Semantic Web 1 (2019) 1–5
IOS Press

An Empirical Evaluation of Cost-based Federated SPARQL Query Processing Engines

Umair Qudus [a], Muhammad Saleem [b] , Axel-Cyrille Ngonga Ngomo [c] and Young-koo Lee [a,*]
[a] *DKE, Kyung Hee University, South Korea*
E-mail:{umair.qudus,yklee}@khu.ac.kr
[b] *AKSW, Leipzig, Germany*
E-mail: {lastname}@informatik.uni-leipzig.de
[c] *University of Paderborn, Germany*
E-mail: axel.ngonga@upb.de

Abstract.
Finding a good query plan is key to the optimization of query runtime. This holds in particular for cost-based federation engines, which make use of cardinality estimations to achieve this goal. A number of studies compare SPARQL federation engines across different performance metrics, including query runtime, result set completeness and correctness, number of sources selected and number of requests sent. However, although they are informative, these metrics are generic and unable to quantify and evaluate the accuracy of the cardinality estimators of cost-based federation engines. In addition, to thoroughly evaluate cost-based federation engines, the effect of estimated cardinality errors on the overall query runtime performance must be measured. In this paper, we address this challenge by presenting novel evaluation metrics targeted at a fine-grained benchmarking of cost-based federated SPARQL query engines. We evaluate the query planners of five different cost-based federated SPARQL query engines using LargeRDFBench queries across. Our results suggest that our metrics are clearly correlated with the overall query runtime performance of the selected federation engines, and can hence provide important solutions when developing the future generations of federation engines.

Keywords: SPARQL, Benchmarking, cost-based, cost-free, federated, Querying

1. Introduction

The availability of increasing amounts of data published in RDF has led to the genesis of many federated SPARQL query engines. These engines vary widely in their approaches to generating a good query plan [1–4]. In general, there exist several possible plans that a federation engine can consider when executing a given query. These plans have a different cost in terms of the resources required and the overall query execution time. Selection of the best possible plan with minimum cost is hence of key importance when devising cost-based federation engines; a fact which is corroborated by a plethora of works in database research [5, 6].

In SPARQL query federation, index-free (heuristics-based) [7–9] and index-assisted (cost-based) [10–20] engines are most commonly used for federated query processing [1]. The heuristics-based federation engines do not store any pre-computed statistics and hence mostly use different heuristics to optimize their query plans [7]. Cost-based engines make use of an index with pre-computed statistics about the datasets [1]. Using cardinality estimates as principal input, such engines make use of cost models to calculate the cost of different query joins and generate optimized query plans. Most state-of-the-art cost-based federated SPARQL

*Corresponding author. E-mail: yklee@khu.ac.kr

processing engines [10, 11, 13–19] achieve the goal of optimizing their query plan by first estimating the cardinality of the query's triple patterns. Then, they use this information to estimate the cardinality of the joins involved in the query. A cost model is then used to compute the cost of performing different query joins while considering network communication costs. One of the query plans with minimum execution costs is finally selected for result retrieval. Since the principal input for cost-based query planning is the cardinality estimates, the accuracy of these estimates is crucial to achieve a good query plan.

The performance of federated SPARQL query processing engines has been evaluated in many recent studies [1, 10, 13, 15, 21–28] by using different federated benchmarks [29–37]. Performance metrics, including query execution time, number of sources selected, source selection time, query planning time, continuous efficiency of query processing, answer completeness and correctness, time for the first answer, and throughput, are usually reported in these studies. While these metrics allow the evaluation of certain components (e.g., the source selection model), they cannot be used to evaluate the accuracy of the cardinality estimators of the cost-based federation engines. Consequently, they are unable to show how the estimated cardinality errors affect the overall query runtime performance of federation engines.

In this paper, we address the problem of measuring the accuracy of the cardinality estimators of federated SPARQL engines, as well as the effect of these errors on the overall query runtime performance. In particular, we propose metrics[1] for measuring errors in the cardinality estimations of (1) triple patterns, (2) joins between triple patterns, and (3) query plans. We correlate these errors with the overall query runtime performance of state-of-the-art, cost-based SPARQL federation engines. The observed results show that these metrics are correlated with the overall runtime performances. In summary, the contributions of this work are as follows:

- We propose metrics to measure the errors in cardinality estimations of cost-based federated engines. These metrics allow a fine-grained evaluation of cost-based federated SPARQL query engines and uncover novel insights about the performance of these types of federation engines that were not reported in previous works evaluating federated SPARQL engines.
- We measure the correlation of the values of the novel metrics with the overall query runtimes. We show that some of these metrics have a strong correlation with runtimes and can hence be used as predictors for the overall query execution performance.
- We present an empirical evaluation of five—CostFed[10], Odyessey[13], SemaGrow [15], LHD [11] and SPLENDID[14]—state-of-the-art cost-based SPARQL federation engines on LargeRDFBench [29] by using the proposed metrics along with existing metrics, affecting the query runtime performance.

The rest of the paper is organized as follows: In Section 2, we present related work. A motivating example is given in Section 3. In Section 4, we present our novel metrics for the evaluation of cost-based federation engines. In Section 5, we give an overview of the cardinality estimators of selected cost-based federation engines. The evaluation of these engines with proposed as well as existing metrics is shown in Section 6. Finally, we conclude in Section 7.

2. Related Work

In this section, we focus on the performance metrics used in the state of the art to compare federated SPARQL query processing engines. Based on the previous federated SPARQL benchmarks [29–31] and performance evaluations [7, 10–15, 21, 22, 25, 27, 28] (see Table 1 for an overview), the performance metrics used for federated SPARQL engines comparison can be categorized as:

- **Index-Related**: Index-assisted approaches [1] make use of stored dataset statistics to generate an optimized query execution plan. The indexes are pre-computed by collecting information from available federated datasets. This is usually a one-time process. However, later updates are required to ensure the result-set completeness of the query processing. The index generation time and its compression ratio (w.r.t. overall dataset size) are important measures to be considered when devising index-assisted federated engines.
- **Query-Processing-Related**: This category contains metrics related to the query processing capabilities of the federated SPARQL engines. The to-

[1] Our proposed metric is open-source and available online at https://github.com/dice-group/CostBased-FedEval

	Index		Processing					Network		Res		RS		Add	
	Cr	Gt	Qp	#Ts	Qet	#A	Sst	#Tt	#Er	Cu	Mu	Cp	Ct	@T	@K
CostFed[10]	✓	✓		✓	✓	✓	✓					✓			
SPLENDID[14]				✓	✓				✓						
SemaGrow[15]			✓	✓	✓										
Odyssey[13]			✓	✓	✓			✓	✓			✓			
LHD[11]					✓			✓	✓	✓	✓	✓			
DARQ[12]			✓		✓										
ANAPSID[25]					✓										
MULDER[22]					✓							✓	✓	✓	✓
FedX[7]					✓	✓			✓						
Lusail[21]			✓		✓		✓								
BioFed[28]				✓	✓	✓	✓					✓	✓		

Table 1: Metrics used in the existing federated SPARQL query processing systems, **Res:** Resource Related, **RS:** Result Set Related, **Add:** Additional, **Cr:** index compression ratio, **Gt:** the index/summary generation time, **Qp:** Query Planning time, **#Ts:** total number of triple pattern-wise sources selected, **Qet:** the average query execution time, **#A:** total number of SPARQL ASK requests submitted, **Sst:** the average source selection time, **#Tt:** number of transferred tuples, **#Er:** number of endpoint requests, **Cu:** CPU usage, **Mu:** Memory usage, **Cp:** Result Set completeness, **Ct:** Result Set correctness, **@K:** dief@k, **@T:** dief@t

tal number of triple-pattern-wise sources selected, number of `ASK` requests used to perform source selection, source selection time, query planning time, and overall query runtime are the reported metrics in this category.

- **Network-Related:** Federated engines collect information from multiple data sources, e.g., SPARQL endpoints. Thus, it is important to minimize the network traffic generated by the engines during query processing. The number of transferred tuples and the number of endpoint requests generated by the federation engine are the two network related metrics used in existing federated SPARQL query processing evaluations.
- **Result-Set-Related:** Two systems are only comparable if they produce exactly the same results. Therefore, result set correctness and completeness are the two most important metrics in this category.
- **Resource-Related:** The CPU and memory resources consumed during query processing dictate the query load an engine can bear. Hence, they are of importance when evaluating the performance of federated SPARQL engines.
- **Additional:** Two metrics dief@t and dief@k are proposed to measure continuous efficiency of query processing approaches.

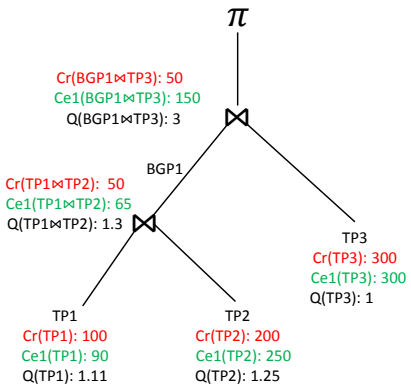
All of these metrics are helpful to evaluate the performance of different components of federated query engines. However, none of these metrics can be used to evaluate the accuracy of the cardinality estimators of cost-based federation engines. Consequently, studying the effect of estimated cardinality errors on the overall query runtime performance of federation engines cannot be conducted based on these metrics. To overcome these limitations, we propose metrics for measuring errors in cardinality estimations of triple patterns, joins between triple patterns, and overall query plan, and show how these metrics are affecting the overall runtime performance of federation engines.

3. Motivating Example

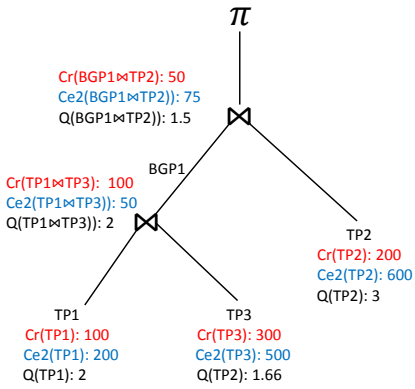
In this section, we present an example to motivate our work and to understand the proposed metrics. We assume that the reader is familiar with the concepts of SPARQL and RDF, including the notions of a triple pattern, the joins between triple patterns, the cardinality (result size) of a triple pattern, and left-deep query execution plans. As aforementioned, most cost-based SPARQL federation engines first estimate individual triple pattern cardinality and use this information to esti-

```
SELECT * WHERE {
?s :p1 ?o1 .
Cr(TP1):100
Ce1(TP1):90
Ce2(TP1):200

?s :p2 ?o2 .
Cr(TP2):200
Ce1(TP2):250
Ce2(TP2):600

?s :p3 ?o3 .
Cr(TP3):300
Ce1(TP3):300
Ce2(TP3):500 }
```

(a) Example query



(b) Engine 1 optimal query plan



(c) Engine 2 subOptimal query plan

Fig. 1.: Motivating Example: A sample SPARQL query and the corresponding query plans of two different federation engines

mate the cardinality of joins found in the query. Finally, the query execution plan is generated by ordering the joins. In general, the optimizer first selects the triple patterns and joins with minimum estimated cardinalities [10].

Figure 1 shows a motivating example containing a SPARQL query with three triple patterns—namely TP1, TP2 and TP3—and two joins. Consider two different cost-based federation engines with different cardinality estimators. Figure 1a shows the real (`Cr`) and estimated cardinalities (`Ce1` for Engine 1 and `Ce2` for Engine 2) for triple patterns of the query. Let us assume that both engines generate left-deep query plans by selecting triple patterns with the smallest cardinalities to perform their first join. The results of this join are then used to perform the second join with the remaining third triple pattern. By using actual cardinalities, the optimal query execution plan would be to first perform the join between TP1 and TP2 and then perform the second join with TP3. The same plan will be generated by Engine 1 as well, as shown in Figure 1b. The subOptimal plan generated by Engine 2 is given in Figure 1c. Note that Engine 2 did not select the optimal plan because of large errors in cardinality estimations of triple patterns and joins between triple patterns.

The motivating example clearly shows that good cardinality estimations are essential to produce a better query plan. The question we aim to answer pertains to how much the accuracy of cardinality estimations affects the overall query plan and the overall query runtime performance. To answer this question, the q-error (Q in Figure 1) was introduced in [6] in the database literature. In the next section, we define this measure and propose new metrics based on similarities to measure the overall triple patterns error E_T, overall joins error E_J as well as overall query plan error E_P.

4. Cardinality Estimation-related Metrics

Now we formally define the q-error and our proposed metrics, namely E_T, E_J, E_P to measure the overall error in cardinality estimations of triple patterns, joins between triple patterns and overall query plan error, respectively.

4.1. q-error

The q-error is the factor by which an estimated cardinality value differs from the actual cardinality value [6].

Definition 1 (q-error). *Let $\vec{r} = (r_1, \ldots, r_n) \in \mathbb{R}$ where $r_i > 0$ be a vector of real values and $\vec{e} = (e_1, \ldots, e_n) \in \mathbb{R}$ be the vector of the corresponding estimated values. By defining $\vec{e}/\vec{r} = \frac{\vec{e}}{\vec{r}} = (e_1/r_1, \ldots, e_n/r_n)$, then q-error of estimation e of r is given as*

$$||e/r||_Q = \max_{1 \leqslant i \leqslant n} ||e_i/r_i||_Q, \text{ where}$$
$$||e_i/r_i||_Q = \max(e_i/r_i, r_i/e_i)$$

In this definition, over- and underestimations are treated symmetrically [6]. In the motivating example given in Figure 1, the real cardinality of TP1 is 100 (i.e., Cr(TP1)=100) while the estimated cardinality by engine 1 for the same triple pattern is 90 (i.e., Cr(TP1) = 90). Thus, the q-error for this individual triple pattern is max(90/100,100/90) = 1.11. The query's overall q-error of its triple patterns (see Figure 1b) is the maximum value of all the q-error values of triple patterns, i.e., $\max(1.11, 1.25, 1) = 1.25$. The q-error of the complete query plan would be the maximum q-error values in all triple patterns and joins used in the query plan, i.e., $\max(1.11, 1.25, 1, 1.3, 3) = 3$.

The q-error makes use of the ratio instead of an absolute or quadratic difference and is hence able to capture the intuition that only relative differences matter for making planning decisions. In addition, the q-error provides a theoretical upper bound for the plan quality if the q-error of a query is bounded. Since it only considers the maximum value amongst those calculated, it is possible that plans with good average estimations are regarded as poor by this measure. Consider the query plans given in Figure 1b and Figure 1c. Both have a q-error of 3, yet the query plan in Figure 1b is optimal, while the query plan in Figure 1c is not. To solve this problem, we introduce the additional metrics defined below.

4.2. Similarity Errors

The overall similarity error of query triple patterns is formalised as follows:

Definition 2 (Triple Patterns Error E_T)**.** *Let Q be a SPARQL query containing triple patterns $T = \{TP_1, \ldots, TP_n\}$. Let $\vec{r} = (Cr(TP_1), \ldots, Cr(TP_n)) \in \mathbb{R}^n$ be the vector of real cardinalities of T and $\vec{e} = (Ce(TP_1), \ldots, Ce(TP_n)) \in \mathbb{R}^n$ be the vector of the corresponding estimated cardinalities of T. Then, we define our overall triple pattern error as follows:*

$$E_T = \frac{\|\vec{r}-\vec{e}\|}{\|\vec{r}\|+\|\vec{e}\|} = \frac{\sqrt{\sum_{i=1}^{n}(Cr(TP_i)-Ce(TP_i))^2}}{\sqrt{\sum_{i=1}^{n}(Cr(TP_i))^2}+\sqrt{\sum_{i=1}^{n}(Ce(TP_i))^2}}$$

In the motivating example given in Figure 1, the real cardinalities vector $\vec{r}$ = (100,200,300) and the Engine 1 estimated cardinalities vector $\vec{e}$ = (90,250,300). Thus, $E_T = 0.0658$. Similarly, Engine 2 estimated cardinality vector is $\vec{e}$ = (200,500,600). Thus, Engine 2 achieves $E_T = 0.388$.

Definition 3 (Joins Error E_J)**.** *Let Q be a SPARQL query containing joins $J = \{J_1, \ldots, J_n\}$. Let $\vec{r} = (Cr(J_1), \ldots, Cr(J_n)) \in \mathbb{R}^n$ a vector of real cardinalities of J and $\vec{e} = (Ce(J_1), \ldots, Ce(J_n)) \in \mathbb{R}^n$ be the vector of the corresponding estimated cardinalities of J, then the overall joins error is defined by the same equation in Definition 2.*

Definition 4 (Query Plan Error E_P)**.** *Let Q be a SPARQL query and TJ be the set of triple patterns and joins in Q. Let $\vec{r} = (r_1, \ldots, r_n) \in \mathbb{R}^n$ be a vector of real cardinalities of TJ and $\vec{e} = (e_1, \ldots, e_n) \in \mathbb{R}^n$ be the vector of corresponding estimated cardinalities of TJ, then the overall query plan error is defined by the same equation in Definition 2.*

In the motivating example given in Figure 1b, the real cardinalities vector of all triple patterns and joins, $\vec{r}$ = (100,200,300,50,50) and the Engine 1 estimated cardinalities vectors $\vec{e}$ = (90,250,300,65,150). Thus, E_P = 0.1391 for Engine 1. Engine 2 achieves E_P = 0.3838. In these matrices, over- and underestimations are also treated symmetrically. The purpose of these definitions is to keep the lower bound at 0, which could be reached if r = e, i.e., there is no error in the estimation; and the upper bound at 1, which could be reached if *e* is much larger than *r*.

5. Selected Federation Engines

In this section, we give a brief overview of the selected cost-based SPARQL federation engines. In particular, we describe how the cardinality estimations for triple patterns and joins between triple patterns are performed in these engines.

CostFed: CostFed [10] makes use of pre-computed statistics stored in index to estimate the cardinality of triple patterns and joins between triple patterns. CostFed benefits from both bind join ($\bowtie_b$) [7, 14, 15] and symmetric hash join ($\bowtie_h$) [25] for joining the results of triple patterns. The decision of join selection is based on calculating the cost of both joins on query runtime. It creates 3 buckets for each distinct predicate used in the RDF dataset. These buckets are used for estimating the cardinality of query triple patterns. Furthermore, Costfed stores selectivity information that is used to estimate the cardinality of triple patterns as well as devising an efficient query plan. CostFed query planner also considers the skew in the distribution of objects and subjects across predicates. Separate cardinality estimation is used for Multi-valued predicates. Multi-valued predicates are the predicates that can have

multiple values, as people can have multiple contact numbers or graduation schools. It performs a join-aware trie-based source selection, which uses common URI prefixes.

Let D represent a datasets or source for short, $tp =< s, p, o >$ be a triple pattern having predicate p, and $R(tp)$ be the set of relevant sources for that triple pattern. The following notations are used to calculate the cardinality of tp.

- $T(p, D)$ is the total number of triples with predicate p in D.
- $avgSS(p, D)$ is the average subject selectivity of p in D.
- $avgOS(p, D)$ is the average subject selectivity of p in D.
- $tT(D)$ is the total number of triples in D.
- $tS(D)$ is the total number of distinct subjects in D.
- $tO(D)$ is the total number of distinct objects in D.

From these notations the cardinality $C(tp)$ of tp is calculated as follows (the predicate b stands for bound):

$$\begin{cases} \sum_{\forall Di \in R(tp)} T(p, D_i) \times 1 \\ \qquad \Rightarrow \text{if } b(\text{p}) \wedge !b(\text{s}) \wedge !b(\text{o}), \\ \sum_{\forall Di \in R(tp)} T(p, D_i) \times avgSS(p, D_i) \\ \qquad \Rightarrow \text{if } b(\text{p}) \wedge b(\text{s}) \wedge !b(\text{o}), \\ \sum_{\forall Di \in R(tp)} T(p, D_i) \times avgOS(p, D_i) \\ \qquad \Rightarrow \text{if } b(\text{p}) \wedge !b(\text{s}) \wedge b(\text{o}), \\ \sum_{\forall Di \in R(tp)} tT(D_i) \times 1 \\ \qquad \Rightarrow \text{if } !b(\text{p}) \wedge !b(\text{s}) \wedge !b(\text{o}), \\ \sum_{\forall Di \in R(tp)} tT(D_i) \times \frac{1}{tS(D_i)} \\ \qquad \Rightarrow \text{if } !b(\text{p}) \wedge b(\text{s}) \wedge !b(\text{o}), \\ \sum_{\forall Di \in R(tp)} tT(D_i) \times \frac{1}{tO(D_i)} \\ \qquad \Rightarrow \text{if } !b(\text{p}) \wedge !b(\text{s}) \wedge b(\text{o}), \\ \sum_{\forall Di \in R(tp)} tT(D_i) \times \frac{1}{tS(D_i) \times tO(D_i)} \\ \qquad \Rightarrow \text{if } !b(\text{p}) \wedge b(\text{s}) \wedge b(\text{o}), \\ 1 \qquad \Rightarrow \text{if } b(\text{p}) \wedge b(\text{s}) \wedge b(\text{o}) \end{cases}$$

A recursive definition is used to define the SPARQL expression E [15, 38] in the query planning phase and is defined as follows: all triple patterns are SPARQL expressions and if $E1$ and $E2$ are SPARQL expressions then $E1 \bowtie E2$ is also a SPARQL expression. The join cardinality of two expressions E_1 and E_2 is estimated as follows:

$$C(E1 \bowtie E2) = M(E1) \times M(E2) \times Min(C(E1), C(E2))$$

where the average frequency of multi-valued predicates in the expression E is defined as $M(E)$. In $M(E)$, E is not the result of joins between triple patterns but the triple pattern itself. $M(E)$ is calculated using the following equation:

$$M(E) = \begin{cases} 1/\sqrt{2} & \text{if } b(\text{p}) \wedge !b(\text{s}) \wedge b(\text{o}), \\ C(E)/distSbjs(p, \mathcal{D}) & \text{if } b(\text{p}) \wedge !b(\text{s}) \wedge !b(\text{o}) \wedge \text{j(s)}, \\ C(E)/distObjs(p, \mathcal{D}) & \text{if } b(\text{p}) \wedge !b(\text{o}) \wedge !b(\text{s}) \wedge \text{j(o)}, \\ 1 & \text{other} \end{cases}$$

If subject of the triple pattern is involved in the join , it is defined as $j(s)$, and b(s), b(o), and b(p) are defined as bound subject, object, predicate respectively.

SPLENDID: SPLENDID [14] also uses VoID statistics to generate a query execution plan. It uses a dynamic programming approach to produce a query execution plan. SPLENIDID makes use of both hash($\bowtie_h$) and bind($\bowtie_b$) joins.

Triple pattern cardinality is estimated as follows:

$$\begin{aligned} \text{card}_d(?, p, ?) &= \text{card}_d(p) \\ \text{card}_d(s, ?, ?) &= |d| \cdot \text{sel}.s_d \\ \text{card}_d(s, p, ?) &= \text{card}_d(p) \cdot \text{sel}.\text{s}_d(p) \\ \text{card}_d(?, ?, o) &= |d| \cdot \text{sel}.o_d \\ \text{card}_d(?, p, o) &= \text{card}_d(p) \cdot \text{sel}.o_d(p) \\ \text{card}_d(s, ?, o) &= |d| \cdot \text{sel}.s_d \cdot \text{sel}.o_d \end{aligned}$$

where the $card_d(p)$ is the number of triple patterns in the data source d having predicate p. The total number of triples in a data source d is defined as $|d|$. If we have a bound predicate then the average selectivity of subject and object is defined as $sel.s_d(p)$ and $sel.o_d(p)$ respectively; if the predicate is not bound then the average selectivity of subject and object is defined as $sel.s_d$ and $sel.o_d$ respectively In star-shaped queries, SPLENDID estimates the cardinality of triple patterns having the same subject separately. All triples with same subjects are grouped and then the minimum cardinality of all

triple patterns with bound object is calculated. Lastly, the cardinality of remaining triples with unbound objects is multiplied with the average selectivity of subjects and the minimum value. Formally the equation is defined as:

$$\text{card}_d(T) = \min\left(\text{card}_d\left(T_{bound}\right)\right) \cdot \prod\left(sel.s_d \cdot \text{card}_d\left(T_{unbound}\right)\right)$$

Join cardinality is estimated as follows.

$$\text{card}(q1 \bowtie q2) = \text{card}\left(q_1\right) \cdot \text{card}\left(q_2\right) \cdot \text{sel}_{\bowtie}(q1, q2)$$

In these equations the $sel_{\bowtie}$ is the join selectivity of two input relations. It defines that how many bindings are matched between two relations. SPLENDID uses the average selectivity of join variables as join selectivity.

LHD: LHD [11] is a cardinality-based and index-assisted approach that aims to maximize parallel execution of sub-queries. It makes use of the VoID statistics for estimating the cardinality of triple patterns and joins between triple patterns. LHD only uses Bind joins for query execution. LHD implements a response-time-cost model by making an assumption that the response time of a query request is proportional to the total number of bindings transferred. LHD determines the total number of triples t_d, distinct subjects s_d and objects o_d from the VoID description of a dataset d. The VoID file also provides other information, such as the number of triples $t_{d.p}$, distinct subjects $s_{d.p}$ and distinct objects $o_{d.p}$ in the dataset d for a predicate p. The federation engine makes an assumption about uniform distribution of objects and subjects in datasets. Let's assume a triple pattern $T : \{SPO\}$ [2], the function to get the set of relevant datasets of T is defined as $S(T)$, the selectivity of x with respect to $S(T)$ is defined as $selT(x)$, and the cardinality of x with respect to $S(T)$ is defined as $cardT(x)$.

For single triple pattern cardinality estimation, the selectivity of each part is estimated as follows:

$$\text{sel}_T(S) = \begin{cases} \frac{\sum_{d \in S(T)} t_d / s_d}{\sum_{d \in S(T)} s_d} & \text{if } \text{var}(P) \wedge \neg \text{var}(S) \\ \frac{\sum_{d \in S(T)} t_{d \cdot p} / s_{d \cdot p}}{\sum_{d \in S(T)} s_{d \cdot p}} & \text{if } P = p \wedge \neg \text{var}(S) \\ 1 & \text{if } \text{var}(S) \end{cases}$$

$$\text{sel}_T(P) = \begin{cases} \frac{\sum_{d \in S(T)} t_{d.p}}{\sum_{d \in S(T)} t_d} & \text{if } P = p \\ 1 & \text{if } \text{var}(P) \end{cases}$$

$$\text{sel}_T(O) = \begin{cases} \frac{\sum_{d \in S(T)} t_d / o_d}{\sum_{d \in S(T)} o_d} & \text{if } \text{var}(P) \wedge \neg \text{var}(O) \\ \frac{\sum_{d \in S(T)} t_{d.p} / o_{d.p}}{\sum_{d \in S(T)} o_{d.p}} & \text{if } P = p \wedge \neg \text{var}(O) \\ 1 & \text{if } \text{var}(O) \end{cases}$$

After calculating the selectivity of each part, LHD estimates the cardinality of the triple pattern as follows:

$$\text{card}(T) = t \cdot \text{sel}_T(S) \cdot \text{sel}_T(P) \cdot \text{sel}_T(O)$$

Given two triple patterns T1 and T2, LHD calculates the join selectivity by using the following equations:

$$\text{sel}\left(T_1 \bowtie T_2\right) = \begin{cases} \frac{\sum_{d \in S(T_1)} s_{d.p1} \cdot \sum_{d \in S(T_2)} s_{d.p2}}{\sum_{d \in S(T_1)} s_d \cdot \sum_{d \in S(T_2)} s_d} & \text{if joined on } S_1 = S_2 \\ \frac{\sum_{d \in S(T_1)} o_{d.p1} \cdot \sum_{d \in S(T_2)} o_{d.p2}}{\sum_{d \in S(T_1)} o_d \cdot \sum_{d \in S(T_2)} o_d} & \text{if joined on } O_1 = O_2 \\ \frac{\sum_{d \in S(T_1)} s_{d.p1} \cdot \sum_{d \in S(T_2)} o_{d.p2}}{\sum_{d \in S(T_1)} s_d \cdot \sum_{d \in S(T_2)} o_d} & \text{if joined on } S_1 = O_2 \\ 1 & \text{if no shared variables.} \end{cases}$$

Using the join selectivity values, join cardinality is estimated by the following equation:

$$\text{card}\left(T_1 \bowtie T_2 \bowtie \ldots \bowtie T_n\right) = \prod_{i=1}^{n} \text{card}\left(T_i\right) \cdot \text{sel}\left(T_1 \bowtie T_2 \bowtie \ldots \bowtie T_n\right)$$

SemaGrow: SemaGrow [15] query planning is based on VoID[3] statistics[39] about datasets. It makes use of the VoID index as well as SPARQL ASK queries to perform source selection. Three types of joins, i.e, bind, merge, and hash are used during the query planning.

[2] In this section, the letters with a question mark (e.g. ?x) denote a variable in an RDF triple - a literal value is represented by a lower-case letter (e.g. o) , and a variable or a literal value is defined by an upper-case letter (e.g. S)

[3] VoID vocabulary: https://www.w3.org/TR/void/

The selection to perform the required join operation is based on a cost function. It uses a reactive model for retrieving results of the joins as well as individual triple patterns. As with CostFed, SemaGrow recursively defines SPARQL expressions. Given a data source S, the cardinality estimations of triple patterns and joins is explained below.

SemaGrow contains a Resource discovery component, which returns the list of relevant sources to a triple pattern along with statistics. The statistics related to the data source include (1) the number of estimated distinct subjects, predicates, and objects matching the triple pattern, and (2) the number of triple patterns in the data sources matching the triple pattern. The cardinality of a triple pattern is provided by the Resource Discovery component. On the other hand, for more complex expressions, SemaGrow needs to make an estimation based on available statistics. In order to estimate complex expressions based on the aforementioned basic statistics, SemaGrow adopts the formulas described by LHD [11]. The cardinality of each expression(E) in a data source S, is defined as $Card([E], S)$).

For estimating the join cardinality we need to calculate the join selectivity (JoinSel $([E1] \bowtie [E2])$), which is given as follows:

$$\text{JoinSel } ([E1] \bowtie [E2]) = \min(\text{ JoinSel } [E1],\ \text{JoinSel } [E2])$$

$$\text{JoinSel } ([T]) = \min(1/d_i)$$

In these equations E1 and E2 reside any join expressions or triple patterns. The T is a single triple pattern. d_i represents the number of distinct values for the i-st join attribute in a T. Hence, the join cardinality is given as following:

$$\text{Card}([E1] \bowtie [E2], S) = \text{Card}([E1], S) \cdot \text{Card}([E2], S) \cdot \text{ JoinSel } ([E1] \bowtie [E2])$$

Odyssey: Odyssey [13] makes use of the distributed characteristic sets (CS) [40] and characteristic pair (CP) [41] statistics to estimate cardinalities. Odyssey estimates the cardinality of each type of query differently using these statistics. For star-shaped queries, where the subject (or object) is the same for all joining triple patterns; estimated cardinality for a given set of properties P (predicates of joining triple patterns) is computed using CSs C_j containing all these properties. The common subject (or object) is defined as an entity. CSs can be computed by scanning once a dataset's triples are sorted by subject; after all the entity properties have been scanned, the entity's CS is identified. For each CS C, Odyssey computes statistics, i.e., $(count(C))$ represents the number of entities sharing C and $(occurrences(p, C))$ represents the number of triples with predicate p occurring with these entities.

Odyssey represents estimatedCardinality$_{Distinct}(P)$ as the estimated cardinality of queries that contain distinct keywords, and estimatedCardinality(P) as the estimated cardinality of those queries that do not contain the distinct keyword. Formally, estimated cardinality for star-shaped queries is defined as following:

$$\text{estimatedCardinality }_{Distinct}(P) = \sum_{P \subseteq C_j} (\text{count } (C_j))$$

$$\text{estimatedCardinality } (P) = \sum_{P \subseteq C_j} \left(\text{count } (C_j) \cdot \prod_{p_i \in P} \frac{\text{ocurrences } (p_i, C_j)}{\text{count } (C_j)} \right)$$

For arbitrary-shaped queries, Odyssey also considers the connections (links) between different CSs. Characteristic pairs (CPs) help in describing the links between Characteristic sets (CSs) using properties. For entities $e1$ and $e2$ the link is defined as $(CS\,s(e1), CS\,s(e2), p)$, given that $(e1, p, e2) \in s$, where s is data source. The number of links between two *CS s*: C_i and C_j, through a property p, is represented in statistics, which is defined as: – $count(Ci, C\,j, p)$. The equation for estimating the cardinality (pairs of entities with a set of properties P_k and P_l) for complex-shaped queries is defined as:

$$\text{estimatedCardinality } (P_k, P_l, p) = \sum_{P_k \subseteq C_i \wedge P_l \subseteq C_j} (\text{count } (C_i, C_j, p) \cdot \prod_{p_k \in P_k - \{p\}} \left(\frac{\text{ocurrences } (p_k, C_i)}{\text{count}(C_i)} \right) \cdot \prod_{p_l \in P_l} \left(\frac{\text{ocurrences } (p_l, C_j)}{\text{count}(C_j)} \right))$$

In order to reduce the complexity, Odyssey treats each star-shaped query as a single meta-node; assuming that the order of joins has already optimized within the star-shaped sub-queries. It uses Characteristics Pairs (CPs) to estimate the cardinality of joins between star-shaped queries (meta-nodes) and uses dynamic programming (DP) to optimize the join order and find the optimal plan.

6. Evaluation and Results

In this section, we discuss our evaluation results. Complete results of our evaluation are also available from our resource homepage. First, we evaluate our novel metrics in terms of how they are correlated with the overall query runtime performances. Thereafter, we compare existing cost-based SPARQL federation engines against the proposed metrics and discuss the evaluation results.

6.1. Experiment Setup and Hardware:

Benchmarks Used: In our experiments, we used the most recent state-of-the-art benchmark for federated engines dubbed LargeRDFBench [29]. The benchmark includes all FedBench[30] queries. LargeRDFBench comprises a total of 40 queries: 14 simple queries (S1-S14) from FedBench, 10 complex queries (C1-C10), 8 complex plus high sources queries (CH1-CH8), and 10 large data queries (L1-L10). Simple queries are relatively fast to execute and include the smallest number of triple patterns ranges from 2 to 7 [29]. Complex queries are more challenging and take more time to execute as compared to simple queries [29]. The queries in this category have least 8 triple patterns and contain more joins and SPARQL operators as compared to simple queries. The complex plus high sources queries are even more challenging as they need to retrieve results from more data sources and they have more triple patterns, joins and SPARQL operators as compared to simple and complex queries [29].

We used all queries except large data queries (L1-L10) in our experiments. The reason for skipping L1-L10 was that the evaluation results [29] show that most engines are not yet able to execute these queries. LargeRDFBench comprises a total of 13 real-world RDF datasets of varying sizes. We loaded each dataset into a Virtuoso 7.2 server.

Cost-based Federation Engines: We evaluated five—CostFed [10], Odyessey[13], SemaGrow[15], LHD[11] and SPLENDID[14]—state-of-the-art cost-based SPARQL federation engines. To the best of our knowledge, these are most of the currently available, open-source cost-based federation engines.

Hardware Used: Each Virtuoso was deployed on a physical machine (32 GB RAM, Core i7 processor and 500 GB hard disk). We ran the selected federation engines on a local client machine with same specifications. Our experiments were run in a local environment where the network cost is negligible. This is a standard setting used in the original LargeRDFBench. Please note that the accuracy of the cardinality estimators of the federated SPARQL query processing is independent of the network cost.

Warm-up and Number of Runs: We warmed up each federation engine for 10 minutes by executing the Linked Data (LD1-LD10) queries from FedBench. Experiments were run 3 times and the results were averaged. The query timeout was set to 30 minutes.

Metrics: We present results for: (1) q-error of triple patterns, (2) q-error of joins between triple patterns, (3) q-error of overall query plans, (4) errors of triple patterns, (5) errors of joins between triple patterns, (6) errors of overall query plans, (7) the overall query runtimes, (8) the number of tuples transferred (intermediate results), (9) the source selection related metrics, and (10) quality of plans generated by query planner of each engine. In addition, we used Spearman's correlation coefficient to measure the correlation between the proposed metrics and the overall query runtimes. The Spearman test is designed to assess how well the dependency between two variables can be described using a monotonic function. We preferred this test over the Pearson test given that it is parameter-free and tests at rank level. Using the Pearson correlation does not perform well in this context given the intrinsic highly multivariate nature of the variables (e.g., runtimes) used in these tests. We used simple linear and robust regression models to compute the correlation.

6.2. Regression Experiments

First, we wanted to investigate the dependency between proposed metrics and overall query runtime performance of the federation engines. Figure 2 shows the results of a simple linear regression experiment aiming to compute the dependency between q-error and similarity errors, and the overall query runtimes. For a particular engine, the left figure shows the dependency between the q-error and overall runtime while the right figure in the same row shows the result of the corresponding similarity error. The higher coefficients (dubbed R in the figure) computed in the experiments with similarity errors suggest that it is likely that the similarity errors are a better predictor for runtime. The positive value of the coefficient suggests that an increase in similarity error also means an increase in the overall runtime. It can be observed from the figure that the outliers are contaminating the results. In Figure

3, we further apply robust regression[42–44] using the Huber loss function [45] to remove the outliers from the results (especially for q-errors). This is because we wanted to avoid the possible high impact of outliers. We observe that after removing outliers using robust regression, the similarity-based error correlation further increases. The lower p-values in the similarity-error-based experiments further confirm that our metrics are more likely to be a better predictor for runtime than the q-error. The reason for this result is clear: Our measures exploit more information and are hence less affected by outliers. This is not the case for the q-error, which can be perturbed significantly by a single outlier.

To investigate the correlation between metrics and runtimes further, we measured Spearman's correlation coefficient between query runtimes and corresponding errors of each of the first six metrics. The results are shown in Table 2 which shows that the proposed metrics on average have positive correlations with query runtimes, i.e., the smaller the error, the smaller the query runtimes. The similarity error of overall query plan (E_P) has the highest impact (i.e. 0.35) on query runtimes, followed by the similarity error of the triple pattern (i.e. E_T with 0.27), q-error of joins (i.e. Q_J with 0.26), similarity error of Join (i.e. E_J with 0.22), q-error of overall plan (i.e. Q_P with 0.17), and q-error of triple patterns (i.e. Q_T with 0.06), respectively.

In order to do a fair comparison between the results, we only take the common queries on which every system passed. We eliminate the LHD [11], because it failed in 20/32 benchmark queries, (which is a very high number and only 12 simple queries passed), and is not adequate for comparison. We apply Spearmam's correlation again on common queries. Table 3 shows that the proposed metric has a stronger positive correlation with query runtime when we deal with only common queries. The similarity error of overall plan (E_P) and triple pattern (E_T) has the highest impact (i.e. 0.40) on query runtime, followed by similarity error of joins (i.e. E_J with 0.39), q-error of joins (i.e. Q_J with 0.17) and overall query plan (i.e. Q_P with 0.17), and q-error of triple patterns (i.e. Q_T with 0.01), respectively.

Furthermore, we remove outliers influencing results by applying robust regression on both the q-error and proposed similarity error metrics. Robust regression is done by Iterated Re-weighted Least Squares (IRLS) [42]. We used Huber weights[45] as weighting function in IRLS. This approach further fine tuned the results and made the correlation for our proposed similarity error and run time stronger. Table 4 shows that all metrics have a stronger positive correlation. However, in our proposed metric this difference is definite. The similarity error of overall query plan (E_P) has the highest impact (i.e. 0.56) on query runtimes, followed by the similarity error of the triple pattern (i.e. E_T with 0.49), similarity error of joins (E_J with 0.45), q-error of joins (i.e. Q_J with 0.22), q-error of overall plan (i.e. Q_P with 0.18) and triple pattern (i.e. Q_P with 0.18), respectively. Table 4 also shows that the q-error for Odyssey is negatively correlated with runtime. We can also observe high q-error values from Figure 4.

Another important factor that is worth mentioning is that the robust regression does not abide by the normality assumptions. By comparing the p-values (at 5% confidence level) of the simple linear regression and robust regression, this gives a hint that the data is sufficiently normally distributed for simple linear regression.

Overall, the results show that the proposed similarity errors correlate better with query runtimes than the q-error. Moreover, the correct estimation of the overall plan is clearly the most crucial fragment of the plan generation. Thus, it is important for federation engines to pay particular attention to the cardinality estimation of the overall query plan. However, given that this estimation commonly depends on triple patterns and join estimations, better means for approximating triple patterns and join cardinalities should lead to better plans. The weak to moderate correlation of the similarity errors with query runtimes suggests that the query runtime is a complex measure affected by multi-dimensional metrics, such as metrics given in the table 1 and the SPARQL features, such as number of triple patterns, their selectivities, use of projection variables, number of joins and their types[46]. Therefore, it is rather hard to pinpoint a single metric or a SPARQL feature which has a high correlation with the runtime [29, 46].

6.3. q-error and Similarity-Based Errors

We now present a comparison of the selected cost-based engines based on the 6 metrics given in Figure 4. Overall, the similarity errors of query plans given in Figure 4a suggests that CostFed produces the smallest errors followed by SPLENDID, LHD, SemaGrow, and Odyssey, respectively. CostFed produces smaller errors than SPLENDID in 10/17 comparable queries (excluding queries with timeout and runtime errors). SPLENDID produces smaller errors than LHD in 12/14 comparable queries. LHD produces smaller errors than SemaGrow in 6/12 comparable queries. In turn, SemaGrow produces smaller errors than Odyssey in 9/15 comparable queries.

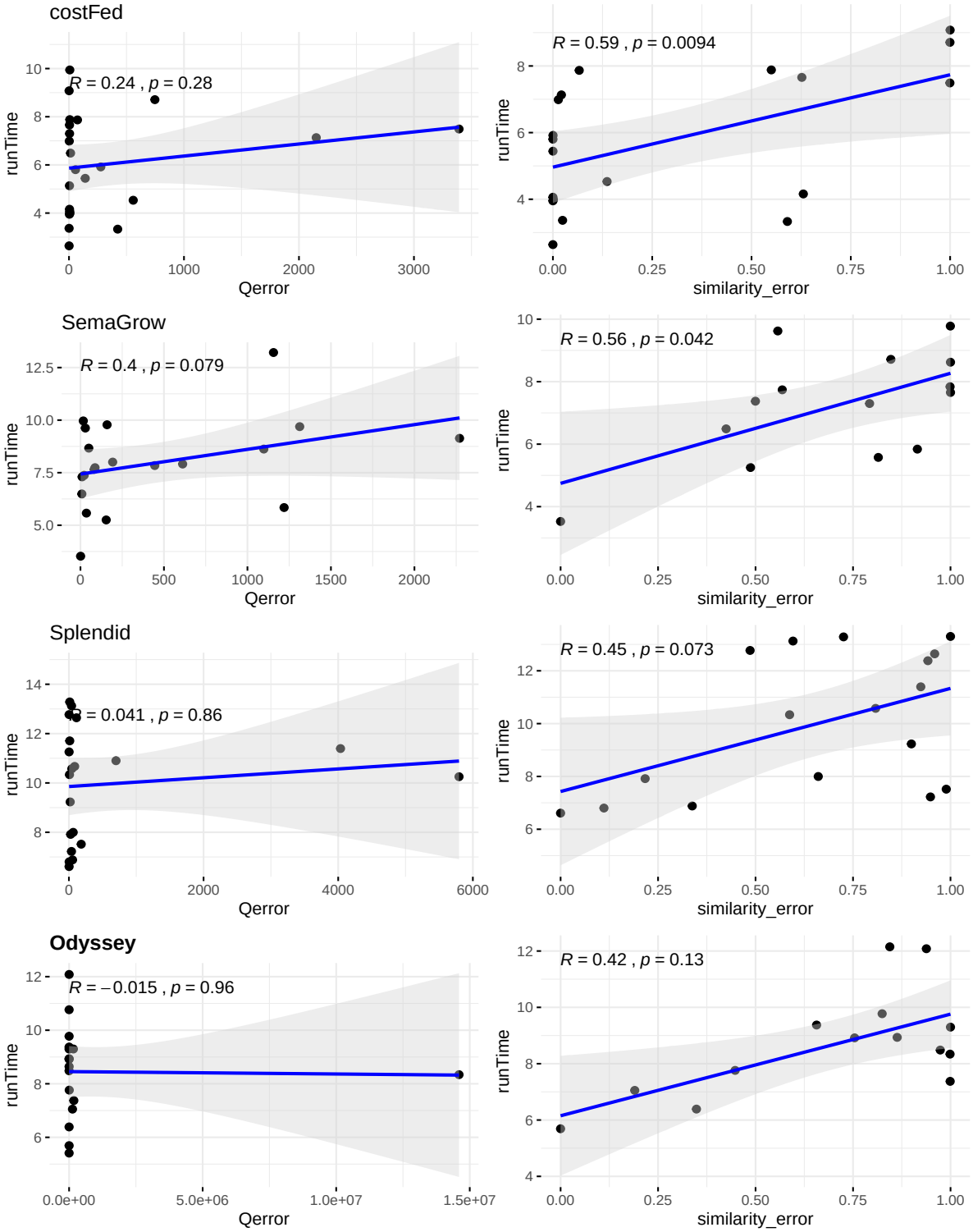


Fig. 2.: q-error and Similarity Error vs. runtime. (Simple Linear Regression Analysis). The grey shaded area in graphs represents the confidence interval (band) in regression line.

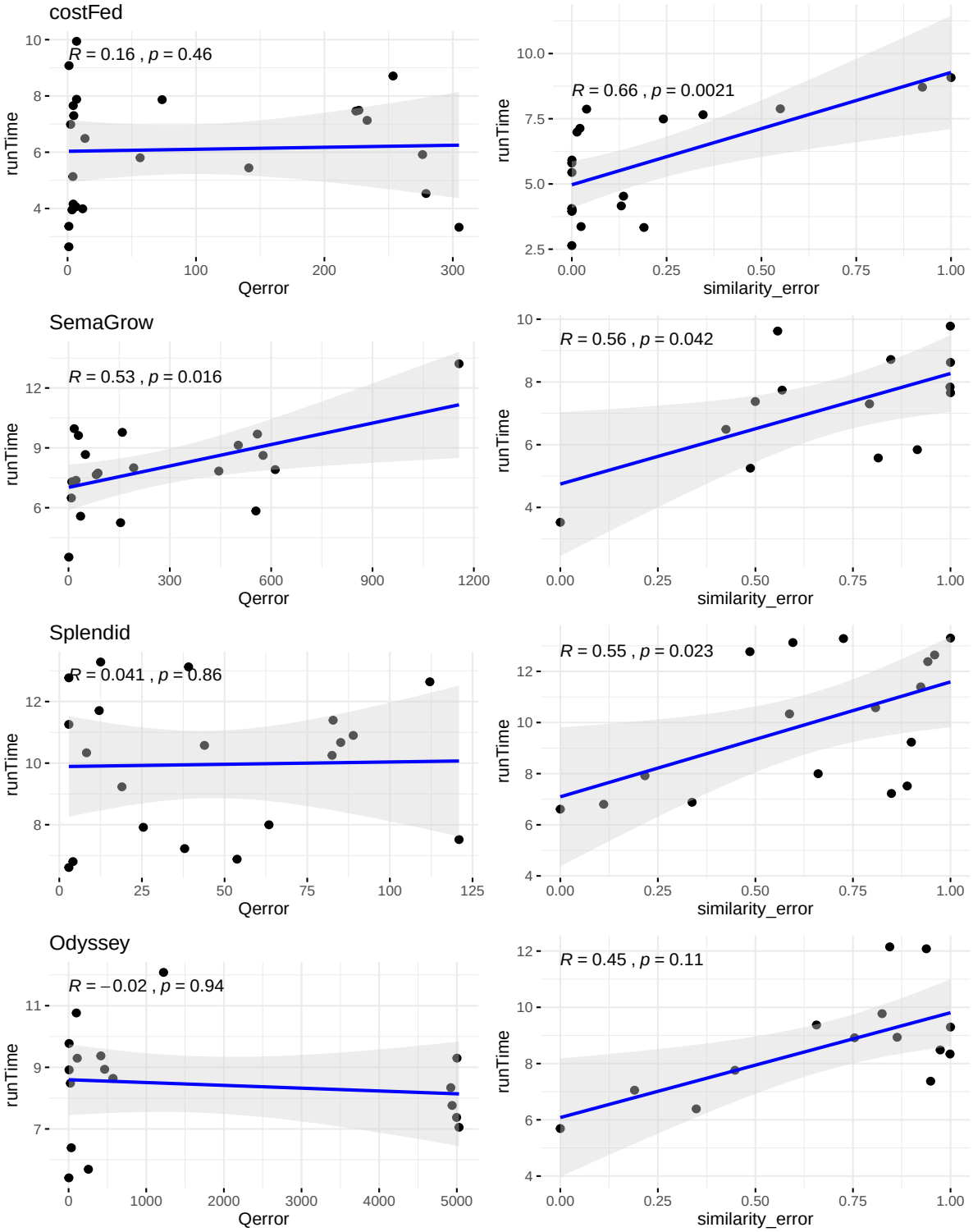


Fig. 3.: q-error and Similarity Error vs. runtime. (Robust Regression Analysis). The grey area in graph represents the confidence interval (band) in regression line.

	Rank	1	2	3		4	5	6	
		Similarity Error				q-error			
	Feature	E_J	E_P	E_T	Average	Q_J	Q_P	Q_T	Average
F.Q Engines	CostFed	0.23	0.59	0.43	0.42	0.14	0.26	0.1	0.17
	SemaGrow	0.33	0.33	0.33	0.33	0.47	0.37	0.001	0.28
	ODYSSEY	0.11	0.14	0.55	0.26	0.01	0.03	−0.06	−0.01
	SPLENDID	0.3	0.4	0.24	0.32	0.17	0.1	0.24	0.17
	LHD	0.16	0.28	−0.2	0.08	0.51	0.11	0.04	0.22
	Average	0.22	0.35	0.27	0.28	0.26	0.17	0.06	0.17

Table 2: Spearman's rank correlation coefficients between query plan features and query runtimes for all queries.

	Rank	1	2	3		4	5	6	
		Similarity Error				q-error			
	Feature	E_J	E_P	E_T	Average	Q_J	Q_P	Q_T	Average
F.Q Engines	CostFed	0.54	0.61	0.36	0.5	0.11	0.23	0.05	0.13
	SemaGrow	0.44	0.56	0.43	0.48	0.49	0.40	−0.02	0.29
	ODYSSEY	0.22	0.42	0.53	0.39	−0.04	−0.01	−0.20	−0.08
	SPLENDID	0.35	0.45	0.27	0.36	0.12	0.04	0.21	0.12
	Average	0.39	0.51	0.40	0.43	0.17	0.17	0.01	0.12

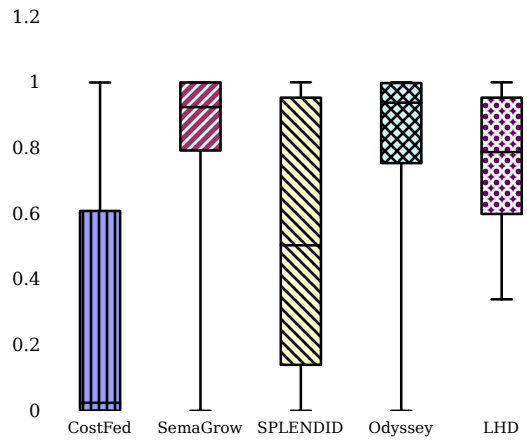
Table 3: Spearman's rank correlation coefficients between query plan features and query runtimes after linear regression (only for common queries between all systems).

	Rank	1	2	3		4	5	6	
		Similarity Error				q-error			
	Feature	E_J	E_P	E_T	Average	Q_J	Q_P	Q_T	Average
F.Q Engines	CostFed	0.60	0.66	0.62	0.63	0.16	0.16	0.16	0.16
	SemaGrow	0.56	0.56	0.57	0.56	0.60	0.53	0.57	0.56
	ODYSSEY	0.25	0.45	0.59	0.43	−0.04	−0.02	−0.20	−0.08
	SPLENDID	0.49	0.55	0.20	0.38	0.14	0.041	0.18	0.12
	Average	0.45	0.56	0.49	0.50	0.22	0.18	0.18	0.19

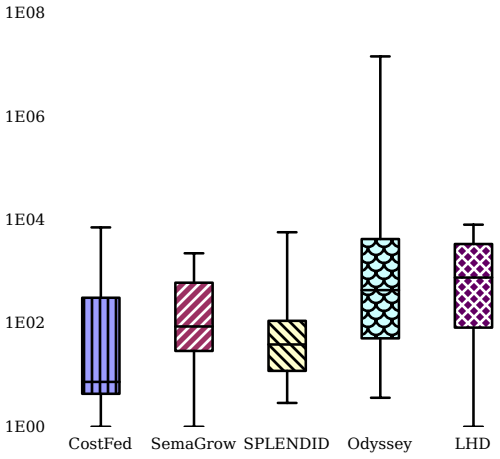
Table 4: Spearman's rank correlation coefficients between query plan features and query runtimes after robust regression (only for common queries between all systems). E_J: Similarity Error of Joins, E_P: Similarity Error of overall query plan, E_T: Similarity Error of Triple Patterns, Q_J: q-error of Joins, Q_P: q-error of overall query plan, Q_T: q-error Error of Triple Patterns, F.Q: Federated Query. Correlations and colors (−+): 0.00…0.19 very weak (●●), 0.20…0.39 weak (●●), 0.40…0.59 moderate (●●), 0.60…0.79 strong (●●), 0.80…1.00 very strong (●●).

An overall evaluation of the q-error of query plans given in Figure 4b leads to the following result: CostFed produces the smallest errors followed by SPLENDID, SemaGrow, Odyssey, and LHD, respectively. In particular, CostFed produces smaller errors than SPLENDID in 9/17 comparable queries (excluding queries with timeout and runtime error). SPLENDID produces smaller errors than SemaGrow in 9/17 comparable queries. SemaGrow produces smaller errors than Odyssey in 8/13 comparable queries. Odyssey is superior to LHD in 5/8 cases.

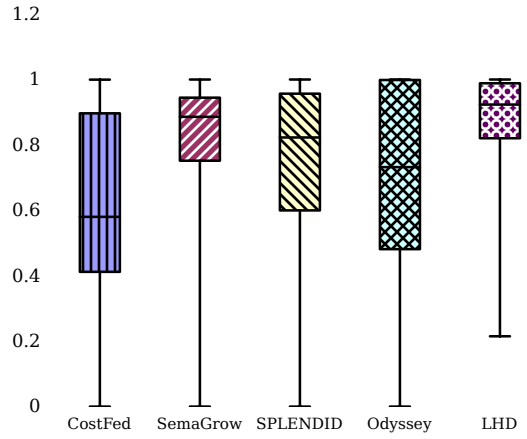
An overall evaluation of the similarity error in joins leads to a different picture (see Figure 4c). While CostFed remains the best system and produces the smallest errors, it is followed by Odyssey, SPLENDID, SemaGrow, and LHD, respectively. In particular, CostFed outperforms Odyssey in 12/17 comparable queries (excluding queries with timeout and runtime error). Odyssey produces less errors than SPLENDID



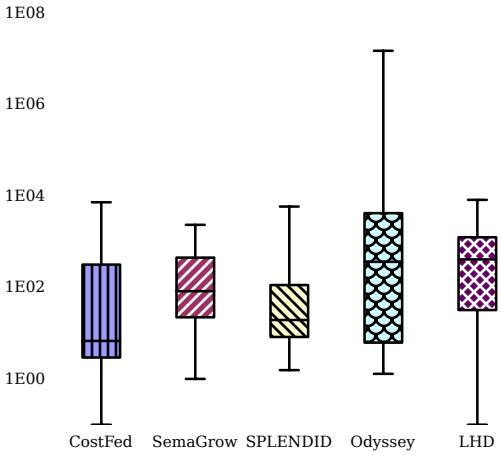
(a) Overall Similarity Error of query plans



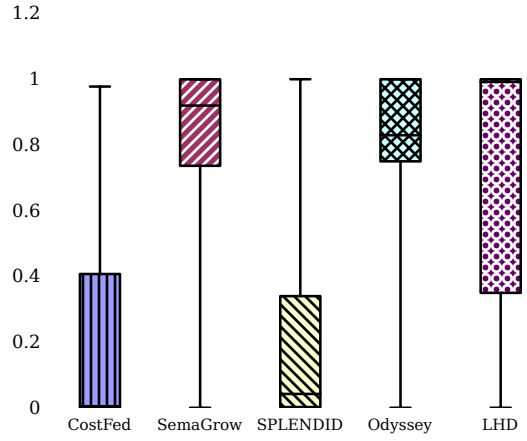
(b) Overall q-error of query plans



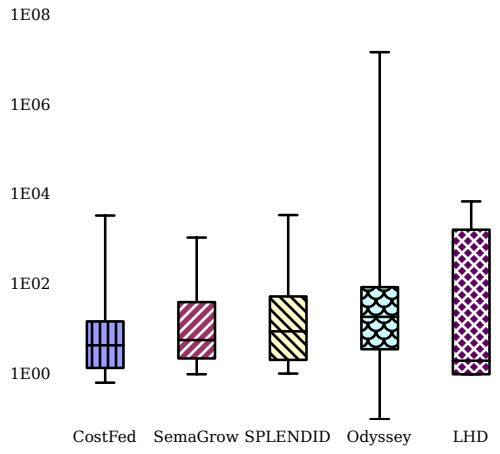
(c) Join Similarity Error of query plans



(d) Join q-error of query plan



(e) Triple pattern Similarity Error of query



(f) Triple pattern q-error of query

Fig. 4.: Similarity and q-error of Query plan

in 7/14 comparable queries. SPLENDID is superior to SemaGrow in 11/17 comparable queries. SemaGrow outperforms LHD in 7/12 comparable queries.

As an overall evaluation of the q-error of joins given in Figure 4d, CostFed produces the smallest errors followed by SPLENDID, SemaGrow, Odyssey, and LHD, respectively. CostFed produces less errors than SPLENDID in 12/17 comparable queries (excluding queries with timeout and runtime error). SPLENDID produces less errors than SemaGrow in 9/17 comparable queries. SemaGrow produces less errors than Odyssey in 9/13 comparable queries. Odyssey produces less errors than LHD in 4/8 comparable queries.

Overall, the evaluation of the similarity errors of triple patterns given in Figure 4e reveals that CostFed produces the smallest errors followed by SPLENDID, Odyssey, SemaGrow, and LHD, respectively. CostFed produces smaller errors than SPLENDID in 10/17 comparable queries (excluding queries with timeout and runtime error). SPLENDID produces smaller errors than Odyssey in 15/17 comparable queries. Odyssey produces smaller errors than SemaGrow in 7/14 comparable queries. SemaGrow outperformed LHD in 6/12 queries.

An overall evaluation of the q−error of triple patterns given in Figure 4f leads to a different ranking: CostFed produces the smallest errors followed by LHD, SemaGrow, SPLENDID, and Odyssey, respectively. CostFed outperforms LHD in 6/11 comparable queries (excluding queries with timeout and runtime error). LHD produces fewer errors than SemaGrow in 5/10 comparable queries. SemaGrow is better than SPLENDID in 10/17 comparable queries. SPLENDID produces fewer errors than Odyssey in 7/14 comparable queries.

In general, the accuracy of the estimation is dependent upon the detail of the statistics stored in the index or data summaries. Furthermore, it is important to pay special attention to the different types of triple patterns (with bound and unbound subject, predicate, objects) and joins types (subject-subject, subject-object, object-object) for the better cardinality estimations. CostFed is more accurate because of the more detailed data summaries, able to handle the different types of triple patterns and joins between triple patterns. The use of the buckets can more accurately estimate the cardinalities of the triple patterns with most common predicates used in the dataset. Furthermore, it handles multi-valued predicates. The Odyssey statistics are more detailed as compared to SPLENDID and SemaGrow (both using VoiD statistics). The distributed characteristic sets (CS) and characteristic pair (CP) statistics generally leads to better cardinality estimations for joins.

6.4. How Much Does An Efficient Cardinality Estimation Really Matter?

We observed that it is possible for a federation engine to produce quite a high cardinality estimation error (e.g., 0.99 is the overall similarity error for the S11 query in SemaGrow), yet it produces the optimal query plan. This leads to the question, how much does the efficiency of cardinality estimators of federation engines matter to generate optimal query plans? To this end we analyzed query plans generated by each of the selected engines for the benchmark queries. In our analysis, there are three possible cases in each plan:

- **Optimal plan**: In the *optimal* plan the best possible join order is selected *based on the given source selection performed by the underlying federation engine*, i.e., the least cardinality joins are always executed first.
- **Sub-optimal plan**: In the *sub-optimal* plan, the engine fails to select the best join *based on the given source selection performed by the underlying federation engine*, i.e., the least cardinality joins are not always executed first. Please note that this also means that the high error in the join cardinality estimation leads to the sub-optimal join order.
- **Only-plan**: In *only-plan* there is only one possible join order *according to the given source selection performed by the underlying federation engine*. This is possible if only 1 join (excluding a left-join due to the OPTIONAL clause in the query) needs to be executed locally by the federation engine. This situation occurs if there is only a single join in the query or the federation engine creates exclusive groups of joins that are executed remotely by the underlying SPARQL endpoints.

Table 5 shows the query plan generated by the query planners of the selected engines according to the aforementioned three cases possible for each plan. Since LHD failed to generate any query plan for the majority of the LargeRDFBench queries, we are skipping it from further discussion. In our evaluation, CostFed produced the smallest sub-optimal plans (i.e, 6) followed by Odyssey (i.e., 11), SemaGrow (i.e., 12), and SPLENDID (i.e, 14), respectively. The reason for CostFed's small number of sub-optimal plans is due the fact that it has the fewest cardinality errors in the estimation, as

	Query	CostFed	SemaGrow	SPLENDID	Odyssey	LHD
Simple Queries	S1	OnlyP	OnlyP	OnlyP	OnlyP	OnlyP
	S2	OnlyP	OnlyP	OnlyP	OnlyP	OptP
	S3	OnlyP	OnlyP	OnlyP	OnlyP	OptP
	S4	OnlyP	OptP	OnlyP	OnlyP	OptP
	S5	OnlyP	OptP	OnlyP	OnlyP	OptP
	S6	OptP	OptP	OptP	subOpt	OptP
	S7	OptP	OptP	OptP	subOpt	OptP
	S8	OnlyP	OnlyP	OnlyP	OnlyP	OnlyP
	S9	OnlyP	OnlyP	OnlyP	OnlyP	OnlyP
	S10	OnlyP	subOpt	OnlyP	OptP	subOpt
	S11	OnlyP	OnlyP	OnlyP	OnlyP	subOpt
	S12	OptP	OptP	OptP	OptP	subOpt
	S13	OnlyP	OptP	OptP	OptP	subOpt
	S14	OnlyP	OptP	OptP	OnlyP	OptP
Complex Queries	C1	OnlyP	OptP	OptP	OnlyP	Failed
	C2	subOpt	subOpt	subOpt	OptP	Failed
	C3	OptP	subOpt	OptP	OptP	Failed
	C4	OptP	subOpt	subOpt	subOpt	Failed
	C5	subOpt	OptP	subOpt	subOpt	Failed
	C6	OnlyP	OptP	subOpt	OnlyP	Failed
	C7	OnlyP	OptP	OptP	OnlyP	Failed
	C8	OnlyP	subOpt	subOpt	OnlyP	Failed
	C9	OnlyP	OptP	subOpt	OnlyP	Failed
	C10	OptP	OptP	subOpt	subOpt	Failed
Complex + High Data Sources Queries	CH1	OptP	subOpt	subOpt	OptP	Failed
	CH2	subOpt	subOpt	subOpt	subOpt	Failed
	CH3	OptP	subOpt	subOpt	subOpt	Failed
	CH4	subOpt	subOpt	subOpt	Failed	Failed
	CH5	Failed	subOpt	subOpt	subOpt	Failed
	CH6	Failed	Failed	OptP	subOpt	Failed
	CH7	subOpt	subOpt	subOpt	subOpt	Failed
	CH8	subOpt	subOpt	subOpt	subOpt	Failed

Table 5: Query Plans generated by query engines for all queries (Simple, Complex, Complex + High Dimensional Queries). **Failed:**(●) Engine Failed to produce Query Plan, **OptP:**(●) Optimal Query Plan generated by engine, **subOpt:**(●) subOptimal Plan generated by engine, **OnlyP:**(●) Only Plan possible.

discussed in the previous section. In addition, it generates the highest number of possible only-plans (which can be regarded as optimal plans for the given source selection information). This is because CostFed's source selection is more efficient in terms of the total triple pattern-wise sources selected without losing recall (ref. see Table 8).

In Table 5, we can see that only a few sub-optimal query plans were generated for simple queries. This is due to the fact that simple category queries of the LargeRDFBench contain very few joins (avg. 2.6, ref. [29]) to be executed by the federation engines. Thus, it is relatively easy to find the best join execution order. However, for complex, and complex-plus-high-data sources queries, more sub-optimal plans were generated. This is because these queries contain more joins (around 4 joins on avg., ref. [29]), hence a more accurate join cardinality estimation is required to generate the optimal join ordering plan. In conclusion, efficient cardinality estimation is more important for complex queries with more possible join ordering.

6.5. Number of Transferred Triples

Table 6 shows the number of tuples sent and received during the query execution for the selected federation engines. The number of sent tuples is related to the number of endpoint requests sent by the federation engine during query processing [7, 13]. The number of received tuples can be regarded as the number of intermediate results produced by the federation engine during query processing [13]. The smaller number of transferred tuples is considered important for fast query

Queries	CostFed		SemaGrow		Odyssey		LHD		SPLENDID	
	sent	received	sent	received	sent	received	sent	received	sent	received
S1	31	100	49	111	47	100	34	91	49	111
S2	12	11	21	12	11	11	10	3	21	12
S3	20	2	22	24	20	2	Failed	Failed	25	4
S4	12	1	46	17	12	1	20	18	15	3
S5	16	17	21	12	16	17	18	13444	18	20
S6	34	1616	TO	TO	6500	3254	283	1766	36	1618
S7	27	642	45	635	TO	TO	20	143	29	371
S8	10	1159	10	1159	10	1159	4	1159	10	1159
S9	25	351	59	382	25	351	34	342	59	382
S10	20	20054	36	14578	TO	TO	16	24540	21	20055
S11	12	13	14	15	12	13	19	4261	14	15
S12	2147	2136	7772	3428	2147	2136	Failed	Failed	7442	3428
S13	68	228	1161	10267	105	131	2456	13079	1161	10267
S14	2877	4033	3852	4366	2877	4033	97	2449	3852	4366
Avg	**379**	**2168**	**1008**	**2693**	**981**	**934**	**250**	**5108**	**910**	**2987**
C1	4234	2573	Failed	Failed	4234	2573	Failed	Failed	5232	4173
C2	1371	2118	131	1532	1352	1354	Failed	Failed	131	1532
C3	6213	13343	8670	15464	6213	13343	Failed	Failed	13854	9796
C4	26	550	TO	TO	11	1093	Failed	Failed	Failed	Failed
C5	TO	TO	Failed	Failed	2232	20532	Failed	Failed	TO	TO
C6	12	11432	TO	TO	12	11432	Failed	Failed	20	125310
C7	87	112	550	335	87	112	Failed	Failed	550	335
C8	622	3519	1365	4768	622	3519	Failed	Failed	1365	4768
C9	7274	21178	3358	10275	TO	TO	Failed	Failed	Failed	Failed
C10	51	5702	112	1541	Failed	Failed	Failed	Failed	979	1312
Avg	**2210**	**6725**	**2364**	**5652**	**1845**	**6745**	**NA**	**NA**	**3164**	**21032**
CH1	390	8253	1709	9439	TO	TO	Failed	Failed	Failed	Failed
CH2	TO	TO	TO	TO	TO	TO	Failed	Failed	Failed	Failed
CH3	167	5053	4686	4011	TO	TO	Failed	Failed	Failed	Failed
CH4	72	25	39	20	Failed	Failed	Failed	Failed	Failed	Failed
CH5	Failed	Failed	Failed	Failed	TO	TO	Failed	Failed	TO	TO
CH6	Failed	Failed	Failed	Failed	TO	TO	Failed	Failed	1551	9401
CH7	2332	85158	Failed	Failed	TO	TO	Failed	Failed	Failed	Failed
CH8	TO	TO	Failed	Failed	TO	TO	Failed	Failed	Failed	Failed
Avg	**740**	**24622**	**2145**	**4490**	**NA**	**NA**	**NA**	**NA**	**1551**	**9401**

Table 6: Number of transferred tuples. **NA:** "Not applicable". **Failed** means either "Runtime Error" or "Incomplete Results" and **TO:** "Timeout", which means Query Execution exceeds threshold value. "green color"(●) means lowest value among all systems, and "red color"(●) means highest value among all systems.

processing [13]. In this regard, CostFed ranked first (31 green boxes, i.e., it had the best results among the selected engines), followed by Odyssey with 24 green boxes, SemaGrow with 12 green boxes, LHD with 10 green boxes, and then SPLENDID with 9 green boxes.

In most queries, CostFed and Odyssey produced the only possible plans *only-plan*, which means only one (excluding the Left join for OPTIONAL SPARQL operator) was locally executed by the federation engine. Consequently, these engines transfer fewer tuples in comparison to other approaches. The largest difference

is observed for S13, where CostFed and Odyssey clearly outperform the other approaches, transferring 500 times fewer tuples. The Number of received tuples in LHD is significantly high in comparison to other approaches. This is because it does not produce normal tree-like query plans. Rather, LHD focuses on generating independent tasks that can be run in parallel. Therefore, independent tasks retrieve a lot of intermediate results, which need to be joined locally in order to get the final query resultset.

Another advantage that CostFed and Odyssey have over other approaches is their join-aware approach for triple pattern-wise sources selected (TPWSS). This join-aware nature of these engines saves many tuples from transferring due to less overestimation of sources. CostFed also performs better because it maintains cache for ask requests and saves many queries from sending to different sources. Another important factor that is worth mentioning here is that the number of transferred tuples does not consider the number of columns in the result set, but only counts the number of rows returned or sent to the endpoints. We also observed that in the case of an *only-plan* or an *optimal* plan, the number of received tuples is less compared to *sub-optimal* plans, clearly indicating that a smaller number tuples is key to fast query processing. The amalgamated average of all queries could also be misleading because in complex queries, there are more failed/timeout queries for some systems while producing answers in others. Therefore, we calculated the separate average for each category of queries i.e., simple, complex and complex and high data. From our analysis of results, it concludes that if an engine produces *optimal* or *only-plan*, the number of intermediate results also decreases.

6.6. Indexing and Source Selection Metrics

A smaller-sized index is essential for fast index lookup during source selection, but it can lack important information. In contrast, large index sizes provide slow index lookup and are hard to manage, but may lead to better cardinality estimations. To this end, it is important to compare the size of the indexes generated by the selected federation engines. Table 7 shows a comparison of the index/data summaries' construction time and the index size[4] of the selected state-of-the-art cost-based SPARQL federation approaches. SemaGrow, SPLENDID and LHD rely on VOID statistics with a size of 1 MB for the complete LargeRDFBench datasets of size 34.3GB. CostFed's index size is 10.5 MB while Odyssey's is 5.2 GB. The much bigger index size used by Odyssey might makes this approach less appropriate to be used for Big RDF datasets such as WikiData, Linked Geo Data etc. CostFed's index construction time is around 1hr and 6 mins for the complete LargeRDFBench datasets. SPLENDID, SemaGrow and LHD took 1hr and 50 mins to generate the index. The Index construction time for Odyssey was 86 hrs and 30 mins, which makes it difficult to use for Big datasets or datasets with frequent updates.

According to [29], the efficiency of source selection can be measured in terms of: (1) total number of triple pattern-wise sources selected (#T), (2) the number of SPARQL ASK requests sent to the endpoints (#A) during source selection, and (3) the source selection time. Table 8 shows a comparison of the source selection algorithms of the select triple stores across these metrics. As discussed previously, the smaller #T leads to better query plan generation [29]. The smaller #A leads to smaller source selection time, which in turn leads to smaller query execution time. In this regard, CostFed ranked first (83 green boxes, i.e., the best results among the selected engines), followed by Odyssey with 56 green boxes, LHD with 15 green boxes, SPLENDID with 10 green boxes, and then SemaGrow with 9 green boxes.

The approaches that perform a join-aware and hybrid (SPARQL + index) source selection lead to smaller #T [29]. Both Odyssey and Costfed perform join-aware source selection and hence lead to smaller #T than other selected approaches. The highest number of SPARQL ASK requests is sent by index-free federation engines, followed by hybrid (SPARQL + index), which in turn is followed by index-only federation engines [29]. This is because, for index-free federation engines such as FedX, the complete source selection is based on SPARQL ASK queries. The Hybrid engines such as CostFed, SPLENDID, SemaGrow and Odyssey make use of both index and SPARQL ASK queries to perform source selection, thus some of the SPARQL ASK requests are skipped due to the information used in the index. The Index-only engines, such as LHD, only make use of the index to perform the complete source selection. Thus, these engines do not consume a single SPARQL ASK query during source selection. The source selection time for such engines is much smaller due to only index-lookup without sending outside requests to endpoints. However, they have more #T than hybrid (SPARQL + index) source selection approaches.

[4]The index size is given by size of summaries used for cardinality estimation (in MBs).

	CostFed	SemaGrow	SPLENDID	Odyssey	LHD
Index Gen. Time(min)	65	110	110	533	110
Index Size(MBs)	10	1	1	5200	1

Table 7: Comparison of index construction time (**Index Gen. Time**) and **Index Size** for selected federation engines

6.7. Query Execution Time:

Finally, we present the query runtime results of the selected federation engines across the different queries categories of LargeRDFBench. Figure 5 gives an overview of our results. In our runtime evaluation on simple queries (S1-S14) (see Figure 5a), CostFed has the shortest runtimes, followed by SemaGrow, LHD, Odyssey, and SPLENDID, respectively. CostFed's runtimes are shorter than SemaGrow's on 13/13 comparable queries (excluding queries with timeout and runtime error) (average runtime = 0.5sec for CostFed vs. 2.5sec for SemaGrow). SemaGrow outperforms LHD on 4/11 comparable queries with an average runtime of 2.5sec for SemaGrow vs. 2.7sec for LHD. LHD's runtimes are shorter than Odyssey's on 8/10 comparable queries with an average runtime of 8.5sec for Odyssey. Finally, Odyssey is clearly faster than SPLENDID on 8/12 comparable queries with an average runtime of 131sec for SPLENDID.

Our runtime evaluation on the complex queries (C1-C10) (see Figure 5b) leads to a different ranking: CostFed produces the shortest runtimes followed by SemaGrow, Odyssey, and SPLENDID, respectively. CostFed outperforms SemaGrow in 6/6 comparable queries (excluding queries with timeout and runtime error) with an average runtime of 3sec for CostFed vs. 9sec for SemaGrow. SemaGrow's runtimes are shorter than Odyssey's in 3/4 comparable queries with an average runtime of 63sec for Odyssey. Odyssey is better than SPLENDID in 5/5 comparable queries, where SPLENDID's average runtime is 98sec.

The runtime evaluation on the complex and high sources queries (CH1-C8) given in Figure 5c establishes CostFed as the best query federation engine, followed by SPLENDID and then SemaGrow, respectively. CostFed's runtimes are smaller than SemaGrow in 3/3 comparable queries (excluding queries with timeout and runtime error), with an average runtime of 4sec for CostFed vs. 191sec for SemaGrow. SPLENDID has no comparable queries with CostFed and SemaGrow. LHD and Odyssey both fail to produce results when faced with complex queries.

7. Conclusion

In this paper, we presented an extensive evaluation of existing cost-based federated query engines. We used existing metrics from relational database research and proposed new metrics to measure the quality of cardinality estimators of selected engines. To the best of our knowledge, this work is the first evaluation of cost-based SPARQL federation engines focused on the quality of the cardinality estimations.

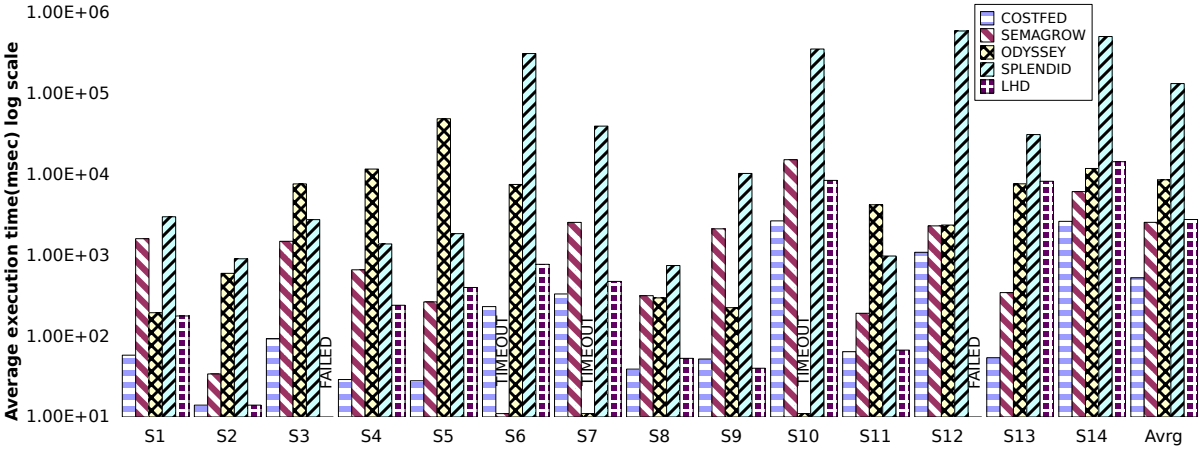
- The proposed similarity-based errors have a more positive correlation with runtimes, i.e., the smaller the error values, the better the query runtimes.
- The higher coefficients (R values) with similarity errors, (as opposed to q-error), suggest that the proposed similarity errors are a better predictor for runtime than the q-error.
- The smaller p-values of the similarity errors, as compared to q-error, further confirm that similarity errors are more likely to be a better predictor for runtime than the q-error.
- Errors in the cardinality estimation of triple patterns have a higher correlation to runtimes than the error in the cardinality estimation of joins.Thus, cost-based federation engines must pay particular attention to attaining accurate cardinality estimations of triple patterns
- The number of transferred tuples have a direct correlation with query runtime, i.e., the smaller the number of transferred tuples the smaller the query runtimes.
- The smaller number of triple pattern-wise sources selected is key to generate maximum only possible query plans (`only-plan`).
- On average, the CostFed engine produces the fewest estimation errors and has the shortest execution time for the majority of LargeRDFBench queries.
- The weak to moderate correlation of the cardinality errors with query execution time suggests that the query runtime is a complex measure affected by multi-dimensional performance metrics and SPARQL query features.

	Odyssey			SPLENDID			LHD			SemaGrow			CostFed		
Qry	#T	#A	ST	#T	#A	ST	#T	#A	ST	#T	#A	ST	#T	#A	ST
S1	11	0	1	11	26	293	28	0	261	11	26	293	4	18	6
S2	3	0	14	3	9	33	10	0	8	3	9	33	3	9	1
S3	5	0	44	12	2	17	20	0	34	12	2	17	5	0	1
S4	5	0	321	19	2	14	20	0	15	19	2	14	5	0	1
S5	4	0	223	11	1	11	11	0	8	11	1	11	4	0	1
S6	6	0	88	9	2	16	10	0	36	9	2	16	8	0	3
S7	6	0	72	13	2	19	13	0	67	13	2	19	6	0	1
S8	1	0	8	1	0	2	1	0	5	1	0	2	1	0	1
S9	4	0	1	11	26	200	28	0	69	11	26	200	4	18	5
S10	7	0	705	12	1	11	20	0	46	12	1	11	5	0	1
S11	7	0	30	7	2	19	15	0	12	7	2	19	7	0	1
S12	7	0	67	10	1	7	18	0	20	10	1	7	7	0	1
S13	10	0	23	9	2	8	17	0	58	9	2	8	5	0	1
S14	5	0	17	6	1	6	6	0	18	6	1	6	6	0	1
T/A	**81**	**0**	**115.3**	**134**	**77**	**46**	**217**	**0**	**47**	**134**	**77**	**46**	**70**	**45**	**1.7**
C1	8	0	38	11	1	11	RE	RE	RE	11	1	11	8	0	1
C2	8	0	44	11	1	7	RE	RE	RE	11	1	7	8	0	1
C3	30	0	121	21	3	12	RE	RE	RE	21	3	12	11	0	1
C4	12	0	24	28	0	3	RE	RE	RE	28	0	3	18	0	1
C5	16	0	320	33	0	3	RE	RE	RE	33	0	3	10	0	1
C6	9	0	311	24	0	2	RE	RE	RE	24	0	2	9	0	1
C7	9	0	38	17	2	9	RE	RE	RE	17	2	9	9	0	1
C8	11	0	27	25	2	11	RE	RE	RE	25	2	11	11	0	1
C9	19	0	452	16	2	17	RE	RE	RE	16	2	17	9	0	1
C10	12	0	142	13	0	3	RE	RE	RE	13	0	3	11	0	1
T/A	**134**	**0**	**151.7**	**199**	**11**	**7.8**	**NA**	**NA**	**NA**	**199**	**11**	**7.8**	**104**	**0**	**1**
CH1	22	0	333	41	48	62	RE	RE	RE	41	48	62	22	0	3
CH2	10	0	196	20	32	96	RE	RE	RE	20	32	96	10	0	5
CH3	18	0	544	37	37	604	RE	RE	RE	37	37	604	13	0	4
CH4	RE	RE	RE	18	28	25	RE	RE	RE	18	28	25	12	0	3
CH5	11	0	522	29	41	48	RE	RE	RE	29	41	48	RE	RE	RE
CH6	15	0	311	34	54	42	RE	RE	RE	RE	RE	RE	RE	RE	RE
CH7	26	0	337	47	65	57	RE	RE	RE	47	65	57	26	0	7
CH8	35	0	126	36	77	66	RE	RE	RE	36	77	66	35	0	6
T/A	**137**	**0**	**338.4**	**262**	**382**	**125**	**NA**	**NA**	**NA**	**228**	**328**	**136**	**98**	**0**	**4.6**

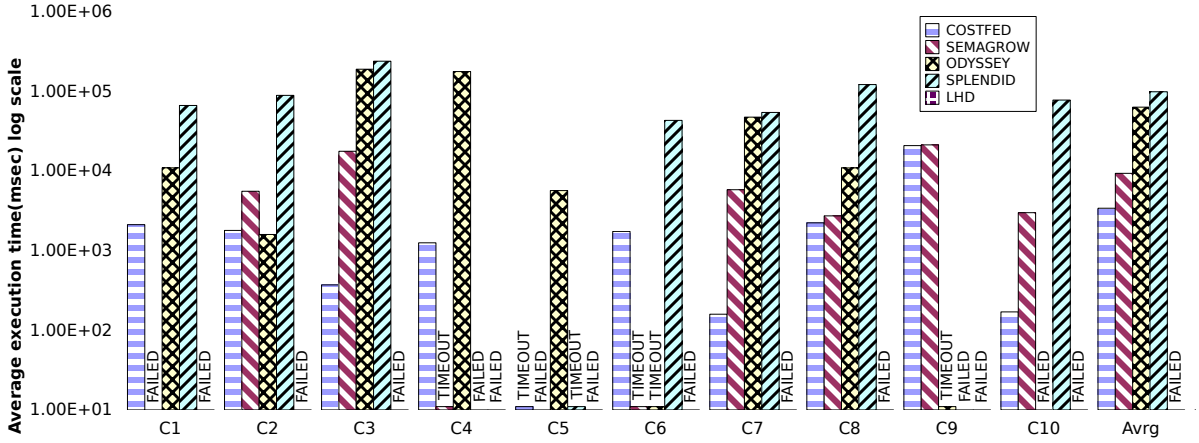
Table 8: Selected federation Engines comparison in terms of source selection time **ST** in msec, total number of SPARQL `ASK` requests **#A**, and total triple pattern-wise sources selected **#T**. (**RE** represents "Runtime Error",**TO** represents "Time Out" of 20 min, **T/A** represents "Total/Average" where Average is for ST, and Total is for #T and #A), **NA** represents "Not Applicable". "green color"(●) means lowest value among all systems, and "red color"(●) means highest value among all systems.

– The proposed cardinality estimating metrics are generic and can be applied to non-federated cardinality-based query processing engines as well.

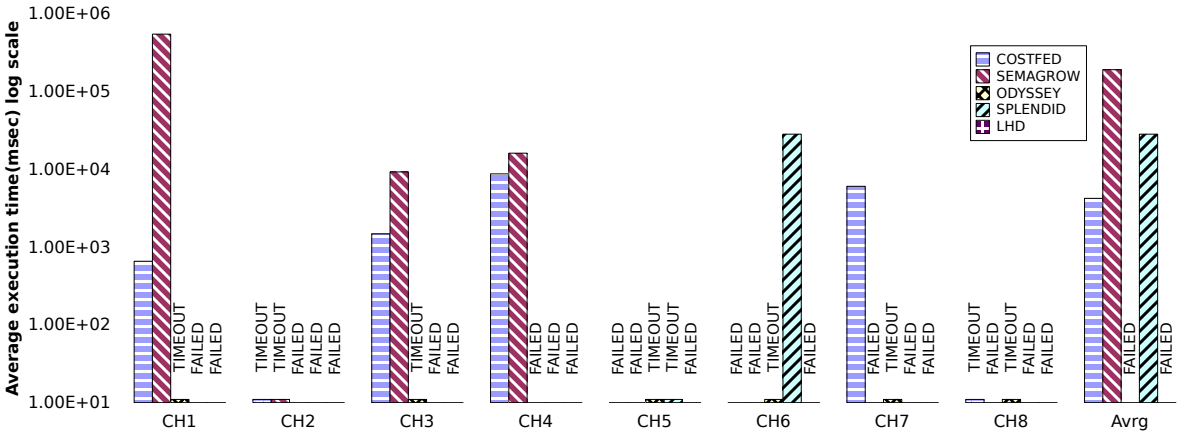
As future work, we want to compare heuristic-based (index-free) federated SPARQL query processing engines with cost-based federated engines. We want to investigate how much an index is assisting a cost-based



(a) Average execution time of simple (S) queries (FedBench)



(b) Average execution time of complex (C) queries (LargeRDFBench)



(c) Average execution time of complex and high data structures (ch) queries(LargeRDFBench)

Fig. 5.: Average execution time of LargeRDFBench and FedBench Queries.

federated SPARQL engine to generate optimized query execution plans.

Acknowledgments

This work is supported by the National Research Foundation of Korea(NRF) (grant funded by the Korea government(MSIT) (No. NRF-2018R1A2A2A05023669)). The work has also been supported by the project LIMBO (Grant no. 19F2029I), OPAL (no. 19F2028A), KnowGraphs (no. 860801), and SOLIDE (no. 13N14456) conducted in the University of Leipzig.

References

[1] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov and A.-C. Ngonga Ngomo, A fine-grained evaluation of SPARQL endpoint federation systems, *Semantic Web Journal* **7**(5) (2016), 493–518. doi:10.3233/SW-150186.

[2] N.A. Rakhmawati, J. Umbrich, M. Karnstedt, A. Hasnain and M. Hausenblas, Querying over Federated SPARQL Endpoints - A State of the Art Survey, *CoRR* **abs/1306.1723** (2013). http://arxiv.org/abs/1306.1723.

[3] M. Wylot, M. Hauswirth, P. Cudré-Mauroux and S. Sakr, RDF Data Storage and Query Processing Schemes: A Survey, *ACM Comput. Surv.* **51**(4) (2018), 84:1–84:36. doi:10.1145/3177850.

[4] D. Kossmann, The State of the Art in Distributed Query Processing, *ACM Comput. Surv.* **32**(4) (2000), 422–469–. doi:10.1145/371578.371598.

[5] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper and T. Neumann, How Good Are Query Optimizers, Really?, *Proc. VLDB Endow.* **9**(3) (2015), 204–215. doi:10.14778/2850583.2850594.

[6] G. Moerkotte, T. Neumann and G. Steidl, Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors, *Proc. VLDB Endow.* **2**(1) (2009), 982–993. doi:10.14778/1687627.1687738.

[7] A. Schwarte, P. Haase, K. Hose, R. Schenkel and M. Schmidt, FedX: Optimization Techniques for Federated Query Processing on Linked Data, in: *The Semantic Web – ISWC 2011*, L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. Noy and E. Blomqvist, eds, Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 601–616. ISBN 978-3-642-25073-6. doi:10.1007/978-3-642-25073-6_38.

[8] G. Montoya, M.-E. Vidal and M. Acosta, A Heuristic-based Approach for Planning Federated SPARQL Queries, in: *Proceedings of the Third International Conference on Consuming Linked Data - Volume 905*, COLD'12, CEUR-WS.org, Aachen, Germany, Germany, 2012, pp. 63–74. http://dl.acm.org/citation.cfm?id=2887367.2887373.

[9] O. Hartig, C. Bizer and J.-C. Freytag, Executing SPARQL Queries over the Web of Linked Data, in: *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 293–309–. ISBN 9783642049293. doi:10.1007/978-3-642-04930-9_19.

[10] M. Saleem, A. Potocki, T. Soru, O. Hartig and A.-C.N. Ngomo, CostFed: Cost-Based Query Optimization for SPARQL Endpoint Federation, Elsevier, 2018, pp. 163–174, Proceedings of the 14th International Conference on Semantic Systems 10th – 13th of September 2018 Vienna, Austria. ISSN 1877-0509. doi:10.1016/j.procs.2018.09.016.

[11] X. Wang, T. Tiropanis and H. Davis, LHD Optimising Linked Data Query Processing Using Parallelisation, in: *Workshop on Linked Data on the Web (LDOẂ13), Proceedings of the WWW2013*, CEUR Workshop Proceedings, Vol. 996, CEUR-WS.org, Rio de Janeiro, Brazil, 2013. ISSN 1613-0073. http://eprints.soton.ac.uk/350719/.

[12] B. Quilitz and U. Leser, Querying Distributed RDF Data Sources with SPARQL, in: *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 524–538. ISBN 3-540-68233-3, 978-3-540-68233-2. http://dl.acm.org/citation.cfm?id=1789394.1789443.

[13] G. Montoya, H. Skaf-Molli and K. Hose, The Odyssey Approach for Optimizing Federated SPARQL Queries, *The Semantic Web – ISWC 2017* (2017), 471–489–. ISBN 9783319682884. doi:10.1007/978-3-319-68288-4_28.

[14] O. Görlitz and S. Staab, SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions, in: *Proceedings of the Second International Conference on Consuming Linked Data - Volume 782*, COLD'11, CEUR-WS.org, Aachen, Germany, Germany, 2010, pp. 13–24. http://dl.acm.org/citation.cfm?id=2887352.2887354.

[15] A. Charalambidis, A. Troumpoukis and S. Konstantopoulos, SemaGrow: Optimizing Federated SPARQL Queries, in: *Proceedings of the 11th International Conference on Semantic Systems*, SEMANTICS '15, ACM, New York, NY, USA, 2015, pp. 121–128. ISBN 978-1-4503-3462-4. doi:10.1145/2814864.2814886.

[16] A. Hasnain, R. Fox, S. Decker and H.F. Deus, Cataloguing and Linking Life Sciences LOD Cloud, in: *1st International Workshop on Ontology Engineering in a Data-driven World (OEDW 2012) collocated with 8th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2012)*, 2012, pp. 114–130.

[17] A. Hasnain, S. Sana e Zainab, M. Kamdar, Q. Mehmood, J. Warren ClaudeN., Q. Fatimah, H. Deus, M. Mehdi and S. Decker, A Roadmap for Navigating the Life Sciences Linked Open Data Cloud, in: *Semantic Technology*, T. Supnithi, T. Yamaguchi, J.Z. Pan, V. Wuwongse and M. Buranarach, eds, Lecture Notes in Computer Science, Vol. 8943, Springer International Publishing, 2015, pp. 97–112. ISBN 978-3-319-15614-9. doi:10.1007/978-3-319-15615-6_8.

[18] G. Ladwig and T. Tran, SIHJoin: Querying Remote and Local Linked Data, in: *The Semantic Web: Research and Applications*, G. Antoniou, M. Grobelnik, E. Simperl, B. Parsia, D. Plexousakis, P. De Leenheer and J. Pan, eds, Lecture Notes in Computer Science, Vol. 6643, Springer Berlin Heidelberg, 2011, pp. 139–153. ISBN 978-3-642-21033-4. doi:10.1007/978-3-642-21034-1_10. http://dx.doi.org/10.1007/978-3-642-21034-1_10.

[19] S. Lynden, I. Kojima, A. Matono and Y. Tanimura, ADERIS: An Adaptive Query Processor for Joining Federated SPARQL Endpoints, in: *R. Meersman, T. Dillon, P. Herrero, A. Kumar, M. Reichert, L. Qing, B.-C. Ooi, E. Damiani, D.C. Schmidt, J. White, M. Hauswirth, P. Hitzler, M. Mohania, editors, On*

the Move to Meaningful Internet Systems (OTM2011), Part II. LNCS, Vol. 7045, Springer Heidelberg, 2011, pp. 808–817.

[20] F. Du, Y. Chen and X. Du, Partitioned Indexes for Entity Search over RDF Knowledge Bases, in: *Proceedings of the 17th International Conference on Database Systems for Advanced Applications - Volume Part I*, DASFAA'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 141–155–. ISBN 9783642290374. doi:10.1007/978-3-642-29038-1_12.

[21] I. Abdelaziz, E. Mansour, M. Ouzzani, A. Aboulnaga and P. Kalnis, Lusail: A System for Querying Linked Data at Scale, *Proceedings of the VLDB Endowment* **11**(4) (2017), 485–498. doi:10.1145/3186728.3164144.

[22] K.M. Endris, M. Galkin, I. Lytra, M.N. Mami, M.-E. Vidal and S. Auer, MULDER: Querying the Linked Data Web by Bridging RDF Molecule Templates, in: *Database and Expert Systems Applications (DEXA'17)*, D. Benslimane, E. Damiani, W.I. Grosky, A. Hameurlain, A. Sheth and R.R. Wagner, eds, Springer International Publishing, Cham, 2017, pp. 3–18. ISBN 978-3-319-64468-4. doi:10.1007/978-3-319-64468-4_1.

[23] M. Saleem and A.-C. Ngonga Ngomo, HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation, in: *The Semantic Web: Trends and Challenges*, V. Presutti, C. d'Amato, F. Gandon, M. d'Aquin, S. Staab and A. Tordai, eds, Lecture Notes in Computer Science, Vol. 8465, Springer International Publishing, 2014, pp. 176–191. ISBN 978-3-319-07442-9. doi:10.1007/978-3-319-07443-6_13.

[24] M. Saleem, A.-C. Ngonga Ngomo, J. Xavier Parreira, H.F. Deus and M. Hauswirth, DAW: Duplicate-AWare Federated Query Processing over the Web of Data, in: *Proceedings of the 12th International Semantic Web Conference - Part I*, Lecture Notes in Computer Science, Springer-Verlag New York, Inc., New York, NY, USA, 2013, pp. 574–590. ISBN 978-3-642-41334-6. doi:10.1007/978-3-642-41335-3_36.

[25] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo and E. Ruckhaus, ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints, in: *The Semantic Web – ISWC 2011*, L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. Noy and E. Blomqvist, eds, Lecture Notes in Computer Science, Vol. 7031, Springer-Verlag Berlin Heidelberg, 2011, pp. 18–34. ISBN 978-3-642-25072-9. doi:10.1007/978-3-642-25073-6_2.

[26] J. Umbrich, A. Hogan, A. Polleres and S. Decker, Link Traversal Querying for a Diverse Web of Data, *Semantic Web Journal* **6**(6) (2015), 585–624. doi:10.3233/SW-140164.

[27] M. Acosta, M.-E. Vidal and Y. Sure-Vetter, Diefficiency Metrics: Measuring the Continuous Efficiency of Query Processing Approaches, in: *The Semantic Web – ISWC 2017*, C. d'Amato, M. Fernandez, V. Tamma, F. Lecue, P. Cudré-Mauroux, J. Sequeda, C. Lange and J. Heflin, eds, Springer-Verlag Berlin Heidelberg, Cham, 2017, pp. 3–19. ISBN 978-3-319-68204-4.

[28] A. Hasnain, Q. Mehmood, S. Sana E Zainab, M. Saleem, C. Warren Jr, D. Zehra, S. Decker and D. Rebholz-Schuhman, BioFed: Federated query processing over life sciences linked open data, *Journal of Biomedical Semantics* **8**(1) (2017), 13. doi:10.1186/s13326-017-0118-0.

[29] M. Saleem, A. Hasnain and A.-C. Ngonga Ngomo, LargeRDFBench: A Billion Triples Benchmark for SPARQL Endpoint Federation, *Journal of Web Semantics* **48** (2018), 85–125. doi:10.1016/j.websem.2017.12.005.

[30] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte and T. Tran, FedBench: A Benchmark Suite for Federated Semantic Data Query Processing, in: *The Semantic Web – ISWC 2011*, L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. Noy and E. Blomqvist, eds, Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 585–600. ISBN 978-3-642-25073-6. doi:10.1007/978-3-642-25073-6_37.

[31] O. Görlitz, M. Thimm and S. Staab, SPLODGE: Systematic Generation of SPARQL Benchmark Queries for Linked Open Data, in: *Proceedings of the 11th International Conference on The Semantic Web - Volume Part I*, The Semantic Web – ISWC'12, Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 116–132. ISBN 978-3-642-35175-4. doi:10.1007/978-3-642-35176-1_8.

[32] M. Morsey, J. Lehmann, S. Auer and A.-C.N. Ngomo, DBpedia SPARQL Benchmark: Performance Assessment with Real Queries on Real Data, in: *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, The Semantic Web – ISWC'11, Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 454–469. ISBN 978-3-642-25072-9. doi:10.1007/978-3-642-25073-6_29.

[33] N.A. Rakhmawati, M. Saleem, S. Lalithsena and S. Decker, QFed: Query Set For Federated SPARQL Query Benchmark, in: *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '14, ACM, New York, NY, USA, 2014, pp. 207–211. ISBN 978-1-4503-3001-5. doi:10.1145/2684200.2684321.

[34] A. Hasnain, M. Saleem, A.N. Ngomo and D. Rebholz-Schuhmann, Extending LargeRDFBench for Multi-Source Data at Scale for SPARQL Endpoint Federation, in: *Proceedings of the 12th International Workshop on Scalable Semantic Web Knowledge Base Systems co-located with 17th International Semantic Web Conference, SSWS@ISWC 2018, Monterey, California, USA, October 9, 2018*, Vol. 2179, 2018, pp. 28–44. http://ceur-ws.org/Vol-2179/SSWS2018_paper3.pdf.

[35] G. Montoya, M.-E. Vidal, O. Corcho, E. Ruckhaus and C. Buil-Aranda, Benchmarking Federated SPARQL Query Engines: Are Existing Testbeds Enough?, in: *P. Cudre Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J.X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, E. Blomqvist, editors, The Semantic Web – ISWC 2012, Part II. LNCS*, Vol. 7650, Springer-Verlag Berlin Heidelberg, 2012, pp. 313–324.

[36] C. Bizer and A. Schultz, The Berlin SPARQL Benchmark, in: *International Journal on Semantic Web and Information Systems (IJSWIS)*, Vol. 5, IGI Global, 2009, pp. 1–24.

[37] M. Schmidt, T. Hornung, G. Lausen and C. Pinkel, SPˆ 2Bench: A SPARQL Performance Benchmark, in: *Proceedings of the 25th International Conference on Data Engineering ICDE*, IEEE, 2009, pp. 222–233.

[38] C. Buil-Aranda, A. Hogan, J. Umbrich and P.-Y. Vandenbussche, SPARQL Web-Querying Infrastructure: Ready for Action?, in: *The Semantic Web – ISWC 2013*, H. Alani, L. Kagal, A. Fokoue, P. Groth, C. Biemann, J.X. Parreira, L. Aroyo, N. Noy, C. Welty and K. Janowicz, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 277–293. ISBN 978-3-642-41338-4.

[39] K. Alexander, L. Talis, Cyganiak, M. Hausenblas and J. Zhao, Describing Linked Datasets-On the Design and Usage of voiD, the'Vocabulary of Interlinked Datasets', *Linked Data on the Web Workshop (LDOW 09), in conjunction with 18th International World Wide Web Conference (WWW 09)* **538** (2010), 13.

[40] T. Neumann and G. Moerkotte, Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins, in: *Proceedings of the 2011 IEEE 27th International Conference on*

Data Engineering, IEEE, 2011, pp. 984–994, IEEE Computer Society. ISSN 1063-6382. doi:10.1109/ICDE.2011.5767868.

[41] A. Gubichev and T. Neumann, Exploiting the query structure for efficient join ordering in SPARQL queries., in: *EDBT*, Vol. 14, 2014, pp. 439–450.

[42] P.W. Holland and R.E. Welsch, Robust regression using iteratively reweighted least-squares, *Communications in Statistics - Theory and Methods* **6**(9) (1977), 813–827. doi:10.1080/03610927708827533.

[43] D.P. O'Leary, Robust Regression Computation Using Iteratively Reweighted Least Squares, *SIAM J. Matrix Anal. Appl.* **11**(3) (1990), 466–480. doi:10.1137/0611032.

[44] P.J. Rousseeuw and A.M. Leroy, *Robust regression and outlier detection*, Vol. 589, 1st edn, John wiley & sons, Inc., New York, NY, USA, 1987. ISBN 0-471-85233-3. doi:10.1002/0471725382.

[45] P.J. Huber, *Robust Estimation of a Location Parameter*, in: *Breakthroughs in Statistics: Methodology and Distribution*, S. Kotz and N.L. Johnson, eds, Springer New York, New York, NY, 1992, pp. 492–518. ISBN 978-1-4612-4380-9. doi:10.1007/978-1-4612-4380-9_35.

[46] M. Saleem, G. Szárnyas, F. Conrads, S.A.C. Bukhari, Q. Mehmood and A.-C. Ngonga Ngomo, How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks, in: *The World Wide Web Conference*, WWW '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1623–1633–. ISBN 9781450366748. doi:10.1145/3308558.3313556.