Eurostars Project

3DFed – Dynamic Data Distribution and Query Federation

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

Deliverable 3.3
Initial Report on the Dynamic Data Exchange

Dissemination Level	Public
Due Date of Deliverable	November 30, 2022
Actual Submission Date	June 15, 2023
Work Package	WP3, Automatic Data Distribution & Dynamic Exchange
Deliverable	D3.3
Type	Report
Approval Status	Final
Version	1.0
Number of Pages	15

Abstract: In deliverables D3.1 and D3.2, we proposed two novel algorithms namely PCG and PCM for static distribution of RDF knowledge graphs. These algorithms are based on querying history, which changes with time. As such, we need to adopt the proposed distribution according to the change in querying history with time. To this end, we need a means to dynamic data exchange between data nodes according to the new querying workload. We propose a method to do the dynamic data exchange between nodes and do initial evaluation based on the Semantic Web Dog Food dataset. The evaluation results certainly hinted at the usefulness of the proposed method.

3DFed Project by Eurostars.

History

Version	Date	Reason	Revised by
0.1	21/04/2023	Initial Template & Deliverable Structure	Mohammad Sajjadi
0.2	16/05/2023	Initial Draft	Asal Alikhani
0.3	06/06/2023	Input on Draft	Muhammad Saleem
0.4	08/06/2023	Issued for review	Mohammad Sajjadi
0.5	14/06/2023	Review	Milos Jovanovik
1.0	15/06/2023	Final Submission	Mohammad Sajjadi

Author List

Organization	Name	Contact Information
elevait GmbH & Co. KG	Asal Alikhani	asal.alikhani@elevait.de
University of Paderborn	Muhammad Saleem	saleem.muhammd@gmail.com
elevait GmbH & Co. KG	Mohammad Sajjadi	mohammad.sajjadi@elevait.de
OpenLink Software	Milos Jovanovik	mjovanovik@openlinksw.com

Contents

1 Introduction

Data partitioning helps improve the scalability, availability, ease of maintenance, and overall query processing performance of storage systems. This is achieved by splitting a big dataset into subsets and distributing it over multiple partitions. In D3.2, we proposed 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. The proposed approach leads to better query runtime because the fewer partitions being consulted by the distributed RDF engine to execute SPARQL triple patterns. This inter-communication cost between multiple worker nodes of the distributed RDF engines was decreased. The proposed *predicate-based* partition has inherent advantages, such as it leads to a natural predicate-based index.

The workload-based static data distribution can have inherent problems because the user queries may change over time. As such, the resulting partition scheme might not be very efficient. In this case, it might be required to construct new partitions according to the new querying workload. In the proposed 3DFed engine, once the data is distributed initially and is being used in practice, the next step is to dynamically exchange the data between the storage solutions provided that it can lead to further performance improvements in query federation. To this end, we then make use of the fresh query logs to dynamically exchange the data between the data storage solutions. The overall goal is to improve the query runtime performance and distribute the load among data storage solutions to improve their availability. Consequently, this will facilitate the development of high-performance federation engines. We aim to exchange the data between storage solutions dynamically and exploit data locality to maximise/balance the amount of computation in a single storage solution. The dynamic exchange of data can be a deletion or an insertion.

Our proposed technique makes use of clustering algorithms (proposed in D3.2) 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 dynamic distribution approach, we recreate clusters based on new workload and compare with previous clusters. If there exist changes between the previous and current clusters, we perform all those changes and new partitions are then created according to suggested changes. For example, assuming that in first workload W1, we get first partition P1 with predicates list $\{p_1, p_2\}$ and second partition $P_2 = \{p_3, p_4\}$. Now let's assume with the more recent workload W2, we again created clusters with $P_1 = \{p_1, p_2, p_3\}$ and $P_2 = \{p_4\}$. This means we need to remove all triples with predicate p_3 from partition P_2 and move them to P_1.

2 RDF Graph Partitioning

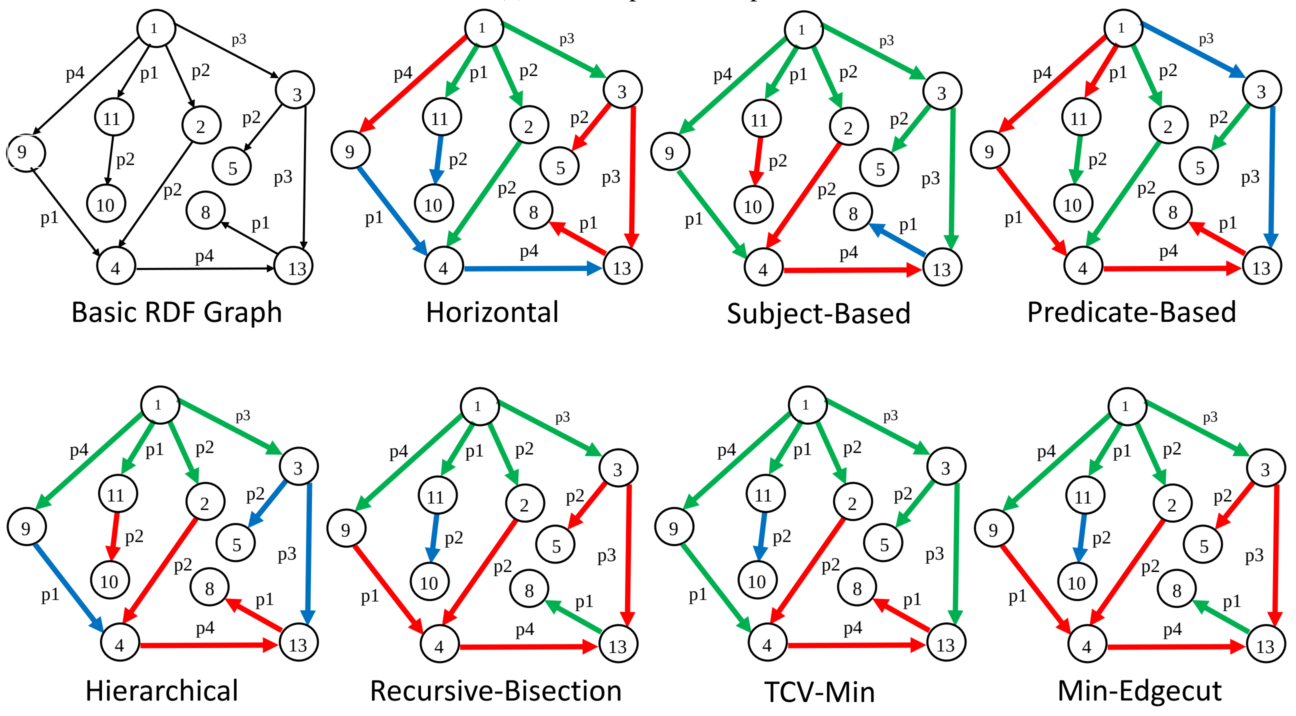
In distributed RDF engines, the given data needs to be distributed among multiple data nodes. The partitioning of big data among multiple data nodes helps in improving systems availability, ease of maintenance, and overall query processing performances. The RDF graph partitioning problem is defined as follows.

Definition 1 (RDF Graph Partitioning Problem) *Given an RDF graph $G = (V, E)$, divide G into n sub-graphs $G_1, \dots G_n$ such that $G = (V, E) = \bigcup_{i=1}^{n} G_i$, where V is the set of all vertices and E is the set of all edges in the graph.*

A recent empirical evaluation [1] of the different RDF graph partitioning showed that the type of partitioning used in the RDF engines has a significant impact on the query runtime performance. They conclude that the data that is queried together in SPARQL queries should be kept in the same node, thus minimizing the network traffic

```
@prefix hierarchy1: <http://first/r/> . @prefix hierarchy2: <http://second/r/> .
@prefix hierarchy3: <http://third/r/> . @prefix schema: <http://schema/> .
hierarchy1:s1       schema:p1        hierarchy2:s11 .  #Triple 1
hierarchy1:s1       schema:p2        hierarchy2:s2 .   #Triple 2
hierarchy2:s2       schema:p2        hierarchy2:s4 .   #Triple 3
hierarchy1:s1       schema:p3        hierarchy3:s3 .   #Triple 4
hierarchy3:s3       schema:p2        hierarchy1:s5 .   #Triple 5
hierarchy3:s3       schema:p3        hierarchy2:s13 .  #Triple 6
hierarchy2:s13      schema:p1        hierarchy2:s8 .   #Triple 7
hierarchy1:s1       schema:p4        hierarchy3:s9 .   #Triple 8
hierarchy3:s9       schema:p1        hierarchy2:s4 .   #Triple 9
hierarchy2:s4       schema:p4        hierarchy2:s13 .  #Triple 10
hierarchy2:s11      schema:p2        hierarchy1:s10 .  #Triple 11
```

(a) An example 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.

among data nodes. In this section, we explain commonly used [4, 6, 9, 7] graph partitioning techniques by using a sample RDF graph shown in Figure 1[1]. In this example, we want to partition the 11 triples into 3 partitions namely green, red, and blue partitions.

Horizontal Partitioning: It is the simplest form of the RDF partitioning which assumes the data is presented in N-triples format and one line per triple). This partitioning technique is adopted from [7] and is not commonly used in state-of-the-art RDF graph partitioning. Let n be the required number of partitions and T be the set of all

[1]We used different example to show a clear difference between the discussed RDF partitioning techniques.

RDF triples in a dataset. The technique assigns (horizontally) the first $|T|/n$ triples in partition 1, the next $|T|/n$ triples in partition two and so on. Using this technique, our example dataset is split such that, triples 1-4 are assigned into green partition, triples 5-8 into are assigned into red partition, and triples 9-11 are assigned into blue partition. The evaluation [1] shows that this technique does not lead to the better query runtime performance as compared to other techniques discussed below.

Vertical Partitioning:

This technique divides the given RDF dataset based on predicates. Ideally, the number of distinct partitions would be equal to the number of distinct predicates used in the dataset. However, it is possible that the number of available data nodes (i.e. the required number of partitions) may be smaller than the number of predicates used in the dataset. In this case the predicate tables are grouped into data nodes, i.e., multiple predicate tables are stored in available data nodes. There can be multiple way of grouping predicate partitions: (1) first come first serve, (2) by looking at the number of triples per predicate and thus group predicates such that maximum load balancing is achieved among data nodes, (3) using some intelligence to determine which predicates will be queried together, and hence grouped their corresponding triples in one partition.

In our motivating example, there are four distinct predicates while the required number of partitions are 3. Thus by using the first come for serve strategy, all the triples with predicate $p1$ will be assigned to first partition (red), $p2$ triples will be assigned to second partition (green), $p3$ triples will be assigned to third partition (blue), and $p4$ triples will be again assigned to first partition. This technique can lead to significant performance improvement, provided that the predicates are intelligently grouped into partitions, such that communication load among data nodes is reduced[1].

Hash-Based Partitioning: There are two techniques used in this category:

- **Subject-Hashed.** This technique assigns triples to partitions according to a the hash value computed on their subjects modulo the total number of required partitions (i.e., hash(subject) modulus total number of partitions)[5]. Thus, all the triples with the same subject are assigned to one partition. However, due to the modulo operation, this technique may result in high partitioning imbalance. In our motivating example given in Figure 1, Using this technique, our example dataset is split such that, triples 3,10 and 11 are assigned into red partition, triple 7 is assigned into blue partition, and the remaining triples are assigned into green partition. Thus, a clear partitioning imbalance (3:1:7 triples) results.
- **Hierarchical Partitioning:** This partitioning is inspired by two assumptions: (1) IRIs have path hierarchy, (2) IRIs with a common hierarchy prefix are often queried together [5]. This partitioning technique extracts path hierarchies from the IRIs and assigns triples having the same hierarchy prefixes into one partition. For instance, the extracted path hierarchy of “http://www.w3.org/1999/02/22-rdf-syntax-ns#type" is “org/w3/www/1999/02/22-rdf-syntax-ns/type". Then, for each level in the path hierarchy (e. g., “org", “org/w3", “org/w3/www", ...) it computes the percentage of triples sharing a hierarchy prefix. If the percentage exceeds an empirically defined threshold and the number of prefixes is equal to or greater than the number of required partitions at any hierarchy level, then these prefixes are used for the hash-based partitioning on prefixes. In comparison to the subject-hash-based partition, this technique requires a higher computational effort to determine the IRI prefixes on which the hash is computed. In our motivating example given in Figure 1, all the triples having `hierarchy1` in subjects are assigned to the green partition, triples having `hierarchy2` in subjects are assigned to the red partition, and triples having `hierarchy3` in subjects are assigned to the blue partition. This partitioning may not produce the best query runtimes as the underlying assumptions about IRIs might not be true in practice [1].

Graph-Based Partitioning: It makes use of the graph-based clustering techniques to split a given graph into the required pairwise disjoint sub-graphs. There are three techniques used in this category:

- **Recursive-Bisection Partitioning.** Recursive bisection is a multilevel graph bisection algorithm aiming to solve the k-way graph partitioning problem as described in [6]. This algorithm consists of the following three phases: (1)) *Coarsening:* The initial phase is coarsening the graph, in which a sequence of smaller graphs $G_1, G_2, ..., G_m$ is generated from the input Graph $G_0 = (V_0, E_0)$ in such a way that $|V_0| > |V_1| > |V_2| > ... > |V_m|$. (2) *Partitioning* In the second phase, computation of a 2-way partition P_m of the graph G_m takes place, such that V_m is split into two parts and each part contains half of the vertices. (3) *Uncoarsening* The third and last phase is uncoarsening the partitioned graph. In this phase the partition P_m of G_m is projected back to G_0 by passing through the intermediate partitions $P_{m-1}, P_{m-2}, ..., P_1, P_0$.

  In our motivating example given in Figure 1, triples 1, 2, 4, 7, and 8 are assigned into green partition, triples 3, 5, 6, 9 and 10 are assigned into red partition, and triple 11 is assigned into blue partition.

- **TCV-Min Partitioning.** Similar to Recursive-Bisection, the TCV-Min also aims to solve the k-way graph partitioning problem. However, the objective of the partitioning is to minimize the *total communication volume* [2] of the partitioning. Thus, this technique also comprises the three main phases of the k-way graph partitioning. However, the objective of the second phase, i.e. the *Partitioning*, is the minimization of communication costs. In our motivating example given in Figure 1, triples 1, 2, 4, 5, 6, 8 and 9 are assigned into green partition, triples 3, 7 and 10 are assigned into red partition, and triple 11 is assigned into blue partition.

- **Min-Edgecut Partitioning.** The Min-Edgecut [6] also aims to solve the k-way graph partitioning problem. However, unlike TCV-Min, the objective is to partition the vertices by minimizing the number of edges connected to them. In our motivating example given in Figure 1, triples 1, 2, 4, 7 and 8 are assigned into green partition, triples 3, 5, 6, 9 and 10 are assigned into red partition, and only triple 11 is assigned into blue partition.

The graph-based partitioning techniques are computational complex, and may be very strong for splitting big RDF datasets.

Workload-Based Partitioning: The partitioning techniques in this category make use of the query workload to partition the given RDF dataset. Ideally, the query workload 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, the real user queries might not be available. In this case the query workload can either be 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.

The Figure 2 below shows different partitioning schemes found in distributed RDF engines.

3 Approach

In this section, we explain the proposed approach. Some of the data and text is re-used from D3.2 for the sake of completeness.

In the following, we suppose we have a workload of eight queries as shown in listing 1.

In discussed in D3.2, 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).

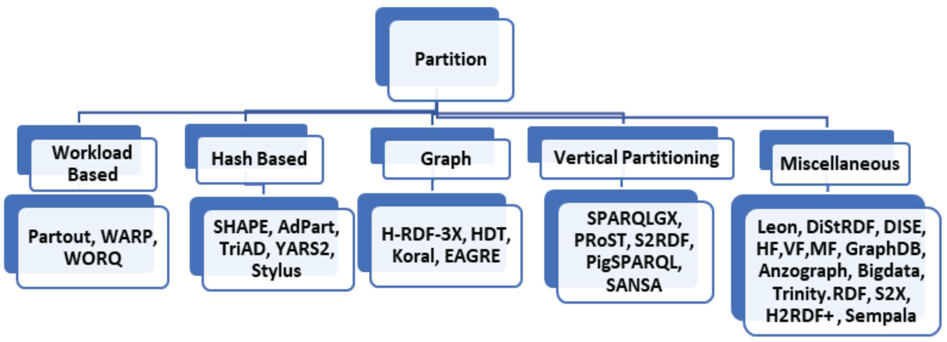

Partition
Workload Based
Hash Based
Graph
Vertical Partitioning
Miscellaneous
Partout, WARP, WORQ
SHAPE, AdPart, TriAD, YARS2, Stylus
H-RDF-3X, HDT, Koral, EAGRE
SPARQLGX, PRoST, S2RDF, PigSPARQL, SANSA
Leon, DiStRDF, DISE, HF,VF,MF, GraphDB, Anzograph, Bigdata, Trinity.RDF, S2X, H2RDF+, Sempala

Fig. 2. Types of Partitioning in centralized and Distributed RDF Engines(*Bold represents the Distributed RDF Engines)

```
 SELECT * WHERE          SELECT * WHERE          SELECT * WHERE          SELECT *
    WHERE
{                       {                        {                      {
   ?S ?P1 ?O1 .          ?S ?P1 ?O.            ?S1 ?P1 ?O.            ?O ?P1 ?S.
   ?S ?P3 ?O3            ?O ?P3 ?O3            ?S3 ?P3 ?O             ?S ?P3 ?S3
}                       }                        }                      }

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

Listing 1: Query examples

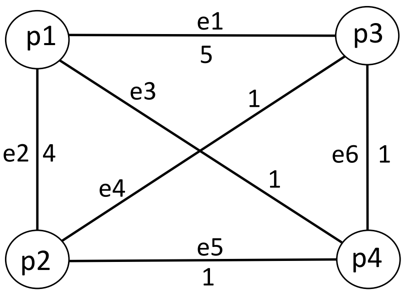
3.1 Graph Modeling

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 workload $Q = \{q_1, \ldots, q_n\}$, 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 3a. 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 3b.

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

Algorithm 1: Adapted Markov Clustering

```
1 MCL($G$, $e$, $r$ ,$n$) /* Input: Weighted predicates graph $G$, sequence of powers $e = 2$,
     sequence of inflation parameters $r = 0.001$, 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 while $\exists_{c \in \{1,\ldots,p\}} (\sum_{r=1}^{p} T_{r,c}) > 1$ do
5     $T := (T)^e$ // Expend
6     $T :=$ Inflate($r$, Power($e, T$)) // Inflate
7 end
8 return $getClusters(T, n)$ /* get $n$ clusters from matrix */
```

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[2] 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 3b 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 4. The normalization of the matrix is done by dividing each element of a particular column by the sum of all the elements in that column. The normalized matrix is shown in Figure 4.

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 there is no column in the transition matrix with a total sum greater than 1 (Lines 4-8 of algorithm 1). The expansion is a simple self-multiplication of the matrix, raised to the power of input parameter e. The inflation matrix results from re-scaling each of the columns of T with power coefficient r. We encourage readers to have a look at the standard Markov clustering algorithm for further details and effects of these steps.

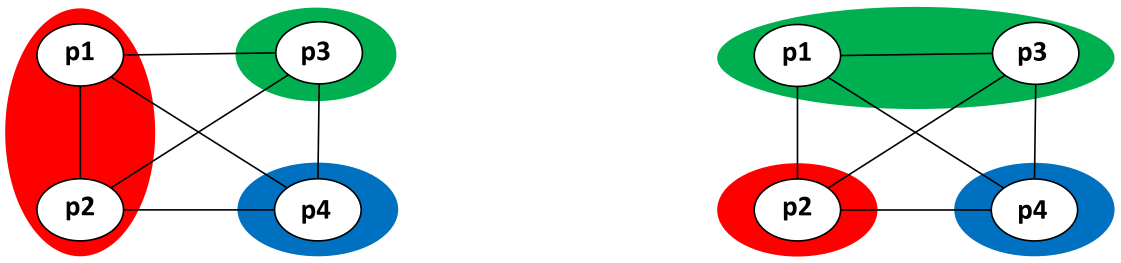
The last step is to interpret the resulting transition matrix to discover n clusters. This is achieved by

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

$$
\begin{array}{c|cccc|}
 & \text{P1} & \text{p2} & \text{p3} & \text{p4} \\
\hline
\text{P1} & 0 & 4 & 1 & 5 \\
\text{P2} & 4 & 0 & 1 & 1 \\
\text{P3} & 5 & 1 & 0 & 1 \\
\text{p4} & 1 & 1 & 1 & 0 \\
\hline
\end{array}
\longrightarrow
\begin{array}{c|cccc|}
 & \text{P1} & \text{p2} & \text{p3} & \text{p4} \\
\hline
\text{P1} & 0 & 4/6 & 1/3 & 5/7 \\
\text{P2} & 4/10 & 0 & 1/3 & 1/7 \\
\text{P3} & 5/10 & 1/6 & 0 & 1/7 \\
\text{p4} & 1/10 & 1/6 & 1/3 & 0 \\
\hline
\end{array}
\longrightarrow
\begin{array}{c|cccc|}
 & \text{P1} & \text{p2} & \text{p3} & \text{p4} \\
\hline
\text{P1} & 0 & 0.66 & 0.33 & 0.71 \\
\text{P2} & 0.4 & 0 & 0.33 & 0.14 \\
\text{P3} & 0.5 & 0.16 & 0 & 0.14 \\
\text{p4} & 0.1 & 0.16 & 0.33 & 0 \\
\hline
\end{array}
$$

Transition Martix Normalized Martix Final Martix

Figure 4: Creation of a matrix during PCM using our weighted graph



(a) PCM (b) PCG

Figure 5: Predicate clusters created by the proposed techniques for the example RDF dataset given in Figure 1a. Clusters are highlighted in different colors)

sequentially adding non-zero row-wise values of matrix T to a cluster. For example, in our final matrix shown in Figure 4, 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 $T_{1,3}$, which corresponds to predicates p_1, p_3. Since p_1 already exists, only $p3$ will be added into the cluster. Finally, p_4 will be added. Now our cluster contains a sequential list of predicates $\{p_1, p_2, p_3, p_4\}$. 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_3) into partition 2 and the final element into partition 3. The final cluster of predicates is shown in figure 5a. 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 3b, 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 the 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 5b. 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.

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  $C = \{c_1 \ldots c_n\}$ ;                                              // 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
```

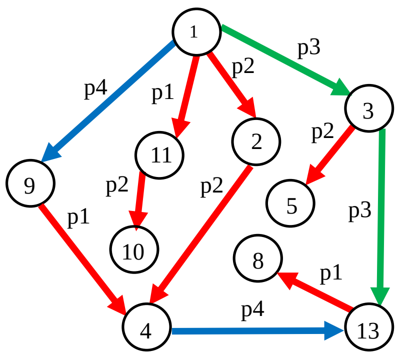
3.3 Assigning Clusters to Partitions

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 6a and 6b 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.

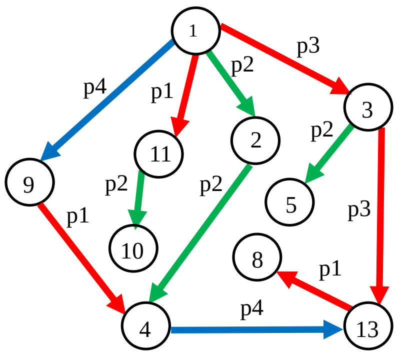
3.4 Dynamic Data Distribution

Now we explain the dynamic data distribution according to the new workload. We propose to collect a one week query log and make a new data distribution, which better reflects the new workload. We follow the following steps to perform this task.

- We get a querying workload W1 and perform predicate-clustering using PCM or PCG. The corresponding cluster of predicates C1 is then assigned to physical partitions as discussed before.
- The triplestore is then used in practice for one week more and the new workload W2 is collected. We perform again the predicate-clustering C2 using PCM or PCG using the new workload.
- We compare C1 and C2 for any changes, i.e., we check if predicate clusters are changed in the C2 w.r.t C1. If there occurs no change, we do not make any dynamic data distribution among physical partitions. If there exist changes, we carry on the required changes (insertion or deletion of triples) in the current physical partitions to exactly reflect the C2.
- We repeat these steps every week or every two weeks, depending on the workload querying frequency.



(a) PCM



(b) PCG

Figure 6: 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 [1] and D3.2. 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 [1].

Datasets. In this initial report, we only used *Semantic Web Dog Food (SWDF)* for partitioning. We chose this dataset because it is small and can be used to get a quick look for the initial performance. In the final deliverable, we will also use *DBpedia 3.5.1* for partitioning. The real-world query logs for the SWDF dataset are freely available from LSQ dataset [8].

Workloads (train) and Benchmark (test) Queries. With the help of ten queries such as listing 2, we fetched SELECT BGP-only queries of SWDF ordered by execution time stamp, and divided them by 10 parts of 49 queries. Each consecutive pair of these parts was used as a pair of train and test queries in one experiment. In each experiment first part was used for partitioning and distributing, and the second part was used for testing. So we used just BGP-only queries for partitioning and benchmarking, because the selected partitioning environment for this initial deliverable does not support fully-featured queries. *SWDF BGP-only* is the SWDF benchmark containing only single BGP queries; the other SPARQL features such as `OPTIONAL`, `UNION` etc. are not used.

Partitioning Environments. In this initial report, we only used a clustered or distributed RDF storage environment, where the given dataset is distributed among n data nodes of a clustered triplestore. The second partitioning environment, i.e., 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, will be used in D3.4. We used *Koral* [3] 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 among data nodes. In addition, it was previously used in [1].

Number of Partitions. Inspired by D3.2, [1] and [7], we generated 10 partitions of the selected datasets. Therefore, 10 slaves were created in Koral, each one responsible for one partition.

Selected RDF Graph Partitioning Technique. We used the PCM clustering algorithm because it has proven to be performing better than PCG in terms of clustering generation.

```
# fetch the second 49 queries of the SELECT BGP-only-queries of
    Semantic Web Dog Food ordered by timestamps

#****without limit results=497

Prefix lsq: <http://lsq.aksw.org/vocab#>
PREFIX prov: <http://www.w3.org/ns/prov#>

SELECT  Distinct ?qId ?tps ?joinVertices ?rs ?rt ?meanJoinVertexDegree
     #?text ?timeStamp

WHERE
{

?qId  lsq:text ?text .
?qId  lsq:hasRemoteExec ?re .
?re  prov:atTime ?timeStamp .

?qId  lsq:hasStructuralFeatures ?sf .
?sf   lsq:joinVertexCount ?projVar .
?sf   lsq:joinVertexCount ?joinVertices .
?sf   lsq:tpCount ?tps .
?sf   lsq:joinVertexDegreeMean ?meanJoinVertexDegree .
?sf   lsq:usesFeature  lsq:Select  .

?qId  lsq:hasLocalExec ?le .
?le   lsq:hasQueryExec ?qe .
?qe   lsq:resultCount ?rs.
?qe   lsq:evalDuration ?rt.


Filter (?rs > 0 && ?rs < 20000000 && ?tps > 1) #queries which have
    some result and more than 1 triple pattern

#bgp-only-filter
FILTER(!regex(?text, "(lsq:fn-if|agg-group_concat|Aggregators|AltPath|
    Bind|Distinct|Filter|Functions|GroupBy|InversePath|Limit|LinkPath|
    Minus|NamedGraph|Offset|Optional|OrderBy|SeqPath|Service|SubQuery|
    TriplePath|Union|Values|ZeroOrMorePath|ZeroOrOnePath|agg-count|agg-
    min|agg-sample|agg-sum|fn-and|fn-bound|fn-contains|fn-day|fn-eq|fn-
    exists|fn-function|fn-ge|fn-gt|fn-in|fn-isIRI|fn-isLiteral|fn-lang|
    fn-langMatches|fn-lcase|fn-le|fn-lt|fn-month|fn-ne|fn-not|fn-
    notexists|fn-notin|fn-now|fn-or|fn-regex|fn-sameTerm|fn-str|fn-
    strstarts|fn-substr|fn-subtract|fn-year|oneOrMorePath|fn-abs|fn-
    isBlank|fn-multiply|agg-max|Describe|fn-add|fn-datatype|fn-concat|
    fn-strends|agg-avg|Ask|Construct|fn-strafter|fn-strlen|fn-replace|
    fn-ucase|fn-encode_for_uri|Reduced|fn-divide|fn-round|fn-isNumeric|
    fn-strbefore|fn-strlang|fn-seconds|fn-coalesce|fn-strdt|fn-floor|fn
    -hours|fn-iri|fn-minutes|lsq:Group|lsq:TriplePattern|count| as )",
    "i"))
}

order by ?timeStamp
LIMIT 49 OFFSET 50
```

Listing 2: Query for fetching train and test queries (490 queries- 10 parts of 49 queries)

Performance Measures. We used Queries per Second (QpS), and the average query execution time to compare the performance of the proposed dynamic data exchange. We used a six minutes timeout for query execution of each query.

Hardware and Software Specifications. The hardware and software configuration for our techniques is the same as [1], i.e., all our experiments are executed on a Ubuntu-based machine with intel i7-11370H 3.30 GHz, 4 cores and 32GB of RAM. We conducted our experiments on local copies of Tenforce / Virtuoso (version 7.2.5) SPARQL endpoints. We used default configurations for Koral, but the slaves were changed from 2 to 10.

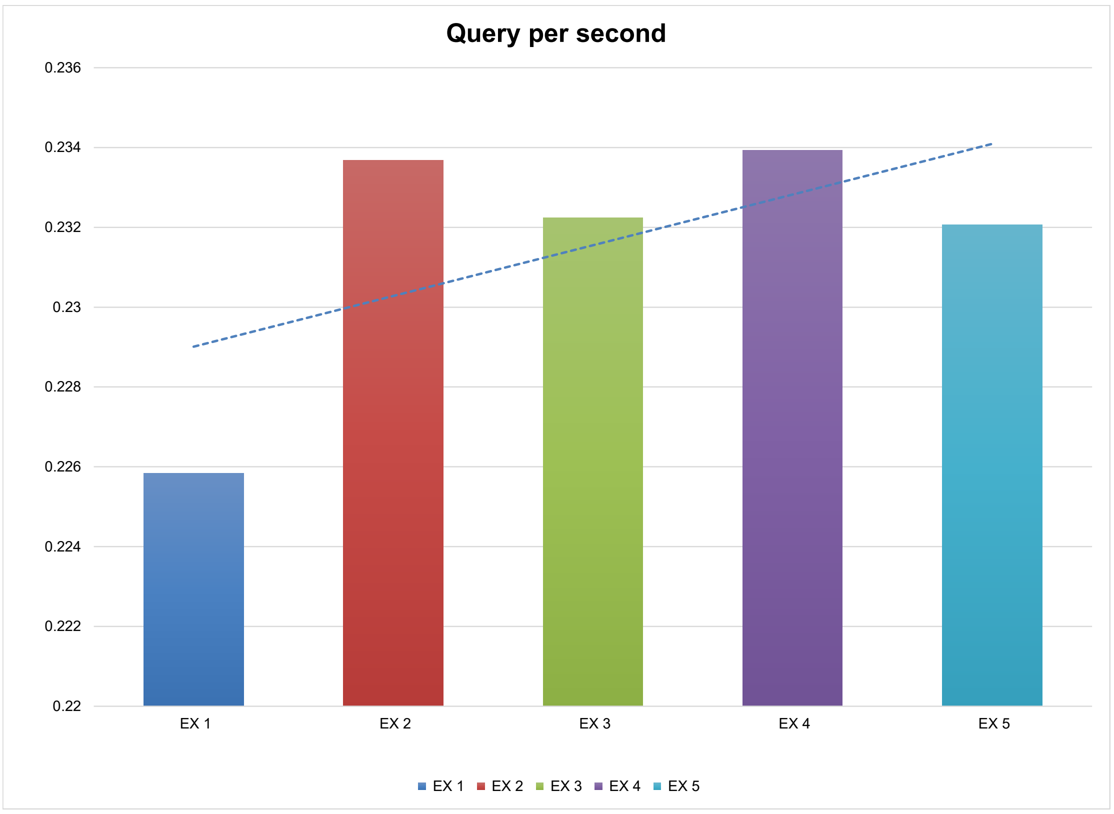


Figure 7: Queries per Second (QpS) with sequential one week querying workload (Ex1-Ex5).

4.2 Evaluation Results

The goal of our evaluation is to show how the query runtime performance is improved with dynamic data partitioning.

Query per Second (QpS). Figure 7 shows a comparison of the QpS values for 5 querying workloads. The higher the QpS the better the query runtime performance. Each query load is before its benchmark queries. The Ex1 represents the initial workload and benchmark pair (each of 49 queries). Ex2 is the next pair of workload and benchmark and so on. We did data distribution according to workload queries of Ex1 and executed benchmark queries of Ex1 and reported QpS. We follow the same process for Ex2 until Ex5. We can clearly see the QpS (in general) is improved with dynamic data distribution while going from Ex1 to Ex5. This shows the effectiveness of the proposed dynamic data distribution. However, we also see the increase in QpS is not linear. For example the QpS of Ex2 is better than Ex3. The possible reason is that the selected SWDF dataset is very small. There are not many big changes in the querying workload as well. We believe experiments on DBpedia will give a more clear picture. However, the initial results are sufficiently satisfactory.

Average Query Runtime. Figure 8 shows a comparison of the average query runtime for the same 5 querying workloads. The smaller the runtime the better the query execution performance. Again, the result suggests that query runtime is decreased with dynamic data distribution while going from Ex1 to Ex5. This shows the effectiveness of the proposed dynamic data distribution.

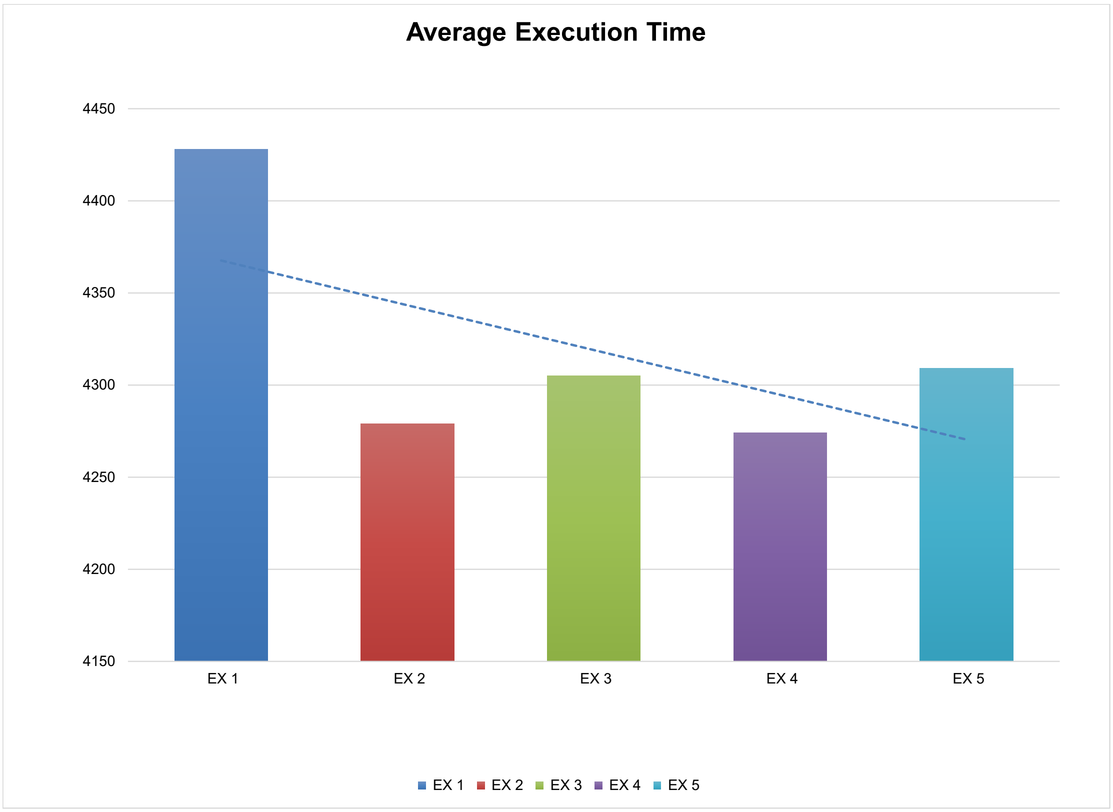


Figure 8: Average query runtime with sequential one week querying workload (Ex1-Ex5).

5 Conclusion and Future Work

In this deliverable, we presented the initial report on dynamic data exchange. We made use of the 5 querying workloads from SWDF and did dynamic data shuffling according to the new workloads. We used QpS and the average query runtime as two performance measures. The results clearly show the effectiveness of the proposed dynamic data partitioning. However, further detailed experiments are needed to draw solid conclusions.

In the future, we plan to use more datasets and different partitioning environments to test the proposed dynamic data exchange techniques.

References

[1] Adnan Akhter, Axel-Cyrille Ngomo Ngonga, and Muhammad Saleem. An empirical evaluation of rdf graph partitioning techniques. In *European Knowledge Acquisition Workshop*, pages 3–18. Springer, 2018.

[2] Aydin Buluc, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning, 2013.

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

[4] Daniel Janke and Steffen Staab. Storing and querying semantic data in the cloud. In *Reasoning Web International Summer School*, pages 173–222. Springer, 2018.

[5] Daniel Janke, Steffen Staab, and Matthias Thimm. On data placement strategies in distributed rdf stores. In *Proceedings of The International Workshop on Semantic Big Data*, SBD '17, New York, NY, USA, 2017. Association for Computing Machinery.

[6] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[7] Saleem et al. A fine-grained evaluation of sparql endpoint federation systems. 2016.

[8] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. LSQ: the linked SPARQL queries dataset. In *ISWC*, pages 261–269. Springer, 2015.

[9] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF querying with SPARQL on Spark. *Proceedings of the VLDB Endowment*, 9(10):804–815, 2016.