DEIM Forum 2020 H4-5

Hypergraph based Partitioning on a Distributed RDF System

XU MINGQIN†, Hieu Hanh LE††, and Haruo YOKOTA††

† Department of Computer Science, Graduate School of Information Science and Engineering
†† Tokyo Institute of Technology, 2-12-1 Ookayama, Meguro, Tokyo, 152-8552 Japan
E-mail: †{xu,hanhlh}@de.cs.titech.ac.jp, ††yokota@cs.titech.ac.jp

Abstract Due to the flexibility and scalability of RDF (Resource Description Framework) data model, more and more communities have released their data in RDF format. Growth of RDF data throws a challenge for data management and query processing in optimized time. Current distributed RDF systems involve large of inter-node communication. This paper will focus on hypergraph based spanning tree partitioning and distributed processing system in order to handle large-scale RDF data efficiently. Algorithm for transformation of RDF graph into hypergraph is proposed: all the subjects and objects connected with a particular predicate in the RDF graph resides under a particular hyper edge. Chunks are generated on the help of the hypergraph spanning trees and distributed into several nodes, which can decrease the SPARQL query response time. The system performance is evaluated using the LUBM benchmark.

Key words data structure, graph database, hypergraph, RDF

1 Introduction

The Resource Description Framework (RDF) is widely used as a versatile data model that provides a simple way to express facts in the semantic web. It uses triples composed of subjects, predicates and objects to store data [1]. Due to the flexibility and scalability of RDF data models, more and more communities have released their data in RDF format. With the rapid growth of RDF data on the Web, semantic Web applications turn to distributed system for help in pursuing better data management performance. Therefore, partitioning algorithm over RDF datasets in the distributed system has become a challenging issue. When applying partitioning algorithms developed over past decades to RDF data using well known RDF model such as Directed Labeled Graphs, Bipartite Graph, the vertices which a triple depends on may be in different partitions. Such partitioning on the RDF models induces huge communication overhead during processing queries.

In this work, a hypergraph based spanning tree partitioning in distributed RDF management system is proposed in order to handle large-scale RDF data efficiently. A hypergraph is a generalization of the graph where the edges connect more than two vertices and such edges are called hyper edges [2]. The system performance is evaluated using the LUBM benchmark [3].

Following are the main contributions of this paper:

1. A hypergraph RDF model is established. We take advantage of the hypergraph structure for representation to better reflect the various characteristics of RDF graphs and provide a theoretical basis for subsequent data partitioning.

2. Chunks are generated on the help of the hypergraph spanning trees and distributed into several nodes in order to ensure that the data load is relatively balanced and the probability of local joins are increased.

The remainder of this paper is organized as follows. Section 2 will give the background of RDF management system and RDF representing model. Section 3 is going to introduce the proposed methods. Section 4 is planned to describe and report the experiment results. Conclusion and future work will be given in section 5.

2 Background

Early researches in the field of RDF management system focus on efficient centralized RDF systems. However, centralized data management does not scale well for complex queries on the huge amount of RDF data. As a result, distributed RDF management systems were introduced by partitioning RDF data among many computing nodes and evaluating queries in a distributed manner. A SPARQL query [4] is decomposed into multiple subqueries that are evaluated by each node independently. Since the data is distributed, the nodes may need to exchange intermediate results during query evaluation. Therefore, queries with large intermediate results incur high communication cost, which is detrimental to the query performance [5], [6]. Distributed RDF systems aim at minimizing the number of decomposed subqueries by partitioning the data among workers. In such a

parallel query processing, each node contributes a partial result of the query; the final query result is the union of all partial results. The goal is that each node has all the data it needs to evaluate the entire query and there is no need for exchanging intermediate results [7]. To achieve this, there has been a lot of research issues such as query cost optimization. In this work, we focus on partitioning method and data presentation.

In terms of the data model types for representing RDF, current researches are divided into two groups: traditional database model and native graph model. the performance bottleneck of traditional database models is inevitable because of the extra cost for data model transformation. When accessing RDF data within a non-graph storage. many useful graph-based operations (e.g., random walk, reachability, community discovery) on RDF data are not supported. As for native graph model, it is not easy to execute the query because predicate represents a link between subject and object. A resource could be a subject, a predicate, and/or an object at the same time, which causes inconsistent representations for vertices and edges. In [8], Jonathan Hayes proposes a hypergraph representation. By letting a hyper edge be composed of three vertices corresponding to the elements of a triple, the hypergraph representation can not only support efficient traversals on the RDF graph, but also overcome the limitations discussed above [9].

As for partitioning method, it is used to speed up the task of query execution and RDF data processing as the RDF data may be very large. Metis is one of famous partitioning method [14]. It takes advantage of the fact that RDF uses a graph data model. This enables triples that are close to each other in the RDF graph to be stored on the same machine. the so-called k-hop guarantee that for any vertex v assigned to partition p, all vertices up to k-hops away and the corresponding edges are replicated in p. This way any query within radius k can be executed without communication.

3 Proposed Methodology

3.1 Formal Definition

Before describing the proposal, we briefly discuss formal definition of the basic terms used in this paper.

[Definition 1] RDF graph: a RDF graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ where $\boldsymbol{V} = \{\boldsymbol{v} | \boldsymbol{v} \in \boldsymbol{S} \cup \boldsymbol{O}\}$ and $\boldsymbol{E} = \{\boldsymbol{e_1}, \boldsymbol{e_2} \ldots\} \exists \boldsymbol{e} = \{\boldsymbol{u}, \boldsymbol{v}\}$ where $\boldsymbol{u}, \boldsymbol{v} \in \boldsymbol{V}$. There are two function $\boldsymbol{l_e}$ *and* $\boldsymbol{l_v}$. $\boldsymbol{l_e}$ is an edge-labeling function. $\boldsymbol{l_e}(\boldsymbol{S}, \boldsymbol{O}) = \mathbf{P}$ and $\boldsymbol{l_v}$ is the node labeling function. $\boldsymbol{l_v}(\boldsymbol{v_t}) = \boldsymbol{t}$ where $\boldsymbol{t} \in (\boldsymbol{S} \cup \boldsymbol{O})$ and $\boldsymbol{S} = \boldsymbol{Subject}(\boldsymbol{URI} \cup \boldsymbol{BLANKS})$, $\boldsymbol{P} = \boldsymbol{Predicate}(\boldsymbol{URI})$, $\boldsymbol{O} = \boldsymbol{Object}(\boldsymbol{URI} \cup \boldsymbol{BLANKS} \cup \boldsymbol{LIT})$.

[Definition 2] Hypergraph: a Hypergraph $\boldsymbol{HG}$ is defined as a set of vertices $\boldsymbol{V}$ and a set of nets (hyperedges) $\boldsymbol{E}$ among those vertices, Every net is a subset of vertices. $\boldsymbol{HG} = (\boldsymbol{V}, \boldsymbol{E})$ where node $\boldsymbol{V} = \{\boldsymbol{v_1}, \boldsymbol{v_2} \ldots \boldsymbol{v_n}\}$ and $\boldsymbol{E} = \{\boldsymbol{e_1}, \boldsymbol{e_2}, \ldots, \boldsymbol{e_n}\}$ where $\boldsymbol{V} = \{\boldsymbol{v} | \boldsymbol{v} \in \boldsymbol{S} \cup \boldsymbol{O} \cup \boldsymbol{P}\}$ and each edge $\boldsymbol{E}$ is a non-empty set of $\boldsymbol{V}$. The union of $\boldsymbol{V}$ is equal to $\boldsymbol{E} : \boldsymbol{E} = \cup_{i=1}^{n} \boldsymbol{V}$.

[Definition 3] Predicate based spanning Tree(PT): for a hypergraph $\boldsymbol{HG} = (\boldsymbol{V}, \boldsymbol{E})$, one of the PT of the $\boldsymbol{HG}$ is: $PT = (V_{PT}, \boldsymbol{E}_{PT})$ if the following conditions hold: (1)$V_{PT} \subseteq V, \boldsymbol{E}_{PT} \subseteq E$; (2) There is only one predicate in $\boldsymbol{E}_{PT}$, but it contains multiple subjects and objects. (3) $\forall \boldsymbol{E} \in \boldsymbol{E}_{PT}$, every v related to a hyperedge $\boldsymbol{E}$ has $v \in V_{PT}$.

[Definition 4] Hypergraph based Traversal Tree Partitioning: for a hypergraph $\boldsymbol{HG} = (\boldsymbol{V}, \boldsymbol{E})$, suppose $P_1, P_2, \ldots, P_k$ is the set of predicate spanning tree, which is in the size of $k\,(1 \leqq k \leqq |\boldsymbol{E}|, P_i = (v_{pi}, \boldsymbol{E}_{pi}), i = 1, 2, \ldots, k)$. If $V_{p_1} \cup V_{P2} \cup \cdots \cup V_{pk} = V, E_{p1} \cup \boldsymbol{E}_{p2} \cup \ldots \cup \boldsymbol{E}_{pk} = \boldsymbol{E}$, then $P_1, P_2, \ldots, P_k$ is the K-way partition of $\boldsymbol{HG}$ [10] .

3.2 RDF Hypergraph Model

We convert RDF graph into a hypergraph by calculating how many predicates which subjects and objects connected with. For the RDF hypergraph, all the subjects and objects connected with a particular predicate in the RDF graph resides under a particular hyper edge. Hypergraph expresses all the semantics of RDF data. Explicitly expressed semantics can be accessed by traversing the hypergraph. Figure 1 is a simple example of RDF graph, Figure 2 is the pictorial representation of data as a hypergraph that is generated from the Figure 1.

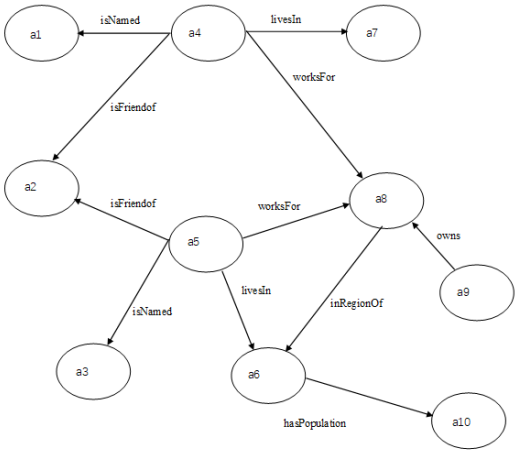

Figure 1 RDF graph

we use set and map data structure to implement the hypergraph transformation, which java could offer interfaces. And algorithm1 is used for building the map

3.3 Spanning Tree Based Chunk Generation

After converting RDF graph to hypergraph, the entire RDF hypergraph is supposed to be divided into several chunks for distribution. For the first step, RDF hypergraph

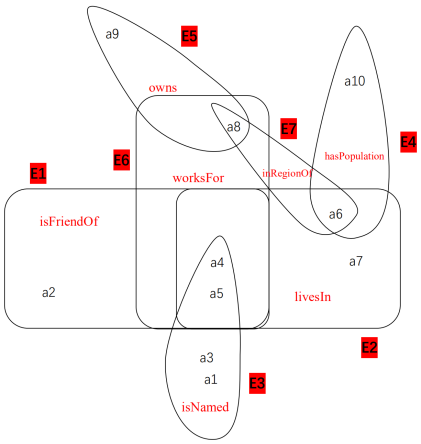

Figure 2 RDF hypergraph

Algorithm 1: Store predicate in a map

Input : Graph containing triples

Output: data: Map storing subjects objects pairs for each predicate

```
for ∅ map ;
for all a_j where 0 ≦ j ≦ a. length-1 do
    map → j + 1;
end
remove dumlicate entries from map;
Store subjects and objects pairs for each predicate;
for ∅ date ;
for all map_i where 0 ≦ i ≦ a. length-1 do
    for all a_j where ≦ j ≦ a. length-1 do
        if map_i = a_j{predicate} then
            date[map_i] → [map_i]∪a_j objective∪a_j subjective
        end
        j → j + 1
    end
    i → i + 1
end
```

is split into a set of spanning trees, which is regarded as the initial partitions. In RDF data, the number of predicates is relatively small. Consequently, predicates are used as the starting point for the spanning tree partitioning. Each predicate spanning tree contains only the data corresponding to the predicate. In order to generate spanning tree for one particular predicate, the algorithm 2 is deployed.

The spanning trees are the disjoint subgraphs of the entire RDF hypergraph. As the relations among each spanning trees are relatively sparse, the further partitioning is needed. In the next step, spanning tree whose size is more than chunk size threshold will be bipartitioned and then clustered together in order to generate chunks with equal size where contained vertices are closely related. Hmetis is used which is an extremely fast hyper graph partitioning algorithm [13]. he weight is redefined for this case. The system will check

Algorithm 2: spanning tree generation

Input : predicate, threshold

```
1  hyperedgeSet={}, subjectSet={}, objectSet={},
     predecessorSet={};
2  subjectSet = findSubfectSetByPredicate (predicate);
3  divide subjectSet into smaller blockSet;
4  parfor each subjectBlock in blockSet do
5      mark − unvisited − to − current − thread(subjectBlock,
         visit array);
6      while true do
7          for each unvisited subject in subjectBlock do
8              objectSet=
                 fjndObjectSetByPredicateAndSubject(predicate,
                 subject);
9              id = chooseSlave(subject, objectSet.size(),
                 threshold, predecessorSet);
10             if id= −1 then
11                 divide objectSet into k subObjectSet;
12                 for each subObjectSet[i] in subObjectSet do
13                     place triples <subject, predicate,
                         subObjectSet[i]> in slave i;
14                 end
15             else
16                 place triples < subject, predicate,
                     objectSet > in slave id
17             end
18             for each unvisited object in objectSet do
19                 hyperedgeSet =hyperedgeSet∪object
20             end
21             mark-unvisited-to-current-thread(objectSet,
                 visit array);
22             if hyperedgeSet == 0 then
23                 break;
24             end
25         end
26     end
27     predecessorSet. swap(subjectBlock);
28     subjectBlock.swap(hyperedgeSet);
29     hyperedgeSet.clear();
30     repeat
31     if blockSet is empty and exist busy thread(s) then
32         threadid = choose-busy-thread();
33         steal-and-execute-rasks(threadid);
34     end
35 end
```

the overlap of the hyper edges to identify the weight. For a spanning tree with hyper edge e_i, it has $w(e_i)$ to measure the strength of the cohesion between the vertices v_i in e_i.

The support of e_i is:

$$\mathrm{S}\,(e_i) = |\mathrm{V}\,(e_i)|$$

There are other hyper edges in e_i because the vertices in e_i can be connected by the way except hyper edge e_i. If there exist hyper edge X, Y, and there is a relationship r that $X \rightarrow Y$, the confidence of r is :

$$\text{con}\, f(r) = S(X \cup Y)/S(X)$$

If there are k relations in e_i, so the mean confidence of e_i is :

$$\text{con}\, f\,(e_i) = \frac{\sum_{j=1}^{k} \text{con}\, f\,(r_i)}{k}$$

In the traditional definition, mean confidence of the hyper edge is the weight. In this work, the weight is not only considered as the cohesion between the vertices v_i in e_i but also connection among other partitions. For example, we suppose there is another spanning tree with hyper edge e_j, The weight of e_i is affected by e_j, if e_i overlaps e_j.

The weight is defined as:

$$w\,(e_i) = \text{con}\, f\,(e_i) - \sum\nolimits_{j=1, j\neq i}^{t} \left[\frac{|V\,(e_j) \cap V\,(e_i)]}{|V\,(e_j)|} \text{con}\, f\,(e_j) \right]$$

Hmetis will find the hyper edge with the smallest weight and truncate it to divide the hypergraph into two parts. This algorithm is repeated until the final clustering result is n chunks. All vertices contained in each chunk are closely linked.

3.4 Choosing Computing Nodes for Chunks

After chunk generation, chunk placement algorithm is designed for meaningful distribution in each computing nodes. From the careful analysis of the SPARQL query patterns, we observed that the large involvement of s-s, s-o and o-o joins. Therefore, when distributing the RDF graph, it is necessary to maintain the above relationships as much as possible. Algorithm 3 is deployed. The system firstly checks the size of chunk to exclude the computing nodes without enough space. The predecessor of the chunk is also checked to exclude the computing nodes without predecessor. In addition, there is a global mapping table that is constantly updated and maintained to record the distribution of hyper edges on each computing node.

Algorithm 3: choosing computing nodes

```
Input : hyperedge, blocksize, threshold, predecessorSet,
        overlapping_table
1  old_val = slave_table [hyperedge];
2  if blocksize > threshold then
3      return -1;
4      slaveid = slave_table [hyperedge];
5  end
6  if slaveid != 0 then
7      return slaveid;
8  else
9      predecessor = getOnePredecessor(hyperedge,
         predecessorSet);
10     if predecessor != -1 then
11         slaveid = slave_table [predecessor];
12     else
13         slaveid = chooseMaximaloverlapping
             (overlapping_table);
14     end
15     if _sync_bool_compare_and_swap(slave_table
         [hyperedge], old_val, slaveid) then
16         return slaveid;
17     else
18         return slave_table[hyperedge];
19     end
20 end
```

4 Experiment

In this section, we evaluate the performance of our algorithm both in terms of dataset loading time and query processing time. We use LUBM benchmark [11]. By leveraging the data generator UBA (Univ-Bench Artificial data generator), we generate three data sets that contain OWL files describing information of 1, 10 and 50 universities respectively. The three data sets are named LUBM(1, 0), LUBM(10,0), and LUBM(50, 0). The detail is shown in Table 1.

A prototype system is developed which contains 1 coordinator, 5 computing nodes(workers). In addition, Jena, which is a java framework, is applied for loading the data sets and providing logical plan in query processing [12]. We also construct a parallel data processing framework using HadoopDB (version 0.1.1), a hybrid of MapReduce and DBMS Technologies [15]. HadoopDB connects multiple single-node database systems using Hadoop as the task coordinator and network communication layer. Queries are parallelized across nodes using the MapReduce framework. We contrast the system with hypergraph model in proposed partitioning method and system with RDF native graph model in metis(2-hop) partitioning.

Firstly, we provide experimental study for total preprocessing time which includes data creation time, partitioning time and loading time. Although proposed method takes longer time because of the extra cost of hypergraph creation, it is acceptable because hypergraph only needs to be created once. Besides, if comparing two methods without considering the data creation time, the proposal is better in dealing large amount of data. The Figure 3 shows the statistics.

Secondly, we provide experimental study for query processing time. Due to Hadoop usually has start-up time, only the time of Map and Reduce phase is measured and the start-

Table 2 Query Response Time on LUBM(50,0)

	Q1(star) high selectivity		Q2(star)		Q3(long circle)		Q4(simple circle) high selectivity		Q5(long chain)		Q6(circle+chain)		Q7(circle+chain)	
time (s)	res	exe	res	exe	res	exe	res	exe	res	exe	res	exe	res	exe
proposal	0.7	0.075	0.4	0.031	110.3	0.276	224.7	0.452	113.2	2.22	324.7	3.432	672.1	9.54
Metis(2-hop)	0.1	0.067	0.377	0.032	167.16	0.382	219.7	0.392	181.7	4.51	741.2	19.56	989.2	9.156

res: processing time in communication module
exe: execution time in storage module

Table 1 the Detail Information of LUBM

	LUBM(1,0)	LUBM(5,0)	LUBM(10,0)
Total size (MB)	8.6	54.2	110.6
Number of triple	103,397	646,128	1,316,993

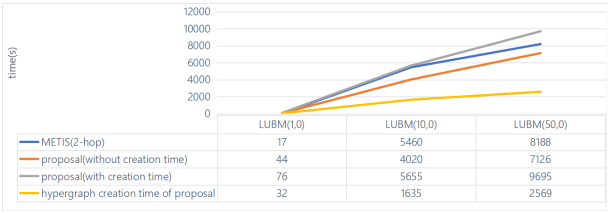

Figure 3 Preprocessing Time of the Proposal and Metis(2-hop)

up time is removed in the experiments. Each query result is divided into two parts: execution time in storage module and processing time in communication module. In storage module, Jena executes queries. Intermediate results of each database system running on each node are sent to communication module (Hadoop layer). Hadoop layer will execute the rest of tasks, such as join on intermediate results. All statistics presented in this work are the averages of 5 runs of the queries. Table 2 shows the query performance over proposed method and Metis. 7 queries are conducted. Those queries are classified based on their structures, selectivity and complexity. Q1 and Q2 are simple star queries while Q1 has high selectivity. Q3 is simple nonselective long circle query. Q4 is highly selective and composed of simple circle. Q5, Q6 and Q7 are complex queries with large intermediate results.

5 Conclusions and Future Work

In general, this work used a hypergraph model for RDF partitioning, which intends to be more concrete than the native graph model to allow more meaningful partitions. Besides, a partitioning method on the help of the hypergraph is proposed which is evaluated comparing with Metis.

Specifically, the system first establishes a hypergraph model of RDF data, and then constructs chunks based on hyper edge spanning tree, continuously divides and places the chunks along the spanning tree. The experimental results on preprocessing time over LUBM show that proposed system can handle large-scale RDF data. And the proposal shows fast partitioning speed and loading speed but suffer in extra cost of hypergraph creation. Moreover, proposed method is compared with Metis on query performance, the first observation is that the query evaluation in database layer changes negligibly for these two methods. The times spending in Hadoop layer are far larger than the query evaluation times in database layer. And proposal shows better performance on chain queries and complicated queries (especially with circle joins) and suffer in high selectivity queries. On the other hand, Metis is better in star queries.

For the proposed system, there is still much room to improve. 1) The join relationship between the entities is given priority when partitioning data. However, there may exist relations among predicates. These relations should be taken into consideration. 2) The system only performs distributed storage and management on the static data. Dynamic adjustment should be considered. 3) Experiments on larger datasets and larger computing nodes are supposed to be conducted.

References

[1] Resource Description Framework (RDF): Concepts and Abstract Syntax W3C, 2004, https://www.w3.org/TR/rdf-concepts/

[2] Hypergraph, https://en.wikipedia.org/wiki/Hypergraph

[3] LUBM. http://swat.cse.lehigh.edu/projects/lubm/.

[4] SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/.

[5] W3C Data Activity Building the Web of Data, 2013, https://www.w3.org/2013/data/

[6] Billion Triple Challenge.http://challenge.semanticweb.org/.

[7] Z. Ma, M. A. Capretz, and L. Yan. Storing MassiveResource Description Framework (RDF) data: a Survey. The Knowledge Engineering Review, 31(4):391-413, 2016.

[8] Hayes J. A graph model for RDF [Master's Thesis]. Department of Computer Science, Technische Universitat Darmstadt, Germany, August 2004

[9] Wu G, Li JZ, Hu JQ et al. System: A native RDF repository based on the hypergraph representation for RDF data model. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 24(4): 652-664 July 2009

[10] Aykanat C, Cambazoglu B B, Ucar B. Multi- level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. Journal of Parallel and Distributed Computing. 68 (5), 2008. 609-625

[11] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics,3(2-3):158-182, 2005.

[12] Jena. http://jena.sourceforge.net.

[13] Hmetis. http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview

[14] George Karypis and V Kumar. Metis-a software package for partitioning unstructured graphs, meshes, and computing fill-reducing orderings of sparse matrices-version 5.0. University of Minnesota, 2011.

[15] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silber-

schatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proc. of the VLDB 2009, 2(1):922-C933, 2009.