Universal Storage Adaption for Distributed RDF-triple Stores

Ahmed Al-Ghezi and Lena Wiese[0000−0003−3515−9209]

Goethe University Frankfurt, Robert-Mayer-Str. 10, 60629 Frankfurt am Main, Germany
{alghezi,lwiese}@cs.uni-frankfurt.de

Abstract. The publication of machine-readable information has been significantly increasing both in the magnitude and complexity of the embedded relations. The Resource Description Framework (RDF) plays a big role in modelling and linking web data and their relations. Dedicated systems (RDF stores / triple stores) were designed to store and query the RDF data. Due to the size of RDF data, a distributed RDF store may use several federated working nodes to store data in a partitioned manner. After partitioning, some of the data need to be replicated to avoid communication cost. In order to efficiently answer queries, each working node needs to put its share of data into multiple indexes. Those indexes have a data-wide size and consume a considerable amount of storage space. The third storage-consuming structure is the join cache – a special index where the frequent join results are cached.
We present a novel adaption approach to the storage management of a distributed RDF store. The system aims to find optimal data assignments to the different indexes, replications, and join cache within the limited storage space. To achieve this, we present a cost model based on the workload that often contains frequent patterns. The workload is dynamically and continuously analyzed to evaluate predefined rules considering the benefits and costs of all options of assigning data to the storage structures. The objective is to reduce query execution time. Our universal adaption approach outperformed the in comparison to state-of-the-art competitor systems.

Keywords: RDF · Workload-aware · Space-adaption.

1 Introduction

The Resource Description Framework (RDF) [10] has been widely used to model the data on the web. Despite its simple triple-based structure (each triple consisting of subject, predicate and object), RDF showed a high ability to model the complex relationships between the web entities and preserve their semantic. It provided the scalability that allowed the RDF data to grow big from the range of billions to the range of trillions of triples [17]. As a result, RDF data experienced a rapid increase both in the size and complexity of the embedded relationships [6]. That emphasises more challenges on the RDF triple stores in

terms of managing and structuring that complex and huge data while still showing acceptable query execution performance. These stores have to have multiple data-wide indexes, cache, and replications (in case of distributed triple stores). The highly important constraint to be considered in managing these structures is the storage space. Thus, many research works tried to optimize their usage by depending on workload analysis (see Section 3). However, they focused on the problems of indexes, replication, and cache separately, despite the fact that the three optimization problems are basically modeling an integrated single optimization problem. They share the same constraint (the storage space) and objective function (maximizing the performance).

We propose a universal adaption approach for the storage layer of RDF-triple store. It uses the workload to evaluate the benefits of indexes, replication, and cache and selects the most beneficial items to fill the limited storage space.
The rest of this paper is organized as follows: Section 2 provides descriptions of the adaption components. Section 3 reviews the related work. In Section 4 we formulate our cost model. In Section 5 we describe our method of analysing the workload. In Section 6 we derive the benefits of the adaptions components. We provide the universal adaption algorithm in Section 7. We practically evaluation the approach in Section 8. Finally we state our conclusion in Section 9.

2 Background

RDF Indexing. The most important data structure in an RDF store is the index. Systems like RDF-3X [16] and Hexastore [20] decided to build the full set of 6 possible indexes to have the full flexibility in query planning. This strategy fully supports the performance at the high expense of storage space. To deal with this high cost, other RDF stores preferred to choose only a limited number of indexes. These indexes are chosen based on some observations of the workload and the system has to live with it. However, the storage availability and workload trends are variable parameters. Thus fixed prior decisions about the indexes might be very far from optimal. Instead, our adaptable system evaluates the status of the workload and space at runtime and adjusts its indexing layer accordingly.

Replications. Due to the huge size and relation complexity of the RDF data, many RDF stores moved towards distributed systems. Instead of a single node, multiple federated nodes are used to host and query the data. However, the RDF data-set needs to be partitioned and assigned to those working nodes. Due to the complexity of this operation, replications are often needed to decrease the communication cost between the nodes. However, those replications may require a lot of storage space, and RDF stores had to manage this problem by trying to select only limited data for replication with the highest expected benefits. That benefit is derived from the workload and the relative locality of the data. However, all of the related works either assume the existence of some initial workload, or fixed parameters and thresholds which are not clearly connected or calculated from the workload. Moreover, the replication and indexes share the

same storage space, and the space optimizing process needs to consider both of them in the same process.

Join Cache. Executing a SPARQL query often requires multiple costly join operations. The size of the join results could be much smaller than the processed triples. Moreover, the real workload typically contains frequent patterns [17]. That makes a join cache very beneficial to the performance. However, it consumes a lot of storage space. Since the cache shares the same storage space of indexes and replication, and shares the same objective of query performance, it should be integrated into the same optimization problem.

Workload Analysis. RDF stores have to manage their needs to indexes and replications with the limited storage space. Historical workload played a vital role in such management [1, 17]. The analysis of the workload can be classified into *active* and *passive*. The active analysis is carried out on a collected workload aiming to derive its trends, detect future behaviour, then adapt its structures accordingly. However, such an adaption could highly degrade the system performance if the workload does not contain the expected frequent patterns. To avoid this problem, many systems preferred making fixed decisions about their indexes and replications upon passive analysis to some workload samples to draw average behaviour. The systems have to live with these decisions on any status of workload and storage space.

Universal Adaption. Instead of separate space optimization towards replication, indexes, or cache, we put the three types of structures into a single optimization process. The system chooses the most beneficial options to fill its limited storage space. We define a single cost model for the three types based on workload analysis.

3 Related Work

RDF-3X [16] is one of the first native RDF store. it uses an excessive indexing scheme by implementing all the 6 possible index permutations besides extra 3 aggregate indexes. The six-indexes scheme was also by Hexastore [20]. To decrease the storage overhead, RDF-3X uses a dictionary [4], where each textual element in the RDF data set is mapped to a small integer code. The **H-RDF-3X** system by **Huang et al.** [12] was the first distributed system that used a grid of nodes, such that each node is hosting an RDF-3X triple store. The data is partitioned using METIS graph partitioning [13]. To reduce the communication cost, H-RDF-3X forces uniform k-hop replication on the partitions' borders. Any query shorter than k is guaranteed to be executed locally. Unfortunately, the storage overhead of the replication increases exponentially with k, and H-RDF-3X did not provide any systematic method to practically calculate the optimal value of k. **Partout** [7] implemented workload awareness on the level of partitioning. It horizontally partitions the data set inspired by the classical approaches of partitioning relational tables [3]. The system tries to assign the most related fragments

to same partitions. Unfortunately, the results of such a partitioning are highly affected by the quality of the used workload. It could end up with small fragments representing the workload and a big single fragment containing everything else. **WARP** [11] proposed to use a combination of Partout and H-RDF-3X. Initially, the system is partitioned and replicated using the H-RDF-3X approach with a *small* value of k. Then, it uses workload to recognize the highest accessed triples for further border replications. Besides lacking the methodology to determine k, WARP supposes sharp threshold between frequent and non frequent triples. **Peng** et al. [17] proposed a partitioning approach inspired by Partout [7] but supported by replications. **AdPart** [9] is an in-memory distributed triple store. It aggressively partitions the data set by hashing the subject of each triple. As this is known to produce high communication costs, AdPart proposes two solutions. The first is by updating the dynamic programming algorithm [16, 14, 8] that is used to find the optimal query execution plan, to include the communication cost. However, this algorithm depends on the accuracy of the cost estimation which is already a challenging issue regarding calculating the optimal join plan in a centralized system like RDF-3X [16]. The second solution to the communication cost problem is by adding workload-driven replications. AdPart collects queries at runtime, builds the workload, and adapts its replications with time dynamically. Yet, AdPart requires a fixed setting of a frequency threshold that is used to differentiate between frequent and non-frequent items, making it a only a semi-automated system. **TriAd** [8] performs hash-based partitioning on both the subject and the object. That allows it to have a two-way locality awareness of each triple. Similar to AdPart, TriAd employs this to decrease the high communication cost caused by the hash partitioning. Aluç et al. [2] uses Tunable-LSH to cluster the workload and assigns the most related patterns to the same page. However, the approach does not count for the distributed replication and join cache.

4 Our Flexible Universal Cost Model

Our system aims to make its storage resources adaptable with the current status of the space and workload. The data set is divided into a set of units called the *consumers*. Each consumer may be assigned to a storage resource equal to its size based on one out of the different index options. The goal now is to find an optimal assignment based on a function that calculates a benefit for each setting. The optimization problem can be reduced to the Knapsack problem [5], where the local storage space is a knapsack of size n, which we want to fill with the most beneficial assignments (the items). However, since that the size of the assignments is small with respect to the total storage size, we may relax the condition of requiring a totally filled knapsack. That allows the problem to be solved greedily (instead of a more costly dynamic programming [15]). That is carried out by dividing the *benefit* of each item by its size and greedily filling the knapsack with most beneficial items. To reduce our model to this problem we need to derive the benefit of each assignment. To achieve the universal adaption,

we derive such benefits for indexes, join cache, and replication. However, the effective benefit is related to how often the system is going to use that assignment. We call this the access rate ρ and calculate it from the workload analysis. Based on this, the effective benefit of each assignment g that has a performance benefit η, and an access rate ρ is given by the following:

$$benefit(g) = \frac{\eta(g) \cdot \rho(g)}{size(g)} \tag{1}$$

The size in the above formula is given in number of triples that are affected by the assignment. By applying the formula on a set of assignment options, we may sort them and select the most beneficial option. In the next sections, we describe detecting the access rates and deriving the performance benefits

5 Workload Analysis

The aim of our workload analysis module is to find the access rate for each option of storage assignment – that is, ρ in Formula 1. First, we generate from the workload *general access rules*, which measures the average behaviour (e.g. the average index usage). Second, we detect the frequent items in the workload using a special structure called *heat query* resulting in a set of *specific rules*. The system actively measures the effectiveness of these rules, and prunes the impact of the rule of low effectiveness. By using this approach, our system gets the benefits of frequent patterns in the workload, and avoids their drawbacks by relying on average workload behaviour.

5.1 Heat Queries

A workload is a collection of the previous queries. However, we need to store the workload in a structure that keeps the relationships between the queries as well as their frequencies. A heat query is inspired from the concept of a *heat map* but instead of the matrix of heat values, we have a graph of heat values representing access rates. The workload is then seen as a set of heat queries. The heat query extends the original concept of *global query graph* originally proposed by Partout [7]. Each heat query is implemented as hashed map, where the key is a triple pattern and the value is a statistics object representing the frequency of appearance for that pattern, and the used indexes to evaluate it as well as performance-benefits values explained in the Section 6. $heatQueryAccess(v, \chi)$ is a function that returns the heat value of $v \in V$ for the index χ in the heat queries set.

5.2 Heat Query Generation

We explain in this subsection the generation of the heat query set out of a workload Q and an RDF graph G illustrated in Figure 1. Each time a query q is executed, it forms a new heat query h, with heat (frequency) set to 1 for

its vertices. Next, h is either added to the current heat query set H or combined with H, if there is a heat query $h_i \in H$ that has at least one shared element; the shared element is either a single vertex or an entire triple pattern. When combining two heat queries, the shared vertices become "hotter" by summing up the heats of the heat query graph and the new heat query. The combining process is shown in Figure 1 for a workload $\langle Q_1, Q_2, Q_3, Q_4 \rangle$. When Q_2 is received, it makes the matching part (to which Q_2 is combined) of the previous heat query hotter – which is illustrated by a darker color. The same applies for Q_3 and Q_4. Any variable in the query (here $?x$, $?y$ and $?z$) is replaced ("unified") by a single variable $?x$ to allow the variables to be directly combined. Note that in the case that a variable subsumes a constant, both the constant and the variable exist as separate vertices in the heat query. This also happens when Q_4 is combined. It increases the heat value of C_1 in Figure 1 and creates a node of variable $?x$ with a heat value equal to 1. By this process, a heat query would be bigger in size with more workload queries getting combined regardless of their order in the workload.

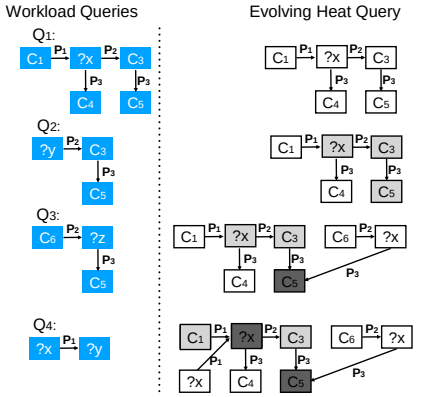

Fig. 1: Heat query evolving from four queries

6 Elements of Adaption

In this section, we describe the performance benefits of the indexes, join cache, and replication. We use these benefits besides the access rates and the storage cost to perform our universal adaption in Section 7.

6.1 Indexes

In any typical key-value RDF store, the data resides basically in indexes. An index is implemented as a hash table. For instance, the SPO index stores all the triples by hashing them on the subject. We can get a set of all triples that match a given subject by one lookup operation. This set is ordered on the predicate then on the object. Thus we can search that set for a certain predicate in logarithmic time. That index is optimal to answer any triple pattern[1] in the form (s_1,p_1,$?o_1$). However, if the triple pattern is in the form (s_1,$?p_1$,o_1), then we can

[1] According to SPARQL syntax $?o_1$ is a variable and others are constants or literals.

still use SPO to answer it, but with an extra linear search for o_1 within the set returned from the hash-table lookup operation on s_1. We refer to this extra cost as $scanLookupTime$ and is recorded in the heat query as performance benefits to the missing index. Another performance benefit of indexes comes from join. A SPARQL query is typically composed of multiple triple patterns that require join operation. The join planner selects a potential optimal join plan given that it implements all the possible 6 indexes. However, in case some indexes are missing, the query can still be executed using a sub-optimal join plan (i.e. the best plan that uses only available indexes). We refer to the cost difference between the two plans as $treeTime$. That cost is considered as performance benefits accounted for the missing indexes for the triple patterns been joined.
The performance benefit of a triple pattern t in an index χ is then the summation of the both given benefits:

$$\eta_{idx}(t, \chi) = scanLookupTime(t, \chi) + treeTime(t, \chi) \tag{2}$$

6.2 Join Cache

Besides the six basic indexes, a triple store can have more cache-indexes to speed up the join process. However, those indexes require even more storage space. Fortunately, our adaptive system can measure and compare the cost-effectiveness of the cache indexes with other indexes and choose to build them only for the data that delivers higher benefits with respect to other indexes. We use two cache indexes: PPX and $typeIndex$. The PPX is hashed on two predicates. Given that the predicate is mostly constant in any triple pattern [9], PPX can store the results of almost any two joined patterns. The $typeIndex$ is especially helpful in the queries that heavily use the predicate "*type*" like the LUBM benchmark [18]. Consider for instance the query: *(?x :graduatedFrom ?y. ?x :type :student. ?y :type :university)*. Its result can be cached in $typeIndex$ with the key (*:student,:graduatedFrom,:university*), and the benefit is the value of the saved join cost. Similar to the basic indexes, our system records the benefit and the usage values of the cache indexes in the heat queries and retrieves them at the adaption time.

6.3 Replication

In a federated distributed triple store with n working nodes, the system needs to generate at least n partitions out of the global RDF graph. For this purpose, a graph-based partitioning approach is widely used based on graph min-cut algorithms. The aim is to decrease the communication cost among the resulted partitions [12, 11, 21, 8, 19]. However, the problem is shifted to the border region where the edges are connecting multiple partitions. That problem is overcome by replication in [12, 11] at the cost of more storage space. Our adaptive system integrates the replication decision with the indexes by modeling it in the cost model 4. For that, we need the benefit, access rate, and size. The performance

benefit of replicating a block of triples is saving the communication cost of transferring the block over the network. Such a cost is related to the network speed and status. We derive the access rate of replications using the workload analysis module explained in Section 5. From the perspective of a certain working node i, the general access rate to a remote triple is related to its distance from the border. This can be formulated in the following:

$$p_{rem}(v,i) = \frac{1}{outdepth(v,i)} \cdot p_{border} \tag{3}$$

Where $outdepth(v,i)$ is the distance to reach vertex v (minimum number of hops) from the border of partition i, and p_{border} is the probability of a query at partition i to access its border region. The value of p_{border} is initially set to 1, but is going to be further updated depending on the workload by counting the rate of accessing the border region by all the executed queries in the system so far. This value is related to the average length of the query. The longer the query the more its probability to touch the border region. Next, we derive a specific access rate for replication, again from the heat queries. This is achieved by recording the replication usage rate for each triple pattern which requires border data. That allows the heat query to extend to the remote data over the border region. Finally, the aggregation phase takes place. Any remote triple that resides in index χ gets the following aggregate access value from node i:

$$repAccess(v,i) = p_{rem}(v,i) + heatQueryAccess(v,\chi) \tag{4}$$

7 Universal Adaption

We have formulated the cost model of the indexes, replications, and join cache in terms of benefits, access rates, and storage costs. Our optimizer builds their statistics during query execution in *stat phase*. It performs an *adaption phase* using Algorithm 1 at each node. The input to the algorithm is two sets of operation rules. We define an operation rule in the following:

Definition 1 (Operational Rule). *An operational rule is defined as* $\varpi_{op} = (\chi, s, a, \Delta)$*, where* χ *is an index,* s *is a set of patterns that defines a set of triples* D*,* a *and* Δ *are functions that assign an access and benefit values to each* $d \in D$*. According to Formula 1, the benefit of each source can be calculated. We refer to* $\bar{s}$ *as the source with the maximum benefit in* s*, and:*

$$b(\varpi_{op}) = \frac{a(\bar{s}) \cdot \Delta(\bar{s})}{size(\bar{s})}$$

We define one rule for each basic and cache index for the local data, and another for the replicated data. Each rule is put in the assigned rules set R_a and the proposed rules set R_p. R_a represents what is already assigned to memory and R_p represents the proposed. The algorithm starts by a loop which first process the workload stats stored in the heat queries in Line 2, and updates the access

functions of the rules accordingly. Line 3 sorts ascendingly the patterns of each $r_a \in R_a$ by relative benefits such that $\bar{s}$ is the pattern at the top. The sort is descendingly for R_p. The relative benefit is calculated using Formula 1 and Definition 1. Line 6 and 7 find the worst rule from R_p and the best from R_a, and swap them in Line 13. The algorithm terminates in Line 9, when the benefit of the best assigned rule is higher than of the worst of the proposed. The time required for the sort operation is bounded by the number of patterns which is limited. Most of the algorithm time is spent on evaluating the patterns and swapping the triples. However, the adaption phase takes place only when the system is idle and has no query to execute.

Algorithm 1: Rules-based space adaption algorithm

```
   input : RDF graph G = {V, E}, heat queries set H and two sets of the system
           operational rules: proposed rules R_p and assigned rules R_a
 1 for each r ∈ R_p ∪ R_a do
 2 |   r ← updateRulesAccess(r, H);
 3 |   r ← sortRuleBySource(r);
 4 end
 5 while true do
 6 |   r_p ← ϖ_op|ϖ_op = (χ, s_p, a_p, Δ_p) : ∀r_i ∈ R_p, b(ϖ_op) ≥ b(r_i) ;
 7 |   r_a ← ϖ_op|ϖ_op = (χ, s_a, a_a, Δ_a) : ∀r_i ∈ R_a, b(ϖ_op) < b(r_i) ;
 8 |   if b(r_a) ≥ b(r_p) then
 9 |   |   break
10 |   end
11 |   V̂_p ← evaluate(s̄_p);
12 |   V̂_a ← evaluate(s̄_a);
13 |   swapAssignment((r_p, V̂_p), (r_a, V̂_a));
14 end
```

8 Evaluation

In this section, we evaluate the universal adaption approach implemented in UniAdapt; source code is available online[2] We use the LUBM [18] is a generated RDF data set that contains data about universities. The size of the generated data is over 1 billion triples. The benchmark contains 14 test queries[3] that are labeled L_1 to L_{14}. The queries can be classified into 2 categories: bounded and unbounded. The bounded queries contain at least one constant vertex (subject or object) excluding the triple pattern that has "type" as predicate. The unbounded queries are L_2 and L_9, and all the others are bounded. The execution of a bounded query is limited to certain part of the RDF graph. Those parts can be efficiently recognized during query execution given the existence of a proper index.

No storage limit Since the systems used in comparison do not have space adaption, we run this part of the tests under no storage restrictions. We "train" the

[2] https://github.com/ahmedalghezi/UniAdapt
[3] http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt

systems by an initial workload of 100 queries similar to the 14 queries of LUBM. Then we run a batch that contains 64 repetitions of each query on the systems shown in Table 1, and record the average runtime for the given queries. There was a clear superiority in the performance regarding the unbounded queries L_2 and L_9, and the bounded query L_8. However, the other bounded queries L_3 to L_6 behaved very closely for both UniAdapt and AdPart due to their relative simplicity. For the bounded queries L_4, L_6, L_7 and L_8, the running times were generally much less than the previous unbounded queries (Table 2). Both systems performed closely for L_4 and L_6. But Unidapt was superior in adapting to L_8, while AdPart showed better performance to L_7.

Table 1: Runtimes (ms) of LUBM queries

	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9
UniAdapt	178	1	1	1	5	45	10	347
AdPart	7K	4	2	3	4	88	370	1K
H-RDF-3X	12K	3K	3K	3K	3K	7K	5K	9K
H-RDF-3X+	11K	3K	3K	3K	3K	5K	5K	8K

Table 2: Runtimes (ms) of bounded queries with 2 batches (b_1 and b_2)

	L4		L6		L7		L8	
	b_1	b_2	b_1	b_2	b_1	b_2	b_1	b_2
UniAdapt	2	1	5	4	45	15	10	1
AdPart	3	1	4	4	88	1	370	1.5

Storage-workload adaption In this part, we evaluate the unique adaption capabilities of UniAdapt with the storage space and the workload. In this context, we set three levels of storage capacity: $S_{2.5}$, S_5, and S_7. With capacity S_5 the system can potentially maintain 5 full indexes but may also decide to use the free space for join cache or replication. Moreover, we generated six types of workload from the LUBM data-set that are given by Table 3. The fact that all the queries have the "type" predicates allowed the UniAdapt to maintain only two indexes that are type indexes. One is sorted on the subject while the other is sorted on the object. The first run of the WLu1 in the space capacity of $S_{2.5}$ took relatively long (Fig. 2), due to the lack of the OPS or SPO indexes that are used to decrease the cost of further join; besides the difficulties to perform enough replications on the limited space to save the communication cost. The uniform workload distribution amplifies the problem by hardening the task of the optimizer to detect highly accessed data using its specific rules. Moving from $S_{2.5}$ to S_5 allows the system to have enough replications, as well as two more indexes besides its two type indexes. This is reflected in an obvious decrease in the execution time. Upgrading the capacity to S_7,

Table 3: Generated workload properties of LUBM data set

Workload	Bounding	Length	Distribution
WLu1	No	3-4	uniform
WLu2	No	3-4	50% to 50%
WLu3	No	3-4	90% to 10%
WLb1	Yes	3-4	uniform
WLb2	Yes	3-4	50% to 50%
WLb3	Yes	3-4	90% to 10%

provides enough space for more and better cache. That showed the highest decrease in the query execution time. In addition, moving towards better workload quality helped the specific rules to better detect the highly accessed data. This is seen when moving from WLu1, WLu2 to WLu3. The workload impact was much higher on the bounded queries in workloads WLb1, WLb2, and WLb3. Starting with WLb1 at capacity level of $S_{2.5}$ caused a high increase of the query execution time. This is because the space is only enough for two full indexes. Increasing the space to S_5 relaxed the optimization process and allowed the system to have both OPS and SPO indexes for most of the data, and overrides the issue of the low workload quality. However, given a low workload quality the system achieves a useful cache only for storage level of s_7. Moving towards the better-quality workload WLb2 resulted in a high decrease in the execution time, even for the limited storage level of $s_{2.5}$, as the system is able to detect the hotter parts of the data using the specific rules. Moving to the excellent workload of WLb3 flattened the differences between the storage levels, as most of the workload is now targeting a very small region of the data, which can be efficiently indexed, replicated, and cached using the specific rules. That results in a high boost to the performance.

9 Conclusion and Future Work

In this paper, we presented the universal adaption approach for the storage layer of a distributed RDF triple store called UniAdapt. The adaption process aims to adapt the limited storage to store the most beneficial data within indexes, replication, and join caches. We defined a dynamic cost model that engages the benefit of each data assignment with its usage rate as well as its storage cost. Our experimental results showed the impact of the universal adaption on the query execution in different levels of storage space and workload quality. UniAdapt was able to excel in difficult levels of storage space capacity. On the other hand, it was able to flexibly adapt to space abundance for more query performance. For a future work, we consider adding the temporal effect of the workload in the heat query structures.

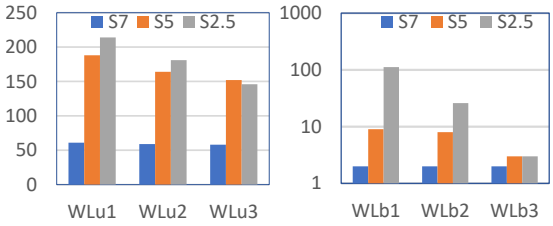

Fig. 2: Adaption with workload and space (average running time per query in milliseconds

References

1. Aluc, G., Özsu, M. T., and Daudjee, K. Workload matters: Why RDF databases need a new design. *Proc. VLDB Endow. 7*, 10 (2014), 837–840.

2. Aluç, G., Özsu, M. T., and Daudjee, K. Building self-clustering RDF databases using tunable-lsh. *VLDB J. 28*, 2 (2019), 173–195.
3. Ceri, S., Negri, M., and Pelagatti, G. Horizontal data partitioning in database design. In *SIGMOD Conference* (1982), ACM Press, pp. 128–136.
4. Chong, E. I., Das, S., Eadon, G., and Srinivasan, J. An efficient SQL-based RDF querying scheme. In *VLDB* (2005), ACM, pp. 1216–1227.
5. Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. V. *Algorithms*. McGraw-Hill, 2008.
6. Erxleben, F., Günther, M., Krötzsch, M., Mendez, J., and Vrandecic, D. Introducing wikidata to the linked data web. In *International Semantic Web Conference (1)* (2014), vol. 8796 of *Lecture Notes in Computer Science*, Springer, pp. 50–65.
7. Galárraga, L., Hose, K., and Schenkel, R. Partout: A distributed engine for efficient RDF processing. In *Proceedings of the 23rd International Conference on World Wide Web* (New York, NY, USA, 2014), WWW '14 Companion, ACM, pp. 267–268.
8. Gurajada, S., Seufert, S., Miliaraki, I., and Theobald, M. TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In *SIGMOD* (2014), ACM, pp. 289–300.
9. Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., and Sahli, M. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *The VLDB Journal 25*, 3 (June 2016), 355–380.
10. Hayes, P. RDF semantics, W3C Recommendation 10 February. https://www.w3.org/TR/rdf-mt/, 2004.
11. Hose, K., and Schenkel, R. WARP: workload-aware replication and partitioning for RDF. In *ICDE Workshops* (2013), IEEE Computer Society, pp. 1–6.
12. Huang, J., Abadi, D. J., and Ren, K. Scalable SPARQL querying of large RDF graphs. *Proc. VLDB Endow. 4*, 11 (2011), 1123–1134.
13. Karypis Lab. METIS: family of graph and hypergraph partitioning software. http://glaros.dtc.umn.edu/gkhome/views/metis, 2020.
14. Moerkotte, G., and Neumann, T. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB* (2006), VLDB Endowment, pp. 930–941.
15. Monaci, M., Pferschy, U., and Serafini, P. Exact solution of the robust knapsack problem. *Computers & operations research 40*, 11 (2013), 2625–2631.
16. Neumann, T., and Weikum, G. The RDF-3X engine for scalable management of rdf data. *The VLDB Journal 19*, 1 (2010), 91–113.
17. Peng, P., Zou, L., Chen, L., and Zhao, D. Query workload-based RDF graph fragmentation and allocation. In *EDBT* (2016), OpenProceedings.org, pp. 377–388.
18. Projects, S. The lehigh university benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/.
19. Wang, L., Xiao, Y., Shao, B., and Wang, H. How to partition a billion-node graph. In *ICDE* (2014), IEEE Computer Society, pp. 568–579.
20. Weiss, C., Karras, P., and Bernstein, A. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow. 1*, 1 (2008), 1008–1019.
21. Zhang, X., Chen, L., Tong, Y., and Wang, M. EAGRE: towards scalable I/O efficient SPARQL query evaluation on the cloud. In *ICDE* (2013), IEEE Computer Society, pp. 565–576.