An Empirical Evaluation of RDF Graph Partitioning Techniques

Adnan Akhter[1], Axel-Cyrille Ngonga Ngomo[1,2], and Muhammad Saleem[1]

[1] AKSW, Germany, {lastname}@informatik.uni-leipzig.de
[2] University of Paderborn, Germany axel.ngonga@upb.de

Abstract. With the significant growth of RDF data sources in both numbers and volume comes the need to improve the scalability of RDF storage and querying solutions. Current implementations employ various RDF graph partitioning techniques. However, choosing the most suitable partitioning for a given RDF graph and application is not a trivial task. To the best of our knowledge, no detailed empirical evaluation exists to evaluate the performance of these techniques. In this work, we present an empirical evaluation of RDF graph partitioning techniques applied to real-world RDF data sets and benchmark queries. We evaluate the selected RDF graph partitioning techniques in terms of their partitioning time, partitioning imbalance (in sizes), and query run time performances achieved, based on real-world data sets and queries selected using the *FEASIBLE* benchmark generation framework.

1 Introduction

Data partitioning is the process of logically and/or physically dividing datasets into subsets to facilitate better maintenance and access. Data partitioning is often used for load balancing, improving system availability and query processing times in data management systems. Over recent years, several Big datasets such as Linked TCGA[3] (around 20 billion triples) and UniProt[4](over 10 billion triples) have been added to the Web of Data. The need to store and query such datasets efficiently has motivated a considerable amount of work on designing clustered triplestores [4,6,8,9,10,11,16,17,18,21,22,27], i.e., solutions where data is partitioned among multiple data nodes. It is noteworthy that current triplestores employ various graph partitioning techniques [22]. It is also well known that the query execution performance of data storage solutions can be greatly affected by the partitioning technique used in the data store [12]. However, no detailed evaluation of the efficiency of the different RDF graph partitioning techniques in terms of scalability, partitioning imbalance, and query run time performances has been undertaken.

We address this research gap by presenting a detailed empirical evaluation of different RDF graph partitioning techniques. We compare them according

[3] TCGA: http://tcga.deri.ie/
[4] UniProt: http://www.uniprot.org/statistics/

to their suitability for balanced load generation, partitioning time, and query runtime performance. Our contributions are as follows:

1. We compared seven RDF graph partitioning techniques in two different evaluation setups.
2. We evaluate the selected RDF partitioning techniques using different performance measures such as partitioning time, variation in the sizes of the generated partitions, number of sources selected in a purely federated environment, and query runtime performance.
3. We perform an evaluation based on two real-world datasets (i.e., DBpedia and Semantic Web Dog Food), and real queries (selected from users' queries log) using the SPARQL benchmark generation framework from queries log FEASIBLE [19].

All of the data, source code, and results presented in this evaluation are available at `https://github.com/dice-group/rdf-partitioning`.

2 RDF Graph Partitioning

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.

In this section, we explain commonly used [14,15,22,20] graph partitioning techniques by using a sample RDF graph shown in Figure 1.

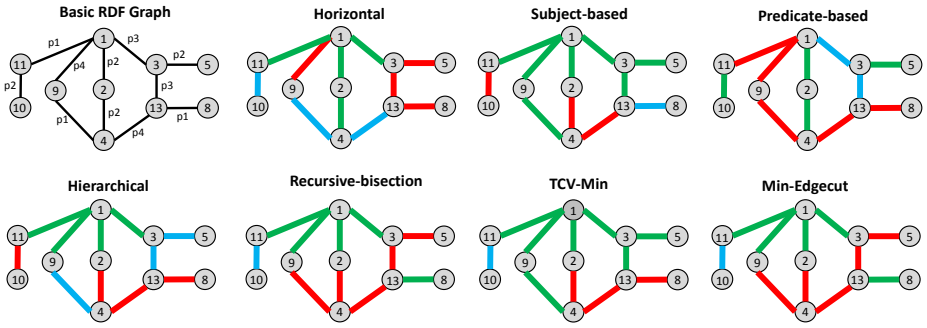
Horizontal Partitioning: This partitioning technique is adopted from [20]. Let T be the set of all RDF triples in a dataset and n be the required number of partitions. The technique assigns the first $|T|/n$ triples in partition 1, the next $|T|/n$ triples in partition 2 and so on. In the example given in Figure 1, the triples 1-4 will be assigned to the first partition (green), triples 5-8 will be assigned to the second partition (red), and triples 9-11 will be assigned to the third partition (blue).

Subject-Based Partitioning: This technique assigns triples to partitions according to a hash value computed on their subjects modulo the total number of required partitions (i.e., hash(subject) modulus total number of partitions)[14]. Thus, all the triples with the same subject are assigned to one partition. However, due to modulo operation this technique may result in high partitioning imbalance. In our motivating example given in Figure 1, triples (3,10,11) are matched to the red partition, only triple 7 is matched to the blue partition, and the remaining are matched to the blue partition. Thus, a clear partitioning imbalance (3:1:7 triples) results.

Predicate-Based Partitioning: Similar to Subject-Based, this technique assigns triples to partitions according to a hash value computed on their predicates

```
@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.

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

modulo the number of required partitions. Thus, all triples with the same predicate are assigned to the same partition. In our motivating example given in Figure 1, all the triples with predicate `p1` or `p4` are assigned to the red partition, triples with predicate `p2` are assigned to the green partition, and all triples with predicate `p3` are assigned to the blue partition.

Hierarchical Partitioning: This partitioning is inspired by the assumption that IRIs have path hierarchy and IRIs with a common hierarchy prefix are often queried together [14]. This partitioning is based on extracting path hierarchy from the IRIs and assigning 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 hash-based subject or predicate 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.

Recursive-Bisection Partitioning: Recursive bisection is a multilevel graph bisection algorithm aiming to solve the k-way graph partitioning problem as described in [15]. 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 (1,2,4,7,8) are assigned to the green partition, triples (3,5,6,9,10) are assigned to the red partition, and only triple 11 is assigned to the 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 (1,2,4,5,6,8,9) are assigned to the green partition, triples (3,7,10) are assigned to the red partition, and only triple 11 is assigned to the blue partition.

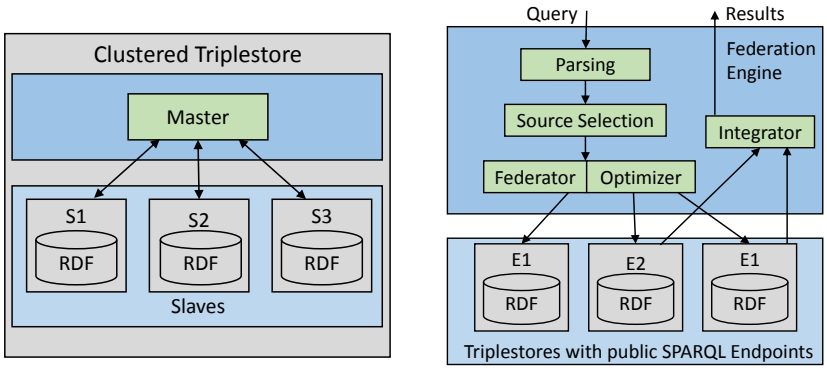
Min-Edgecut Partitioning: The Min-Edgecut [15] also aims to solve the k-way graph partitioning problem. However, unlike TCV-Min, the objective is to partition the vertices by minimizing the number of edges connected to them. In our motivating example given in Figure 1, triples (1,2,4,7,8) are assigned to the green partition, triples (3,5,6,9,10) are assigned to the red partition, and only triple 11 is assigned to the blue partition.

3 Evaluation

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

3.1 Evaluation Setup

Partitioning Environments: We used two distinct evaluation environments to compare the selected RDF graph partitioning techniques. **(1) Clustered RDF Storage Environment** In this environment, the given RDF data is distributed among different data nodes within the same machine as part of a single RDF storage solution. Figure 2a shows the very generic master-slave



(a) Clustered Architecture

(b) Physically Federated Architecture

Fig. 2: Evaluation Environments

architecture used in our clustered environment. The master assigns the tasks and the slaves perform RDF storage and query processing tasks. There are many RDF storage solutions [4,6,8,9,10,11,16,17,18,21,22,27] that employ this architecture. We chose *Koral* [14] in our evaluation. The reason for choosing this platform was because it allows the data partitioning strategy to be controlled, it is a state-of-the art distributed RDF store, and it is well-integrated with the famous RDF partitioning system *METIS* [15]. **(2) Purely Federated Environment** In this environment, the given RDF data is distributed among several physically separated machines and a federation engine is used to do the query processing task. We chose the well-known SPARQL endpoint federation setup [20] in which data is distributed among several SPARQL endpoints and a SPARQL federation engine is used to do federated query processing over multiple endpoints. Figure 2b shows the two main components (i.e., the federation engine and the SPARQL endpoints) of this architecture. The general steps involved to process a SPARQL query in this evaluation environment are as follows: Given a SPARQL query, the first step is to parse the query and get the individual triple patterns. The next step is source selection, for which the goal is to identify the set of relevant data sources (endpoints in our case) for the query. Using the source selection information, the federator divides the original query into multiple sub-queries. An optimized sub-query execution plan is generated by the optimizer and the sub-queries are forwarded to the corresponding data sources. The results of the sub-queries are then integrated by the integrator. The integrated results are finally returned to the agent that issued the query. Many SPARQL endpoint federation engines [23,3,25,1,7] abide by this architecture. We chose *FedX* [23] and *SemaGrow* [3] in our evaluation. The reason for choosing these two federation engines is their use of different query execution plans. FedX is an index-free heuristic-based SPARQL endpoint federation engine, while SemaGrow is an index-assisted cost-based federation engine. Note that the query execution plan greatly affects the query runtime performances, therefore we

wanted to choose federation engines that employ different query planners (FedX is left-deep-trees-based, and SemaGrow is a busy-tree-based solution).

Datasets: We wanted to benchmark the selected partitioning techniques based on real-world RDF datasets and real-world SPARQL queries submitted by users to the SPARQL endpoints of underlying datasets. To achieve this goal, we used two real-word datasets: *DBpedia 3.5.1* and the *Semantic Web Dog Food (SWDF)* for partitioning. The reason for choosing these two datasets is that they are used by the *FEASIBLE* [19] SPARQL benchmark generation framework to generate customized SPARQL benchmarks from the queries log of the underlying datasets. These two datasets vary greatly in their high-level statistics: the DBpedia 3.5.1 contains 232,536,510 triples, 18,425,128 distinct subjects, 39,672 distinct predicates, and 65,184,193 distinct objects while SWDF contains 304,583 triples, 36,879 distinct subjects, 185 distinct predicates, and 95,501 distinct objects.

Queries: We generated the following benchmarks for evaluation using FEASIBLE: **(1) SWDF BGP-only** benchmark contains a total of 300 BGP-only SPARQL queries from the queries log of the SWDF data set. These queries only contain single BGP; the other SPARQL features such as `OPTIONAL`, `ORDER BY`, `DISTINCT`, `UNION`, `FILTER`, `REGEX`, aggregate functions, `SERVICE`, property paths etc. are not used, **(2) SWDF fully-featured** contains a total of 300 queries which are not only single BGPs and may include more features (e.g., the above mentioned) of the SPARQL queries, **(3) DBpedia BGP-only** contains 300 BGP-only, and **(4) DBpedia fully-featured** contains 300 fully-featured SPARQL queries selected from the queries log of DBpedia 3.5.1. Thus, in our evaluation we used a total of 1200 SPARQL queries selected from two different data sets. Note that we only used BGP-only benchmarks with Koral since it does not support many of the SPARQL features used in the fully-featured SPARQL benchmarks.

Number of partitions: Inspired by [20], we created 10 partitions for each of the selected data sets and the partitioning technique. In Koral, we ran 10 slaves each containing one partition. In the *purely federated environment*, we used 10 Linux-based Virtuoso 7.1 SPARQL endpoints, each containing one partition.

Performance measures: We used six performance measures to benchmark the selected partitioning techniques – partitions generation time, overall benchmark execution time, average query execution time, number of timeout queries for each benchmark, the ranking score of the partitioning techniques, total number of sources selected for the complete benchmark execution in a purely federated environment, and the partitioning imbalance among the generated partitions. Three minutes was selected as the timeout time for query execution [19]. In addition, we also measured the Spearman's rank correlation coefficients to ascertain the correlation between the sources selected and the query run time in a purely federated environment. The rank score of the partitioning technique is defined as follows:

Definition 2 (Rank Score). Let t be the total number of partitioning techniques and b be the total number of benchmark executions used in the evaluation. Let $1 \leq r \leq t$ denote the rank number and $O_p(r)$ denote the occurrences 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 seven partitioning techniques (i.e., t = 7) and 10 benchmarks executions (b = 10, 4 benchmarks by FedX, 4 benchmarks by SemaGrow, and 2 benchmarks by Koral).

The partitioning imbalance in the sizes of the generated partitions is defined as follows:

Definition 3 (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 number of triples. The imbalance in partitions is defined as Gini coefficient:

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

Hardware and software configuration: All experiments were run on an 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 METIS 5.1.0.dfsg-2 [5] to create TCV-Min, Min-Edgecut and Recursive-Bisection. We used default configurations for FedX, SemaGrow and Koral (except the slaves were changed from 2 to 10 in Koral).

3.2 Evaluation Results

Partition Generation Time: Figure 3 shows a comparison of the time taken by each technique to generate the required 10 partitions, both for DBpedia 3.5.1 and SWDF datasets. As an overall evaluation, the Horizontal partitioning method requires the smallest time followed by the Subject-Based, Predicate-Based, Hierarchical, TCV-Min, Recursive-Bisection, and Min-Edgecut, respectively. The reason for the Horizontal partitioning taking the least time lies in this simplicity: the technique creates the range of triples and assigns them to the desired partitions in the first come first server basis. Both Predicate-Based and Subject-Based partitioning techniques take almost the same time because both techniques simply traverse each triple in the dataset and apply hash functions on the subject or predicate

[5] http://glaros.dtc.umn.edu/gkhome/metis/metis/download

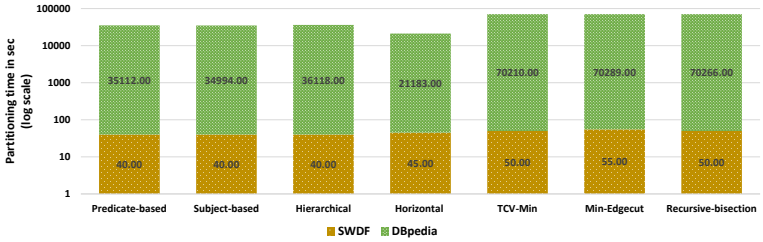
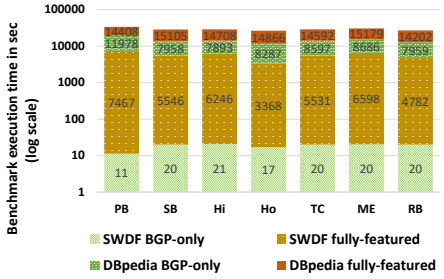


Fig. 3: Time taken for the creation of 10 partitions

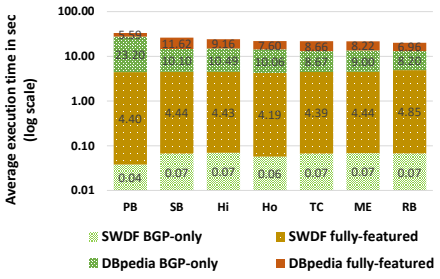
of the triple. Thus, they have the same computational complexity. Hierarchical partitioning takes more time compared with the Subject and Predicate-Based hash partitioning techniques due to the extra time required to compute path hierarchies before hash function is applied. The k-way implementations of graph partitioning, i.e., TCV-Min, Min-Edgecut and Recursive-Bisection consumed even more time (almost double) compared to the other techniques. This is because of their higher complexity in terms of the time required to perform the coarsening, partitioning, and uncoarsening phases.

Query runtime performances: One of the most important results is the query runtime performances achieved by using each of the selected partitioning techniques. We used the total benchmark (300 queries) execution time (including timeout queries) and the average query execution time (excluding timeout queries) to encapsulate the runtime performances of the partitioning techniques. To measure the former performance metric, we executed the complete 300 queries from each benchmark over the data partitions created by the selected partitioning techniques and calculated the total time taken to execute the complete benchmark queries. For each timeout query, we add 180 seconds to the total benchmark execution time. For the latter performance metric, we only considered those queries which were successfully executed within the timeout limit and present the average query execution time for each of the selected partitioning technique. Figure 4 presents the query runtime performances achieved by each of the selected techniques pertaining to the two aforementioned query execution metrics.

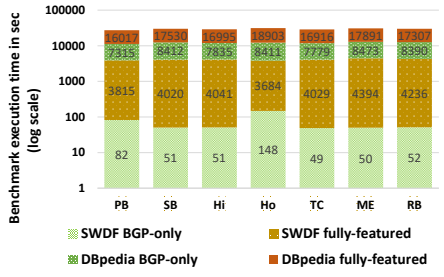
Figure 4a shows the total execution time of the complete benchmarks for the selected partitioning techniques based on FedX federation engine. Including all the benchmark execution results (over 4 benchmarks), Horizontal partitioning consumed the least time (26538.7 seconds), followed by Recursive-Bisection (26962.6 seconds), Subject-Based (28629.3 seconds), TCV-Min (28739.9 seconds), Hierarchical (28867.5 seconds), Min-Edgecut (30482.8 seconds) and Predicate-Based (33864.2 seconds), respectively. The total benchmark execution time of the individual benchmarks (i.e., two from SWDF and two from DBpedia3.51) can be seen from the bar stacked graphs directly. Figure 4b shows the average query execution times of the selected partitioning techniques based on four benchmarks on FedX. The overall (over 4 benchmarks) average query execution results show



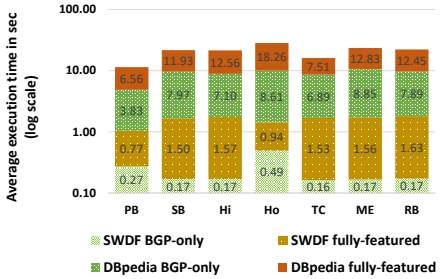
(a) FedX benchmarks execution time



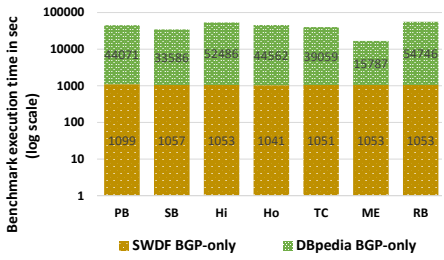
(b) FedX average query runtimes



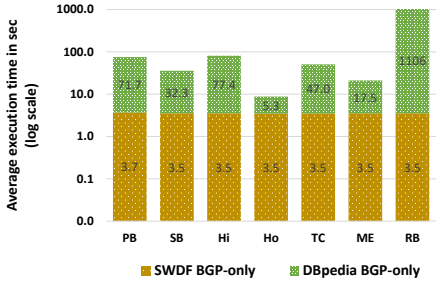
(c) SemaGrow benchmarks execution time



(d) SemaGrow average query runtimes



(e) Koral benchmarks execution time



(f) Koral average query runtimes

Fig. 4: Benchmarks (300 queries each) total execution time including timeouts and average query runtimes excluding timeouts. (PB = Predicate-Based, SB= Subject-Based, Hi= Hierarchical, Ho = Horizontal, TC = TCV-Min, ME Min-Edgecut, RB = Recursive Bisection)

Recursive-Bisection has the smallest average query runtime (5.020557271 seconds), followed by Min-Edgecut (5.4330126 seconds), TCV-Min (5.4456308 seconds), Horizontal (5.4801338 seconds), Hierarchical (6.0390115 seconds), Subject-Based (6.5591146 seconds) and Predicate-Based (8.3071525 seconds), respectively.

Figure 4c shows the total execution time of the complete benchmarks for the selected partitioning techniques based on SemaGrow federation engine. From all (over 4 benchmarks) benchmark execution results, Predicate-Based partitioning consumed the least time (27227.9 seconds) followed by TCV-Min (28772.8

seconds), Hierarchical (28921.6 seconds), Recursive-Bisection (29983.9 seconds), Subject-Based (30012.5 seconds), Min-Edgecut (30807.5 seconds) and Horizontal (31145.9 seconds), respectively. Figure 4d shows the average query execution times of the selected partitioning techniques based on four benchmarks on SemaGrow. Of all (over 4 benchmarks) average query execution results, Predicate-Based has the smallest average query runtime (2.857210203 seconds) followed by Subject-Based (5.393390726 seconds), Hierarchical (5.349322361 seconds), Horizontal (7.077052279 seconds), TCV-Min (4.024567032 seconds), Min-Edgecut (5.850084384 seconds) and Recursive-Bisection (5.535637211 seconds) respectively.

Since both FedX and SemaGrow federation engines represent the *purely federated environment*, we now present the combined results of the two federation engines. Including all (over FedX+SemaGrow and over 4 benchmarks) benchmark execution results, Recursive-Bisection partitioning consumed the smallest time (28473.233 seconds), followed by TCV-Min (28756.337 seconds), Horizontal (28842.264 seconds), Hierarchical (28894.5275 seconds), Subject-Based (29320.9305 seconds), Predicate-Based (30546.0905 seconds) and Min-Edgecut (30645.1825 seconds), respectively. Considering all (over FedX+SemaGrow and over 4 benchmarks) average query runtime results, TCV-Min has the smallest average query execution time (5.278097241 seconds), followed by Recursive-Bisection (5.278097241 seconds), Predicate-Based (5.582181367 seconds), Min-Edgecut (5.641548479 seconds), Hierarchical (5.694166918 seconds), Subject-Based (5.976252639 seconds) and Horizontal (6.27859305 seconds), respectively.

Figure 4e shows the total execution time of the complete benchmarks for the selected partitioning techniques based on Koral. Including all (over two benchmarks) benchmark execution results, the Min-Edgecut consumed the least time (16839 seconds), followed by Subject-Based (34643 seconds), TCV-Min (40110 seconds), Predicate-Based (45170 seconds), Horizontal (45602 seconds), Hierarchical (53539 seconds) and Recursive-Bisection (55798 seconds), respectively. Figure 4f shows the average query execution times of the selected partitioning techniques based on four benchmarks on Koral. From all (over the 4 benchmarks) average query execution results, Horizontal partitioning has the smallest average query runtime (4.393116824 seconds), followed by Min-Edgecut (10.48653731 seconds), Subject-Based (17.91570378 seconds), TCV-Min (25.26057554 seconds), Predicate-Based (37.66883389 seconds), Hierarchical (40.43121192 seconds) and Recursive-Bisection (554.618705 seconds), respectively.

The complete benchmark execution results are best summarized in terms of total timeout queries, overall rankings, and the rank scores of the partitioning techniques and are presented in the subsequent sections.

Number of timeout queries: Table 1 shows the total number of timeout queries for each of the 4 benchmarks and for each of the partitioning techniques using FedX, SemaGrow and Koral. Overall (i.e., over FedX + SemaGrow + Koral), Min-Edgecut has the smallest timeouts (344 queries), followed by the Subject-Based (422 queries), TCV-Min (455 queries), Predicate-Based (485 queries),

Table 1: Timeout queries using FedX, SemaGrow and Koral

	FedX				SemaGrow				Koral	
	SWDF		DBpedia		SWDF		DBpedia		SWDF	DBpedia
Partitioning	BGP	FF	BGP	FF	BGP	FF	BGP	FF	BGP	BGP
Predicate-Based	0	35	32	73	0	20	35	81	0	209
Subject-Based	0	24	29	69	0	20	35	83	0	162
Hierarchical	0	28	28	70	0	20	33	79	0	286
Horizontal	0	12	31	73	0	19	34	83	0	246
TCV-Min	0	24	35	70	0	20	33	85	0	188
Min-Edgecut	0	30	35	74	0	22	34	84	0	65
Recursive-Bisection	0	19	32	70	0	21	35	81	0	298

Horizontal (498 queries), Hierarchical (544 queries), and Recursive-Bisection (556 queries), respectively.

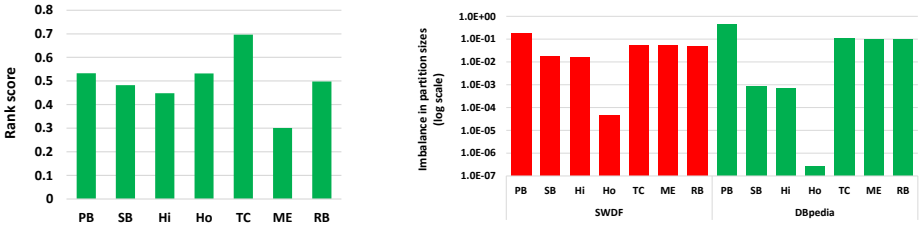
Overall Ranking of Partitioning Techniques: Table 2 shows the results of the overall rank-wise ranking of the selected partitioning techniques based on the total benchmark execution time from a total of 4 benchmarks. Based on FedX, Predicate-Based partitioning ranked 1^{st} and 2^{nd} once each,and 7^{th} twice, suggesting this technique either produces the best or worst query runtime performances among the selected partitioning techniques. Subject-Based partitioning ranked mostly in the middle (once 2^{nd}, twice 4^{th} and once 6^{th}), suggesting this technique produces average runtime performances among the selected partitioning techniques. Hierarchical partitioning ranked in the top, middle, and lower positions, suggesting unpredictable runtime performances. Horizontal partitioning has given the best results twice and on the other two occasions it gave the average results. TCV-Min was very consistent by producing the third best result on three times. Min-Edgecut runtime performance is usually on the lower side. Recursive-Bisection gave three results at the best side of the scale, however it ranked 5^{th} once.

Based on SemaGrow, Predicate-Based partitioning mostly results to good query runtime performances. The query runtime performances of the Subject-Based and Hierarchical partitioning techniques is on the average or lower sides. Horizontal has given best results once and the rest three times were on the lower ranked side. TCV-Min performance is mostly on the high ranked side. Again, Min-Edgecut runtime performance is usually on the lower side. Recursive-Bisection, however, has stayed on the lower side.

Based on Koral, Predicate-Based partitioning gave below average query runtime performances. Subject-Based ranked 2^{nd} and 6^{th} one time each. Hierarchical ranked on the lower side. Horizontal ranked 1^{st} and 5^{th} one time each. TCV-Min has produced good results by ranking 2^{nd} and 3^{rd} one time each. Similar to TCV-Min, Min-Edgecut also produced better query runtime performances.

Table 2: Overall rank-wise ranking of partitioning techniques based on two benchmarks from SWDF and DBpedia each. (PB = Predicate-Based, SB= Subject-Based, Hi= Hierarchical, Ho = Horizontal, TC = TCV-Min, ME Min-Edgecut, RB = Recursive Bisection)

	FedX							SemaGrow							Koral						
PT	1st	2nd	3rd	4th	5th	6th	7th	1st	2nd	3rd	4th	5th	6th	7th	1st	2nd	3rd	4th	5th	6th	7th
PB	1	1	0	0	0	0	2	2	1	0	0	0	1	0	0	0	0	1	0	0	1
SB	0	1	0	2	0	1	0	0	0	2	0	1	1	0	0	1	0	0	0	1	0
Hi	1	0	0	1	1	0	1	0	0	2	1	1	0	0	0	0	0	0	1	1	0
Ho	1	1	0	1	1	0	0	1	0	0	0	1	0	2	1	0	0	0	1	0	0
TC	0	0	3	0	1	0	0	1	2	0	1	0	0	0	0	1	1	0	0	0	0
Mi	0	0	0	0	0	3	1	0	1	0	0	0	1	2	1	0	0	1	0	0	0
Re	1	1	1	0	1	0	0	0	0	0	2	1	1	0	0	0	1	0	0	0	1



(a) Rank scores

(b) Partitioning imbalance

Fig. 5: 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)

Recursive-Bisection ranked 3^{rd} and 7^{th} once each. Please note that Koral ranking is based on a total of 2 (BGP-only) benchmarks.

Rank scores: From Table 2, it is hard to decide which partitioning technique is generally ranked better. We used Table 2 to compute the rank scores (ref., Definition 2) pertaining to each of the partitioning techniques and presented in Figure 5a. TCV-Min results in the highest rank score, followed by Property-based, Horizontal, Recursive-Bisection, Subject-Based, Hierarchical, and Min-Edgecut respectively.

Partitioning imbalance: Figure 5b shows the partitioning imbalance (defined in Definition 3) values of the partitions generated by the selected partitioning techniques. As expected, the Horizontal portioning results the smallest partitioning imbalance, followed by Hierarchical, Subject-Based, Min-Edgecut, Recursive-Bisection, TCV-Min and Predicate-Based, respectively.

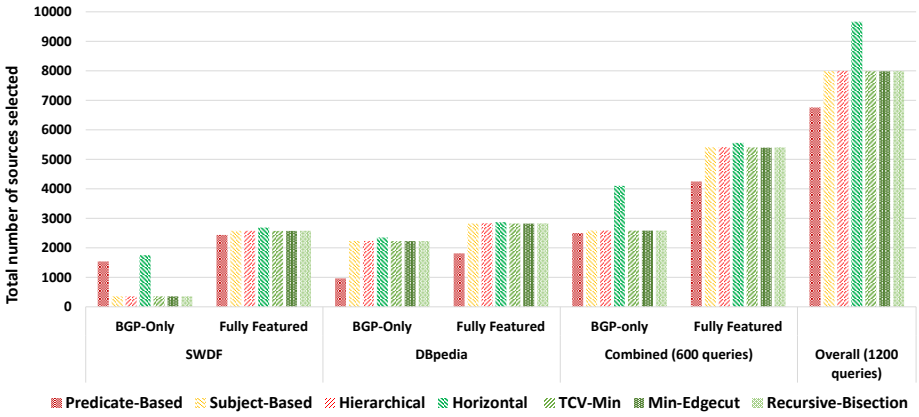


Fig. 6: Total distinct sources selected

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 [20]. Figure 6 shows the total distinct sources selected by FedX and SemaGrow. Note that the source selection algorithm of both FedX and SemaGrow select exactly the same sources. Generally (over 4 benchmarks) source selection evaluation, Predicate-Based selects the smallest number of sources, followed by Min-Edgecut, TCV-Min, Recursive-Bisection, Subject-Based, Hierarchical and Horizontal, respectively.

Spearman's rank correlation coefficients: Finally, we want to show how the number of sources selected affects the query execution time. To this end, we computed the Spearman's rank correlation between the number of sources selected and the query execution time. Table 3 shows Spearman's rank correlation coefficients values for the four evaluation benchmarks and the selected partitioning techniques. The results suggest that the number of sources selected, in general, have a positive correlation with the query execution times, i.e. the smaller the sources selected the smaller the execution time and vice versa.

4 Related Work

A plethora of clustered triplestores have been designed in previous works [4,6,8,9,10,11,16,17,18,21,22,27] and mentioned across the paper. Here, we only target the RDF graph partitioning literature. Koral [14] is a distributed RDF triplestore which allows the integration of different RDF graph partitioning techniques. An analysis of three partitioning techniques, i.e., Subject-Based, Hierarchical and Min-Edgecut is presented in [5] based on synthetic data and queries. A brief survey of RDF graph partitioning is provided in [24]. [13] suggests that hash-based partitioning is more scaleable as hash values can be computed

	Benchmark	Pred	Sub	Hierar	Horiz	TCV	Mincut	Recur	Average
FedX	DBpedia BGP-only	0.22	0.30	0.30	0.28	0.26	0.27	0.29	0.27
	DBpedia Fully-featured	0.14	0.11	0.11	0.16	0.17	0.12	0.17	0.14
	SWDF BGP-only	−0.10	0.57	0.57	0.10	0.57	0.57	0.57	0.41
	SWDF Fully-featured	0.22	0.11	0.13	0.09	0.11	0.13	0.10	0.12
S-Grow	DBpedia BGP-only	−0.02	0.11	0.10	0.06	0.09	0.30	0.29	0.13
	DBpedia Fully-featured	0.14	0.18	0.23	0.02	0.24	0.26	0.16	0.18
	SWDF BGP-only	0.23	0.64	0.64	0.65	0.66	0.64	0.64	0.59
	SWDF Fully-featured	0.07	−0.02	−0.02	−0.07	−0.02	−0.06	−0.01	−0.02
	Average	0.11	0.25	0.26	0.16	0.26	0.28	0.28	0.23

Table 3: Spearman's rank correlation coefficients between number of sources selected and query runtimes. Pred: Predicate-Based, Sub: Subject-Based, Hierar: Hierarchical, Horiz: Horizontal, TCV: TCV-Min, Mincut: Min-Edgecut, Recur: Recursive-Bisection, S-Grow: SemaGrow.
Correlations and colors: −0.00… − 0.19 very weak (●-), 0.00…0.19 very weak (●+), 0.20…0.39 weak (●+), 0.40…0.59 moderate (●+), 0.60…0.79 strong (●+).

in parallel. A signature tree-based triple indexing scheme is proposed in [26] to efficiently store the partitions of the RDF graph. To the best of our knowledge, no detailed empirical evaluation exists to position the different RDF graph partitioning techniques based on real data and real queries in two different evaluation environments.

5 Conclusion and Future Work

We presented an empirical evaluation of seven RDF partitioning techniques. Our overall results of query runtime suggest that TCV-Min leads to smallest query runtimes followed by Property-Based,Horizontal, Recursive-Bisection, Subject-Based, Hierarchical, and Min-Edgecut, respectively. Our T-test[6] analysis shows significant differences in the runtime performances achieved by different partitioning techniques. In addition, the number of sources selected has a direct relation with query runtimes. Thus, partitioning techniques which minimize the total number of sources selected generally lead to better runtime performances. In future, we will add more querying engines into the clustered evaluation environment. We will test the scalability of the partitioning techniques using different sizes of the same datasets and use some more Big RDF datasets. We will also focus on the effects of partitioning pertaining to a given use-case, such as when involving reasoning tasks or data updates etc.

Acknowledgements

This work was supported by the H2020 project HOBBIT (no. 688227)

[6] Please see T-Test tab of the excel sheet goo.gl/fxa4cJ

References

1. M. Acosta et al. ANAPSID: An adaptive query processing engine for sparql endpoints. In *ISWC*, 2011.
2. Buluç et al. Recent advances in graph partitioning. In *AE*. 2016.
3. A. Charalambidis et al. SemaGrow: Optimizing federated sparql queries. In *SEMANTICS*, 2015.
4. O. Erling and I. Mikhailov. Towards web scale rdf. *Proc. SSWS*, 2008.
5. D. et. al. Impact analysis of data placement strategies on query efforts in distributed rdf stores. *JWS*, 2018.
6. L. Galárraga et al. Partout: a distributed engine for efficient rdf processing. In *WWW*, 2014.
7. O. Görlitz and S. Staab. SPLENDID: Sparql endpoint federation exploiting void descriptions. In *COLD*, 2011.
8. S. Gurajada et al. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *SIGMOD*, 2014.
9. M. Hammoud et al. Dream: distributed rdf engine with adaptive query planner and minimal communication. *VLDB*, 2015.
10. S. Harris et al. 4store: The design and implementation of a clustered rdf store. In *SSWS*, 2009.
11. A. Harth et al. Yars2: A federated repository for querying graph structured data from the web. In *Semantic Web*. 2007.
12. H. Herodotou et al. Query optimization techniques for partitioned tables. In *SIGMOD*, 2011.
13. J. Huang et al. Scalable sparql querying of large rdf graphs. *VLDB*, 2011.
14. D. Janke et al. Koral: A glass box profiling system for individual components of distributed rdf stores. In *BLINK-ISWC*, 2017.
15. G. Karypis et al. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM JSC*, 1998.
16. A. Khandelwal et al. Zipg: A memory-efficient graph store for interactive queries. In *ACM ICMD*, 2017.
17. T. Neumann et al. The rdf-3x engine for scalable management of rdf data. *VLDB*, 2010.
18. A. Owens et al. Clustered tdb: A clustered triple store for jena. 2008.
19. M. Saleem et al. FEASIBLE: A featured-based sparql benchmark generation framework. In *ISWC*, 2015.
20. M. Saleem et al. A fine-grained evaluation of sparql endpoint federation systems. *SWJ*, 2016.
21. Schätzle et al. Sempala: interactive sparql query processing on hadoop. In *ISWC*, 2014.
22. A. Schätzle et al. S2rdf: Rdf querying with sparql on spark. *VLDB*, 2016.
23. A. Schwarte et al. FedX: Optimization techniques for federated query processing on linked data. In *ISWC*, 2011.
24. Tomaszuk et al. Rdf graph partitions: A brief survey. In *BDAS*, 2015.
25. X. Wang et al. LHD: optimising linked data query processing using parallelisation. In *LDOW*, 2013.
26. Yan et al. Efficient indices using graph partitioning in rdf triple stores. In *ICDE*, 2009.
27. Zeng et al. A distributed graph engine for web scale rdf data. In *Proceedings of the VLDB Endowment*, 2013.