Partitioning RDF Exploiting Workload Information

Rebeca Schroeder
Univ. Federal do Paraná
Curitiba, PR, Brazil
rebecas@inf.ufpr.br

Raqueline Penteado
Univ. Federal do Paraná
Curitiba, PR, Brazil
rrmpenteado@inf.ufpr.br

Carmem S. Hara
Univ. Federal do Paraná
Curitiba, PR, Brazil
carmem@inf.ufpr.br

ABSTRACT

One approach to leverage scalable systems for RDF management is partitioning large datasets across distributed servers. In this paper we consider workload data, given in the form of query patterns and their frequencies, for determining how to partition RDF datasets. Our experimental study shows that our workload-aware method is an effective way to cluster related data and provides better query response times compared to an elementary fragmentation method.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Dist. databases*

Keywords

RDF; data fragmentation

1. INTRODUCTION

An increasing and expressive amount of data are becoming available in RDF format. DBpedia had extracted 1.89 billion RDF triples from Wikipedia last year, and the Linked Open Data project[4] had collected over 20 billion triples from 295 interlinked datasets on the Web. Not surprisingly, the massive scale of data overwhelms computing resources on commodity infrastructures. A scalable distributed system is required to store and manage these large datasets. However, scalable processing of large RDF graphs also requires careful partitioning of data across servers. Otherwise, the system performance may decrease when related data is spread over large geographic distances.

RDF datasets have been fragmented in different ways. Simple hash functions have been applied on RDF triples to cluster them by subject or property[2]. As an example, consider the RDF graph in Figure 1 extracted from the Berlin SPARQL Benchmark[1]. In this example, triples related to product entities may be joined by a hash function on the subject (*<product1, label, ProductXYZ>*, *<product1, offer, offer1>* and so on). An alternative approach is to apply workload data in order to keep properties accessed together in the same fragment [3]. For instance, the properties *label*, *producer* and *feature* are candidates to be placed in the same fragment according to the query $Q2$ defined by the Berlin workload and given in Figure 2. In this paper we show that a workload-based fragmentation is an effective way to improve the performance of a distributed RDF datastore, compared to an elementary approach like hash partitioning.

Copyright is held by the author/owner(s).
WWW 2013 Companion, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2038-2/13/05.

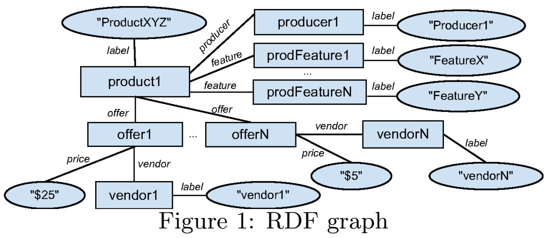


Figure 1: RDF graph

```
Q2-Basic information about a product  Q7-Extra information about a product
------------------------------------  ------------------------------------
SELECT ?a ?b ?c                       SELECT ?a ?b ?c ?d
WHERE { %ProductXYZ% rdfs:label ?a .  WHERE { %ProductXYZ% rdfs:label ?a .
%ProductXYZ% bsbm:producer ?p .       %ProductXYZ% bsbm:feature ?f .
?p rdfs:label ?b .                    ?f rdfs:label ?b . ?offer bsbm:
%ProductXYZ% bsbm:feature ?f .        product %ProductXYZ% . ?offer bsbm:
?f rdfs:label ?x}                     price ?c . ?offer bsbm:vendor
                                      ?vendor . ?vendor rdfs:label ?d }
```

Figure 2: Queries from the Berlin SPARQL Benchmark

We have applied *xAFFrag*[5], a workload-aware algorithm, which has originally been proposed for partitioning XML schemas. Section 2 presents the partitioning process on RDF. Results of applying *xAFFrag* are compared to the native method of the Berlin Benchmark [1] in Section 3. We conclude in Section 4 with some remarks for future work.

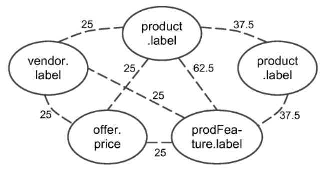
2. WORKLOAD-AWARE PARTITIONING

The goal of the *xAFFrag* algorithm is to minimize the execution of distributed queries, by keeping the most correlated data in the same fragment. To this end, we need to measure the correlations among data according to a given workload. In order to exemplify the workload characterization process applied by [5], consider queries $Q2$ and $Q7$ as the representative workload for the RDF graph of Figure 1. The workload may be initially represented as a usage matrix as depicted in Figure 3a. In the matrix, each query is represented as a column, and each data item as a row. In addition, the expected frequency of each query is represented in the last row. If a given query q traverses a data item d, then M [q, d] = 1; otherwise, M [q, d] = 0. In the example, $Q2$ is executed 37.5 times in a given period of time involving items *product.label*, *producer.label* and *prodFeature.label*.

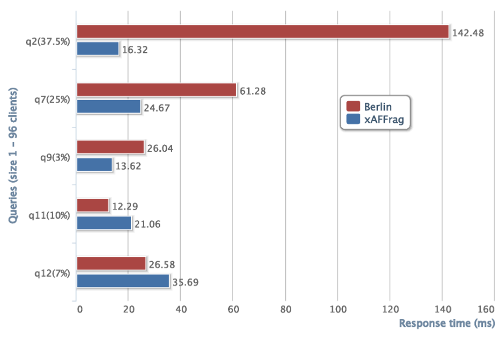
Each data item is represented as a node in an affinity graph (Figure 3b), and the affinity measure between two nodes n_i and n_j is defined by the sum of the query frequencies that involve both n_i and n_j. As an example, the

Data item	*Q2*	*Q7*
product.label	1	1
producer.label	1	0
prodFeature.label	1	1
offer.price	0	1
vendor.label	0	1
Frequency	37.5	25

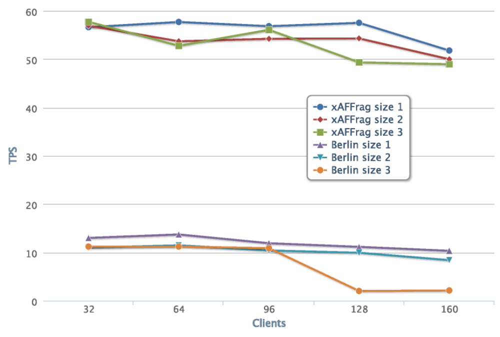
(a) Usage Matrix



(b) Affinity graph



(c) Query Performance



(d) System Throughput

Figure 3: Workload data and Experimental Results

affinity between *product.label* and *prodFeature.label* consists of the sum of frequencies of queries $Q2$ and $Q7$ (62.5). The affinity values are used to label edges in the affinity graph.

xAFFrag takes an affinity graph and a storage threshold to find a fragmentation schema. As an example, assume that the storage threshold is set such that each fragment can keep up to 3 nodes. In this case, the fragmentation schema generated by *xAFFrag* is represented by two fragments: $f_1 = \{product.label, producer.label, prodFeature.label\}$ and $f_2 = \{offer.price, vendor.label\}$. Notice that the size of fragments fits in the storage threshold and the affinity between any node in f_1 with any node in f_2 is lower than the affinity between any pair of nodes in the same fragment. Thus, the most related data items are placed in the same fragment.

3. EXPERIMENTAL EVALUATION

In order to measure the effect of our proposed method for RDF partitioning, we compare *xAFFrag* with the native fragmentation of Berlin. The workload data has been extracted from the benchmark for a representative subset of 5 SPARQL queries and given as input to the workload-aware process. The native partition method fragments the dataset in a user-defined number of files. Thus, data related to a specific product may be in a single file or distributed among a set of files. Berlin supports the creation of arbitrarily large datasets using the number of products as scale factor. We generated datasets with 50, 100 and 150 products. They are referred to as datasets of size 1, 2 and 3, respectively. Berlin and *xAFFrag* partition each product in 9 fragments given that the threshold considered for the fragment size is 11 kB. The datasets are stored in Scalaris [6], a key-value store running on 16 Amazon EC2 micro instances. Thus, the SPARQL queries are directly mapped to Scalaris requests.

In order to determine the impact of the 2 fragmentation approaches, queries were executed with increasing number of clients and dataset sizes. The results in terms of query response time are depicted in Figure 3c. They present the average value of 300 executions for a dataset with size 1 and 96 clients. The query mix is given beside each query name and consists of the percentage of queries of this type in the set of queries of each run. The results reported for *xAFFrag* and Berlin can be explained by the number of fragments required to process the queries. For executing $Q2$ on *xAFFrag*, we need only 2 accesses to the datastore, which takes 16.32 ms. The execution of the same query on Berlin increases the number of requests to 9 fragments and has the same effect on the response time (142.48 ms). In other words, the affinity-based approach clusters data required by the most frequent query ($Q2$) in 2 fragment, while Berlin spread the related data in 9 fragments. With the increase on the number of fragments, the number of requests to process a query is also likely to be larger, which reflects on the results. Notice that Berlin presents better performance for $Q11$ and $Q12$. This is because they are the most infrequent ones and the *xAFFrag* approach favors the performance of frequent queries.

Both approaches present a similar degradation on the throughput when the database size is increased as shown in Figure 3d. With respect to the number of clients, we observed a degradation when the number of clients is between 96 and 160. We believe that with higher number of requests the competition for resources increases, leading to CPU overhead and query conflicts, which impacts the system performance.

4. CONCLUSIONS

We have demonstrated that an algorithm originally proposed to fragment XML can be effectively applied for partitioning RDF. The experimental results show that a workload-aware method for partitioning RDF is effective to improve the performance of a distributed datastore, compared to an approach that ignores workload data. We are currently working on the extension of *xAFFrag*, which includes the adjustments performed in this work to support RDF. The main issue related to this topic involves processing cycles in RDF graphs. Future works include comparative studies with other partitioning approaches, as well as replication, query load balancing and dynamic workloads.

5. REFERENCES

[1] C. Bizer and A. Schultz. The berlin sparql benchmark. *IJSWIS*, 5(2):1–24, 2009.

[2] M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW '04*, pages 650–657, 2004.

[3] L. Galarraga, K. Hose, and R. Schenkel. Partout: A distributed engine for efficient rdf processing. *CoRR*, abs/1212.5636, 2012.

[4] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan and Claypool, 2011.

[5] R. Schroeder, R. Mello, and C. S. Hara. Affinity-based xml fragmentation. In *WebDB*, pages 61–66, 2012.

[6] Zuse Institute Berlin. Scalaris-distributed transactional key-value store. Available at http://code.google.com/p/scalaris/, 2013.