WARP: Workload-Aware Replication and Partitioning for RDF

Katja Hose *1, Ralf Schenkel #2

*Department of Computer Science, Aalborg University, Denmark
#Max Planck Institute for Informatics, Saarbrücken, Germany

Abstract—With the increasing popularity of the Semantic Web, more and more data becomes available in RDF with SPARQL as a query language. Data sets, however, can become too big to be managed and queried on a single server in a scalable way. Existing distributed RDF stores address this problem using data partitioning, aiming at limiting the communication between servers and exploiting parallelism. This paper proposes a distributed SPARQL engine that combines a graph partitioning technique with workload-aware replication of triples across partitions, enabling efficient query execution even for complex queries from the workload. Furthermore, it discusses query optimization techniques for producing efficient execution plans for ad-hoc queries not contained in the workload.

I. INTRODUCTION

The Resource Description Framework (RDF), together with the query language SPARQL, has become the standard for many Semantic Web applications and especially the Linked Open Data Community. The number of RDF data sources is continuously growing, and with it also the amount of data provided by each source. DBpedia1, for instance, has now reached a size of about 1.9 billion RDF triples, and according to the W3C commercial data sets have already exceeded the 1 trillion triples barrier2. Facing this large amount of data that keeps on growing, centralized single server solutions will not much longer be able to efficiently answer queries and provide results with reasonable performance.

To deal with this large amount of data, first clustered RDF systems have been proposed [1]–[3]. Their key consideration is to partition triples among machines based on hash values of their subjects, predicates, and objects, or using range-partitioned B+-trees. For query evaluation, these systems aim at exploiting parallelism and minimizing communication between machines. Single triple patterns can usually be evaluated efficiently in such systems since the constants in the triple pattern identify only one of the partitions as relevant. However, evaluating joins is often very expensive because partial results from different machines must be matched. Thus, complex queries might require multiple rounds of communication and data exchange over the network.

For reducing this communication overhead, Huang et al. [4] proposed to partition the RDF graph into compact subgraphs and assign each subgraph to a host. Using replication at the borders of the partitions, queries that do not exceed a certain diameter can be processed in parallel over all partitions without further communication between hosts. All other queries are evaluated using cross-partition joins implemented as MapReduce [5] jobs, which is very expensive in terms of execution time (with 20s startup overhead according to [4]). The key to improve performance hence is to replicate more triples at partition borders and process more queries locally, but this comes at the price of highly increased storage cost.

The typical query workload of many applications, which may result from regular form-based Web interfaces, consists of queries with a very similar structure with similar or even the same predicates. Exploiting such a regular workload for partitioning the RDF data can improve performance and response time for future queries similar to the workload. To the best of our knowledge, this is the first paper that considers a workload to define partitions of RDF data sets. In particular, building upon the techniques by Huang et al. [4], we present WARP (Workload-Aware Replication and Partitioning) with the following main contributions:

- advanced replication methods on top of graph-based partitioning that take a workload into account and enable efficient execution of all workload queries, and
- cost-aware query optimization and efficient query execution for arbitrary queries without the need for expensive MapReduce jobs.

The remainder of this paper is structured as follows. We first discuss related work in Section II and present the basic partitioning approach in Section III. Section IV then introduces our workload-aware replication technique, Section V introduces efficient query processing approaches, Section VI presents evaluation results, and Section VII concludes the paper.

II. RELATED WORK

With more and more RDF datasets becoming available, there is an increasing number of techniques for efficient query answering over multiple independent RDF sources. Ideally, all sources provide SPARQL endpoints, enabling the application of efficient distributed query processing techniques [6], [7]. Alternatively, for sources without SPARQL access, the data can be downloaded either at query time or in advance into a data warehouse. To handle queries over large amounts of RDF
data (single sources or data warehouses), parallel processing in clusters of machines has been proposed, for instance, by applying MapReduce [5]-style techniques [8], [9], where complex SPARQL queries are executed as a series of distributed hash joins on the join variables. However, the triples are repartitioned for each query, which makes evaluating complex queries expensive.

Similar observations hold for P2P-based solutions [10], [11], which mostly rely on hashing to assign triples to servers during bootstrapping. Even if each triple is stored three times according to its values for subject, predicate, and object, computing joins can easily become very expensive because evaluation usually involves multiple roundtrips over the partitions to find and evaluate all bindings for all join variables.

There is also a number of commercial systems using clusters of machines for scalable processing of RDF data without using MapReduce. Again, the main problem in this setup is how to create appropriate partitions of the data. Bigdata3 uses B+ tree indexes to organize the data, dynamically partitions the B+ tree into index partitions (key-range shards), and assigns the partitions to the servers. For query processing, Bigdata faces similar problems as the systems mentioned above. The OWLIM Replication Cluster4 uses multiple machines to increase query throughput, but does not partition the data; instead, all data is replicated at all nodes.

Partitioning of data (i.e., fragmentation and allocation) are common in traditional relational distributed database systems. In this context, workloads have been considered to determine optimal partitions applying horizontal (row-wise) and vertical (column-wise) fragmentation of relations [12]–[14], i.e., the data is partitioned according to predicates contained in frequent queries. As a result, these queries can often be fully evaluated on a single partition. However, it is not straightforward how to apply the same methods to RDF and SPARQL since they were developed for scenarios with different characteristics, e.g., joins between multiple relations vs. many self-joins for each query in case of RDF data.

The recently proposed Schism [15] approach uses graph partitioning for workload-based partitioning of relational data. Nodes in the constructed graph correspond to tuples, edges connect two tuples when a transaction accesses both tuples. A graph partitioner splits the graph into partitions, and a machine-learning technique determines a predicate-based explanation of the resulting partitions. As SPARQL queries usually reference a high number of triples and would therefore generate a huge number of connections between nodes, Schism is not applicable to large scale RDF scenarios.

III. BASELINE APPROACH

In this section, we outline a baseline approach [4] that applies graph-based RDF partitioning and distributed parallel SPARQL query processing in combination with MapReduce. We extend this approach with efficient techniques for distributed SPARQL query processing avoiding expensive

MapReduce jobs in Section III-C, and further extend it with workload-aware replication in Section IV.

A. Graph-based Data Partitioning and Replication

We consider RDF triples of the form (subject, predicate, object). Given a set $T$ of such triples, we consider the RDF data graph $G_T = (V_T, E_T)$ where each vertex $v \in V_T$ corresponds to a URI or literal occurring as subject or object of a triple. For each triple $t \in T$, an edge $e \in E_T$ exists that connects the vertices $v_1$ and $v_2$ corresponding to the $t$’s subject and object, $e$ is labelled with $t$’s predicate. Furthermore, for a SPARQL query $q$, we consider only its triple patterns (essentially triples where each component can be variable, denoted by a leading $\tau$) since they determine the structure of the results. The query graph $G_q$ of $q$ has all variables and constants of $q$ as nodes and an edge $(v, w)$ for each triple pattern $v \tau p w$ in $q$.

The technique proposed in [4] now first partitions $G_T$ into a set of $k$ disjoint partitions (where $k$ is the number of servers). As graph partitioning is a well-studied problem, a number of efficient solutions are available; we use the METIS partitioner5 which was also used in [4]. The partitioning process assigns each vertex to a partition. Triples are then assigned to partitions according to their subjects, i.e., a triple is assigned to the partition containing the vertex corresponding to its subject. In the following, we will often say that the partition owns the vertex or triple. Each partition is stored at a dedicated host in a triple store; we currently use RDF-3X [16]. With such a partitioning, a single triple pattern can be evaluated without missing any results by sending it to all hosts, evaluating it there in parallel on the local data, and sending the results back to the query initiator. The same technique can be applied for queries with multiple triple patterns that all join on the subject variable (star queries). For more complex queries, however, it is necessary to perform join(s) of partial results, requiring communication between hosts. An efficient solution requires a trade-off between execution time (communication costs) and replication (storage space). Huang et al. [4] propose to replicate triples at the border of partitions into neighbor partitions. This allows for executing certain classes of more complex queries locally at each partition without the need of further cross-join of partitions and communication.

Figure 1 shows an example where all triples connected to vertexes assigned to a partition in a distance of 2 hops

3http://www.bigdata.com/
4http://www.ontotext.com/owlim/replication-cluster
5http://glaros.dtc.umn.edu/gkhome/views/metis
are replicated, i.e., although the vertexes corresponding to the triples’ subjects have been assigned to another partition, they are replicated at the partition whose vertexes they are connected to within a distance of 2 hops. Formally, the partitioning realizes a 2-hop (undirected) replication horizon.

Definition 1: Let $G = (V, E)$ be a graph. Given an initial partition $W \subseteq V$ of $V$, we define the directed $n$-hop replication horizon $P_n(W)$ for $W$ recursively as follows:

(i) $P_1(W) = \{V_0, E_0\}$ where $V_0 = W$ and $E_0 = \emptyset$

(ii) $P_{n+1}(W) = \{V_{n+1}, E_{n+1}\}$ where

$$V_{n+1} = \{(v, v') \in E, v' \in V_n \cup V_n, E_{n+1} = \{(v, v') | (v, v') \in E, v \in V_n, v' \in V_{n+1}\} \cup E_n.$$

The directed 1-hop replication horizon of a graph partition adds all vertexes to the partition that are connected via a single directed edge from any vertex already in the partition, along with the edges that connect them to the original set of vertexes in the partition. The directed 2-hop replication horizon starts with the directed 1-hop replication horizon and adds another layer of vertexes and edges connected to this subgraph. Similarly, we can define undirected replication horizons, where not only ‘outgoing’, but also ‘incoming’ edges are considered; see [4] for details. We will consider only directed replication horizons in the remainder of this paper, but all presented techniques can be extended in a straightforward way to undirected replication.

As the same triple may be included in multiple partitions due to replication and therefore appear multiple times in the result of a query, the original (or primary) partition $P$ of a triple $t = (s, p, o)$ is marked by an additional triple $(\text{isOwned}, P)$ and added to $P$.

B. Processing One-Pass Queries

Queries consisting of single triple patterns as well as star queries can be evaluated in parallel in each partition independent of the replication horizon. For queries that are more complex than star queries, we need to analyze the structure of a query to decide whether the replication horizon allows to evaluate the query in parallel without further communication. We refer to such queries as one-pass queries. A key concept towards this decision is the center node of a query:

Definition 2: The center node of a query is the node with the smallest distance of the farthest edge (DoFE) and can be determined as follows:

- For each node $v$ in the query graph $G_q = (V_q, E_q)$ determine DoFE($v, G_q$):

$$\forall e = (v_1, v_2) \in E_q \text{ compute } \text{dist}(v, e) = \min(\text{distance}(v, v_1)) + 1$$

- DoFE($v, G_q$) = $\max(\text{dist}(v, e))$

- Choose a node $v \in V_q$ as center node so that

$$\forall v' \in V_q : \text{DoFE}(v, G_q) \leq \text{DoFE}(v', G_q)$$

A one-pass query (OPQ) for an $n$-hop directed replication horizon is a query $G_q$ whose center node $v_c$ has a distance of the farthest edge (DoFE) that is smaller than $n$, i.e., $\text{DoFE}(v_c, G_q) \leq n$; any query that is not a one-pass query is a multi-pass query (MPQ). We extend each query by one triple pattern with predicate $\text{isOwned}$ and the center node as subject. Therefore, only the primary partition will produce a result for the query and we do not need to filter out duplicates produced by other partitions.

Figure 2(a) shows an abstract example query with its highlighted center node $?P$, which has a DoFE of 2. Figure 2(b) illustrates that even though the center node binds to a vertex that is located at the border of a partition, the 2-hop replication (Figure 1) guarantees that the query of Figure 2(a) can be answered without further communication between partitions.

C. Processing Multi-Pass Queries

Multipass queries are processed by a dedicated coordinator host. We split such queries up into multiple OPQ subqueries, evaluate them in parallel, combine the result with additional joins at the coordinator, reduce the results to bindings for output variables, and remove duplicates. Unlike [4], where MapReduce jobs are used to compute the joins connecting the subqueries, we propose an alternative using efficient merge joins and pipelining at the coordinator, which results in a much smaller overhead (as our experiments show).

To determine a good execution plan, our optimizer first creates all possible splits for an MPQ into a set of OPQs under a directed $n$-hop replication horizon. First, it computes all OQP subqueries by considering each vertex in the query graph as center node and traversing the graph from there up to a depth of $n$. We refer to the nodes at the border of the DoFE limit connecting to further nodes as bridging nodes because they serve as the bridge between subqueries. A split is then any set of subqueries that (1) form the complete query graph when unioned and (2) overlap only in bridging nodes.

For each of these splits, the optimizer creates a query execution plan consisting of a chain of join operators (left deep join tree) between subqueries that need to be evaluated to compute the final result. Leave nodes in this tree correspond to results of the subqueries received from all partitions. To find a good execution plan, our current implementation uses the heuristic to choose the split with the smallest number of subqueries that was already used in [4]; applying a cost model and statistics is subject of future work.

For query execution, the leaf node subqueries are sent to all hosts, evaluated, sorted according to the join key, and the results are reported back to the coordinator, which combines them using efficient merge union operators. The joins in the query plan are executed using either efficient merge join operators or hash join operators in dependence on whether resorting a partial result according to the next join variable is expensive or not. Note that all these operators use pipelining and parallel processing on the coordinator so that first results can be reported already after a short time.

IV. WORKLOAD-AWARE REPLICATION

As pointed out before, query processing is most efficient for OPQ queries. To increase the fraction of OPQ queries, we have two options: (i) increase the $n$-hop replication horizon, which would ultimately result in a high amount of replicated data, or (ii) choose a reasonable replication horizon, e.g., $n = 2$ [4].
and add systematic replication for frequently issued queries. In this paper, we advocate the second option since, as we will show in the experiments, comes with only a small storage overhead. We first discuss how to select and normalize a representative workload and then explain our workload-driven replication strategy.

A. Representative Workload

For most applications, it will be possible to determine a representative workload, e.g., based on query logs or Web forms. Given such a workload, the first step is to analyze and normalize it.

To avoid overfitting, we remove all infrequent constants (strings and URIs) at subject and object positions in the triple patterns and replace them with variables. Likewise, we remove filter expressions from the workload as they only reduce the result size but never generate additional results. Only if a constant or a filter expression frequently occurs in combination with a specific triple pattern (threshold $t_c$), we keep the constant/filter and consider the triple pattern with the filter as a fixed unit.

Afterwards, we prune infrequent normalized queries occurring less than $t_p$ times. Next, for queries with optional patterns, all possible combinations of the non-optional patterns with a subset of the optional patterns are added to the workload; this is needed to ensure that all necessary triples are replicated.

B. Replication for MPQs

We now assume that the RDF graph has already be partitioned with an $n$-hop replication horizon, and that a system has been set up that allows to execute any SPARQL query on this collection (as explained in the previous Section). The goal is now to determine, for each MPQ $q$ in the workload, which triples to additionally replicate so that $q$ is turned into an OPQ under this extended replication. The following algorithm gives a high-level overview of this process:

1) Compute, for each triple pattern $t_p$ of $q$, all result bindings $R(t_p, v, h)$ for each variable $v$ at each host $h$.

2) Compute all possible splits of $q$ into OPQ subqueries, resulting in $S(q) = \{s_1, ..., s_{mn}\}$.

3) For each split $S = \{s_1, ..., s_n\} \in S(q)$
   a) For each subquery $s_i \in S$
      - Select $s_i$ as candidate seed subquery
      - Determine bindings for all bridging nodes of $s_i$
      - Determine all triples that need to be replicated
     b) Choose the seed subquery resulting in the minimum number of triples to replicate

4) Choose the split with the minimum number of replication triples and materialize the replication

The center node of $q$ is the center node of the seed query in the chosen split. We will now discuss details of step 3.

Let us consider the example query in Figure 2(c). The query asks for published articles and the articles that cite them. Normally, this query would appear in a Web form with a constant for variable ?title – this constant was removed during normalization. Figures 2(c) also illustrates a possible split in to OPQ subqueries $s_1$ (green) and $s_2$ (blue). Let us assume we chose $s_1$ as candidate seed subquery for replication. The triples we need to replicate are involved in producing answers to $s_2$ and “connect” to results obtained for $s_1$ via the bridging node (two-colored node in Figure 2(c)). Thus, we first evaluate the seed query $s_1$ in parallel at all hosts and collect bindings for the bridging node ?author2 together with the hosts they originate from. We then consider the boundary query connected to this node, which is $s_2$, and evaluate it to collect all bindings for its variables. In general, such a boundary query may be MPQ, so we need to run it through the existing distributed system. We can use the known bindings for the bridging node to restrict results – for instance, using FILTER expressions in SPARQL 1.0 or BINDINGS in SPARQL 1.1. Each set of bindings for $s_2$ determines a set of triples (by replacing variables in the triple patterns by their binding). For a binding of the bridging node that originated from host $h$, we need to make sure that all triples are at that host that correspond to bindings for $s_2$ with the same bridging node binding. Since most of these triples will usually already exist at $h$, we use the per-triple-pattern bindings $R(t_p, v, h)$ computed in step 1 to determine all triples that need replication.

Note that even our example considered only two subqueries, the same algorithm can be applied in the more general case of split with more than two subqueries. Figure 2(d) shows a split for the same query with respect to a directed 1-hop replication horizon. In this case, the above algorithm needs to be applied recursively, i.e., we have a chain of connected queries. Starting with $s_1$ (green) as seed subquery $s_1$, we
evaluate it in parallel over all partitions and obtain pairs of bindings for \(?art2\) and \(?author1\). We evaluate \(s_2\) (red) over all partitions and replicate the triples producing results for \(s_2\) at the partitions producing results for \(s_8\) with the same binding for the bridging node \(?author1\). Likewise, we evaluate the (MPQ) query consisting of \(s_3\) (blue) and \(s_4\) (yellow) in the existing distributed system and determine the necessary triples to replicate.

This approach can also handle queries with cycles. The only difference is that replication triples need to be identified based on their bindings for multiple bridging nodes.

C. Further Optimizations

This optimization procedure can be improved in several ways. First, we do not necessarily have to compute all combinations of splits and seeds and apply an exhaustive search but instead use heuristics to reduce the search space. For instance, we can stop computing the number of replication triples for a split as soon as the number exceeds the total number for a (split, seed) combination computed for any of the previous splits. However, as most SPARQL queries have only a few nodes, we can in general afford exhaustive search.

Second, we can consider overlap between queries. Replicating triples for one query might already replicate triples that another query might benefit from. Thus, to find the overall optimum, we would have to consider the sets of triples to replicate for each query, each split, and each seed subquery and compute the global optimum. This is possible and requires to keep track of all possible replication triples for all queries, splits, and seeds in the workload so that the optimum of all possible combinations over all queries can be found.

A third optimization could use an estimator based on statistics and selectivities to estimate the number of replication triples instead of computing them. Estimations reduce the time required for optimization, but as the optimization is run as an offline algorithm before the system is actually used, we can usually invest some more time into the optimization process, avoid the risk of working with erroneous estimations instead, and work with the exact values instead.

V. QUERY PROCESSING

After having turned all MPQs in the workload into OPQs by replicating triples in addition to the \(n\)-hop replication horizon, queries are processed in a similar way as with the \(n\)-hop replication only. Given a user query \(q\), we first determine its center node and DoFE value. If this value is at most \(n\), \(q\) is an OPQ and is sent to all hosts, evaluated in parallel, results are received, duplicated are removed, and the result is output to the user without further computation. If the DoFE value is greater than \(n\), we check if \(q\) was contained in the workload that we have optimized for. In this case, we process it as an OPQ because our optimization turned the MPQ into an OPQ. Otherwise, we decompose the query into OPQ subqueries and create an efficient query execution plan as discussed in Section III-C.

However, queries without an exact match in the workload can still be considered OPQs if they are contained in at least one query \(q_c\) that we have optimized for. In our current implementation, we do not attempt a full containment mapping but adhere to some heuristics. A query \(q\) is contained in a query \(q_c\) if one of the following conditions hold:

- \(q\) is the same as \(q_c\) (different names for variables are mapped) and constants in \(q\) correspond to variables in \(q_c\).
- \(q\) is the same as \(q_c\) (different names for variables are mapped) and \(q\) has additional triple patterns within the \(n\)-hop replication horizon of the seed subquery used for replication.

These queries can be supported because they strictly reduce the number of results and, in the second case, all triples required for matching additional triple patterns are available due to the original \(n\)-hop replication horizon.

Queries containing additional triple patterns for non-seed subqueries can be handled with compensating queries, i.e., the bindings for bridging nodes for non-seed subqueries with additional constraints need to be re-evaluated and removed from the overall result. As this might become expensive, we need to define a cost model that decides whether it is better to use compensating queries or evaluate the query as a non-OPQ. For the time being, we leave the details for future work.

Queries containing optional patterns are evaluated as OPQs if the same or more general query was contained in the workload (see above) or otherwise if the workload contained (i) a query that includes the optional part and (ii) a query that does not include the optional part. Splitting up queries containing optional patterns works similar to standard queries, the optional part and the main query are evaluated (and possibly split into OPQs) as separate queries. Only in the very last step, we need to consider the semantics of the optional query and combine the two result sets according to the original query.

VI. EXPERIMENTS

We evaluated our proposed technique in comparison to a single server RDF-3X\(^4\) installation and an improved version of the approach proposed by Huang et al. [4], where in contrast to the original approach we do not use MapReduce to evaluate MPQ but already apply the techniques for query optimization outlined in Section III-C.

Our test data originates from the Billion Triple Challenge (BTC) 2008\(^7\), which was already used in [16] to evaluate RDF-3X. It contains more than 562M triples originating from different datasets that are a part of the Linked Open Data cloud. We generated 16 random queries from the BTC data set by randomly picking predicates from the data, using the predicate as seed to generate a triple pattern, and extending the triple pattern according to the available data to SPARQL queries. The obtained queries consist of 4–10 triple patterns and have a maximum path length of connected triple patterns of about 2–4 (determining the DoFE values of the queries). We omitted smaller queries because their execution times would

\(^4\)http://code.google.com/p/rdf3x/
\(^7\)http://challenge.semanticweb.org
be roughly the same for all distributed setups we consider in our experiments.

We do not present results for a comparison of graph-partitioning and hash-partitioning approaches as this was already done in [4]. Instead, we focus on presenting our findings with respect to (i) the centralized execution of the query load (BTC), (ii) 1/2-hop replication horizon without workload aware replication as proposed in [4] (BTC-5-H1, BTC-5-H2), and (iii) 1/2-hop replication horizon with workload aware replication (BTC-5-H1+WARP, BTC-5-H2+WARP). Note that although we only present experimental results for a configuration of 5 hosts and directed replication horizons, the same findings also hold for other setups. In fact, we benefit from the scalability when adding more hosts and an undirected replication horizon in the same way as Huang et al. [4].

As Figure 3 shows, OPQs can be evaluated efficiently in the distributed approach with similar execution times (sometimes even faster) as the centralized approach. For query optimization and execution at the coordinator as outlined in Section III-C our approach executes MPOs with an overhead that is much less than the reported 20s overhead per join between subqueries (MapReduce job) by Huang et al. [4].

Figure 3 also reveals the difference in execution time between OPQs and MPQs, e.g., queries 2, 8, and 9, which are all MPQs with respect to the original $n$-hop replication horizon. Thus, execution times for these queries without WARP is relatively high compared to the tests with WARP, where we replicated additional triples for the query so that it could efficiently be evaluated as OPQ. These queries illustrate that in spite of using an $n$-hop replication horizon some frequently occurring queries can become expensive and that the techniques we propose in this paper effectively allow for an efficient evaluation of such queries.

The replication overhead introduced by the method by Huang et al. [4] is significant, resulting in approximately 80 million extra triples for 1-hop replication. Our workload aware method results in a negligible additional replication overhead of 3056 triples in this case (1803 triples for 2-hop replication).

VII. CONCLUSIONS AND FUTURE WORK

This paper presented WARP, an efficient approach towards workload-aware partitioning and replication for large scale data sets. By partitioning the data and replicating the triples referenced by workload queries, we can efficiently evaluate these queries independently over all partitions in parallel.

There are some optimizations that we will consider in future work, e.g., considering the overall balance and sizes of partitions and their replicas, more complex containment mappings to decide if a query can be answered in one pass, and compensation queries for queries that are partially contained in the workload. For many applications and data sets with rare updates, it is sufficient to set up the system once and re-optimize the partitioning after a sufficient number of updates have been gathered. For each update, we in principle need to consider each partition and decide whether to replicate the triple to the partition or not in dependence on the graph partitioning, the $n$-hop replication horizon, and the workload. A detailed analysis was out of the scope of this paper, so we consider it in our future work.

REFERENCES