Space-adaptive and Workload-aware Replication and Partitioning for Distributed RDF Triple Stores

Ahmed Al-Ghezi and Lena Wiese

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

Abstract. The efficient distributed processing of big RDF graphs requires typically decreasing the communication cost over the network. This requires on the storage level both a careful partitioning (in order to keep the queried data in the same machine), and a careful data replication strategy (in order to enhance the probability of a query finding the required data locally). Analyzing the collected workload trend can provide a base to highlight the more important parts of the data set that are expected to be targeted by future queries. However, the outcome of such analysis is highly affected by the type and diversity of the collected workload and its correlation with the used application. In addition, the replication type and size are limited by the amount of available storage space. Both of the two main factors, workload quality and storage space, are very dynamic on practical system. In this work we present our adaptable partitioning and replication approach for a distributed RDF triples store. The approach enables the storage layer to adapt with the available size of storage space and with the available quality of workload aiming to give the most optimized performance under these variables.

1 Introduction

The sources that produce web-data are rapidly growing everyday, by the increase in the digital life applications. The requirements of analysis and querying of these accumulated data are moving in the same trend. The Resource Description Framework (RDF) is widely used to represent the pieces of data that describe the things in the web and their relationships. RDF data can be seen as a set of triples in the form (Subject, Predicate, Object). A set of triples forms naturally one graph, such that each triple in the data set is modeled as a directed edge which points from a subject to an object and is labeled with a predicate. This featured structure of RDF makes it suitable for representing big web data, which could be big enough to hit the limits of central systems resources, and provide obvious motivation towards distributed storing and processing. Many recent work in this decade targeted the distributed processing of RDF data, focusing mainly to provide more distributed storage space. Unfortunately, partitioning of big RDF data sets is not a trivial task since it can dramatically affect the efficiency of distributed query processing. SPARQL is the standard language used to query

RDF data. Each SPARQL query could be represented as a graph on its own with some vertices and/or edges represented as variables. The query execution is then the process of finding all the sub-graphs in the RDF-graph that match the query graph. However, if a query can find only a part of the graph locally, it has to collect all missing parts from other hosts. The latter process is usually a costly operation in typical networked hosts. To deal with this problem two main directions are usually followed:

- better partitioning strategy, and
- replication of selected important triples across hosts.

The analysis of a given workload could identify recognizable trends, and highlight the parts of the data set that have higher probability to be targeted together by future queries [9]. However, in practical systems it might be difficult to have an initial workload at system start-up, besides that the quality of any available workload in term of its recognizable trends is not assured, as it behaves practically as a variable on its own.
On the other hand, supporting the partitioning by replications increases the chance of a query finds the missing local graph's parts in the replicas [5]. However, the replication requires more storage space which is a very precious resource in a distributed system that is required to handle big data, and especially with RDF processing systems where the space is highly required to build more indices to boost the performance [7] . The replication approaches which are designed to save disk space can't directly give the full performance in case the system has plenty of storage space. The amount of available storage space is considered a variable in practical systems since it is directly related to the data set size, and the size of other data container in the system like indices and statistics tables. In this context, the partitioning and replication strategies that are designed to perform well under certain assumptions about the storage space and workload might not show good performance in different practical environments.

In this work, we present our partitioning and replication approach that is superior to related approaches with its adaption to any given workload quality *and* to the availability of storage space. We differentiate between two types of replications: full and compressed. The two types have different sizes and benefits. Our system optimizes the decision of making full, compressed or no replication on the level of a single triple, by comparing derived cost and benefit values highlighted by the workload and the availability of storage space.
We extend our prior work [1] by providing a solid experimental evaluation with respect to other related systems, besides the details of compressed replication and the triples workload engagement. The rest of this paper is structured as follows: We first review the related work in Section 2. We then explain our system's start-up behaviour in Section 3. Section 4 describes our approach to analyze the workload and derive numerical values for the triples engagement with the workload. Section 5 describes the basics of the system's adaptability to storage space. We show the practical evaluation of our approach in Section 6, and finally we conclude in Section 7.

2 Related Work

The problem of graph partitioning has been a topic of many recent works including works which considered the problem of RDF graph partitioning. J. Huang [5], WARP [4], and TriAD [3] used METIS as a baseline tool to statically partition the RDF graph. TriAD assigns the METIS partitions to hosts based on a hash function. Other works focused on the semantic found in the RDF data set; a recent work by Qiang Xu [11] clusters the classes of RDF data set by applying a page-rank based algorithm prior to partitioning. EAGRE [12] recognizes specific entities and group all the entities that belong to the same RDF classes. The work in [10] considers path partitioning by finding closed paths within the RDF graph, and assigns paths sharing merged vertices to the same partition. Margo et al [6] used another semantic-based concept which is edge partitioning; they generate an elementary tree from the RDF graph before partitioning it while trying to keep the general objective of reducing communication cost. Multiple works considered the workload when performing partitioning, however WARP [4] and Partout [2] had more distinguished approaches. Partout horizontally partitions the RDF data set into fragments inspired by the horizontal fragmentation of a classical relational table. It first normalizes and transfers the workload into a global queries graph preserving the connectivity and load impact information. The graph is used to generate fragments of matching triples from the data set. However, the result of Partout's partitioning is highly depending on the quality of the available workload in terms of its queries' frequency of appearance and on how it is fairly representing all parts of the data set. WARP [4] starts initially by statically partitioning the RDF graph by METIS, then uses a given workload to identify some important border triples to replicate. However, the approach treats all the workload queries whose frequencies are higher than a fixed threshold equally; this might give weak performance if the workload is varying and the storage space available for replication is limited.
A recent work of [8] identifies the properties which are queried together in a given workload, although that is embedded in the work of Partout [2]. Peng [9] scans the workload looking for frequent patterns, measured by the frequency of appearance. A fragmentation phase takes place by matching the frequent patterns with the data set. However, the concept used to generate the frequent patterns beside the workload normalization is still very similar to the min-terms used in Partout and WARP.

3 Initial Partitioning

At the time of system start-up, we typically have no assumption about the workload or the data set. In order for the distributed triple store to start working and collecting workload, it needs to initially partition the input data set without relying on the workload. The well tested METIS partitioner represents a very attractive choice at this stage, and it plays such role in systems like WARP [4]. METIS partitions the RDF graph into n partitions such that we have an

approximately minimum number of edges in the cut. Each of the n partitions is assigned to one host of the n hosts in the cluster. In order to execute a query, it should be sent to all working hosts. Each host tries to find all the sub-graphs that match the input query. However, we may pay communication cost whenever a query is long enough such that two or more partitions hold part of its complete results.

4 Workload Adaptability

After performing initial partitioning, the next step is to support it with replications of selected data. The objective is to increase the chance of local query execution and avoid paying any extra communication cost. The operation is limited by the availability of storage space. Thus, a wise decision about the specific data to replicate is very important. We describe in this section, our approach to analyze the workload, build a global queries graph, then measure the engagement level of the data-set triples with the workload. We will use this engagement level to systematically optimize the replication decision in Section 5.2.

Global Queries Graph

Instead of treating all the border triples equally, we make use of a collected workload to measure the importance of the border triples to the future queries, and group them upon their importance into fragments. Consider the sample workload in our running example Fig. 1. We first replace all of the variables, and non-frequent items in the workload (excluding properties) by a 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. The normalization prunes the first statistical quartile of the items in ascending order by their total frequencies; in the example, we prune "Kim" and ":person" and keep the other 5 item. We then convert the normalized query workload into a global queries graph G_p as shown in Fig.2. Each vertex in the graph models a normalized triple pattern. There is an edge between any two normalized triple patterns when they fall in one or more queries in the workload, while the respective edge's weight is the summation of those queries' frequencies. The projection of this graph on the data set gives the necessary information about the engagement levels of the data set triples with the workload. Each vertex in the global queries graph represents a fragment of matching triples in the data set. The concept of the global queries graph is used in Partout [2]. However, since we are interested in replicating the triples that are located at partitions' borders, we use the global query graph to build fragments of triples at those locations instead of building fragments for the whole data set.

4.1 Measuring The Engagement Level

For each border triple d at host h, we can measure two vital values, which we would then use to calculate the benefit of replicating it. The first value is

its distance from the border *depth* while the second value is its engagement level with the global queries graph G_p. Each vertex in G_p builds fragment of assigned triples. In order to measure the engagement level of a triple d, we first find how many vertices in G_p match d; then for each vertex we look at its neighbour vertices and check how d is engaged with previous triples assigned to their fragments – then we consider the maximum possible engagement. In our running example, suppose that we are checking a border triple d = (b:karl :name "Karl"); we can find that it is matching in G_p the vertices: v_1 = (Ω :name "Karl") and v_2 = (Ω :name Ω). We first take v_1 and check its neighbour vertices starting by (Ω :born Ω) call it v_3. We check if we have already a triple assigned to the fragment of v_3 and can be joined with d; such triple should be in the form (b:karl :born Ω). If the check is true, we add 10 – which is the weight of edge (v_1,v_3) – to the current engagement level of d. We repeat for other vertices connected to v_1. Then we repeat the calculation for v_2 and assign d to the fragment whose vertex gives the maximum engagement value. This method assigns to any triple a numerical engagement level, which we use in the next section to optimize the replication decision.

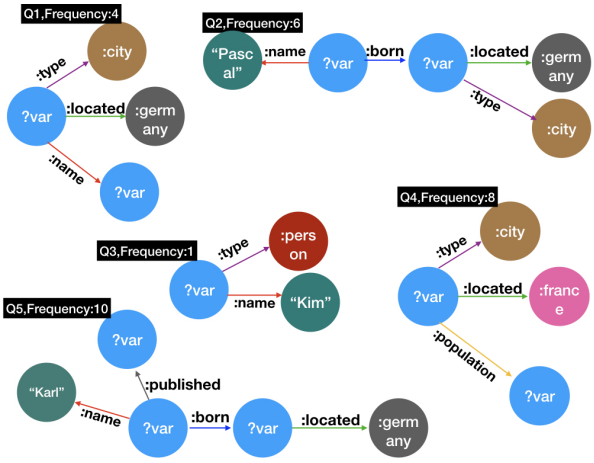

Fig. 1. Sample Queries Workload

5 Space Adaptability

Having more local replications increases the chance of local execution of query. However, the main limiting factor is the physical availability of storage space. Each host has a local main partition and needs to make a binary decision about each triple that is located in the neighbour hosts, whether to replicate this triple or not. We propose adding a third option which is to replicate the triple in

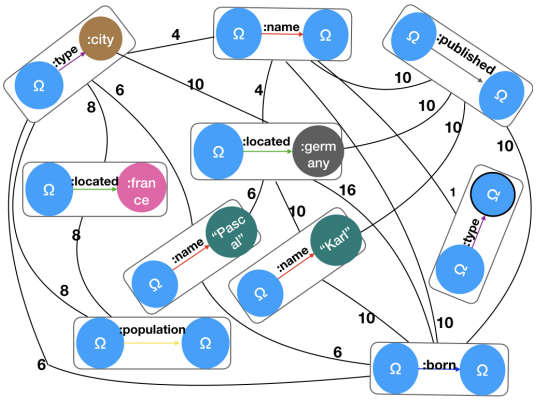

Fig. 2. Global Query Graph

compressed form. This highly increases the system flexibility to adapt to different levels of storage space.

5.1 Compressed Replications

In triple stores like RDF3X, a dictionary is used to map the textual URIs (found in the triples) to compressed numerical data; the mappings are then stored in different indices called dictionaries. We exploit these different data representations in the context of replication by defining 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 compressed numerical data. In order to show the effect of the two types on query execution we give the following example: Consider a query q which is composed of 4 triple patterns, and assume for simplicity that it was received for execution by 2 hosts (h_1 and h_2). Assume further that in the first round of execution, each host was able to find matches for one half of q, that is 2 out of 4 triple patterns. A second round of execution is necessary to produce the final query result, but it requires this time moving the first-round result of one host to the other. If h_1 has a full replication of the triples resulting from the first round in h_2, it could directly perform the join locally, do the dictionary reverse mapping and deliver the textual query result. The only practical reason for h_1 not having the required replication is the insufficient storage space. If h_1 has a compressed replication (the numerical form) of the data produced at h_2, it could perform the full second round locally, but requires sending the query result to the host where the dictionary entries are physically stored, in order to perform the dictionary reverse mapping and produce the textual final results. The size of the compressed data is mainly related to the number of triples in the data set, while the size of the textual data is related to the length of the text used. If G is an RDF graph and has a size of T triples, the minimum length of a 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:

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

Thus the size of the compressed data set is mainly relative to the number of the triples in the data set, with small logarithmic factor. We practically compared the sizes of both compressed and full data sets with different numbers of triple, and we find that the full data sizes is approximately 8 times greater than the corresponding compressed data. We use this value later in our cost functions in the next section.

5.2 Optimizing The Replication Decision

In Section 4.1 we obtained the engagement level of any triple with the global query graph which was built from the given workload. We use the engagement level of a triple d besides its *depth* (its distance from the partition border) to measure the importance of d for replication. The highly important triples are fully replicated, while the non-important triples are not replicated at all. The triples in between are having the possibility to have compressed replication. In order to have a systematic operation, we define cost and benefit functions that are smooth and flexible such that the system can directly make a decision about any triple. We start by finding the relative cost that a host i has to pay from its storage space in order to make a full replication to all the triples at depth δ in its neighbours partitions.

$$cost(i, \delta) = \frac{8D_i \cdot s_t}{\grave{S}} \tag{2}$$

Where D_i is the total number of triples at depth δ in all hosts other than host i, s_t is the size of a single compressed triple given by Formula (1), $\grave{S}$ is the current remaining storage space available for replications at host i, and the factor 8 represents the size ratio of full to compressed data.
On the other hand, the benefit for host i of replicating the triples at depth δ is given by the following.

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

The factor R can be more than 1 when the network performance is relatively poor, or the length of the queries is expected to be small. Our replication system works from the perspective of each host, iteratively considers the triples by their depth, and performs full replication to any triple which has more benefit than cost; otherwise it performs compressed replication. The cost-benefit functions are designed such that the cost increases as the host is running out of space, while the benefit of replicating a triple is related to its distance and to the extent of its relative engagement to the workload-derived global queries graph. If the system

happens to have abundance of storage space, the cost will resist the increase with deeper depth; if the system is very short in storage space, the cost would get rapidly high when the depth increases, such that the compressed replication would be favoured in such situations. If the host runs out of space at certain depth, the replication operation would halt and the triples located at deeper depth are excluded from any replication. This replication approach is highly adaptable with any available storage space and available quality of collected workload, and provides optimized support for the initial data set partitioning.

6 Evaluation

In order to evaluate our partitioning and replication approach, we developed a distributed version of RDF3X[1] and adopted its basic operators to support the distributed processing of SPARQL queries. Our test system is composed of four shared-nothing hosts. Each host maintains an adopted version of RDF3X. RDF3X maps the textual RDF data into compressed numerical form, before storing them in six indices. The compressed data fit directly in our model of compressed replications. One out of the four hosts contains a separate partitioning layer that communicates with the external METIS tool to achieve the initial static partitioning with the initial absence of any workload.
Then the system starts receiving and executing queries; it uses the queries with their frequencies to build its workload. The partitioning layer in each host analyzes the workload and builds the global queries graph which is then used to measure the engagement level of the border triples. The replication is then performed based on the benefit and cost functions described earlier in Section 5.2.
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. Since Partout doesn't explicitly describe a replication method, and for the sake of fair comparison, we followed the same fragment assignment method used in the partitioning to build replication within the available space.

We focused in our evaluation on measuring how the system practically responds to different levels of workload and storage space. Regarding the storage space, we made three system-level runs with three gradual limits on the available storage space, and let the four systems perform replications according to their approaches as much as the storage space limit permits. The *first* level run has less than 3% of storage space available for replication, while the *second* and *third* have 10% and 50%, respectively.
The second factor to consider is the quality of the workload in terms of the heterogeneity of frequencies. We performed two runs to the workload-based systems (excluding Huang which does not consider workload, and excluding the initial static partitioning periods from our system and WARP). The first level of workload is designed to be poorer in terms of its queries frequencies distribution and

[1] https://code.google.com/archive/p/rdf3x/
[2] https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/

its correlation to the future queries.
The last factor to consider is the type of queries used for evaluation. We generated random queries for testing purpose with different types and lengths, but we focused on *chain queries* since it is better showing the effect of the partitioning and replication approach. The longer the query, the more its chance to go beyond the partition border. In this regard, we classified the queries also into three levels upon their length and complexity, starting from level 1 short and simple queries towards level 3 with the longest and most complex queries. On these queries complexity levels we aggregate the output execution times.

Fig. 3 and Fig. 4 show the response times in milliseconds of our query execution over the four approaches: WARP, Partout and Huang, and our approach which we labeled APR. The Q.Type represents the query type explained above, while S.Level refers to the level of storage available in the system. Before executing the queries shown in Fig. 3, the system first sets up its partitioning and replication under the level 1 quality of workload, and with the three levels of storage space, giving a total of 3 full system setups. The queries in Fig. 3 are then subdivided into three levels based on their complexity. The poor workload affects directly the average response times of WARP and Partout especially on the low storage space level: here, WARP and Partout show performance similar to the static approach Huang. However, WARP enhances when there is enough storage space making use of its fixed-hop replications. Our system shows obvious better response time making use of its compressed replications which could still provide border replications in difficult storage space condition. Moreover, our system was less affected by the non-homogeneous workload, since its able to distinguish the highly referenced parts of the workload giving them higher priority to replicate. Fig. 4 shows the systems behaviour in case of a more representative and homogeneous workload, where WARP could perform well when the queries were not too long. Our system is still showing average better performance and was able to locally execute most of the queries. Huang shows no difference in execution with respect to the workload but was able to perform better when there was more storage space employed for static replication.

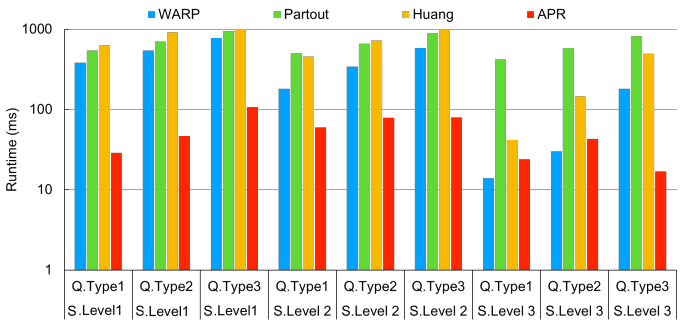

Fig. 3. Query Performance under Workload Level 1

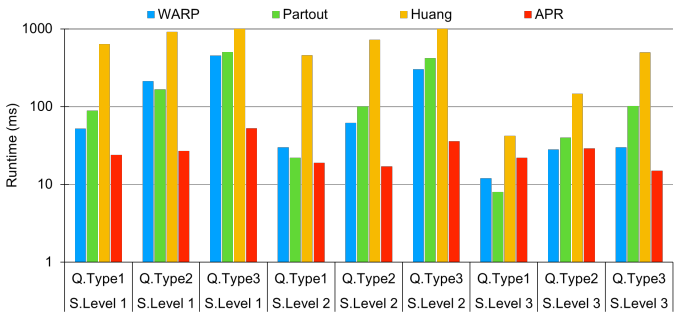

Fig. 4. Query Performance under Workload Level 2

7 Conclusion and Future Work

In this work we have presented a partitioning and replication layer that can be efficiently integrated in a distributed RDF triple store that is built on shared-nothing working nodes. Our approach is optimized towards the real world physical factors that impact the query processing performance of partitioned data. Our replication approach directly adapts itself with the availability of storage space. The replication approach shows privileged adaption with differing quality of workload compared to other approaches. As a future work we first consider adding more flexibility to the partitioning and replication approach to tune itself more smoothly on running time and while the workload is being collected and built – instead of the staged steps that we have considered in our work so far. We also consider enhancing the initial partitioning stage that needs currently to be fully carried out by one host, and optimizing the memory consumption to increase the size of the data set that the system can handle.

Acknowledgements

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

References

1. Al-Ghezi, A., Wiese, L.: Adaptive workload-based partitioning and replication for rdf graphs. In: Database and Expert Systems Applications. pp. 377–388. Springer International Publishing (2018)
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. Margo, D., Seltzer, M.: A scalable distributed graph partitioner. Proc. VLDB Endow. 8(12), 1478–1489 (Aug 2015)
7. Neumann, T., Weikum, G.: The rdf3x engine for scalable management of rdf data 19, 91–113 (02 2010)
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. 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)
11. 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)
12. 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)