UPSP: Unique Predicate-based Source Selection for SPARQL Endpoint Federation

Ethem Cem Ozkan[1], Muhammad Saleem[2], Erdogan Dogdu[1], and Axel-Cyrille Ngonga Ngomo[2]

[1] TOBB University of Economics and Technology
{cozkan,edogdu}@etu.edu.tr
[2] Universität Leipzig, IFI/AKSW, PO 100920, D-04009 Leipzig
{lastname}@informatik.uni-leipzig.de

Abstract. Efficient source selection is one of the most important optimization steps in federated SPARQL query processing as it leads to more efficient query execution plan generation. An over-estimation of the data sources will generate extra network traffic by retrieving irrelevant intermediate results. Such intermediate results will be excluded after performing joins between triple patterns. Consequently an over-estimation of sources may result in increased query execution time. Devising triple patterns join-aware source selection approaches has shown to yield great improvement potential. In this work, we present UPSP, a new source selection approach for SPARQL query federation over multiple SPARQL endpoints. UPSP makes use of the subject-subject, subject-object, object-subject, and object-object joins information stored in an index structure to perform efficient triple patterns join-aware source selection. Our evaluation results on FedBench shows that UPSP outperforms state-of-the-art source selection approaches by selecting smaller number of sources (without losing recall) and reducing the query execution times.

Keywords: source pruning, linked data, federated query, HiBISCuS

1 Introduction

Federated SPARQL query engines mostly perform *triple pattern-wise source selection* (TPWSS) for SPARQL query federation over multiple SPARQL endpoints (data sources or only sources for short) [3,5,6,12,13,9,8,2]. The main goal of TPWSS is to find the relevant data sources for each triple pattern in the query. However, it is possible that the sources selected by TPWSS may not contribute to the final result of the query since the results from certain data sources may be excluded after join operations with the results from other data sources in the same query [8]. These overestimated sources increase the network traffic and the overall query processing time by retrieving irrelevant intermediate results. The join-aware source selection approaches [8,1] aim to select those triple pattern-wise sources that contribute to the triple pattern results as well as to the final result of the query. If the source selection could be done better by eliminating those unneeded sources, more efficient query execution plans can be achieved [8].

HiBISCuS [8], a *join-aware approach to TPWSS*, was proposed with the aim of only selecting those sources that actually contribute to the final result set of the query.

This approach makes use of the different *URI authorities*[3] to prune irrelevant sources during the source selection. While HiBISCuS can significantly remove irrelevant sources [8], it fails to prune those sources that share the same URI authority. For example, all `Bio2RDF`[4] sources contain the same URI authority `bio2rd.org`, therefore it is not possible for the HiBISCuS (based on distinct URIs authorities) to perform efficient join-aware source selection .

In this paper, we propose an index-assisted join-aware source selection approach called *Unique Predicate Source Pruning* (UPSP). A *unique predicate* is a predicate that can only be found in one data source and does not participate in subject-subject, subject-object, object-subject, object-object joins between triple patterns. UPSP algorithm is developed as an extension of HiBISCuS [8] to overcome the limitation of the same URI authorities from different data sources. We improve the existing HiBISCuS index by adding the unique predicates information and using this information in an efficient source pruning algorithm (UPSP). Overall, our contributions are as follows:

- We present a new source pruning algorithm called Unique Predicate Source Pruning (UPSP) as an extension of HiBISCuS,
- We extend the existing HiBISCuS index structure with unique predicate information,
- We implemented UPSP on two existing HiBISCuS distributions, which are implemented on two state-of-the-art federated SPARQL query engines, namely SPLENDID [3] and FedX [12],
- We evaluated UPSP algorithm by comparing its performance against the plain HiBISCuS implementations. Our evaluation results shows that we improved the original HiBISCuS source selection method in 4 out of 25 queries in the FedBench dataset and improved query execution time up to 93%.

The structure of this paper is as follows: first we give an overview of the federated query engines and the related work in section 2. Our proposal, UPSP algorithm, and its index creation algorithm are explained in detail in section 4. Then, we present the evaluation results of USPS algorithm against the original HiBISCuS approach in section 5. We conclude and point to future work in section 6.

2 Related Work

The source selection approaches for SPARQL query federation over multiple dataset endpoints can be divided into three main categories [7]:

Index-only Source Selection: This approach only makes use of an index to perform source selection. Besides other dataset statistics, the index typically stores the set of distinct predicates for each each dataset in the federation [6,5]. The source selection algorithm in this case matches the query triple pattern predicate against the set of distinct predicates for each dataset recorded in the index. A dataset is selected as relevant for a triple pattern if it contains the query triple pattern predicate. This approach is in general fast since it only performs index lookups [7]. However, they are less efficient as the

[3] URI: `http://tools.ietf.org/html/rfc3986`
[4] Bio2RDF: `http://download.bio2rdf.org/release/2/release.html`

source selection is only based on triple pattern predicates without considering the subject and object of the triple pattern [7]. In addition, the result completeness (100% recall) must be ensured by keeping the index up-to-date. Well-known examples of index-only source selection approaches are DARQ [6] and ADERIS [5].

Index-free Source Selection: This approach does not make use of any pre-stored index and can thus always computes complete and up-to-date records. The source selection algorithm is performed by using SPARQL `ASK` queries, by sending a SPARQL `ASK` query for each query triple pattern in the federated query to each of the datasets (i.e., SPARQL endpoint) and select those datasets that return true for the submitted `ASK` queries. This approach is more efficient as compared to previous approach since, beside predicates, it also matches triple pattern subjects and objects [7]. However, the query execution time can be larger depending upon the number of SPARQL `ASK` request used during the source selection [7]. FedX [12] is a well-known example of index-free source selection.

Hybrid Source Selection: This approach makes use of both index and SPARQL `ASK` queries for source selection. It has the advantages of the previous two approaches. Well-known examples are HiBISCuS [8], DAW [9], ANAPSID [1], SemaGrow [2], TopFed [10], SAFE [4], and SPLENDID [3].

All of the above mentioned contributions (except HiBISCuS, ANAPSID) perform simple triple pattern-wise source selection and do not consider the joins between query triple patterns during the source selection. Consequently they greatly overestimate the set of relevant sources that actually contribute to the final resultset of the query [7]. Further, it has be shown that the join-aware source selection (as performed in HiBISCuS, ANAPSID) has great potential to reduce the network traffic and query execution time.

In this paper, we present a hybrid join-aware source selection algorithm, which is designed to be an extension of HiBISCuS. Our goal is to improve HiBISCuS hybrid source selection method. In particular, when the same URI authorities are distributed among different data sources.

3 Preliminaries

In the following, we present some of the concepts and notation that are used throughout this paper. We reused some of the definitions and concepts from HiBISCuS [8] for better understanding. RDF resources are identified by using a Unified Resource Identifier (URI). Each URI has a generic syntax consists of a hierarchical sequence of components namely the *scheme*, *authority*, *path*, *query*, and *fragment*[5]. For example, the prefix $ns1 = <http://auth1/schema/>$ used in Figure 1 consist of scheme $http$, authority $auth1$, and path $schema$. The details of the remaining two components are out of the scope of this paper. In the rest of the paper, we jointly refer to the first two components (path, authority) as *authority* of a URI.

The standard for querying RDF is SPARQL[6]. The result of a SPARQL query is called its *result set*. Each element of the result set of a query is a set of *variable bindings*.

[5] URI syntax: `http://tools.ietf.org/html/rfc3986`
[6] `http://www.w3.org/TR/rdf-sparql-query/`

Federated SPARQL queries are defined as queries that are carried out over a set of sources D = $\{d_1, \ldots, d_n\}$. Given a SPARQL query q, a source $d \in D$ is said to *contribute* to query q if at least one of the variable bindings belonging to an element of q's result set can be found in d.

Definition 1 (Relevant source Set). *A source $d \in D$ is* relevant *(also called* capable*) for a triple pattern $tp_i \in TP$ if at least one triple contained in d matches tp_i.*[7] *The relevant source set $R_i \subseteq D$ for tp_i is the set that contains all sources that are relevant for that particular triple pattern.*

For example, the set of relevant sources for the triple pattern $<?s, \texttt{cp:p2}, ?p>$ is $\{d_1, d_2\}$. It is possible that a relevant source for a triple pattern does not *contribute* to the final result set of the complete query q. This is because the results computed from a particular source d for a triple pattern tp_i might excluded while performing *joins* with the results of other triple patterns contained in the query q.

Definition 2 (Optimal source Set). *The optimal source set $O_i \subseteq R_i$ for a triple pattern $tp_i \in TP$ contains the relevant sources $d \in R_i$ that actually contribute to computing the complete result set of the query.*

For example, the set of optimal sources for the triple pattern $<?s, \texttt{cp:p2}, ?v2>$ is $\{d_3\}$, while the set of relevant sources for the same triple pattern is $\{d_1, d_2\}$. Formally, the problem of TPWSS can then be defined as follows:

Definition 3 (Problem Statement). *Given a set D of sources and a query q, find the optimal set of sources $O_i \subseteq D$ for each triple pattern tp_i of q.*

Most of the source selection approaches [3,5,6,12,13] used in SPARQL endpoint federation systems only perform TPWSS, i.e., they find the set of relevant sources R_i for individual triple patterns of a query and do not consider computing the optimal source sets O_i. In this paper, we present an index-assisted approach for (1) the time-efficient computation of relevant source set R_i for individual triple patterns of the query and (2) the approximation of O_i out of R_i. HiBISCuS approximates O_i by determining and removing irrelevant sources from each of the R_i. We denote our approximation of O_i by RS_i. HiBISCuS and our extended approach UPSP relies on directed labelled hypergraphs (DLH) to achieve this goal. In the following, we present our formalization of SPARQL queries as DLH. Subsequently, we show how we make use of this formalization to approximate O_i for each tp_i.

3.1 Queries as Directed Labelled Hypergraphs (DLH)

The basic intuition behind HiBISCuS, and also our extended approach UPSP, is that each of the Basic Graph Pattern (BGP)[8] in a query can be executed separately. Thus, in the following, we will mainly focus on how the execution of a single BGP can be

[7] The concept of matching a triple pattern is defined formally in the SPARQL specification found at `http://www.w3.org/TR/rdf-sparql-query/`

[8] BGP: `http://www.w3.org/TR/sparql11-query/#BasicGraphPatterns`

optimized. The representation of a query as Directed Labelled Hypergraphs (DLH) is the union of the representations of its BGPs. Note that the representations of BGPs are kept disjoint even if they contain the same nodes to ensure that the BGPs are processed independently. The DLH representation of a BGP is formally defined as follows [8]:

Definition 4. *Each Basic Graph Pattern BGP_i of a SPARQL query can be represented as a DLH $HG_i = (V, E, \lambda_e, \lambda_{vt})$, where*

1. *$V = V_s \cup V_p \cup V_o$ is the set of all vertices of HG_i, V_s is the set of all subjects in HG_i, V_p the set of all predicates in HG_i and V_o the set of all objects in HG_i;*
2. *E =$\{e_1,\ldots, e_t\}\subseteq V^3$ is a set of directed hyperedges (short: edge). Each edge e= (v_s,v_p,v_o) emanates from the triple pattern $<v_s, v_p, v_o>$ in BGP_i. We represent these edges by connecting the head vertex v_s with the tail hypervertex (v_p, v_o). In addition, we use $E_{in}(v) \subseteq E$ and $E_{out}(v) \subseteq E$ to denote the set of incoming and outgoing edges of a vertex v;*
3. *$\lambda_e : E \mapsto 2^D$ is a hyperedge-labelling function. Given a hyperedge $e \in E$, its edge label is a set of sources $R_i \subseteq D$. We use this label to the sources that should be queried to retrieve the answer set for the triple pattern represented by the hyperedge e;*
4. *λ_{vt} is a vertex-type-assignment function. Given an vertex $v \in V$, its vertex type can be 'star', 'path', 'hybrid', or 'sink' if this vertex participates in at least one join. A 'star' vertex has more than one outgoing edge and no incoming edge. 'path' vertex has exactly one incoming and one outgoing edge. A 'hybrid' vertex has either more than one incoming and at least one outgoing edge or more than one outgoing and at least one incoming edge. A 'sink' vertex has more than one incoming edge and no outgoing edge. A vertex that does not participate in any join is of type 'simple'.*

An example of the DLH representation of a query can be found in Figures 2 to 5. Consider the query given in Figure 2, containing two triple patterns $<?s1\ cp:p1\ ?v1>$ and $<?s1\ cp:p4\ ?v2>$. Dataset $d1$ is the single relevant source for the first triple pattern and datasets $d1$, $d2$, $d3$ which can be found in Figure 1 are the relevant sources for the second triple pattern. These triple patterns form a subject-subject join on variable $?s1$. Consider the first triple pattern (represented as hyperedge (s1, {cp:p1, v1})), $s1$ is the head of the hyperedge and $\{cp : p1, v1\}$ (represented in a box) is the tail of the hyperedge. The relevant source, i.e., dataset $d1$ is the label of the hyperedge. Note the vertex $s1$ represent a "star" node because it has only outgoing hyperedges and no incoming hyperedge. We can now formally define our problem statement as follows:

Definition 5 (Problem Re-definition). *Given a query q represented as a set of hypergraphs $\{HG_1, \ldots, HG_x\}$ (each HG representing a BGP in q), find the labelling of the hyperedges of each hypergraph HG_i that leads to an optimal source selection.*

4 Unique Predicate Source Pruning (UPSP)

In this section, we explain our approach for source selection called Unique Predicate Source Pruning (UPSP) in detail.

Dataset *d1*

```
@prefix ns1:<http://auth1/schema/>
@prefix ns2:<http://auth2/schema/>
@prefix ns3:<http://auth3/schema/>
@prefix ns4:<http://auth4/schema/>
@prefix ns1_1:<http://auth11/schema/>
@prefix ns1_2:<http://auth12/schema/>
@prefix cp:<http://common/schema/>

ns1:s1      cp:p1     ns:o10
ns3:s2      cp:p2     ns2:o5
ns1:s1      cp:p4     ns4:o7
ns2:s3      cp:p3     s2:o1
ns4:s4      cp:p2     ns2:s3
ns1_1:s2    cp:p1     ns1_2:s1
ns1_1:s1    cp:p3     "o30"
```

Dataset *d2*

```
@prefix ns1:<http://auth1/schema/>
@prefix ns2:<http://auth2/schema/>
@prefix ns3:<http://auth3/schema/>
@prefix ns1_1:<http://auth11/schema/>
@prefix ns1_2:<http://auth12/schema/>
@prefix cp:<http://common/schema/>

ns1_1:s1    cp:p2     ns1:o1
ns3:s2      cp:p5     ns3:o2
ns3:o2      cp:p4     ns1:s1
ns3:o2      cp:p7     ns1:o1
ns1_2:s1    cp:p8     "o20"
ns1_2:s1    cp:p5     "o21"
ns2:s3      cp:p2     ns1_2:s1
```

Dataset *d3*

```
@prefix ns1:<http://auth1/schema/>
@prefix ns2:<http://auth2/schema/>
@prefix ns4:<http://auth4/schema/>
@prefix ns1_1:<http://auth11/schema/>
@prefix ns1_2:<http://auth12/schema/>
@prefix cp:<http://common/schema/>

ns2:s3      cp:p6     ns1_2:o1
ns2:s3      cp:p4     ns2:o5
ns4:s2      cp:p2     ns1_2:o1
ns1_2:o1    cp:p7     ns1:s1
ns2:o5      cp:p6     ns1_1:o1
```

Fig. 1: Unique Predicate Motivating Example

4.1 Unique Predicates

Unique predicate is a predicate that can only be found in one data source, and further is not involved in subject-subject, subject-object, object-subject, object-object relations (explained below). Accordingly, there are four types of unique predicates. These are explained below using examples in Figure 1.

- Subject-Subject: If a predicate is unique and the subjects of the predicate are not used in other datasets as subjects then this predicate is a *subject-subject unique predicate*. For example in Figure 1 predicate $cp{:}p1$ is a subject-subject unique predicate. This is because cp:p1 is only found in dataset $d1$ and its subjects, which are $ns1{:}s1$ and $ns1_1{:}s2$, are not used as subjects in other datasets.
- Subject-Object: If a predicate is unique and the subjects of the predicate are not used in other datasets as objects then this predicate is a *subject-object unique predicate*. For example in Figure 1 predicate cp:p3 is a subject-object unique predicate. This is because cp:p3 is only found in dataset $d1$ and its subjects, which are ns2:s3 and ns1_1:s1, are not used as objects in other datasets. However predicate cp:p3 is not subject-subject unique because subject ns2:s3, which is one of its subjects, is used as subject in dataset $d2$. Therefore cp:p3 is not subject-subject unique.
- Object-Subject: If a predicate is unique and the objects of the predicate are not used in other data soruces as subjects then this predicate is a *object-subject unique predicate*. For example in Figure 1 predicate cp:p5 in dataset $d2$ is object-subject unique because cp:p5 is only found in dataset $d2$ and its object, which is only ns3:o2, is not used as subject in other datasets.
- Object-Object: If a predicate is unique and the objects of the predicate are not used in other data sources as objects then this predicate is a *object-object unique predicate*. For example in Figure 1 $cp{:}p5$, which is only found in dataset $d2$, is object-object unique. This is because $cp{:}p5$ is only found in dataset $d2$ and its object, which is only $ns3{:}o2$, is not used as object in other datasets.

Each predicate can be of more than one unique predicate types as seen in the examples. We propose to utilize unique predicate information for source pruning in

Listing 1.1: UPSP Index Example

```
a ds:Service ;
     ds:url   <http://d1.ecozkan.com/sparql> ;
     ds:capability
          [ ds:predicate  <http://common/schema/p1>;
            ds:sbjAuthority <http://auth1>, <http://auth11> ;
            ds:objAuthority <http://auth1>, <http://auth12> ;
            ds:ssUnique true ; ] ;
.
[] a ds:Service ;
     ds:url   <http://d2.ecozkan.com/sparql> ;
          ds:capability
          [ ds:predicate  <http://common/schema/p5>;
            ds:sbjAuthority <http://auth3>, <http://auth12> ;
            ds:objAuthority <http://auth3> ;
            ds:osUnique true ;
            ds:ooUnique true ; ] ;
.
```

federated SPARQL query optimization so that data sources are eliminated from query processing if they do not contribute to the final result of a query. To be able to decide whether to prune or not to prune a data source from query processing, we build an index to store all unique predicate information that is later used in the source pruning algorithm.

4.2 Index Creation

The index structure we build is based on HiBISCuS index structure, i.e., we store distinct predicates for each data source. Further, for each distinct predicate we store the subject and object authorities [8]. We keep the same index structure and extend with the requirements of our proposed UPSP algorithm, so that both source selection algorithms will run together.

UPSP extends HiBISCuS index by adding four new fields to store the information about unique predicate types. These fields are ssUnique, soUnique, osUnique, and ooUnique; they represent subject-subject, subject-object, object-subject and object-object unique predicate types respectively. By default these fields' values are "false" and they are not kept in the index. They are saved in the index only if they are "true" to keep the index structure compact. A sample index can be seen in Listing 1.1. As seen in the figure, only true unique predicate types are listed for each predicate, otherwise they are not mentioned (considered false). For example, the predicate $http://common/schema/p5$ is osUnique and ooUnique, but not soUnique and not ssUnique. UPSP source pruning algorithm checks this index only to find if a unique predicate type is true, otherwise it is assumed false.

Algorithm 1 shows the unique predicate index building procedure. It checks all predicates in all dataset endpoints (line 1) and first assumes it is subject-subject, subject-object, object-subject, and object-object unique (line 2-5). Then, it finds subject and object authorities of the predicate (line 7-8) and checks if the predicate is unique among other dataset endpoints (line 9). If it is unique then the algorithm checks the current predicate against other predicates to find if it is subject-subject, subject-object, object-subject, and object-object unique (Line 10-24). As explained before, we first compare the authorities of predicates before actually executing costly queries. If the intersection

Algorithm 1 Unique Predicate Index Creation Algorithm

Require: datasets D // list of available datasets
1: **for** each $d_i \in D$ **do**
2: ssUnique = true;
3: soUnique = true;
4: osUnique = true;
5: ooUnique = true;
6: **for** each $p_i \in predicates(d_i)$ **do**
7: $sbjAuthP_i = sbjAuth(p_i, d_i)$
8: $objAuthP_i = objAuth(p_i, d_i)$
9: **if** $isUnique(p_i)$ **then**
10: **for** each $d_j \in D \wedge d_i \neq d_j$ **do**
11: **for** each $p_j \in predicates(d_j)$ **do**
12: $sbjAuthP_j = sbjAuth(p_j, d_j)$
13: $objAuthP_j = objAuth(p_j, d_j)$
14: **if** $sbjAuthP_i \cap sbjAuthP_j \neq \emptyset$ **then**
15: ssUnique = $checkSubjectSubjectUnique(p_i, p_j, d_i, d_j)$
16: **end if**
17: **if** $sbjAuthP_i \cap objAuthP_j \neq \emptyset$ **then**
18: soUnique = $checkSubjectObjectUnique(p_i, p_j, d_i, d_j)$
19: **end if**
20: **if** $objAuthP_i \cap sbjAuthP_j \neq \emptyset$ **then**
21: osUnique = $checkObjectSubjectUnique(p_i, p_j, d_i, d_j)$
22: **end if**
23: **if** $objAuthP_i \cap objAuthP_j \neq \emptyset$ **then**
24: ooUnique = $checkObjectObjectUnique(p_i, p_j, d_i, d_j)$
25: **end if**
26: **end for**
27: **end for**
28: **end if**
29: **end for**
30: **end for**

of authorities is not empty, then the algorithm sends the relevant ASK query to find if the predicates are subject-subject, subject-object, object-subject, object-object unique between each other. ASK queries are created using SPARQL 1.1 standard. Template ASK queries used in the algorithm are listed bellow.

- Subject-Subject:
  `ASK { SERVICE <e2> {?s <p2> ?o2.} ?s <p1> ?o1.}`
  This query returns true if the subjects of p1 are used as the subjects of p2.
- Subject-Object:
  `ASK { SERVICE <e2> {?s1 <p2> ?s.} ?s <p1> ?o1.}`
  This query returns true if the subjects of p1 are used as the objects of p2.
- Object-Subject:
  `ASK { SERVICE <e2> {?o1 <p2> ?s.} ?s1 <p1> ?o1.}`
  This query returns true if the objects of p1 are used as the subjects of p2.
- Object-Object:
  `ASK { SERVICE <e2> {?s2 <p2> ?o.} ?s1 <p1> ?o.}`
  This query returns true if the objects of p1 are used as the objects of p2.

In the algorithm, each predicate is checked against other predicates in all dataset endpoints to find for example if it is subject-subject unique and so on. If it is found subject-subject unique at the end, then the index field ssUnique is set to true, otherwise no index entry is put (to save space) for that predicate.

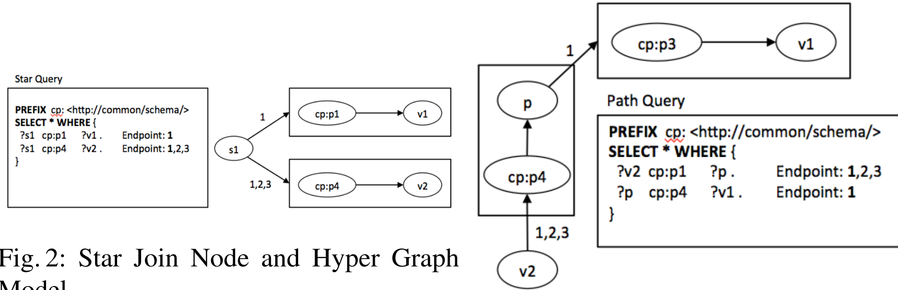

Fig. 2: Star Join Node and Hyper Graph Model

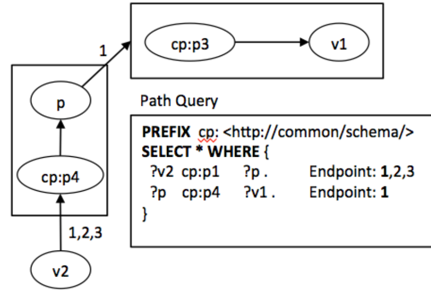

Fig. 3: Path Join Node and Hyper Graph Model

We first implemented the index creation algorithm as a single threaded process. This implementation took 60 hours to run to create the index, which includes both HiBISCuS original and extended information, against FedBench datasets. We also implemented a multi-threaded version and the execution time is reduced to 36 hours that can be further improved with multi-nodes on a cluster. We should also note that this is a one-time process and rarely changes.

4.3 The Pruning Approach

In order to perform join-aware source selection, UPSP makes use of the four – 'star', 'path', 'hybrid', 'sink'– different types of join nodes, according to the DLH representation explained in Section 3.1. Below are the examples of the different join types and the pruning steps we take in UPSP algorithm.

- Star join node: As mentioned before, a star join node has only outgoing hyperedges and no incoming hyperedge. An example of such a join node is given Figure 2 where the two triple patterns make a join on a common subject ($?s1$). Only dataset $d1$ contributes to the first triple pattern and dataset $d1$, $d2$, and $d3$ contribute to the second triple pattern. The predicate $cp{:}p1$ of the first triple pattern is subject-subject unique, which means this triple pattern cannot make a subject-subject join with an external data source (dataset $d2$ and $d3$ are external data soruces for dataset $d1$). Thus, we can prune the external data sources from the second triple pattern, their contributing results will not be used in the final join query. As a result of this pruning, only dataset $d1$ will be selected and used in the computations of these two triple patterns.
- Path join node: A path join node contains exactly one incoming hyperedge and one outgoing hyperedge. An example of a such node is given in Figure 3 where the object ($?p$) of the first triple pattern is used as such of the second triple pattern thus forming a subject-object join. In path join nodes, the UPSP algorithm checks if any of the predicates is subject-object or object-subject unique and prune the external data sources from the other triple pattern. In Figure 3, $cp{:}p3$ is a subject-object unique predicate, the UPSP algorithm can prune all external sources, namely dataset

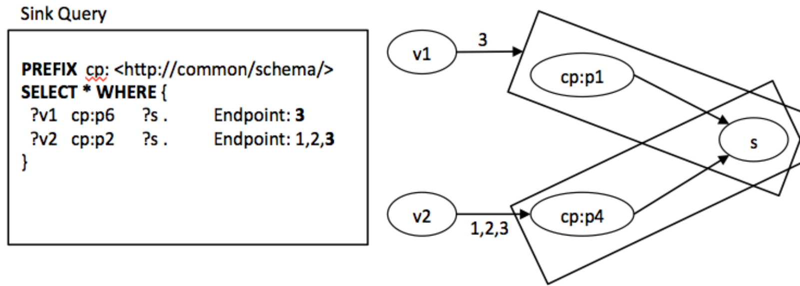

Fig. 4: Sink Join Node and Hyper Graph Model

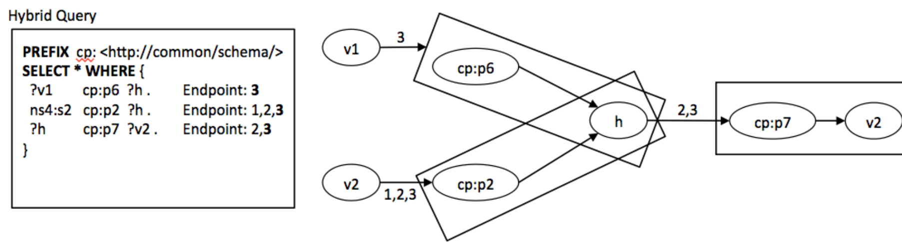

Fig. 5: Hybrid Join Node and Hyper Graph Model

$d2$ and $d3$, from the incoming hyperedeges of $?p$ node. Thus dataset $d1$ is the only source finally selected as capable for both of the triple patterns.

- Sink join node: Sink join nodes only contain incoming hyperedges and no outgoing hyperedges. Figure 4 depicts a sink join node example. A sink join node is created as a result of the object-object joins between two triple patterns. If one of the predicates in sink join is object-object unique, then we can prune all external sources from the other hyperedeges. This is because if a predicate is object-object unique, it cannot join with the objects in other datasets. In Figure 4 $cp{:}p1$ is an object-object unique predicate. Therefore, the results of $\{?v1\ cp{:}p1\ ?s\}$ triple pattern cannot join with the other datasets' results. The UPSP algorithm simply prunes external sources in $cp{:}p4$'s hyperedge, namely datasets $d1$ and $d2$.
- Hybrid join node: A node which has at least one incoming and more than one outgoing hyperedge or at least one outgoing and more than one incomoing hyperedges is called hybrid node. Figure 5 depicts a hybrid node. A hybrid join is a combination of sink, path, and/or star queries. Algorithm basically applies the same rules applied to sink, path, and star queries. In Figure 5 predicate $cp{:}p6$ is object-object and object-subject unique. Therefore, we can prune all external sources from outgoing and incoming edges, namely datasets $d1$ and $d2$. Only the dataset $d3$ is selected as the only capable source for the three triple patterns. It is important to note that if a query does not contain unique predicates then the UPSP source selection results in the same source selection as performed by HiBISCuS.

The UPSP algorithm is listed in Algorithm 2. Algorithm loops over each hypergraph and all of its nodes in the query. However, as mentioned before, the pruning is performed

Algorithm 2 Unique Predicate Source Pruning (UPSP) Algorithm

```
Require: DHG // Hypergraph of query
 1: for each HG ∈ DHG do
 2:   for each v ∈ vertices(HG) do
 3:     if isStarNode(v) then
 4:       for each e ∈ v.outEdge do
 5:         if e.predicate is ssUnique then
 6:           pruneExternal(e, v.outEdge)
 7:         end if
 8:       end for
 9:     else if isPathNode(v) then
10:       eOut = v.outEdge[0]
11:       eIn = v.inEdge[0]
12:       if eOut.predicate is soUnique then
13:         pruneExternal(e, v.inEdge)
14:       end if
15:       if eIn.predicate is osUnique then
16:         pruneExternal(e, v.outEdge)
17:       end if
18:     else if isSinkNode(v) then
19:       for each e ∈ v.inEdge do
20:         if e.predicate is ooUnique then
21:           pruneExternal(e, v.inEdge)
22:         end if
23:       end for
24:     else if isHybridNode(v) then
25:       for each eIn ∈ v.inEdge do
26:         if eIn.predicate is osUnique then
27:           pruneExternal(eIn, v.outEdge)
28:         end if
29:         if eIn.predicate is ooUnique then
30:           pruneExternal(eIn, v.inEdge)
31:         end if
32:       end for
33:       for each eOut ∈ v.outEdge do
34:         if eOut.predicate is soUnique then
35:           pruneExternal(eOut, v.inEdge)
36:         end if
37:         if eOut.predicate is ssUnique then
38:           pruneExternal(eOut, v.outEdge)
39:         end if
40:       end for
41:     end if
42:   end for
43: end for
```

at join nodes only. If a node is a star node and if one of the outgoing predicates of the star node is a subject-subject unique predicate, then it prunes all external sources from the other outgoing hyperedeges (lines 3-8). If a node is a path node and if the outgoing node is a subject-object unique predicate, then it prunes all external sources from the incoming hyperedeges (lines 9-14). And for the path node, if the incoming node is an object-subject unique predicate, then the algorithm prunes all external sources from the outgoing hyperedge (lines 15-17). If a node is a sink node and if one of the incoming nodes is an object-object unique predicate, then it prunes all external sources from the incoming hyperedge (line 18-23). If a node is a hybrid node and if an incoming node is an object-subject unique predicate, then it prunes all external sources from the outgoing hyperedeges (line 24-28). If an incoming node of the hybrid node is object-object unique, then it prunes external sources from the incoming hyperedeges (line 29-31). If an outgoing node of the hybrid node is subject-object unique, then it prunes all external sources from the incoming hyperedeges (line 34-36). If an outgoing node

of the hybrid node is subject-subject unique, then it prunes external sources from the outgoing hyperedeges (line 37-39).

5 Evaluation

In this section we first describe our test environment. Then, we present our evaluation results in detail.

5.1 Experimental Setup

We used FedBench [11] dataset for evaluation. We loaded all 9 of FedBench datasets on virtual machines created on Amazon Web Services (AWS) EC2 Virtual Server Hosting[9]. AWS EC2 offers dynamic server configuration. All experiments are executed on m3.xlarge type server instances, each having Intel Xenon x5 4-core CPUs and 15GB main memory. Since our tests are executed among AWS servers, the network communication cost was negligible. Each Fedbech query is executed 5 times and the execution times are averaged after the minimum and the maximum execution times are removed.

We extended two federated query engines with UPSP algorithm, namely FedX [12] and SPLENDID [3] with HiBISCuS extensions already implemented. We compared our results with the original HiBISCuS system in Index-Dominant and ASK-Dominant modes [8]. For each query we measured: (1) the total number of triple pattern-wise (TPW) sources selected, (2) the average source selection time (msec), and (3) the average query execution time (msec).

5.2 Experimental Results

FedX and SPLENDID implementations of UPSP are tested with FedBench dataset and the results are presented in Tables 1 and 2. Bold printed results show the results for the queries where UPSP has improved the execution time (#UET column) and pruned more sources (#US column).

FedX. We compared UPSP with the original HiBISCuS algorithmn. We run our tests in both Index-Dominant and ASK-Dominant modes. We improved source selection in LS5, LS7, LD6, and LD11 queries (Table 1). We also improved execution time substantially in LD6 and LD7 queries. LS5 and LS7 shows no impromevent in execution time even though their source selection improved (1 datasource eliminated in each). This is because the eliminated endpoints are not too many. On the other hand, for queries LD6 and LD11, 3 and 2 datasources are pruned, respectively. The execution time is improved substantially, especially for LD11 the execution time is reduced by one third. In these experiments for FedX, UPSP algorithm improved HiBISCuS source selection in 4 out of 25 queries (16%). Execution time results are not available in query LS6 because the

[9] Amazon EC2 https://aws.amazon.com/ec2

Table 1: FedX experimental results. #S: Original number of sources, #HS: Number of sources after HiBISCuS pruning, #US: Number of sources after UPSP pruning, HET: HiBISCuS Average Execution Time (msec), UET: UPSP Average Execution Time (msec), HST: HiBISCuS Average Source Selection Time (msec), UST: UPSP Average Source Selection Time (msec)

FedX																
	ASK Dominant							**Index Dominant**							
Query	**#S**	**#HS**	**#US**	**HET**	**UET**	**HST**	**UST**	**#S**	**#HS**	**#US**	**HET**	**UET**	**HST**	**UST**
CD1	11	4	4	307	302	73	59	16	12	12	669	681	321	322
CD2	3	3	3	188	189	13	14	7	3	3	373	325	84	81
CD3	12	5	5	304	312	31	33	12	5	5	533	525	125	101
CD4	20	5	5	408	385	56	47	20	5	5	618	618	184	188
CD5	11	4	4	286	278	26	26	11	4	4	435	414	94	77
CD6	10	8	8	860	851	26	25	10	8	8	1088	1121	90	94
CD7	13	6	6	652	642	28	33	13	6	6	914	855	109	83
LS1	1	1	1	394	386	16	16	1	1	1	563	538	39	30
LS2	11	7	7	407	391	70	63	11	7	7	696	733	255	274
LS3	12	5	5	3970	3892	30	34	12	5	5	4205	4088	100	117
LS4	7	7	7	197	195	16	16	7	7	7	313	306	48	43
LS5	10	8	**7**	1948	1946	26	31	10	8	**7**	2195	2216	78	102
LS6	9	7	7	-	-	-	-	9	7	7	-	-	-	-
LS7	6	6	**5**	1793	1823	21	27	6	6	**5**	1986	2146	65	87
LD1	11	3	3	290	281	24	20	11	3	3	451	416	79	65
LD2	3	3	3	230	226	13	13	3	3	3	372	347	38	36
LD3	19	4	4	277	253	35	28	19	4	4	421	399	104	92
LD4	5	5	5	189	189	14	15	5	5	5	287	282	44	41
LD5	5	3	3	179	182	20	20	5	3	3	258	315	46	66
LD6	14	8	**5**	314	**298**	35	36	14	7	**5**	496	**436**	112	68
LD7	4	4	4	731	726	17	15	4	4	4	999	988	48	45
LD8	15	5	5	462	463	35	34	15	5	5	655	648	102	104
LD9	3	3	3	164	165	12	13	5	3	3	259	248	46	57
LD10	10	3	3	352	353	25	30	10	3	3	515	489	77	78
LD11	21	7	**5**	1286	**385**	38	31	21	7	**5**	1457	**583**	122	92

query did not terminate for a long time and therefore manually cancelled, but the source selection result is reported (no change).

We should also note that even though FedBench results did not show much improvement in execution times after prunings, it is very much dependent on the dataset where the pruning is executed. This is shown clearly in query LD11, where the execution time reduced from 1286 msec to 385 msec (reduced to approx. 1/3). Therefore, our proposed pruning algorithm is very promising for large and diverse datasets in real-life settings.

SPLENDID. We compared UPSP with the original HiBISCuS algorithm for SPLENDID implementation as well. We ran our tests in both Index-Dominant and ASK-Dominant modes. Results in Table 2 show that we improved source selection in LS5, LD6, LS7, and LD11 queries. Note this is exactly the same improvement reported in FedX. The reason for this is that both SPLENDID and FedX select exactly the same triple pattern-wise sources. For LS7 the query execution did not terminate (therefore,

Table 2: SPLENDID Experimental Results. #S: Original number of sources, #HS: number of resources after HiBISCuS pruning, #US: Number of sources after UPSP pruning, HET: HiBISCuS Average Execution Time (msec), UET: UPSP Average Execution Time (msec), HST: HiBISCuS Average Source Selection Time (msec), UST: UPSP Average Source Selection Time (msec)

SPLENDID																
	ASK Dominant								Index Dominant							
Query	**#S**	**#HS**	**#US**	**HET**	**UET**	**HST**	**UST**	**#S**	**#HS**	**#US**	**HET**	**UET**	**HST**	**UST**
CD1	11	4	4	371	374	129	138	16	12	12	767	729	270	292
CD2	3	3	3	226	240	11	11	7	3	3	344	332	63	55
CD3	12	5	5	340	375	27	26	12	5	5	452	463	82	73
CD4	20	5	5	297	293	43	43	20	5	5	432	441	162	140
CD5	11	4	4	298	322	32	27	11	4	4	427	437	80	67
CD6	10	8	8	8764	8694	21	29	10	8	8	8979	8780	69	66
CD7	13	6	6	2344	2345	34	22	13	6	6	2447	2391	73	73
LS1	1	1	1	387	449	8	10	1	1	1	460	468	21	23
LS2	11	7	7	667	775	99	127	11	7	7	1026	924	263	209
LS3	12	5	5	6090	5964	31	24	12	5	5	6127	6021	89	92
LS4	7	7	7	272	280	17	15	7	7	7	398	380	34	33
LS5	10	8	**7**	16105	**13660**	24	29	10	8	**7**	16033	**14040**	67	81
LS6	9	7	7	1817	1828	20	26	9	7	7	1983	1977	67	71
LS7	6	6	**5**	-	-	-	-	6	6	**5**	-	-	-	-
LD1	11	3	3	334	322	18	21	11	3	3	370	386	60	55
LD2	3	3	3	286	293	12	11	3	3	3	329	330	33	31
LD3	19	4	4	306	300	33	29	19	4	4	378	361	82	62
LD4	5	5	5	261	220	14	14	5	5	5	256	252	31	31
LD5	5	3	3	223	249	14	18	5	3	3	248	267	43	37
LD6	14	7	**5**	588	**662**	36	36	14	7	**2**	743	**551**	105	**75**
LD7	4	4	4	8403	8277	14	16	4	4	4	8510	8516	36	35
LD8	15	5	5	968	1223	25	25	15	5	5	945	1001	86	72
LD9	3	3	3	219	219	13	12	5	3	3	240	240	43	42
LD10	10	3	3	276	285	24	29	10	3	3	386	365	58	64
LD11	21	7	**5**	5578	**417**	33	33	21	7	**5**	5823	**474**	123	**64**

cancelled) and thus the execution times are not reported. LS5 and LD11 queries show improvements in execution times, in the case of LD11 quite substantially (93% less) due to the eliminated datasource sizes being quite large. Again our claim is proven that data source selection has the potential to improve federated queries quite considerably in real-life settings with large and diverse datasets.

Overall, UPSP improves (selected fewer sources) HIBISCuS source selection in 4 out of 25 FedBench queries both for FedX and SPLENDID setup. By looking into the details of HIBISCuS source selection, we have found out that there are 13 out of 25 queries for which the HIBISCuS total triple pattern-wise selected sources is equal to the number of triple patterns in the query, i.e., a single source is only selected for each query triple pattern, thus no further source pruning was possible in these queries. To this end, UPSP improves HIBISCuS in 4 out of 12 queries (i.e., 33%) for which the improvement was possible.

6 Conclusion and Future Work

In this paper we presented the UPSP algorithm, a unique predicate based source pruning approach designed to be an extension of HiBISCuS [8]. The UPSP algorithm uses an extended index structure on top of HiBISCuS and prunes irrelevant data sources according to the query and unique predicate type. Our approach improves source selection by eliminating more data sources than HiBISCuS and experimental results on FedBench dataset shows that federated query execution times can be reduced up to 93% (LD11 query in SPLENDID implementation). As for future work, we will first improve our index creation algorithm by using big data technologies like Hadoop. We also intend to further optimize federated query execution with better methods.

7 Acknowledgments

This article was funded by the EU H2020 HOBBIT (GA No. 688227), Eurostars projects DIESEL (E!9367), QAMEL (E!9725), and BMWi project SAKE (01MD15006E).

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 *Proceedings of the 11th International Conference on Semantic Systems*, SEMANTICS '15, pages 121–128, New York, NY, USA, 2015. ACM.
3. O. Görlitz and S. Staab. Splendid: Sparql endpoint federation exploiting void descriptions. In *COLD at ISWC*, 2011.
4. Y. Khan, M. Saleem, A. Iqbal, M. Mehdi, A. Hogan, P. Hasapis, A.-C. N. Ngomo, S. Decker, and R. Sahay. Safe: Policy aware sparql query federation over rdf data cubes. In *SWAT4LS*, 2014.
5. S. Lynden, I. Kojima, A. Matono, and Y. Tanimura. Aderis: An adaptive query processor for joining federated sparql endpoints. In *OTM*. 2011.
6. B. Quilitz and U. Leser. Querying distributed rdf data sources with sparql. In *ESWC*, 2008.
7. M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. N. Ngomo. A fine-grained evaluation of sparql endpoint federation systems. *Semantic Web Journal*, 2014.
8. M. Saleem and A.-C. N. Ngomo. Hibiscus: Hypergraph-based source selection for sparql endpoint federation. In *ISWC*. Springer, 2014.
9. 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.
10. M. Saleem, S. S. Padmanabhuni, A.-C. N. Ngomo, A. Iqbal, J. S. Almeida, S. Decker, and H. F. Deus. Topfed: Tcga tailored federated query processing and linking to lod. *Journal of Biomedical Semantics*, 2014.
11. 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.
12. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *ISWC*, 2011.
13. X. Wang, T. Tiropanis, and H. C. Davis. Lhd: Optimising linked data query processing using parallelisation. In *LDOW at WWW*, 2013.