UniAdapt: Universal Adaption of Replication and Indexes in Distributed RDF Triples Stores

Ahmed Al-Ghezi
ahmed.al-ghezi@cs.uni-goettingen.de
Institute of Computer Science, University of Göttingen

Lena Wiese
wiese@cs.uni-goettingen.de
Institute of Computer Science, University of Göttingen

ABSTRACT

The Resource Description Framework (RDF), which has become the standard data model to represent web data, is facing the exploding size of the web resources leading to difficulties in terms of maintaining and querying the data. Distributed RDF triple stores and their storage layers were already under research for a decade. While multiple systems tried to employ the workload to guide partitioning and replicating the data set, they are not able to find optimal levels for both the replication and local index storage as well as the main memory cached indexes.

In this paper we propose our novel unified optimization approach that enables a distributed RDF triple store to adapt its RDF storage layer in two aspects: the first aspect considers replication indexes, while the second aspect considers secondary and main memory indexes. Our system can dynamically analyze the workload, detect its queries trends, measure their effectiveness and apply them in triples' benefit functions. The system uses those functions to make fully automated decisions by either horizontally expanding each node's secondary storage by replication, or by vertically building more indexes. In the same context the system makes horizontal or vertical decisions about working nodes' main memory. The final objective of the optimization process is to decrease future query execution time.

CCS CONCEPTS

• **Information systems → Parallel and distributed DBMSs**; **Resource Description Framework (RDF)**; Storage replication.

KEYWORDS

Resource Description Framework, distributed triple store, workload space adaption

ACM Reference Format:
Ahmed Al-Ghezi and Lena Wiese. 2019. UniAdapt: Universal Adaption of Replication and Indexes in Distributed RDF Triples Stores. In *The International Workshop on Semantic Big Data (SBD'19), July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3323878.3325803

SBD'19, July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6766-0/19/07…$15.00
https://doi.org/10.1145/3323878.3325803

1 INTRODUCTION

The Resource Description Framework (RDF) is considered the standard model to represent the web data by describing their relations and semantics. The big scale and high heterogeneity of the web data is directly reflected in the requirements of the RDF triple store which is expected to maintain, process and efficiently query RDF data. Every "thing" or resource on the web can be described by a set of RDF triples, where each triple is in the form of (subject, predicate, object). The subject is any certain web resource, the predicate is relating that resource to an object, which in turn is modeling either another web resource or a fixed constant. The scale of current RDF triples is in the ranges of tens of billions, while some commercial RDF data sets have already reported going beyond 1 trillion triples [3]. Efficiently evaluating a SPARQL query over such a big data set goes beyond the capacity of central systems. However, maintaining the RDF data in a distributed environment raises new challenges with respect to partitioning, fragmentation and allocation of the RDF graph. The general objective of the previous operations is to avoid moving data across the network during query execution. However, given the complexity of the web resources' relations that are embedded in RDF data, the previous operations face complex problems. In this regard, a forth operation is required which is replication. Given the fact that replication requires more storage space, more questions arise: how much to replicate? and what to replicate?

Some works tried to answer the second question by focusing on saving storage space like WARP [5]; others acted more federated in this regard like Huang [6]. However, the practical value for any space constraint that should be imposed by the system on replications should be dynamic with respect to the availability of the storage space. This is because space availability is tightly related to the degree of space consumption by other highly needed system's structures especially the indexes.

Indexes are crucial structures for RDF. There are two basic structures of RDF indexes: hashed and sorted. The sorted one is more compact and flexible but typically slightly slower. What defines the index type is the triple elements that form the key. For example, in the sorted SPO index, triples are sorted first on subject, then on predicate, and finally on object. Based on this, the SPO can find the triples if their subject is known, or if their subject and predicate are known. Any practical RDF store should have at least two types of indexes where the subject and object can be used as first order keys.

Increasing the number of indexes would serve the performance and flexibility in executing queries, but requires more storage space. Basically a new full index could require storing a whole fresh copy of all triples in the data set. The Hexastore [12] and RDF3x [10] adopted the whole of six indexes representing the 3! combinations

of subject, predicate, and object. However, many other works considered such an indexing scheme a too huge consumption to storage space, and adopted a smaller set of indexes upon empirical observations to some workload samples.

Given that the constraint which limits building more indexes is the storage space, and that the optimal types of the indexes is recognized based on the workload, we can identify an optimization problem that aims to minimize the query execution time. Moreover, we can identify also the strong relation of this problem to the system's adaption for a proper replication amount, since they share the same storage space constraint as we have introduced earlier.

The storage space adaption is also extended to the main memory, as it is a much faster but smaller version of storage space. Given the expected big size of an RDF data set, the secondary storage is the practical place to store the data and its indexes. However, the usual SPARQL query workloads do not target all parts of the RDF graph evenly. In many cases, a workload may target only small hot spot of the data set [11]. Thus, the ability of the system to detect hot spots in the RDF graph and cache them in main memory would extremely enhance the performance. To summarize and formalize the problem that we are targeting in this work:

We are given a distributed RDF triple store with n federated working nodes. Each node has s_i bytes of secondary storage, and m_i bytes of main memory storage. We investigate how each node can make automated decisions about the following:

- The amount of secondary storage space assigned for replication s_r, and for indexes s_x; as well as the parts of data to replicate and parts of the data to put in each index.
- The parts of the data to load into main memory m_i, and the types of indexes that are optimal to contain them.

In order for the system to make the given optimized decisions for sake of better performance, it needs measurable expected benefits of data triples to a future query. The workload history has been effectively used in different data processing systems to anticipate users' future queries. Recently, SPARQL workload also gained attention in optimizing the storage layer in RDF triple stores [3–5]. The general direction is to detect frequent patterns, then assign higher priority for triples answering those patterns. However, such approaches assume previous knowledge about the data set and queries, besides they don't consider the space as the optimization constraint, which leads to unbounded behaviour with respect to different levels of space availability. Based on the above we summarize our contributions as follows:

- We present in this work a dynamic and fully adaptable novel approach to detect related frequent patterns that is independent of the data set and workload types, their quantity and their quality.
- We formalize dynamic triples benefit functions based on a mining approach, that allows the system to adapt its replication, secondary storage and main memory index schemes.

The rest of this paper is structured as follows: in the next section, we formalize our definition of the RDF graph and queries workload. In Section 3, we describe our method to detect workload trends using heat queries, and derive formulas for expected triples access rates. In Section 4, we derive the benefit functions on the level of triples and indexes, and use them to perform the optimization process in Section 5. We mention the most related work in Section 6 and provide experimental evaluation in Section 7. Finally we state our conclusion in Section 8.

2 PRELIMINARIES

Definition 2.1 (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 edge 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 2.2 (Queries Workload, Query Answer). A query q is a set of triple patterns $\{t_1, t_2, ..., t_n\}$; each triple pattern $(\grave{s}, \grave{p}, \grave{o}) \in q$ 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.

3 WORKLOAD TRENDS

The idea of detecting frequent patterns in a SPARQL workload was first used by Partout [3], then it was extended and used by multiple works like [1, 5, 11]. In [4], a so-called heat map is derived essentially from frequent patterns but with better modelling for frequencies and relations to future queries. We further extend the idea into heat queries, where related queries can be connected, information about patterns' direction are recorded and used to derive indexes usage and benefit. Moreover, we produce a more general version of a heat query by performing anonymization on its vertices, which allows its projection on more parts of the RDF graph as explained in the next subsection.

3.1 Heat Queries

Given a workload Q and an RDF graph G, we would like to find the probability of a vertex $v \in V$ to contribute to a future query $\grave{q}$. During the accumulative building of the workload, each time a query is executed, we mark the vertices in G that are part of its answer as $h_q \subseteq G$ and refer to h_q as the heat query. The vertices of h_q record two pieces of information: the count of this query as frequency or heat value, and the count of each index used (or to be used if the system doesn't have yet the optimal index for executing this query). If the answers of any two queries share one or more vertices, they would have their heat query combined into a single bigger heat query, such that the shared vertices would be hotter by getting the summation of heats of both heat queries. Figure 1 illustrates the proposed heat query evolution. The heat query would be bigger in size with more workload queries get combined independent of their order. To detect other parts of the RDF graph that give the same query trend as a certain heat query h_q, it needs to be "anonymized" by replacing all of its vertices with variables while keeping the properties (or the predicates). Similar

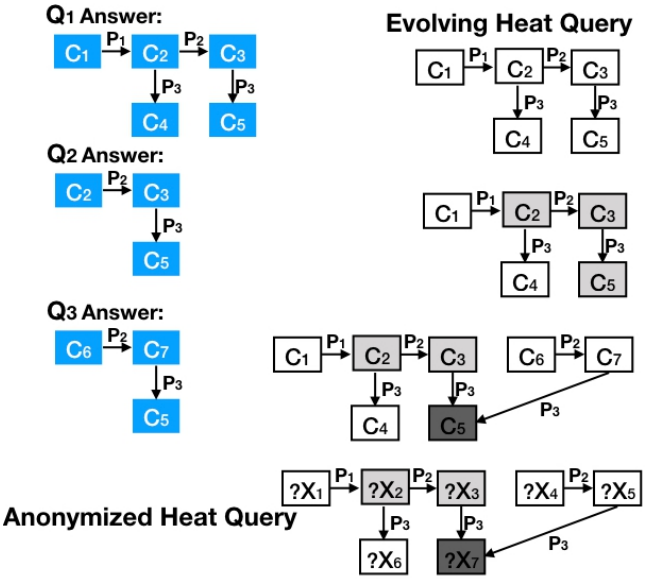

Figure 1: Heat Query Evolving from three Queries

approaches are used by [3] and WARP [5] to normalize a workload. The anonymized heat query can now be projected to other parts of the graph G, which allows us to detect similar queries trends. However, those trends are detected after anonymization, which raises the question: to what extent are the detected trends expected to contribute to any future queries? We denote this expectation as the effectiveness of the anonymized heat query. In order to enable the system to have real adaption with the workload, we don't assume fixed factors about the heat map distribution, but in contrast, we measure the effectiveness of any heat query h_q to the extent that h_q is being repeated in the workload.

The count of sub-graphs that match an anonymized heat query h_a is its frequency $freq(h_a)$. The value of $freq(h_a)$ gives us an important numerical measurement for the effectiveness of h_q. The higher $freq(h_a)$ the more we expect h_q to contribute in future queries:

$$effective(h_a) = 1 - \frac{1}{freq(h_a)} \tag{1}$$

if H represents a set of all anonymized heat queries in the system so far, then we can define $H_q(v)$ which returns for any $v \in V$ the heat query that v is associated with, or null if v does not belong to any heat query.

3.2 Formulating Triples Access Rate

In this section, we find the triple access rate, and use it to derive the benefit functions in Section 4. Given a query q that is to be executed on RDF graph G, the probability of a any vertex $v \in V$ to be part of the query answer q_a is uniformly given by:

$$p(v) = \frac{|q_a|}{|V|}$$

But when we take previous workload w into consideration, there is a frequency of appearance per vertex. Thus the above probability can be changed to:

$$p_w(v) = \frac{freq(v)}{\sum\limits_{\forall v_i \in V} freq(v_i)} \tag{2}$$

To keep the math compact, we assume a length of query answer equal to one. The value of $p_w(v)$ represents the usage factor or rate of access of v by the its frequency in the heat query. The rate of access of v as expected by the anonymized heat query is then given by:

$$access(v) = \frac{freq(H_q(v), v) \cdot effective(H_q(v))}{\sum\limits_{\forall v_i \in V} freq(v_i)} \tag{3}$$

where: $freq(h, v)$ is the frequency of v as given by the anonymized heat query h.

$access(v)$ in Formula 3 can be separately specified for each certain index type, by specifying the index type in $freq(h, v, index)$:

$$access(v, index) = \frac{freq(H_q(v), v, index) \cdot effective(H_q(v))}{\sum\limits_{\forall v_i \in V} freq(v_i)} \tag{4}$$

4 OPTIMIZING STORAGE FOR INDEXES, REPLICATIONS, AND CACHING

The heat query set defined in Section 3 provides information about the index usage in the previous workload. The anonymization and projection on the RDF graphs reflects further the trend of workload usage of the system's indexes. In this section we require the system to measure the benefit of having a vertex v in a certain index type x; and v can be in a node's secondary storage, main memory or in a remote node. The benefits formulas of this section will be used further in the optimization process in Section 5.

4.1 Measuring Index Benefit

The query optimizer receives a query and selects the best indexes to use out of its set of available indexes according to the query shape. If the optimal index is not available, the optimizer would use the nearest available index and issue a conversion plan that includes an extra filter. The cost of this conversation is related to the cost of the extra filter: the benefit of having the optimal index available per single access. Each heat query records per vertex the average of conversion cost from the used index to the optimal index, and the function $conv(x, v)$ returns the cost for index type x and vertex v. In order to have the above benefit with respect to the usage of v, we multiply it by the access function from Formula 4, then we set the formula for benefit of secondary storage index $\omega_i(index, v)$:

$$\omega_i(index, v) = access(v, index) \cdot conv(index, v) \tag{5}$$

4.2 Extending to Main Memory Benefits

The fact that the main memory is much faster than the hard disk, pushes many RDF systems to consider maintaining the whole RDF data set and indexes in the main memory [4, 14]. However, considering the huge and explosive size of RDF data, practical systems should use a hybrid approach – that is, to cache in main memory the most important parts of data. The workload and its trends play

again the best role of importance recognition in that context. Fortunately, we can extend our workload trends formulas to support selecting the most expected parts to contribute in future queries. However, some key differences should be taken into consideration. The secondary storage contains at least the SPO and OPS indexes for each triple. In the main memory the triple may exist in any type of indexes or does not exist at all. If the triple doesn't exist in any main memory index, then the benefit of caching it, is saving the time difference between accessing a triple in memory and on disk.

$$\omega_{mr}(v_d) = access(v_d) \cdot \mu \tag{6}$$

where μ is the time required to access a triple in memory.
On the other hand, if the triple exists in a main memory index, the benefit would be the conversation cost towards a more optimal index.

$$\omega_{im}(index, v_m) = access(v_m, index) \cdot conv(index, v_m) \tag{7}$$

Since we assumed no given existence of indexes, it could be that some memory-available indexes cannot be converted to the required index. For instance, if there is the memory-resident index SPO, while the required one is the OPS, then the system would have to use the disk index OPS, and as a result, the $conv(index, v_m)$ would be much higher. This would produce a higher benefit value in Formula 7 and hint the optimizer to favour building the required index which cannot be produced from available indexes.

4.3 Measuring Replication Benefit

A triple store that is distributed on several working nodes, needs to have its RDF graph partitioned. Each node would then have its share of data to be stored in the secondary storage in some indexes. To avoid synchronizing intermediate results, a replication is needed from the partition border [8]. The border is defined as all the vertices in the local partition, that have at least one edge coming from, or going to a remote partition that is located in another working node. During the execution of a query, any working node may require access to vertices at some distance from its border. However, this distance can't be more than the query length minus 1 (since that in the worse case the query starts from the partition border). On average half of the query length would be inside the local partition while the other half would be in the remote partitions. Thus the access factor for any vertex at distance d from the border is given by:

$$\delta(v) = \begin{cases} d \cdot \dfrac{2}{averageQueryLength} & \text{if } d \leqslant maxQueryLength \\ 0 & \text{otherwise} \end{cases}$$

Accessing a triple which is located in a remote node requires paying a delay cost proportional to the network delay per triple denoted as η. Thus we could rewrite the index benefit function in Formula 5 for a vertex that is located in a remote node by replacing the index conversion cost with the network access cost, and multiplying the access by the distance factor:

$$\omega_r(v_r) = access(v_r) \cdot \delta(v_r) \cdot (\eta - \Delta) \tag{8}$$

where Δ is the delay of accessing v in the secondary storage using the optimal index.

5 SYSTEM START-UP AND OPTIMIZATION PROCESS

In this section we describe how the system starts from scratch, and how it performs the optimization process for its storage space on the levels of main memory and secondary storage.

5.1 Starting up

The system contains n working nodes connected by a dedicated network. A node i has its own m_i memory storage and s_i secondary storage. The data set is one or multiple RDF files that can be downloaded at any node. The single RDF graph is partitioned into n partitions using METIS [9]. If one node cannot hold the whole data set, the partitioning node can still generate the METIS graph file, which is a relatively small text file containing only lines of numbers. Each line represents a vertex, and each number at that line represents the vertex that has an edge to this vertex. This file is much smaller in size than the original textual data set, and can be maintained by any node given that there is no index built at this point. At the end of this step each node will have its own data partition, besides a dictionary which maps the textual elements in the data set to compact integer code numbers. These codes will be used to convert the data set into two initial indexes: SPO and OPS. After having those indexes ready, the original data set files are no longer required and can be safely deleted.

5.2 Building The Workload

The system has the option to supply an initial set of queries to build initial workload, or start totally from scratch and build the workload from the received queries. After executing each query, the system saves it with its result to a temporary buffer, and continues executing any further query waiting in the queue. If the query queue is empty, the system would trigger the workload module which processes the temporary buffer and builds or updates the heat queries as stated in Section 3. If both of the temporary and queries queues are empty, the system triggers the optimization module. This module has two levels of execution as explained in the following sub-section.

5.3 Filling The Main Memory and Secondary Storage

The optimizer module makes sure that the main memory space is always filled with the parts of the data that are most expected to contribute in the next queries' execution. Given m bytes of main memory, there are always two options to fill them. Either to have triples from secondary storage cached into one type of indexes, or to replicate triples which are already in a main-memory index into another index. For this purpose, the optimizer maintains two limited-size priority queues, one for the disk resident vertices, and has conceptually the vertices ordered descendingly by their disk benefits as given by Formula 6, and a second queue where the memory vertices follow Formula 7 to arrange their benefits. A third priority queue is used to track what has been assigned to memory, but ordered reversely by benefits in ascendant manner. The optimizer would then choose greedily a more beneficial element

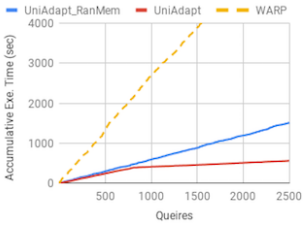

Figure 2: Adaption with first workload level

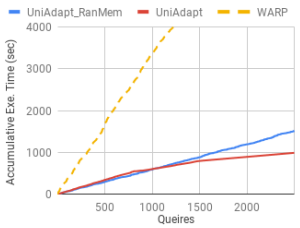

Figure 3: Adaption with second workload level

from either the first or second queue, and try to replace the head of the third queue, if it has less given benefit. The three queues are of limited size c (100k in our implementation) and contain only the first c best or worst elements according to their benefit functions. To avoid performance issues, the priority queue does not fully arrange itself with each insertion but performs the process on chunks of buffered elements. For the secondary storage, the optimizer also maintains three priority queues: the first one for the remote vertices; the second for local indexes, with priorities determined according to formulas 8 and 5 respectively; the third for the worst benefit vertices on indexes disk. The optimizer applies the same eviction process which has been described earlier for main memory.

6 RELATED WORK

Distributed RDF management systems have been analyzed under different research questions; see for example the survey in [13]. We focus in this section on the works that are most related to adaption. The first workload-aware partitioning was Partout [3]; the authors change a set of workload queries into a global query graph by removing non-frequent items. This graph is used to generate min-terms aiming to keep together the terms which are often queried together. However, the partitioning needs an initial workload to start, and it can easily degrade very badly, if the next queries are dissimilar to the workload queries. WARP [5] tried to avoid this problem by performing partitioning using static METIS, then using a workload to select important triples to replicate at the borders of the partitions. However, the adaption given by WARP is very limited as it sets a single frequency threshold, and treats equally any query beyond this threshold. In addition, the performance might be poor when the workload is small, has low quality, or when the system has shortage or abundance in storage space. Peng [11] mined the workload looking for frequent patterns with a concept being still similar to the Partout min-terms. Chameleon-db [2] used the workload to minimize the cost of intermediate results joining by using a decision tree. AdPart [4] partitions the data set by hashing the subjects of data set triples, and loading the whole data in main memory. The system monitors the workload and issues an eviction process upon some thresholds. However, maintaining big data-sets in main memory might not always feasible in practical systems. In contrast to all these works, our system exploits all available storage resources and involves them in a universal adaption approach.

7 EVALUATION

We evaluated our adaptive system on a cluster of 4 working nodes connected by dedicated network. We used the Billion Triple Challenge data set (BTC14) [7]. We assumed no workload available at system start-up, which makes each node follow the what we call "zero protocol" by partitioning the data using static METIS, performing replication with distance 2 from the partition border, and using the remaining disk space to build indexes in the order: SPO, OPS, POS, SOP, OSP, PSO. The main memory caches the indexes in the same given order to the maximum level as its free size allows.

7.1 Workload Generation

The main objective of our system is to adapt its available storage with the workload to the best performance of queries execution. It continuously uses the previous query workload as a training set to predict the next behaviour. In order to test this adaption and its behaviour under different workload quality levels, we generate multiple workloads each with different properties. In this context, we define the properties of the workload by three parameters:

- Size of queries: defined by two sub-parameters: its volume which is its vertices count, and the query's length, which is the maximum distance (minimum number of hops) found between any of its vertices.
- Quality level: we denote a workload as having a higher quality level when it has more obvious detectable trends. For testing purpose, the generated workload repeats a vertex from the data set by a rate that follows a normal distribution. The standard deviation would be the quality metric of the queries trends within the generated workload. The smaller is the standard deviation the better are the query trends.
- Rate of arriving: the number of queries per second that the system receives. We assumed a Poisson distribution to model the system's traffic of queries, as it is widely used to model the traffic received by servers.

7.2 Workload Adaptation

In order to test the system's adaption towards a stream of workload's queries, we generated two levels of workload. The first has the best quality with repetition of 40% and standard deviation of 1. A second level workload is also generated with repetition of 20% and standard deviation of 4. We tested the adaption of UniAdapt versus WARP and UniAdapt_RanMem which randomly caches indexes from the hard disk to main memory. Within the first level workload in Figure 2, UniAdapt was able to detect the workload trends, and adapt its memory indexes within the first 700 queries. In lower workload quality in Figure 3, the UniAdapt tolerates the low quality of queries' trends and performs similar to random approach up to 1500 queries, where it starts to show obvious better adaption. The WARP showed a linear behaviour with respect to both workload levels but with lower slope when it had better workload allowing it to better decrease the network communication costs.

7.3 Storage and Workload Adaptation

In this part, we deeply measure the adaption of the system with the workload on a range of main memory availability. For this purpose, we define a main-memory to data-set size ratio with range between

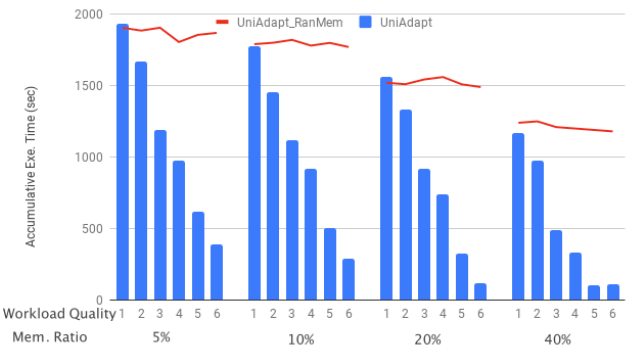

Figure 4: Adaption to workload with respect to memory availability

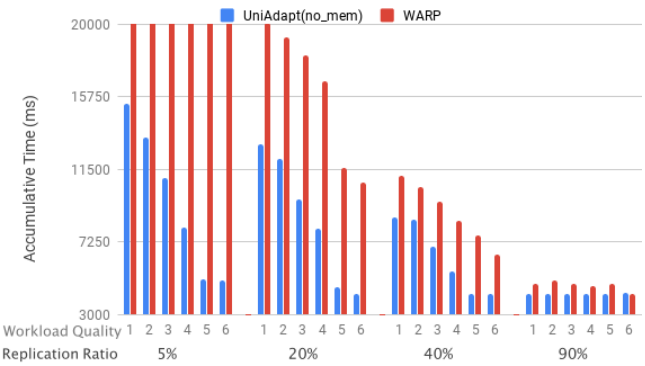

Figure 5: Adaption of remote replication with workload

(5%–40%). For each ratio level, we took six levels of workload quality and aggregated the accumulative execution time of 2500 queries on UniAdapt and UniAdapt_RanMem. The results of both approaches are shown in Figure 4 . UniAdapt maintained general shape of adaption with the workload in all memory levels, and recorded higher exploiting of memory at 40% ratio. At 5% memory ratio, UniAdapt was able to decrease the query execution time at higher workload quality by replicating highly referenced triples in memory in spite of its limited size with respect to the data set size.

7.4 Replication Adaption

In this part we consider testing the replication which each node needs to perform at its partition's border. To have clear evaluation of this part, we disabled the memory part of UniAdapt and compared it against an adopted version of WARP. Although WARP has static and workload-aware replication levels, it doesn't explicitly describe setting the amount of this static part. For our testing purpose, we allow this static part to grow as much as the replication space allows. Figure 5 shows the replication-layer adaption of two systems with the workload under the constraints of 4 levels of replication ratio. The replication ratio is the percentage ratio of replication that a node can perform to the total required amount. Within the 5% region, UniAdapt was able to balance its replication's needs with its local indexes needs. It made use of the workload and triples' distance to evaluate their benefits and importance, which is reflected in its highly shown performance. The adaption of UniAdapt is still superior in the remaining regions of replication ratio until the 90% region, where the WARP and UniAdapt shows similar behaviour, typically because of the high level of replications which approximately eliminates the needs of any communication.

8 CONCLUSION

In this paper we presented a novel approach that performs universal storage layer adaptation with the workload and the storage space. This enables the distributed RDF triples store to efficiently perform on different data sets types and sizes, and within any resources availability. Our system requires no fixed setting and can automatically settle values to replication and indexing schemes, besides efficiently caching triples to main memory. Our experimental evaluation supports the highly adaption behavior of our system.

ACKNOWLEDGMENT

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

REFERENCES

[1] Ahmed I. A. Al-Ghezi and Lena Wiese. 2018. Adaptive Workload-Based Partitioning and Replication for RDF Graphs. In *DEXA*.
[2] Günes Aluc, M. Tamer Özsu, Khuzaima Daudjee, and Olaf Hartig. 2013. chameleon-db: a Workload-Aware Robust RDF Data Management System. Technical Report CS-2013-10, University of Waterloo. (09 2013).
[3] Luis Galárraga, Katja Hose, and Ralf Schenkel. 2014. Partout: A Distributed Engine for Efficient RDF Processing. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14 Companion)*. ACM, New York, NY, USA, 267–268. https://doi.org/10.1145/2567948.2577302
[4] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. 2016. Accelerating SPARQL Queries by Exploiting Hash-based Locality and Adaptive Partitioning. *The VLDB Journal* 25, 3 (June 2016), 355–380.
[5] K. Hose and R. Schenkel. 2013. WARP: Workload-aware replication and partitioning for RDF. In *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*. 1–6. https://doi.org/10.1109/ICDEW.2013.6547414
[6] Jiewen Huang, Daniel J Abadi, and Kun Ren. 2011. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment* 4, 11 (2011), 1123–1134.
[7] Tobias Käfer and Andreas Harth. 2014. Billion Triples Challenge data set. Downloaded from http://km.aifb.kit.edu/projects/btc-2014/.
[8] Zoi Kaoudi and Ioana Manolescu. 2015. RDF in the clouds: a survey. *The VLDB Journal* 24, 1 (01 Feb 2015), 67–91. https://doi.org/10.1007/s00778-014-0364-z
[9] George Karypis. 2011. METIS and ParMETIS. In *Encyclopedia of parallel computing*. Springer, 1117–1124.
[10] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* 19, 1 (01 Feb 2010), 91–113. https://doi.org/10.1007/s00778-009-0165-y
[11] P. Peng, L. Zou, L. Chen, and D. Zhao. 2019. Adaptive Distributed RDF Graph Fragmentation and Allocation based on Query Workload. *IEEE Transactions on Knowledge and Data Engineering* 31, 4 (April 2019), 670–685. https://doi.org/10.1109/TKDE.2018.2841389
[12] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB* 1 (08 2008), 1008–1019. https://doi.org/10.5167/uzh-8938
[13] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. 2018. RDF Data Storage and Query Processing Schemes: A Survey. *ACM Comput. Surv.* 51, 4, Article 84 (Sept. 2018), 36 pages. https://doi.org/10.1145/3177850
[14] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A distributed graph engine for web scale RDF data. In *Proceedings of the 39th international conference on Very Large Data Bases (PVLDB'13)*. VLDB Endowment, 265–276. http://dl.acm.org/citation.cfm?id=2488329.2488333