Eurostars Project

3DFed – Dynamic Data Distribution and Query Federation

Project Number: E!114681 **Start Date of Project:** 2021/04/01 **Duration:** 36 months

Deliverable 3.1
Initial Report on the Automatic Data Distribution

Dissemination Level	Public
Due Date of Deliverable	March 31, 2022
Actual Submission Date	December 12, 2022
Work Package	WP3, Automatic Data Distribution & Dynamic Exchange
Deliverable	D3.1
Type	Report
Approval Status	Final
Version	1.1
Number of Pages	10

Abstract:
Data partitioning is an effective way to manage large datasets. While a broad range of RDF graph partitioning techniques has been proposed in previous works, little attention has been given to workload-aware RDF graph partitioning. In this deliverable we propose two techniques that make use of the querying workload to detect the portions of RDF graphs that are often queried concurrently. Our techniques leverage predicate co-occurrences in SPARQL queries. By detecting highly co-occurring predicates, our techniques can keep data pertaining to these predicates in the same data partition.

3DFed Project by Eurostars.

History

Version	Date	Reason	Revised by
0.1	12/09/2021	Initial Template & Deliverable Structure	Muhammad Saleem
0.2	25/02/2022	Automatic Data Distribution Results	Akhter et al.
0.3	23/03/2022	Issued for review	Mohammad Sajjadi
0.4	28/03/2022	Review	Milos Jovanovik
1.0	29/03/2022	Final	Mohammad Sajjadi
1.1	12/12/2022	Adjustment on proposed techniques	Mohammad Sajjadi & Muhammad Saleem

Author List

Organization	Name	Contact Information
University of Paderborn	Adnan Akhter	akhter@informatik.uni-leipzig.de
University of Paderborn	Muhammad Saleem	saleem@informatik.uni-leipzig.de
University of Paderborn	Alexander Bigerl	alexander.bigerl@uni-paderborn.de
University of Paderborn	Axel-Cyrille Ngonga Ngomo	axel.ngonga@upb.de
elevait GmbH & Co. KG	Mohammad Sajjadi	mohammad.sajjadi@elevait.de
OpenLink Software	Milos Jovanovik	mjovanovik@openlinksw.com
OpenLink Software	Mirko Spasić	mspasic@openlinksw.com

Contents

1 Introduction

Partitioning large amounts of data among multiple data nodes helps improve the scalability, availability, ease of maintenance, and overall query processing performance of storage systems. Current distributed triple stores employ various RDF graph partitioning techniques [16]. A recent performance evaluation of various RDF graph partitioning techniques shows that there is no clear winner in terms of overall query runtime performance improvement in different partitioning environments [2]. This is because the evaluated RDF partitioning techniques are mostly generic and can be applied on any data graphs and hence the specific properties of RDF graphs are not taken into account. Akhter et al. [2] suggest that data (i.e., a portion of a large dataset) that is queried (i.e., accessed) together in user queries, should be kept in their same partitions. The partitioning technique that take data locality into account minimize the inter-communication between partitions, thus potentially leading to better query runtimes.

The majority of the state-of-the-art RDF graph partitioning techniques only consider the underlying RDF data [16]. Consequently, they fail to leverage the querying history, i.e., they do not make use of information pertaining to the likelihood of particular portions of the data being queried concurrently to answer user queries. Only a few approaches address workload-based RDF partitioning, in particular [6, 13]. Both approaches leverage the *joins* between triple patterns in the querying workload. On the other hand, we propose a novel workload-based RDF partitioning technique that leverages the *predicates co-occurrences* in the querying workload. The idea is that all RDF triples with predicates that are most commonly queried together should be stored in the same partition. Ideally, this should lead to one partition being consulted by the distributed RDF engine to execute SPARQL triple patterns with the most commonly co-occurred predicates. This would decrease the inter-communication cost between multiple worker nodes of the distributed RDF engines and hence, lead to better query runtime performance. The *predicate-based* partition has inherent advantages, such as its ease of managing index updates as well as dynamic data redistribution and replication [16]. In addition, the number of distinct predicates in the RDF datasets is usually much smaller than the number of subjects or objects, thus it is faster to group them in clusters and create the required partitions.

We propose two RDF graph partitioning techniques: 1. predicates co-occurrence-based partitioning using a greedy algorithm (PCG), and 2. predicates co-occurrence-based partitioning using extended markov clustering (PCM). Both of these techniques make use of clustering algorithms to first cluster all the predicates used in the input querying workload. The partitions are then created according to the clusters such that all triples pertaining to predicates in a given cluster are distributed into the same partition. In the future, we will measure the effect of querying workloads on the accuracy of data distribution in terms of different measurements.

2 State-of-the-art RDF Graph Partitioning Techniques

State-of-the-art RDF graph partitioning techniques can be divided into various categories [16]:

- **Hash-Based Partitioning.** This type of partitioning is based on applying hash functions on the individual elements of the triples (i.e., subject , predicate, object), followed by the modulo operation: the distribution of triples to required n number of partitions is carried out by using `hash(triple element) mod n`. The *subject-hash-based*, *predicate-hash-based*, and *hierarchical-hash-based* partitioning are common examples of partitioning from this category [2, 16, 11]. There are many distributed RDF engines[1] that use *hash-based* partitioning, including Virtuoso [5] and TriAD [8].
- **Graph-Based Partitioning.** This type of partitioning is based on clustering/distributing vertices or

[1]A complete list is provided in [16].

edges of the RDF graph. METIS[2] library provides several graph-based partitioning techniques [11, 10]. Graph-based partitioning has been used in many distributed RDF engines [16], including Koral [11] and H-RDF-3X [10].

- **Workload-Aware Partitioning.** This type of partitioning makes use of the query workload to distribute RDF triples among required partitions. Worq [13] and Partout [6] are examples of workload-aware RDF graph partitioning [6, 13].Other examples are [3, 14, 4].
- **Range Partitioning.** In this type of partitioning, RDF triples are distributed based on certain range values of the partitioning key. For example, it creates a separate partition of all RDF triples with Predicate age and object values between 30 and 40. Range partitioning has been used in Yars2 [9] and in [17].
- **Vertical Partitioning.** Rather than distributing RDF triples, vertical partitioning distributes individual elements of triples into different partitions or tables. Therefore, rather than storing the complete triples, it generally stores two out of the three elements of the triples. For example, SPARQLGX [7] divides triples by their predicates and only stores the subject and object parts of the triples in n (equals number of distinct predicates in the RDF) predicate tables. Other examples are [12, 1, 15].

We refer readers to [16] for a more exhaustive overview of state-of-the-art RDF graph partitioning techniques used in state-of-the-art distributed RDF engines. An empirical evaluation of the state-of-the-art RDF graph partitioning techniques is presented in [2, 11], in which seven RDF graph partitioning techniques are evaluated. For better understanding of the proposed and state-of-the-art techniques, we use a motivating example which we will carry out throughout this deliverable.

Motivating Example. Consider the set of RDF triples given in Figure 1a. Suppose we want to create three partitions of this graph and represent them in different colors (i.e., red, blue and green). Figure 1b shows the resulting partitions created by the different techniques and is explained in the subsequent paragraph.

Let T be the set of all RDF triples in a dataset and n be the required number of partitions. The Horizontal partitioning technique assigns the first $|T|/n$ triples in partition 1, the next $|T|/n$ triples in partition 2 and so on. Using this technique, our example dataset is split such that triples 1-4 are assigned into the green partition, triples 5-8 into are assigned into the red partition, and triples 9-11 are assigned into the blue partition. The Subject-Based partitioning technique assigns all triples with the same subject into the same partition. Using this technique, our example dataset is split such that triples 3, 10 and 11 are assigned into the red partition, triple 7 is assigned into the blue partition, and the remaining triples are assigned into the green partition. The Predicate-Based partitioning technique assigns all the triples with the same predicate into same partition. Using this technique, our example dataset is split such that triples 1, 7, 8, 9 and 10 are assigned into the red partition, triples 2, 3, 5, and 11 are assigned into the green partition, and remaining triples are assigned into the blue partition. The Hierarchical Partitioning technique assigns all IRIs with a common hierarchy prefix into the same partition. Using this technique, our example dataset is split such that triples 3, 7, 10 and 11 are assigned into the red partition, triples 1, 2, 4 and 8 are assigned into the green partition, and the remaining triples are assigned into the blue partition. The Recursive-Bisection partitioning technique splits the graph in two, and repeatedly applies this strategy until the desired number of partitions are generated. Using this technique, our example dataset is split such that triples 1, 2, 4, 7, and 8 are assigned into the green partition, triples 3, 5, 6, 9 and 10 are assigned into the red partition, and triple 11 is assigned into the blue partition. The TCV-Min partitioning technique makes partitions by minimizing the communication costs of connected nodes. Using this technique, our example dataset is split such that triples 1, 2, 4, 5, 6, 8 and 9 are assigned into the green partition, triples 3, 7 and 10 are assigned into the red partition, and triple 11 is assigned into the blue partition. The Min-Edgecut partitioning technique distributes nodes by minimizing the number of edges connected to them. Using this technique, our example dataset is split such that triples 1, 2, 4, 7 and 8 are assigned into the green partition, triples 3, 5, 6, 9 and

[2]METIS: `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`

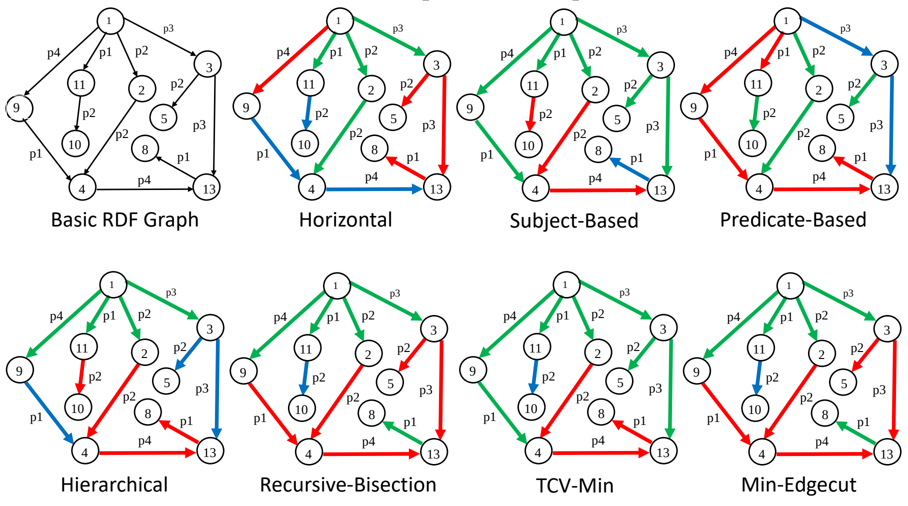
10 are assigned into the red partition, and only triple 11 is assigned into the blue partition. In the next section, we explain our techniques in detail and show how they partition our example dataset by using a querying workload.

```
@prefix hierarchy1: <http://first/r/> . @prefix hierarchy2: <http://second/r/> .
@prefix hierarchy3: <http://third/r/> . @prefix schema: <http://schema/> .

#Triple1)  hierarchy1:s1   schema:p1      hierarchy2:s11 .
#Triple2)  hierarchy1:s1   schema:p2      hierarchy2:s2 .      #Triple7)   hierarchy2:s13    schema:p1    hierarchy2:s8 .
#Triple3)  hierarchy2:s2   schema:p2      hierarchy2:s4 .      #Triple8)   hierarchy1:s1     schema:p4    hierarchy3:s9 .
#Triple4)  hierarchy1:s1   schema:p3      hierarchy3:s3 .      #Triple9)   hierarchy3:s9     schema:p1    hierarchy2:s4 .
#Triple5)  hierarchy3:s3   schema:p2      hierarchy1:s5 .      #Triple10)  hierarchy2:s4     schema:p4    hierarchy2:s13 .
#Triple6)  hierarchy3:s3   schema:p3      hierarchy2:s13 .     #Triple11)  hierarchy2:s11    schema:p2    hierarchy1:s10 .
```

812

(a) An example of RDF triples



(b) Graph representation and partitioning. Only node numbers are shown for simplicity.

Figure 1: Partitioning an example RDF into three partitions using different partitioning techniques. Partitions are highlighted in different colors.

3 Proposed Techniques

Both of our techniques are comprised of three main steps: (i) extract a list of predicate co-occurrences from a querying workload and model them as a weighted graph (Section 3.1), (ii) use this weighted graph as an input to generate clusters of predicates (Section 3.2), and (iii) allocate the obtained clusters to partitions (Section 3.3). In the following discussion, we suppose we have a workload of eight queries as shown in Table 1.

SELECT * WHERE	SELECT * WHERE	SELECT * WHERE	SELECT * WHERE	SELECT * WHERE	SELECT * WHERE	SELECT * WHERE	SELECT * WHERE
{	{	{	{	{	{	{	{
?S :P1 ?O1.	?S :P1 ?O.	?S :P1 ?O1.	?S :P1 ?O.	?S1 :P1 ?O.	?O :P1 ?S.	?S1 :P1 ?O.	?S :P1 ?O.
?S :P2 ?O2	?O :P2 ?O2	?S :P3 ?O3	?O :P3 ?O3	?S3 :P3 ?O	?S :P3 ?S3	?S2 :P2 ?O.	?S :P2 ?O.
}	}	}	}	}	}	}	?S :P3 ?O.
							?S :P4 ?O
							}

Table 1: Query examples

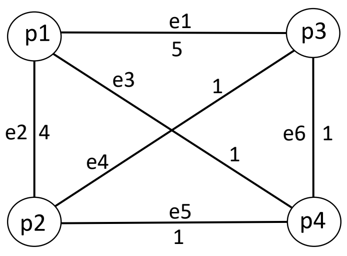
3.1 Graph Modeling

Since both techniques are based on query workload, we assume that we are given a query workload $Q = \{q_1, \ldots, q_n\}$ of SPARQL queries. Ideally, the query workload Q contains real-world queries posted by the users of the RDF dataset, which can be collected from the query log of the running system. However, real user queries might not be available. In this case the query workload can be either estimated from queries in applications accessing the RDF data or synthetically generated with the help of the domain experts of the given RDF dataset that needs to be partitioned.

For a given work load $Q = \{q_1, \ldots, q_n\}$, we create a predicates co-occurrence list $L = \{e_1, \ldots, e_m\}$ where each entry is a tuple $e =< p_1, p_2, c >$, with p_1, p_2 two different predicates used in the triple patterns of SPARQL queries in the given workload, and c is the co-occurrence count, i.e. the number of queries in which both p_1 and p_2 are co-occurred. By looking at our query examples given in Table 1, the predicates p_1, and p_2 co-occurred in a total of 4 queries, thus one entry of the L will be $< p_1, p_2, 4 >$. For the sake of simplicity, the corresponding predicate-to-predicate co-occurrence list for our query examples is shown in Figure 2a. Finally, we model the list L as a weighted graph, such that for a given list entry $e =< p_1, p_2, c >$, we create two nodes (one each for p_1 and p_2) that are connected by a link with weight equalling c. The corresponding weighted graph is shown in Figure 2b.

P1	P2	Co-occurrences
p1	p2	4
p1	p3	5
p1	p4	1
p2	p3	1
p2	p4	1
p3	p4	1

(a) Predicate co-occurrences



(b) Weighted graph of the predicate co-occurrences

Figure 2: The predicate co-occurrences table and corresponding weighted graph for the example queries given in table 1.

3.2 Graph Clustering

We propose two clustering algorithms to generate clusters of predicates from the weighted predicates graph generated in the previous section.

PCM Clustering. Algorithm 1 shows the predicate clustering using a modified version of the well-known Markov[3] clustering. For the input weighted predicates graph G, a transition matrix T is created which is then normalized (Lines 2-3 of algorithm 1). A transition matrix is basically a matrix representation of a weighted graph. Since our weighted graph shown in Figure 2b has four nodes, a 4×4 (one row and column for each predicate vertex) matrix will be created. The corresponding transition matrix is shown in Figure 3. The normalization of the matrix is done by dividing each element of a particular row by the sum of all the elements in that row. The normalized matrix is show in Figure 3.

[3]Markov clustering: `https://micans.org/mcl/`

Algorithm 1: Adapted Markov Clustering

```
1 MCL($G$, $maxR\, e$, $maxZero$ ,$n$) /* Input: Weighted predicates graph $G$, maximum residual
      $maxR = 0.001$, inflation exponent for Gamma operator $e = 2$, maximum value
      considered zero for pruning operations $maxZero = 0.001$, and $n$ number of
      required clusters                                                              */
2 $T \in \mathbb{R}^{p \times p} := GetTransitionMatrix(G)$ ;
3 $T \in \mathbb{R}^{p \times p} := Normalize(T)$ ;
4 $doubleresidual := 1.0$ ;
5 while residual > maxR do
6     $T := (T)^e$ // Expend
7     $residual = inflate(T, e, maxZero)$;
8 end
9 return $getClusters(T, n)$ /* get $n$ clusters from matrix                           */
```

$$\underset{\text{Transition Matrix}}{\begin{array}{c|cccc} & P1 & P2 & P3 & P4 \\ \hline P1 & 0 & 4 & 5 & 1 \\ P2 & 4 & 0 & 1 & 1 \\ P3 & 5 & 1 & 0 & 1 \\ P4 & 1 & 1 & 1 & 0 \end{array}} \Rightarrow \underset{\text{Normalized Matrix}}{\begin{array}{c|cccc} & P1 & P2 & P3 & P4 \\ \hline P1 & 0 & 4/10 & 5/10 & 1/10 \\ P2 & 4/6 & 0 & 1/6 & 1/6 \\ P3 & 5/7 & 1/7 & 0 & 1/7 \\ P4 & 1/3 & 1/3 & 1/3 & 0 \end{array}} \Rightarrow \underset{\text{Final Matrix}}{\begin{array}{c|cccc} & P1 & P2 & P3 & P4 \\ \hline P1 & 1.0 & 0.0 & 0.0 & 0.0 \\ P2 & 0.0 & 0.0 & 0.0 & 0.99 \\ P3 & 1.0 & 0.0 & 0.0 & 0.0 \\ P4 & 0.0 & 0.0 & 0.0 & 0.99 \end{array}}$$

Figure 3: Creation of a matrix during PCM using our weighted graph

The next two steps are the standard *expansion* and *inflation* of the Markov clustering, applied on the normalized transition matrix. These steps are continued until residual value is greater than maximum residual (Lines 4-8 of algorithm 1). The expansion is a simple self-multiplication of the matrix, raise to power of input parameter e. The inflate part is according to the inflate stochastic matrix by Hadamard (elementwise) exponentiation[4].

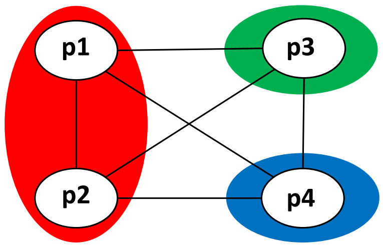
The last step is to interpret the resulting transition matrix to discover n clusters. This is achieved by sequentially adding non-zero row-wise values of matrix T to a cluster. For example, in our final matrix shown in Figure 3, the first non-zero row-wise value is 0.66 at position $T_{1,2}$. Thus, the corresponding predicates, i.e. p_1, p_2, will be added into a single cluster. The next non-zero row-wise value is at position $T_{2,4}$, which corresponds to predicates p_1, p_4. Since p_1 already exists, only $p4$ will be added into the cluster. Finally, p_3 will be added. Now our cluster contains a sequential list of predicates $\{p_1, p_2, p_4, p_3\}$. Since we need n partitions, we simply divide the total elements from the cluster by n number of required partitions to get the number of elements from the sequential list of elements to be combined into a single partition. In our case, the number of elements is 4 while desired partitions are 3. Thus, we divide 4/3 and assign the first two elements (i.e., p_1, p_2) to partition 1 and the next element (i.e., p_4) into partition 2 and the final element p_3 into partition 3. The final cluster of predicates is shown in Figure 4a. Please note that it is possible that there exist many predicates in the RDF dataset that are not used in the query workload. In that case we assign a single separate partition for all unused predicates.

PCG Clustering. Algorithm 2 shows the predicate clustering using the proposed greedy clustering method. The first step is to calculate the expected size (in terms of the number of triples) of each partition. The next step is to obtain all edges between predicates according to their increasing order of weights. For the graph given in Figure 2b, our sorted list of edges will be $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$. The next step is to loop through each edge $e_j \in E$ and get the corresponding predicates that are connected by the given edge e_j (Lines 6-7 of algorithm 2).

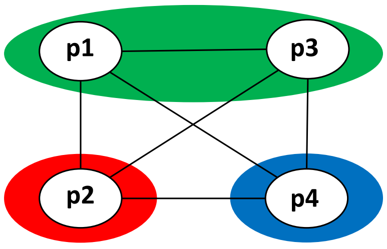
[4]Inflate: `http://java-ml.sourceforge.net/api/0.1.1/net/sf/javaml/clustering/mcl/MarkovClustering.html`

Algorithm 2: Greedy Clustering

```
1  PCG($G$, $D$, $n$) /* Input: Weighted predicates graph $G$, Dataset $D$ to be partitioned, $n$
       number of required clusters */
2  $t = |D|/n - 1$ ;                                   // Size of a partition
3  $E$ = getSortedEdges($G$) ; /* Obtain all edges between the predicates according to their
     weight */
4  $C = \{c_1 \dots c_n\}$ ;                            // Required clusters
5  $i = 1$ ;
6  forall $e_j \in E$ do
7      $P(p_k, p_l)$ = getNodesPair($G$, $e_j$) /* Obtain both nodes (predicates) that are connected
           by the edge $e_i$ */
8      $T$ = getTriplesCount($D$, $P(p_k, p_l)$) /* get the combined count of the triples for
           predicates $p_k$ and $p_l$ from dataset $D$ */
9      if $|c_i| < t$ /* if size of triples in cluster $c_i$ is less than the threshold $t$ */
10      then
11         $c_i \leftarrow \{p_k, p_l\}$ ;              // assign both predicates to cluster
12     else
13         $i = i + 1$ ;                                // move to next cluster
14     end
15 end
16 return $C$ ;                                          // Clusters
```



(a) PCM



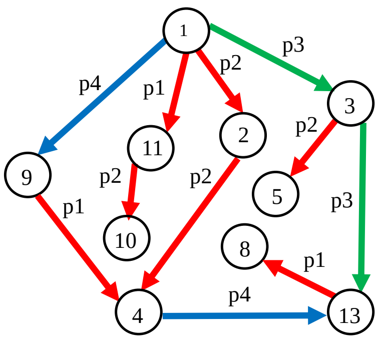
(b) PCG

Figure 4: Predicate clusters created by the proposed techniques for the example RDF dataset given in Figure 1a. Clusters are highlighted in different colors

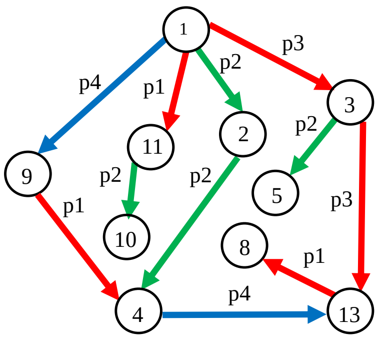
We then get the combined count of the triples for predicates p_k and p_l from input dataset D. If the current size of the cluster c_i is less than the threshold t, both predicates are added into the same cluster c_i. However, if the size of the current cluster exceeds the threshold, a new cluster is created for the upcoming predicates (Lines 8-14 of algorithm 2). The final three clusters of predicates are shown in Figure 4b. Please note that, as with PCM, it is possible that there exist many predicates in the RDF dataset that are not used in the query workload. In that case, we assign a single separate partition for all unused predicates.

3.3 Assigning Clusters to Partitions

The clustering algorithms explained in the previous steps give n clusters of predicates. In the last step, triples from a given RDF dataset D are distributed into partitions according to the aforementioned predicate-based partitions: for each predicate p in a specific cluster c_i, assign all the triples with predicate $p \in D$ into the same partition. Figure 5a and 5b show the final partitions created by both of the proposed techniques. Please note that



(a) PCM



(b) PCG

Figure 5: Final three partitions created by the proposed techniques for the example RDF dataset given in Figure 1a. Partitions are highlighted in different colors)

these partitions are different from all the techniques shown in Figure 1b.

4 Conclusion and Future Work

Two RDF graph partitioning algorithms PCM and PCG based on querying workloads were developed that leverage the predicate co-occurrences in these workloads. Both techniques extract a list of predicate co-occurrences from a querying workload and model them as a weighted graph and use it to generate clusters of predicates and finally allocate the obtained clusters to partitions. These techniques differs in clustering algorithms to generate clusters of predicates. PCM uses a modified version of the well-known Markov3 clustering, and PDG uses a greedy clustering method. Different state-of-the-art RDF graph partitioning techniques were selected to be compared to our techniques. Selecting these techniques was based on being open source and configurable, working for RDF data, being scaleable to medium-large datasets, taking the RDF dataset and/or workload as input, giving the required number of RDF chunks as output, and not requiring on-line services such as cloud or configuring online datasets. Ten partition methods such as partout, horizontal, Hierarchical, vertical, subject-wise, property-wise, Recursive-bisection, min-edge-cut and tcv-min are selected. In the future, the evaluation of our techniques compared to the previous techniques in terms of better query runtime performances, number of timeout queries, overall rank score, and number of distinct sources selected, is recommended.

References

[1] Abadi et al. Scalable semantic web data management using vertical partitioning. 2007.

[2] Akhter et al. An empirical evaluation of rdf graph partitioning techniques. In *European Knowledge Acquisition Workshop*, 2018.

[3] Al-Ghezi et al. Adaptive workload-based partitioning and replication for rdf graphs. In *International Conference on Database and Expert Systems Applications*, 2018.

[4] Aluç et al. chameleon-db: a workload-aware robust rdf data management system. *University of Waterloo, Tech. Rep. CS-2013-10*, 2013.

[5] Erling et al. Rdf support in the virtuoso dbms. In *Networked Knowledge-Networked Media*, 2009.

[6] Galárraga et al. Partout: A distributed engine for efficient rdf processing. 2014.

[7] Graux et al. Sparqlgx: Efficient distributed evaluation of sparql with apache spark. 2016.

[8] Gurajada et al. Triad: A distributed shared-nothing rdf engine based on asynchronous message passing. 2014.

[9] Harth et al. Yars2: A federated repository for querying graph structured data from the web. In *The Semantic Web*, 2007.

[10] Huang et al. Scalable sparql querying of large rdf graphs. 2011.

[11] Janke et al. Koral: A glass box profiling system for individual components of distributed rdf stores. 2017.

[12] Lehmann et al. Distributed semantic analytics using the sansa stack. In *Proceedings of 16th International Semantic Web Conference-Resources Track*, 2017.

[13] Madkour et al. Worq: Workload-driven rdf query processing. 2018.

[14] Padiya et al. Dwahp: workload aware hybrid partitioning and distribution of rdf data. 2017.

[15] Schätzle et al. S2rdf: Rdf querying with sparql on spark. 2016.

[16] Waqas et al. Storage, indexing, query processing, and benchmarking in centralized and distributed rdf engines: A survey. 2020.

[17] Whitman et al. Distributed spatial and spatio-temporal join on apache spark. 2019.