Eurostars Project

3DFed – Dynamic Data Distribution and Query Federation

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

Deliverable 5.3
A report on 3DFed evaluation based on document data use case

Dissemination Level	Public
Due Date of Deliverable	March 31, 2024
Actual Submission Date	March 31, 2024
Work Package	WP5, Use Cases
Deliverable	D5.3
Type	Report
Approval Status	Final
Version	1.0
Number of Pages	14

Abstract: In this deliverable we evaluate dynamic data exchange between data nodes against a RDF datset of elevait as a real word use case. We propose a method to compare the dynamic partitioning vs. static partitioning and evaluate both method based on the elevait dataset in terms of different measurs. The evaluation results certainly hinted at the effectiveness of the dynamic method.

History

Version	Date	Reason	Revised by
0.1	01/02/2024	Initial Template & Deliverable Structure	Mohammad Sajjadi
0.2	07/02/2024	Initial Draft	Asal Alikhani
0.3	23/03/2024	Issued for review	Mohammad Sajjadi
0.4	28/03/2024	Review	Muhammad Saleem
0.5	30/03/2024	Final Revision	Asal Alikhani & Martin Voigt
1.0	31/03/2024	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@informatik.uni-leipzig.de
elevait GmbH & Co. KG	Martin Voigt	martin.voigt@elevait.de
elevait GmbH & Co. KG	Mohammad Sajjadi	mohammad.sajjadi@elevait.de

Contents

1 Introduction

Partitioning large datasets among multiple data nodes is essential for enhancing the scalability, availability, and overall performance of storage systems. Distributed triple stores commonly employ RDF graph partitioning techniques to optimize query processing. However, existing approaches often overlook the specific properties of RDF graphs, leading to sub-optimal query runtime. Novel approach to RDF graph partitioning that focuses on workload-based analysis, leveraging predicate co-occurrences in querying patterns. State-of-the-art RDF graph partitioning techniques can be divided into various categories [4]. Predicates Co-occurrence-based Partitioning using Extended Markov Clustering (PCM). These methods employ clustering algorithms to group predicates, enabling the creation of partitions that optimize data distribution. This approach aims to minimize inter-communication between partitions by storing RDF triples with commonly queried predicates together. By prioritizing predicates based on their frequency of occurrence in user queries, we anticipate improved query runtime performance. Notably, this technique offers advantages in managing index updates, dynamic data redistribution, and replication.

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. 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.

In previous deliverables we proposed two novel algorithms namely PCG and PCM for static distribution of RDF knowledge graphs. We evaluated these algorithms with public dataset such as Semantic Dog food dataset and some of dbpedia datasets. These algorithms are based on querying history, which changes with time. As such, we needed 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. In this deliverable we proposed a method to do the dynamic data exchange between nodes and do final evaluation based on the one of elevait's dataset. The evaluation results certainly hinted at the usefulness of the proposed method.

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 $G_1, \ldots G_n$ 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

```
@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
```

Figure 1: An example RDF triples

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

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

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

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

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

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

```
SELECT *
{
?O ?P1 ?S.
?S ?P3 ?S3
}

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

Listing 1: Query examples

that is queried together in SPARQL queries should be kept in the same node, thus minimizing the network traffic among data nodes. In this example, we want to partition the 11 triples into 3 partitions namely green, red, and blue partitions.

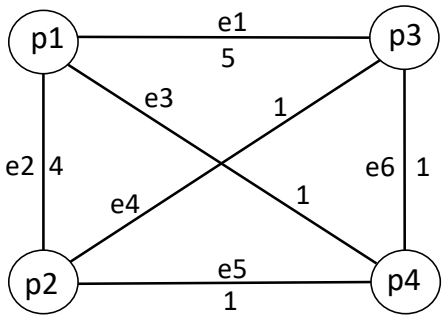
Workload-Based Partitioning: The partitioning techniques in this category make use of the query workload to partition the given RDF dataset. Ideally, the query workload contains real-world queries posted by the users of the RDF dataset which can be collected from the query log of the running system. 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.

3 Our proposed partitioning:

In the following 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. PCM and PCG make 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.

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.

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).

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 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.

3.2 Graph Clustering

Two proposed clustering algorithms (PCM, PCG) 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[1] 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

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

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 $\quad T := (T)^e$ // Expend
6 $\quad T :=$ Inflate(r, Power(e, T)) // Inflate
7 **end**
8 **return** $getClusters(T, n)$ /* get n clusters from matrix */

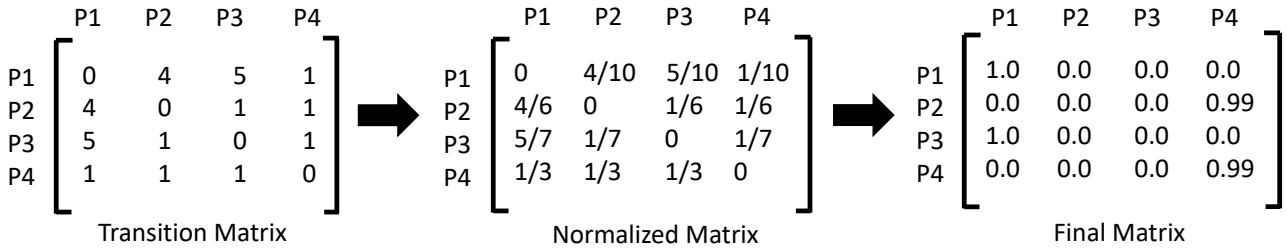


Figure 3: Creation of a matrix during PCM using our weighted graph

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 column by the sum of all the elements in that column. The normalized matrix is shown in Figure 3.

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 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 $T_{1,3}$, 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 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

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 ... c_n} ;                                      // Required clusters
5  i = 1 ;
6  forall e_j ∈ 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 ← {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
```

$e_j \in E$ 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 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.

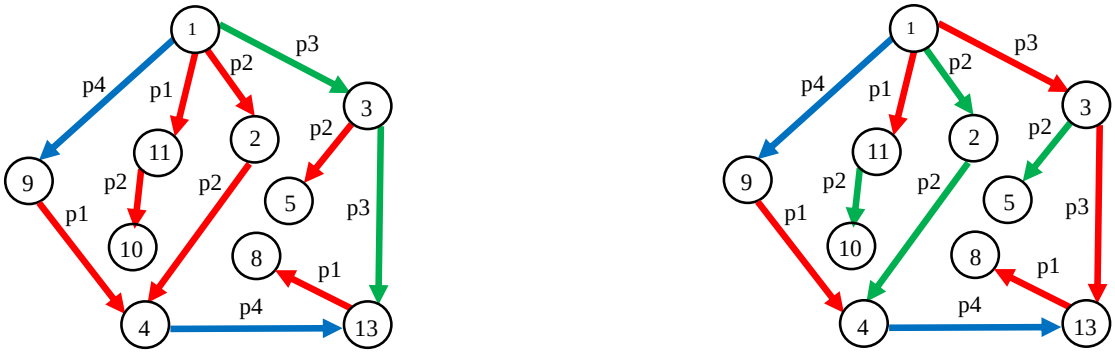
3.3 Assigning Clusters to Partitions

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.

4 Dynamic Data Distribution

Now we explain the dynamic data distribution according to the new workload. We propose to collect a query log of a time frame and make a new data distribution. We follow the following steps in each experiment to perform this task.

- We get a querying workload W1 and perform predicate-clustering using PCM. The corresponding cluster of predicates C1 is then assigned to physical partitions as discussed before.



(a) PCM (b) PCG

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

(a) PCM (b) PCG

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

- The triple store is then used in practice for next time frame and the new workload W2 is collected. We perform again the predicate-clustering C2 using PCM and 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 for consequent time frames. Deciding about the length of time frame depends on the different use cases of the dataset during the time.

5 Evaluation

In this section, we present our evaluation setup followed by evaluation results.

5.1 Evaluation Setup

Dataset: We used *intemp* one of *elevait's* RDF datasets which its data model can be seen in Figure 6. it had *237430 triples* of *Job vacancies* and *CVs* and their relations to load and to be queried, We chose this dataset because it is small but has enough diverse queries. The number of triples and distinct subjects, predicates and objects can be seen in Table 1.

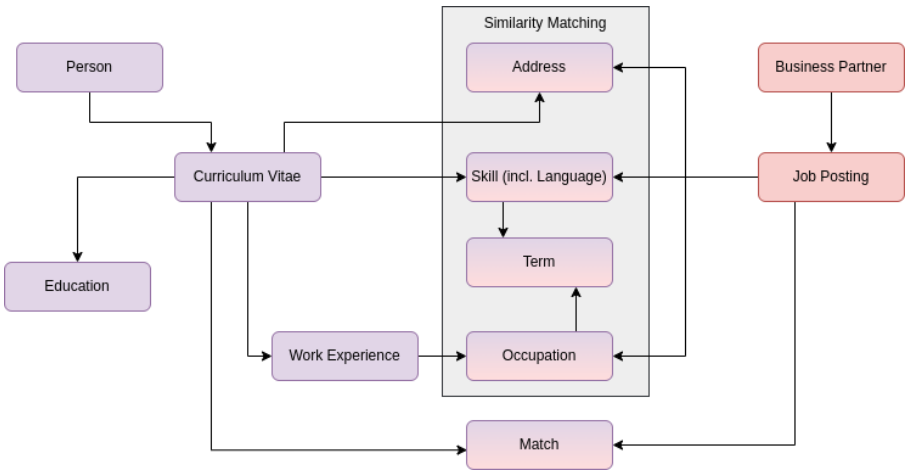


Figure 6: elevait dataset of job vacancies and CVs

Triples	Subjects	Predicates	Objects
238,644	13,313	120	30,708

Table 1: The total number of triples, distinct subjects, predicates, and objects within the used dataset.

Query Workloads (train) and Benchmark (test) Queries: We used a query log of *9910 queries* on *intemp* dataset. This query log includes *six time frame* each focused on similar queries, which was used for dynamic data exchange. Users of this dataset at the beginning of the query log, mostly queried Job Postings and CVs, afterwards they had a time frame to investigate related jobs and CVs and at the end they used more aggregation, statistics and demographic queries. Considering these 3 use cases we divided this query log to *six Query Sets* For conducting six experiments. We considered the first 80 percent of each Query Set as a *query workload* (train queries) and the rest 20 percent as benchmark (test queries). The number of queries of each workload and benchmark set are listed in Table 2. In each experiment workload was used by Partitioning technique, and the benchmark was executed by the evaluation environment.

Evaluation Environments: We used Costfed as the evaluation environment to distribute partition files, execute benchmark queries and evaluate the performance measures. The reason why we chose Costfed is that it is a

Query Set	Total queries	Workload	Benchmark
1	150	120	30
2	1980	1584	396
3	200	160	40
4	3740	2992	748
5	170	136	34
6	3670	2936	734

Table 2: Query Workloads and Benchmark queries

purely federated environment, and is the best system in terms of parallel execution of queries based on [3]. In this environment the given RDF data is distributed among several physically separated machines and a federation engine is used to do federated query processing over multiple SPARQL endpoints.

Number of partitions: Inspired by [3], we created 10 partitions based on each workload of the selected data set and the partitioning technique.

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

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

Partitioning Imbalance: Inspired by [2] we used *Gini* coefficient to measure The imbalance in partitions. For n partitions ($P1$, $P2$, . . . Pn) generated by the partitioning technique, ordered according to the increasing number of triples, Gini coefficient is calculated as:

$$Gini = \frac{n+1}{n} - \frac{2 * \sum_{i=1}^{n}(n+1-i)xi}{n\sum_{i=1}^{n} xi} \tag{1}$$

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, 8 cores and 32GB of RAM*. We conducted our experiments on local copies of *Linux-based virtuoso-opensource-7 (version 7.2.12)* SPARQL endpoints.

5.2 Evaluation Results

Partitioning:

In each experiment the query workload was used by PCM to create 10 partition files. PCM finds unique queries in the workload and distributes triples of dataset based on the co-occurrences of predicates in the queries. For each query workload the number of triples of each partition is shown in Table 7c

Imbalance in partitions: Imbalance in partitions for each experiment is shown in Table 7c and Figure 7b. Imbalance can decrease by using other partitioning techniques based on [1] and [2].

Dynamic data exchange:

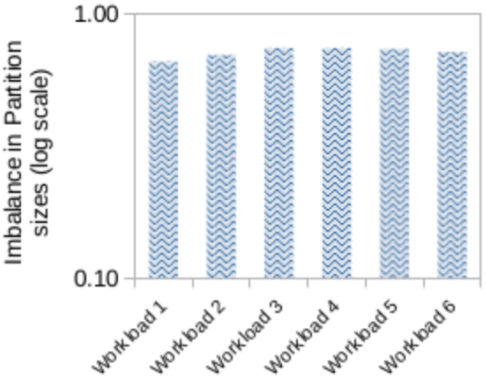
Table 8c shows performance measures of 6 experiments in dynamic method. In this method each workload is used to distribute dataset before executing its related benchmark. The Ex1 represents the initial workload and benchmark pair. Ex2 is the next pair of workload and benchmark and so on. We did data distribution according to workload queries of Ex1 and executed benchmark queries of Ex1 and reported performance measures. We follow the same process for Ex2 until Ex6.

Average Query Runtime: Figure 8a shows a comparison of the average query runtime for the same 6 experiments in dynamic method. Again, the result suggests that query runtime is improved (decreased) with dynamic data distribution while going from Ex1 to Ex6. This shows the effectiveness of the proposed dynamic data distribution.

Query per Second (QpS): Figure 8b shows a comparison of the QpS values for the 6 experiments in dynamic method. We can clearly see the QpS is improved (increased) with dynamic data distribution while going from Ex1 to Experiment Ex6. This shows the effectiveness of the proposed dynamic data distribution.

Query Set	Total queries	Workload	PCM queries	PCM time(ms)
1	150	120	14	823
2	1980	1584	197	1651
3	200	160	18	879
4	3740	2992	373	2775
5	170	136	16	777
6	3670	2936	357	1146

(a) PCM queries and partitioning time



(b) Imbalance in partitions

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
643	3655	22212	90578	5636	19808	2951	5642	106	86199
5742	85140	21971	5688	555	491	1431	4211	5726	106475
1431	86139	3244	3008	3047	2880	1393	6014	2144	128130
2880	7164	1689	1681	3158	1431	1393	7512	82392	128130
4789	8902	88957	179	5924	3047	3030	1431	1393	119778
3185	7295	85029	6923	2997	4656	5742	106	1719	119778

(c) The number of triples in each Partition created by PCM based on 6 workloads

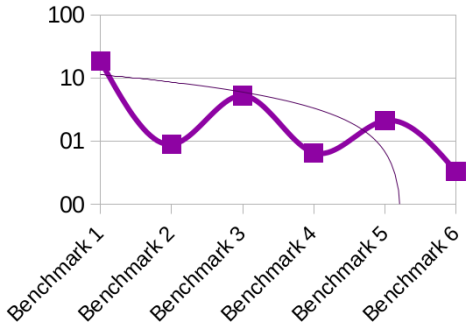
Figure 7: Partitioning by PCM

Static data exchange:

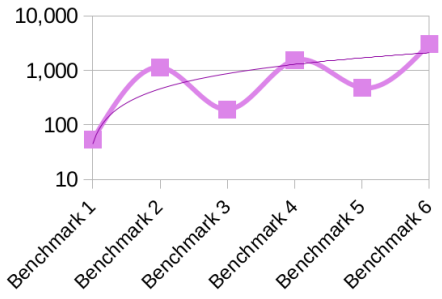
Table 9c shows a comparison of the performance measures for 6 experiments in the static method. In this method the first workload is used to distribute dataset before executing all of the benchmark queries. The Ex1 represents the workload queries for the related benchmark queries in Ex1 but for Ex2 to Ex6 the workload was not changed for different benchmarks. We did data distribution according to workload queries of Ex1 and executed benchmark queries of Ex1 to Ex6 and reported performance measures for each.

Average Query Runtime: Figure 9a shows a comparison of the average query runtime for the same 6 experiments in static method. Again, the result suggests that query runtime have increased (not improved) with static data distribution while going from Ex1 to Ex6. This shows the effectiveness of the proposed dynamic data distribution.

Query per Second (QpS): Figure 9b We can clearly see the QpS (in general) have decreased (not improved) with static data distribution while going from Ex1 to Experiment Ex6. This shows the effectiveness of the proposed dynamic data distribution.



(a) (Avg. runtime of dynamic data exchange



(b) QpS of dynamic data exchange

Experiment	Workload	Benchmark	Queries	Total RT(ms)	Avg. RT(ms)	QpS
Ex1	1	1	30	56	18.7	53.57
Ex2	2	2	396	346	0.88	1,135.84
Ex3	3	3	126	14,510	5.25	190.48
Ex4	4	4	481	220,223	0.65	1,536.38
Ex5	5	5	69	67,157	2.09	478.26
Ex6	6	6	239	13,065	0.33	3,071.13

(c) CostFed results of dynamic data exchange

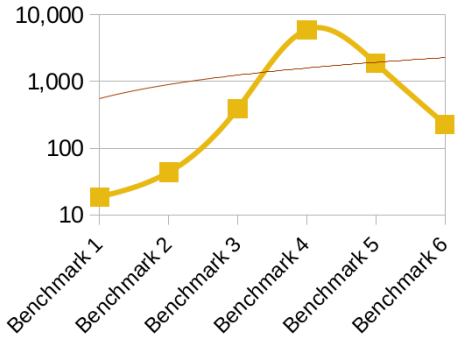
Figure 8: Performance measures of each benchmarks after loading partitions based on the related workload

Dynamic data exchange vs Static data exchange:

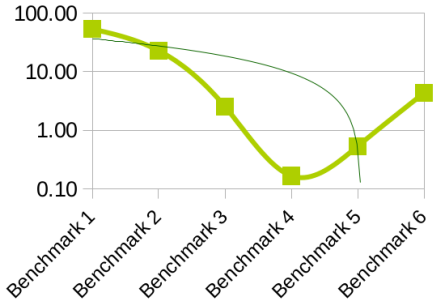
We can clearly see that the performance measures of the various tests have improved (increased) over time with the dynamic data distribution, however, we can also see that the improvement is not linear. For example in Figure 8 the QpS and Average query runtime of Ex2 is better than Ex3 (and Ex4 is better than Ex5). The possible reason is that dividing of query log for different experiments was done based on use cases at different periods of time. In each period of time, the most frequently querying concepts can be completely different and independent from other periods. Therefore we compared these metrics of each experiment in both methods.

Average Query Runtime: Figure 10a shows a comparison of the average query runtime for each experiment in both methods. Again, the result suggests that query runtime is better(less) in dynamic data distribution than static method in each experiment. This shows the effectiveness of the proposed dynamic data distribution.

Query per Second (QpS): Figure 10b shows a comparison of the QpS values for each experiment in both methods. We can clearly see the QpS is better(more) in dynamic data distribution than static method in each experiment. This shows the effectiveness of the proposed dynamic data distribution.



(a) Avg. runtime of Static workload

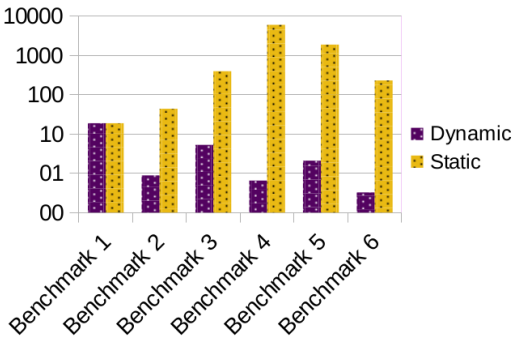


(b) QpS of Static workload

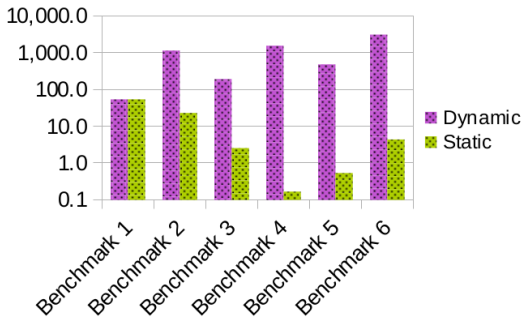
Experiment	Workload	Benchmark	Queries	Total RT(ms)	Avg. RT(ms)	QpS
Ex1	1	1	30	56	19	53.57
Ex2	1	2	396	8,070	44	22.92
Ex3	1	3	40	14,510	392	2.55
Ex4	1	4	748	220,223	5,952	0.17
Ex5	1	5	34	67,157	1,865	0.54
Ex6	1	6	734	13,065	229	4.36

(c) CostFed results of static data exchange

Figure 9: Performance measures of each benchmark after loading partitions based on the first workload



(a) Avg. runtime of dynamic vs. Avg runtime of Static data exchange



(b) QpS of dynamic vs. QpS of Static data exchange

Figure 10: Comparing performance measures of dynamic vs. static data exchange for each benchmark

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?focusNode ?subject ?predicate ?object WHERE {
    {SELECT DISTINCT ?node WHERE {
            {SELECT DISTINCT ?node WHERE {
                    { ?node a <http://ai4bd.com/resource/edm/Education
                      > }
                }}} LIMIT 24 OFFSET 0
    }
    BIND ( ?node AS ?focusNode )
    {BIND ( ?node AS ?subject )
        ?subject ?predicate ?object .}
    UNION
    {BIND (rdfs:label as ?predicate)
        ?node ?p ?subject .
        ?subject ?predicate ?object .}
    UNION
    {BIND (rdfs:label as ?predicate)
        ?subject ?p ?node .
        ?subject ?predicate ?object .}
}
```

Figure 11: a sample of the most common query type used in elevait

6 Conclusion and Future Work

In this deliverable, we evaluated dynamic data exchange using a real world dataset. We made use of 6 querying workloads on a dataset of Job Postings and CVs from elevait as a real use case 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 significant differences in the performance measures achieved by the proposed dynamic data partitioning mechanism based on experienced workload, as it was also examined and proved within the previous tasks.

In the future, more datasets and different partitioning techniques such as PCG which creates more balanced partitions can be used to evaluate the proposed dynamic data exchange. We suggest to focus on the effects of partitioning pertaining to use cases which involve validating and reasoning tasks by SHACL shapes or data updates etc. In elevait's dataset, SHACL shapes are used to define properties of a class or relationships between classes, therefore there are lots of queries similar to the sample in Listing Figure 11 which currently is not supported by proposed algorithms.

References

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

[2] Adnan Akhter, Muhammad Saleem, Alexander Bigerl, and Axel-Cyrille Ngonga Ngomo. Efficient rdf knowledge graph partitioning using querying workload. 11 2021.

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

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