Efficient RDF Knowledge Graph Partitioning Using Querying Workload

Adnan Akhter
akhter@informatik.uni-leipzig.de
Data Science Group, Department of Computer Science,
Paderborn University, Germany

Muhammad Saleem
saleem@informatik.uni-leipzig.de
AKSW Research Group, University of Leipzig
Liepzig, Germany

Alexander Bigerl
alexander.bigerl@uni-paderborn.de
Data Science Group, Department of Computer Science,
Paderborn University, Germany

Axel-Cyrille Ngonga Ngomo
axel.ngonga@upb.de
Data Science Group, Department of Computer Science,
Paderborn University, Germany

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 paper, 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. We evaluate the proposed partitioning techniques using various real-data and query benchmarks generated by the FEASIBLE SPARQL benchmark generation framework. Our evaluation results show the superiority of the proposed techniques in comparison to previous techniques in terms of better query runtime performances.

CCS CONCEPTS

• **Information systems → Distributed storage**.

KEYWORDS

RDF knowledge graph partitioning; querying workload; predicate co-occurrence; PCG; PCM

ACM Reference Format:
Adnan Akhter, Muhammad Saleem, Alexander Bigerl, and Axel-Cyrille Ngonga Ngomo. 2021. Efficient RDF Knowledge Graph Partitioning Using Querying Workload. In *Proceedings of the 11th Knowledge Capture Conference (K-CAP '21), December 2–3, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3460210.3493577

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 [19]. 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 (here we mean 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 [19]. 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 [19]. 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

K-CAP '21, December 2–3, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8457-5/21/12…$15.00
https://doi.org/10.1145/3460210.3493577

clusters such that all triples pertaining to predicates in a given cluster are distributed into the same partition. Our overall contributions are as follows: (1) We propose two novel RDF graph partitioning techniques by using different clustering techniques that make use of the predicates co-occurrences in the querying workload. (2) We evaluate our proposed techniques by using different performance measures by using real-data benchmarks. We show the superiority of proposed techniques in comparison to the state-of-the-art RDF graph partitioning techniques. The source codes, datasets, and instructions for reproducing the complete results can be found at https://github.com/dice-group/workload-aware-rdf-partitioning.

The rest of the paper is organized as follows: we introduce the state of the art in RDF graph partitioning, followed by the description of the proposed partitioning techniques and the evaluation results with our key findings and conclusions.

2 STATE-OF-THE-ART RDF GRAPH PARTITIONING TECHNIQUES

State-of-the-art RDF graph partitioning techniques can be divided into various categories [19]:

- **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, 11, 19]. 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 edges of the RDF graph. METIS[2] library provides several graph-based partitioning techniques [10, 11]. Graph-based partitioning has been used in many distributed RDF engines [19] 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, 4, 14].
- **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 [20].
- **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 [1, 12, 17**?**].

We refer readers to [19] 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 paper.

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.

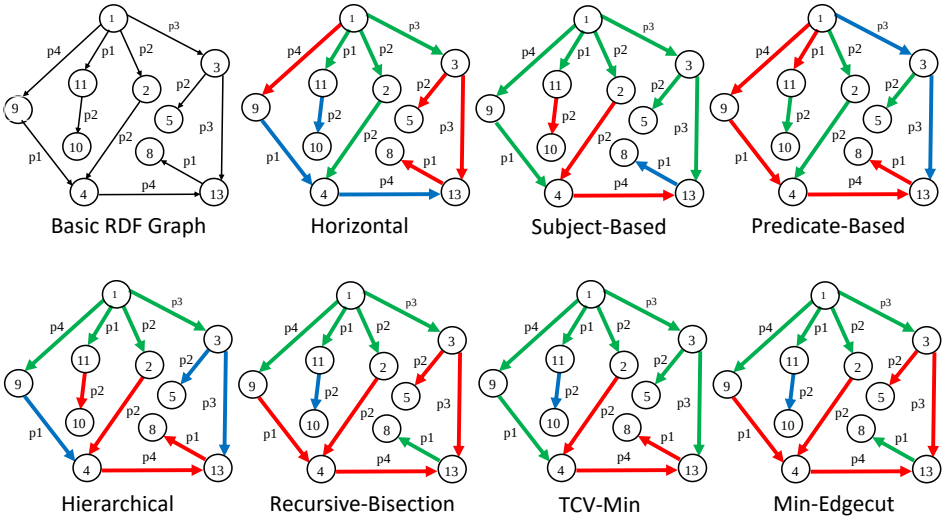
Let T be the set of all RDF triples in a dataset and n be the required number of partitions. The Horizontal partitioning technique assigns the first $|T|/n$ triples in partition 1, the next $|T|/n$ triples in partition 2 and so on. Using this technique, our example dataset is split such that triples 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 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.

[1] A complete list is provided at [19].
[2] METIS: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

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

(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.

```
SELECT * WHERE
{
 ?S :P1 ?O1.
 ?S :P2 ?O2
}

SELECT * WHERE
{
 ?S :P1 ?O.
 ?O :P2 ?O2
}

SELECT * WHERE
{
 ?S :P1 ?O1.
 ?S :P3 ?O3
}

SELECT * WHERE
{
 ?S :P1 ?O.
 ?O :P3 ?O3
}

SELECT * WHERE
{
 ?S1 :P1 ?O.
 ?S3 :P3 ?O
}

SELECT * WHERE
{
 ?O :P1 ?S.
 ?S :P3 ?S3
}

SELECT * WHERE
{
?S1 :P1 ?O.
?S2 :P2 ?O.
}

SELECT * WHERE
{
 ?S :P1 ?O.
 ?S :P2 ?O.
 ?S :P3 ?O.
 ?S :P4 ?O
}
```

Listing 1: Query examples

3 PROPOSED TECHNIQUES

Both of our techniques comprise 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, we suppose we have a workload of eight queries as shown in listing 1.

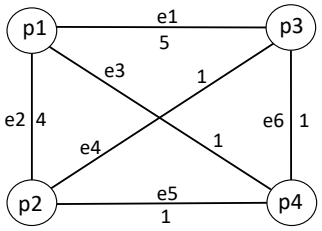
3.1 Graph Modeling

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 listing 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

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 Listing 1.

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 ∈ ℝ^{p×p} := GetTransitionMatrix(G) ;
3 T ∈ ℝ^{p×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                                    */
```

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.

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/

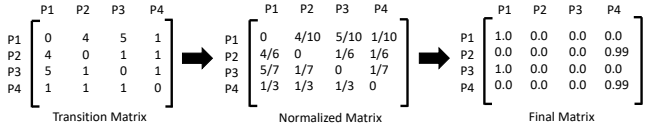


Figure 3: Creation of a matrix during PCM using our weighted graph

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].

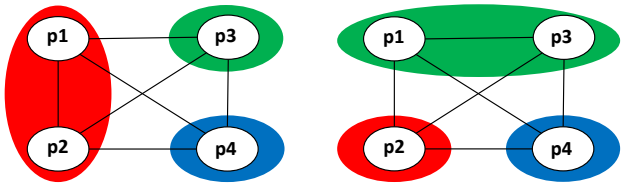
The last step is to interpret the resulting transition matrix to discover n clusters. This is achieved by sequentially adding non-zero row-wise values of matrix T to a cluster. For example, in our final matrix shown in Figure 3, the first non-zero row-wise value is 0.66 at position $T_{1,2}$. Thus, the 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). 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.

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

Algorithm 2: Greedy Clustering

```
PCG(G, D, n) /* Input: Weighted predicates
    graph G, Dataset D to be partitioned,
    n number of required clusters          */
t = |D|/n − 1 ;          // Size of a partition
E = getSortedEdges(G) ;   /* Obtain all edges
  between the predicates according to
  their weight */
C = {c_1 ... c_n} ;          // Required clusters
i = 1 ;
forall e_j ∈ E do
    P(p_k, p_l) = getNodesPair(G, e_j) /* Obtain both
        nodes (predicates) that are
        connected by the edge e_i          */
    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
        */
    if |c_i| < t /* if size of triples in
        cluster c_i is less than the
        threshold t                        */
     then
        c_i ← {p_k, p_l} ; // assign both predicates
         to cluster
    else
        i = i + 1 ;       // move to next cluster
    end
end
return C ;                                 // Clusters
```

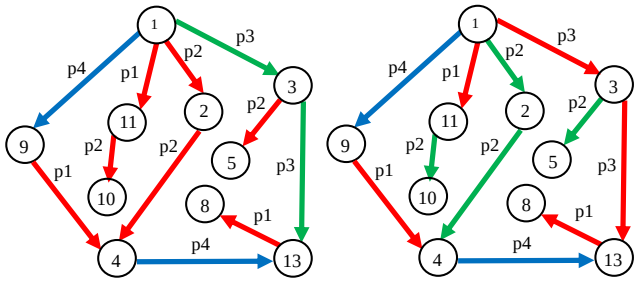


(a) PCM (b) PCG

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

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



(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)

4 EVALUATION

4.1 Evaluation Setup

We have exactly reused the evaluation setup discussed in [2]. The reasons for choosing this evaluation setup are two-fold: (1) since our proposed techniques require query workloads, we wanted to use real-world query workloads (i.e., collected from public SPARQL endpoints of real-world RDF datasets), and real-world RDF benchmarks, (2) we wanted our results to be comparable with the results presented in [2].

Datasets. As in [2], we used two real-world datasets *DBpedia 3.5.1* and the *Semantic Web Dog Food (SWDF)* for partitioning.

Benchmark Queries (test queries). As in [2], we used four sets of real-world SPARQL benchmark queries (300 queries each): *(1) SWDF BGP-only* is the SWDF benchmark containing only single BGP queries; the other SPARQL features such as `OPTIONAL`, `UNION` etc. are not used, *(2) SWDF fully-featured* is the SWDF benchmark containing fully-featured (multiple BGPs, aggregates, functions etc.) SPARQL queries, *(3) DBpedia BGP-only* is the DBpedia benchmark only containing single BGP queries, and *(4) DBpedia fully-featured* benchmark queries contain not only single BGPs but may also include additional constructs. These benchmarks are generated by using FEASIBLE [15], a real benchmark generation framework, out of query logs.

Workloads (train queries). We used a query workload of 3000 queries each for DBpedia and SWDF, which are selected from real-world query logs of these datasets. The reason for choosing 3000 was according to the *10-fold cross validation*[5], which suggests choosing 10% test queries and 90% training queries.

Partitioning Environments. As in [2], we used two distinct partitioning environments to evaluate our techniques: (1) a clustered or distributed RDF storage environment, where the given dataset is distributed among n data nodes of a clustered triple store, (2) a purely federated environment, in which the dataset is distributed among multiple SPARQL endpoints that are physically separated from each other and a federation engine is used to perform the query processing task. We used *Koral* [11] distributed RDF engines for the first type of partitioning environment. We chose *Koral* due to its flexibility in choosing different partitioning methods for data distribution

[5] https://machinelearningmastery.com/k-fold-cross-validation/

among data nodes. In addition, it was previously used in [2]. We used *FedX* [18] and *SemaGrow* for the second type of partitioning environment. The reason for choosing these two engines was because of their different query planning strategies: *FedX* implements an index-free and heuristic-based query planner, while *SemaGrow* implements an index-assisted and cost-based query planner. Both were also used in [2]. It is important to note that Koral does not support many of the SPARQL features used in the fully-featured SPARQL benchmarks. Therefore, we used BGP-only queries in our Koral-based evaluation.

Number of Partitions. Inspired by [2] and [16], we generated 10 partitions of the selected datasets. Therefore, 10 slaves were created in Koral, each one responsible for one partition. Similarly, we used 10 Linux-based Virtuoso7.1 SPARQL endpoints and each of these endpoints store one partition. The selected federation engines physically federate the given SPARQL query over these endpoints.

Selected RDF Graph Partitioning Techniques. We selected state-of-the-art RDF graph partitioning techniques based on the following criteria: (1) open source and configurable, (2) working for RDF data, (3) scaleable to medium-large datasets, such as DBpedia in our case, (4) take the RDF dataset and/or workload as input and give the required number of RDF chunks as output, and (5) do not require online services such as cloud or configuring online datasets. Based on this criteria, we selected ten – Horizontal, Subject-Based, Predicate-Based, Hierarchical, Recursive-Bisection, TCV-Min, Min-Edgecut, Partout, PCG, PCM – RDF graph partitioning techniques to consider in the evaluation results. Please note that the workload-aware technique Partout only worked for SWDF datasets; for DBpedia, it was unable to partition the dataset in 3 days[6].

Performance Measures. As in [2], we used five performance measures: partitions generation time, Queries per Second (QpS) [2, 15], overall rank score, partitioning imbalance, and the total number of sources selected for the complete benchmark execution in a purely federated environment. We used a three minutes timeout for query execution [2, 15] of each query.

The rank score of the partitioning technique is defined as follows [2]:

Definition 4.1 (Rank Score). Let t be the total number of partitioning techniques and b be the total number of benchmark executions that are used in the evaluation. Let $1 \leq r \leq t$ denote the rank number and $O_p(r)$ denote the occurrence of a partitioning technique p placed at rank r. The rank score of the partitioning technique p is defined as follows:

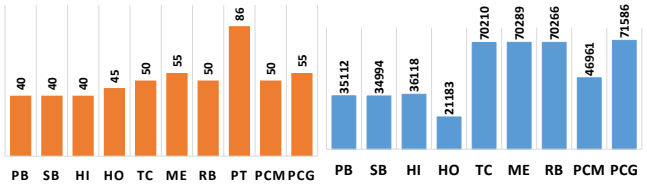
$$s := \sum_{r=1}^{t} \frac{O_p(r) \times (t-r)}{b(t-1)}, 0 \leq s \leq 1$$

In our evaluation, we have a total of ten partitioning techniques (i.e., t = 10 for SWDF, and 9 for DBepdia) and total benchmarks executions b = 10 (i.e., 4 benchmarks by FedX + 4 benchmarks by SemaGrow + 2 benchmarks by Koral). The partitioning imbalance in the size of the generated partitions is defined as follows [2]:

Definition 4.2 (Partitioning Imbalance). Let n be the total number of partitions generated by a partitioning technique and $P_1, P_2, \ldots P_n$ be the set of these partitions, ordered according to the increasing size of the number of triples. The imbalance in partitions is defined as a Gini coefficient:

$$b := \frac{2 \sum_{i=1}^{n} (i \times |P_i|))}{(n-1) \times \sum_{j=1}^{n} |P_j|} - \frac{n+1}{n-1}, 0 \leq b \leq 1$$

[6]We have discussed this issue with the authors of the Partout system



(a) SWDF partitioning time **(b) DBpedia partitioning time**

Figure 6: Time taken for the creation of 10 partitions in seconds. (PB = Predicate-Based, SB= Subject-Based, HI= Hierarchical, HO = Horizontal, TC = TCV-Min, ME Min-Edgecut, RB = Recursive Bisection, PT = Partout)

Hardware and Software Specifications. The hardware and software configuration for our techniques is the same as [2], i.e., all our experiments are executed on a Ubuntu-based machine with intel Xeon 2.10 GHz, 64 cores and 512GB of RAM. We conducted our experiments on local copies of Virtuoso (version 7.1) SPARQL endpoints. We used default configurations for FedX, SemaGrow and Koral (except the slaves were changed from 2 to 10 in Koral).

4.2 Evaluation Results

Please note that the PartOut (PT) results are only shown for SWDF as it was unable to partition DBpedia dataset.

Partition Generation Time. Figure 6 shows a comparison of the total time taken to generate the required 10 partitions for both datasets used in our evaluation. PT took the highest amount of time followed by PCG, Min-Edgecut, Recursive-Bisection, TCV-Min, PCM, Hierarchical, Predicate-Based, Subject-Based and Horizontal, respectively. The remainder of the discussion is focused on PCG, as this the best performing method proposed in this paper.

Query per Second (QpS). Query per Second (QpS) is important to measure the query runtime performances pertaining to different partitioning techniques. The idea is to find out how many queries are executed by a technique in one second. The higher the QpS, the better the query runtime performance. Figure 7 shows a comparison of the QpS values of the selected partitioning techniques for each of the four benchmarks and three different query execution engines. Since Koral only supports BGP-only queries, we used SWDF-BGP and DBpedia-BGP benchmarks. For every timeout query, we added an extra 180 seconds to the total benchmark execution time. The results suggest that the proposed PCG method clearly outperforms the other partitioning methods in the majority of benchmark executions. In particular, PCG ranked first or second in 7/10 benchmark executions.

Rank Scores. From the QpS results, it is rather hard to determine the overall winner in terms of the query runtime performance. The

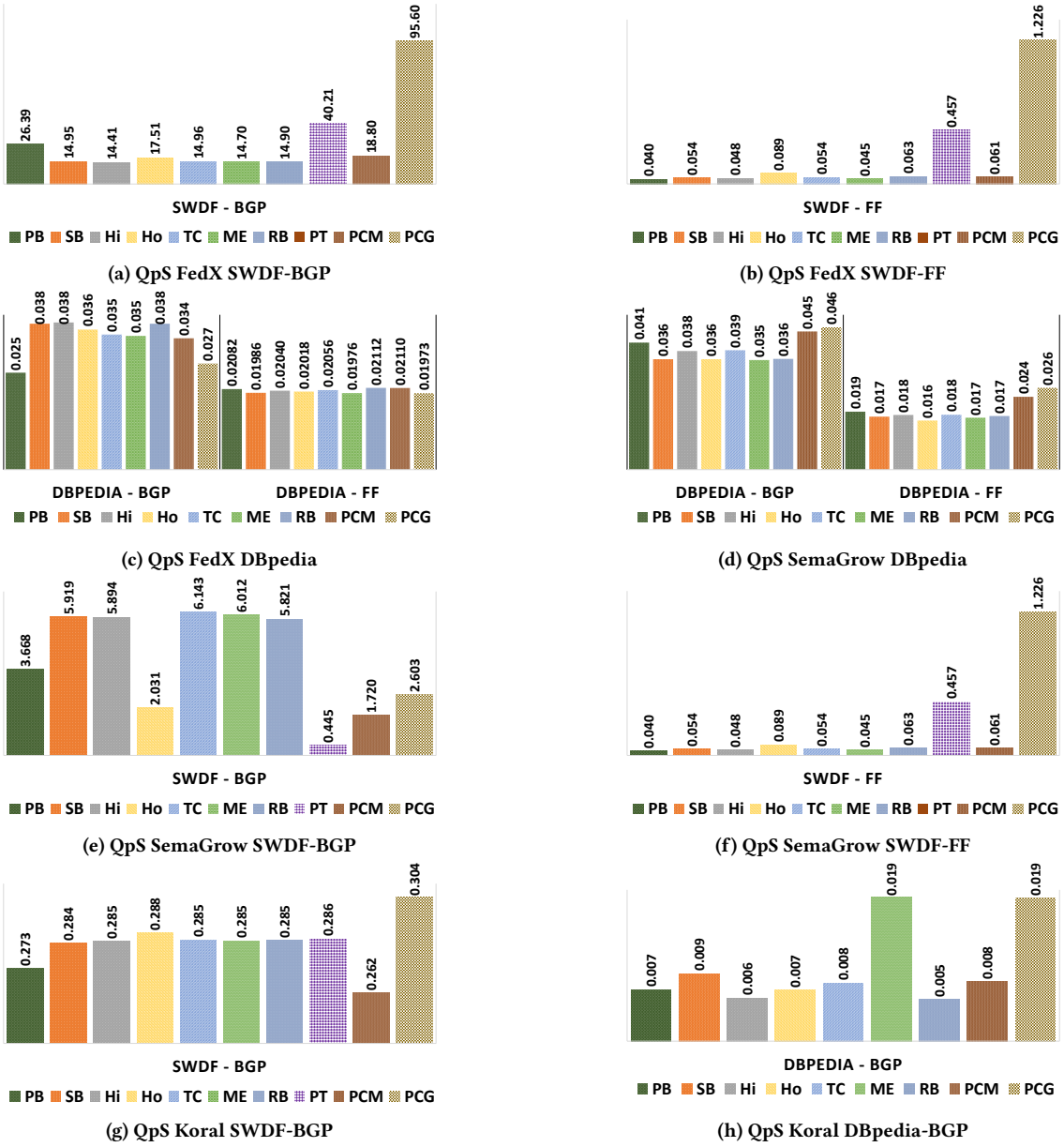
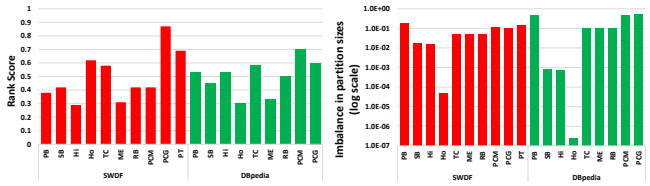


Figure 7: QpS (for all four benchmarks) including timeouts. (SW = Semantic web dog food, DB = DBpedia, BGP = Basic Graph Pattern, FF = Fully Featured, PB = Predicate-Based, SB= Subject-Based, Hi= Hierarchical, Ho = Horizontal, TC = TCV-Min, ME Min-Edgecut, RB = Recursive Bisection, PT = Partout)

rank score shows the overall ranking of a particular method with respect to other selected methods across the completed benchmark executions. The rank score is a value between 0 and 1, where 1 represents the highest ranking. Figure 8a represents the computed rank scores pertaining to each partitioning technique according to Theorem 4.1. The overall results show that that PCG has the highest ranked score, followed by PT, PCM, TCV-Min, Predicate-Based, Horizontal, Recursive-Bisection, Subject-Based, Hierarchical, and Min-Edgecut, respectively.

Partitioning Imbalance. Figure 8b shows the partitioning imbalance values (defined in Theorem 4.2) of the partitions generated by the selected partitioning techniques. Horizontal portioning results in the smallest partitioning imbalance, followed by Hierarchical, Subject-Based, PCM, PCG, Min-Edgecut, Recursive-Bisection, TCV-Min, Partout, and Predicate-Based, respectively.

Number of Sources Selected. The number of sources selected (SPARQL endpoints in our case) by the federation engine to execute a given SPARQL query is a key performance metric for federated



(a) Rank scores (b) Partitioning imbalance

Figure 8: Rank scores and partitioning imbalance of the partitioning techniques. (PB = Predicate-Based, SB= Subject-Based, Hi= Hierarchical, Ho = Horizontal, TC = TCV-Min, ME Min-Edgecut, RB = Recursive Bisection, PT = Partout)

SPARQL querying engines [16]. The smaller the number of sources selected, the smaller the communication cost, and hence the better the query runtime performance [2, 16]. Figure 9 shows the total number of distinct sources selected by FedX and SemaGrow. For SWDF, PT selects the smallest sources followed by PCG and PCM. As an overall (1200 queries) source selection evaluation, PCG selects the least number of sources, followed by PCM, Predicate-Based, Min-Edgecut, TCV-Min, Recursive-Bisection, Subject-Based, Hierarchical and Horizontal, respectively.

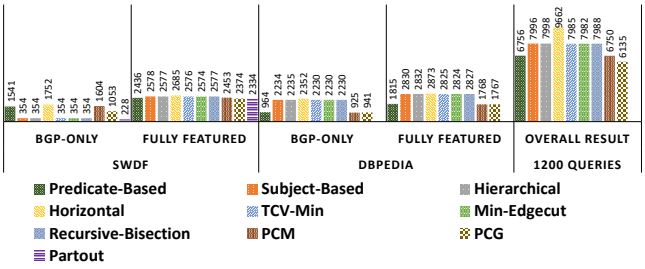


Figure 9: Total distinct sources selected

Key observation. The results show that PCG significantly outperformed the other selected techniques for SWDF benchmarks (ref. Figure 7a, Figure 7b, Figure 7e, Figure 7f, Figure 7g) in comparison to to DBpedia benchmarks (ref. Figure 7c, Figure 7d, Figure 7h). The average QpS of PCG is 20.07 for SWDF benchmarks, which is 3.30 times faster than the second-best performing partitioning method. On the other hand, the average QpS of PCG is 0.028 for DBpedia benchmarks which is only 1.06 times faster than the second-best performing partitioning method. A detailed investigation of query workload and RDF datasets reveals that the query workload used for our SWDF evaluation already covered 63.7% of the total 185 predicates used in the SWDF dataset. Thus, more predicates were correctly grouped into the desired partitions. On the other hand, the DBpedia dataset contains a total of 39672 distinct predicates and only 0.55% were covered by the used workload. As a result, a majority of the predicates were grouped into a separate partition of unused predicates. Consequently, only a small portion of the predicates were correctly mapped into correct partitions. In conclusion, the complexity of dataset and workload's quality and size can have a significant impact on the quality of partitioning achieved via the proposed methods in this paper.

5 CONCLUSION AND FUTURE WORK

In this paper, we presented two RDF graph partitioning techniques based on querying workloads that leverage the predicate co-occurrences in these workloads. Our overall results suggest the superiority of our proposed 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. It has been observed that the quality and size of the workload is key to achieving better results via the proposed methods. The partitioning techniques that take the data locality (i.e., data chunks that are queried together by users are kept in same partition) into account can lead to significant performance improvements. Our proposed techniques naturally lead to predicate-based indexing used in existing state-of-the-art RDF engines. Furthermore, the created partitions are easy to manage in terms of index updates or dynamic shuffling of data among multiple data nodes of a clustered triplestore. In the future, we want to measure the effect of querying workloads on the accuracy of data distribution. In addition, it is highly possible that the initial distribution of data is sub-optimal and thus dynamic shuffling of data is necessary. To this end, we want to propose a self updating, dynamic data distribution mechanism based on experienced work-load.

6 ACKNOWLEDGEMENTS

This work has been supported by the projects 3DFed(Grant no. 01QE2114B) and KnowGraphs(Grant no. 860801).

REFERENCES

[1] Abadi et al. 2007. Scalable Semantic Web Data Management Using Vertical Partitioning. (2007).
[2] Akhter et al. 2018. An empirical evaluation of RDF graph partitioning techniques. In *European Knowledge Acquisition Workshop*.
[3] Al-Ghezi et al. 2018. Adaptive workload-based partitioning and replication for RDF graphs. In *International Conference on Database and Expert Systems Applications*.
[4] Aluç et al. 2013. chameleon-db: a workload-aware robust RDF data management system. *University of Waterloo, Tech. Rep. CS-2013-10* (2013).
[5] Erling et al. 2009. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*.
[6] Galárraga et al. 2014. Partout: A Distributed Engine for Efficient RDF Processing.
[7] Graux et al. 2016. Sparqlgx: Efficient distributed evaluation of sparql with apache spark.
[8] Gurajada et al. 2014. TriAD: A Distributed Shared-Nothing RDF Engine Based on Asynchronous Message Passing.
[9] Harth et al. 2007. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *The Semantic Web*.
[10] Huang et al. 2011. Scalable SPARQL querying of large RDF graphs. (2011).
[11] Janke et al. 2017. Koral: A Glass Box Profiling System for Individual Components of Distributed RDF Stores.
[12] Lehmann et al. 2017. Distributed Semantic Analytics using the SANSA Stack. In *Proceedings of 16th International Semantic Web Conference-Resources Track*.
[13] Madkour et al. 2018. WORQ: Workload-driven RDF Query Processing.
[14] Padiya et al. 2017. DWAHP: workload aware hybrid partitioning and distribution of RDF data.
[15] Saleem et al. 2015. Feasible: A feature-based sparql benchmark generation framework. In *International Semantic Web Conference*.
[16] Saleem et al. 2016. A fine-grained evaluation of SPARQL endpoint federation systems. (2016).
[17] Schätzle et al. 2016. S2RDF: RDF Querying with SPARQL on Spark. (2016).
[18] Schwarte et al. 2011. Fedx: Optimization techniques for federated query processing on linked data.
[19] Waqas et al. 2020. Storage, Indexing, Query Processing, and Benchmarking in Centralized and Distributed RDF Engines: A Survey. (2020).
[20] Whitman et al. 2019. Distributed Spatial and Spatio-Temporal Join on Apache Spark. (2019).