Heuristics-based Query Reordering for Federated Queries in SPARQL 1.1 and SPARQL-LD

Thanos Yannakis[1], Pavlos Fafalios[2], and Yannis Tzitzikas[1]

[1] Computer Science Department, University of Crete, and FORTH-ICS, Greece
{yannakis, tzitzik}@ics.forth.gr
[2] L3S Research Center, Leibniz University of Hannover, Germany
fafalios@L3S.de

Abstract. The federated query extension of SPARQL 1.1 allows executing queries distributed over different SPARQL endpoints. SPARQL-LD is a recent extension of SPARQL 1.1 which enables to directly query any HTTP web source containing RDF data, like web pages embedded with RDFa, JSON-LD or Microformats, without requiring the declaration of named graphs. This makes possible to query a large number of data sources (including SPARQL endpoints, online resources, or even Web APIs returning RDF data) through a single one concise query. However, not optimal formulation of SPARQL 1.1 and SPARQL-LD queries can lead to a large number of calls to remote resources which in turn can lead to extremely high query execution times. In this paper, we address this problem and propose a set of query reordering methods which make use of heuristics to reorder a set of SERVICE graph patterns based on their restrictiveness, without requiring the gathering and use of statistics from the remote sources. Such a query optimization approach is widely applicable since it can be exploited on top of existing SPARQL 1.1 and SPARQL-LD implementations. Evaluation results show that query reordering can highly decrease the query-execution time, while a method that considers the number and type of unbound variables and joins achieves the optimal query plan in 88% of the cases.

Keywords: Query reordering, SPARQL 1.1, SPARQL-LD, Linked Data

1 Introduction

A constantly increasing number of data providers publish their data on the Web following the Linked Data principles and adopting standard RDF formats. According to the Web Data Commons project [16], 38% of the HTML pages in the Common Crawl[3] of October 2016 contains structured data in the form of RDFa, JSON-LD, Microdata, or Microformats[4]. This data comes from millions of different pay-level-domains, meaning that the majority of Linked Data is nowadays available through a large number of different data sources. The question is: how

[3] http://commoncrawl.org/
[4] http://webdatacommons.org/structureddata/2016-10/stats/stats.html

can we *efficiently* query this large, distributed, and constantly increasing body of knowledge?

SPARQL [2] is the *de facto* query language for retrieving and manipulating RDF data. The SPARQL 1.1 Federated Query recommendation of W3C allows executing queries distributed over different SPARQL endpoints [17]. SPARQL-LD [7,8] is an extension (generalization) of SPARQL 1.1 Federated Query which extends the applicability of the `SERVICE` operator to enable querying any HTTP web source containing RDF data, like online RDF files (RDF/XML, Turtle, N3) or web pages embedded with RDFa, JSON-LD, or Microformats. Another important characteristic of SPARQL-LD is that it does not require the named graphs to have been declared, thus one can even fetch and query a dataset returned by a portion of the query, i.e., whose URI is derived at query execution time. Thereby, by writing a single concise query, one can query hundreds or thousands of data sources, including SPARQL endpoints, online resources, or even Web APIs returning RDF data [8].

However, not optimal query writing in both SPARQL 1.1 and SPARQL-LD can lead to a very large number of `SERVICE` calls to remote resources, which in turn can lead to an extremely high query execution time. Thus, there arises the need for an effective query optimization method than can find a near-optimal query execution plan. In addition, given the dynamic nature of Linked Data and the capability offered by SPARQL-LD to query any remote HTTP resource containing RDF data, we need a widely-applicable method that does not require the use of statistics or metadata from the remote sources and that can operate on top of existing SPARQL 1.1 and SPARQL-LD implementations.

To this end, in this paper we propose and evaluate a set of query reordering methods for SPARQL 1.1 and SPARQL-LD. We focus on fully heuristics-based methods that reorder a query's `SERVICE` graph patterns based on their restrictiveness (selectivity), without requiring the gathering and use of statistics from the remote sources. The objective is to decrease the number of intermediate results and thus the number of calls to remote resources. We also propose the use of a greedy algorithm for computing a near-optimal query execution plan for cases of large number of `SERVICE` patterns.

In a nutshell, in this paper we make the following contributions:

- We propose a set of heuristics-based query reordering methods for SPARQL 1.1 and SPARQL-LD, which can also exploit a greedy algorithm for choosing a near-optimal query execution plan. The query optimizer is publicly available as open source.[5]
- We report the results of an experimental evaluation which show that a method that considers the number and type of unbound variables and the number and type of joins achieves the optimal query plan in 88% of the examined queries, while the greedy algorithm has an accuracy of 94% in finding the reordering with the lowest cost.

The rest of this paper is organized as follows: Section 2 presents the required background and related works. Section 3 describes the proposed query reordering

[5] `https://github.com/TYannakis/SPARQL-LD-Query-Optimizer`

methods. Section 4 reports experimental results. Finally, Section 5 concludes the paper and discusses interesting directions for future work.

2 Background and Related Literature

2.1 SPARQL-LD

The `SERVICE` operator of SPARQL 1.1 (`SERVICE` a P) is defined as a graph pattern P evaluated in the SPARQL endpoint specified by the URI a, while (`SERVICE` $?X$ P) is defined by assigning to the variable $?X$ all the URIs (of endpoints) coming from partial results, i.e. that get bound after executing an initial query fragment [5]. The idea behind SPARQL-LD is to enable the evaluation of a graph pattern P not absolutely to a SPARQL endpoint a, but generally to an RDF graph G_r specified by a Web Resource r. Thus, now a URI given to the `SERVICE` operator can also be the dereferenceable URI of a resource, the Web page of an entity (e.g., of a person), an ontology (OWL), Turtle or N3 file, or even the URL of a service that dynamically creates and returns RDF data. In case the URI is not the address of a SPARQL endpoint, the RDF data that may exist in the resource are fetched at real-time and queried for the graph pattern P. Currently, SPARQL-LD supports a variety of standard formats, including RDF/XML, N-Triples, N3/Turtle, RDFa, JSON-LD, Microdata, Microformats [7, 8].

SPARQL-LD is a generalization of SPARQL 1.1 in the sense that every query that can be answered by SPARQL 1.1 can be also answered by SPARQL-LD. Specifically, if the URI given to the `SERVICE` operator corresponds to a SPARQL endpoint, then it works exactly as the original SPARQL 1.1 (the remote endpoint evaluates the query and returns the result). Otherwise, instead of returning an error (and no bindings), it tries to fetch and query the triples that may exist in the given resource. SPARQL-LD has been implemented using Apache Jena [1], an open source Java framework for building Semantic Web applications. The implementation is available as open source[6].

Listing 1 shows a query that can be answered by SPARQL-LD. The query returns all co-authors of Pavlos Fafalios together with the number of their publications and the number of distinct conferences in which they have a publication. The query first accesses the RDFa-embedded web page of Pavlos Fafalios to collect his co-authors, then queries a SPARQL endpoint over DBLP to retrieve the conferences, and finally accesses the URI of all co-authors to gather their publications. Notice that the co-author URIs derive at query-execution time. In the same query, one could further integrate data from any other web resource, or from a web API which can return results in a standard RDF format.

The query in Listing 1 is answered within a few seconds. However, if we change the order of the first two `SERVICE` patterns, then its execution time is dramatically increased to many minutes. To cope with this problem, in this paper we propose methods to reorder the query's `SERVICE` patterns and thus improve the query execution time in case of non optimal query formulation.

[6] `https://github.com/fafalios/sparql-ld`

```
SELECT DISTINCT ?authorURI (count(distinct ?paper) AS ?numOfPapers)
                           (count(distinct ?series) AS ?numOfDiffConfs) WHERE {
  SERVICE <http://l3s.de/~fafalios/> { ?p <http://purl.org/dc/terms/creator> ?authorURI }
  SERVICE <http://dblp.l3s.de/d2r/sparql> {
     ?p2 <http://purl.org/dc/elements/1.1/creator> ?authorURI .
     ?p2 <http://swrc.ontoware.org/ontology#series> ?series }
  SERVICE ?authorURI { ?paper <http://purl.org/dc/elements/1.1/creator> ?authorURI }
} GROUP BY ?authorURI ORDER BY DESC(?numOfPapers)
```

Listing 1. Example SPARQL query that can be answered by SPARQL-LD.

2.2 Related Works

SPARQL Endpoint Federation

The idea of *query federation* is to provide integrated access to distributed sources on the Web. DARQ [18] and SemWIQ [12] are two of the first systems to support SPARQL query federation to multiple SPARQL endpoints. They provide access to distributed RDF data sources using a mediator service that transparently distributes the execution of queries to multiple endpoints. Given the need to address query federation, in 2013 the SPARQL W3C working group proposed a query federation extension for SPARQL 1.1 [17]. Buil-Aranda et al. [5] describe the syntax of this extension, formalize its semantics, and implement a static optimization for queries that contain the OPTIONAL operator, the most costly operator in SPARQL.

There is also a plethora of query federation engines to support efficient SPARQL query processing to multiple endpoints. The work in [20] provides a comprehensive analysis, comparison, and evaluation of a large number of SPARQL endpoint federation systems.

The ANAPSID system [3] adapts query execution schedulers to data availability and run-time conditions. It stores information about the available endpoints and the ontologies used to describe the data in order to decompose queries into sub-queries that can be executed by the selected endpoints, while adaptive physical operators are executed to produce answers as soon as responses from the available remote sources are received. The *query optimizer* component of ANAPSID exploits statistics about the distribution of values in the different datasets in order to identify the best combination of sub-queries.

The work in [15] proposes a heuristic-based approach for endpoint federation. Basic graph patterns are decomposed into sub-queries that can be executed by the available endpoints, while the endpoints are described in terms of the list of predicates they contain. Similar to ANAPSID, sub-queries are combined in a bushy tree execution plan, while the SPARQL 1.1 federation extension is used to specify the URL of the endpoint where the sub-query will be executed.

SPLENDID [10] is another endpoint federation system which relies on statistical data obtained from VoID descriptions [4]. For triple patterns with bound variables not covered in the VoID statistics, SPLENDID sends ASK queries to all the pre-selected data sources and removes those which fail the test. Bind and hash joins are used to integrate the results of the sub-queries, while a dynamic

programming strategy is exploited to optimize the join order of SPARQL basic graph patterns.

ADERIS [13] is a query processing system for efficiently joining data from multiple distributed endpoints. ADERIS decomposes federated SPARQL queries into multiple source queries and integrates the results through an adaptive join reordering method for which a cost model is defined.

The FedX framework [21] provides join processing and grouping techniques to minimize the number of requests to remote endpoints. Source selection is performed without the need of preprocessed metadata. It relies on SPARQL ASK queries and a cache which stores the most recent ASK requests. The input query is forwarded to all of the data sources and those sources which pass the SPARQL ASK test are selected. FedX uses a rule-based join optimizer which considers the number of bound variables. One of the methods we examine in this paper (UVC) is also based on the same heuristic.

Regarding more recent works, SemaGrow [6] is a federated SPARQL querying system that uses metadata about the federated data sources to optimize query execution. The system balances between a query optimizer that introduces little overhead, has appropriate fall backs in the absence of metadata, but at the same time produces optimal plans in many situations. It also exploits non-blocking and asynchronous stream processing to achieve efficiency and robustness.

Finally, Odyssey [14] is a cost-based query optimization approach for endpoint federation. It defines statistics for representing both entities and links among datasets, and uses the computed statistics to estimate the size of intermediate results. It also exploits dynamic programming to produce an efficient query execution plan with a low number of intermediate results.

Our approach. In this work, we focus on optimizing SPARQL 1.1 and SPARQL-LD queries through plain *query reordering*. The input is a query containing two or more `SERVICE` patterns, and the output is a near-optimal (in terms of query execution time) reordering of the contained `SERVICE`s, i.e., an optimized *reordered* query. Given the dynamic nature of Linked Data as well as the advanced query capabilities offered by SPARQL-LD (enabling to query any remote HTTP resource containing or returning RDF data), we aim at providing a general query reordering method that does not require statistics or metadata from the remote resources and that, contrary to the aforementioned works, can be directly applied on top of existing SPARQL 1.1 and SPARQL-LD implementation.

Selectivity-based Query Optimization

Another line of research has investigated optimization methods for non-federated SPARQL queries based on selectivity estimation.

The work in [23] defines and analyzes heuristics for selectivity-based basic graph pattern optimization. The heuristics range from simple triple pattern variable counting to more sophisticated selectivity estimation techniques that consider pre-computed triple pattern statistics. Likewise, [24] describes a set of

heuristics for deciding which triple patterns of a SPARQL query are more selective and thus it is in the benefit of the planner to evaluate them first. The planner tries to maximize the number of merge joins and reduce intermediate results by choosing triples patterns most likely to have high selectivity. [22] extends these works by considering more SPARQL expressions, in particular the operators `FILTER` and `GRAPH`.

In [11] the authors study the star and chain patterns with correlated properties and propose two methods for estimating their selectivity based on precomputed statistics. For star query patterns, Bayesian networks are constructed to compactly represent the joint probability distribution over values of correlated properties, while for chain query patterns the chain histogram is built which can obtain a good balance between the estimation accuracy and space cost.

Our approach. Similar to [23], [24] and [22], we exploit *heuristics* for selectivity estimation. However, we focus on reordering a set of `SERVICE` graph patterns in order to optimize the execution of SPARQL 1.1 and SPARQL-LD queries. Some of the heuristics we examine in this paper are based on the results of these previous works.

3 Query Reordering

We first model query reordering as a *cost minimization* problem (Section 3.1). Then we describe four heuristics-based methods for computing the cost of a `SERVICE` graph pattern (Section 3.2). We also discuss how we handle some special query cases (Section 3.3). At the end we motivate the need for a greedy algorithm for computing a near-optimal reordering for cases of large number of `SERVICE` graph patterns (Section 3.4).

3.1 Problem Modeling

Let Q be a SPARQL query and let $S = (s_1, s_2, \ldots, s_n)$ be a *sequence* of n `SERVICE` patterns contained in Q. For a service pattern s_i, let g_i be its nested graph pattern and B_i be the list of bindings of Q *before* the execution of s_i. Our objective is to compute a reordering S' of S that minimizes its *execution cost*. Formally:

$$R^* = \underset{S'}{\operatorname{argmin}}\, cost(S') \tag{1}$$

In our case, the *execution cost* of a sequence of `SERVICE` patterns S' corresponds to its total execution time. However, the execution time of a `SERVICE` pattern $s_i \in S'$ highly depends on the query patterns that precede s_i, while the bindings produced by s_i affect the execution time of the succeeding `SERVICE` patterns. Considering the above, we can estimate $cost(S')$ as the weighted sum of the cost of each service pattern $s_i \in S'$ given B_i. Formally:

$$cost(S') = \sum_{i=1}^{n} (cost(s_i|B_i) \cdot w_i) \tag{2}$$

where $cost(s_i|B_i)$ expresses the cost of SERVICE pattern s_i given B_i (i.e., given the already-bound variables before executing s_i), and w_i is the weight of SERVICE pattern s_i which expresses the degree up to which it influences the execution time of the sequence S'. We define $w_i = \frac{n-i+1}{n}$. In this case, for a sequence of four SERVICE patterns $S' = (s_1, s_2, s_3, s_4)$, the weights are: $w_1 = 1.0$ (since s_1 influences the execution time of 3 SERVICE patterns), $w_2 = 0.75$ (s_2 affects 2 SERVICE patterns), $w_3 = 0.5$ (s_3 affects 1 SERVICE pattern), and $w_4 = 0.25$ (s_4 does not affect any other SERVICE pattern).

Now, the cost of each SERVICE pattern s_i can be estimated based on the *selectivity/restrictiveness* of its graph pattern g_i given B_i. Formally:

$$cost(s_i|B_i) = unrestrictiveness(g_i|B_i) \tag{3}$$

A SERVICE graph pattern that is very unrestrictive will return a large number of intermediate results (large number of bindings), which in turn will increase the number of calls to succeeding SERVICE patterns, resulting in higher total execution time. In the query of Listing 1 for example, a large number of bindings of the variables in the first SERVICE pattern will result in many calls of the second SERVICE. Thus, our objective is to first execute the more restrictive SERVICE patterns that will probably return small result sets.

As proposed in [23] and [24] (for the case of triple patterns), the restrictiveness of a graph pattern can be determined by the *number* and *type* of new (unbound) variables in the graph pattern. The most restrictive graph pattern can be considered the one containing the less unbound variables (since fewer bindings are expected). Regarding the type of the unbound variables, subjects can be considered more restrictive than objects, and objects more restrictive than predicates (usually there are more triples matching a predicate than a subject or an object, and more triples matching an object than a subject) [23]. Moreover, the number and type of joins can also affect the restrictiveness of a graph pattern since, for example, an unusual subject-predicate join will probably return less bindings. Finally, literals and filter operators usually restrict the number of bindings and thus increase the restrictiveness of a graph pattern. Below, we define formulas for *unrestrictiveness* that consider the above factors.

3.2 Methods for Estimating Unrestrictiveness

We examine four methods for computing the *unrestrictiveness cost* (Equation 3) of a SERVICE graph pattern:

- I. Variable Count (VC)
- II. Unbound Variable Count (UVC)
- III. Weighted Unbound Variable Count (WUVC)
- IV. Joins-aware Weighted Unbound Variable Count (JWUVC)

I. Variable Count (VC). The first unrestrictiveness measure simply considers the number of graph pattern variables without considering whether they are bound or not. For a given graph pattern g_i, let $V(g_i)$ be the set of variables of

g_i. The unrestrictiveness of g_i can be now defined as:

$$unrestrictiveness(g_i|B_i) = |V(g_i)| \tag{4}$$

With the above formula, more variables in a graph pattern means higher unrestrictiveness score. Consider for example the query in Listing 2. The second `SERVICE` pattern contains one variable and is more likely to retrieve a smaller number of results than the first one which contains three variables. Thus the second `SERVICE` pattern is more restrictive and should be executed first.

```
1 SELECT * WHERE {
2   SERVICE <http://resource1> { ?s ?p ?o }
3   SERVICE <http://resource2> { ?s a :fish } }
```

Listing 2. Example SPARQL query for VC reordering.

II. Unbound Variable Count (UVC). A `SERVICE` pattern containing many new unbound variables is more likely to retrieve a higher number of results compared to a `SERVICE` pattern with less unbound variables. Thereby, we can also consider the set of binding B_i before the execution of a `SERVICE` pattern s_i. Let first $V^u(g_i, B_i)$ be the set of new (unbound) variables of g_i given B_i. The unrestrictiveness of g_i can be now defined as:

$$unrestrictiveness(g_i|B_i) = |V^u(g_i, B_i)| \tag{5}$$

Listing 3 shows an example for this case. After the execution of the first `SERVICE` pattern, we should better run the third one since all its variables are already bound. The second `SERVICE` pattern contains one unbound variable, although its total number of variables is less than those of the third `SERVICE` pattern.

```
1  SELECT * WHERE {
2    SERVICE <http://resource1> {
3      <http://entity1> :birthPlace ?place1 ; :friend ?entity2 ; :workPlace ?place2 }
4    SERVICE <http://resource2> { ?entity2 a ?type }
5    SERVICE <http://resource3> { ?entity2 :birthPlace ?place1 ; :workPlace  ?place2 } }
```

Listing 3. Example SPARQL query for UVC reordering.

III. Weighted Unbound Variable Count (WUVC). The above formulas do not consider the type of the unbound variables in the graph pattern, i.e., whether they are in the subject, predicate or object position in the triple pattern. For a graph pattern g_i and a set of bindings B_i, let $V_s^u(g_i, B_i)$, $V_p^u(g_i, B_i)$ and $V_o^u(g_i, B_i)$ be the set of subject, predicate and object unbound variables in g_i, respectively. Let also w_s, w_p and w_o be the weights for subject, predicate and object variables, respectively. The unrestrictiveness of g_i can be now defined as:

$$unrestrictiveness(g_i|B_i) = |V_s^u(g_i, B_i)| \cdot w_s + |V_p^u(g_i, B_i)| \cdot w_p + |V_o^u(g_i, B_i)| \cdot w_o \tag{6}$$

According to [23], subjects are in general more restrictive than objects and objects are more restrictive than predicates, i.e., there are usually more triples

matching a predicate than an object, and more triples matching an object than a subject. When considering variables, selectivity is opposite: a subject variable may return more bindings than an object variable and an object variable more bindings than a predicate variable. Consider for example the query in Listing 4. The subjects having *Greece* as the birth place (1st `SERVICE` pattern) are expected to be more than the friends of *George* (2nd `SERVICE` pattern), while the friends of *George* are expected to be more than the different properties that connect *George* with *Nick* (3rd `SERVICE` pattern). Thus, one can define weights so that $w_s > w_o > w_p$. Based on the distribution of subjects, predicates and objects in a large Linked Data dataset of more than 28 billion triples (gathered from more than 650 thousand sources) [9], we define the following weights: $w_s = 1.0$, $w_o = 0.8$, $w_p = 0.1$. Moreover, if a variable exists in more than one triple pattern position (e.g., both as subject or object), we consider it as being in the more restrictive position.

```
SELECT * WHERE {
  SERVICE <http://resource1> { ?entity1 :birthPlace :Greece }
  SERVICE <http://resource2> { <http://George> :friend ?entity1 }
  SERVICE <http://resource3> { <http://George> ?p <http://Nick> } }
```

Listing 4. Example SPARQL query for WUVC reordering.

IV. Joins-aware Weighted Unbound Variable Count (JWUVC). When a graph pattern contains joins, its restrictiveness is usually increased depending on the number and type of joins (star, chain, or unusual join) [11]. For a graph pattern g_i, let $J_*(g_i)$, $J_\rightarrow(g_i)$, and $J_\times(g_i)$ be the number of star, chain, and unusual joins in g_i, respectively. We consider the subject-subject and object-object joins as *star joins*, the object-subject and subject-object as *chain joins*, and all the others as *unusual joins*. Let also j_*, $j_\rightarrow$ and $j_\times$ be the weights for star, chain, and unusual joins, respectively. Based on the assumption that, in general, unusual joins are much more restrictive than chain joins, and chain joins are more restrictive than star joins [24], one can define weights so that $j_\times > j_\rightarrow > j_*$. We define: $j_\times = 1.0$, $j_\rightarrow = 0.6$, $j_* = 0.5$. The following unrestrictiveness formula considers both the number and the type of joins in the graph pattern g_i:

$$unrestrictiveness(g_i|B_i) = \frac{|V_s^u(g_i, B_i)| \cdot w_s + |V_p^u(g_i, B_i)| \cdot w_p + |V_o^u(g_i, B_i)| \cdot w_o}{1 + J_*(g_i) \cdot j_* + J_\rightarrow(g_i) \cdot j_\rightarrow + J_\times(g_i) \cdot j_\times} \tag{7}$$

Listing 5 shows an example for this case. The first `SERVICE` pattern contains a star join, the second a chain join, and the third an unusual join. The unusual join will probably return fewer results than the star and chain joins.

```
SELECT * WHERE {
  SERVICE <http://resource1> { ?ent1 :birthPlace :Greece ; :workPlace :Germany }
  SERVICE <http://resource2> { <http://George> :friend ?ent1 . ?ent1 :friend <http://Nick> }
  SERVICE <http://resource3> { <http://George> ?p <http://Nick> . ?p :label "best friend" } }
```

Listing 5. Example SPARQL query for JWUVC reordering.

In Section 4 we evaluate the effectiveness of the above four methods on finding the optimal, in terms of query execution time, query reordering.

3.3 Handling of Special Cases

Query plans with same cost. In case the lowest unrestrictiveness cost is the same for two or more query reorderings, we consider the number of *literals* and *filter* operators contained in the graph patterns. Literals are generally considered more selective than URIs [24], while a filter operator limits the bindings of the filtered variable and thus increases the selectivity of the corresponding graph pattern [23]. Thus, we count the total number of literals and filter operators in each `SERVICE` pattern, and consider it when we get query plans with the same unrestrictiveness cost. If the corresponding `SERVICE` patterns contain the same number of literals and filter operators, then we maintain their original ordering, i.e., we order them based on their order in the input query.

SERVICE within OPTIONAL. In case a `SERVICE` call is within an *optional* pattern, then we separately reorder the `SERVICE` patterns that exist before and after it. An optional pattern requires a left outer join and thus changing its order can distort the query result.

Variable in SERVICE clause. If a `SERVICE` clause contains a variable instead of a URI, we should ensure that this variable gets bound before the execution of the `SERVICE` pattern. Thereby, during reordering we ensure that all other `SERVICE`s containing this variable in their graph patterns are placed before the `SERVICE` pattern having the variable in its clause.

Projection variables. The set of variables that appear in the `SELECT` clause of a `SERVICE` pattern are called the projection variables. Since these are part of the answer and affect the size of the bindings, we only consider these variables in all the proposed formulas.

UNION operator, nested patterns, combination of triple and SERVICE patterns. In this work we do not study the case of queries containing the `UNION` operator as well as nested patterns. Such queries require the reordering of groups of `SERVICE` patterns which is not currently supported by our implementation. In addition, our implementation does not yet support the case of queries containing both triple patterns (that query the "local" endpoint) and `SERVICE` patterns. We leave the handling of these cases as part of our future work.

3.4 Computing a near-optimal query-execution plan

Computing the unrestrictiveness score for all the different query reorderings may be prohibitive for large number of `SERVICE` patterns, since the complexity is $n!$ (where n is the query's number of `SERVICE` patterns). This applies in all the proposed optimization methods apart from VC where no all permutations are needed to be computed. For example, for queries with 5 `SERVICE` patterns there are 5! (=720) different permutations, however for 10 `SERVICE` patterns this

number is increased to more than 3.6 million permutations and for 15 to around 1.3 trillion.

Table 1 shows the time required for computing the reordering with the lowest cost for different number of SERVICE patterns using the JWUVC method (the time is almost the same for also UVC and WUVC). Our implementation (cf. Footnote 5) is in Java and uses Apache Jena for decomposing the SPARQL query, while we run the experiments in an ordinary computer with processor Intel Core i5 @ 3.2Ghz CPU, 8GB RAM and running Windows 10 (64 bit).

Table 1. Time to compute the reordering with the lowest cost for different number of SERVICE patterns in a SPARQL query.

Number of SERVICE patterns	Time
5	8 ms
6	22 ms
7	89 ms
8	290 ms
9	2.4 sec
10	25 sec
11	6 min
12	67 min
13	>5 hours
14	>5 hours

We see that the time is very high for queries with many SERVICE patterns. For example, more than 1 hour is required for just finding the reordering with the lowest cost for a query with 12 SERVICE patterns. This illustrates the need for a cost-effective approach which can find a near-optimal query execution plan without needing to check all the different permutations. We adopt a greedy algorithm starting with the SERVICE pattern with the smaller unrestrictiveness score (local optimal choice) and continuing with the next SERVICE pattern with the smaller score, considering at each stage the already bound variables of the previous stages. To find the local optimal choice, we can use any of the proposed unrestrictiveness formulas. Considering the UVC formula for example, in the query of Listing 6 the greedy algorithm first selects the 2nd SERVICE pattern since it contains only 1 variable. In the next stage, it selects the 3rd SERVICE pattern which contains 2 unbound variables, fewer than those of the 1st SERVICE pattern.

```
1 SELECT * WHERE {
2  SERVICE <http://resource1> { ?ent1 :birthPlace ?place1 ; :workPlace ?place2 ; :friend ?ent2 }
3  SERVICE <http://resource2> { ?ent2 a :Actor }
4  SERVICE <http://resource3> { ?ent2 :birthPlace ?place1 ; :workPlace  ?place2 } }
```

Listing 6. Example SPARQL query for choosing a near-optimal query plan.

4 Evaluation

We evaluated the effectiveness of the proposed query reordering methods using real federated queries from the *LargeRDFBench* [19] dataset[7]. From the provided 32 SPARQL 1.1 queries, we did not consider 10 queries that make use of the `UNION` operator (it is not currently supported by our implementation) and 5 "large data" queries (due to high memory requirements). To consider larger number of possible query permutations, and since some of the queries contain only 2 `SERVICE` patterns, we removed the `OPTIONAL` operators keeping though the embedded `SERVICE` pattern(s).[8] For instance, we transformed the query:

```
SELECT * WHERE { SERVICE <ex1> {..} OPTIONAL { SERVICE <ex2> {..} } }
```

to the query:

```
SELECT * WHERE { SERVICE <ex1> {..} SERVICE <ex2> {..} }
```

The final evaluation dataset contains 17 queries of varying complexity (each one containing at least two `SERVICE` patterns), while their `SERVICE` patterns require access to totally 7 remote SPARQL endpoints. Note that there is no benchmark for SPARQL-LD, however this does not affect the objective of our evaluation since the proposed methods do not distinguish between SPARQL 1.1 and SPARQL-LD queries (a SPARQL endpoint can be considered an HTTP resource containing all the endpoint's triples).

For each query, we found the optimal reordering by computing the execution time of all possible permutations (average of 5 runs). Then, we examined the effectiveness of the proposed optimization methods (VC, UVC, WUVC, and JUWVC, as described in Section 3.2) on finding the optimal query execution plan. Figure 1 shows the results. VC finds the optimal query plan in 8/17 queries (47%), UVC in 10/17 queries (59%), WUVC in 9/17 queries (53%), and JUWVC in 15/17 queries (88%). We notice that the JUWVC method, which considers the number and type of joins, achieves a very good performance. Given the infrastructure used to host the SPARQL endpoints in our experiments[9], query reordering using JUWVC achieves a very large decrease of the query execution time for many of the queries (for example, from minutes to some seconds for the queries S4, S10, S12, C7, C10).

JUWVC fails to find the optimal query plan for the queries S13 and C6, which both contain 2 `SERVICE` patterns. The first `SERVICE` pattern of S13 contains 1 star join and the second 2 star joins. As regards C6, its first `SERVICE` pattern contains 1 star join and 1 chain join, and its second 5 star joins. In both queries, although the second `SERVICE` pattern contains more joins than the first `SERVICE` pattern, it returns larger number of bindings and this increases the number of calls to the first remote endpoint and thus the overall query execution time. Note that, without exploiting dataset statistics, such cases are very difficult to be caught by an unrestrictiveness formula.

[7] `https://github.com/dice-group/LargeRDFBench`

[8] Although this transformation changes the query results, it does not affect the objective of our evaluation.

[9] 2x Intel Xeon CPU E5-2630 @ 2.30GHz, 6-core, 384GB RAM.

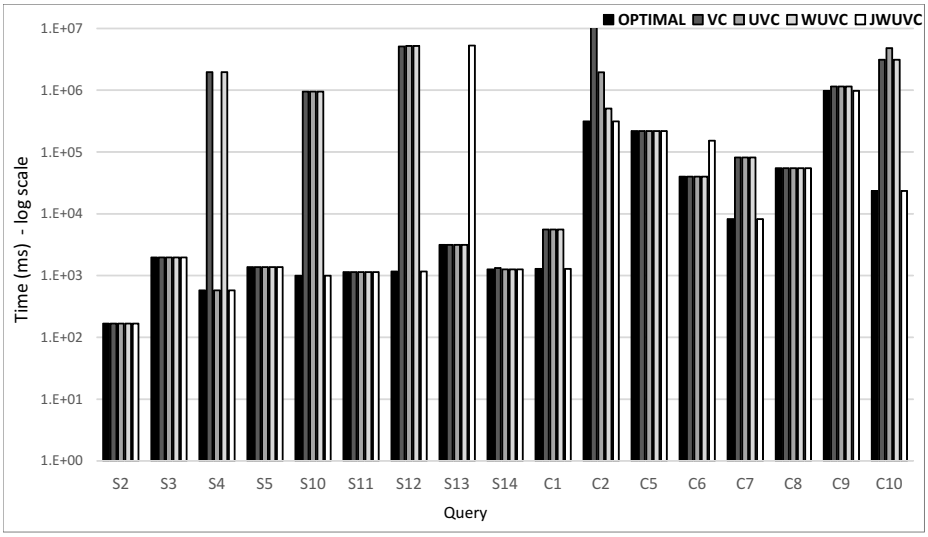

Figure 1. Effectiveness of the different query reordering methods (VC: variable count; UVC: unbound variable count; WUVC: weighted unbound variable count; JWUVC: joins-aware weighted unbound variable count).

As regards the effectiveness of the greedy algorithm which avoids computing the cost of all possible permutations (cf. Section 3.4), it manages to find the reordering with the lowest cost using JUWVC in 16/17 queries (94%). It fails for the query C2, however the returned reordering is very close to the optimal (the difference is only a few milliseconds).

One of the limitations of such a fully heuristics-based method is that it is practically impossible to always find the optimal query plan. However, this is the case also for methods that pre-compute and exploit metadata and statistics from the remote resources, or which make use of caching. The reason is that the Web of Data is a huge and constantly evolving information space, meaning that we may always need to query a new, unknown resource discovered during query execution. A solution to this problem is the exploitation of VoID [4], in particular the publishing of a rich VoID file alongside each resource. In this case, an optimizer can access (and exploit for query reordering) such VoID descriptions at query execution time, considering though that all publishers follow a common pattern for publishing these VoID files.[10]

5 Conclusion

We have proposed and evaluated a set of fully heuristics-based query reordering methods for federating queries in SPARQL 1.1 and SPARQL-LD. The proposed methods reorder a set of `SERVICE` graph patterns based on their selectivity (restrictiveness) and do not require the gathering and use of statistics or metadata

[10] `https://www.w3.org/TR/void/#void-file`

from the remote resources. Such an approach is widely-applicable and can be exploited on top of existing SPARQL 1.1 and SPARQL-LD implementations.

Since the new query functionality offered by SPARQL-LD (allowing to query any HTTP resource containing RDF data) can lead to queries with large number of `SERVICE` patterns which in turn can dramatically increase the time to find the optimal reordering, we proposed the use of a simple greedy algorithm for finding a near-optimal query execution plan without checking all possible query reorderings. The results of an experimental evaluation using an existing benchmark showed that a query reordering method which considers the number and type of unbound variables and the number and type of joins achieves the optimal query plan in 88% of the examined queries, resulting in a large decrease of the overall query execution time (from minutes to a few seconds in many cases). Regarding the greedy algorithm, it has an accuracy of 94% in finding the reordering with the lowest cost.

As part of our future work, we plan to offer a holistic query reordering approach which will cover any type of federated queries. This involves the handling of queries containing `UNION` and nested graph patterns, as well as queries which combine triple and `SERVICE` patterns. We also plan to offer this query reordering functionality as a web service, allowing for on-the-fly query optimization.

Acknowledgements

The work was partially funded by the European Commission for the ERC Advanced Grant ALEXANDRIA under grant No. 339233.

References

1. Apache Jena. `http://jena.apache.org/`
2. SPARQL 1.1 Query Language (W3C). `http://www.w3.org/TR/sparql11-query/`
3. Acosta, M., Vidal, M.E., Lampo, T., Castillo, J., Ruckhaus, E.: ANAPSID: an adaptive query processing engine for SPARQL endpoints. In: International Semantic Web Conference. pp. 18–34. Springer (2011)
4. Alexander, K., Hausenblas, M.: Describing linked datasets-on the design and usage of VoID, thevocabulary of interlinked datasets. In: Linked Data on the Web Workshop (LDOW'09). Citeseer (2009)
5. Buil-Aranda, C., Arenas, M., Corcho, O., Polleres, A.: Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. Web Semantics: Science, Services and Agents on the World Wide Web **18**(1) (2013)
6. Charalambidis, A., Troumpoukis, A., Konstantopoulos, S.: SemaGrow: Optimizing federated SPARQL queries. In: Proceedings of the 11th International Conference on Semantic Systems. pp. 121–128. ACM (2015)
7. Fafalios, P., Tzitzikas, Y.: SPARQL-LD: A SPARQL Extension for Fetching and Querying Linked Data. In: The Semantic Web–ISWC 2015 (Posters & Demonstrations Track). Bethlehem, Pennsylvania, USA (2015)
8. Fafalios, P., Yannakis, T., Tzitzikas, Y.: Querying the Web of Data with SPARQL-LD. In: International Conference on Theory and Practice of Digital Libraries. pp. 175–187. Springer (2016)

9. Fernández, J.D., Beek, W., Martínez-Prieto, M.A., Arias, M.: LOD-a-lot. In: International Semantic Web Conference. pp. 75–83. Springer (2017)
10. Görlitz, O., Staab, S.: SPLENDID: SPARQL endpoint federation exploiting VoID descriptions. In: Proceedings of the Second International Conference on Consuming Linked Data-Volume 782. pp. 13–24. CEUR-WS. org (2011)
11. Huang, H., Liu, C.: Estimating selectivity for joined RDF triple patterns. In: 20th ACM international conference on Information and knowledge management. pp. 1435–1444. ACM (2011)
12. Langegger, A., Wöß, W., Blöchl, M.: A semantic web middleware for virtual data integration on the web. In: 5th ESWC. Springer-Verlag (2008)
13. Lynden, S., Kojima, I., Matono, A., Tanimura, Y.: ADERIS: An adaptive query processor for joining federated SPARQL endpoints. In: On the Move to Meaningful Internet Systems: OTM 2011. pp. 808–817. Springer (2011)
14. Montoya, G., Skaf-Molli, H., Hose, K.: The Odyssey approach for optimizing federated SPARQL queries. In: International Semantic Web Conference. pp. 471–489. Springer (2017)
15. Montoya, G., Vidal, M.E., Acosta, M.: A heuristic-based approach for planning federated SPARQL queries. In: Proceedings of the Third International Conference on Consuming Linked Data-Volume 905. pp. 63–74. CEUR-WS. org (2012)
16. Mühleisen, H., Bizer, C.: Web data commons-extracting structured data from two large web corpora. LDOW **937**, 133–145 (2012)
17. Prud'hommeaux, E., Buil-Aranda, C., et al.: SPARQL 1.1 federated query. W3C Recommendation **21**, 113 (2013)
18. Quilitz, B., Leser, U.: Querying distributed RDF data sources with SPARQL. In: 5th ESWC. Springer (2008)
19. Saleem, M., Hasnain, A., Ngomo, A.C.N.: LargeRDFBench: a billion triples benchmark for SPARQL endpoint federation. Journal of Web Semantics (2018)
20. Saleem, M., Khan, Y., Hasnain, A., Ermilov, I., Ngonga Ngomo, A.C.: A fine-grained evaluation of SPARQL endpoint federation systems. Semantic Web **7**(5), 493–518 (2016)
21. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization techniques for federated query processing on linked data. In: The Semantic Web–ISWC 2011. Springer (2011)
22. Song, F., Corby, O.: Extended Query Pattern Graph and Heuristics-based SPARQL Query Planning. Procedia Computer Science **60**, 302–311 (2015)
23. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th international conference on World Wide Web. pp. 595–604. ACM (2008)
24. Tsialiamanis, P., Sidirourgos, L., Fundulaki, I., Christophides, V., Boncz, P.: Heuristics-based query optimisation for SPARQL. In: 15th International Conference on Extending Database Technology. pp. 324–335. ACM (2012)