The Odyssey Approach for Optimizing Federated SPARQL Queries

Gabriela Montoya, Hala Skaf-Molli, Katja Hose

The *Odyssey* Approach for Optimizing Federated SPARQL Queries

Gabriela Montoya[1], Hala Skaf-Molli[2], and Katja Hose[1]

[1] Aalborg University, Denmark {gmontoya,khose}@cs.aau.dk
[2] Nantes University, France hala.skaf@univ-nantes.fr

Abstract. Answering queries over a federation of SPARQL endpoints requires combining data from more than one data source. Optimizing queries in such scenarios is particularly challenging not only because of (i) the large variety of possible query execution plans that correctly answer the query but also because (ii) there is only limited access to statistics about schema and instance data of remote sources. To overcome these challenges, most federated query engines rely on heuristics to reduce the space of possible query execution plans or on dynamic programming strategies to produce optimal plans. Nevertheless, these plans may still exhibit a high number of intermediate results or high execution times because of heuristics and inaccurate cost estimations. In this paper, we present *Odyssey*, an approach that uses statistics that allow for a more accurate cost estimation for federated queries and therefore enables *Odyssey* to produce better query execution plans. Our experimental results show that *Odyssey* produces query execution plans that are better in terms of data transfer and execution time than state-of-the-art optimizers. Our experiments using the FedBench benchmark show execution time gains of at least 25 times on average.

1 Introduction

Federated SPARQL query engines [1, 4, 7, 14, 17] answer SPARQL queries over a federation of SPARQL endpoints. Query optimization is a particularly complex and challenging task in a federated setting. The query optimizer minimizes processing and communication costs by selecting only relevant sources for a query. It decomposes the query into subqueries, and produces a query execution plan with good join ordering and physical operators. With limited access to statistics, however, most federated query engines rely on heuristics [1, 17] to reduce the huge space of possible plans or on dynamic programming (DP) [5, 7] to produce optimal plans. However, these plans may still exhibit a high number of intermediate results or high execution times because of inadequate heuristics or inaccurate estimations of cost functions [8].

In this paper, we propose *Odyssey*, a cost-based query optimization approach for federations of SPARQL endpoints. *Odyssey* defines statistics for representing entities inspired by [12] and statistics for representing links among datasets while guaranteeing result completeness. In a federated setting, computing statistics naturally requires access to more than one dataset. To reduce the overhead, *Odyssey* uses entity synopsis to identify links among datasets. This comes at the risk of losing some accuracy in the link identification but still guarantees that no links will be missed during query optimization,

i.e., there is a small risk that more sources are queried than strictly necessary but the query result will be complete.

Odyssey uses the computed statistics to estimate the sizes of intermediate results and dynamic programming to produce an efficient query execution plan with a low number of intermediate results. In summary, this paper makes the following contributions:

- Concise statistics of adequate granularity representing entities and describing links among datasets while guaranteeing result completeness.
- A lightweight technique to compute federated statistics in a federated setup that relies on entity synopsis.
- A query optimization algorithm based on dynamic programming using our statistics to find the best plan.
- Extensive evaluation using a well-accepted standard benchmark for federated query processing [16], including comparison against a broad range of state-of-the-art related work [5, 7, 15, 17]. The results show *Odyssey*'s superiority with a speed-up of up to 126 times and a reduction of transferred data of up to 118 times on average.

This paper is organized as follows. Section 2 presents related work, Section 3 describes the *Odyssey* approach and its algorithms. Section 4 discusses our experimental results. Finally, conclusions and future work are outlined in Section 5.

2 Related Work

Query optimization in state-of-the-art federated query engines, such as FedX [17] and ANAPSID [1], relies on heuristics. For instance, FedX [17] integrates the variable counting heuristic, where relative selectivity of triple patterns is heuristically estimated according to the presence of constants and variables in the triple patterns. These heuristics are lightweight but might not lead to the best query execution plan [18]. To find an optimal plan, several approaches [5, 7, 14, 19] rely on dynamic programming. However, given the high number of alternative query plans for SPARQL queries with many triple patterns, dynamic programming is very expensive [8]. Another important factor of query optimization is source selection. Several approaches [1, 7, 15, 17, 19] try to determine the relevance of a source by sending ASK queries, which increases the costs for a single query but might amortize in large federations for an overlapping query load. Another technique is to estimate whether combining the data of multiple sources can lead to any join results, e.g., by computing the intersection of the sources' URI authorities [15] or detailed statistics [10, 13].

Federated query optimization can also rely on cardinality estimations based on statistics and used, for instance, to reduce sizes of intermediate results. Most available statistics [3] use the Vocabulary of Interlinked Datasets VOID [2], which describes statistics at dataset level (e.g., the number of triples), at the property level (e.g., for each property, its number of different subjects), and at the class level (e.g., the number of instances of each class). However, approaches based on VOID [5, 7, 9] and other statistics, such as QTrees [10] and PARTrees [13], share the drawback of missing the best query execution plans because of errors in estimating cardinalities caused by relying on assumptions that often do not hold for arbitrary RDF datasets [12], e.g., a uniform data distribution and that the results of triple patterns are independent.

Characteristic sets (CS) [6, 12] aim at solving this problem in centralized systems by capturing statistics about sets of entities having the same set of properties. This in-

formation can then be used to accurately estimate the cardinality and join ordering of star-shaped queries. Typically, any set of joined triple patterns in a query can be divided into connected star-shaped subqueries. Subqueries in combination with the predicate that links them, define a characteristic pair (CP) [8, 11]. Statistics about such CPs can then be used to estimate the selectivity of two star-shaped subqueries. Such cardinality estimations can be combined with dynamic programming on a reduced space of alternative query plans. Whereas existing work on CSs and CPs were developed for centralized environments, this paper proposes a solution generalizing these principles for federated environments.

3 The *Odyssey* Approach

Inspired by the latest advances in statistics for centralized triple stores [8, 11, 12], *Odyssey* uses statistics about individual datasets to derive detailed statistics for optimizing federated queries. In the following, we first describe the foundations of our statistics on individual datasets (Section 3.1) and then propose a novel method for computing such statistics in a federated environment based on entity descriptions (Section 3.2). As the detailed entity descriptions cause too much overhead in a federated setup, we propose a method for reducing the sizes of the descriptions (Section 3.3). Finally, we present the *Odyssey* approach for query optimization and its main steps (Section 3.4): source selection, join ordering, and query decomposition.

3.1 Dataset Statistics on Individual Datasets

Star-Shaped Subqueries To estimate the cardinality and costs of BGPs sharing the same subject (or object), i.e., *star-shaped subqueries*, we exploit the principle that entities sharing the same set of properties are similar. In this context, we refer to the set of an entity's properties as its characteristic set (CS) and use $cs_s(e)$ to denote the CS of entity *e* in dataset *s* or *cs(e)* if *s* is clear from the context. For instance, in DBpedia 3.5.1 *cs(dbr:Gary_Goetzman)*=C_1={*dbo:birthDate*, *foaf:name*, *rdf:type*, *dbo:activeYearsStartYear*, *rdfs:label*, *skos:subject*}. In total, 260 entities share this set of properties and therefore CS C_1.

Listing 1.1: Statistics for CS C_1

```
{ count: 260,
  elems:
  {{ pred: dbo:birthDate, ocurrences : 260 },
   { pred: foaf:name, ocurrences: 326 },
   { pred: rdf:type, ocurrences: 1023 },
   { pred: dbo:activeYearsStartYear, ocurrences: 260 },
   { pred: rdfs:label, ocurrences: 260 },
   { pred: skos:subject, ocurrences: 1336 }}}
```

CSs can be computed by scanning once a dataset's triples sorted by subject; after all the entity properties have been scanned, the entity's CS is identified. For each CS *C*, we compute statistics, i.e., the number of entities sharing *C* (*count(C)*) and the number of triples with predicate *p* occurring with these entities (*occurrences(p, C)*). Listing 1.1 shows the statistics for the above mentioned example CS C_1. Entities of C_1 occur on average in 1 triple with property *dbo:birthDate* and in 3.94 triples with property *rdf:type*.

For a star-shaped query, only CSs including all of the query's properties are relevant as entities that only satisfy a subset of these properties cannot contribute to the answer.

Listing 1.2: Find persons that have been active

```
SELECT DISTINCT ?person WHERE {
  ?person dbo:birthDate ?date .              (tp1)
  ?person dbo:activeYearsStartYear ?sy .     (tp2)
  ?person foaf:name ?name                    (tp3)
}
```

For star-shaped queries asking for the set of unique entities described by some properties (query with DISTINCT modifier), the exact number of answers can be determined precisely (no estimation). For example, the cardinality of the query given in Listing 1.2 can be obtained by adding up the *count(C)* of all *CS*s containing the properties *dbo:birthDate*, *dbo:activeYearsStartYear*, and *foaf:name*. In DBpedia 3.5.1, there are 7,059 CSs that include these three properties, and the total number of entities with these CSs is 83,438. Formally, the number of entities for a given set of properties *P*, *cardinality(P)*, is computed based on the CSs C_j that include all the properties in *P* as:

$$cardinality(P) = \sum_{P \subseteq C_j} count(C_j) \tag{1}$$

For queries without the DISTINCT modifier, we need to account for duplicates by considering the number of triples with predicate $p_i \in P$ that an entity is associated with on average:

$$estimatedCardinality(P) = \sum_{P \subseteq C_j} \left(count(C_j) * \prod_{p_i \in P} \frac{ocurrences(p_i, C_j)}{count(C_j)} \right) \tag{2}$$

In DBpedia 3.5.1, as mentioned above, there are 7,059 CSs relevant for the query in Listing 1.2 with 83,438 entities as answer. These 83,438 entities are described by 109,830 triples with predicate *foaf:name*, 83,448 with predicate *dbo:birthDate*, and 110,460 with predicate *dbo:activeYearsStartYear*. If the query is considered without the DISTINCT modifier, i.e., considering duplicated results, we estimate: 148,486 matching entities in the result, which is very close to the real number (149,440).

Once the relevant CSs for a query have been identified, they can be used to find the join order minimizing the sizes of intermediate results. For the query in Listing 1.2, we start by estimating the cardinalities for each subquery with two out of the three triple patterns using Formula 1: {tp1, tp2}: 98,281, {tp1, tp3}: 209,731, and {tp2, tp3}: 127,712. The triple pattern not included in the cheapest subquery ({tp1, tp2}) is executed last (tp3). We proceed recursively with the cheapest subquery and determine the cardinalities for its subsets: {tp1}: 232,608 and {tp2}: 143,004. Again, the triple pattern not included in the cheapest subquery (tp1) will be executed last of the currently considered set of triple patterns. As a result, we will execute the join between tp2 and tp1 first and afterwards compute the join with tp3. We also get the order in which the triple patterns should be evaluated for the first join: first tp2 and then tp1.

Arbitrary Queries To estimate the cardinality for queries with more complex shapes, we need to consider the connections (links) between entities with different CSs. Entity *dbr:Evan_Almighty*, for example, is linked to *dbr:Tom_Shadyac* via property *dbo:director* by triple *(dbr:Evan_Almighty, dbo:director, dbr:Tom_Shadyac)*.

The links between CSs via properties can formally be described by characteristic pairs (CPs), they are defined as *(cs_s(e1), cs_s(e2), p)* for entities e1 and e2 if *(e1, p,*

e2) $\in s$. The statistics – $count((C_i, C_j, p))$ – capture the number of links between a pair of CSs (C_i and C_j) using a particular property p. For example, given the CSs of *dbr:Tom_Shadyac* and *dbr:Evan_Almighty* as C_1 and C_2 the number of links via property *dbo:director* is given by: *count*((C_2, C_1, *dbo:director*)).

Listing 1.3: Find movies and their directors

```
SELECT DISTINCT ?film ?director WHERE {
  ?film dbo:runtime ?runtime .              (tp1)
  ?film dbo:director ?director .            (tp2)
  ?film dbo:budget ?budget .                (tp3)
  ?director dbo:birthDate ?date .           (tp4)
  ?director dbo:activeYearsStartYear ?sy .  (tp5)
  ?director foaf:name ?name                 (tp6)
}
```

The number of unique results (pairs of entities, query with DISTINCT modifier) can be exactly computed (not estimated) using the formula:

$$cardinality((P_i, P_j, p)) = \sum_{P_i \subseteq C_k \wedge P_j \subseteq C_l} count((C_k, C_l, p)) \tag{3}$$

For the query in Listing 1.3 property *dbo:director* links several pairs of CSs representing movies and actors. Hence, we need to compute $\Sigma_{f_1 \wedge f_2}$ $count$((C_k, C_l, *dbo:director*)), where f_1 is {*dbo:runtime*, *dbo:director*, *dbo:budget*} $\subseteq C_k$ and f_2 is {*dbo:birthDate*, *dbo:activeYearsStartYear*, *foaf:name*} $\subseteq C_l$, one of the operands of this sum is $count$((C_2, C_1, *dbo:director*)) mentioned in the example above. For this query, DBpedia 3.5.1 contains 1,509 CPs linking entities from two CSs via property *dbo:director*.

If a query does not involve the DISTINCT modifier, result cardinality estimation considers the property occurrences in the CSs:

$$\begin{aligned} estimatedCardinality((P_i, P_j, p)) = \sum_{P_i \subseteq C_k \wedge P_j \subseteq C_l} \Big(& count((C_k, C_l, p)) \\ & * \prod_{p_i \in P_i - \{p\}} \left(\frac{ocurrences(p_i, C_k)}{count(C_k)}\right) * \prod_{p_j \in P_j} \left(\frac{ocurrences(p_j, C_l)}{count(C_l)}\right) \Big) \end{aligned} \tag{4}$$

Assuming that the order of joins within star-shaped subqueries has already been determined based on the CSs as described above, we treat each star-shaped subquery as a single meta-node to reduce complexity. We estimate the cardinalities of the meta-nodes using the statistics on CPs and use dynamic programming (DP) to determine the optimal join order that minimizes the size of intermediate results. Although the presentation in this section focuses on subject-subject joins, the same principle can be applied to other types of joins, e.g., object-object.

3.2 Federated Statistics

In general, entities might occur in multiple datasets in a federation S. Hence, we define a *federated characteristic set* (FCS) as follows: $fcs_S(e) = \bigcup_{s \in S} cs_s(e)$, S might be omitted if clear from the context. However, triples describing the same entity are typically part of a single dataset so that most CSs can be computed over each dataset independently from the others[3]. The *federated characteristic pair* (FCP) of entities *e1* and *e2* via property *p*

[3] FCSs describing entities across multiple datasets are very rare. In FedBench, for instance, they affect less than 0.5% of all CSs.

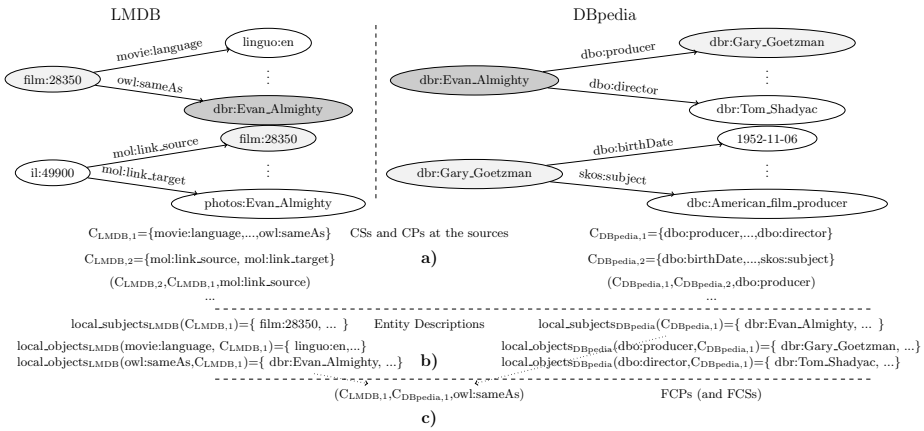


Fig. 1: Federated Computation of Statistics

in federation *S* is defined as *(fcs_S(e1), fcs_S(e2), p)*. For FCSs FC_i and FC_j and property p, we compute statistics *count(FC_i)*, *occurrences(p, FC_i)*, and *count((FC_i,FC_j,p))* as before for CSs and CPs. For simplicity, the following sections focus on FCPs connecting CSs instead of FCSs. The generalization using FCSs is straightforward.

Whereas single dataset statistics can be computed once and provided by the sources in the same way they currently provide VOID statistics, FCSs and FCPs require more effort and centralized knowledge about all entities in the considered datasets. A naive way to compute FCSs and FCPs is evaluating expensive SPARQL queries with FILTER expressions involving NOT EXISTS, but this can take weeks for a dataset with thousands of CSs. It is much more efficient if the sources directly share information about local subjects and objects with the federated query engine. *local_subjects$_s$*(C) contains the entity set with CS C for source s, while *local_objects$_s$*(p, C) contains the set of entities connected via p to CS C. Such information can, for instance, be obtained efficiently while computing CSs and CPs locally and then shared with the federated query engine.

The federated query engine can then use this information to compute FCSs and FCPs. Consider, for instance, the two datasets LMDB and DBpedia in Fig. 1; based on the CSs (Fig. 1a)), the sources compute *entity descriptions* (*local_subjects$_i$* and *local_objects$_i$* in Fig. 1b)). Entity *film:28350* has properties { movie:language,...,owl:sameAs}= $C_{\text{LMDB},1}$. Hence, *film:28350* $\in$ *local_subjects$_{LMDB}$*($C_{\text{LMDB},1}$). Entity *dbr:Evan_Almighty* is the value of property *owl:sameAs* for an entity with CS $C_{\text{LMDB},1}$ (*film:28350*) so *dbr:Evan_Almighty* $\in$ *local_objects$_{LMDB}$*(owl:sameAs, $C_{\text{LMDB},1}$) (Fig. 1b)). The overlap between the set of entities *local_objects$_{DBpedia}$*($C_{\text{DBpedia},1}$) and *local_subjects$_{LMDB}$*(owl:sameAs, $C_{\text{LMDB},1}$) are links from LMDB to DBpedia via property owl:sameAs. Hence, we obtain FCP ($C_{\text{LMDB},1}$, $C_{\text{DBpedia},1}$, owl:sameAs) (Fig. 1c)). *count*(($C_{\text{LMDB},1}$, $C_{\text{DBpedia},1}$, owl:sameAs)) corresponds to the cardinality of the intersection between all the *local_objects$_{DBpedia}$* and *local_subjects$_{LMDB}$*.

Algorithm 1 describes in more detail how to compute FCPs only based on the pre-

Algorithm 1 Compute FCPs Algorithm

Input: $local\_objects_{d1}$ and $local\_subjects_{d2}$ for datasets $d1$ and $d2$
Output: A set of FCPs (*FCPs*) with links from $d1$ to $d2$
count(fcp) for each fcp in FCPs

1: **function** *ComputeFCPs*(local_subjects$_{d2}$, local_objects$_{d1}$)
2: FCPs ← { }
3: count ← a function with default value 0
4: **for** (p, $C_{d1,i}$) ∈ domain(local_objects$_{d1}$) **do**
5: entities ← local_objects$_{d1}$(p,$C_{d1,i}$)
6: **for** $C_{d2,j}$ ∈ domain(local_subjects$_{d2}$) **do**
7: entities ← entities ⋂ local_subjects$_{d2}$($C_{d2,j}$)
8: **if** entities ≠ ∅ **then**
9: FCPs ← FCPs ⋃ { ($C_{d1,i}$, $C_{d2,j}$, p) }
10: count(($C_{d1,i}$, $C_{d2,j}$, p)) ← count(($C_{d1,i}$, $C_{d2,j}$, p)) + cardinality(entities)
11: **end if**
12: **end for**
13: **end for**
14: **return** CPs, count
15: **end function**

computed statistics $local\_objects_{d1}$ and $local\_subjects_{d2}$. First, all common entities in $local\_objects_{d1}$ and $local\_subjects_{d2}$ are identified in line 7. These common entities represent links between CSs $C_{d1,i}$ and $C_{d2,j}$ via property p and are captured by a FCP (lines 9-10).

Listing 1.4: Find LMDB movies that are also DBpedia movies

```
SELECT ?film ?movie WHERE {
 ?film dbo:budget ?budget .
 ?film dbo:director ?director .
 ?movie owl:sameAs ?film .
 ?movie lmdb:sequel ?seq
}
```

FCPs can be used for cardinality estimation and join ordering using the same principles as described in Section 3.1. Consider a federation consisting of DBpedia (160,061 CSs) and LMDB (8,466 CSs) with 22,592 FCPs. We can use Formula 4 with FCPs to estimate the result cardinality: 171. This is close to the real cardinality (293).

3.3 Reducing the Sizes of Entity Descriptions

As the entity descriptions ($local\_subjects_d$ and $local\_objects_d$) introduced above are often very expensive to compute, maintain and exchange, we propose a technique to reduce their sizes. We organize the entity descriptions in a tree structure that summarizes the entities used as subject or object in any pf the dataset's triples. Inspired by [10, 13, 15], we factorize common prefixes, transform suffixes into integers, and summarize sets of integers in buckets, i.e., a set synopsis consisting of minimum value (mn), maximum value (mx), $[mn, mx]$, number of elements, *num*, and their set of two least significant bytes (*lsb*). *lsb*(i) is computed as i **mod** 2^{16} and is included to improve the synopsis' accuracy.

The tree structure is organized in three levels. The top level summarizes the prefixes of entity IRIs occurring as subjects and objects in the dataset. Suffixes are mapped to integers using a hash function, these integers are summarized in the middle and bottom levels. The middle level includes buckets where parent nodes subsume the synopsis of their children (ranges are included, *num*s are added) and aids in efficiently accessing

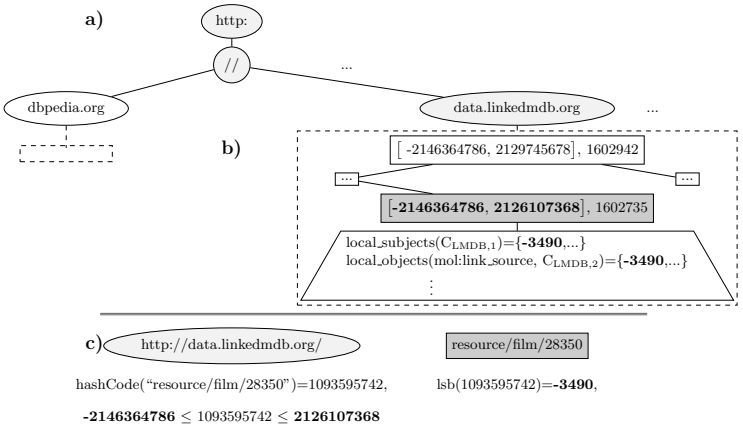


Fig. 2: Reduced Entity Descriptions for LMDB in Fig. 1. The tree factorizes common prefixes in the top level (in the ellipses) and summarizes the suffixes in the middle (in the rectangles) and bottom (in the trapezium) levels

the bottom level. The bottom level (leaves) stores (in *local_subjects* and *local_objects*) only the integer's *lsb* to reduce the space use while improving the synopsis' accuracy.

In Fig. 2 we present a fragment of the reduced descriptions for LMDB. The reduced descriptions include all the entities that are subject or object in the dataset's triples. In particular, it includes the entity with IRI *http://data.linkedmdb.org/resource/film/28350* (Fig. 2c)). This IRI prefix identifies the subtree that summarizes the entity (light gray ellipses in Fig. 2a)), while the hash code of its suffix (resource/film/28350), 1093595742, is used to identify the leaf that includes its lsb (-3490), i.e., with 1093595742 between its minimum and maximum values (gray rectangle in Fig. 2b)). Its lsb is in *local_subjects*($C_{LMDB,1}$) and *local_objects*(mol:link_source, $C_{LMDB,2}$) in the identified leaf (trapezium in Fig. 2b)). This tree structure exhibits size reduction and eases the computation of FCPs by allowing to discard large portions of the descriptions contrary to descriptions in Fig. 1b), where all the *local_subjects* and *local_objects* should be pair-wise tested for overlap.

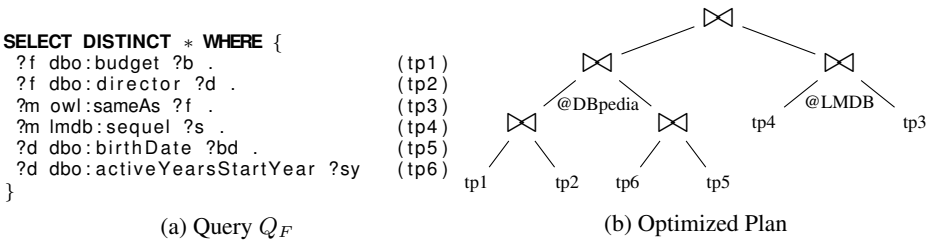
Computation costs are greatly reduced by pruning large portions of the tree and comparing only a few pairs of leaves, the ones that have common prefixes and overlapping representation of the suffixes. An important feature of these summaries is that entities present in more than one dataset are always detected.

These trees are considerably lighter than the entity descriptions discussed in Section 3.2, but they might reduce accuracy. For FedBench's DBpedia 3.5.1 subset, a dataset with 43,126,772 triples that occupies 6.1GB, the *local_subjects* and *local_objects* occupy 1.37GB and the tree only occupies 68MB[4]. They have compression ratios of 4.45 and 91.86, respectively. Regarding the quality, the tree summary allows for computing all the FCSs and FCPs.

[4] Implementation based in Java's HashSet and HashMap was used to measure their sizes.

To reduce the resources used by the tree, we have reduced the number of CSs as suggested in [8, 12] to 10,000. Only the CSs that are shared by a greater number of entities are kept, and the others are merged into existing CSs if possible. For instance, by selecting from the CSs that include all the properties of the merged CS the one with the smallest number of properties and adding *count* and *ocurrences* values to its own values, or splitting the CS into property subsets that can be merged with other CSs. This may reduce the accuracy of the query cardinality estimation, but it allows to bound the resources used to store and access these statistics.

Entity summaries can be kept up-to-date in two ways. For datasets that are rarely updated the subtree representing the entities with the prefix affected by the updates, e.g., Fig. 2b) in our example, can be re-computed. For datasets that are often updated, leaves should support removal of entities, this can be easily done by storing the multiplicity of each least significant byte, so they are removed only if all the entities with that least significant byte have been removed from the dataset.



(a) Query Q_F (b) Optimized Plan

Fig. 3: Query Q_F and its Optimized Plan

3.4 Optimizing Federated Queries

Query optimization in *Odyssey* can logically be divided into the following steps: *i)* preprocessing and source selection, *ii)* join ordering, and *iii)* query decomposition. Arbitrary queries can be handled incrementally by optimizing its subqueries. In the following, we address the optimization of queries with bounded predicates, *Odyssey* relies on existing optimizers to handle other queries.

Preprocessing and source selection We first parse the query and identify its star-shaped subqueries. Then, properties in each star-shaped subquery are used to identify relevant CSs and sources. For example, the subquery composed by tp3 and tp4 in Fig. 3(a) has relevant CSs that include both owl:sameAs and movie:sequel. In the FedBench federation described in Table 1, these CSs are only part of LMDB. Therefore, LMDB is the only relevant source for this subquery. Afterwards, we use CPs/FCPs to identify relevant sources for the links between the star-shaped subqueries.

Join ordering Once we have identified the set of relevant sources, we can estimate cardinalities of subqueries and find the best join ordering. We first optimize the order of joins and triple patterns within each star-shaped subquery as explained in Section 3.1 using the reduced entity descriptions described in Section 3.3. Afterwards, as described in Section 3.1, each subquery is treated as a meta-node and we estimate cardinalities of

the joins between these meta-nodes using formulas in Section 3.1 for FCPs to estimate subquery cost and apply DP. Fig. 4 (left) shows the estimated cardinality and cost of the subqueries of Q_F (Fig. 3(a)), arrows indicate which smaller subqueries were combined by the DP algorithm to form a larger subquery. As the number of subqueries is usually considerably lower than the number of triple patterns, applying DP becomes affordable.

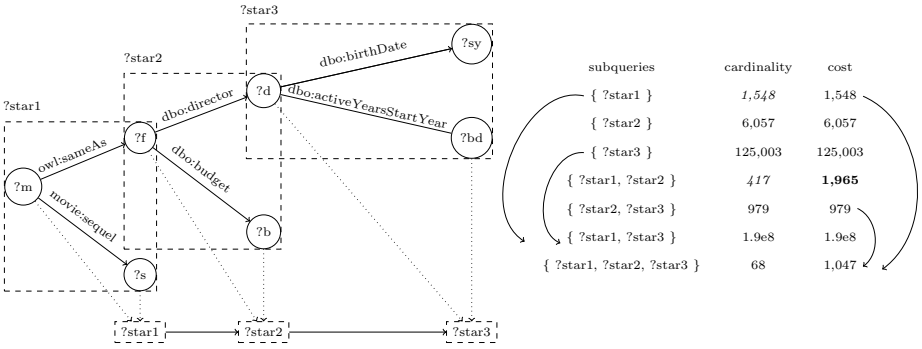


Fig. 4: Example query optimization

In our current implementation, the cost function is solely defined on the cardinalities of intermediate results and how many results need to be transferred from endpoints during execution. This favors query plans with selective subqueries. For instance, the cost of the join between meta-nodes ?star1 and ?star2 (**1,965**) includes the result sizes (*417*) and the incurred intermediate results (*1,548*). This cost function assumes that all endpoints have the same characteristics. We can easily extend this cost function by additional parameters that can be fine-tuned to represent the characteristics of each endpoint individually, e.g., communication delays, response times, etc.

Query decomposition Finally, we optimize the SPARQL queries that are actually sent to the endpoints and try to minimize their number. For instance, we combine all triple patterns and logical subqueries to a particular endpoint into a single SPARQL query to a particular endpoint whenever possible. For instance, meta-nodes ?star2 and ?star3 in Fig. 4 are combined into one subquery (Fig. 3(b)) and evaluated by the DBpedia endpoint.

4 Evaluation

In this section, we present the results of our experimental study that compares our approach, *Odyssey*, with state-of-the-art federated query engines: HiBISCuS (FedX-HiBISCuS, cold and warm cache) [15], SemaGrow [5], FedX (cold and warm cache) [17], and SPLENDID [7]. Full implementations, statistics, and results are available at `https://github.com/gmontoya/federatedOptimizer`.

Datasets and queries: We use the real datasets and queries proposed in the FedBench benchmark [16]. Queries are divided into three groups Linked Data (LD1-LD11), Cross Domain (CD1-CD7), and Life Science (LS1-LS7). They have 2-7 triple patterns and star and hybrid shapes. They have between 1 and 9,054 answers. Basic statistics about

Table 1: FedBench [16] dataset statistics: number of distinct triples (#DT), number of predicates (#P), number of CSs (#CS), number of CPs (#CP), *Odyssey* statistics and synopsis computation time in s, HiBISCuS summaries computation time in s, and VOID statistics computation time in s

Dataset	#DT	#P	# CS	#CP	#FCP	*Odyssey*	HiBISCuS	VOID
ChEBI	4,772,706	28	978	9,958	19,360	82.91	96.02	73.89
KEGG	1,090,830	21	67	239	13,822	30.15	95.23	12.84
Drugbank	517,023	119	3,419	12,589	103,070	1,299.9	76.4	6.98
DBpedia subset	42,855,253	1,063	10,000	1,069,431	6,583	2,739	770.48	1,465.36
Geonames	107,949,927	26	673	7,707	322,672	1,885.97	609.52	39,694.07
Jamendo	1,049,647	26	42	190	1,259	31.25	99.17	14.66
SWDF	103,595	118	547	6,713	17,557	7.27	69.21	2.03
LMDB	6,147,916	222	8,466	94,188	359,340	947.16	317.21	355.45
NYTimes	335,119	36	47	158	3.96	10.01	72.56	4.22
Federated						620.35		
Total						7,654.27	2,205.8	41,629.5

the datasets are listed in Table 1. We ran each query ten times and report the averages over the last nine runs. Standard deviation is included as error bars on the plots.

Implementation: *Odyssey* is implemented in Java using the Jena library to parse and transform queries into queries with SPARQL 1.1 service clauses. Our implementation uses the FedX 3.1 framework with deactivated native optimization to execute *Odyssey*'s query plans.

Hardware configuration: For our experiments we used virtual machines (VMs). A VM using up to 4GB of RAM to run the federated query engine and nine VMs with 2 processors, 8GB of RAM and CPU 2294.250 MHz to host Virtuoso endpoints with the datasets described in Table 1 (one dataset and endpoint per VM).

Statistics computation: As DBpedia has a very high number of CSs (160,061), we reduced them to 10,000 by merging (as suggested in [8,12] and explained in Section 3.3) without significant losses in the quality of estimations. Details on creation times of statistics are listed in Table 1. *Odyssey*'s statistics can be more expensive to compute for datasets with more than 3,419 CSs and cheaper than HiBISCuS's for datasets with less than 67 CSs. In total, *Odyssey*'s statistics are computed five times faster than VOID's.

Evaluation metrics: *i) Optimization time (OT):* is the elapsed time since the query is issued until the optimized query plan is produced, *ii) number of selected sources (NSS):* is the number of sources that have been selected to answer a query, *iii) number of subqueries (NSQ):* is the number of subqueries that are included in the query plan, *iv) execution time (ET):* is the time elapsed since the evaluation of the query plan starts until the complete answer is produced (with a timeout of 1,800 seconds), *v) number of transferred tuples (NTT):* is the number of tuples transferred from all the endpoints to the query engine during query evaluation.

Result completeness: All approaches produce the complete result set for non-timed out queries, except SPLENDID for query LS7.

4.1 Experimental Results

Optimization time Fig. 5 shows the optimization time (OT) for the studied approaches. Because of the detailed statistics and dynamic programming, one might expect *Odyssey* to suffer from a considerable overhead in OT. As our experimental results show, however, *Odyssey*'s query planner is competitive to most other approaches with a slight

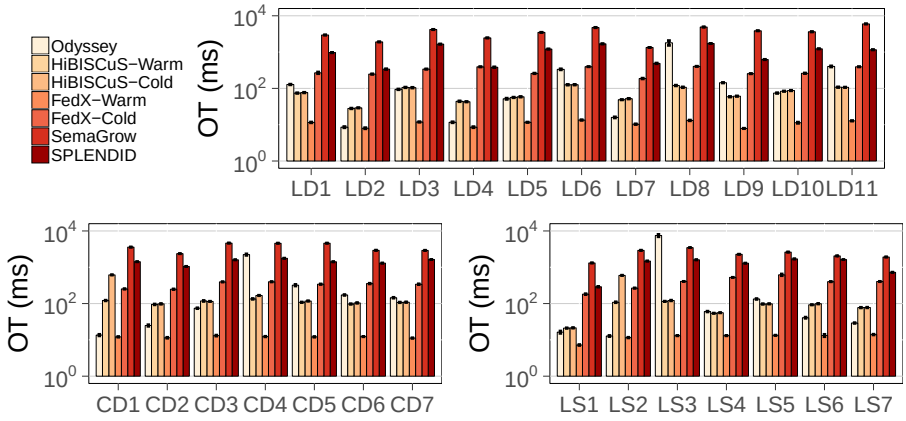


Fig. 5: Optimization Time in ms (OT, log scale). CD1 and LS2 have variable predicates and *Odyssey* relies on FedX to find plan.

advantage for FedX-Warm as this system has cached information about the query relevant sources. For instance, *Odyssey* is up to 69 times faster (SemaGrow) than other approaches on average.

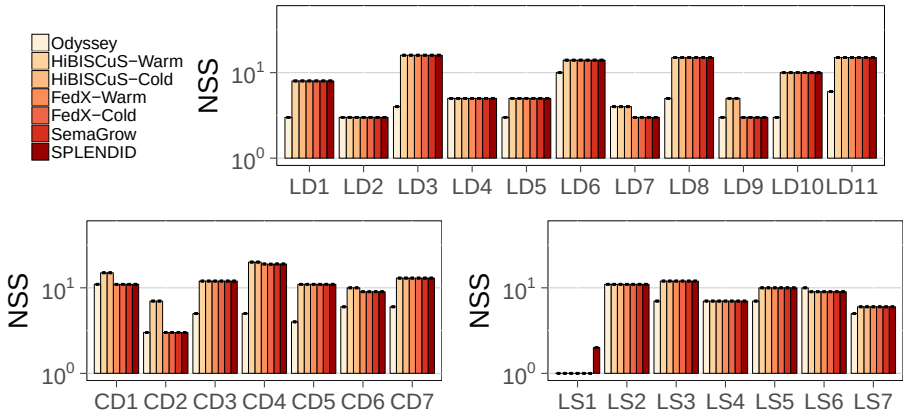


Fig. 6: Number of Selected Sources (NSS, log scale, 10^0=1)

Number of selected sources As Fig. 6 shows, *Odyssey* selects only a small number of relevant sources; for instance, at least 1.81 times less (FedX-Cold/Warm and SemaGrow) and up to 1.93 times less (HiBISCuS-Cold/Warm) on average. For some queries, e.g., LS4, existing approaches already select the optimal number of sources. For LD7, *Odyssey* selects a larger number of sources than the optimum because our approach does not perform ASK queries during execution to prune irrelevant sources. Sometimes *Odyssey* overestimates the set of relevant sources – but on the other hand it never misses

any relevant sources. For LS1, most approaches select just one (10^0) source because there is only one dataset that have triples with the predicate in the query.

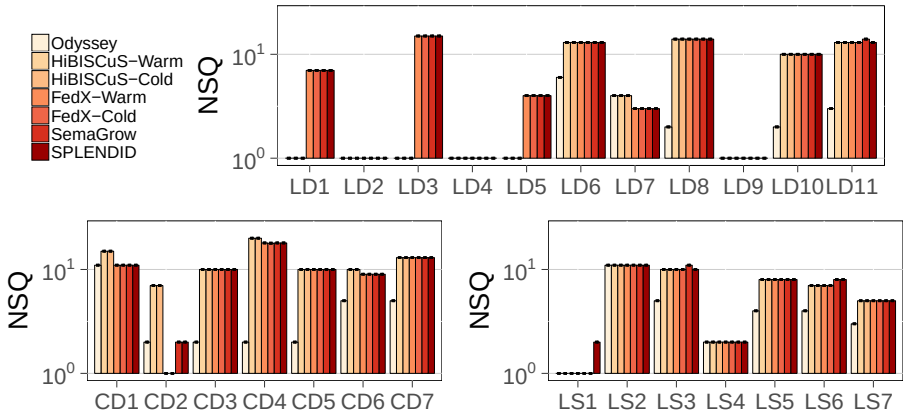


Fig. 7: Number of Subqueries (NSQ, log scale, 10^0=1)

Number of subqueries As Fig. 7 shows, *Odyssey* uses considerably fewer subqueries than other approaches, at least 2.62 times less (HiBISCuS-Cold/Warm) and up to 3.41 times less (SPLENDID) on average. The fact that Odyssey always produces the correct and complete answers confirms that *Odyssey* correctly identifies and exploits cases for which it is advantageous to combine subqueries. *Odyssey*'s reduction of the number of relevant sources has a positive impact on the number of subqueries (NSQ), *Odyssey*'s pruning of non relevant sources allows for combining triple patterns into subqueries without affecting the result completeness. Some queries, like LD2, LD4, and LD9, include triple patterns that can be evaluated by a unique endpoint of the federation and existing approaches already decompose the query into the optimal NSQ. Only for LD7, FedX-Cold/Warm, SPLENDID, and SemaGrow decompose the query into fewer subqueries than *Odyssey*, this is because they use ASK queries to assess a source's relevance. *Odyssey* could be enhanced with this strategy.

Execution time Some approaches failed to answer all queries before the timeout: SPLENDID (2 queries) and SemaGrow (4 queries). Even when considering only those queries that completed before the timeout, *Odyssey* is on average 126.26 times faster than SPLENDID and 28.30 times faster than SemaGrow. Fig. 8 shows the execution times (ET) for the studied approaches. *Odyssey* is on average at least 25.46 times faster (FedX-Warm). Only for few queries *Odyssey* is (slightly) slower than other approaches, e.g., LS3. As for the other metrics, *Odyssey*'s ET can be improved if ASK queries were used during query execution to further reduce the relevant sources similarly as it is done by other approaches. For five of the queries, *Odyssey* is one of the fastest approaches and for 11 queries, *Odyssey* is the fastest approach. *Odyssey*'s achieved reductions on the NSS and NSQ have a positive impact on the ET, as fewer endpoints are queried fewer times, *Odyssey* produces results faster than most approaches in most cases.

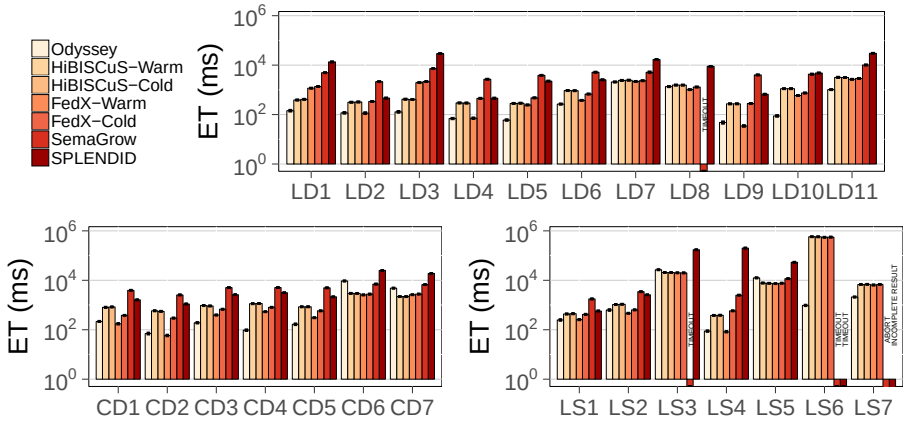


Fig. 8: Execution Time in ms (ET, log scale)

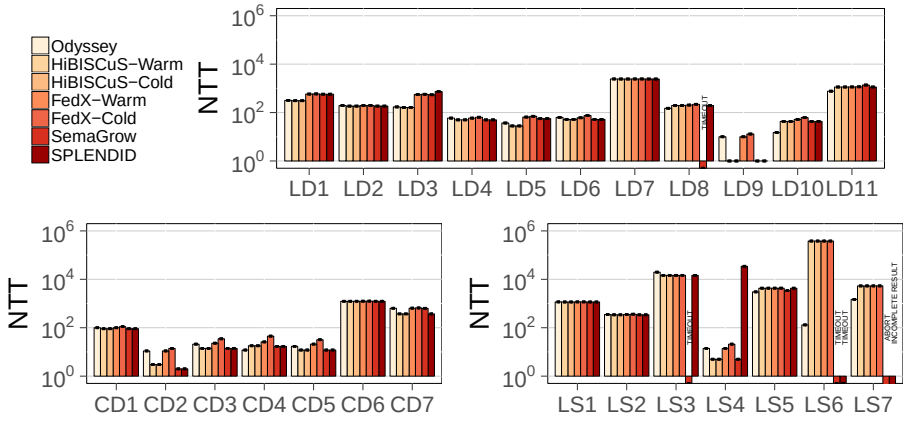


Fig. 9: Number of Transferred Tuples (NTT, log scale, 10^0=1)

Number of transferred tuples Fig. 9 shows the number of transferred tuples (NTT) for the studied approaches. *Odyssey* transfers fewer tuples than other approaches. Even when considering only those queries that completed before the timeout, *Odyssey* transfers on average 1.15 times fewer tuples faster than SemaGrow and 108.4 times fewer tuples than SPLENDID. For the approaches that completed all the queries, *Odyssey* transfers at least 117.55 fewer tuples (HiBISCuS-Cold/Warm) on average. Most approaches are competitive in terms of NTT. The largest difference is observed for LS6, where *Odyssey* clearly outperforms the other approaches transferring 500 times fewer tuples. In contrast to other approaches, *Odyssey* not only reduces the number of requests sent to the endpoints but also avoids non-selective queries, which significantly reduces network traffic and the local load at the endpoints.

4.2 Combining *Odyssey* with Existing Optimizers

We have also integrated *Odyssey* techniques into an the FedX optimizer and obtained:

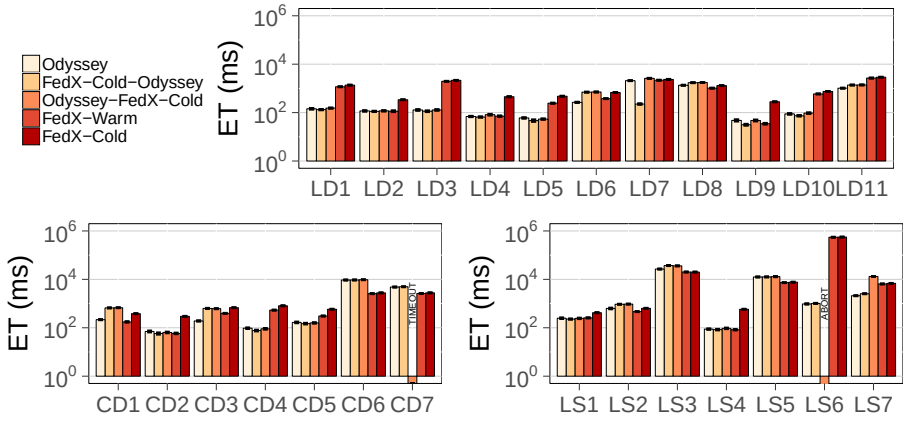


Fig. 10: Execution Time in ms (ET, log scale)

- *Odyssey*-FedX-Cold, which relies on CSs and CPs to select sources and decomposes the query but uses FedX join ordering.
- FedX-Cold-*Odyssey*, which relies on the FedX optimizer for source selection but uses *Odyssey* for query decomposition and join ordering.

Fig. 10 compares the execution times (ET) of these two implementations with *Odyssey*, FedX-Cold, and FedX-Warm. In most cases the combined approaches are considerably faster than native FedX. In a few cases, however, their ET can increase considerably. In these cases, queries include a highly selective subquery with one triple pattern and using FedX's heuristic to execute subqueries with more than one triple pattern first leads to plans that are more expensive than others. On average, the combined approaches are 26.86 and 3.99 times faster than FedX-Cold.

For query LD7, *Odyssey* and FedX-Cold/Warm exhibit similar ETs whereas FedX-Cold-*Odyssey* is considerably faster. For this query it happens that the advantages of both *Odyssey* and FedX coincide, i.e., we can take advantage of the good join ordering by *Odyssey* but also of the additional pruning based on ASK queries by FedX.

Even if *Odyssey*'s OT can be higher in comparison to existing approaches, *Odyssey* produces better plans composed of fewer subqueries and fewer selected sources per triple pattern without compromising result completeness. Benefits of these features have been evidenced with significantly faster ETs and less transferred data from endpoints to the federated query engine.

5 Conclusion

In this paper, we have presented *Odyssey*, an approach for optimizing federated SPARQL queries based on statistics. These statistics detail information about the data provided by remote endpoints as well as the links between them. This enables more accurate cost estimations, query optimization, and selection of relevant sources. Our extensive experimental evaluation shows that *Odyssey* produces query execution plans that are better in terms of data transfer and execution time than state-of-the-art optimiz-

ers. In our future work, we plan to further improve *Odyssey* by considering in which situations exactly it is worthwhile to use additional aspects of other optimizers, such as ASK queries and associated statistics. Another interesting perspective work is to further reduce the computation time and sizes of the entity descriptions and provide efficient strategies to update the descriptions and statistics.

Acknowledgments
This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-4093-00301.

References

1. M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *ISWC'11*, pages 18–34, 2011.
2. K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing Linked Datasets. In *LDOW'09*, 2009.
3. C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *ISWC'13*, pages 277–293, 2013.
4. C. Basca and A. Bernstein. Querying a Messy Web of Data with Avalanche. *J. Web Sem.*, 26:1–28, 2014.
5. A. Charalambidis, A. Troumpoukis, and S. Konstantopoulos. SemaGrow: Optimizing Federated SPARQL queries. In *SEMANTICS'15*, pages 121–128, 2015.
6. F. Du, Y. Chen, and X. Du. Partitioned Indexes for Entity Search over RDF Knowledge Bases. In *DASFAA*, pages 141–155, 2012.
7. O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *COLD'11*, 2011.
8. A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT'14*, pages 439–450, 2014.
9. S. Hagedorn, K. Hose, K. Sattler, and J. Umbrich. Resource planning for SPARQL query execution on data sharing platforms. In *COLD*, pages 49–60, 2014.
10. A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich. Data Summaries for On-Demand Queries over Linked Data. In *WWW'10*, pages 411–420, 2010.
11. M. Meimaris, G. Papastefanatos, N. Mamoulis, and I. Anagnostopoulos. Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization. In *ICDE'17*.
12. T. Neumann and G. Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *ICDE'11*, pages 984–994, 2011.
13. F. Prasser, A. Kemper, and K. A. Kuhn. Efficient Distributed Query Processing for Autonomous RDF Databases. In *EDBT'12*, pages 372–383, 2012.
14. B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *ESWC*, pages 524–538, 2008.
15. M. Saleem and A. N. Ngomo. HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation. In *ESWC*, pages 176–191, 2014.
16. 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'11*, pages 585–600, 2011.
17. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *ISWC'11*, pages 601–616, 2011.
18. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *WWW'08*, pages 595–604, 2008.
19. X. Wang, T. Tiropanis, and H. C. Davis. LHD: Optimising Linked Data Query Processing Using Parallelisation. In *LDOW*, 2013.