related works in Sec. 5 and conclude this paper in Sec. 6.

[1] The query patterns are assumed to be available, and the efforts of analyzing query workloads and identifying frequent patterns exceed the scope of this paper.

2 System Architecture

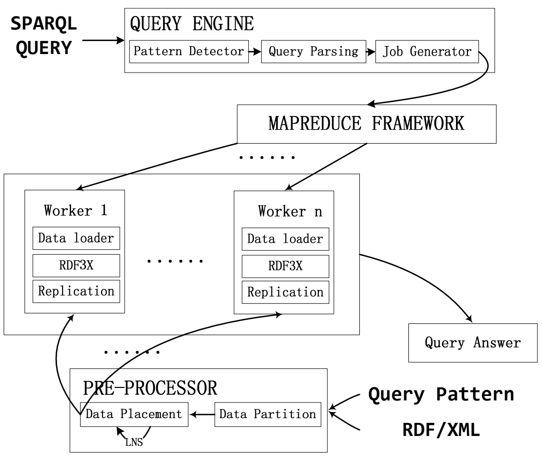


Fig. 2. System Architecture

Our system architecture is illustrated in Figure 2. This system contains three modules including *Data Pre-Processor*, *Query Engine* and a hybrid platform combining Map/Reduce and RDF-3X.

In the data pre-processor, RDF data are partitioned according to query patterns. The *data partition* procedure guarantees that there are no cross-node joins when evaluating any queries compiling to any registered pattern. Each query pattern would have an independent partitioning. All the partitions generated by all the frequent query patterns need to be put into the nodes, through the *data placement* procedure. Note that a triple may appear in multiple partitions since we perform an independent partitioning for each query pattern. Hence the major concern of the placement procedure is to reduce the data redundancy. Once obtaining the partitions, the *data loader* procedure on each worker node will load all triples to the RDF-3X database installed in that machine.

In the query engine, after receiving a SPARQL query, the *pattern detector* figures out whether this query matches any query pattern. If YES, because the triples of each sub-graph matching this query have been placed to a single worker, we can pushdown the whole query to RDF-3X and simply generate one M/R job to retrieve the results. For those queries matching no patterns, we just generate M/R jobs according to the algorithm in [10]. In practice, we can design query patterns to accommodate as many queries as possible, e.g. by replacing more constants with variables. Finally, the query engine submits the jobs to MapReduce framework.

Queries are executed using MapReduce with RDF-3X as a local engine. MapReduce checks configuration files and locates data replications with job parameters. Then each task sends SPARQL queries to RDF-3X installed in each worker node. Query results are returned back as the *InputFormat* of `Mapper`.

3 Data Partitioning

We now illustrate the data partitioning process, which contains two steps, i.e. query-driven partitioning and placement.

3.1 Query-Driven Partitioning

Since each SPARQL query pattern can be regarded as a directed graph, we will use "edge" and "triple pattern" interchangeably when the meaning is clear from the context. For each such sub-graph matching a given query pattern, if all triples in this sub-graph are placed into the same worker node, there would be no cross-node joins when evaluating all queries within this pattern. This property will be guaranteed by our partitioning algorithm. Before going into the algorithm details, we first illustrate a framework to present the principle idea of our partitioning algorithm.

Algorithm 1: PartitioningFramework(RDF dataset D, Query Q)

```
1  $Q' \leftarrow$ an empty graph;
2  Randomly choose an edge $e_0$ from $Q$ and insert it into $Q'$ ;
3  $S \leftarrow$ all triples matching $e_0$;
4  Randomly choose a partition for each triple in $S$;
5  Record the partition information;
6  while $E(Q') < E(Q)$ do
7      Select an edge $e$ such that $e \in E(Q) \setminus E(Q')$ and $e$ is connected to $Q'$;
8      Insert $e$ into $Q'$;
9      $S \leftarrow$ all triples matching $e$;
10     foreach triple $t$ in $S$ do
11         Find the set of triples $S'$ that can be joined with $t$ according to $Q$
           and have already been partitioned;
12         Put $t$ in each of the partition containing at least one triple in $S'$;
13     Record the partition information;
```

Algorithm Framework Alg. 1 illustrates a framework of our partition algorithm. We first initialize an empty graph Q', which will be used to record the progress of partitioning. Next, Step 2 tries to randomly choose an edge from Q and add it to Q'. All triples matching this edge will be loaded by scanning the whole dataset (Step 3), and each matched triple will be assigned to a random partition (Step 4). We will store the partitioning informations, e.g. the partition which a triple is assigned to, in a table (Step 5). After processing this initial edge, we then add other edges in Q to Q' one by one. Step 7 claims that the edge which we choose at each step must be connected to Q', which is a critical requirement to ensure the algorithm target. Again, we retrieve all triples matching this new edge (Step 9). Now we need to decide the partitions for these matched triples. For this purpose, we basically conduct a join according to the structure of Q between these new coming triples and those have already been partitioned (Step 11).

Lemma 1. *For any query pattern Q, all sub-graphs matching Q will have their triples assigned to the same partition if we divide the original dataset according to Algorithm 1.*

Proof. Without lose of generalization, suppose Q contains m edges. We change the indices of these m edges and get $e_1, \cdots, e_m$ such that e_i is the i^{th} edge inserted into Q' according to Algorithm 1. Any sub-graph matching Q also contains m triples. Again, we change the indices of the m triples in a specific sub-graph to obtain $t_1, \cdots, t_m$ so that t_i has e_i as its corresponding triple pattern in Q. It is not hard to see that t_i must be placed into the same partition as t_{i-1} for $i = 2, \cdots, m$. Thus all these m triples must appear in the same partition which is initially decided by t_1.

The Partition Algorithm The framework in Algorithm 1 cannot be directly applied in practice. This algorithm requires a table or index to record the partitioning decision for each processed triple, which is not feasible due to the large volume of RDF dataset. Moreover, the RDF dataset is usually quite skew. Some triple patterns may have huge number of matched triples. This skewness must be handled carefully otherwise the performance could be very bad. We now illustrate several techniques utilized for overcoming these shortcomings and, based on these techniques, present the partitioning algorithm.

A compact data structure for storing the partitioning results. In Alg. 1, for each e chosen to be inserted into Q', all the triples matching e should be checked to see whether they could join with previous partitioned triples. This triple-to-triple join is quite costly in terms of both computation and memory with the existence of billions of triples. In order to improve the performance, we now present a compact data structure to store the partitioning results.

This structure is based on the following observation. Any triple t which could be joined with existing ones must satisfy the following two requirements: (1) its corresponding triple pattern e must share at least one common variable with one or more edges in Q', and (2) there exists at least one partitioned triple t' such that both t and t' are assigned with the same value for this common variable. Therefore, instead of recording all the partitioned triples, we just need to keep all the distinct values of each variable that have shown in any partitioned triples. Specifically, we construct a Hash table for each variable v in Q' containing a set of (var_key, pos) entries with one var_key per distinct value of v, and the corresponding pos be the identifier no. of the partition storing all the triples having the same value, i.e. var_key, to this variable v.

Thus every time we join a candidate triple with previous allocated ones, our algorithm will first look up the Hash table with its value on the joined variable and then assign this triple to the corresponding partition indicated by the returned *pos* value. In the meanwhile, if this triple contains another variable, its value should be added to the index for the follow-up operations.

Algorithm 2: Partitioning(TripleSet D, Query Q, Int n)

```
1  $TP \leftarrow$ estimate$(Q, D)$;
2  $Q' \leftarrow$ an empty graph;
3  while $|E(Q')| < |E(Q)|$ do
4      $e \leftarrow$ chooseEdge$(Q, Q', TP)$;
5      $S_{Temp} \leftarrow$ loadTriples($D$, $e$);
6      foreach $t \in S_{Temp}$ do
7          if $E(Q')=0$ then
8              $i \leftarrow$ a random value in $[1, \cdots, n]$;
9              foreach $v$ in $Var(e)$ do
10                 putIndex($\Gamma$,hash($t$,$e$,$v$),$i$);
11         else
12             foreach $v$ in $Var(e) \cap Var(Q')$ do
13                 $i \leftarrow$ readIndex($\Gamma$,hash($t$,$e$,$v$));
14                 if $i < 0$ then
15                     continue;
16                 else
17                     add $t$ into $S_i$;
18             foreach $v$ in $Var(e) \cap (Var(Q) \setminus Var(Q'))$ do
19                 $i \leftarrow$ readIndex($\Gamma$,hash($t$,$e$,$Var(e) \setminus v$));
20                 putIndex($\Gamma$,hash($t$,$e$,$v$),$i$);
21     insert $e$ to $Q'$;
```

Choosing edges based on selectivity estimation. In practice, the selectivities of the triple patterns could be quite skew. For LUBM dataset containing 1 billion triples, if we first choose the edge *?*X_1 *type* *?*X_4 in the example query in Fig.1, about 10^9 of records need to be retrieved and recorded in the index. Clearly, not all the records can satisfy the query. Many retrieved triples have no contributions to the query results. In this work, we adopt the *selectivity estimation* technique to improve the performance. As a classical method in the database community, the principle of *selectivity estimation* is to evaluate predicates with low selectivities first in order to reduce the number of tuples involved in joins. We utilize a simple heuristic to estimate the selectivity of each triple pattern. Suppose the number of triples contained in a predicate is num_p, and the number of distinct values of the variable in this triple pattern is num_v, the selectivity of this triple pattern is estimated by num_p/num_v. For details, please refer to our technical report [18].

The Algorithm Now, we are ready to present the partitioning algorithm, which is listed in Alg. 2. Step 1 is to construct a list TP storing the selectivities of each triple pattern by analyzing D. The *chooseEdge* function in Step 4 is to select a new edge which should be connected to Q'. This selection is based on the priorities stored in TP. Steps 7-10 is to process the first edge. Each triple matching the first edge will be assigned to a random partition (Step 8). A *hash* function computes a Hash code for the variable value of this triple with this triple pattern, and this code would be used as the key to store the partition result in an index Γ(Step 10). For subsequent edges, a variable on the edges may be *join variable*, i.e. appearing in Q', or non-joinable variable, i.e. not in Q'. The partition of the current triple is decided by its *join variables* through checking the index(Step 13). Note that a triple cannot be joined if we cannot find its key in the index(Steps 14-15). We also store the partition information for the non-joinable variables (Steps 18-20), which may be used to link with follow-up edges. Note that there is at most one non-joinable variable for each edge.

3.2 Placement

According to the partitioning process shown in Sec. 3.1, each query pattern will generate n partitions. We now discuss how to physically place these partitions in different worker nodes. Note that when partitioning the original RDF dataset according to a query pattern Q, those triples which could not satisfy this pattern will not be assigned to any partition. Thus, after processing all the m query patterns, there are still a large portion of triples satisfying no patterns and therefore are not partitioned. In practice, such triples will be seldom accessed and we call them *cold* triples. Each of these cold triples would be assigned to a randomly chosen worker node. In this section, we focus on the placement of the *hot* triples, i.e. those probably satisfying at least one query pattern. Since our partitioning algorithm guarantees no cross-node joins for queries compiled to frequent patterns, the major concern of placement is to reduce the data redundancy, i.e. the number of replicated triples among different worker nodes. We will discuss how to estimate data redundancy and give the definition of the placement problem. Then we will prove that the placement problem is NP-hard. Finally we will illustrate an efficient approximate solution for this problem.

Problem Definition During the partitioning process, we totally generate $m \cdot n$ partitions. Hence each reasonable placement solution, denoted by P, needs to arrange these $m \cdot n$ partitions in a $m \cdot n$ matrix such that: i) the i^{th} column of P contains all the partitions corresponding to the i^{th} query pattern ($i = 1, \cdots, m$) and, ii) all partitions in the j^{th} row would be put into the j^{th} worker node ($j = 1, \cdots, n$). Thus a placement solution is exactly a $m \cdot n$ matrix and there could be n^m different possible solutions. The data redundancy of a placement solution P could be evaluated by the following equation.

$$\gamma_P = \sum_{\forall i,j(i \neq j)} \gamma_{i,j} - \sum_{i=1}^{n} \sum_{j=1}^{m} \gamma_{i,j} \qquad (1)$$

Here $\gamma_{i,j}$ is the number of replicated triples in the i^{th} and j^{th} partitions, and γ_P means the overall redundancy of this solution. The value of each $\gamma_{i,j}$ can be estimated based on the index described in Sec. 3.1. The details are skipped due to page limit. Interested readers are recommended to read our technical report [18]. The first part on the right side is basically the total number of replicated triples among all the $m \cdot n$ partitions. The value of $\sum_{j=1}^{m} \gamma_{i,j}$ is the overall redundancy among all partitions in the j^{th} row. The second part is just to compute the sum of all redundancy in each row. When evaluating γ_P, the row-level redundancy should be subtracted since all partitions in a row would be put in the same worker node. Note that the first part is a constant for all different placement solutions. Hence in order to minimize γ_P, we just need to maximize the value of the second part.

Definition 1. *Given the partitioning results of the m query patterns, i.e. a set of $m \cdot n$ partitions,* **the placement problem** *is to minimize the data redundancy as defined in Equation 1 by arranging these $m \cdot n$ partitions into the n worker nodes.*

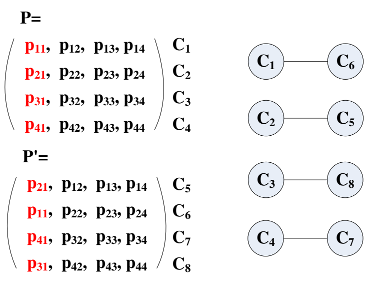


Fig. 3. An example of placement solution

Complexity Analysis. The *placement problem* could be transformed to the *maximum weight independent set* problem (MWIS in short). For any graph where each vertex is attached with a positive weight, a maximum independent set (MIS) is a set of vertexes in this graph which are pairwise disconnected and, all other vertexes should have at least one neighbor in this set. A MWIS is just the heaviest MIS. Let us define a *reasonable combination*, denoted as C, as a set of m partitions each of which corresponds with a distinct query, i.e. a row in a placement solution. Hence there could be totally n^m reasonable combinations. We then build a graph $\mathcal{G}$ by adding a node for each reasonable combination, and adding an edge between two nodes if and only if their underlying combinations contain overlapping partitions. Each node is attached with a weight which is equal to the overall redundancy of this combination. Fig. 3.2 illustrates a simple example for partitioning a dataset into four nodes according to four queries. On the left part, there are two possible placement solutions, P and P'. Each element of the two matrixs, e.g. p_{11}, is a partition generated in the partitioning process. The difference between P and P' are highlighted in red color. Each row represents a reasonable combination with their labels, i.e. $C_1, \cdots, C_8$, listed on the right. The graph on the right part of this figure contains eight nodes for these combinations, with edges connecting combinations with overlapped elements. For example, the nodes C_1 and C_6 are connected since both nodes contain the same element p_{11}.

Algorithm 3: Placement(Solution P)

```
1 $SearchArray \leftarrow$ Relax($P$);
2 $best \leftarrow$ Evaluate($P$);
3 $scoreOfOneSearch \leftarrow$LocalSearch($SearchArray$, $best$);
4 while $scoreOfOneSearch > best$ do
5     $best \leftarrow scoreOfOneSearch$;
6     $ideal \leftarrow$ array stored in LocalSearch;
7     $SearchArray \leftarrow$ Relax($ideal$);
8     LocalSearch($SearchArray$, $best$);
9 return $ideal$;
```

Note that in any placement solution, there will be no repeated partitions and all the n^m partitions should show up. Clearly, each maximum independent set will constitute a solution P, and the optimal solution is exactly the one with the largest sum-of-weight. The MWIS problem is known to be NP-hard [16], and we have to resort to some approximate approaches.

A LNS-based Approximate Algorithm In this paper, we adopt the LNS(Large Neighborhood Search) algorithm [14]. The principle of LNS is an iterative process. Starting from an initial solution, each iteration will search the nearby solutions of the previous optimal solution, and repeat this procedure if finding a better solution. Otherwise, if no better solutions are found in the neighborhood, the current optimal solution will be output as the result. The algorithm is illustrated in Algorithm 3. Firstly, we takes the output of the partitioning algorithm as the initial solution. Then in relaxation part, some placed parts in the initial solution will be selected to be relaxed according to proportion ratios given as parameters (Step 2). In the core phrase of LNS, the LocalSearch function will explore nearby solutions and try to find better solutions (Step 4,9). The process will be repeated if a better solution is met during the local search process (Steps 5-9).

4 Experimental Analysis

4.1 Experiment Setup

Hardware We perform our experiments on a 8 node cluster. Each node has the following configuration: two 2.4GHz Intel(R) Xeon(R) E5654 processors, 48GB main memory and 250G disk space. We run the partitioning and placement algorithms on one of the machine in cluster with an extra 4.5T hard disk.

Software. We modify the source codes of Hadoop-0.20.2 in order to adapt with RDF storage. The version of the RDF-3X engine used in our experiments is 0.3.7. In the *pre-processor* module, we parse each SPARQL query with jena-2.6.2 together with arq-2.8.3 [17]. In comparison with the state-of-art, we also implement the system in [10], called *TKDE11* in this paper. We do not compare the performance with the system in [9], since it usually evaluates SPARQL queries over database clusters instead of the Map/Reduce platform.

DataSets and Queries. Throughout our experiments, we use one synthetic dataset, the Lehigh University Benchmark(LUBM) [8] and one realistic dataset, the Billion Triple Challenge 2010(BTC) [1]. LUBM generates synthetic data about universities on a university domain ontology. Besides data generator, LUBM also has 14 standard queries focusing on both scalability and inference testing. In our experiments, we generate a dataset of 10,000 universities with default parameters. The LUBM dataset contains around 1.1 billion triples. BTC contains the triples from twelve sources such as Yago, DBPedia, and Freebase [7]. We choose BTC to test effectiveness and efficiency of our solutions on large scale and real-life RDF data. After cleaning the noisy or duplicated data, we obtain about 1.28 billion triples. Before the query execution, we also encode each field of these triples into integers to facilitate the query processing.

Query patterns are obtained by rewriting the SPARQL queries. We simply unbound some concrete values as variables in some triple patterns. For example, in LUBM query1, we turn triple pattern `?X ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0>` into `?X ub:takesCourse ?Y` to generate a query pattern. The complete list of all queries used in our experiments can be found in our technical report [18].

Table 1. Query running time in seconds of LUBM 10000 dataset

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Our Solution	16.3	235.2	16.3	16.3	16.3	255.4	22.4
TKDE11	116.3	687.6	174.4	757.8	371.1	342.8	289.2
	Q8	Q9	Q10	Q11	Q12	Q13	Q14
Our Solution	28.3	165.2	16.4	15.3	17.5	74.5	213.3
TKDE11	1320.3	1371.8	184.5	103.3	56.0	91.0	325.9

4.2 Evaluation

Partitioning Time. In our system, the time for partitioning the LUBM dataset is about 20 minutes. The RDF-3X needs about 40 minutes for loading these triples and building the index.

Query Performance Next, we will compare the performance between our solution and TKDE11, and evaluate the effectiveness of our placement algorithm for reducing data redundancy.

Table 1 illustrates the query performance of our solution and TKDE11. We can observe that our solution always performs better than TKDE11. Moreover, for most queries such as Q1, Q3-Q5, and Q7-Q11, the performance of our solution is about one order of magnitude faster than its competitor. These results validate the principle idea that good data partitioning can significantly improve query performance. Next we will analyze the results in detail.

1. For the queries with simple semantic and small result set like Q1, Q3-Q5, Q7-Q8, Q10, and Q11-Q12, our solution performs about 10 times faster than TKDE11. The reasons are two folds: 1) there are only one or two join variables in those queries and their triple patterns usually contain constants, which ensure RDF-3X to utilize its index for selecting candidate triples very fast and, 2) their result sets are quite small, less than 10^3 triples, and little time cost is needed in data transformation between the map and reduce processes.
2. Q2 and Q9 are queries with complex structure, and small result set. Specifically, there is a triangle relationship among three variables, say $?X$ $?Y$ and $?Z$. When dealing with these queries, TKDE11 will generate two M/R jobs with job1 evaluating two joins on $?X$ and $?Y$ separately, and job2 joining the output of job1 with $?Z$. The time cost is huge with large middle files written to and read from HDFS files. On the contrast, our solution can benefit from the powerful capabilities of RDF-3X for evaluating joins. Moreover, our system needs only M/R job based on our partitioning approach.
3. The performance of our method is slightly better than TKDE11 in Q6, Q13 and Q14. These queries all have huge results, larger than 10^8 triples. Lots of triples should be scanned and large amount of triples need to be transfered between RDF-3X engine and Map/Reduce framework. Hence both the query execution and result retrieval phases will cost a lot of time compared with those queries in the above cases.

For the queries matching no frequent patterns, our method has similar performance as TKDE11. The only difference between our solution and TKDE11 method for processing these queries is where the data comes from. We evaluate this case using some queries in LUBM query set by processing each triple pattern at a time. There are no evident difference in the time costs of both methods. Thus we do not report the numbers here.

It should be noted that the queries above are tested using cold runs, which means that the main memory and file system cache were cleared before execution.

Also, because RDF-3X does not support inferencing, we rewrite the reasoning-needed queries in LUBM query set to equivalent ones using union operations before query execution.

The time of query processing tested on BTC is presented in Table 2. Due to the feature of large number of distinct predicates for BTC dataset, the result set for queries over BTC is much smaller than that of the queries over LUBM. Relatively simple query and smaller results explains the resemblance of the query time for our solution in Table 2. It should be noted that there is predicate variable in Q1 and Q4, in which case our solution performs far better than TKDE11 strategy because we do not need to scan the whole triples to get the result.

Table 2. Query running time in seconds of BTC dataset

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Our Solution	16.1	16.2	16.2	15.2	15.9	16.1	16.3
TKDE11	296.1	45.1	95.6	330.3	22.3	74.5	22.8

Placement We test the effectiveness of our placement algorithm with various column adjust number K and row adjust number M, i.e. at each iteration of Algorithm 3, K columns and M rows of the previous matrix are relaxed which define a neighborhood containing $(M!)^K$ different solutions. The y-axis in Figure 4 and Figure 5 is the ratio of redundancy decrease, which is computed by $1 - \gamma'/\gamma$ with γ' and γ are the redundancies of the final solution and initial solution respectively.

As illustrated in Figure 4, it is obvious that the higher K and M are, the better performance is achieved. Figure 5 illustrates the redundancy decrease ratio obtained after each iteration of searching neighborhood with different K and M. Clearly, the redundancy becomes smaller after each iteration, and after several runs the redundancy becomes stable.

5 Related Work

SPARQL query processing. Most previous works on evaluating SPARQL queries over RDF data are based on a single node [13, 7, 17]. The RDF-3X [13] is widely accepted as the state of art for SPARQL query engine, which stores all triples in B+ tree, and builds exhausted indexes of all SPO permutations. Due to the centralized mode, these works cannot scale to handle huge volume of RDF triples which are still increasing in high velocity.

In order to process such huge RDF datasets, [12, 10, 11] suggest to store RDF triples in HDFS files and evaluate SPARQL queries by rewriting them as a series of Map/Reduce jobs. [10] presents a method for generating Map/Reduce jobs and heuristics of dividing RDF triples into separate HDFS files. Myung et. al. [12] propose an algorithm for basic graph pattern matching, and they will process a

SPARQL query by a sequence of Map/Reduce jobs. None of these works consider to improve query performance through better data partitioning. The work in [9] proposes to partition a RDF dataset according to its graphical features and try to avoid cross-node communications by replicating some vertexes near the boundaries of each partition. However, as illustrated in Sec.1, our solution has several evident advantages compared with [9].

Data partitioning. In distributed databases, data partitioning is one of the most important technologies for achieving platform scalability. The partitioning solutions are realized mainly in horizontal partitioning or vertical partitioning [4]. In brief, horizontal partitioning, such as Hash, Round-robin, Range, etc., is to divide a relational table into multiple groups of rows, whereas vertical partitioning is tries to divide a table into several clusters of columns. Recently, along with the quick development of practical applications, researchers tend to use nested horizontal or vertical partitioning methods [15], or even hybrid approaches [4], to achieve better performance. The idea of improving performance through clever partitioning gives us a good inspiration, but all of these works focus on relational databases and cannot be applied directly to partition RDF datasets. The works in [6] design partitioning based on elaborative analysis on query workloads such that frequent queries could be answered more quickly, which is also adopted in this paper in our data partitioning solution.

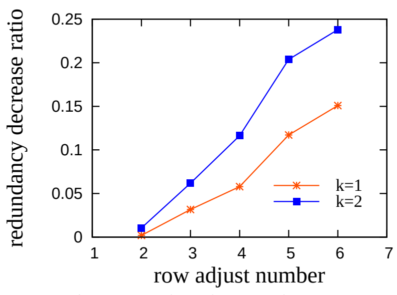


Fig. 4. redundancy decrease

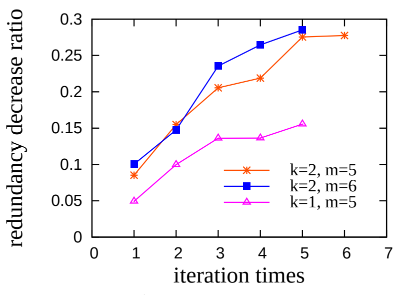


Fig. 5. iteration

6 Conclusion and Future Works

In this paper, we propose a automatic data partitioning approach in order to improve the performance of processing SPARQL queries in a Map/Reduce framework. Compared with previous works, our system can completely avoid cross-node joins for frequent queries and reduce data redundancy. According to the simulation results, our method could accelerate the query processing time by up to two orders of magnitude. Our on-going projects include how to perform update or even migration when there are significant changes on the dataset and/or workload, e.g. large number of new-coming tiples, new identified query patterns, and skewed query accessing.

Acknowledgements. This work is funded by the National Science Foundation of China under Grant No. 61170010, National Basic Research Program of

China (973 Program) No. 2012CB316205, and HGJ Project 2010ZX01042-002-002-03. We also want to express our thankfulness to Sa-Shixuan International Research Centre for Big Data Management and Analytics (hosted in Renmin University of China) which is partially funded by a Chinese National "111" Project "Attracting International Talents in Data Engineering and Knowledge Engineering Research".

References

1. Btc 2010. `http://www.hpi.uni-potsdam.de/naumann/sites/btc2010`.
2. Metis. `http://glaros.dtc.umn.edu/gkhome/views/metis/index.html/`.
3. A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, v.2 n.1:p.992–933, August 2009.
4. S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. *In SIGMOD'04*, pages p.359–370, 2004.
5. K. Andreev and H. Räcke. Balanced graph partitioning. In *SPAA*, pages 120–124, 2004.
6. C. Chang, T. M. Kurç, A. Sussman, Ü. V. Çatalyürek, and J. H. Saltz. A hypergraph-based workload partitioning strategy for parallel data aggregation. In *PPSC*, 2001.
7. F. Du, Y. Chen, and X. Du. Partitioned indexes for entity search over rdf knowledge bases. In *DASFAA'12*, pages 141–155, 2012.
8. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, v.3 n.2-3:p.158–182, October, 2005.
9. J. Huang and D. J. K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, v.4 n.11:p.1123–1134, 2011.
10. M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. Thuraisingham. Heuristics based query processing for large rdf graphs using cloud computing. *IEEE TKDE*, v.23 n.9:p.1312–1327, September 2011.
11. H. Kim, P. Ravindra, and K. Anyanwu. Scan-sharing for optimizing rdf graph pattern matching on mapreduce. In *IEEE CLOUD*, pages 139–146, 2012.
12. J. Myung, J. Yeon, and S.-g. Lee. Sparql basic graph pattern processing with iterative mapreduce. In *Proc. of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC '10, pages 6:1–6:6, 2010.
13. T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, v.1 n.1:p.647–659, August 2008.
14. A. Pavlo, V. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. *In SIGMOD'12*, pages p.61–72, 2012.
15. J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. *In SIGMOD'02*, pages p.558–569, 2002.
16. S. Sanghavi, D. Shah, and A. S. Willsky. Message passing for maximum weight independent set. *IEEE Trans. on Information Theory*, 55(11):4822–4834, 2009.
17. K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *ISWC'03*, pages 131–150, 2003.
18. T. Yang, J. Chen, X. Wang, Y. Chen, and X. Du. Efficient sparql query evaluation via automatic data partitioning (technical report), 2012. `http://iir.ruc.edu.cn/~jchchen/rdfpartition.pdf`.