Partitioning Templates for RDF

Rebeca Schroeder[1] and Carmem S. Hara[2]

[1]Universidade do Estado de Santa Catarina - UDESC, Joinville-SC, Brazil, 89.219-710
[2]Universidade Federal do Paraná - UFPR, Curitiba-PR, Brazil, 81531-990
[1]rebeca.schroeder@udesc.br, [2]carmem@inf.ufpr.br

Abstract. In this paper, we present an RDF data distribution approach which overcomes the shortcomings of the current solutions in order to scale RDF storage both with the volume of data and query requests. We apply a workload-aware method that identifies frequent patterns accessed by queries in order to keep related data in the same partition. In order to avoid exhaustive analysis on large datasets, a summarized view of the datasets is considered to deploy our reasoning through partitioning templates for data items in an RDF structure. An experimental study shows that our method scales well and is effective to improve the overall performance by decreasing the amount of message passing among servers, compared to alternative data distribution approaches for RDF.

1 Introduction

We have witnessed an ever-increasing amount of RDF data made available in different application domains. The DBpedia dataset[1] has now reached a size of 2.46 billion RDF triples extracted from Wikipedia. According to the W3C, some commercial datasets may be even bigger reaching the score of 1 trillion triples[2]. The envisioned architecture to manage these huge datasets is based on elastic cloud-based datastores supported by parallel techniques for querying massive amounts of data [5]. In order to scale RDF storage, datasets must be partitioned across multiple commodity servers. By placing partitions on different servers, it is possible to speedup query processing when each server can scan its partitions in parallel. On the other hand, message passing among servers can be required at query time when related data is spread among arbitrary partitions. These rounds of communication over the network can become a performance bottleneck, leading to high query latencies. Therefore, the scalability of query processing depends on how data is partitioned or replicated across multiple servers.

RDF data are represented by triples given by `subject-predicate-object (s, p, o)` statements. In an RDF dataset, triples are related to each other representing a graph. Thus, the RDF partitioning problem has been addressed as a graph cut problem [5], [15]. Likewise the general problem, partitioning a distributed database is known to be NP-hard[8] and, therefore, heuristic-based approaches become more attractive. In general, the heuristics applied by current methods are solely based on the RDF graph structure, generating partitions that do not express query patterns of the

[1] http://wiki.dbpedia.org/Datasets
[2] http://www.w3.org/wiki/LargeTripleStores

workload. As result, the query performance decreases when data required by the same query pattern is distributed over different servers. Besides the workload-oblivious reasoning, most of the current approaches apply a graph partitioner algorithm on the whole RDF graph. However, large graphs are hard to partition.

In this paper, we introduce a data partitioning approach which overcomes the shortcomings of current solutions by reasoning over a set of query patterns assumed as the expected workload. The contribution of this approach is twofold. First, partitions are extracted from clusters of data accessed together by frequent query patterns. Such coverage of query patterns provides scalability for query processing by reducing the amount of message passing among machines at query time. Second, we are able to define how data items must be clustered solely based on the structure of query patterns. The query patterns are formulated over a summarization schema that represents the data structures for an RDF dataset. Thus, we define partitioning *templates* as the partitioning strategy to be applied to instances of an RDF structure. By doing so, we avoid exhaustive analyses on the whole data graph for defining data partitioning.

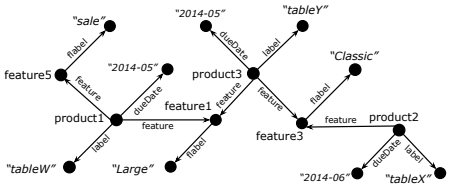
Despite the fact that most RDF datasets are schema-free, the lack of a schema makes it harder to formulate queries on RDF graphs and define suitable strategies for indexing and clustering. In fact RDF datasets range from structured data (e.g DBLP) to unstructured data (e.g. Wikipedia). However, there is a bit of regularity in RDF data[9] and it is relatively easy to recover large part of the implicit class structure underlying data stored in RDF triples as demonstrated in [7]. In our approach, RDF structures are applied to identify the query patterns in order to partition datasets. By following such a workload-agnostic approach, we are able to efficiently handle the most frequent queries. Likewise in traditional design approaches and the so-called 20-80 rule, we favor the important 20% of queries which corresponds to 80% of the total database load.

The rest of the paper is organized as follows. Section 2 introduces the partitioning problem. Our workload characterization method is presented in Section 3. In Sections 4 and 5, we describe our partitioning method involving data fragmentation and allocation. In Section 6, we experimentally investigate the impact of our method and compare to related approach. We discuss related work in Section 7 and conclude in Section 8.

2 Preliminaries and Partitioning Objective

RDF data can be defined as a finite set of triples composed of `subject`, `property` and `object` (`s`, `p`, `o`). Assume there are pairwise disjoint infinite sets $\mathcal{U}$ and $\mathcal{L}$, where $\mathcal{U}$ are URIs denoting Web resources, and $\mathcal{L}$ are literals. Thus, an RDF triple (`s`, `p`, `o`) $\in (\mathcal{U} \times \mathcal{U} \times \{\mathcal{U} \cup \mathcal{L}\})$. RDF follows a data model in which triples are related to each other, which can be represented as a directed graph. We denote an RDF graph as D. That is, D is a set of triples which denote facts where the subject is the origin node of a property labelled edge directed to its object node. As an example, the subject `product1` is related to the object `feature1` through the property `feature` in Figure 1a.

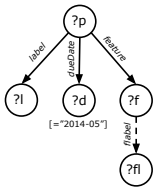
SPARQL is the W3C Recommendation language for querying RDF datasets. The SPARQL core syntax is based on a set of triple patterns like RDF triples except that subjects, properties and objects may be defined as variables. In our work, pattern graphs represent the conjunctive fragment of SPARQL queries. We assume the existence of a



(a) RDF Graph

```
SELECT ?l ?fl
WHERE {
  ?p label ?l .
  ?p dueDate ?d
  ?p feature ?f .
  FILTER (?d = "2014-05") .
  OPTIONAL {
    ?f flabel ?fl .
  }
}
```

(b) SPARQL Query



(c)

Fig. 1: RDF Graph and a SPARQL Query Example

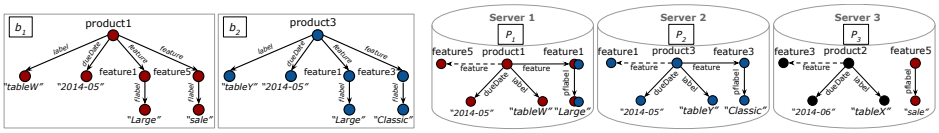
set $\mathcal{V}$ of variables that is disjoint of the sets $\mathcal{U}$ and $\mathcal{L}$. Variables in $\mathcal{V}$ are denoted by a question mark (?) prefix.

Definition 1. *(Pattern Graph): A pattern graph is denoted by $G = (V, E, r)$ where: (1) $V \subseteq \{\mathcal{V} \cup \mathcal{U} \cup \mathcal{L}\}$; (2) $E \subseteq (V \times \mathcal{U} \times V)$, where for each edge ($\hat{s}$,$\hat{p}$,$\hat{o}$) $\in E$, $\hat{s}$ is the source of the edge, $\hat{p}$ is the property, $\hat{o}$ is the target of the edge; and (3) r is a set of filter expressions for variable nodes in G. A filter is expressed in the form $?x\ \theta\ c$, where $?x \in \mathcal{V}$, $c \in \{\mathcal{U} \cup \mathcal{L}\}$ and $\theta \in \{=, >, \leqslant, <, \geqslant\}$. Hereafter, we use $V(G)$ and $E(G)$ to denote the set of vertices and the set of edges of a pattern graph, respectively.*

An example of pattern graph is given in Figure 1c where variable nodes are annotated with the associated filter expressions. The conjunctive fragment of SPARQL queries involving operators AND, FILTER, OPTIONAL and UNION can be represented as graph patterns as follows. Pattern triples are represented by connected nodes denoting operators *AND* (solid edges) and *OPTIONAL* (dashed edges). To simplify, we represent pattern graphs connected by the *UNION* operator as independent graphs. Figure 1b shows a SPARQL query that retrieves data for products and features associated with product where the *dueDate* is "2014-05". The equivalent representation for the pattern graph is shown in Figure 1c. Observe that although in the example the query is represented as a tree, cycles are admitted by the pattern graph definition.

The workload is defined as pattern graphs representing a set of SPARQL queries Q. Given that SPARQL is a graph-matching language, processing a query against RDF graphs consists of a subgraph matching problem which can be computed by graph homomorphism[17]. The subgraphs shown in Figure 2a correspond to matches of the pattern graph of Figure 1c applied to the RDF graph in Figure 1a. We use $B(q) = \{b_1, ..., b_n\}$ to denote the result of a query q, where b_i is a subgraph of an RDF graph D, i.e., $b_i \subseteq D$.

Consider now processing the same query over a partitioned dataset. Figure 2b illustrates the graph in Figure 1a partitioned across 3 server. When the query is issued, it is processed in parallel in all servers. Ideally, each subgraph in a result should be stored in a single server. However, in our example, subgraphs b_1 and b_2 are segmented across two servers. Retrieving b_1 requires *Server1* and *Server3* to be accessed, while *Server1* and *Server2* are needed to retrieve b_2. In order to avoid this message passing among servers, the main goal of our approach is to partition data so that query can be processed in parallel without inter-server communication whenever it is possible. More formally, we



(a) SPARQL Query Results (b) Query Segmentation on Partitioned Datasets

Fig. 2: SPARQL Query Results on Partitioned Data

are interested in generating a partitioning $\mathcal{P} = \{P_1, ..., P_m\}$, for an RDF graph denoted by D across m servers, where the amount of partitions required to retrieve each subgraph in a query result $B(q)$ is minimized. To this end, we define the segmentation of the subgraphs in $B(q)$ with respect to a partitioning $\mathcal{P}$ and a query q as follows :

Definition 2. *(Query Segmentation): Given a partitioning $\mathcal{P}$ of an RDF graph D, the query segmentation measure $\hat{P}$ of $\mathcal{P}$ with respect to q is defined as:*

$$\hat{P}(q, \mathcal{P}) = \Big|\{(b, P) \in (B(q) \times \mathcal{P}) | b \cap P \neq \oslash\}\Big| - \Big|B(q)\Big| \tag{1}$$

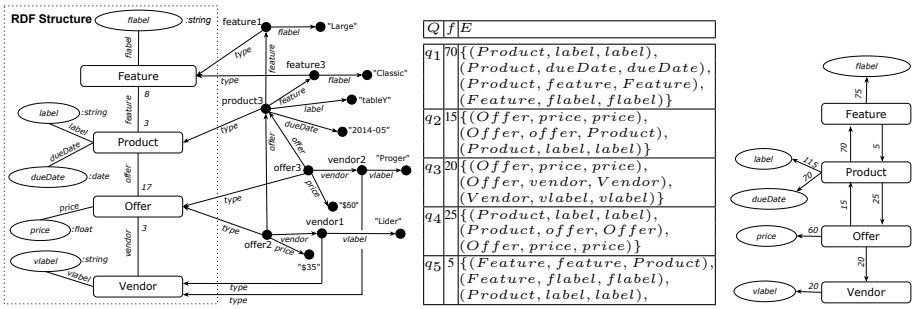
In this equation, the minuend determines how many partitions (or servers) have to be accessed to retrieve all triples in each subgraph result. That is, given a subgraph result $b \in B(q)$ and a partition P, a pair (b, P) is in the minuend set whenever P contains a triple in b. Ideally, no subgraph should be segmented. That is, the size of the minuend should be equal to the number of subgraphs in the result $B(q)$, which leads to $\hat{P} = 0$. Intuitively, $\hat{P}$ measures the amount of inter-server communication to compute a query result. Given that a workload consists not only of a single query, but a set of queries Q, the overall objective of our partitioning strategy is to minimize $\hat{P}$ for the set Q. To this end, we assume that each query q in the set is associated with its expected frequency in a period of time, which is denoted by $f(q)$. Thus, we can formally define our problem as to find a partitioning $\mathcal{P}$ that minimizes the following equation:

$$min \sum_{q \in Q} f(q).\hat{P}(q, \mathcal{P}) \tag{2}$$

Observe that frequent queries have a higher impact on the equation than infrequent ones. Intuitively, our strategy is based on favoring the most frequent queries in the workload. To achieve our goal, we characterize the workload for examining the paths traversed by the queries and their frequencies in order to quantify the affinity between pairs of nodes. Such affinity measure is the basis for our partitioning reasoning.

3 Workload Characterization

In this section we present a method for representing workload information. The core of this method is based on identifying and measuring affinity relations among RDF nodes. We start by defining an RDF Structure, containing both the structure of the RDF graph



(a) RDF Structure (b) Usage Matrix (c) Affinity graph

Fig. 3: Workload data

and the expected size of its instances. Although RDF can define a schema-free model, in general an RDF graph represents both schema and instances. Most datasets define the `type` property connecting entities to their respective classes. In Figure 3a, the RDF Structure is illustrated in the dashed shape containing classes as well as relationships among them. An RDF Structure is an undirected cyclic graph defined as a 6-tuple $S = (C, L, l, A, s, o)$, where (1) C is a set of labelled nodes representing RDF classes; (2) L is a set of labelled nodes denoting class properties with literal values; (3) l assigns a data type to each node in L; (4) A is a set of undirected edges $(n_1, n_2) \in (C \times \{C \cup L\})$ which corresponds to associations between nodes; (5) s is a function that assigns the expected size for the instances of nodes in $\{C \cup L\}$; and (6) o gives the expected cardinality of associations between two nodes; that is, it is a function that maps a pair in $(C \times \{C \cup L\})$ to an integer that defines for each node $n_1 \in C$ the expected number of occurrences of associations to a node $n_2 \in \{C \cup L\}$.

Figure 3a shows an RDF Structure. In the example, $o(\texttt{Product}, \texttt{Feature}) = 8$ because the average number of occurrences of `Feature` associated to an instance of `Product` is 8. Similarly, an instance of `Feature` is related to 3 instances of `Product` in average. That is, $o(\texttt{Feature}, \texttt{Product}) = 3$. Besides, there are multi-valued relationships between $(\texttt{Product}, \texttt{Offer})$ and $(\texttt{Vendor}, \texttt{Offer})$. We assume that for the remaining associations relating any other nodes n_1 and n_2 in the example, $o(n_1, n_2) = 1$. The size of a node n is not depicted in the example. If n is a literal node, $s(n)$ is the number of bytes needed for storing its value. For class nodes, on the other hand, the size corresponds to the size required to store their property structures. To simplify the example, we consider that for any node n, $s(n) = 1$.

Given a representation of an RDF Structure, we now turn to the workload characterization. We define a workload as a set of queries Q represented as pattern graphs and a function f that defines the expected frequency of each query in Q. The workload can be represented as a usage matrix as depicted in Figure 3b. According to the example, q_1 is expected to be executed 70 times and involves the literal nodes `label, dueDate, flabel` and the classes `Product` and `Feature`.

Given a workload on an RDF Structure, the affinity of two nodes n_i and n_j in an RDF Structure as the frequency they are accessed together by any query in the work-

load. Towards this goal, an affinity function $aff(n_i, n_j)$ takes as input a set of queries Q and computes the sum of frequencies of queries that involve both n_i and n_j by a path in a specific direction, i.e., n_i is the source node and n_j is the target node. More formally, we define $Q_{ij} = \{q \in Q \mid (n_i, p_{ij}, n_j) \in q\}$, and $aff(n_i, n_j) = \sum f(q)$, $q \in Q_{ij}$. As an example, consider the workload given in Figure 3b. The affinity between *Product* and *label* consists of the sum of frequencies of queries q_1, q_2, q_4 and q_5. Thus, $aff(\texttt{Product}, \texttt{label}) = f(q_1) + f(q_2) + f(q_4) + f(q_5) = 115$. The affinity function can be used to label edges in a directed graph involving all nodes in an RDF Structure, as depicted in Figure 3c. We refer to this graph as an affinity graph, which is defined as a tuple $\mathcal{A} = (N, \hat{E}, \texttt{aff})$, where N is the set of nodes in the RDF Structure and $\hat{E}$ is a set of edges which relates two nodes n_i and n_j by an affinity value ($\texttt{aff}(n_i, n_j)$).

We present our partitioning technique in two steps. The first consists of data fragmentation. That is, determining how to cut an RDF Structure in order to keep closely related data by affinity relations in a storage unit. The second concerns data clustering thus, it relates to the problem of allocating related fragments in the same server.

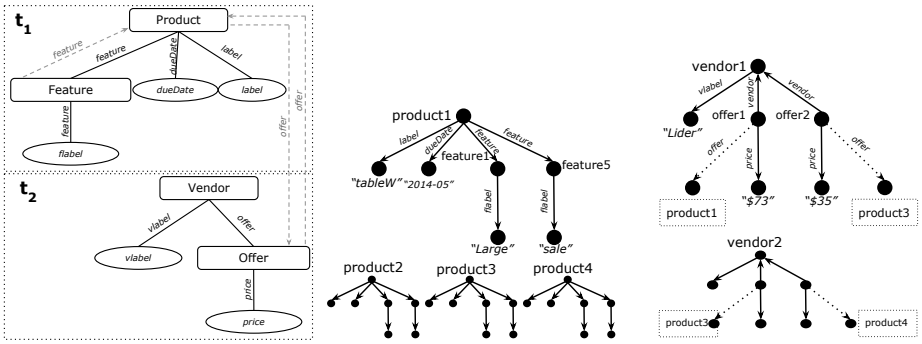
4 RDF Fragmentation

Distributed query processing performance is not only affected by the amount of message passing, but also by the size of the messages. A suitable size for messages motivated us to adopt a storage threshold as the basis for our partitioning technique. We refer to this storage threshold as Γ. Intuitively, our goal is to partition nodes of an RDF Structure, such that partitions contain as many correlated nodes as possible that can fit in a given storage size. In what follows, we introduce the RDF fragmentation problem and our proposal for solving it.

Given an RDF Structure $S = (C, L, l, A, s, o)$ and an affinity graph $\mathcal{A} = (N, \hat{E}, aff)$, we are interested in obtaining a fragmentation template $T = \{t_1, ..., t_m\}$, $m \geq 1$, such that t_i is a subgraph of S, $\bigcup_{i=1}^{m}(t_i) = (N, E')$, where $E' \subseteq E$ and each t_i is defined with disjoint sets of nodes. Figure 4a presents an example of a fragmentation template for the RDF Structure depicted in Figure 3a. Instances of template t_1 extracted from an RDF graph according to this fragmentation template are illustrated in Figure 4.

Given that the fragmentation process is based on a storage threshold, we also need the notion of the size of a fragmentation template $t_i \in T$. The size of t_i is given by the sum of the expected number of occurrences of nodes multiplied by their sizes. The tree composition of fragmentation templates requires us to measure the node occurrence in the nested structure. The function $occ(n)$ maps each node in a template t_i to its expected number of occurrences in an instance of t_i. It is recursively defined as follows: $occ(n) = 1$ if n is the root node of t_i, and $occ(n) = occ(p) \times o(p, n)$ where p is a parent node of n in t_i. The size of t_i is denoted by $size(t_i) = \sum_{n \in t_i}(occ(n) \times s(n))$.

In order to formally state our problem, we need the notion of a strongly correlated set *scs* for a node in the affinity graph, defined as follows: $scs(n) = \{n' | aff(n, n') \geq aff(n', n'')$ for every node n'' directly connected to $n'\}$. Intuitively, *scs* determines which nodes have stronger affinity with n than with any other in the graph. We denote by scs^+ the transitive closure of the *scs* relation.



(a) Fragmentation template (b) Fragments for t_1 (c) Fragments for t_2

Fig. 4: Templates and Fragments

We can now state our fragmentation problem: Find T such that the following conditions are satisfied: (1) $size(t_i) \leq \Gamma$ for every $t_i \in T$; and (2) if n_1 and n_2 are nodes in the same fragment then $n_2 \in scs^+(n_1)$. The first condition defines that all fragments in T must fit in Γ and the second generates fragments that are related by affinity values higher than the values with nodes in other fragments.

As an example, consider $\Gamma = 20$ and the affinity graph depicted in Figure 3c. The fragmentation template in Figure 4a satisfies our conditions because (1) the size of templates fits in the storage threshold, that is $size(t_1) = 19$ and $size(t_2) = 4$; and (2) the affinity between any node in t_1 with any node in t_2 is lower than the affinity between any pair of nodes in the same fragment, for example, $aff(\texttt{Offer}, \texttt{Product}) < aff(\texttt{Offer}, \texttt{Vendor})$.

We propose a fragmentation algorithm based on RDF Structures and workload. The Algorithm *affFrag* takes as input an RDF Structure S with information on node sizes and number of occurrences, an affinity graph $\mathcal{A}$ and a storage threshold Γ. The algorithm computes templates of fragments based on strongly correlated sets of nodes if their sizes lie within Γ.

The algorithm processes the edges in $\mathcal{A}$ in descending order of affinity. Given an edge (n_1, n_b), the primary goal is to compute $scs(n_1)$. The node n_1 is set to be the root of the fragment being computed because it is the source node of the edge with the highest affinity. A new fragment is generated by processing edges (n_1, n_b) in *border* as follows: n_b is only considered to be inserted in the current fragment if it is related with higher affinity to some element in the current fragment than to any other outside the fragment (Lines 14-15). According to Line 13, the candidate nodes are processed in descending order of affinity in order to fill up the fragment with those with highest affinity. At the end, all nodes have been assigned to some fragment. However, before inserting new nodes in the $tNodes$ we check whether it is possible to do so within the size of Γ given the size and occurrence of the node to be included (Line 16-17).

As an example, consider the affinity graph of Figure 3c and $\Gamma = 20$ as the input to *affFrag*. The first edge to be processed is the one with highest affinity involving nodes `Product` and `label`. `Product` is inserted into a fragment t_1 as the root node. The

Algorithm affFrag

Input: RDF Structure $S = (C, L, l, A, s, o)$, Affinity Graph $\mathcal{A} = (N, E, aff)$ and Γ
Output: T fragmentation template

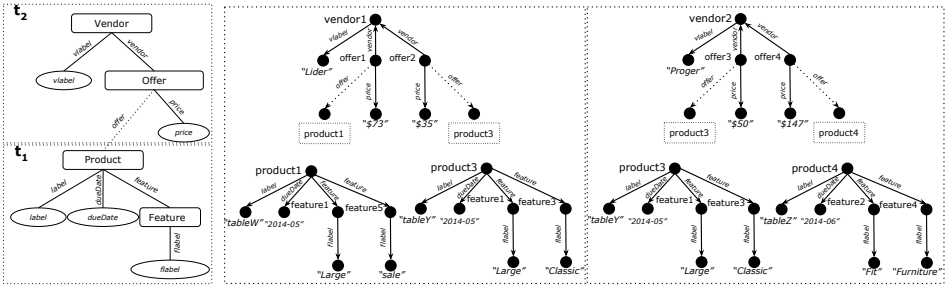
```
1  T ← {};
2  allNodes ← N;
3  allEdges ← E;
4  repeat
5      (n1, nb) ←edge in allEdges with highest affinity;
6      tNodes ← {n1};
7      tEdges ← {};
8      tSize ← s(n1);
9      Occ(nl) ← 1;
10     border ← {(n1, nb)|nb ∈ allNodes};
11     allNodes ← allNodes − {n1};
12     while tSize < Γ and border! = {} do
13         (n1, nb) ← extract edge from border with highest affinity, where n1 ∈ tNodes and nb ∉ tNodes;
14         nb Edges ← {(nb, n) ∈ allEdges|n ∈ allNodes};
15         if for all edges e ∈ nb Edges: aff(e) ≤ aff(n1, nb) then
16             Occ(nb) ← Occ(nl) × o(nb);
17             if s(nb) × Occ(nb) + tSize ≤ Γ then
18                 tNodes ← tNodes ∪ {nb};
19                 tEdges ← tEdges ∪ {(n1, nb)};
20                 border ← border ∪ nb Edges;
21                 allNodes ← allNodes − {nb};
22                 tSize ← tSize + s(nb) × Occ(nb);
23             end
24         end
25     end
26     T ← T ∪ {(tNodes, tEdges)};
27     allEdges ← allEdges − tEdges;
28 until allNodes = {};
29 output T;
```

size of t_1 is initially set to 1, given our assumption that all nodes have size 1. Since this is below the threshold, we keep inserting nodes to t_1 among those connected to `Product` which are kept in *border*. The one with highest affinity is `label`. Such node is inserted in t_1, since it is not connected to any other node with higher affinity and this insertion does not exceed the value of Γ. The same happens for inserting nodes `dueDate`, `Feature` and `flabel` into t_1. At this point, $tSize = 19$ given the simple occurrence of `dueData` and `label` with the multiple occurrence of `Feature` and `flabel`. The next edges in *border* to be considered relates `Product` to `Offer` and `price`. `Offer` should not be inserted in the fragment because its affinity is higher with nodes that are not in the current fragment. Thus, the first fragment is created with nodes `Product`, `label`, `dueDate`, `Feature` and `flabel`. A similar process creates the second fragment with `Offer`, `price`, `Vendor` and `vlabel`. The final fragmentation template generated is the one depicted in Figure 4a.

The fragmentation template defines how to partition instances of an RDF Structure, i.e., an RDF graph. Thus, a fragment is generated for each instance of the *root* node according to the fragmentation template of $t_i \in T$. In the example, t_1 must generate fragments for each *product* instance. According to the RDF graph of Figure 3a, the fragment generated for product instances may be represented by the trees in Figure 4b.

5 Clustering Fragments

Given our approach for the fragmentation problem, we now turn to the allocation problem. That is, given that a fragment is our storage unit, we are now interested in determining which fragments should be allocated in the same server. Although our fragmentation algorithm cuts the affinity graph based on affinity relations, nodes in distinct fragments may still keep strong affinity relations. This is because the fragmentation process



(a) (b)

Fig. 5: Clustering Templates and Fragments

has been designed to satisfy a storage threshold. Since there may be several template elements connected by affinity relations, we choose to group the ones with stronger affinities. More specifically, consider a fragmentation template $T = \{t_1, ..., t_m\}$ defined based on an affinity graph $\mathcal{A} = (N, \hat{E}, \texttt{aff})$. Let $E_T \subseteq \hat{E}$ be the set of edges connecting a node in a fragment t_i to the root of a fragment t_j. Observe that it is possible that $i \neq j$ as well as $i = j$. By connecting *templates* through a *root* node, we are able to extend their tree structures to define a nesting arrangement among related data. We define a clustering template as $G = \{g_1, ..., g_n\}, n \leq m$, such that G is a forest of linked fragmentation templates. Similar to the *affFrag* algorithm, groups in G are built considering edges in E_T in descending order of affinity values. Although we do not define a threshold for the group size, it is limited by the storage capacity of the server.

According to the fragmentation template in Figure 4a, the dashed arrows denote unprocessed edges in the fragmentation process. As discussed before, only edges directed to root nodes in template elements are considered to define clusters of fragments. Here, the edges (`Feature`, `Product`) and (`Offer`, `Product`) meet this requirement. Given that both edges are directed to `Product`, we choose only one of them in order to nest `Product` and keep the tree structure among the template elements. To do so, we choose the one with the highest affinity. Figure 5a presents a clustering template that relates t_1 and t_2 through the edge (`Offer`, `Product`) with the highest affinity. Instances of this cluster template are presented in Figure 5b.

We apply a clustering template to an RDF graph in order to extract fragments and cluster them properly. Some issues can arise in this process. First, a fragment should be generated for each of the root classes in the fragmentation template. However, it is possible that more fragments are required given by the variability of the size of the nodes and the number of instances for multi-valued relationships in the RDF graph. It is important to remind that both the size and the instances considered in the RDF structure correspond to average values provided as the expected workload. These values are applied to predict the size of fragments in order to define fragmentation templates. In addition, we create edges to represent edges unprocessed by the fragmentation process. To do so, edges are created in the fragments that contain their source nodes. As an example, notice that the edge (`Offer, Product`) denotes the cut between the frag-

mentation templates t_1 and t_2 in Figure 4a. However, the edges among `Offer` and `Product` instances are created in the instances of t_2 in order to keep the connection among fragments as depicted in Figure 5a.

The tree structure created by clustering and fragmentation templates may produce some data redundancy of nested data related to multi-valued relationships. However, we control the amount of replicas by applying a threshold to the amount of replicated data allowed. Due to space limitations, we omit a detailed discussion here.

6 Experimental Study

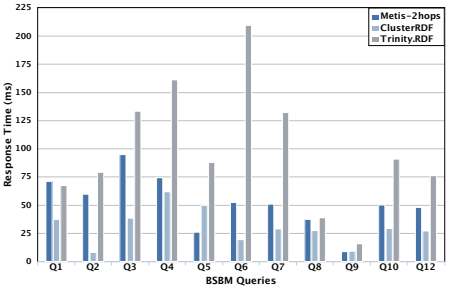
We have developed *ClusterRDF*, a system to deploy our approach based on an architecture where RDF data is partitioned across a set of servers over a distributed in-memory key-value store. We use the key-value datastore Scalaris[12] as a scalable system to leverage scalability and content locality in order to support our clustering solution. We have conducted an experimental study for determining the effect of our approach on the performance of query data retrieval. We compare *ClusterRDF* with its closest related approaches: the one introduced by Huang et al[5] and Trinity.RDF[16] using the Berlin SPARQL Benchmark (BSBM).

Huang et al. applies the METIS[1] partitioner on an RDF graph, followed by a replication step to overlap data across partitions according to an *n-hop* guarantee. We refer to this approach as *METIS-2hops* because we have implemented the undirected 2-hop guarantee version of this method. Although Trinity.RDF is focused on providing a query engine for RDF data, this system considers a hash partitioning of RDF nodes and the power law distribution of node degrees to cluster data.

BSBM provides a workload with 12 queries and a data generator that supports the creation of arbitrarily large datasets using the number of products as scale factor. Among the 12 queries defined for the benchmark, we have chosen 11, because the remaining one does not satisfy our definition of a pattern graph. For a specific dataset size and workload provided by BSBM, we have generated data clusters according to *ClusterRDF*, *METIS-2hops* and *Trinity.RDF*. Table 6b summarizes the statistics of the datasets used in this study. As expected, *ClusterRDF* and *Metis-2hop* produce space overhead in terms of triple replication. However, *Metis-2hop* produces twice as many triples compared to our method.

The goal of the experiments reported in this section is to determine the effect of our clustering method on the system performance, and compare it with both *Metis-2hops* and *Trinity.RDF*. The comparison is based on the response time required to retrieve query data from the datastore.

First, we compare the clustering approaches on a cluster of 8 servers and BSBM_5 dataset. The results are shown in figures 6a-7b. The reported times in milliseconds are the average values computed over multiple runs of the experiment and represent the cost of retrieving query data in parallel on a distributed datastore. Each server in the distributed system starts a thread and performs an arbitrary number of local or cross-server requests to retrieve the query data. In such a parallel retrieval, the thread that executes the highest number of cross-server requests determines the query response time. We have collected both the maximum number of distributed requests issued by



(a) Response Time - 8 servers and BSBM_5

Dataset	#Triples	Size	Triple Overhead	
			ClusterRDF	*Metis-2hops*
BSBM_1	40405	10.2MB	14141	27071
BSBM_2	75620	19.2MB	22686	44615
BSBM_3	191650	48.9MB	67329	120739
BSBM_4	375163	96MB	105045	213842
BSBM_5	3567636	922.3MB	891909	1748141
BSBM_6	35300350	9.97GB	7766077	15532154
BSBM_7	100399052	27GB	20079810	40159620

(b) Statistics of datasets

Fig. 6: Response Time and Statistics

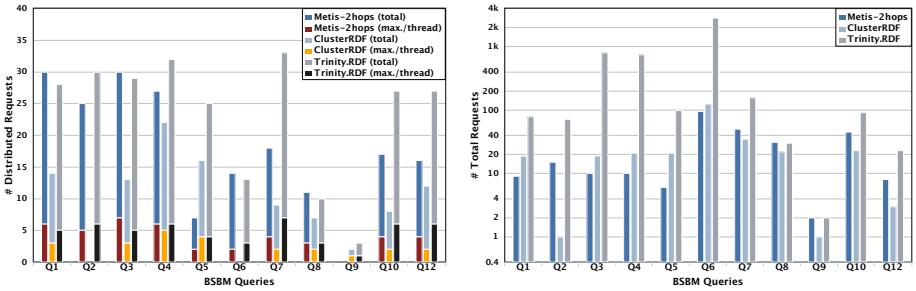
a single server as well as the total number of distributed requests for all threads in Figure 7a. Observe that the total number of distributed requests corresponds to the query segmentation denoted by the $\hat{P}$ measure (Definition 2). In addition, we have collected the total number of requests (local and distributed) in Figure 7b. Observe that the latter corresponds to the size of query results. That is, it is a measure of the total number of fragments retrieved.

Cross-server requests. As expected, there is a direct correspondence between the number of distributed requests and the response time. That is, a high number of cross-server requests induces a high cost to retrieve data spread among distributed servers. Indeed, observe that the execution of $Q1$ on *ClusterRDF* requires at most 4 servers accesses per thread, which takes 37.27 ms. The execution of the same query on the *Metis-2hops* and *Trinity.RDF* almost doubles the number of requests and has the same effect on the response time (70.94 ms and 67.52, respectively).

Intuitively, the number of cross-server requests required to retrieve query data measures the effectiveness of the partitioning methods. The difference between the results for the approaches can be explained by the coverage that each method provides in terms of the query patterns. We may say that *Metis-2hops* assures a 2-hop coverage for any pattern graph. However, a 2-hop guarantee is not enough to cover the whole pattern of the majority of queries in the BSBM workload.

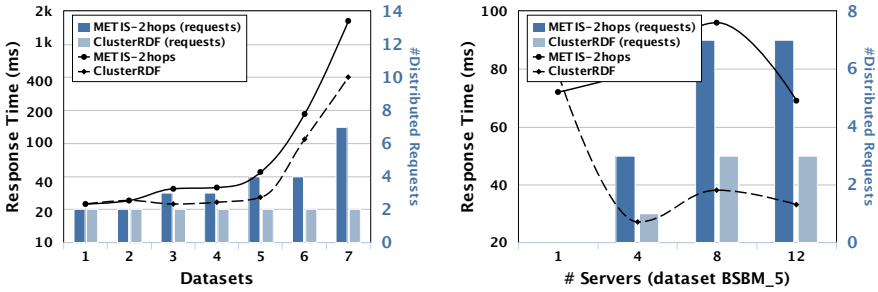
Trinity.RDF provides a simple pattern graph coverage in most cases given its fine-grained storage unit based on RDF nodes. This explains why *Trinity.RDF* presents the worst results among the three. *ClusterRDF* provides a complete coverage for queries $Q2$ and $Q6$, given that requests are issued to only one server. For the remaining queries, *ClusterRDF* does not avoid cross-server requests. However, it reduces the number of servers to be accessed if compared to the two other alternatives. The results reported in Figure 6a show that *ClusterRDF* outperforms *Metis-2hops* and *Trinity.RDF* for most queries, except for *Q5* and *Q9*. This is because *ClusterRDF* assigns data to clusters according to the access pattern of the most frequent queries of the workload.

Total requests. The size of query results is reported by the quantity of total requests in Figure 7b. This measure represents the total amount of fragments (storage units) retrieved. Scalaris provides a functionality for packing a set of requests for the same server



(a) # Distributed Requests (b) # Total Requests

Fig. 7: Number of Requests- 8 servers and BSBM_5



(a) Data Scalability for Q7 (b) Server Scalability for Q3

Fig. 8: Data and Server Scalability

into a single message for minimizing the cost of message passing. We have observed that the cost of these packed message can be ignored when the amount of requests is up to 10 requests per server. This measure is also related to the amount of irrelevant data in the fragments being retrieved. Notice that *ClusterRDF* requires a lower number of server requests than *Metis-2hops* in $Q6$, however *ClusterRDF* achieves a higher number of fragment requests. This can be explained by the fact that the requested data are in the same cluster but probably not in the same fragment. In *Trinity.RDF*, this amount is even bigger for all queries because of the its fine-grained storage model.

Data scalability. We test the methods running on a cluster of 8 servers on 7 datasets (BSBM_1 to BSBM_7) of increasing sizes. The results are shown in Figure 8a for query 7(in logarithmic scale). In general, the results of these queries increase as the size of the dataset increases. The increase of the dataset size leads to a higher number of distributed requests in most cases. This may be explained by a higher degree of the RDF nodes which requires to balance the load among servers. However, this only happens when the whole set of query data items is not set to be clustered.

Server scalability. We have deployed the systems in clusters with varying number of servers, and test its performance on dataset BSBM_5. The results are shown in Figure 8b for query 3. In general, the increase on the number of servers brings the benefits of the parallel processing and reduces the load of servers. However, this increase can also lead data to be distributed among servers when query data items are not set to be clustered. We believe that the high number of requests being performed by each thread in parallel increases the competition for resources and impacts the system performance. The worst effect of this competition is observed in $Q3$ on a cluster of 8 server for *METIS-2hops*, where each thread requires to access all servers. Notice that the effect of the parallel processing only reduces the response time when system capacity is increased to 12 servers and the number of server requests remains stable.

7 Related Work

Similar to our work, there are several graph-based approaches focused on database partitioning. However, they differ on the data model and the heuristics applied. A similar heuristic is used in the traditional algorithm *MakePartition* [6] proposed for relational databases. However, the number of fragments generated for a given dataset tends to be larger given that they do not focus on the storage capacity of the fragments. Affinity-based solutions have also been applied to XML fragmentation [3] [13] [11]. Our approach targets the RDF model and provides an extended coverage of such affinity-based approaches by clustering affinity fragments.

Our approach to generate fragmentation templates is similar to traditional vertical fragmentation techniques. Here, each instance of a template root node produces a fragment with its adjacent nodes. It is also similar to the hierarchical data model applied by Google F1 [14]. Clustering templates may also be associated to horizontal partitioning of traditional databases. In this paper we have compared *ClusterRDF* to other methods based on RDF graphs. As pointed out in Section 6, Huang et al.[5] assigns an RDF graph to a traditional graph partitioner and replicates cross-partition nodes in order to improve the query coverage. However, they only consider the associations of RDF vertexes and not the query patterns in order to provide an approximated coverage. *Trinity.RDF*[16] applies a simplest heuristic on RDF graph. In this case, high-degree nodes are identified to be clustered together with their adjacent nodes. We have demonstrated through a benchmark use case that a clustering approach based on workload analysis achieves a better approximation in terms of the coverage of frequent query patterns.

8 Conclusion Remarks

We have proposed an approach for partitioning RDF data according to an application workload defined on the structure of RDF graphs. This work makes contributions in the context of highly distributed databases, where communication costs must be reduced to provide a scalable service. In particular, *ClusterRDF* is able to reduce communication costs for distributed query evaluation by providing a suitable partition for datasets. Our experiments show that *ClusterRDF* can improve the query performance by roughly 27% to 86%, compared to *METIS-2hops*[5], a closely related approach for RDF partitioning.

We have also reported that *ClusterRDF* can perform up to 10 times faster then the hash-partitioning introduced by *Trinity.RDF*. Although *ClusterRDF* and *METIS-2hops* replicates RDF data in order to provide better results, *ClusterRDF* reduces by 50% the replication storage overhead produced by *METIS-2hops*.

Recent works evidence both the feasibility of such methods [2], [10] as well as the availability of workload data [4]. In *ClusterRDF*, both the query patterns as well as the partitioning strategy are formulated over a summarization schema that represents the data structures for an RDF dataset. By doing so, the same partitioning *template* for a query workload may be continually applied to new data. However, considering dynamicity of query patterns is a topic for future work. In addition, we plan to investigate metadata management, indexing structures and query optimization strategies.

Acknowledgments. This work was partially supported by CAPES, CNPq, Fundação Araucária and by AWS in Education.

References

1. METIS. Available at: http://glaros.dtc.umn.edu/gkhome/views/metis (2013)
2. Aluc, G., Özsu, M.T., Daudjee, K.: Workload Matters: Why RDF Databases Need a New Design. PVLDB 7(10), 837–840 (2014)
3. Bordawekar, R., Shmueli, O.: An Algorithm for Partitioning Trees Augmented with Sibling Edges. Information Processing Letters 108(3), 136–142 (2008)
4. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: A Workload-driven Approach to Database Replication and Partitioning. VLDB Endowment 3(1-2), 48–57 (2010)
5. Huang, J., Abadi, D.J.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB 4(11), 1123–1134 (2011)
6. Navathe, S., Ra, M.: Vertical Partitioning for Database Design: A Graphical Algorithm. ACM SIGMOD International Conference on Management of Data 18, 440–450 (1989)
7. Neumann, T., Moerkotte, G.: Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In: ICDE. pp. 984–994 (2011)
8. Ozsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Prentice-Hall, Inc. (1991)
9. Pham, M.: Self-organizing Structured RDF in MonetDB. In: IEEE International Conference on Data Engineering Workshops. pp. 310–313 (2013)
10. Quamar, A., Kumar, K.A., Deshpande, A.: SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads. In: EDBT. pp. 430–441 (2013)
11. Schroeder, R., Mello, R., Hara, C.: Affinity-based XML Fragmentation. In: International Workshop on the Web and Databases (WebDB). Scottsdale (2012)
12. Schütt, T., Schintke, F., Reinefeld, A.: Scalaris: Reliable Transactional P2P Key/Value Store. In: ACM SIGPLAN Workshop on ERLANG. pp. 41–48 (2008)
13. Shnaiderman, L., Shmueli, O.: IPIXSAR: Incremental Clustering of Indexed XML Data. In: International Conference on Extending Database Technology - Workshops. pp. 74–84 (2009)
14. Shute, J., Whipkey, C., Menestrina, D., et.al.: F1: A Distributed SQL Database That Scales. VLDB Endowment 6(11) (2013)
15. Yang, T., Chen, J., Wang, X., Chen, Y., Du, X.: Efficient SPARQL Query Evaluation via Automatic Data Partitioning. In: DASFAA, pp. 244–258 (2013)
16. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A Distributed Graph Engine for Web Scale RDF Data. VLDB Endowment 6(4), 265–276 (2013)
17. Zou, L., Mo, J., Chen, L., Özsu, M.T., Zhao, D.: gStore: answering SPARQL queries via subgraph matching. VLDB Endowment 4(8), 482–493 (2011)