Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 137 (2018) 163–174

Procedia Computer Science

www.elsevier.com/locate/procedia

SEMANTiCS 2018 – 14th International Conference on Semantic Systems

CostFed: Cost-Based Query Optimization for SPARQL Endpoint Federation

Muhammad Saleem, Alexander Potocki, Tommaso Soru, Olaf Hartig, Axel-Cyrille Ngonga Ngomo

Abstract

The runtime optimization of federated SPARQL query engines is of central importance to ensure the usability of the Web of Data in real-world applications. The efficient selection of sources (SPARQL endpoints in our case) as well as the generation of optimized query plans belong to the most important optimization steps in this respect. This paper presents CostFed, an index-assisted federation engine for federated SPARQL query processing. CostFed makes use of statistical information collected from endpoints to perform efficient source selection and cost-based query planning. In contrast to the state of the art, it relies on a non-linear model for the estimation of the selectivity of joins. Therewith, it is able to generate better plans than the state-of-the-art federation engines. Our experiments on the FedBench benchmark shows that CostFed is 3 to 121 times faster than the current federation engines.

1. Introduction

Two challenges must be addressed when optimizing federated query processing. The first is the *generation of efficient query plans*: For a given query, there are most likely several possible plans that a federation system may consider executing to gather results. These plans have different costs in terms of the amount of resources they necessitate and the overall time necessary to execute them. Detecting the most cost-efficient query plan for a given query is hence one of the key challenges in federated query processing. In addition, for a given query, an *optimized selection of sources* is one of the key steps towards the generation of efficient query plans [11]. A poor source selection can lead to increases of the overall query processing time [11].

Current cardinality/cost-based SPARQL endpoint federation approaches [2, 5] address the first challenge by making use of average selectivities to estimate the cardinality of triple patterns. Hence, they assume that the resources pertaining to a predicate are uniformly distributed. However, previous work [4] shows that real RDF datasets do not actually abide by uniform frequency distributions[1]. An example of the skew in the distribution of subject frequencies for the predicate `foaf:name` in DBpedia2015-10 is shown in Figure 1b. The assumption of a uniform distribution of subjects or objects across predicates can lead to a poor estimation of the cardinality of triple patterns when a high-frequency resource (i.e., a resource that occurs in a large number of triples) is used in that triple pattern. Consequently, the query

[1] Our analysis of FedBench confirms that the resources in FedBench are not uniformly distributed. See `https://goo.gl/1vBwt7`

10.1016/j.procs.2018.09.016

planning can be significantly affected as suggested by our evaluation (see Section 7). To address the second challenge, most SPARQL query federation approaches [2, 5, 14, 12] rely on a *triple pattern-wise source selection* (TPWSS) to optimize their source selection. The goal of the TPWSS is to identify the set of sources that are relevant for each individual triple pattern of a query [12]. However, it is possible that a relevant source does not *contribute* (formally defined in section 5.1) to the final result set of a query. This is because the results from a particular data source can be excluded after performing *joins* with the results of other triple patterns contained in the same query. The *join-aware TPWSS* strategy has been shown to yield great improvement [1, 11].

In this work, we present CostFed,[2] an index-assisted SPARQL endpoint federation engine. CostFed addresses the two challenges aforementioned: CostFed's *query planning* is based on estimating query costs by using selectivity information stored in an index. In contrast to the state of the art, CostFed takes the skew in distribution of subjects and objects across predicates into account. In addition, CostFed includes a *trie-based source selection* approach which is a *join-aware approach to TPWSS* based on common URI prefixes. Overall, our contributions are as follows: (1) We present a source selection algorithm based on labelled hypergraphs [11], which makes use of *data summaries* for SPARQL endpoints based on most common prefixes for URIs. We devise a pruning algorithm that allows discarding irrelevant sources based on common prefixes used in joins. (2) We propose a *cost-based query planning approach* which makes use of cardinality estimations for (a) triple patterns as well as for (b) joins between triple patterns. We considered the skew in resource frequency distribution by creating resource buckets (ref. see figure 1b) with different cardinality estimation functions. Our join implementation is based on both bind [2, 14, 5] and symmetric hash joins[3]. (3) We compare CostFed with the state-of-the-art federation engines ANAPSID [1], SemaGrow [2], SPLENDID [5], HiBISCuS [11], and FedX [14]. Our results show that we outperform these engines by (a) reducing the number of sources selected (without losing recall) and by (b) reducing the source selection time as well as the overall query runtime on the majority of the FedBench [13] queries. Our results on the more complex queries from LargeRDFBench [9] confirm the results on FedBench.

2. Related Work

FedX [14], SPLENDID [5], ANAPSID [1], SemaGrow [2], ODYSSEY [6], LUSAIL [7] etc. are examples of state-of-the-art SPARQL federation engines. A more exhaustive overview of these systems can be found in [10]. However, they do not consider the skew distribution of subjects and objects across predicates. CostFed is most closely related to HiBISCuS [11] in terms of source selection. HiBISCuS makes use of the different *URIs authorities*[3] to prune irrelevant sources during the source selection. While HiBISCuS can significantly remove irrelevant sources [11], it fails to prune those sources which share the same URI authority. For example, all the `Bio2RDF` sources contains the same URI authority. We address the drawback of HiBISCuS by proposing a *trie-based source selection approach*. By moving away from authorities, CostFed is flexible enough to distinguish between URIs from different datasets that come from the same namespace (e.g., as in `Bio2RDF`). In addition, we propose a *cost-based query planner* that takes into account the skew in frequency distribution of subject or object resources pertaining to a predicate.

3. Preliminaries

To denote the set of solution mappings that is defined as the result of a SPARQL query Q over an RDF graph G we write $[\![Q]\!]_G$. As the basis of our query federation scenario, we assume that the federation consists of SPARQL endpoints. Formally, we capture this federation as a finite set $\mathcal{D}$ whose elements denote SPARQL endpoints, which we simply refer to as *data sources*. For each such data source $D \in \mathcal{D}$, we write $G(D)$ to denote the underlying RDF graph exposed by D. Hence, when requesting data source D to execute a SPARQL query Q, we expect that the result returned by D is the set $[\![Q]\!]_{G(D)}$. Then, we define the result of a SPARQL query Q over the federation $\mathcal{D}$ to be a set of solution mappings that is equivalent to $[\![Q]\!]_{G_\mathcal{D}}$ where $G_\mathcal{D} = \bigcup_{D\in\mathcal{D}} G(D)$.

[2] CostFed is open-source and available online at `https://github.com/AKSW/CostFed`.
[3] URI authority: `https://tools.ietf.org/html/rfc3986#section-3.2`

4. Data Summaries

The basis of the CostFed approach is an index that stores a dedicated data summary for each of the data sources in the federation. The innovation of these data summaries is twofold: First, they take into account the skew distribution of subjects and objects per predicate in each data source. Second, they contain prefixes of URIs that have been constructed such that our source selection approach (cf. Section 5) can use them to prune irrelevant data sources more effectively than the state-of-the-art approaches. This section describes these two aspects of the CostFed data summaries (beginning with the second) and, thereafter, defines the statistics captured by these summaries.

As mentioned in Section 2, HiBISCuS fails to prune the data sources that share the same URI authority. CostFed overcomes this problem by using source-specific sets of strings that many URIs in a data source begin with (hence, these strings are prefixes of the URI strings). These *common URI prefixes* are determined as follows: Let ρ be a set of URIs for which we want to find the most common prefixes; in particular, such a set ρ shall be all subject or all object URIs with a given predicate in a data source. We begin by adding all the URIs in ρ to a temporary *trie* data structure (also called prefix tree). While we use a character-by-character insertion in our implementation, we present a word-by-word insertion for the sake of clarity and space in the paper. For instance, inserting the URIs *wiwiss.fu-berlin.de/drugbank/resource/drugs/DB00201* and *wiwiss.fu-berlin.de/drugbank/resource/ references/1002129* from DrugBank leads to the trie shown Figure 1a. We say that a node in the trie is a *common-prefix end node* if (1) it is not the root node and (2) the branching factor of the node is higher than a pre-set threshold. For example, by using a threshold of 1, the node `resource` would be a common-prefix end node in Figure 1a. After populating the trie from ρ and marking all common-prefix end nodes, we can now compute the set of all common URI prefixes for ρ by simply traversing the trie and concatenating each path from the root to one of the marked nodes. In our example, given the branching factor threshold of 1, the only common prefix is *wiwiss.fu-berlin.de/drugbank/resource/*. In the end we delete the temporary trie.

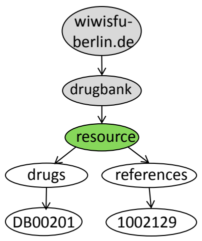
To take into account the skew distribution of subjects and objects per predicate in a data source, CostFed retrieves the frequencies of all the subject/object resources that appear with each of the predicates and orders them in ascending order w.r.t. these frequencies. We then compute the differences in the frequencies between each pair of consecutive resources (e.g., subtract the second ranked frequency from the first, and third from the second, and so on) in the ordered list of subjects/objects. An example skew distribution of the subject frequencies of the DBpedia2015-10 property `foaf:name` is given in Figure 1b. We use this distribution to map resources to one of three mutually disjoint buckets—b0, b1, and b2—which we summarize with a decreasing amount of detail. Informally, we construct the buckets such that b0 contains the high-frequency resources (that appear most often with the predicate in question), b1 contains the middle-frequency resources, and b2 is for the long tail of low-frequency resources. The choice of three buckets was chosen according to the level of details we are storing. In b0 we store resources along with their frequencies. In b1 we store resources along a single avg. frequency of all the resources in the bucket. In b2, we only store avg. frequency. In this way we care the skew distribution of the resources while keeping the resulting index smaller for fast lookup.

The cutting point x_{k+1} for bucket b_k is formally defined as follows. Let f_n be the frequency for a resource r_n where $n = \{1, ..., N\}$, the sequence of the frequencies is expressed as $a_n = \{f_1, ..., f_N\}$ such that $f_n \geq f_{n+1}$ $\forall n < N$. We find the cutting points as:

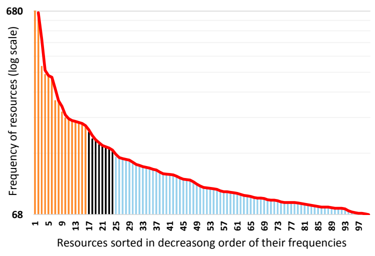
$$x_{k+1} = \min\left(\operatorname*{argmax}_{n \in \{x_k+1, N-1\}} \delta_n \right)$$

where $\delta_n = a_n - a_{n+1}$ is the sequence of the drops and the first cutting point x_0 is zero by definition. In other words, we iteratively look for the first largest drop. The x_{k+1}th frequency is included in bucket b_k. In our implementation, we force $x_1 \geq 10$, which means the first 10 resources are always assigned to b0 (e.g., see Figure 1b). The maximal number of resources in b0 and b1 is limited to 100 to keep our index small enough for fast index lookup during source selection and query planning.[4]

[4] We performed various experiments and these values turned out to be the most suitable.



(a) Trie of URIs



(b) Construction of buckets in skew distribution: Resources in brown go into `b0`, black into `b1`, and blue into `b2`.

Fig. 1: Construction of Trie or prefix tree and buckets

We now define the summarization of the buckets: Informally, for each of the resources in bucket b0, we index it individually together with its corresponding frequency. The resources in b1 are indexed individually along with their average frequency across all resources in b1, and for the long-tail resources in b2 we only record the average selectivity w.r.t. the predicate (i.e., without storing the individual resources). Formally, we capture these summaries by the following notion of *capabilities*. Given a data source $D \in \mathcal{D}$, let p be a predicate used in the data of D (i.e., $p \in \{p' \mid (s', p', o') \in G(D)\}$). Moreover, let $sbjs(p, D)$, respectively $objs(p, D)$, be the set of subjects, respectively objects, in all triples in D that have p as predicate, and let the sets $sb0, sb1, sb2 \subseteq sbjs(p, D)$ and $ob0, ob1, ob2 \subseteq objs(p, D)$ represent the corresponding three subject buckets and object buckets, respectively (hence, $sb0$–$sb2$ are pairwise disjoint, and so are $ob0$–$ob2$). We define the *p-specific capability of D* as a tuple that consists of the following elements:

1. The map *b0Sbjs* associates each subject resource $s \in sb0$ with a corresp. cardinality $b0Sbjs(s) = \big|\{(s', p', o') \in G(D) \mid s' = s \text{ and } p' = p\}\big|$.
2. The map *b0Objs* associates each object resource $o \in ob0$ with a corresp. cardinality $b0Objs(o) = \big|\{(s', p', o') \in G(D) \mid o' = o \text{ and } p' = p\}\big|$.
3. In $b1Sbjs = (sb1, c)$, c is the average cardinality in the corresp. bucket b1, i.e., $c = \frac{1}{|sb1|} \sum_{s \in sb1} \big|\{(s', p', o') \in G(D) \mid s' = s \text{ and } p' = p\}\big|$.
4. In $b1Objs = (ob1, c)$, c is the average cardinality in the corresp. bucket b1, i.e., $c = \frac{1}{|ob1|} \sum_{o \in ob1} \big|\{(s', p', o') \in G(D) \mid o' = o \text{ and } p' = p\}\big|$.
5. $sbjPrefix(p, D)$ is a set of common URI prefixes computed for the set $sbjs(p, D)$ of URIs (by using the trie data structure as described above).
6. $objPrefix(p, D)$ is a set of common URI prefixes computed for $objs(p, D)$.[5]
7. $avgSS(p, D)$ is the average subject selectivity of p in D considering only the corresponding bucket b2; i.e., $avgSS(p, D) = 1/|sb2|$.
8. $avgOS(p, D)$ is the average object selectivity of p in D considering only the corresponding bucket b2; i.e., $avgOS(p, D) = 1/|ob2|$.
9. $T(p, D)$ is the total number of triples with predicate p in D.

Note that the total number of capabilities that CostFed indexes for a source D is equal to the number of distinct predicates in D. However, the predicate rdf:type is treated in a special way. That is, the rdf:type-specific capability of any source $D \in \mathcal{D}$ does not store the set $objPrefix(\text{rdf:type}, D)$ of common object prefixes, but instead it stores the *set of all distinct class URIs* in D, i.e., the set $\{o \mid (s, \text{rdf:type}, o) \in G(D)\}$. The rationale of this choice is that the set of distinct *classes* used in a source D is usually a small fraction of the set of all resources in D. Moreover, triple patterns with predicate rdf:type are commonly used in SPARQL queries. Thus, by storing the complete class URIs instead of the respective prefixes, we can potentially perform a more accurate source selection.

[5] We do not consider literal object values for $objPrefix(p, D)$ because, in general, literals do not share longer common prefixes and we want to keep the index small.

For each data source $D \in \mathcal{D}$, CostFed also stores the overall number of distinct subjects $tS(D)$, the overall number of distinct objects $tO(D)$, and the overall size $tT(D)$ of $G(D)$. An excerpt of a data summary is given in the supplementary material.[6] Most of the statistics in our data summaries can be obtained by simply sending SPARQL queries to the underlying SPARQL endpoints. Any later updates in the data sources do not require the complete index update. Rather, we only need to update the specific set of *capabilities* where changes are made. CostFed can perform an index update on a specified point of time as well as on a regular interval.

5. Source Selection

5.1. Foundations

As a foundation of our source selection approach we represent any basic graph pattern (BGP) of a given SPARQL query as some form of a directed hypergraph. In general, every edge in a directed hypergraph is a pair of sets of vertexes (rather than a pair of two single vertexes as in an ordinary digraph). In our specific case, every hyperedge captures a triple pattern; to this end, the set of source vertexes of such an edge is a singleton set (containing a vertex for the subject of the triple pattern) and the target vertexes are given as a two-vertex sequence (for the predicate and the object of the triple pattern). For instance, consider the query in Figure 2a whose hypergraph is illustrated in Figure 3a (ignore the edge labels for the moment). Note that, in contrast to the commonly used join graph representation of BGPs in which each triple pattern is an ordinary directed edge from a subject node to an object node [15], our hypergraph-based representation contains nodes for all three components of the triple patterns. As a result, we can capture joins that involve predicates of triple patterns. Formally, our hypergraph representation is defined as follows.

Definition 5.1 (Hypergraph of a BGP). *The hypergraph representation of a BGP B is a directed hypergraph $HG = (V, E)$ whose vertexes are all the components of all triple patterns in B, i.e., $V = \bigcup_{(s,p,o)\in B}\{s, p, o\}$, and that contains a hyperedge $(S, T) \in E$ for every triple pattern $(s, p, o) \in B$ such that $S = \{s\}$ and $T = (p, o)$.*

Note that, given a hyperedge $e = (S, T)$ as per Definition 5.1, since T is a (two-vertex) sequence (instead of a set), we may reconstruct the triple pattern $(s, p, o) \in B$ from which the edge was constructed. Hereafter, we denote this triple pattern by $tp(e)$. Then, given the hypergraph $HG = (V, E)$ of a BGP B, we have that $B = \{tp(e) \mid e \in E\}$. For every vertex $v \in V$ in such a hypergraph we write $E_{\text{in}}(v)$ and $E_{\text{out}}(v)$ to denote the set of incoming and outgoing edges, respectively; i.e., $E_{\text{in}}(v) = \{(S, T) \in E \mid v \in T\}$ and $E_{\text{out}}(v) = \{(S, T) \in E \mid v \in S\}$. If $|E_{\text{in}}(v)| + |E_{\text{out}}(v)| > 1$, we call v a *join vertex*.

During its hypergraph-based source selection process, CostFed manages a mapping λ that labels each hyperedge e with a set $\lambda(e) \subseteq \mathcal{D}$ of data sources (i.e., SPARQL endpoints). These are the sources selected to evaluate the triple pattern $tp(e)$ of the hyperedge. In the initial stage of the process, such a label shall consist of all the sources that contain at least one triple that matches the triple pattern. We call each of these sources *relevant* (or *capable*) for the triple pattern. More specifically, a data source $D \in \mathcal{D}$ is relevant for a triple pattern tp if the result of evaluating tp at D is nonempty; that is, $[\![tp]\!]_{G(D)} \neq \emptyset$. Hereafter, let $R(tp) \subseteq \mathcal{D}$ denote the set of all relevant sources for tp.

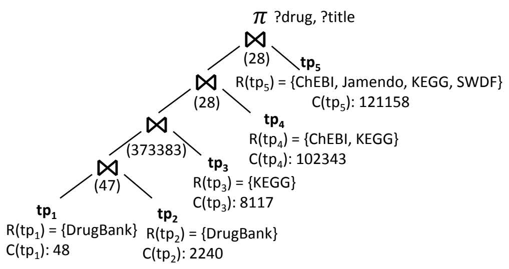
Note that this notion of relevance focuses on each triple pattern independently. As a consequence, there may be a source (or multiple) that, even if relevant for a triple pattern of a query, the result for that triple pattern from this source cannot be joined with the results for the rest of the query and, thus, does not contribute to the overall result of the whole query. Therefore, we introduce another, more restrictive notion of relevance that covers only the sources whose results contribute to the overall query result. That is, a data source $D \in \mathcal{D}$ is said to *contribute* to a SPARQL query Q if there exists a triple pattern tp in Q and a solution mapping μ in the result of Q over the federation $\mathcal{D}$ such that $\mu(tp)$ is a triple that is contained in $G(D)$. Hereafter, for every triple pattern tp of a SPARQL query Q, we write $C_Q(tp)$ to denote the set of all data sources (in $\mathcal{D}$) that are relevant for tp and contribute to Q; hence, $C_Q(tp) \subseteq R(tp)$. Then, the problem statement underlying the source selection of CostFed is given as follows:

Definition 5.2 (CostFed Source Selection Problem). *Let Q be a SPARQL query that consists of n BGPs. Given a set $DHG = \{(V_1, E_1), \ldots, (V_n, E_n)\}$ of hypergraphs that represent these BGPs, determine an edge labeling $\lambda : (E_1 \cup \cdots \cup E_n) \rightarrow 2^{\mathcal{D}}$ such that for each hyperedge $e \in (E_1 \cup \cdots \cup E_n)$ it holds that $\lambda(e) = C_Q(tp(e))$.*

[6] CostFed supplementary material: `https://goo.gl/otj9kq`

```
SELECT ?drug ?title WHERE {
?drug db:drugCategory dbc:micronutrient.
 # R(tp1)={DrugBank}
?drug db:casRegistryNumber ?id.
 # R(tp2)={DrugBANK}
?keggDrug rdf:type kegg:Drug.
 # R(tp3)={KEGG}
?keggDrug bio2rdf:xRef ?id.
 # R(tp4)={ChEBI, KEGG}
?keggDrug purl:title ?title.
 # R(tp5)={ChEBI, Jamendo, KEGG, SWDF}}
```

(a) Triple pattern-wise relevant sources



(b) Unoptimized left-deep plan

Fig. 2: Motivating Example: FedBench LS6 query. C(tp) represents the cardinality of triple pattern tp.

5.2. Source Selection Algorithm

CostFed's source selection comprises two steps: Given a query Q, we first determine an *initial edge labeling* for the hypergraphs of all the BGPs of Q; i.e., we compute an initial $\lambda(e)$ for every $e \in E_i$ in each $(V_i, E_i) \in DHG$. In a second step, we *prune the labels of the hyperedges* assigned in the first step. The first step[7] works as follows: For hyperedges of triple patterns with unbound subject, predicate, and object (i.e., $tp = <?s, ?p, ?o>$) we select all sources in $\mathcal{D}$ as relevant. For triple patterns with predicate `rdf:type` and bound object, an index lookup is performed and all sources with matching capabilities are selected. For triple patterns with either bound subject or bound object or common predicate (i.e., appears in more than 1/3 of $\mathcal{D}$), we perform an `ASK` operation; that is, an `ASK` query with the given triple pattern is sent to each of the sources in $\mathcal{D}$, respectively, and the sources that return `true` are selected as relevant sources for the triple pattern. The results of the `ASK` operations are stored in a cache for future lookup. Figure 3a shows the resulting hyperedge labeling of the example query.

5.2.1. Pruning approach

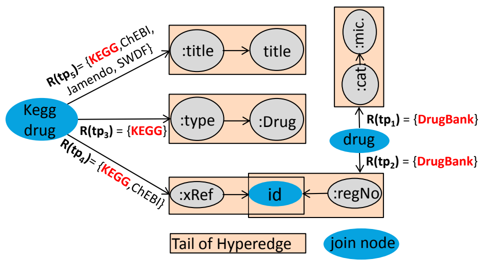
After labeling the edges of the hypergraphs, we prune irrelevant sources from the labels by using the *source-pruning* algorithm shown in Algorithm 1. The intuition behind our pruning approach is that knowing which stored prefixes are relevant to answer a query can be used to *discard triple pattern-wise (TPW) selected sources that will not contribute* to the final result set of the query. Our algorithm takes the set of all labeled hypergraphs as input and prunes labels of all hyperedges that are either incoming or outgoing edges of a join node. Note that our approach deals with each hypergraph $(V_i, E_i) \in DHG$ of the query separately (Line 1 of Algorithm 1). For each node $v \in V_i$ that is a join node, we first retrieve the sets (1) $SPrefix$ of the subject prefixes contained in the elements of the label of each outgoing edge of v (Lines 2–7 of Algorithm 1) and (2) $OPrefix$ of the object prefixes contained in the elements of the label of each ingoing edge of v (Lines 8–10 of Algorithm 1)[8].

Now we merge these two sets to the set P (set of sets) of all prefixes (Line 11 of Algorithm 1). Next, we plot all of the prefixes of P into a trie data structure. For each prefix p in P we then check whether p ends at a child node of the trie. If a prefix does not end at a child node, then we get all of the paths from the prefix last node (say n) to each leaf of n. These new paths (i.e., prefixes) resulted from p are then replaced in the prefix (Lines 12–22 of Algorithm 1). The intersection $I = (\bigcap_{p_i \in P} p_i)$ of these element sets is then computed. Finally, we recompute the label of each hyperedge e that is connected to v. To this end, we compute the subset of the previous label of e which is such that the set of prefixes of each of its elements is not disjoint with I (see Lines 24 onwards of Algorithm 1). These are the only sources that will potentially *contribute* to the final result set of the query. The sources that are finally selected after the pruning are

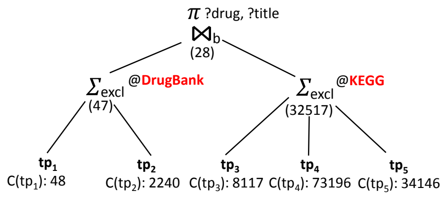
[7] Due to space limitation, the pseudo code of this algorithm can be found (along with a description) in the supplementary material at https://goo.gl/otj9kq.
[8] We encourage readers to refer to https://goo.gl/JJby23 during the subsequent steps of the pruning algorithm. The file contains a running example of the pruning algorithm.

Algorithm 1: CostFed's source pruning algorithm

```
input  : $DHG$ ;                    /* set of hypergraphs that represent the BGPs of a query */
output : $DHG$ ;    /* set of hypergraphs that represent the BGPs of a query with prunned sources */
foreach hypergraph $(V_i, E_i) \in DHG$ do
    foreach hypergraph vertex $v \in V_i$ do
        if $v$ is a join vertex then
            $SPrefix = \emptyset$; $OPrefix = \emptyset$ ;
            foreach hyperedge $e \in E_{out}(v)$ do
                $SPrefix = SPrefix \cup$ {subjectPrefixes($e$)} ;    /* subjectPrefixes($e$) is a function to get all subject
                  prefixes from index for the triple pattern represented by the hyperedge $e$.  */
            end
            foreach hyperedge $e \in E_{in}(v)$ do
                $OPrefix = OPrefix \cup$ {objectPrefixes($e$)}
            end
            $P = SPrefix$ concat $OPrefix$ ;                                        /* merged set */
            $Tr$ = getTrie($P$) ;                         /* get Trie of all prefixes, no branching limit */
            foreach prefix $p \in P$ do
                if $!isLeafPrefix(p,Tr)$ then                  /* prefix does not end at a leaf of Trie */
                    $C$ = getAllChildPaths($p$) ;   /* get all paths from prefix last node n to each leaf of n */
                    $A = \emptyset$ ;                          /* to store all possible prefixes of a given prefix */
                    foreach path $c \in C$ do
                        $A = A \cup p$.concatenate($c$) ;
                    end
                    $P$.replace(p,A) ;                          /* replace p with its all possible prefixes */
                end
            end
            $I = P$.get(1) ;                                            /* get first element of prefixes */
            foreach prefix $p \in P$ do
                $I = I \cap p$ ;                                   /* intersection of all elements of $P$ */
            end
            foreach hyperedge $e \in E_{in}(v) \cup E_{out}(v)$ do
                $label = \emptyset$ ;                                    /* variable for final label of $e$ */
                foreach data source $d_i \in \lambda(e)$ do
                    if $prefixes(d_i) \cap I \neq \emptyset$ then
                        $label = label \cup d_i$ ;
                    end
                end
                $\lambda(e) = label$
            end
        end
    end
end
```



(a) DLH of Figure 2a query and source selection



(b) Query plan

Fig. 3: CostFed source selection and query plan for query given in Figure 2a. **Bold red** are the sources finally selected after the pruning Algorithm 1.

shown in bold red in Figure 3a. We are sure not to lose any recall by this operation because joins act in a conjunctive manner. Consequently, for a data source D_i in the initial label of a hyperedge, if the results of D_i cannot be joined with the results of at least one source of each of the other hyperedges, it is guaranteed that D_i will not *contribute* to the final result set of the query.

6. Query Planning

6.1. Triple Pattern Cardinality Estimation

Let $tp =< s, p, o >$ be a triple pattern having predicate p and $R(tp)$ be the set of relevant sources for that triple pattern. By using the notations used in Section 4, the cardinality $C(tp)$ of tp for b2 is calculated as follows (the predicate b stands for bound):

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

We perform an index lookup if the triple pattern contains a bound subject or object belonging to buckets b0 or b1. The frequencies (i.e., individual for b0 and averaged across b1) are already stored in the index. For other cases, the cardinality estimation remains the same as explained above.

By using the equation, we can estimate the cardinality of the first two triple patterns of the query given in Figure 2a as follows: Since `DrugBank` is the only relevant source for triple patterns $tp1$ and $tp2$, we have that C(tp1) = $T($`db:drugCategory`, `DrugBank`$) \times avgOS($`db:drugCategory`, `DrugBank`$) = 4602 \times 0.0017123287671232876 \approx 8$, and C(tp2) = T(`db:casRegistryNumber`, `DrugBank`) $\approx$ 2240 . The actual cardinalities are 47 and 2240 for $tp1$ and $tp2$, respectively.

6.2. Join Cardinality Estimation

Our query planning relies on the subsequent recursive definition of a BGP B [2]: (1) A triple pattern is a BGP; (2) If B1 and B2 are BGPs, then $B1$ JOIN $B2$ represented as $B1 \bowtie B2$ is a BGP. The join cardinality of two BGPs B_1 resp. B_2 w.r.t. the datasets D_i resp. D_j is estimated as follows:

$$C(B1 \bowtie B2) = M(B1) \times M(B2) \times Min(C(B1), C(B2)) \quad (1)$$

where $M(B)$ is the average frequency of multivalued predicate (a predicate which can have multiple values, e.g., a person x can have more than one contact numbers) in BGP B (note B is the triple pattern in this case and not the result of previously computed joins between triple patterns) and is calculated as follows:

M(B) =

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

where $j(s)$ means that the subject of the triple pattern is involved in the join and $tp =< s, p, o >$ is a triple pattern. Note that the multivalued predicates can have dramatic effect on the cardinalities of joins between triple patterns. Consider, the sample dataset and SPARQL query given in Figure 4, the cardinality of the first triple pattern tp1 is 4 and cardinality of the second triple pattern tp2 is 2. A query planner which ignores the multivalued predicates (e.g. SemaGrow) will simply selects the minimum of the cardinalities of the two triple patterns (i.e. 2) as cardinality of the joins (on variable ?s) between the given triple patterns in the query. However, the actual cardinality of the join is 6. In CostFed, $M(tp1) = 2$ (i.e. 4/2), $M(tp1) = 2$ (2/1) and $Min(C(tp1), C(tp2)) = 2$. Thus CostFed's estimated cardinality according to Equation 1 is 2*2*2 = 8.

```
:s1 :p1 :o1, :o2, :o3.
:s1 :p2 :o4, :o5.
:s2 :p1 :o6.
```

```
SELECT SELECT ?o1 ?o2 WHERE
{ ?s :p1 ?o1 .   //tp1
  ?s :p2 ?o2 .   //tp2 }
```

Fig. 4: Effect of multivalued predicates: Sample RDF with multivalued predicates p1, p2 and SPARQL query. Prefixes are ignored for simplicity

6.3. Join Cost Estimation

We make use of both bind join ($\bowtie_b$) [2, 14, 5] and symmetric hash join ($\bowtie_h$) [1] in our query planning, and decide (explained in the next section) between the two based on a cost estimations given as follow:

$$Cost(B1 \bowtie_h B2) = \frac{(1+TC)}{TC} * CSQ + C(B2) * CRT + (C(B1) + C(B2)) * CHT$$

$$Cost(B1 \bowtie_b B2) = CSQ + C(B1) * CRT + CSQ * \frac{(\frac{C(B1)+BSZ-1}{BSZ}) + CTC - 1}{CTC}$$

CSQ is the cost of sending a SPARQL query, CRT is the cost of receiving a single result tuple, CHT is the cost of handling a single result tuple, BSZ is the binding block size, TC is the number of threads used to query SPARQL endpoints. We chose CSQ=100, CRT=0.01, CHT=0.0025, TC=20, BSZ=20 after an empirical study based on LSQ [8] and looking at the values used in SemaGrow [2]. For the query given in Figure 2a, the $Cost(T1 \bowtie_h T2) = 100 + 8 \times 0.01 + (8 + 2240) \times 0.0025 = 105.7$ and $Cost(T1 \bowtie_b T2) = 100 + 8 \times 0.01 + 1 \times 100 = 200.08$

6.4. Exclusive Groups

Many of the existing SPARQL federation engines [14, 5, 1] make use of the notion of exclusive groups: a set of triple patterns whose single relevant source is D. The advantage of exclusive groups of size greater than 1 is that they can be combined to a conjunctive query and sent to D in a single sub-query, thus minimizing the local computation [14]. As an example, consider CostFed's source selection given in Figure 3a. The first two triple patterns (i.e., tp1, tp2) form an exclusive group, since `DrugBank` is the single relevant source for these triple patterns. Similarly, the last three triple patterns (i.e., tp3, tp4, tp5) form another exclusive group for `KEGG` data source.

6.5. Join Ordering

CostFed's join ordering approach is shown in Algorithm 2. It takes a set of join arguments (i.e., triple patterns or groups of triple patterns) as input. We first estimate the cardinality of each argument in the input set. We then store it together with the argument in a pair. For example $(tp1, C(tp1))$ is a pair which consists of triple pattern tp and its estimated cardinality. We store all these pairs in the argPairs collection (Lines 1-4) and sort the collection by cardinality values in ascending order (Line 5). Thus, the first element of this collection is the join pair with minimal cardinality. It will be the left argument of the first binary join operator of the resulting output join tree (Line 6). We then try to find the right argument of the first binary join operator. It should be an argument having at least one common variable with the left argument and minimal possible cardinality. For this, we traverse the remaining pairs in argPairs collection (Lines 10-16). If there is no argument having common variables we just get the next argument with minimal cardinality (Lines 17-19). We next decide between bind and symmetric hash joins selecting the join implementation that leads to the smaller cost (Line 21). Subsequently, we create an appropriate join node and set it as the left argument for the next binary join (Lines 22-25). We repeat the procedure for finding the right argument for the next binary join (Lines 9-26) and get a left-deep join tree as final output.

The final CostFed query plan for the motivating example query is shown in Figure 3b. In this query plan we have two exclusive groups. Thus, these joins are executed remotely. The results of the exclusive groups are then joined locally using a bind join. Note the query plan for the same query is also provided by FedX [14] and SemaGrow [2]

Algorithm 2: Join Order Optimization

```
input  : a list joinargs of join arguments
output : a root of join tree nodes
argPairs ← $\emptyset$;
foreach arg in the joinargs do
    argPairs ← argPairs ∪ Pair(arg, FindCardinality(arg));
end
argPairs ← SortByCardinalityAsc(argPairs);
leftJoinArgumentPair ← PopFront(argPairs) ;               /* get the smallest card pair */
rightJoinArgumentPair ← $\emptyset$ ;                      /* needs to find the right join arg */
joinVars ← FreeVars(First(leftJoinArgumentPair)) ;        /* first tuple vars */
while argPairs $\neq \emptyset$ do
    foreach pair in the argPairs do
        if GetCommonVars(joinVars, First(pair)) $\neq \emptyset$ then
            rightJoinArgumentPair ← pair;
            argPairs ← argPairs– pair ;                   /* remove pair from argPairs */
            break;
        end
    end
    if rightJoinArgumentPair = $\emptyset$ then
        rightJoinArgumentPair ← PopFront(argPairs);
    end
    joinVars ← joinVars ∪ FreeVars(First(rightJoinArgumentPair));
    if HashJoinCost(leftJoinArgumentPair, rightJoinArgumentPair) < BindJoinCost(leftJoinArgumentPair, rightJoinArgumentPair) then
        leftJoinArgumentPair ← CreateHashJoin(leftJoinArgumentPair, rightJoinArgumentPair);
    else
        leftJoinArgumentPair ← CreateBindJoin(leftJoinArgumentPair, rightJoinArgumentPair);
    end
end
result ← leftJoinArgumentPair;
```

in the corresponding papers. Both of these systems overestimate the set of relevant sources, i.e., select a total of nine sources (one each for first three triple patterns, two for fourth triple pattern, four for fifth triple pattern, ref., Section 7.2). In contrast, CostFed selects the optimal five sources, one for each triple pattern and then only performs a single local join (both FedX and SemaGrow perform more than one local join for the same query). The resulting query runtime improvement is shown in Table 1.

CostFed fully supports all the SPARQL features supported by Sesame API. We used multi-threaded worker pool to execute both the joins and union operators in a highly parallelized fashion. In addition, to achieve a higher throughput, we used the pipelining approach.

7. Evaluation

7.1. Experimental Setup

We used FedBench [13] for our evaluation which comprises 25 queries, 14 of which (CD1-CD7, LS1-LS7) are for SPARQL endpoint federation approaches (the other 11 queries (LD1-LD11) are for Linked Data federation approaches [10]). Hence, we used all 14 SPARQL endpoint federation queries in our evaluation. In addition, we used the Complex queries (C1-C10) from LargeRDFBench [9] to test CostFed's performance on more complex queries. These complex queries have more triple patterns (at least 8 vs. a maximum of 7 in FedBench), more join vertices (3-6 vs. 1-5 in FedBench) and a higher mean join vertex degree (2-6 vs. 2-3 in FedBench). In addition, they were designed to use more SPARQL clauses (especially, `DISTINCT`, `LIMIT`, `FILTER` and `ORDER BY`) that are missing in the FedBench queries. Further details about the complex queries can be found at the aforementioned LargeRDFBench project website.

Each of FedBench's nine datasets was loaded into a separate physical Virtuoso 7.2 server, each of which equipped with a 3.2GHz i7 processor, 32 GB RAM and a 500 GB hard disk. The client machine that ran the experiments had the same specification. We conducted the experiments in a local network. Hence, the network costs were negligible. Each query was executed 15 times and the results were averaged. We best choose 4 as trie branching factor for the index construction. The query timeout was 20 min. We compared the selected engines based on: (1) the total number of triple pattern-wise sources selected, (2) the total number of SPARQL `ASK` requests submitted during the source selection, (3) the average source selection time, (4) the average query execution time, (5) the index/data summary generation time (if applicable), and (6) index compression ratio.

7.2. *Experimental Results*

We first measured the compression ratio[9] achieved by each system. CostFed achieves an index size of 9.5 MB for the complete FedBench data dump (19.7 GB), leading to a high compression ratio of 99.99%. The other approaches achieve similar compression ratios. CostFed's index construction time is around 60 min for all of FedBench. ANAPSID requires only 5 min while SPLENDID and SemaGrow need 110 min. HiBISCuS's indexing runtime lies around 41 min.

More importantly, we analysed the results of the source selection and overall query runtime. We define efficient source selection in terms of: (1) the total number of triple pattern-wise sources selected (#T), (2) the total number of `ASK` requests (#A) used to obtain (1), and (3) the source selection time (ST). Table 1 shows the results of these three metrics for the selected approaches. Overall, CostFed is the most efficient source selection approach w.r.t. all metrics. It selects the smallest #T, i.e., 70 for FedBench and 104 for Complex queries (see the average/total values in Table 1). Similarly, it requires the smallest number of `ASK` queries during the source selection along with the smallest source selection time. CostFed outperforms HiBISCuS w.r.t. source selection time as CostFed's index is loaded as hash tables and addressed using sorted tables (HiBISCuS relies on a Sesame model and SPARQL queries for lookup). It is important to mention that FedX, HiBISCUS, and CostFed cache the results of `ASK` requests used during the source selection. Thus, they always perform a cache lookup before sending an `ASK` request to the underlying SPARQL endpoint. The runtime results in Table 1 are the results for the warm cache of these federation engines. We can clearly see that the join-aware source selection approaches, i.e., CostFed, ANAPSID, and HiBISCUS select around half (e.g., 70 for CostFed vs. 134 for FedX on FedBench) of the total #T selected by the non-join-aware source selection federation engines. As mentioned before, such an overestimation of sources can be very costly (extra network traffic, irrelevant intermediate results). The effect of such overestimation is even more critical while dealing with large data queries.

The query execution time is often used as key metric to compare federation engines. Herein, we consider the query execution time to be the time necessary to gather all the results from the result set iterator of each engine. Moreover, we considered each time-out to be equal to a runtime of 20min while computing the average runtimes (RT) presented in the row T/A of Table 1. Since HiBISCUS is only a source selection approach, it cannot provide query runtimes. Overall, CostFed clearly outperforms the other selected systems. On FedBench, CostFed is better than FedX on 11/14 queries and outperforms SPLENDID, ANAPSID and SemaGrow on all 14 queries. CostFed's average runtime across all 14 FedBench queries is only 440ms while FedX needs 7,468ms (i.e., 16 times the runtime of CostFed), SPLENDID's is 5,3404ms (i.e., 121× slower than CostFed), ANAPSID's is 12,467ms (i.e., 28× that of CostFed), and SemaGrow's is 1,203ms (i.e., 3× slower than CostFed). Since the execution times for the FedBench queries are very small, i.e., less than 3 seconds on CostFed, the average runtime performance for a system is greatly affected if a particular query takes too long. For example, FedX takes 94,519ms to execute LS6, due to which overall runtime performance is greatly decreased comparing to CostFed. If we remove the LS6 runtime, then FedX's average (across the remaining 13 queries) runtime is 771 ms (2× CostFed's). To address the drawbacks of the short runtimes of FedBench queries, we used more complex queries from LargeRDFBench which have longer (up to 20 min) query runtimes. On complex queries from LargeRDFBench, CostFed is better than FedX on 8/9 comparable queries (C5 time outs for both systems), better than SPLENDID on 7/8 comparable queries (C4 and C9 result in runtime errors for SPLENDID), better than ANAPSID on 8/9 queries (C5 results in a runtime error for ANAPSID) queries, and better than SemaGrow on 9/9 queries (C5 times out for both systems). On these queries, CostFed's average (over all 10 queries) query runtime is 122,574ms while FedX's average is 246,296ms (i.e., 2× of CostFed). SPLENDID necessitates 1.73× CostFed's runtime while ANAPSID has an average runtime of 147,265ms (i.e., 1.2× CostFed's) while SemaGrow achieved 367,496ms (i.e., 3× of CostFed).

Overall, our results show clearly that CostFed outperforms the state of the art on both benchmark datasets. On the queries where FedX is faster than CostFed (3/14 FedBench queries), the maximum runtime difference is only of 6ms, which suggest that our approach performs well across the diverse types of queries in our experiments. We also compared the effectiveness of only the query planning (excluding source selection) of the selected engines. Due to space limitation, the complete evaluation results can be found at extended version of this paper[10].

[9] Compression ratio = (1 - index size/total data dump size).

[10] Extended CostFed's version available from `https://svn.aksw.org/papers/2018/SEMANTICS_CostFed/public.pdf`

Table 1: Comparison of the federation engines in terms of total triple pattern-wise sources selected **#T**, total number of SPARQL `ASK` requests **#A**, source selection time **ST** in msec, and average query runtime **RT**. (**ST*,RT*** = FedX source selection time and query runtime with cold cache, respectively, **TO** = Time Out of 20 min, **RE** = Runtime Error, **T/A** = Total/Average, where Total is for #T, #A, and Average is ST, RT)

	FedX						SPLENDID				ANAPSID				SemaGrow				CostFed				HiBISCuS		
Qry	#T	#A	ST*	ST	RT*	RT	#T	#A	ST	RT	#T	#A	ST	RT	#T	#A	ST	RT	#T	#A	ST	RT	#T	#A	ST
CD1	11	27	295	5	304	10	11	26	293	430	3	19	261	294	11	26	293	1686	4	18	6	16	4	18	227
CD2	3	27	229	1	231	2	3	9	33	60	3	1	8	26	3	9	33	50	3	9	1	4	3	9	46
CD3	12	45	330	4	358	27	12	2	17	230	5	2	34	74	12	2	17	155	5	0	1	22	5	0	82
CD4	19	45	319	3	351	33	19	2	14	122	5	3	15	46	19	2	14	176	5	0	1	8	5	0	74
CD5	11	36	306	3	329	21	11	1	11	65	4	1	8	35	11	1	11	104	4	0	1	8	4	0	54
CD6	9	36	297	4	777	479	9	2	16	25210	9	10	36	130899	9	2	16	857	8	0	3	196	8	0	35
CD7	13	36	280	5	828	545	13	2	19	5246	6	5	67	351	13	2	19	1004	6	0	1	275	6	0	32
LS1	1	18	149	1	167	18	1	0	2	75	1	0	5	78	1	0	2	80	1	0	1	17	1	0	55
LS2	11	27	241	4	275	33	11	26	200	1440	15	19	69	361	11	26	200	787	4	18	5	17	7	18	356
LS3	12	45	326	3	5141	4818	12	1	11	16868	5	11	46	8755	12	1	11	6544	5	0	1	2698	5	0	262
LS4	7	63	419	4	432	5	7	2	19	200	7	0	12	2444	7	2	19	89	7	0	1	7	7	0	333
LS5	10	54	377	3	1709	1336	10	1	7	90396	7	4	20	4059	10	1	7	1614	7	0	1	701	8	0	105
LS6	9	45	330	3	94848	94519	9	2	8	5586	5	12	58	20312	9	2	8	162	5	0	1	34	7	0	180
LS7	6	45	317	3	2800	2702	6	1	6	601731	5	2	18	6804	6	1	6	3542	6	0	1	2169	6	0	81
T/A	134	549	302	3	7753	7468	134	77	46	53404	80	89	463	12467	134	77	46	1203	70	45	1.7	440	76	45	137
C1	11	104	455	4	4110	3710	11	1	11	61415	8	1	11	2753	11	1	11	TO	8	0	1	1849	9	0	114
C2	11	104	461	3	1665	1205	11	1	7	80212	8	2	30	TO	11	1	7	1347	8	0	1	1057	9	0	16
C3	21	104	458	4	13608	13155	21	3	12	200171	10	33	79	1403	21	3	12	11580	11	0	1	654	11	0	200
C4	28	156	580	5	TO	TO	28	0	3	RE	28	32	60	87615	28	0	3	5695	18	0	1	837	18	0	45
C5	33	104	451	4	TO	TO	33	0	3	TO	8	3	17	RE	33	0	3	TO	10	0	1	TO	10	0	55
C6	24	117	499	4	24648	24151	24	0	2	40276	9	3	14	3038	24	0	2	47185	9	0	1	2057	9	0	445
C7	17	117	502	3	840	340	17	2	9	11517	9	5	20	1120	17	2	9	6114	9	0	1	116	9	0	175
C8	25	143	540	2	2625	2084	25	2	11	93691	11	2	19	2332	25	2	11	2437	11	0	1	996	11	0	187
C9	16	117	515	317	17987	17668	16	2	17	RE	9	16	52	9085	16	2	17	TO	9	0	1	18040	9	0	170
C10	13	130	535	4	1177	645	13	0	3	14680	11	6	31	18040	13	0	3	611	11	0	1	136	11	0	140
T/A	199	1196	500	4	246666	246296	199	11	7.8	212745	111	103	33.3	147265	199	11	7.8	367496	104	0	1	122574	106	0	154.7

8. Conclusion and Future Work

We presented CostFed, a federated engine for SPARQL endpoint federation. We showed that a cost-based optimization combined with join-aware source selection can lead to significant performance improvements. In future, beside caching the results of the SPARQL `ASK` requests we use, we will also cache the most common intermediate results as well as the actual join cardinalities and use them during the query planning and execution to optimize the performance of CostFed.

Acknowledgements: This work was supported by the H2020 project HOBBIT (GA number 688227) and CENIIT program at Linköping University (project no. 17.05).

References

[1] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *ISWC*, 2011.

[2] A. Charalambidis, A. Troumpoukis, and S. Konstantopoulos. Semagrow: Optimizing federated sparql queries. In *SEMANTICS*, 2015.

[3] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 1(1), Jan. 2007.

[4] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *ACM SIGMOD*, 2011.

[5] O. Görlitz and S. Staab. Splendid: Sparql endpoint federation exploiting void descriptions. In *COLD at ISWC*, 2011.

[6] G. Montoya, H. Skaf-Molli, and K. Hose. The odyssey approach for optimizing federated sparql queries. In *ISWC*, 2017.

[7] I. A. E. M. M. Ouzzaniù and A. A. P. Kalnisú. Lusail: A system for querying linked data at scale. *VLDB*, 2017.

[8] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A.-C. N. Ngomo. Lsq: the linked sparql queries dataset. In *ISWC*, 2015.

[9] M. Saleem, A. Hasnain, and A.-C. N. Ngomo. Largerdfbench: a billion triples benchmark for sparql endpoint federation. *JWS*, 2018.

[10] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. N. Ngomo. A fine-grained evaluation of sparql endpoint federation systems. *SWJ*, 2015.

[11] M. Saleem and A.-C. Ngonga Ngomo. HiBISCuS: Hypergraph-based source selection for sparql endpoint federation. In *ESWC*, 2014.

[12] M. Saleem, A.-C. Ngonga Ngomo, J. X. Parreira, H. F. Deus, and M. Hauswirth. Daw: Duplicate-aware federated query processing over the web of data. In *ISWC*, 2013.

[13] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench: a benchmark suite for federated semantic data query processing. In *ISWC*, 2011.

[14] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *ISWC*, 2011.

[15] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.