Exploring Controlled RDF Distribution

Raqueline R. M. Penteado
Universidade Estadual de Maringá
Maringá-PR, Brazil 87020-900
Email: raque@din.uem.br

Rebeca Schroeder
Universidade do Estado de Santa Catarina
Joinville-SC, Brazil 89219-710
Email: rebeca.schroeder@udesc.br

Carmem S. Hara
Universidade Federal do Paraná
Curitiba-PR, Brazil 81531-990
Email: carmem@inf.ufpr.br

***Abstract*—RDF datasets have increased rapidly over the last few years. In order to process SPARQL queries on these large datasets, much effort has been spent on developing horizontally scalable techniques, which involve data partitioning and parallel query processing. While distribution may provide storage scalability, it may also incur high communication costs for processing queries. In this paper, we present a parallel and distributed query processing approach that explores the existence of data allocation patterns, provided by a *controlled data distribution*, that determine how RDF triples should be grouped and stored on the same server. Fragments of the RDF datastore follow a given allocation pattern and correspond also to units of communication among servers. Based on this distribution model, we define two communication strategies for query processing: *get-frag*, which requests remote servers to send fragments that contain data required by a query, and *send-result*, which forwards intermediate results. These strategies are combined on a method, called *2ways*, that chooses the adequate communication strategy whenever queries traverse fragment boundaries. We provide a cost function used to determine this choice and present experimental results. They show that our proposed technique effectively reduces the communication cost and improves the response time for processing SPARQL queries on a distributed RDF datastore.**

I. Introduction

In the last decade, the *Web* has become a major source for knowledge acquisition of the contemporary society. The Semantic Web has been proposed as a new form to publish Web data in order to make them meaningful to computers. RDF (*Resource Description Framework*) is the standard model of the Semantic Web[1]. An RDF dataset is a set of triples (s, p, o), representing that a subject s has a property p with object (or value) o. Since o can be the subject of another triple, an RDF dataset can be viewed as a directed labeled graph where subjects and objects are vertices connected by their properties. The flexibility and simplicity of the model motivated the proliferation of RDF datasets such as DBPedia, a knowledge base extracted from Wikipedia. According to the *W3C* consortium, some commercial datasets have already reached the size of 1 trillion triples[2]. Efficient query processing on such huge datasets is a big challenge for existing RDF data management systems (RDF-DMSs).

Some RDF-DMSs adopt a centralized storage approach, such as [1] and [2]. However, they lack horizontal scalability, and systems with distributed storage have been proposed, such as [3], [4], [5] and [6]. In these systems, both data and queries can be distributed among servers in order to promote distributed and parallel query processing. Distributed processing implies communication costs, since data involved in a query may be spread among multiple servers. Different methods of data distribution have been proposed in the literature to minimize this cost, based on workload [7], query patterns [8], and graph partition algorithms [3].

In this paper, we propose a distributed query processing approach that minimizes the communication overhead by exploring *controlled fragmentation and allocation strategies*, such as those proposed in [7] and [8]. More specifically, we propose a query optimization strategy that explores the information on how the RDF dataset is partitioned in order to generate a query execution plan and choose between two communication models, which we denote as *get-frag* and *send-result*. Intuitively, given that the dataset is partitioned into fragments, whenever a query involves data that is not locally stored, we consider two ways to continue the execution: request the required data from another server (*get-frag*) or send the intermediate results to other servers (*send-result*). The choice of the strategy may depend on the number of messages and the volume of data to be transmitted.

To illustrate these ideas, consider the RDF dataset represented as a graph in Figure 1(a). Observe that it is distributed over five servers (*V, X, Y, W* and *Z*), and dashed rectangles denote fragments. Note also that fragments have been created following a pattern. That is, they follow a controlled data distribution method. In this example, there are four patterns, *PaProducer, PaProduct, PaOffer* and *PaPerson* (as illustrated in the RDF structure graph of Figure 1(b)). The dataset contains five fragments of *PaOffer*, five fragments of *PaPerson*, one fragment of *PaProducer* and one of *PaProduct*. In our setting, a fragment determines both co-allocation of its components and also a transmission unit among servers.

Now consider the SPARQL query in Figure 2(a) for *retrieving data related to offers with* $value < 60,000$. For these offers, the query returns the names of their products, buyers and buyers' friends. The fragments involved in the query are determined by searching for the required properties inside patterns of the RDF structure graph. Consider a query plan that follows the order defined by the query, starting with pattern *PaProduct* (containing property *name*), traversing to pattern *PaOffer* to obtain the offer value and following twice pattern

[1] https://www.w3.org/TR/rdf-primer/
[2] http://www.w3.org/wiki / LargeTripleStores

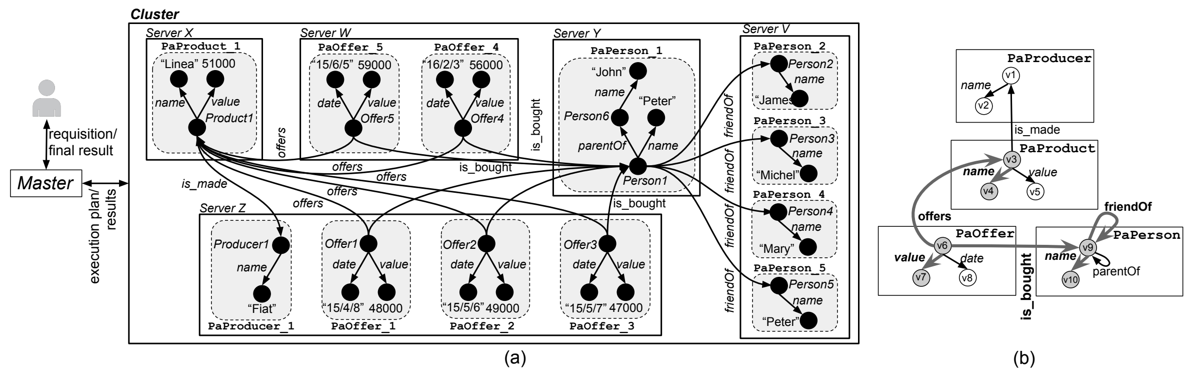


Fig. 1: (a) RDF graph; (b) RDF structure graph

PaPerson to obtain the buyer and his/her friends. To execute the query, the plan is sent to all servers to be processed in parallel, starting with locally stored fragments of *PaProduct*. However, in this particular dataset, only server X contains a fragment of *PaProduct*. From this fragment the name of the product (*"Linea"*) is extracted. In order to continue the query processing, fragments of *PaOffer* stored on servers *W* and *Z* are required. At this point, server X may choose between two communication strategies, namely *send-result* and *get-frag*. In this case, there is a single intermediate result to be sent, and five *PaOffer* fragments to be retrieved. Thus, server *X* chooses strategy *send-result*, and forwards the product *name* with some associated data to servers W and Z, which in turn continue the query execution in parallel. Both servers, after processing *PaOffer* fragments, realize that they are from the same *PaPerson* fragment, which is located on server *Y*. Thus, each server should choose a communication strategy to continue. In this case, the size of the intermediate results is larger than the single *PaPerson* fragment and thus both servers choose the *get-frag* strategy, requesting server *Y* to send the required fragment. Finally, both servers processes the fragment *PaPerson_1* and choose the *send-result* strategy to forward their intermediary results to server V. This strategy is chosen because the size of the four fragments of *PaPerson* stored on V is larger than the intermediary results on both servers. When finishing the execution, V sends the final results to the master server, that is, the server to which the query has been initially submitted.

We have implemented a SPARQL query processing system based on a graph exploration algorithm and the two communication strategies. Our experimental study shows that the combination of these strategies, which we call *2ways*, presents better performance than each strategy considered alone. To the best of our knowledge, this is the first distributed query execution model that combines more than one communication strategy. The rest of this paper is organized as follows. In Section II we discuss related work. Section III introduces the basic concepts related to this work. Our query processing approach is presented at Section IV. In Section V we experimentally investigate the impact of the *2ways* strategy and conclude in Section VI.

II. Related Work

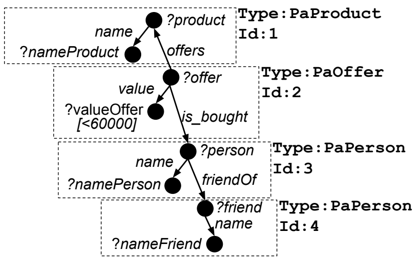
The distributed execution of SPARQL queries is a three-fold problem involving *query processing*, *query planning* and *communication*. Regarding *processing*, a traditional approach for speeding up queries consists of creating indexes on RDF triple elements (subject, predicate, object). Works such as *chameleon-db* [9] and *Triad* [6] improve query performance classifying and indexing data fragments in order to prune RDF triples that cannot contribute to the results of a given SPARQL query. Our approach indexes data fragments exploring the pattern-based data distribution model.

With respect to *planning*, in distributed storage systems, correlated data should ideally be allocated in the same server in order to avoid data exchange at query time. When the distribution model is known, query optimizers may generate plans accordingly. Systems such as *RAPID+* [10], [11], *SHAPE* [12] and *AdPart* [13] adopt data distribution patterns based on hash partitioning on the subject, predicate and/or object. More complex graph structures are considered by graph partitioners in [3], *Warp* [14], *CliqueSquare* [15] and [16]. In this case, graph cut algorithms define partitions solely based on graph properties. In addition, systems such as [3], [12], [14] and [13] adopt data replication methods to extend their patterns and maximize processing locality on servers. The query optimization method proposed in this paper is orthogonal to the partition model, as long as the *structural summarization* of the data used to group them on the same server is provided.

Different approaches have been proposed to distributed SPARQL query processing. Regarding the *communication* strategy, *send-result* is used by the majority of RDF systems. Considering the works cited above, [10], [15] and [16] adopt *MapReduce* on their computation model [17]. Given a query, *reducer jobs* receive and process intermediary results generated by *mapper jobs*. [3], [12] and [14] adopt hybrid processing approaches where *reducer jobs* process intermediary results generated by local query processors. Conversely, [13] adopt an owner processing approach where each server executes entire query plans asking intermediary results of subqueries to others servers. To minimize their communication

```
SELECT ?nameProduct, ?namePerson,
?nameFriend
WHERE {
  ?product name ?nameProduct .
  ?offer offers ?product .
  ?offer value ?valueOffer .
  ?offer is_bought ?person .
  ?person name ?namePerson .
  ?person friendOf ?friend .
  ?friend name ?nameFriend .
  FILTER (valueOffer < 60000)
}
```

(a)



(b)

S = [
s_1: ((*?product*, name, *?nameProduct*), *out*, *PaProduct*, 1, { }),
s_2: ((*?offer*, offers, *?product*), *in*, *PaProduct*, 1, { }),
s_3: ((*?offer*, value, *?valueOffer*), *out*, *PaOffer*, 2, {(?valueOffer<60000)}),
s_4: ((*?offer*, is_bought, *?person*), *out*, *PaOffer*, 2, { }),
s_5: ((*?person*, name, *?namePerson*), *out*, *PaPerson*, 3, { }),
s_6: ((*?person*, friendOf, *?friend*), *out*, *PaPerson*, 3, { }),
s_7: ((*?friend*, name, *?nameFriend*), *out*, *PaPerson*, 4, { })
]

(c)

Fig. 2: (a) SPARQL query; (b) Query graph; (c) Sequence of m_{QS}

costs, these systems use different techniques to reduce the data volume exchanged in the network. For instance, the replication methods proposed in [14] and [13] dynamically adapt their allocation patterns based on query workloads. The communication model adopted by *MAPSIN* [11] is similar to the *get-frag* strategy. *MAPSIN* is a framework that aims at reducing the communication cost of *MapReduce* jobs executing *map-side* joins. To this end, each server locally processes triple sets retrieved from the other servers at query time. Works such as [18] and [19] consider that data fragments are retrieved from federated databases by *clients* (requesters), distributing the execution load among the clients.

Our processing approach defines the *2ways* method which provides the ability to choose between two strategies at query time. This choice depends on the amount of data involved. Such flexibility is not found in any other related work. The results in Section V show that *2ways* presents better performance than each strategy considered alone.

III. Preliminaries

This section presents basic definitions used throughout the paper. An RDF dataset is a set of triples *(subject, property, object)*, which can be viewed as a directed labeled graph where subjects and objects are vertices connected by their properties. In this work, we define an RDF graph G_D as a set of vertices with their adjacency list. That is, each vertex v is defined by a tuple *(uri,val,in,out)*, where: *uri(v)* is the vertex's unique identifier, *val(v)* is a literal value, *in(v)* is the set of incident edges in v, and *out(v)* is the set of outgoing edges from v. In the RDF graph of Figure 1(a), vertices in *PaProducer_1* are define as: *Producer1 = (uri: "http://example", val: undefined, in:*$\{$*(is_made, Product1)*$\}$*, out:* $\{$*(name,* v_{Fiat}*)*$\}$*)*, and v_{Fiat} = *(uri: undefined, val: "Fiat", in:*$\{$*(name, Producer1)*$\}$*, out:*$\{\}$*)*.

We consider a controlled data distribution of the RDF graph, which has been partitioned into fragments of a given structure. Although some RDF datasets are schema-free, [20] shows that a large number of real datasets have regular structures. Our approach targets these datasets. More specifically, we consider that an RDF structure graph $G_S = (V_S, E_S)$ has been extracted from the RDF graph. We apply a technique similar to the one proposed in [20] in our experiments, which is based on the concept of *characteristic sets* and ensures that triples that share the same subject are allocated on the same server. However, our query processing model can be applied to any distribution model that fragments/categorizes data through predefined patterns. Consider again the RDF graph G_D in Figure 1(a). The RDF structure graph in Figure 1(b) models G_D, and there is a total mapping from nodes in G_D to V_S. For example, *Producer1* $\mapsto v1$, and we say that *type(Producer1)* = $v1$.

A partition $\mathcal{P}$ of the RDF structure graph G_S defines disjoint set of (connected) vertices in V_S, which we call *allocation patterns* (PAs). PAs are used to partition an RDF graph G_D into fragments that follow their structure, as illustrated in Figure 1(b). Allocation patterns define storage co-allocation and communication units. That is, vertices in G_D that belong to the same fragment are allocated in the same server and any communication between servers carries along all vertices in a fragment. We define a function Pa mapping V_S to $\mathcal{P}$, and thus *Pa(*$v1$*)* = *PaProducer*. Figure 1(b) shows four PAs. As an example, *PaProducer* = $\{v1, v2\}$. Observe that edges in G_S can connect vertices in the same PA or in distinct PAs. We call the first as an *intra-PA* property (*I*) and the latter as an *inter-PA* property (*E*). In our running example, *PaPerson* = $\{v9{,}v10\}$, where $v9 = (in : \{(is\_bought, v6, E), (parentOf, v9, I), (friendOf, v9, E)\}, out : \{(name, v10,\ I), (parentOf, v9,\ I), (friendOf, v9, E)\})$. Intuitively this PA defines that persons in the same family are grouped in the same fragment, while offers and friends are stored in distinct fragments, as shown in Figure 1(a).

We adopt a SPARQL fragment with the operators *select*, *filter* and *concatenation* via a point symbol, as illustrated in Figure 2(a). Queries of this form can be represented as a query graph $G_Q = (V_Q, E_Q, F, P)$, where V_Q is a set of vertices that correspond to variables, E_Q is a set of directed labeled edges, F is a set of filters, and P are the projection variables on the *select* clause. Figure 2(b) shows the query graph that corresponds to the query in Figure 2(a). Observe that although in the example the query is represented as a tree, cycles are admitted. However, we only consider connected query graphs.

SPARQL query processing can be transformed to a problem of subgraph matching where subgraphs of G_D that are homomorphic to G_Q are retrieved. That is, given that in G_Q pairs of vertices (u, v) are connected by property p – $(u, v, p) \in E_Q$ – and E_D connects vertices in G_D, the homomorphism is defined as a function f such that $(p, f(v))$ is in the set of outgoing edges of $f(u)$ whenever $(u, v, p) \in E_Q$. Our distributed query processing approach is a variation of the subgraph isomorphism algorithm proposed in [21]. It is a backtracking algorithm which finds solutions by incrementing partial solutions or abandoning them when it determines they cannot be completed [22].

IV. Query Processing Approach

The approach proposed in this paper considers a *shared-nothing Master-Slave* architecture. The master server receives and analyses query requisitions, generates execution plans and requires slaves in a cluster to execute the plan. All slaves start to process the plan in parallel with their local data. When the query requires data stored on remote servers, a communication with each of them is started following one of the strategies proposed in our model: *get-frag* or *send-result*. This architecture follows the inter-node parallelism with asynchronous BSP (*Bulk Synchronous Parallel*) computation model, similar to [23]. In this model, each server executes in parallel the same query plan, without requiring synchronism among servers for transitions between steps. Our processing model is divided in query planning and execution.

A. Planning

Given that our model is based on graph exploration, a query plan determines in which order triple patterns in a query will be traversed on an RDF graph. Our approach explores the controlled distribution storage model for minimizing inter-server communication. More specifically, given that nodes in the same allocation pattern (PA) are guaranteed to be stored on the same server, the query plan explores intra-PA properties before traversing inter-PA properties.

The RDF structure and allocation patterns are used to reduce the search space for entry points in the RDF graph for processing a query. Recall that each vertex in the RDF structure belongs to a single allocation pattern. Thus, by matching the query graph with the RDF structure, we are able to determine which patterns are required for processing the query and thus reduce the search space to only fragments of these patterns. To do so, we adopt a two steps process. In the first step, a graph exploration algorithm finds subgraphs of the RDF structure G_S that is homomorphic to the query graph G_Q. Although graph matching has exponential time complexity, in practice, query graphs are usually small and RDF structures are much smaller than the RDF dataset itself. The result is a set of mappings M_{QS} from V_Q to nodes in V_S. Considering again the query graph in Figure 2(a), the bolded edges in Figure 1(b) highlight its homomorphic subgraph in G_S. Observe that in general, there may exist multiple resulting mappings. For example, the query *(SELECT ?nameAll WHERE {?x name ?nameAll.})* finds three homomorphic subgraphs in G_S in Figure 1(b). For each such homomorphism, we build a query plan, and the execution of a query is defined as the union of the results generated by these *set* of query plans. Thus, for the second step, we detail how query plans are generated for each mapping $m_{QS} \in M_{QS}$. Initially, the query graph is partitioned into PA occurrences, based on m_{QS} as follows. PA occurrences are defined as the largest connected subgraphs (V_{PA}, E_{PA}) of V_Q such that all nodes in V_{PA} are mapped to nodes in the same PA and they are connected only by intra-PA properties. As an example, Figure 2(b) shows the four PA occurrences of the query in Figure 2(a). Each PA occurrence *oc* has a *type*, consisting of the PA to which all nodes in *oc* are mapped to, and is associated with a unique identifier *id*(*oc*).

PA occurrences are used to generate a linear query plan in which nodes in the same allocation pattern are visited successively, before moving to other patterns. Formally, given a query graph $G_Q = (V_Q, E_Q, F, P)$, and a structure graph G_S, the result of the query plan generator for a mapping m_{QS} is a *linear* graph traversal sequence $S = [s_1, \ldots, s_n]$, such that each step s_i of the traversal is a tuple (a, dir, pat, id, f), where: (**1**) a is a triple pattern $(s, p, o) \in E_Q$, (**2**) $dir \in \{in, out\}$; if $dir = out$ the traversal is from the subject s to the object *oc*, and we define *source(s_i) =* s, and *target(s_i)=* o. Conversely, if $dir = in$ the traversal is from the object *oc* to subject s, and we define *source(s_i)=* o, and *target(s_i)=* s; (**3**) pat is the type of the PA occurrence which contains the triple pattern (s, p, o); (**4**) id is a unique identifier of a PA occurrence. Here, we say that *all* steps with the same id compose a *PA traversal*. (**5**) f is a set of filters defined on *target*. Given a query plan S, the *source* of the first step in the sequence defines the graph *initial point of exploration*. Consider again the example in Figure 2(b). A possible plan traversal sequence consists of the steps shown in Figure 2(c), where *?product* is the initial point of exploration.

We say that a traversal sequence S is *valid* with respect to a graph query $G_Q = (V_Q, E_Q, F, P)$, the structure graph G_S, and a mapping m_{QS} if it satisfies the following conditions: (**C1**) it processes G_Q, that is, $\bigcup_{s \in S}\{s.a\} = E_q$ and $|S| = |E_q|$; (**C2**) for every step s_j, $2 <= j <= |S|$, if *source(s_j)* $= v$ then there exists a step s_i, $i < j$ such that v =*source(s_i)* or v =*target(s_i)*; (**C3**) for any pair of step s_i, s_j, $i < j$, if $id(s_i)$=$id(s_j)$ then for every step s_k, $i < k < j$, $id(s_k)$=$id(s_i)$; (**C4**) for every step s_i, if *source*(s_i)=v and v is a vertex in the PA occurrence *oc* then $id(s_i) = id(oc)$ and $pat(s_i) = type(o)$. The condition C1 requires that all steps of S are executed. The condition C2 requires that starting from the *initial points* of exploration every step traverses an edge *from vertices obtained in previous steps*. The condition C3 requires traversals inside each pattern to be grouped in the sequence. The condition C4 requires that all PA occurrences have the correct patterns and steps.

Note that the order in which PAs are explored impacts query processing performance. However, determining an *optimized* PA exploration order is outside the scope of this work. Our experimental results show that *2ways* is effective regardless the ordering of PAs.

B. Execution

Query execution plans, composed of a set of traversal sequences $\mathcal{S} = \{S_1, \ldots S_m\}$ are generated by the master and sent to all servers in a cluster to start execution in parallel. As slaves finish computing the entire sequence, their results are sent to the master, which is responsible for gathering and returning them to the user. We assume that each server in the cluster has an index mapping patterns to their fragments that are *locally* stored.

The Pseudo-code in Figure 3 shows the query execution algorithm, where each sequence $S_i \in \mathcal{S}$ is processed by a server L as follows. Let $S_i = [s_1, \ldots, s_n]$ and G_L be the subset of the RDF dataset stored in L. First, fragments of $s_1.pat$ are retrieved to obtain nodes in the RDF graph that correspond to the initial points of exploration $source(s_1)$ (lines 1-5). Recall that function *type* maps nodes in an RDF graph to nodes in an RDF structure G_S, and m_{QS} is a mapping from nodes in a query graph G_Q to G_S. Thus, we compute the initial points of exploration for a sequence S_i in a server L as $[\![s_0]\!]_L = \{\{source(s_1) \mapsto v_D\} \mid type v_D = v_S, m_{QS}(source(s_1)) = v_S$ and $v_D \in G_L\}$. In our running query example, considering the RDF graph in Figure 1(a), we have $[\![s_0]\!]_X = \{\{?product \mapsto Product1\}\}$, since $type(Product1) = v_3$, $m_{QS}(?product) = v_3$ and *Product1* is stored on server X. For the other servers, $[\![s_0]\!]_Y$=$[\![s_0]\!]_Z$=$[\![s_0]\!]_W$=$[\![s_0]\!]_V$=$\{\}$.

Starting from these initial points of exploration, each step s_i of the traversal sequence adds a new variable mapping by following the RDF graph according to its triple pattern (s, p, o) and direction *dir*; if the pattern is not found or the new variable mapping does not satisfy the filters, the corresponding mapping is removed from the result set (lines 6-14). Filters of s_i consider constant values and instantiations on previous triple patterns of its *target*. In our example, after processing the first step, $[\![s_1]\!]_X = \{\{?product \mapsto Product1, ?nameProduct \mapsto v_{\text{“}Linea''}\}\}$. Observe that s_2 can be processed in the same server X as step s_1 because $s_1.id = s_2.id$ (line 16). That is, the *source* nodes in both steps are in the same pattern allocation and thus they are *guaranteed* to be stored on the same server. Continuing with our running example, the result of processing step s_2 in X contains five mapping: $[\![s_2]\!]_X = \{[\![s_1]\!]_X \cup \{?offer \mapsto Offer1\}, [\![s_1]\!]_X \cup \{?offer \mapsto Offer2\}, [\![s_1]\!]_X \cup \{?offer \mapsto Offer3\}, [\![s_1]\!]_X \cup \{?offer \mapsto Offer4\}, [\![s_1]\!]_X \cup \{?offer \mapsto Offer5\}\}$. However, the processing can not continue in the same server for step s_3 since $s_3.id \neq s_2.id$ (line 18). Indeed, in our example every *?offer* is mapped to a node stored on a remote server via an inter-PA property *offers*. As a result, in order to continue the query, either server X requests the remote servers to transmit the fragments in which these *offer* nodes are stored and the query continues to be processed on server X, or the existing mappings are forwarded to the remote servers, and they continue the execution of the plan (lines 20-33). We call the first communication strategy as *get-frag*, and the second as *send-result*.

The choice of the communication strategy is made during query execution, based on the number of requisitions and the volume of the data to be transmitted. Let s_i be the step in which a traversal to a different pattern occurs. That is, $s_i.id \neq s_{i+1}.id$, $i >= 1$. Let $varJoin$ be the s_{i+1}'s *source* variable

```
Function PlanExecution on server $L$
Input: $m_{QS}$, sequence $S = [s_1, \ldots, s_n]$, step number $j$, M;
Output: a set of mappings $M$;
 1. Let $s_j$ be $((s,p,o), dir, pat, id, f)$;
 2. if $j = 1$ then    /* looking for initial points*/
 3.   for each vertex $v$ in a fragment $f$ of type $pat$ stored in $L$ do
 4.     if $type(v)$ = $m_{QS}(source(s_1))$ then
 5.       insert $\{(source(s_1) \mapsto v)\}$ in $M$;
 6. newM:= $\{\}$;
 7. for each $m$ in $M$ do
 8.   Let $v$ be a node in $G_D$ such that $source(s_j) \mapsto v$ is in $m$;
 9.   if $dir$ = $in$ then
10.     $targetSet$ = $\{v_{new} \mid (p, v_{new})$ is an incoming edge in $v.in\}$;
11.   else $targetSet$ = $\{v_{new} \mid (p, v_{new})$ is an outgoing edge in $v.out\}$;
12.   for all $v_{new}$ in $targetSet$ do
13.     if $v_{new}$ satisfies filters $f$ then
14.       newM:= newM $\cup$ $\{m \cup \{target(s_j) \mapsto v_{new}\}\}$;
15. if $j < |S|$ then
16.   if $s_{j+1}.id = s_j.id$ then
17.     execute PlanExecution($m_{QS}$,$S$,$j+1$,$newM$) on $L$;
18.   else   /* continue process in server that stores $s_{j+1}$ */
19.     Initialize partial[R] and frag[R] with empty sets for every server $R$;
20.     for each mapping $m$ in newM do
21.       Let $joinNode$ be a node such that $source_{j+1} \mapsto joinNode$ is
22.        in $m$ and let $R$ be the server that stores $joinNode$;
23.       partial[R]:= partial[R] $\cup$ $\{m\}$;
24.       frag[R]:= frag[R] $\cup$ $\{joinNode\}$;
25.     for each server $R$ with frag[R] $\neq$ { } do
26.       if R=L then
27.         execute PlanExecution($m_{QS}$,$S$,$j$+1,$partial[L]$) on server $L$;
28.       else
29.         if $Cost_{SR}[R] < Cost_{GF}[R]$ then
30.           execute PlanExecution($m_{QS}$,$S$,$j$+1,$partial[R]$) on server $R$;
31.         else
32.           request frag[$R$] from server R;
33.           execute PlanExecution($m_{QS}$,$S$,$j$+1,$partial[R]$) on server $L$;
34. else return newM to master server; /* execution plan completed*/
```

Fig. 3: Execution algorithm for a query plan

and $[\![s_i]\!]|_R$ be the subset of $[\![s_i]\!]$ with mappings of $varJoin$ to a node in server R. That is, $[\![s_i]\!]|_R = \{m \in [\![s_i]\!] | varJoin \mapsto joinNode\ is\ in\ m,$ and $joinNode \in G_R\}$. Intuitively, these are the *intermediate results* to be sent to each server R by the *send-result* strategy. In this case, the number of messages to be transmitted to a remote server R (NM_R) is the size of set $[\![s_i]\!]|_R$ (line 23). The volume of data (SM_R) to be transmitted in each message is the size (in bytes) of an intermediate result. Given these, a server L may compute for each remote server R, the cost of applying the *send-result* (SR) strategy by:

$$Cost_{SR}[R] = NM_R * (cc + SM_R/tr)$$

where cc is the (hardware-dependent) cost of establishing a connection from L to R and tr is the (network-dependent) transmission rate from L to R.

Observe that several mappings in $[\![s_i]\!]|_R$ may share the same value for $varJoin$. That is, there may exist more than one intermediate result that “point to” the same node stored on a remote server R. Thus, if we apply the *get-frag* strategy, only one fragment has to be transmitted from the remote server to L in order to continue processing multiple intermediate results. We define these sets of (distinct) nodes to be requested from a server R to apply the *get-frag* strategy as $[\![s_i]\!]|_R[varJoin] = \{joinNode \mid m \in [\![s_i]\!]|_R, (varJoin \mapsto joinNode)\ is\ in\ m\}$ (line 24). Thus, the number of messages to be transmitted from a remote server R to L (NM_R) is the size of the set $[\![s_i]\!]|_R[varJoin]$. The volume of data to be transmitted in each message varies for each pattern allocation. We consider that the average size of fragments of a given pattern *pat* (SM_{pat}) is determined during loading time of the dataset. Given these, a server L may compute for each remote server R, the cost of applying the *get-frag* (GF) strategy by:

$$Cost_{GF}[R] = NM_R * (cc + SM_{pat}/tr + SM_{pat} * st)$$

where cc and tr are as already defined, and st (repository-dependent) is the cost of storing a fragment locally on L.

Then, if the *send-result* cost ($Cost_{SR}[R]$) is less than the *get-frag* cost ($Cost_{GF}[R]$) the intermediate results (mappings) are sent to server R, which continues processing the query (line 30). Otherwise, server L requests the required fragments and continues the query execution (lines 32-33). Note that L may also contain fragments pointed by $joinNode$ (line 26). Finally, when a slave server explores the last step of the traversal sequence, its set of mappings is projected over the variables in the *select* clause and the result is sent to the master (line 34) .

Note that the cost for each communication strategy is computed for each remote server (line 25). As a result, it is possible that for a remote server R_1, $Cost_{SR}[R_1] < Cost_{GF}[R_1]$, and for another server R_2, $Cost_{SR}[R_2] > Cost_{GF}[R_2]$. Thus, we propose the method *2ways*, that chooses for each server, the strategy that is expected to have lower communication cost. Observe that we consider a model in which multiple messages to the same server are not "packed" in order to minimize the cost of establishing connections. For such systems, the cost function would have to be modified accordingly. Moreover, we have defined the cost function based solely on information locally acquired by each individual server. Other variables that require global knowledge, such as the load of each server, could be considered in the cost function. However, obtaining such information requires additional communication among the servers. As a final remark, the actual value of *cc*, tr and st may differ between pairs of servers and periods of time. Their (dynamic) calibration is outside the scope of this paper.

V. Experimental Study

We have conducted an experimental study for determining the effect of the communication strategies on the performance and scalability of SPARQL query processing on top of a distributed graph exploration processor. In particular, we compare *send-result*, *get-frag* and *2ways*, where *send-result* represents the communication strategies used in [3] and *SHAPE* [12], and *get-frag* represents the strategy used in *MAPSIN* [11].

A. Experimental Settings

The query processing approach presented in Section IV was implemented in Java adopting its RMI native communication model. The system was deployed on a cluster of dedicated Amazon EC2 m4.large instances, each one with 8 GB of memory and 2 virtual 24 GHz CPUs. In order to evaluate the scalability of communication methods, three different clusters were built varying the amount of EC2 instances: 4 ($C1$), 8 ($C2$) and 12 ($C3$) servers. We use the Berkeley DB repository[3] as the storage layer. Berkeley DB was deployed as an in-memory datastore on top of the EC2 cluster.

Datasets and queries applied in this study are extracted from BSBM [24], which is built around an e-commerce use case where the schema models the relationships between products, product features, producers, offers, vendors and product reviews. BSBM provides a query mix and a data generator that uses the number of products as scale factor. We have generated 3 datasets for our experiments: $BSBM\_1$ with 5,000 products (1,811,316 triples), $BSBM\_2$ with 10,000 products (3,567,636 triples) and $BSBM\_3$ with 15,000 products (5,323,644 triples). Data and metadata were uniformly distributed among the servers with fragments following the model described in Section III.

We considered a workload of 10 queries ($Q1 - Q10$). In this set, $Q1 - Q5$ are based on BSBM. Queries $Q6 - Q10$ were created to address different communication situations in order to properly evaluate the communication strategies. We highlight some experimental results in the following sections. Query statements and detailed results are available as a web supplement[4].

B. Performance

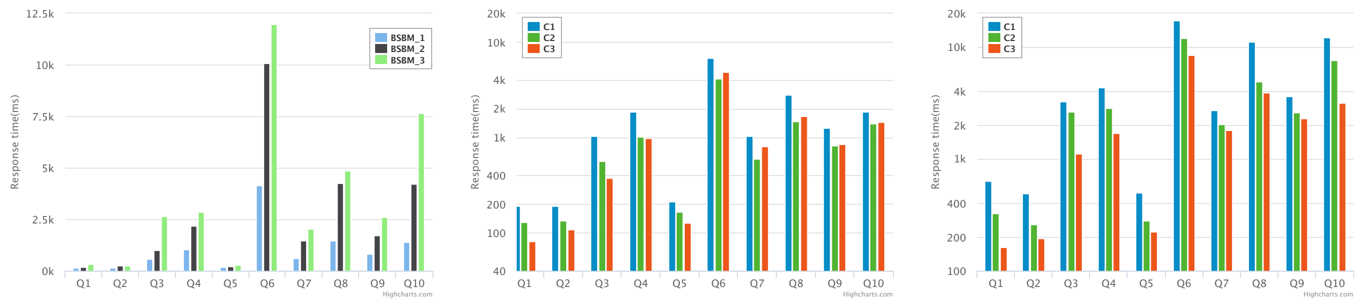
The purpose of this experiment is to determine the response time for queries in the workload. Queries were processed using only the *send-result (SR)* and *get-frag (GF)* strategies, and their performance is compared with the *2ways* approach. Table I presents the average query response time in milliseconds, considering the $BSBM\_2$ dataset on a $C3$ cluster. Each query may involve one ($Q1$), two ($Q2, Q4, Q7, Q9$), or three PAs ($Q3, Q5, Q6, Q8, Q10$) and the exploration order defined in the query plan follows the order they are presented in each row, from PA_1 to PA_3. Columns *#fr* and *#rt* refer to the number of fragments required to process the query, and the number of intermediate results generated after processing each PA. Indeed, they correspond to the *sum* of the number of messages (NM) *for all servers* in the cost functions presented in Section IV for *get-frag* and *send-result*, respectively.

Table I shows that for some queries *send-result* presents better performance compared to *get-frag*, and for others *get-frag* performs better. Moreover, the *2ways* performs as well as the best strategy, and for some cases better, such as for $Q6$. $Q1$ is processed locally given that only one PA is required. Therefore, all communication models achieve the same response time. For queries $Q2$, $Q3$, $Q5$, $Q7$ and $Q8$, *send-result* was the best strategy for all transitions, and thus the response time of *2ways* is similar to *send-result*. For most of them, this choice has been made mainly because the number of intermediate results after processing one PA is smaller than the required fragments of the next PA. As an example, for $Q2$ the query processor can either send 1 intermediate result

[3]http://www.oracle.com/technetwork/database/database-technologies/berkeleydb
[4]http://www.inf.ufpr.br/rrmpenteado/2ways/

TABLE I: Allocation patterns and response time of three communication strategies on $BSBM\_2 - C3$

Q	PA_1			PA_2			PA_3			*Response Time (ms)*		
	pattern	*#fg*	*#rt*	*pattern*	*#fg*	*#rt*	*pattern*	*#fg*	*#rt*	*SR*	*GF*	*2ways*
Q1	*PaProduct*	10k	1	-	-	-	-	-	-	110	110	110
Q2	*PaProduct*	10k	1	*PaFeature*	17	17	-	-	-	134	161	133
Q3	*PaOffer*	200k	1	*PaProduct*	1	1	*PaProducer*	1	1	1003	1027	980
Q4	*PaReview*	100k	7036	*PaProduct*	2	7036	-	-	-	4068	1427	1430
Q5	*PaProduct*	10k	1	*PaOffer*	14	14	*PaVendor*	2	14	143	160	144
Q6	*PaVendor*	508	213	*PaOffer*	80157	80157	*PaProduct*	4	80157	15002	24917	9826
Q7	*PaProducer*	206	206	*PaProduct*	10k	10k	-	-	-	1231	2748	1230
Q8	*PaProducer*	206	206	*PaProduct*	10k	10k	*PaOffer*	200k	14330	2919	31986	2920
Q9	*PaProduct*	10k	10k	*PaProducer*	206	10k	-	-	-	1755	1618	1620
Q10	*PaOffer*	200k	14330	*PaProduct*	2	14330	*PaProducer*	2	14330	11298	2849	2853



(a) Data scalability on C2 cluster (b) Server scalability - $BSBM\_1$ (c) Server scalability - $BSBM\_3$

Fig. 4: Data and server scalability of the *2ways* strategy

computed from a *PaProduct* fragment or request 17 fragments of *PaFeature* from remote servers, and thus chooses the first. For $Q2$, the number of transmissions were the same for both strategies, but the size of the intermediate result were smaller than the fragments. For $Q5$, the 14 *PaOffer* results were spread over the cluster, each of them requiring 2 *PaVendor* fragments. Since the choice is made locally, all of them chose the *send-result* strategy.

For queries $Q4$, $Q9$ and $Q10$ *get-frag* presented better performance for all PA transitions, and thus the response time of *2ways* is similar to *get-frag*. In our query mix, $Q6$ was the only query for which *2ways* chose *send-result* from PA_1 to PA_2 and *get-frag* from PA_2 to PA_3. This mixed strategy resulted in a response time 35% lower than *send-result* and 61% lower than *get-frag*. These results show the following. First, there may exist a difference of several orders of magnitude between the response time of *send-result* and *get-frag* such as in queries $Q8$ and $Q4$. Thus, our approach outperforms systems that rely on a single communication strategy for all queries such as *SHAPE* [12] and *MAPSIN* [11]. Second, our cost function for strategy *2ways* has correctly chosen the best strategy for every transition between PAs in our workload. We can conclude that *2ways* is an effective and efficient method for query optimization.

As a final notice, we would like to point out that queries $Q1 - Q8$ have "optimized" query plans, in the sense that PAs have been ordered by their selectivity in order to reduce the number of intermediate results. This optimization has not been applied for the last two queries. In fact, $Q9$ and $Q10$ execute the same query as $Q7$ and $Q8$, respectively, but with a different (PA) exploration order. Comparing their response times, we can see that indeed the "optimized" query plan has better performance for *send-result*, but this is not the case for *get-frag*. For this strategy, the "unoptimized" query plans are 59% and 11 times faster. However, the *2ways* strategy have similar response times for both plans. It shows that our proposed query optimization technique is orthogonal to query planning.

Regarding experiments with different cluster sizes and datasets, the communication strategy applied by *2ways* were similar to the ones reported in Table I. However, $Q9$ had its predominant communication strategy changed on $BSBM\_3 - C2$ and $BSBM\_3 - C3$. The *send-result* strategy presented better performance in these scenarios and *2ways* has correctly chosen it. The change was caused by the larger size (in bytes) of *PaProducer* fragments in these scenarios, which increased the cost of the *get-frag* strategy.

C. Scalability

The scalability of *2ways* is evaluated by scaling the dataset size and the number of servers in the cluster. Figure 4a shows the scalability of our processing model running on the $C2$ cluster on datasets $BSBM\_1$, $BSBM\_2$ and $BSBM\_3$. As expected, the response time of the queries increase with the size of the dataset, given that the input data size usually determines the number of intermediate results and data exchange among servers. Moreover, this increase also affects the local processing time of data fragments on servers. This effect is more evident for queries involving large data volumes, such as $Q3$, $Q6$, $Q8$ and $Q10$. In $Q3$, for example, the number of messages is the same for the three datasets. However, the reported difference on the response time results from the number of *PaOffer* fragments processed: 100, 200 and 300 thousand on $BSBM\_1$, $BSBM\_2$ and $BSBM\_3$, respectively.

We have also determined the effect of the number of servers in the cluster on the query response time. The results are shown in Figures 4b and 4c on $BSBM\_1$ and $BSBM\_3$ datasets, respectively. The dataset $BSBM\_2$ behaved as $BSBM\_3$. It can be noticed that when the number of server increases, the response time decreases for datasets. However, increasing the number of servers to 12 ($C3$) does not have the same effect for most queries on $BSBM\_1$ and some queries on $BSBM\_3$. This shows that distributing data and processing queries in parallel are beneficial up to a point in which the cost of communication among servers to retrieve all required data in a query surpasses the gain. In our experiments, this point is reached with 8 servers for $BSBM\_1$ and 12 servers for $BSBM\_3$. Thus, for this particular workload, allocating more servers than this number decreases performance both energy-wise and time-wise.

VI. Conclusion

This paper proposes an optimization technique called *2ways*, which aims at reducing the communication cost for processing queries on a distributed RDF datastore. We assume that fragments of the datastore have been distributed among servers, and that fragments group RDF triples according to allocation patterns (PA). Information on PAs are used to generate query plans that explore data inside fragments, which are guaranteed to be stored on the same server, before exploring other fragments, which may be stored remotely. Whenever data on a remote fragment is required, *2ways* chooses between two communication strategies, *send-result* and *get-frag*, based on a cost function that takes into consideration the number of messages and the volume of data to be transmitted. Our experimental analysis shows that *2ways* can effectively and efficiently reduce the query response time compared to strategies *send-result* and *get-frag* considered alone. This paper brings contributions in the context of large-scale databases, since the proposed strategy can be exploited by RDF data management systems to provide scalability for processing queries. Future work includes studies on the ordering of PAs during query planning and the calibration of hardware, network and repository-dependent variables in the cost function.

Acknowledgment

This work was supported in part by CAPES and AWS in Education.

References

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management," *The VLDB Journal*, vol. 18, no. 2, pp. 385–406, 2009.

[2] T. Neumann and G. Weikum, "The RDF-3X Engine for Scalable Management of RDF Data," *The VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.

[3] J. Huang, D. J. Abadi, and K. Ren, "Scalable SPARQL Querying of Large RDF Graphs," *37th International Conference on Very Large Data Bases*, vol. 4, no. 11, pp. 1123–1134, 2011.

[4] M. Przyjaciel-Zablocki, A. Schaetzle, E. Skaley, T. Hornung, and G. Lausen, "Map-Side Merge Joins for Scalable SPARQL BGP Processing," in *5th International Conference on Cloud Computing Technology and Science*, vol. 1. IEEE Computer Society, 2013, pp. 631–638.

[5] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale RDF data," in *39th international conference on Very Large Data Bases*, 2013, pp. 265–276.

[6] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing," in *ACM SIGMOD International Conference on Management of Data*. New York, USA: ACM, 2014, pp. 289–300.

[7] R. Schroeder and C. S. Hara, "Partitioning Templates for RDF," in *Advances in Databases and Information Systems: 19th East European Conference*, ser. Lecture Notes in Computer Science, M. Tadeusz, P. Valduriez, and L. Bellatreche, Eds., vol. 9282. Springer, 2015, pp. 305–319.

[8] L. Gai, W. Chen, and T. Wang, "A partition-based Summary-Graph-Driven Method for Efficient RDF Query Processing," *CoRR*, vol. abs/1510.07749, 2015.

[9] G. Aluç, M. T. Özsu, K. Daudjee, and O. Hartig, "chameleon-db: a Workload-Aware Robust RDF Data Management System," Tech. Rep. CS-2013-10, 2013.

[10] P. Ravindra, H. Kim, and K. Anyanwu, "An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce," in *The Semanic Web: Research and Applications: 8th Extended Semantic Web Conference*, 2011, pp. 46–61.

[11] M. Przyjaciel-Zablocki, A. Schtzle, T. Hornung, C. Dorner, and G. Lausen, "Cascading Map-side Joins over HBase for Scalable Join Processing," *CoRR*, vol. abs/1206.6293, 2012.

[12] K. Lee and L. Liu, "Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning," *VLDB Endowment*, vol. 6, no. 14, 2013.

[13] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, "Accelerating SPARQL Queries by Exploiting Hash-based Locality and Adaptive Partitioning," *The VLDB Journal*, vol. 25, no. 3, pp. 1–26, 2016.

[14] K. Hose and R. Schenkel, "WARP: Workload-aware replication and partitioning for RDF," in *ICDE Workshops*. IEEE Computer Society, 2013, pp. 1–6.

[15] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis, "CliqueSquare: efficient Hadoop-based RDF query processing," in *BDA'13 - Journées de Bases de Données Avancées*, 2013.

[16] B. Wu, H. Jin, and P. Yuan, "Scalable SAPRQL Querying Processing on Large RDF Data in Cloud Computing Environment," in *Pervasive Computing and the Networked World: Joint International Conference, ICPCA/SWS 2012*, ser. Lecture Notes in Computer Science, Q. Zu, B. Hu, and A. Elçi, Eds., vol. 7719. Springer, 2012, pp. 631–646.

[17] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM - 50th Anniversary Issue: 1958 - 2008*, vol. 51, no. 1, pp. 107–113, 2008.

[18] R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, and P. Colpaert, "Triple Pattern Fragments: a low-cost knowledge graph interface for the Web," *Journal of Web Semantics*, vol. 37, pp. 184–206, 2016.

[19] C. Buil-Aranda, A. Polleres, and J. Umbrich, "Strategies for Executing Federated Queries in SPARQL1.1," in *International Semantic Web Conference (2)*, 2014, pp. 390–405.

[20] P. Minh-Duc, P. Linnea, E. Orri, and P. Boncz, "Deriving an Emergent Relational Schema from RDF Data," in *24th International Conference on World Wide Web*, 2015, pp. 864–874.

[21] J. R. Ullmann, "An Algorithm for Subgraph Isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[22] J. Lee, W.-S. Han, R. Kasperovics, and J. Lee, "An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases," in *39th international conference on Very Large Data Bases*, 2013, pp. 133–144.

[23] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *10th USENIX Conference on Operating Systems Design and Implementation*, 2012, pp. 17–30.

[24] C. Bizer and A. Schultz, "The Berlin SPARQL Benchmark," *International Journal on Semantic Web and Information Systems*, vol. 5, no. 2, pp. 1–24, 2009.