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

Umair Qudus, Muhammad Saleem, Axel-Cyrille Ngonga Ngomo and Young-koo Lee

DKE, Kyung Hee University, South Korea
E-mail: {umair.qudus,yklee}@khu.ac.kr

AKSW, Leipzig, Germany
E-mail: {lastname}@informatik.uni-leipzig.de

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. Albeit informative, these metrics are generic and unable to quantify and evaluate the accuracy of the cardinality estimators of cost-based federation engines. 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 five cost-based federated SPARQL query engines using existing as well as novel evaluation metrics by using LargeRDFBench queries. Our results provide a detailed analysis of the experimental outcomes that reveal novel insights, useful for the development of future cost-based federated SPARQL query processing 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 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 addition, we compared state-of-the-art cost-based SPARQL federation engines using existing metrics pertaining to indexing, query processing, network, and overall query runtime using different evaluation setups.

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], Odyssey[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.

\(^1\)Our proposed metric is open-source and available online at https://github.com/dice-group/CostBased-FedEval
An Empirical Evaluation of Cost-based Federated SPARQL Query Processing Engines

<table>
<thead>
<tr>
<th></th>
<th>Index Processing</th>
<th>Network</th>
<th>Res</th>
<th>RS</th>
<th>Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cr</td>
<td>Gt</td>
<td>Qp</td>
<td>#Ts</td>
<td>Qet</td>
<td>#A</td>
</tr>
<tr>
<td>CostFed[10]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SPLENDID[14]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SemaGrow[15]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Odyssey[13]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LHD[11]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DARQ[12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ANAPSID[25]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MULDER[22]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FedX[7]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lusail[21]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BioFed[28]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

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...
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

As aforementioned, most cost-based SPARQL federation engines first estimate individual triple pattern cardinality and use this information to estimate 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 \(\vec{r} = (r_1, \ldots, r_n) \in \mathbb{R}^n\) where \(r_i > 0\) be a vector of real values and \(\vec{e} = (e_1, \ldots, e_n) \in \mathbb{R}^n\) be the vector of the corresponding estimated values. By defining \(\vec{e}/\vec{r} = (e_1/r_1, \ldots, e_n/r_n)\), then q-error of estimation \(e\) of \(r\) is given as
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 \( \vec{r} = (Cr(TP_1), \ldots, Cr(TP_n)) \) ∈ \( \mathbb{R}^n \) be the vector of real cardinalities of \( T \) and \( \vec{e} = (Ce(TP_1), \ldots, Ce(TP_n)) \) ∈ \( \mathbb{R}^n \) 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 \( J = \{J_1, \ldots, J_m\} \). Let \( \vec{r} = (Cr(J_1), \ldots, Cr(J_m)) \) ∈ \( \mathbb{R}^m \) a vector of real cardinalities of \( J \) and \( \vec{e} = (Ce(J_1), \ldots, Ce(J_m)) \) ∈ \( \mathbb{R}^m \) 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 \( \vec{r} = (r_1, \ldots, r_n) \) ∈ \( \mathbb{R}^n \) be a vector of real cardinalities of \( TJ \) and \( \vec{e} = (e_1, \ldots, e_n) \) ∈ \( \mathbb{R}^n \) 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 (\( \sim \) [7, 14, 15]) and symmetric hash join (\( \sim \) [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. The CostFed query planner also considers the skew in the distribution of objects and subjects across predicates. Separate cardi-
nality 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 dataset 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 object selectivity of \(p\) in \(D\).
- \(IT(D)\) is the total number of triples in \(D\).
- \(IS(D)\) is the total number of distinct subjects in \(D\).
- \(IO(D)\) 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):

\[
C(tp) = \begin{cases}
\sum_{D \in R(tp)} T(p, D_i) \times 1 & \text{if } b(p) \land b(s) \land b(o), \\
\sum_{D \in R(tp)} T(p, D_i) \times \text{avgSS}(p, D_i) & \text{if } b(p) \land b(s) \land !b(o), \\
\sum_{D \in R(tp)} T(p, D_i) \times \text{avgOS}(p, D_i) & \text{if } b(p) \land b(o) \land !b(s), \\
\sum_{D \in R(tp)} IT(D_i) \times 1 & \text{if } !b(p) \land b(s) \land b(o), \\
\sum_{D \in R(tp)} IT(D_i) \times \frac{1}{\text{avgSS}(D_i)} & \text{if } b(p) \land b(s) \land !b(o), \\
\sum_{D \in R(tp)} IT(D_i) \times \frac{1}{\text{avgOS}(D_i)} & \text{if } !b(p) \land b(s) \land b(o), \\
\sum_{D \in R(tp)} IT(D_i) \times \frac{1}{\text{avgSS}(D_i) \times \text{avgOS}(D_i)} & \text{if } !b(p) \land b(s) \land b(o), \\
1 & \text{other}
\end{cases}
\]

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 \(E_1\) and \(E_2\) are SPARQL expressions then \(E_1 \bowtie E_2\) is also a SPARQL expression. The join cardinality of two expressions \(E_1\) and \(E_2\) is estimated as follows:

\[
C(E_1 \bowtie E_2) = M(E_1) \times M(E_2) \times \text{Min}(C(E_1), C(E_2))
\]

where the average frequency of multi-valued predicates in the expression \(E\) is defined as \(C(E)\). 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:

\[
M(E) = \begin{cases}
\frac{1}{\sqrt{2}} & \text{if } b(p) \land b(s) \land b(o), \\
C(E)/\text{distS}(bjs(p, D)) & \text{if } b(p) \land b(s) \land !b(o) \land j(s), \\
C(E)/\text{distOb}(js(p, D)) & \text{if } b(p) \land b(o) \land !b(s) \land j(o), \\
1 & \text{other}
\end{cases}
\]

If subject of the triple pattern is involved in the join, it is defined as \(j(s)\), and \(b(s), b(o), \text{ and } b(p)\) 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. SPLENDID makes use of both hash(\(\sigma_d\)) and bind(\(\alpha_d\)) joins.

Triple pattern cardinality is estimated as follows:

\[
\text{card}_d(?, p, ?) = \text{card}_d(p)
\]

\[
\text{card}_d(s, ?, ?) = |d| \cdot \text{sel} \cdot s_d
\]

\[
\text{card}_d(s, p, ?) = \text{card}_d(p) \cdot \text{sel} \cdot s_d(p)
\]

\[
\text{card}_d(?, ?, o) = |d| \cdot \text{sel} \cdot o_d
\]

\[
\text{card}_d(s, p, o) = \text{card}_d(p) \cdot \text{sel} \cdot o_d(p)
\]

\[
\text{card}_d(s, ?, o) = |d| \cdot \text{sel} \cdot s_d \cdot \text{sel} \cdot o_d
\]

where the \(\text{card}_d(p)\) 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 \(\text{sel} \cdot s_d(p)\) and \(\text{sel} \cdot o_d(p)\) respectively; if the predicate is not bound then the average selectivity of subject and object is defined as \(\text{sel} \cdot s_d\) and \(\text{sel} \cdot o_d\) 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:

\[
\text{card}_d(T) = \min (\text{card}_d(T_{\text{bound}})) \cdot \prod \left( \text{sel}_{sd} \cdot \text{card}_d(T_{\text{unbound}}) \right)
\]

Join cardinality is estimated as follows.

\[
\text{card}(q_1 \bowtie q_2) = \text{card}(q_1) \cdot \text{card}(q_2) \cdot \text{sel}_w(q_1, q_2)
\]

In these equations the \( \text{sel}_w \) is the join selectivity of two input relations. It defines 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 \( I_{dp} \), distinct subjects \( s_{dp} \) and distinct objects \( o_{dp} \) 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 \( T : \{ SPO \} \), 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 \( \text{sel}_T(x) \), and the cardinality of \( x \) with respect to \( S(T) \) is defined as \( \text{card}_T(x) \).

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

\[
\text{sel}_T(S) = \frac{\sum_{d \in S(T)} I_d}{\sum_{d \in S(T)} I_d} \cdot \frac{\sum_{d \in S(T)} I_{dp}}{\sum_{d \in S(T)} I_{dp}} \cdot \frac{\sum_{d \in S(T)} I_{dp}}{\sum_{d \in S(T)} I_{dp}}
\]

If \( \text{var}(P) \land \neg \text{var}(S) \) and \( \text{var}(P) \land \neg \text{var}(O) \), then:

\[
\text{sel}_T(P) = \begin{cases} 
\frac{\sum_{d \in S(T)} I_d/|P|}{\sum_{d \in S(T)} I_d} & \text{if } P = P \land \neg \text{var}(S) \\
1 & \text{if } \text{var}(S) \\
1 & \text{if } \text{var}(O) 
\end{cases}
\]

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 \( T_1 \) and \( T_2 \), LHD calculates the join selectivity by using the following equations:

\[
\text{sel} (T_1 \bowtie T_2) = \begin{cases} 
\frac{\sum_{d \in S(T_1)} I_d \cdot \sum_{d \in S(T_2)} I_d}{\sum_{d \in S(T_1)} I_d} & \text{if joined on } S_1 = S_2 \\
\frac{\sum_{d \in S(T_1)} I_d \cdot \sum_{d \in S(T_2)} I_d}{\sum_{d \in S(T_1)} I_d} & \text{if joined on } O_1 = O_2 \\
\frac{\sum_{d \in S(T_1)} I_d \cdot \sum_{d \in S(T_2)} I_d}{\sum_{d \in S(T_1)} I_d} & \text{if joined on } S_1 = O_2 \\
1 & \text{if no shared variables.}
\end{cases}
\]

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

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

**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.

---

\(^3\)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 ([E] ⊲ [E2]), which is given as follows:

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

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., \((\text{count}(C))\) represents the number of entities sharing C and \((\text{occurrences}(p, C))\) represents the number of triples with predicate p occurring with these entities.

Odyssey represents estimatedCardinality \(_{\text{distinct}}(P)\) 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:

\[
\text{estimatedCardinality}_{\text{distinct}}(P) = \sum_{P \subseteq C_i} (\text{count}(C_j))
\]

\[
\text{estimatedCardinality} (P) = \sum_{P \subseteq C_j} \left( \text{count}(C_j) \cdot \prod_{p \in P} \frac{\text{occurrences}(p, C_j)}{\text{count}(C_j)} \right)
\]

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 CSs \(C_i\) and \(C_j\), through a property p, is represented in statistics, which is defined as: \(- \text{count}(C_i, C_j, p)\). 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:

\[
\text{estimatedCardinality} (P_k, P_l, p) = \sum_{P_i \subseteq C_i \land P_j \subseteq C_j} \left( \text{count}(C_i, C_j, p) \cdot \prod_{p_i \in P_l - \{p\}} \left( \frac{\text{occurrences}(p_i, C_j)}{\text{count}(C_j)} \right) \right)
\]

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], Odysseus [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
runtime 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 run-
times, 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_I 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 sys-
tem 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 Spearman’s
correlation again on common queries. Table 3 shows
that the proposed metric has a stronger positive correla-
tion with query runtime when we deal with only com-
mon queries. The similarity error of overall plan (E_P)
and triple pattern (E_I) 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 met-
rics have a stronger positive correlation. However, in
our proposed metric this difference is definite. The simi-
ilarity 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_I 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_T with 0.18), respectively.
Table 4 also shows that the q-error for Odyssey is neg-
atively 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 normal-
ity assumptions. By comparing the p-values (at 5% con-
fidence 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 similar-
ity 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 gen-
eration. Thus, it is important for federation engines to
pay particular attention to the cardinality estimation of
the overall query plan. However, given that this estima-
tion commonly depends on triple patterns and join esti-
mations, 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]. The
proposed similarity error metric is related to the query
planning component of the federation engines and is
useful to evaluate the quality of the query plans gener-
ated by these engines.

6.2.1. Outlier Analysis

In the robust regression model, the outliers are ad-
justed with new values according to Huber loss func-
tion. In similarity error, the list of queries which are
re-weighted after applying robust regression are: C2,
C1, S14, CH7 in CostFed; S2 in SemaGrow; S8 and
S2 in SPLENDID; and S8 in Odyssey. In q-error, the
list of queries which are re-weighted after applying ro-
 bust regression are: C6, C2, C4, CH7, S3 in CostFed;
CH3, CH4, S13, C2 in SemaGrow; CH6, C2, C7, S5 in
SPLENDID; and S11, C2, C1, S4 in Odyssey.
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.
Table 2: Spearman’s rank correlation coefficients between query plan features and query runtimes for all queries.

<table>
<thead>
<tr>
<th>Rank</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>q-error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature</td>
<td>( E_I )</td>
<td>( E_P )</td>
<td>( E_T )</td>
<td>Average</td>
<td>( Q_I )</td>
<td>( Q_P )</td>
<td>( Q_T )</td>
</tr>
<tr>
<td>FQ Engines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CostFed</td>
<td>0.23</td>
<td>0.59</td>
<td>0.43</td>
<td>0.42</td>
<td>0.14</td>
<td>0.26</td>
<td>0.1</td>
</tr>
<tr>
<td>SemaGrow</td>
<td>0.33</td>
<td>0.53</td>
<td>0.33</td>
<td>0.33</td>
<td>0.47</td>
<td>0.37</td>
<td>0.001</td>
</tr>
<tr>
<td>ODYSSEY</td>
<td>0.11</td>
<td>0.14</td>
<td>0.55</td>
<td>0.26</td>
<td>0.01</td>
<td>0.03</td>
<td>-0.06</td>
</tr>
<tr>
<td>SPLENDID</td>
<td>0.35</td>
<td>0.4</td>
<td>0.24</td>
<td>0.32</td>
<td>0.17</td>
<td>0.1</td>
<td>0.24</td>
</tr>
<tr>
<td>LHD</td>
<td>0.16</td>
<td>0.28</td>
<td>-0.2</td>
<td>0.08</td>
<td>0.54</td>
<td>0.11</td>
<td>0.04</td>
</tr>
<tr>
<td>Average</td>
<td>0.22</td>
<td>0.35</td>
<td>0.27</td>
<td>0.28</td>
<td>0.26</td>
<td>0.17</td>
<td>0.06</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Rank</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>q-error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature</td>
<td>( E_I )</td>
<td>( E_P )</td>
<td>( E_T )</td>
<td>Average</td>
<td>( Q_I )</td>
<td>( Q_P )</td>
<td>( Q_T )</td>
</tr>
<tr>
<td>FQ Engines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CostFed</td>
<td>0.54</td>
<td>0.61</td>
<td>0.36</td>
<td>0.36</td>
<td>0.5</td>
<td>0.11</td>
<td>0.23</td>
</tr>
<tr>
<td>SemaGrow</td>
<td>0.44</td>
<td>0.56</td>
<td>0.43</td>
<td>0.48</td>
<td>0.49</td>
<td>0.4</td>
<td>-0.02</td>
</tr>
<tr>
<td>ODYSSEY</td>
<td>0.22</td>
<td>0.42</td>
<td>0.53</td>
<td>0.39</td>
<td>-0.04</td>
<td>-0.01</td>
<td>-0.20</td>
</tr>
<tr>
<td>SPLENDID</td>
<td>0.35</td>
<td>0.45</td>
<td>0.27</td>
<td>0.36</td>
<td>0.12</td>
<td>0.04</td>
<td>0.21</td>
</tr>
<tr>
<td>Average</td>
<td>0.39</td>
<td>0.51</td>
<td>0.40</td>
<td>0.43</td>
<td>0.17</td>
<td>0.17</td>
<td>0.01</td>
</tr>
</tbody>
</table>

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, \( Q_I \): q-error of Joins, \( Q_P \): q-error of overall query plan, \( Q_T \): q-error Error of Triple Patterns, FQ: 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 (●●●).

In these queries the residual values are either significantly higher or lower than the regression line. For example, in CostFed the average of the similarity errors across all queries is 0.272 and the range of the residual values for unmodified queries is between -0.17 to 0.17, while C2 similarity error is 0.99 with residual value 0.73, CH7 similarity error is 0.99 with residual value 0.19, and C1 similarity error is 0.62 with residual value 0.32. Hence, by re-weighting these queries, the similarity error and q-error values are re-adjusted close to the regression line, in order to get a more clear and concise picture of the regression experiments. As it can be observed from the Simple Linear Regression figure, the outliers are influencing the results. For example, in CostFed, the R value in simple regression is 0.59 and after re-weighting in robust regression, the value is increased to 0.66. Furthermore, we observe that, in similarity error the robust regression increases the R value which is generally decreased in q-error, furthering
An Empirical Evaluation of Cost-based Federated SPARQL Query Processing Engines

Fig. 4.: Similarity and q-error of Query plan
suggesting that similarity error is the better predictor of the runtime as compared to q-error.

Finally, the overall q-error is more affected by robust regression as compared to similarity error. This is because a q-error takes the maximum of all the errors in the cardinality estimation of the joins and triple patterns. Consequently, some queries produce very high q-error values due to a single less efficient cardinality estimation for a join or a triple pattern.

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 SPLENID, LHD, SemaGrow, and Odyssey, respectively. CostFed produces smaller errors than SPLENID in 10/17 comparable queries (excluding queries with timeout and runtime errors). SPLENID 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.

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 SPLENID, SemaGrow, Odyssey, and LHD, respectively. In particular, CostFed produces smaller errors than SPLENID in 9/17 comparable queries (excluding queries with timeout and runtime error). SPLENID produces smaller errors than SemaGrow in 9/17 comparable queries. SemaGrow produces smaller errors than Odyssey in 9/15 comparable queries.

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, SPLENID, 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 SPLENID in 7/14 comparable queries. SPLENID 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 SPLENID, SemaGrow, Odyssey, and LHD, respectively. CostFed produces less errors than SPLENID in 12/17 comparable queries (excluding queries with timeout and runtime error). SPLENID 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 SPLENID, Odyssey, SemaGrow, and LHD, respectively. CostFed produces smaller errors than SPLENID in 10/17 comparable queries (excluding queries with timeout and runtime error). SPLENID 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, SPLENID, 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 SPLENID in 10/17 comparable queries. SPLENID 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 SPLENID 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.,
<table>
<thead>
<tr>
<th>Query</th>
<th>CostFed</th>
<th>Semagrow</th>
<th>SPLENDID</th>
<th>Odyssey</th>
<th>LHD</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
</tr>
<tr>
<td>S2</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OptP</td>
</tr>
<tr>
<td>S3</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OptP</td>
</tr>
<tr>
<td>S4</td>
<td>OnlyP</td>
<td>OptP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OptP</td>
</tr>
<tr>
<td>S5</td>
<td>OnlyP</td>
<td>OptP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OptP</td>
</tr>
<tr>
<td>S6</td>
<td>OptP</td>
<td>OptP</td>
<td>OptP</td>
<td>subOpt</td>
<td>OptP</td>
</tr>
<tr>
<td>S7</td>
<td>OptP</td>
<td>OptP</td>
<td>OptP</td>
<td>subOpt</td>
<td>OptP</td>
</tr>
<tr>
<td>S8</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
</tr>
<tr>
<td>S9</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
</tr>
<tr>
<td>S10</td>
<td>OnlyP</td>
<td>subOpt</td>
<td>OnlyP</td>
<td>OptP</td>
<td>subOpt</td>
</tr>
<tr>
<td>S11</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>OnlyP</td>
<td>subOpt</td>
</tr>
<tr>
<td>S12</td>
<td>OptP</td>
<td>OptP</td>
<td>OptP</td>
<td>OptP</td>
<td>subOpt</td>
</tr>
<tr>
<td>S13</td>
<td>OnlyP</td>
<td>OptP</td>
<td>OptP</td>
<td>OptP</td>
<td>subOpt</td>
</tr>
<tr>
<td>S14</td>
<td>OnlyP</td>
<td>OptP</td>
<td>OptP</td>
<td>OnlyP</td>
<td>OptP</td>
</tr>
</tbody>
</table>

**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.

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 discussed in the previous section. In addition, it generates the highest number of optimal 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 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 (i.e., the number of projection variables in the query) in the result set, but only counts the number of rows (i.e., the number of results) 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
An Empirical Evaluation of Cost-based Federated SPARQL Query Processing Engines

<table>
<thead>
<tr>
<th>Queries</th>
<th>CostFed</th>
<th>SemaGrow</th>
<th>Odyssey</th>
<th>LHD</th>
<th>SPLENDID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sent</td>
<td>received</td>
<td>sent</td>
<td>received</td>
<td>sent</td>
</tr>
<tr>
<td>S1</td>
<td>31</td>
<td>100</td>
<td>49</td>
<td>111</td>
<td>47</td>
</tr>
<tr>
<td>S2</td>
<td>12</td>
<td>11</td>
<td>21</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>S3</td>
<td>20</td>
<td>2</td>
<td>22</td>
<td>24</td>
<td>20</td>
</tr>
<tr>
<td>S4</td>
<td>12</td>
<td>1</td>
<td>46</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>S5</td>
<td>16</td>
<td>17</td>
<td>21</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>S6</td>
<td>34</td>
<td>1616</td>
<td>TO</td>
<td>TO</td>
<td>6500</td>
</tr>
<tr>
<td>S7</td>
<td>27</td>
<td>642</td>
<td>45</td>
<td>635</td>
<td>TO</td>
</tr>
<tr>
<td>S8</td>
<td>10</td>
<td>1199</td>
<td>10</td>
<td>1159</td>
<td>10</td>
</tr>
<tr>
<td>S9</td>
<td>25</td>
<td>351</td>
<td>59</td>
<td>382</td>
<td>25</td>
</tr>
<tr>
<td>S10</td>
<td>20</td>
<td>20054</td>
<td>36</td>
<td>14578</td>
<td>TO</td>
</tr>
<tr>
<td>S11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>S12</td>
<td>2147</td>
<td>2136</td>
<td>7772</td>
<td>3428</td>
<td>2147</td>
</tr>
<tr>
<td>S13</td>
<td>68</td>
<td>228</td>
<td>1161</td>
<td>10267</td>
<td>105</td>
</tr>
<tr>
<td>S14</td>
<td>2877</td>
<td>4033</td>
<td>3852</td>
<td>4366</td>
<td>2877</td>
</tr>
<tr>
<td>Avg</td>
<td>379</td>
<td>2168</td>
<td>1008</td>
<td>2693</td>
<td>934</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queries</th>
<th>CostFed</th>
<th>SemaGrow</th>
<th>Odyssey</th>
<th>LHD</th>
<th>SPLENDID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sent</td>
<td>received</td>
<td>sent</td>
<td>received</td>
<td>sent</td>
</tr>
<tr>
<td>C1</td>
<td>4234</td>
<td>2573</td>
<td>Failed</td>
<td>Failed</td>
<td>4234</td>
</tr>
<tr>
<td>C2</td>
<td>1371</td>
<td>2118</td>
<td>131</td>
<td>1532</td>
<td>1352</td>
</tr>
<tr>
<td>C3</td>
<td>6213</td>
<td>13343</td>
<td>8670</td>
<td>15464</td>
<td>6213</td>
</tr>
<tr>
<td>C4</td>
<td>26</td>
<td>3550</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>C5</td>
<td>TO</td>
<td>TO</td>
<td>Failed</td>
<td>Failed</td>
<td>2232</td>
</tr>
<tr>
<td>C6</td>
<td>12</td>
<td>11432</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>C7</td>
<td>87</td>
<td>112</td>
<td>550</td>
<td>335</td>
<td>87</td>
</tr>
<tr>
<td>C8</td>
<td>622</td>
<td>3519</td>
<td>1365</td>
<td>4768</td>
<td>622</td>
</tr>
<tr>
<td>C9</td>
<td>7274</td>
<td>21178</td>
<td>3358</td>
<td>10275</td>
<td>TO</td>
</tr>
<tr>
<td>C10</td>
<td>51</td>
<td>5702</td>
<td>112</td>
<td>1541</td>
<td>Failed</td>
</tr>
<tr>
<td>Avg</td>
<td>2210</td>
<td>6725</td>
<td>2364</td>
<td>5652</td>
<td>1845</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Queries</th>
<th>CostFed</th>
<th>SemaGrow</th>
<th>Odyssey</th>
<th>LHD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sent</td>
<td>received</td>
<td>sent</td>
<td>received</td>
</tr>
<tr>
<td>CH1</td>
<td>390</td>
<td>8255</td>
<td>1709</td>
<td>9439</td>
</tr>
<tr>
<td>CH2</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>CH3</td>
<td>167</td>
<td>5053</td>
<td>4686</td>
<td>4011</td>
</tr>
<tr>
<td>CH4</td>
<td>72</td>
<td>25</td>
<td>39</td>
<td>20</td>
</tr>
<tr>
<td>CH5</td>
<td>Failed</td>
<td>Failed</td>
<td>Failed</td>
<td>Failed</td>
</tr>
<tr>
<td>CH6</td>
<td>Failed</td>
<td>Failed</td>
<td>Failed</td>
<td>Failed</td>
</tr>
<tr>
<td>CH7</td>
<td>2332</td>
<td>85158</td>
<td>Failed</td>
<td>Failed</td>
</tr>
<tr>
<td>CH8</td>
<td>TO</td>
<td>Failed</td>
<td>Failed</td>
<td>Failed</td>
</tr>
<tr>
<td>Avg</td>
<td>740</td>
<td>24622</td>
<td>2145</td>
<td>4490</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Queries</th>
<th>CostFed</th>
<th>SemaGrow</th>
<th>SPLENDID</th>
<th>Odyssey</th>
<th>LHD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Index Gen. Time(min)</td>
<td>65</td>
<td>110</td>
<td>110</td>
<td>533</td>
</tr>
<tr>
<td></td>
<td>Index Size(MBs)</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>5200</td>
</tr>
</tbody>
</table>

Table 8: Comparison of index construction time (**Index Gen. Time**) and **Index Size** for selected federation engines.
<table>
<thead>
<tr>
<th>Qry</th>
<th>Odyssey #T</th>
<th>Odyssey #A</th>
<th>Odyssey ST</th>
<th>SPLENDID #T</th>
<th>SPLENDID #A</th>
<th>SPLENDID ST</th>
<th>LHD #T</th>
<th>LHD #A</th>
<th>LHD ST</th>
<th>SemaGrow #T</th>
<th>SemaGrow #A</th>
<th>SemaGrow ST</th>
<th>CostFed #T</th>
<th>CostFed #A</th>
<th>CostFed ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>11</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>26</td>
<td>293</td>
<td>28</td>
<td>0</td>
<td>261</td>
<td>11</td>
<td>26</td>
<td>293</td>
<td>4</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>S2</td>
<td>3</td>
<td>0</td>
<td>14</td>
<td>3</td>
<td>9</td>
<td>33</td>
<td>10</td>
<td>0</td>
<td>8</td>
<td>3</td>
<td>9</td>
<td>33</td>
<td>3</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>5</td>
<td>0</td>
<td>44</td>
<td>12</td>
<td>2</td>
<td>17</td>
<td>20</td>
<td>0</td>
<td>34</td>
<td>12</td>
<td>2</td>
<td>17</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S4</td>
<td>5</td>
<td>0</td>
<td>321</td>
<td>19</td>
<td>2</td>
<td>14</td>
<td>20</td>
<td>0</td>
<td>15</td>
<td>19</td>
<td>2</td>
<td>14</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S5</td>
<td>4</td>
<td>0</td>
<td>223</td>
<td>11</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>0</td>
<td>8</td>
<td>11</td>
<td>1</td>
<td>11</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S6</td>
<td>6</td>
<td>0</td>
<td>88</td>
<td>9</td>
<td>2</td>
<td>16</td>
<td>10</td>
<td>0</td>
<td>36</td>
<td>9</td>
<td>2</td>
<td>16</td>
<td>8</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>S7</td>
<td>6</td>
<td>0</td>
<td>72</td>
<td>13</td>
<td>2</td>
<td>19</td>
<td>13</td>
<td>0</td>
<td>67</td>
<td>13</td>
<td>2</td>
<td>19</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S8</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S9</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>26</td>
<td>200</td>
<td>28</td>
<td>0</td>
<td>69</td>
<td>11</td>
<td>26</td>
<td>200</td>
<td>4</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>S10</td>
<td>7</td>
<td>0</td>
<td>705</td>
<td>12</td>
<td>1</td>
<td>11</td>
<td>20</td>
<td>0</td>
<td>46</td>
<td>12</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S11</td>
<td>7</td>
<td>0</td>
<td>30</td>
<td>7</td>
<td>2</td>
<td>19</td>
<td>15</td>
<td>0</td>
<td>12</td>
<td>7</td>
<td>2</td>
<td>19</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S12</td>
<td>7</td>
<td>0</td>
<td>67</td>
<td>10</td>
<td>1</td>
<td>7</td>
<td>18</td>
<td>0</td>
<td>20</td>
<td>10</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S13</td>
<td>10</td>
<td>0</td>
<td>23</td>
<td>9</td>
<td>2</td>
<td>8</td>
<td>17</td>
<td>0</td>
<td>58</td>
<td>9</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S14</td>
<td>5</td>
<td>0</td>
<td>17</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>18</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T/A</td>
<td>81</td>
<td>0</td>
<td>115.3</td>
<td>134</td>
<td>77</td>
<td>46</td>
<td>217</td>
<td>0</td>
<td>47</td>
<td>134</td>
<td>77</td>
<td>46</td>
<td>70</td>
<td>45</td>
<td>1.7</td>
</tr>
</tbody>
</table>

| C1  | 8          | 0          | 38         | 11          | 1           | 11         | 11      | 0     | 11     | 11         | 1           | 11          | 8        | 0        | 1        |
| C2  | 8          | 0          | 44         | 11          | 1           | 7          | 7       | 0     | 8      | 11         | 1           | 7           | 8        | 0        | 1        |
| C3  | 30         | 0          | 121        | 21          | 3           | 12         | 12      | 0     | 12     | 3          | 12          | 12          | 3        | 12       | 0        |
| C4  | 12         | 0          | 24         | 28          | 0           | 3          | 3       | 0     | 18     | 3          | 3           | 18          | 3        | 18       | 0        |
| C5  | 16         | 0          | 320        | 33          | 0           | 3          | 3       | 0     | 3      | 3          | 3           | 3           | 3        | 3        | 0        |
| C6  | 9          | 0          | 311        | 24          | 0           | 2          | 2       | 0     | 2      | 2          | 2           | 2           | 2        | 2        | 0        |
| C7  | 9          | 0          | 38         | 17          | 2           | 9          | 9       | 0     | 9      | 9          | 9           | 9           | 9        | 9        | 0        |
| C8  | 11         | 0          | 27         | 25          | 2           | 11         | 11      | 0     | 11     | 11         | 11          | 11          | 11       | 11       | 0        |
| C9  | 19         | 0          | 452        | 16          | 2           | 17         | 17      | 0     | 17     | 17         | 17          | 17          | 17       | 17       | 0        |
| C10 | 12         | 0          | 142        | 13          | 0           | 3          | 3       | 0     | 11     | 13         | 0           | 11          | 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 | 18         | 0          | 18         | 28          | 25          | RE         | 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     | 34         | 54          | 42          | 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 index size is given by size of summaries used for cardinality estimation (in MBs).
MB while Odyssey’s is 5.2 GB. The much bigger index size used by Odyssey might make 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.

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

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

(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)
The impact of our proposed work is to provide new measures for the development of better cost-based federated SPARQL query engines. Furthermore, our proposed metrics will help in determining the quality of the generated query plans, such as, it tells if the join orders are correct or not. This kind of information is not revealed from the query runtime because the overall query runtime is affected by all metrics given in Table 1. 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 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 Korean government(MSIT) (No. NRF-2018R1A2A2A05023669)). The work has also been supported by the project LIMBO (Grant no. 19F2029), OPAL (no. 19F2028A), KnowGraphs (no. 860801), and SOLIDE (no. 13N14456) conducted in the University of Leipzig.

References


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


and

A. Hasnain, M. Saleem, A.-C. Ngonga Ngomo

LargeRDFBench: A Billion Triples Benchmark for SPARQL Endpoint Federation


M. Schmidt, O. Görzli, P. Haase, A. Schwarte and M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte and

A. Hasnain, Q. Mehmood, A. C. Bukhari, Q. Mehmood and A.-C. Ngonga Ngomo


doi:10.1007/978-3-642-25073-6_29.


M. Schmidt, O. Görzli, P. Haase, A. Schwarte and


doi:10.1007/978-3-642-25073-6_29.


