Adaptive Workload-based Partitioning and Replication for RDF Graphs

Ahmed Al-Ghezi and Lena Wiese

Institute of Computer Science, University of Göttingen
{ahmed.al-ghezi|wiese}@cs.uni-goettingen.de

Abstract. Distributed processing of RDF data requires partitioning of big and complex data sets. The partitioning affects the performance of the distributed query processing and the amount of data transfer between the network-connected nodes. Static graph partitioning aims to generate partitions with lowest number of edges in between but suffers high communication cost when a query trespasses a partition's border, because then it requires moving partial results across the network. Workload-aware partitioning is an alternative but faces complex decisions regarding the storage space and the workload orientation. In this paper, we present an adaptive partitioning and replication strategy on three levels. We initialize our system with static partitioning where it collects and analyzes the received workload; then we let it adapt itself with two levels of dynamic replications, besides applying a weighting system to its initial static partitioning to decrease the ratio of border nodes.

1 Introduction

The exploding size of RDF-represented web data led to methods that warehouse and process the data in a distributed system. The first challenge for such systems is to partition the big RDF graph into several fragments, which then need to be assigned wisely to the clustered working nodes. A SPARQL query is represented as a graph on its own with some vertices and/or edges represented as variables. Query execution is the process of finding all the sub-graphs in the RDF-graph that match the query graph. Using several hosts connected in a cluster, a SPARQL query can be received and executed in parallel by all the hosts on their local share of data. Unfortunately, we have an expensive communication cost, whenever the query requires matching sub-graphs located in different hosts. Working towards local query execution and avoiding the communication cost is usually following two directions: (1) a better partitioning strategy, and (2) replication of selected important triples across hosts. Partitioning requires no extra storage space but finding the best strategy often involves solving a difficult problem due to the complex relationships which are embedded in a typical RDF graph. Replication has, in turn, two factors that mainly affect its feasibility: (1) a good selection of graph parts to replicate, and (2) the available storage space which restricts the amount of replicated data. Regarding the first factor, [9] studied a set of DBpedia's queries: more than 90% of the queries target only 163 frequent sub-graphs. This highlights the impact of the query workload to identify important parts of the RDF graph and parts which should be kept together in a single host. On the other hand, the storage space as the second factor affecting a replication strategy limits the amount of replicated data which any host can handle. However, the amount of storage space in a practical triple-store system is variable; it depends on the size of the data set and of other important data contained in the system like indices and statistics tables. Hence, a partitioning and replication strategy focusing on saving disk space might perform poorly if the system happens to have a lot of free storage space, and vice versa.

To address these challenges, our main contributions in this paper are:

– We present our RDF partitioning and replication approach, which is built on two stages – with respect to the existence and absence of workload – and three levels – with respect to storage space consumption.
– Given that the availability of storage space is variable, we describe the optimized system behaviour with respect to the amount and type of replications, to adapt to the different possible levels of storage space availability.

To the best of our knowledge, this is the first work in a distributed RDF triple store, that considers the adaption of a partitioning and replication layer with both the workload and the storage space availability.

The paper is structured as follows. We discuss related work in Section 2, then we introduce the preliminaries and terminologies used in this paper in Section 3. We describe the system's initial cold partitioning and replication approach in Section 4. Section 5 describes how the system reacts to a collected workload and adapts its storage layer. Finally we state our bench marking report and conclusion in Section 6.

2 Related Work

Many recent works deal with general graph partitioning, including works targeting the problem of RDF graph partitioning. METIS [6] is a popular baseline for many works like J. Huang [5], WARP [4], and TriAD [3] which applies a hash function to assign many METIS partitions to hosts. Other works considere the semantic embedded in the RDF data; Qiang Xu [12] apply a Page Rank inspired algorithm to cluster the RDF data. EAGRE [13] identifies an entity concept for the RDF data, which is then used to group and compress the entities which belong to the same RDF classes. Wu et al. [11] follow path partitioning: first identify closed paths in the RDF graph, then assign the paths that share merged vertices to the same partition. Margo et al. [7] perform edge partitioning by converting the graph into an elimination tree and then trying to reduce communication cost while performing partitioning on the resulting tree. Among the workload-based approaches, however WARP [4] and Partout [2] can be considered baselines. Following the work of Partout, Padiya et al. [8] identify properties which are queried together and reflect this when partitioning the data set. Similar to the min-terms used in Partout and WARP, Peng et al. [9] detect frequent patterns in the workload, measured by the frequency of appearance.

3 Preliminaries

In this section, we state the main definitions (RDF graph, METIS partitioning [6] and the partition's border region) needed throughout this paper.

Definition 1 (RDF Graph). *Let $G = \{V, E, P\}$ be a graph representing the RDF data set. V is a set of all the subjects and objects in the set of RDF triples D; $E \subseteq V \times V$ is a set of directed edges representing all the triples in the data set; P is a set of all the edges' labels in the RDF data, and we denote p_e as the property associated with edge $e \in E$. The RDF data set is then defined as $D = \{(s, p_e, o) \mid \exists e = (s, o) : e \in E \wedge p_e \in P\}$*

Definition 2 (METIS Partitioning). *We refer to METIS as a function $metis(v)$ which for any $v \in V$ returns the static partition number which v belongs to. We could then define the partition $r_i = \{v \in V \mid metis(v) = i\}$.*

Definition 3 (Border Region). *For a partition r_i, $border(i) = \{v \in r_i \mid \exists(v, v_m) \in E : v_m \notin r_i\}$. The border region with depth δ is defined as the follows: $border(i, \delta) = \{v \in V \mid v \notin r_i, outdepth(v, i) \leq \delta\}$, where the $outdepth(v, i)$ is the distance between any vertex $v \notin r_i$ and the partition border $border(i)$.*

Note that $border(i)$ refers to nodes inside a partition while $border(i, \delta)$ refers to nodes in neighboring partitions. Last, we define the query workload.

Definition 4 (Queries Workload, Query Answer). *A query q is a set of triple patterns $\{t_1, t_2, ..., t_n\}$; each triple pattern $(\grave{s}, \check{p}, \grave{o}) \in q$ has at least one but not more than two constants, while the rest are variables. This set composes a query graph q_G. The query answer q_a is the set of all sub-graphs in RDF graph G that match the query graph and substitute the corresponding variables. A workload is defined as a set: $Q = \{(q_1, f_1), (q_2, f_2), ..., (q_m, f_m)\}$, where q_i is a SPARQL query, and f_i is the frequency of its appearance in the workload. The workload answer Q_a is the set of the query answers of Q. The length of query q is the maximum distance between any two vertices in its graph q_G.*

4 Cold-Start Partitioning

Without a pre-assumption about the query workload, our system starts by performing static graph partitioning using METIS. This produces n partitions for given n hosts. Each host handles its share of data and can provide space to accept replications. The best area for replications is located at the hosts' border regions with some distance in depth. We differentiate in our system between two types of replication: full and compressed.

4.1 Compressed Replicated Data

In triple stores like RDF3X[1], a dictionary is used to map the textual URIs found in the triples to compressed numerical data which are then stored in

[1] https://code.google.com/archive/p/rdf3x/

different indices. In the context of replication, the later architecture produces two types: *full replication* where the host has full information about the graph data including the dictionary entries, and the *compressed replication* where the host has only the numerical data or even part of it. Given RDF data set represented by graph G which has size T triples, the minimum length of numerical code value used in dictionary is given by: $log_2 \frac{T}{\kappa}$ bits, where κ is the average number of edges per vertex in G. The total size in bytes of the T triples when stored in the numerical form is given by:

$$size(T) = \frac{3T \cdot log_2(\frac{T}{\kappa})}{8} \tag{1}$$

4.2 Initial Cold Replication

At this stage, each host has a given amount of storage space assigned for replication denoted by S, and employs all of this space by replicating triples from its neighbour hosts which are located at $outdepth = 0$ form its border; it then iteratively increases the *outdepth* and replicates the affected triples until the storage space is fully taken. However, at each *outdepth* the host needs to make a decision whether it should replicate the full or compressed data which was explained in Section 4.1. The hypothesis here is that the full replication for certain *outdepth* costs more space but performs faster than the corresponding compressed replication. On the other hand, the importance of triples decreases at their *outdepth* increases, since their probability to contribute in queries workload decreases [5].

We now provide cost and benefit functions for each *outdepth* level δ and host i such that each host can make a systematic decision about the type of replication it should perform. The cost function at $outdepth = \delta$ reflects the fraction of the storage space which the host i would pay if this *outdepth* was fully replicated.

$$cost(i, \delta) = \frac{(|border(\delta, i)| - |border(\delta - 1, i)|) \cdot 8 \cdot s_t}{\grave{S}} \tag{2}$$

Where s_t is the size of a single compressed triple in Bytes given by Equation 1, and $\grave{S}$ is the size in Bytes of the currently remaining storage space of the original S input. The factor 8 is estimated practically from [1] as the ratio of full to compressed size. The benefit of host i performing full replication of the triples at $outdepth = \delta$ is related to the fraction of the triples that are expected to participate in the workload, which is inversely proportional with δ :

$$benefit(i, \delta) = \frac{1}{\delta} \cdot \frac{1}{R} \tag{3}$$

The factor R can be greater than 1 when the network performance is relatively poor, or the length of the queries is expected to be small.
For every partition i, we calculate the cost and benefit ratios starting with $\delta = 1$; we make a decision to build full-data replication if the benefit is greater than the corresponding cost, or otherwise consider building compressed replication, or stop if there is no more storage space left.

5 Load-Aware Partitioning and Replication Step

Initial partitioning of the RDF graph with METIS provides solid ground for the system to start executing queries while decreasing the amount of network data transfer between working hosts. Next, our system collects the executed queries in order to perform another partitioning which is based on the workload. In this step, we first update the existing METIS partitioning by performing optimization based on the available workload knowledge, we then support the METIS partitioning by providing efficient replications on multiple levels.

5.1 Weighted METIS

The cold partitioning which took place in Section 4, produces METIS partitions that have a minimum number of edges across the partitions, given that all the edges are equally weighted with 1. Since METIS provides the possibility of assigning numerical weights to edges of the input graph, we provide a weighting system that instructs the METIS with the more important edges in order to avoid putting them on the resulting partitions' cut. This weighting system is produced from the collected workload and its results. The key advantage here is that we remove the important vertices from the partition borders, and at the same time we consume no more space, because no new data is added, but it could require moving triples across hosts. The work of [10] compared between different graph partitioning algorithms that try to achieve balanced partitions in terms of the workload, but none of them directly considered the weighting provided by METIS to partition RDF graphs. In contrast, we define the function for each $e \in E$ as $w(e) = J \cdot (f(e) + f(p_e)) + \beta$ where

- $f(e)$ is the frequency of appearance of e in the workload queries answer Q_a,
- $f(p_e)$ is the frequency of appearance of p_e (the property associated with edge e) in the workload query answer,
- β is equal to 0 when the frequency of p_e in the data set is larger than a fixed threshold, or equal to 1 otherwise,
- J is the ratio between the average count of edges per vertex in the RDF graph and the average query frequency in the workload.

5.2 Building Global Query and Fragments Graphs

In this step we again employ the storage space S assigned by each host, by replicating border triples to support the METIS partitions. But we make use of the collected workload by generating full replication of those border triples which have more priority, and compressed replications for the other lower priority triples. For this we first normalize the workload, generate a global queries-graph, and then describe the algorithm we run to compute the border triples priority.

Workload Normalization

In a query workload Q, we first remove any non-frequent constants (excluding properties) and any variables found in each $q \in Q$ and replace them with single variable Ω. This normalizes the workload and avoids the generation of irrelevant small fragments. The item is considered non-frequent if its frequency is less than a normalization threshold. This threshold was left to the application case in WARP and Partout. We let the system find this threshold by determining the first statistical quartile of the items frequencies, such that we have a fully automated process. Recall from Definition 4 that each q is a set of triple patterns; the normalized workload can be defined as: $T_\Theta = \{(t_{\Theta 1}, f_1), (t_{\Theta 2}, f_2), .., (t_{\Theta k}, f_k)\}$.

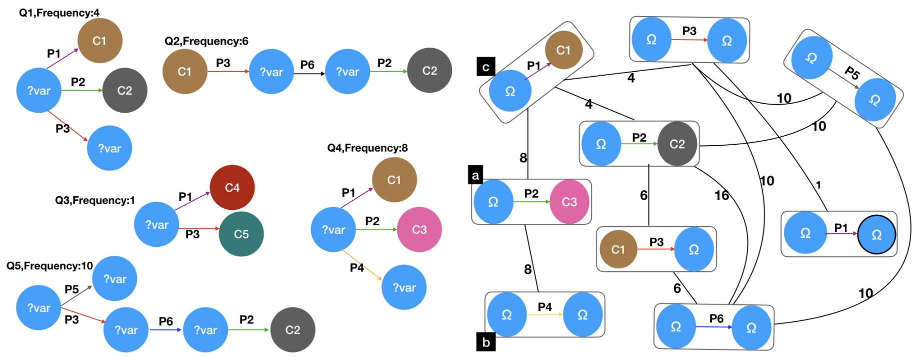

Fig. 1. Example workload (left) and global queries-graph (right)

The Global Queries-Graph

The normalized queries workload is converted into a global queries-graph by modeling each distinct triple pattern as a vertex. Fig. 1 shows an example workload and its global queries graph. An edge between two vertices in the graph means that the two triple patterns modeled by the two vertices occur together in one or more queries in the normalized workload. The edge weight represents the total count of this occurrence. We define the global queries-graph as:

Definition 5 (Global Queries-Graph). *For a normalized workload* $T_\Theta = \{(t_{\Theta_1}, f_1), (t_{\Theta_2}, f_2), ..., (t_{\Theta_m}, f_m)\}$*, the global queries-graph is* $G_p = \{V_p, E_p\}$*, where* $V_p = T_\Theta$ *and* $E_p = \{(t_{\Theta_i}, t_{\Theta_j}) \mid \exists (q_k, f_k) \in Q : t_i \in q_k \wedge t_j \in q_k\}$*. Each edge* $(t_{\Theta_i}, t_{\Theta_j})$ *in* G_p *is weighted with the total number of times that the normalized triple pattern* t_{Θ_i} *falls with* t_{Θ_j} *in the same query.*

Algorithm 1: Generate The Workload-based Replication

```
   input : A global queries-graph $Q_p = \{V_p, E_p\}$, and RDF graph $G = \{V, E\}$
1  for each host $i$ do
2      //Create a fragment $F_v$ for each $(v, f) \in V_p$, each $F_v$ is a set of assigned triples:
         $F = \{F_v \mid (v, f) \in V_p, F_v = \{d \in D \mid FragAssign(d) = v\}\}$ ;
3      for $\delta = 0$ to $\infty$ do
4          $D_t = \{(s, p, o) \in D \mid (s \in border(i, \delta) \wedge o \in border(i, \delta - 1)) \vee (o \in$
             $border(i, \delta) \wedge s \in border(i, \delta - 1))\}$ ;
5          if $D_t = \emptyset$ then
6              break;
7          end
8          for each $d' \in D_t$ do
9              for each $(t_\Theta, f) \in V_p$ do
10                 if $d$ matches $t_\Theta$ then
11                     $matchImpact \leftarrow matchImpact + f$ ;
12                     $\grave{q} = \{(t_{\Theta j}, f_j) \in V_p \mid \exists (t_\Theta, t_{\Theta j}) \in E_p\}$;
13                     $fragmentMatch \leftarrow fragmentMatch + getFragMatch(\grave{q}, t_\Theta, d')$;
14                     $FragNo \leftarrow FragAssign(d')$
15                 end
16             end
17             $F_{FragNo} \leftarrow F_{FragNo} \cup \{d'\}$ ;
18             $benefit(d') \leftarrow$
                 $benefit(i, \delta) \cdot (\frac{fragmentMatch}{maxRecordedfragmentMatch} + \frac{matchImpact}{maxRecordedMatchImpact})$;
19             $cost(d') \leftarrow cost(i, \delta)$;
20             if $benefit(d') > cost(d')$ then
21                 Build full replication of triple $d$ in host $i$ ;
22             else
23                 Build compressed replication of triple $d$ in host $i$ ;
24             end
25         end
26     end
27 end
```

Generating The Workload-based Replications

The global queries-graph contains the workload-based information which enables the recognition of the border triples that have more probability to participate in future queries. Replicating those triples increases the chance of local query executions. However, such probability is related to the *outdepth* of the border triple and to the engagement level of this border triple with the global queries-graph. The steps we follow in this stage are shown in Algorithm 1. Each host starts from its border and processes the triples at its neighbours which are located at *outdepth* $= 0$, then iteratively increases the *outdepth*. Those triples are given by set D_t at step 4 in Algorithm 1. We look at each triple $d \in D_t$, and compute two values: The first is *matchImpact* which is the accumulative frequencies of the normalized triple patterns in G_p that d matches; while the second value is the $fragmentMatch$ which gives an indication about the engagement level of D with G_p. This level is computed by the function $getFragMatch(\grave{q}, t_\Theta, d')$, which computes multiple splits of the triple patterns of $\grave{q}$, given that each split must contain t_Θ, and at least one other $t_{\Theta_j} \in \grave{q}$. The function finds the split l with the maximum number of triple patterns that d' matches, and returns the summation of all edges' weights between t_Θ and other triple patterns in l.

6 Conclusion

In this work we presented a novel adaptive partitioning and replication approach that can highly adapt to different levels of both workload and storage space. We report on a benchmark based on three types of queries and three types of storage levels in [1]. As a test set, we used the YAGO core[2] data set with more than 50M triples. We compare our system with three implementations based on WARP, Partout and Huang.

Acknowledgements. The authors would like to thank Deutscher Akademischer Austauschdienst (DAAD) for providing fund for research on this project.

References

1. Al-Ghezi, A., Wiese, L.: Space-adaptive and workload-aware replication and partitioning for distributed rdf triple stores (2018), under review
2. Galárraga, L., Hose, K., Schenkel, R.: Partout: a distributed engine for efficient rdf processing. In: Proceedings of the 23rd International Conference on World Wide Web. pp. 267–268. ACM (2014)
3. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: Triad: A distributed shared-nothing rdf engine based on asynchronous message passing. In: Proceedings of the ACM International Conference on Management of Data. pp. 289–300. ACM, New York, NY, USA (2014)
4. Hose, K., Schenkel, R.: Warp: Workload-aware replication and partitioning for rdf. In: IEEE 29th International Conference on Data Engineering Workshops (ICDEW). pp. 1–6 (2013)
5. Huang, J., Abadi, D.J., Ren, K.: Scalable sparql querying of large rdf graphs. Proceedings of the VLDB Endowment 4(11), 1123–1134 (2011)
6. Karypis, G.: Metis and parmetis. In: Encyclopedia of parallel computing, pp. 1117–1124. Springer (2011)
7. Margo, D., Seltzer, M.: A scalable distributed graph partitioner. Proc. VLDB Endow. 8(12), 1478–1489 (Aug 2015)
8. Padiya, T., Kanwar, J.J., Bhise, M.: Workload aware hybrid partitioning. In: Proceedings of the 9th Annual ACM India Conference. pp. 51–58. ACM (2016)
9. Peng, P., Chen, L., Zou, L., Zhao, D.: Query workload-based rdf graph fragmentation and allocation. In: EDBT. pp. 377–388 (2016)
10. Shang, Z., Yu, J.X.: Catch the wind: Graph workload balancing on cloud. In: Proceedings of the IEEE International Conference on Data Engineering. pp. 553–564. IEEE Computer Society, Washington, DC, USA (2013)
11. Wu, B., Zhou, Y., Yuan, P., Liu, L., Jin, H.: Scalable sparql querying using path partitioning. In: Data Engineering (ICDE), 2015 IEEE 31st International Conference on. pp. 795–806. IEEE (2015)
12. Xu, Q., Wang, X., Wang, J., Yang, Y., Feng, Z.: Semantic-aware partitioning on rdf graphs. In: Chen, L., Jensen, C.S., Shahabi, C., Yang, X., Lian, X. (eds.) Web and Big Data. pp. 149–157. Springer International Publishing, Cham (2017)
13. Zhang, X., Chen, L., Tong, Y., Wang, M.: Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud. In: Jensen, C.S., Jermaine, C.M., Zhou, X. (eds.) ICDE. pp. 565–576. IEEE Computer Society (2013)

[2] https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/