Eurostars Project

3DFed – Dynamic Data Distribution and Query Federation

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

Deliverable 4.1
Initial Report on the 3DFed Federation Engine

Dissemination Level	Public
Due Date of Deliverable	July 31, 2022
Actual Submission Date	July 31, 2022
Work Package	WP4, Distributed Query Processing and Optimization
Deliverable	D4.1
Type	Report
Approval Status	Final
Version	1.0
Number of Pages	12

Abstract: Source selection is one of the most important steps in federated SPARQL query processing. The goal of the source selection is to identify the potentially relevant sources pertaining to a given query. In this deliverable we present an intelligent join-aware source-selection technique for SPARQL-endpoint federation. The source-selection information will later be used to generate optimized query execution plans.

History

Version	Date	Reason	Revised by
0.1	01/07/2022	Initial Template & Deliverable Structure	Muhammad Saleem
0.2	15/07/2022	Initial draft completed	Muhammad Saleem
0.3	19/07/2022	Final draft completed and submitted for review	Muhammad Saleem
0.4	28/07/2022	Review	Milos Jovanovik
1.0	31/07/2022	Finalizing	Muhammad Saleem

Author List

Organization	Name	Contact Information
University of Paderborn	Muhammad Saleem	saleem@informatik.uni-leipzig.de
OpenLink Software	Milos Jovanovik	mjovanovik@openlinksw.com
OpenLink Software	Mirko Spasić	mspasic@openlinksw.com
elevait GmbH & Co. KG	Martin Voigt	martin.voigt@elevait.de

Contents

1 Introduction **3**

2 Related Work **3**

3 Preliminaries **4**

4 Data Summaries **4**

5 Source Selection **7**

5.1 Foundations 7

5.2 Source Selection Algorithm 8

6 Evaluation **10**

6.1 Experimental Setup 10

6.2 Experimental Results 10

7 Conclusion and Future Work **11**

References **11**

1 Introduction

Answering complex queries on the Web of data often requires merging partial results contained across different data sources. The optimization of engines that support this type of queries, called *federated query engines*, is thus of central importance for the efficient and scalable deployment of Semantic Web technologies [7]. Two challenges must be addressed when optimizing federated query processing.

The first is the efficient source selection, i.e., identifying the set of relevant also called capable sources that can answer part of the given query. For a given query, an *optimized selection of sources* is one of the key steps towards the generation of efficient query plans [8]. A poor source selection can lead to increases of the overall query processing time [8]. The second 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. This report targets the first challenge.

To address the first challenge, most SPARQL query federation approaches [2, 3, 11, 9] 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 [9]. 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, 8].

In this deliverable, we present a *trie-based source selection* approach which is a *join-aware approach to TPWSS* based on common URI prefixes. The proposed source selection algorithm is based on labelled hypergraphs [8], 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. We compared the efficiency of the proposed source selection approach with the source selection approaches used in state-of-the-art federation engines ANAPSID [1], SemaGrow [2], SPLENDID [3], HiBISCuS [8], and FedX [11]. 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 on the majority of the FedBench [10] queries. Our results on the more complex queries from LargeRDFBench [6] confirm the results on FedBench.

2 Related Work

FedX [11], SPLENDID [3], ANAPSID [1], SemaGrow [2], ODYSSEY [4], LUSAIL [5] etc. are examples of state-of-the-art SPARQL federation engines. A more exhaustive overview of these systems can be found in [7]. However, they do not consider the skew distribution of subjects and objects across predicates. Our proposed approach is most closely related to HiBISCuS [8] in terms of source selection. HiBISCuS makes use of the different *URIs authorities*[1] to prune irrelevant sources during the source selection. While HiBISCuS can significantly remove irrelevant sources [8], 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, our approach is flexible enough to distinguish between URIs from different datasets that come from the same namespace (e.g., as in `Bio2RDF`).

[1] URI authority: `https://tools.ietf.org/html/rfc3986#section-3.2`

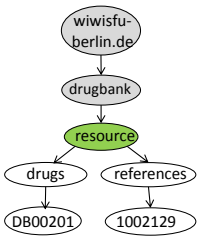
3 Preliminaries

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 $[\![Q]\!]_{G(D)}$.

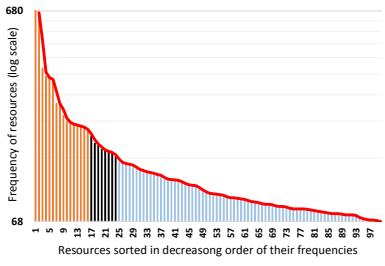
4 Data Summaries

The basis of the our 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 proposed 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. We 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.



(a) Trie of URIs



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

Figure 1: Construction of Trie or prefix tree and buckets

To take into account the skew distribution of subjects and objects per predicate in a data source, we retrieve 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:

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

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.[2]

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).

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

```
@prefix ds:<http://aksw.org/CostFed/> .
[] a ds:Service ;
     ds:url    <http://drugbank.endpoint.url/sparql> ;
 #Predicate based statistics
  ds:capability
     [
            ds:predicate   db:drugCategory ;
            ds:b0Sbjs
               [ ds:subject db:DB01075; ds:card 11 ],
               [ ds:subject db:DB00563; ds:card 11 ],
               [ ds:subject db:DB00424; ds:card 9  ],
               [ ds:subject db:DB00668; ds:card 7  ],
               [ ds:subject db:DB00466; ds:card 7  ];
             ds:b1Sbjs
                db:DB00136, db:DB00169, db:DB00153 ;
             ds:b1SbjsAvgCard 6 ;
     ds:b0Objs
      [ ds:object db:antineoplasticAgents; ds:card 158  ],
      [ ds:object db:enzymeInhibitors;     ds:card 131  ],
      [ ds:object db:antihypertensiveAgents; ds:card 107],
      [ ds:object db:vasodilatorAgents;    ds:card 62   ];
     ds:b1Objs
      db:micronutrient, db:antifungalAgents ;
      ds:b1ObjsAvgCard 45 ;
            ds:objPrefix db:resource/drugs/DB0 ;
            ds:b2AvgSS       5.3219797764768 49E-4 ;
            ds:objPrefix db:resource/drugcategory/ ;
            ds:b2AvgOS  0.0017123287671232876 ;
            ds:T     4602 ;  # No. of triples
     ] ;
#Overall dataset statistics
      ds:totalSbj 20785 ;
      ds:totalObj 90039 ;
      ds:totalTriples 522775 ;
```

Listing 1: Sample Proposed index.

6. $objPrefix(p, D)$ is a set of common URI prefixes computed for $objs(p, D)$.[3]

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 we 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.

For each data source $D \in \mathcal{D}$, we 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 Listing 1. 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. We can perform an index update on a specified point of time as well as on a regular interval.

[3]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.

5 Source Selection

5.1 Foundations

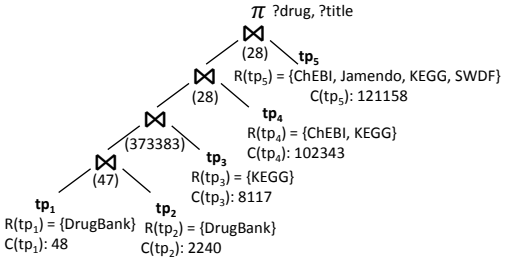
As a foundation of our source selection approach we represent any basic graph pattern (BGP) of a given SPARQL query as some form of a directed hypergraph. In general, every edge in a directed hypergraph is a pair of sets of vertexes (rather than a pair of two single vertexes as in an ordinary digraph). 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 [12], 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 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 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*.

```
1  SELECT ?drug ?title WHERE {
2  ?drug db:drugCategory dbc:micronutrient .
3    # R(tp1)={DrugBank}
4  ?drug db:casRegistryNumber ?id .
5    # R(tp2)={DrugBANK}
6  ?keggDrug rdf:type kegg:Drug .
7    # R(tp3)={KEGG}
8  ?keggDrug bio2rdf:xRef ?id .
9    # R(tp4)={ChEBI, KEGG}
10 ?keggDrug purl:title ?title .
11   # R(tp5)={ChEBI, Jamendo, KEGG, SWDF}}
```

(a) Triple pattern-wise relevant sources



(b) Unoptimized left-deep plan

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

During its hypergraph-based source selection process, we manage 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 the proposed approach is given as follows:

Definition 2 (Our 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 $\lambda\colon (E_1 \cup \cdots \cup E_n) \rightarrow 2^{\mathcal{D}}$ such that for each hyperedge $e \in (E_1 \cup \cdots \cup E_n)$ it holds that $\lambda(e) = C_Q(tp(e))$.*

5.2 Source Selection Algorithm

Our 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[4] 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.

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)[5].

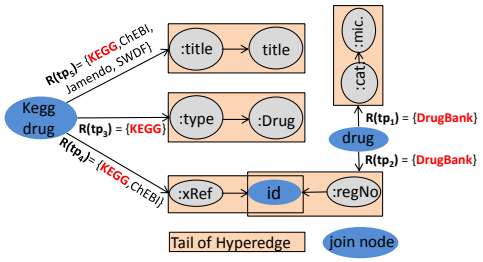
Now we merge these two sets to the set P (set of sets) of all prefixes (Line 11 of Algorithm 1). Next, we plot all of the prefixes of P into a trie data structure. For each prefix p in P we then check whether p ends at a child node of the trie. If a prefix does not end at a child node, then we get all 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 $I = \left(\bigcap_{p_i \in P} p_i\right)$ 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

[4]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`.

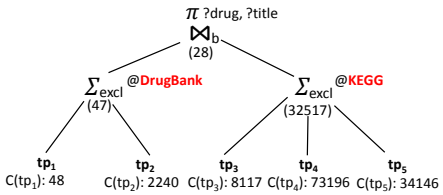
[5]We encourage readers to refer to `https://goo.gl/JJby23` during the subsequent steps of the pruning algorithm. The file contains a running example of the pruning algorithm.

Algorithm 1: Proposed 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 */
 1 foreach hypergraph (V_i, E_i) ∈ DHG do
 2     foreach hypergraph vertex v ∈ V_i do
 3         if v is a join vertex then
 4             SPrefix = ∅; OPrefix = ∅ ;
 5             foreach hyperedge e ∈ E_out(v) do
 6                 SPrefix = SPrefix ∪ {subjectPrefixes(e)} ; /* subjectPrefixes(e) is a function to get all subject prefixes from
                     index for the triple pattern represented by the hyperedge e.  */
 7             end
 8             foreach hyperedge e ∈ E_in(v) do
 9                 OPrefix = OPrefix ∪ {objectPrefixes(e)}
10             end
11             P = SPrefix concat OPrefix ;                                              /* merged set */
12             Tr = getTrie(P) ;                                  /* get Trie of all prefixes, no branching limit */
13             foreach prefix p ∈ P do
14                 if !isLeafPrefix(p,Tr) then                          /* prefix does not end at a leaf of Trie */
15                     C = getAllChildPaths(p) ;          /* get all paths from prefix last node n to each leaf of n */
16                     A = ∅ ;                                  /* to store all possible prefixes of a given prefix */
17                     foreach path c ∈ C do
18                         A = A ∪ p.concatenate(c) ;
19                     end
20                     P.replace(p,A) ;                        /* replace p with its all possible prefixes */
21                 end
22             end
23             I = P.get(1) ;                                                /* get first element of prefixes */
24             foreach prefix p ∈ P do
25                 I = I ∩ p ;                                  /* intersection of all elements of P */
26             end
27             foreach hyperedge e ∈ E_in(v) ∪ E_out(v) do
28                 label = ∅ ;                                          /* variable for final label of e */
29                 foreach data source d_i ∈ λ(e) do
30                     if prefixes(d_i) ∩ I ≠ ∅ then
31                         label = label ∪ d_i ;
32                     end
33                 end
34                 λ(e) = label
35             end
36         end
37     end
38 end
```



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



(b) Query plan

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

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 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 Evaluation

6.1 Experimental Setup

We used FedBench [10] 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 [7]). Hence, we used all 14 SPARQL endpoint federation queries in our evaluation. In addition, we used the Complex queries (C1-C10) from LargeRDFBench [6] to test 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 index/data summary generation time (if applicable), and (5) index compression ratio.

6.2 Experimental Results

We first measured the compression ratio[6] achieved by each system. We achieve 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. Our 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, our approach 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. We outperform HiBISCuS w.r.t. source selection time as our 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 our proposed approach 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., our approach, ANAPSID, and HiBISCUS select around half (e.g., 70 for our approach 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

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

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			Our Approach			HiBISCuS		
Qry	#T	#A	ST*	ST	#T	#A	ST	#T	#A	ST	#T	#A	ST	#T	#A	ST	#T	#A	ST
CD1	11	27	295	5	11	26	293	3	19	261	11	26	293	4	18	6	4	18	227
CD2	3	27	229	1	3	9	33	3	1	8	3	9	33	3	9	1	3	9	46
CD3	12	45	330	4	12	2	17	5	2	34	12	2	17	5	0	1	5	0	82
CD4	19	45	319	3	19	2	14	5	3	15	19	2	14	5	0	1	5	0	74
CD5	11	36	306	3	11	1	11	4	1	8	11	1	11	4	0	1	4	0	54
CD6	9	36	297	4	9	2	16	9	10	36	9	2	16	8	0	3	8	0	35
CD7	13	36	280	5	13	2	19	6	5	67	13	2	19	6	0	1	6	0	32
LS1	1	18	149	1	1	0	2	1	0	5	1	0	2	1	0	1	1	0	55
LS2	11	27	241	4	11	26	200	15	19	69	11	26	200	4	18	5	7	18	356
LS3	12	45	326	3	12	1	11	5	11	46	12	1	11	5	0	1	5	0	262
LS4	7	63	419	4	7	2	19	7	0	12	7	2	19	7	0	1	7	0	333
LS5	10	54	377	3	10	1	7	7	4	20	10	1	7	7	0	1	8	0	105
LS6	9	45	330	3	9	2	8	5	12	58	9	2	8	5	0	1	7	0	180
LS7	6	45	317	3	6	1	6	5	2	18	6	1	6	6	0	1	6	0	81
T/A	134	549	302	3	134	77	46	80	89	463	134	77	46	70	45	1.7	76	45	137
C1	11	104	455	4	11	1	11	8	1	11	11	1	11	8	0	1	9	0	114
C2	11	104	461	3	11	1	7	8	2	30	11	1	7	8	0	1	9	0	16
C3	21	104	458	4	21	3	12	10	33	79	21	3	12	11	0	1	11	0	200
C4	28	156	580	5	28	0	3	28	32	60	28	0	3	18	0	1	18	0	45
C5	33	104	451	4	33	0	3	8	3	17	33	0	3	10	0	1	10	0	55
C6	24	117	499	4	24	0	2	9	3	14	24	0	2	9	0	1	9	0	445
C7	17	117	502	3	17	2	9	9	5	20	17	2	9	9	0	1	9	0	175
C8	25	143	540	2	25	2	11	11	2	19	25	2	11	11	0	1	11	0	187
C9	16	117	515	317	16	2	17	9	16	52	16	2	17	9	0	1	9	0	170
C10	13	130	535	4	13	0	3	11	6	31	13	0	3	11	0	1	11	0	140
T/A	199	1196	500	4	199	11	7.8	111	103	33.3	199	11	7.8	104	0	1	106	0	154.7

large data queries.

Overall, our results show clearly that our approach outperforms the state of the art on both benchmark datasets.

7 Conclusion and Future Work

We implements innovative solutions for the selection of sources. We showed a join-aware source selection can lead to significant relevant sources reduction without loosing the recall. The complete evaluation result of the query runtimes will be presented in the final report of the 3DFed federation engine.

References

[1] Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *ISWC*, 2011.

[2] Angelos Charalambidis, Antonis Troumpoukis, and Stasinos Konstantopoulos. SemaGrow: Optimizing Federated SPARQL Queries. In *SEMANTICS*, 2015.

[3] Olaf Görlitz and Steffen Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *COLD at ISWC*, 2011.

[4] Gabriela Montoya, Hala Skaf-Molli, and Katja Hose. The Odyssey approach for optimizing federated SPARQL queries. In *ISWC*, 2017.

[5] Ibrahim Abdelazizú Essam Mansourù Mourad Ouzzaniù and Ashraf Aboulnagaù Panos Kalnisú. Lusail: A System for Querying Linked Data at Scale. *VLDB*, 2017.

[6] Muhammad Saleem, Ali Hasnain, and Axel-Cyrille Ngonga Ngomo. LargeRDFBench: a Billion Triples Benchmark for SPARQL Endpoint Federation. *JWS*, 2018.

[7] Muhammad Saleem, Yasar Khan, Ali Hasnain, Ivan Ermilov, and Axel-Cyrille Ngonga Ngomo. A Fine-Grained Evaluation of SPARQL Endpoint Federation Systems. *SWJ*, 2015.

[8] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. HiBISCuS: Hypergraph-based source selection for sparql endpoint federation. In *ESWC*, 2014.

[9] Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, Josiane Xavier Parreira, Helena F Deus, and Manfred Hauswirth. DAW: Duplicate-AWare Federated Query Processing over the Web of Data. In *ISWC*, 2013.

[10] Michael Schmidt, Olaf Görlitz, Peter Haase, Günter Ladwig, Andreas Schwarte, and Thanh Tran. FedBench: a benchmark suite for federated semantic data query processing. In *ISWC*, 2011.

[11] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *ISWC*, 2011.

[12] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.