A Heuristic-Based Approach for Planning Federated SPARQL Queries

Gabriela Montoya[1], Maria-Esther Vidal[1], and Maribel Acosta[1,2]

[1] Universidad Simón Bolívar, Venezuela
{gmontoya, mvidal, macosta}@ldc.usb.ve
[2] Institute AIFB, Karlsruhe Institute of Technology, Germany
maribel.acosta@kit.edu

Abstract. A large number of SPARQL endpoints are available to access the Linked Open Data cloud, but query capabilities still remain very limited. Thus, to support efficient semantic data management of federations of endpoints, existing SPARQL query engines require to be equipped with new functionalities. First, queries need to be decomposed into sub-queries not only answered by the available endpoints, but also executable in a way that the bandwidth usage is minimized. Second, query engines have to be able to gather the answers produced by the endpoints and merge them following a plan that reduces intermediate results. We address these problems and propose techniques that only rely on information about the predicates of the datasets accessible through the endpoints, to identify bushy plans comprise of sub-queries that can be efficiently executed. These techniques have been implemented on top of one existing RDF engine, and their performance has been studied on the FedBench benchmark. Experimental results show that our approach may support successful evaluation of queries, when other federated query engines fail, either because endpoints are unable to execute the sub-queries or federated query plans are too expensive.

1 Introduction

Over the past decade, the number of datasets in the Linked Open Data cloud has exploded as well as the number of SPARQL endpoints[1]. Although in theory endpoints should be able to execute any SPARQL query, many requests may be unsuccessful and time out without producing any answer. This undesirable behavior is mainly caused by limited query capabilities that characterize existing endpoints. For example, the majority of them are designed for very lightweight use, say, to execute queries for just two minutes, or they may be unable to retrieve data from other endpoints. Accordingly, some endpoints reject the execution of queries whose estimated execution time is greater than a certain number, while others simply time out without producing any answer. However, real-world queries may be complex and require gathering data from different and distant endpoints. Therefore, there is a need to develop techniques

[1] http://labs.mondeca.com/sparqlEndpointsStatus/

to decompose complex queries into sub-queries that can be efficiently executed by the existing endpoints in a way that the bandwidth usage is minimized, as well as, strategies to efficiently merge the retrieved data.

So far several approaches have addressed the problem of decomposing a SPARQL query into sub-queries that can be executed by existing endpoints [1, 2, 5, 9, 11]. Some approaches rely the decision on statistics collected from the sources [5] or simply consider all possible sub-queries and choose the most promising ones [2]. Other methods implement heuristic-based solutions to identify the sub-queries that can be executed by the available sources or endpoints [1, 11]. FedX, for example, is a rule-based system able to generate left-linear plans comprised of sub-queries that can be exclusively answered by existing endpoints. FedX does not derive the query decomposition decision on knowledge about schema alignments or data distributions. Although these approaches may succeed when existing endpoints fail, they may still exhibit problems. Particularly, performance may be deteriorated, if queries with a large number of triples patterns are executed or large intermediate results are retrieved from the endpoints.

In this work we address the problem of executing SPARQL 1.0 [2] queries with a large number of triple patterns, and devise a two-fold solution. First, sets of triple patterns that compose Basic Graph Patterns (BGPs) in the SPARQL 1.0 query, are decomposed into sub-queries that can be executed by available endpoints. Endpoints are then described in terms of the list of predicates of the RDF triples accessible through the endpoint. In a second step, sub-queries are combined in an execution plan that induces a bushy tree fashion execution. The SPARQL 1.1 federation extension [3] is used to specify the URL of the endpoint where a sub-query will be executed. Throughout the rest of the paper we refer to SPARQL 1.0 queries, but our approach can also handle queries in SPARQL 1.1; this would just simplify the problem and being only required the second step of our solution. We empirically analyze the performance of our approach, and show that our plans are competitive with the plans generated by state-of-the-art RDF engines. In addition, our techniques are able to execute queries with large BGPs and produce results when other engines may time out.

This paper is comprised of five additional sections. Section 2 gives a motivating example. Section 3 summarizes the related work. Section 4 presents our two-fold approach. Experimental results are reported in Section 5. Finally, we conclude in Section 6 with an outlook to future work.

2 Motivating Example

Example 1. Consider the following SPARQL 1.0 query: "Drugs and their components' url and image", see Listing 1.1.

[2] `http://www.w3.org/TR/rdf-sparql-query/`
[3] `http://www.w3.org/TR/2010/WD-sparql11-federated-query-20100601/`

Listing 1.1. Query LS5 from FedBench [10]

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX purl:<http://purl.org/dc/elements/1.1/>
PREFIX bio2rdf:<http://bio2rdf.org/ns/bio2rdf#>
SELECT $drug $keggUrl $chebiImage WHERE {
    $drug rdf:type drugbank:drugs .
    $drug drugbank:keggCompoundId $keggDrug .
    $drug drugbank:genericName $drugBankName .
    $keggDrug bio2rdf:url $keggUrl .
    $chebiDrug purl:title $drugBankName .
    $chebiDrug bio2rdf:image $chebiImage
}
```

The result set of this query is comprised of 393 tuples when data from Drugbank, KEGG and Chebi are retrieved. However, if this query is run against any of the existing endpoints, Drugbank[4] or KEGG[5] or Chebi[6], the answer is empty. This problem is caused by the need to traverse links between these datasets to answer the query. Still, the majority of endpoints have been created for lightweight use and they are not able to dereference data from other datasets. Another solution is the decomposition of the original query into portions that are executable by the endpoints and the combination of the up-to-date retrieved data. State-of-the-art approaches [4, 11] are able to decompose this query into sub-queries that can be exclusively executed by a single endpoint, and then, gather the results produced by these sub-queries to bind variables of other sub-queries. This query produces small intermediate results and contains only six triple patterns; thus, these techniques are quite effective and efficient, i.e., there is a good trade-off between completeness of the answer and time required to produce it.

Example 2. Consider a more complex query, i.e., with larger number of triple patterns that may generate a large set of intermediate results. SPARQL 1.0 query: "Drugs that interact with antibiotics, antiviral and antihypertensive agents", see Listing 1.2. Approaches that rely on exclusive groups [4, 11] time out after 30 minutes without producing any answer. We present an alternative decomposition and planning technique that overcomes this limitation. Queries are decomposed into simpler sub-queries which are executed in a bushy tree fashion to minimize intermediate results and to simplify the requests submitted to the endpoints. Thus, our query engine is able to scale up to queries with a large number of triple patterns connected by any SPARQL operator, and also to queries that may produce a large number of intermediate results.

[4] `http://www4.wiwiss.fu-berlin.de/drugbank/sparql`, July 2012.
[5] `http://www4.wiwiss.fu-berlin.de/drugbank/sparql`, July 2012.
[6] `http://chebi.bio2rdf.org/sparql`, July 2012.

Listing 1.2. Query C1

```
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX dbcategory: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugcategory/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbowl: <http://dbpedia.org/ontology/>
PREFIX kegg: <http://bio2rdf.org/ns/kegg#>
SELECT DISTINCT ?drug ?enzyme ?reaction  Where {
    ?drug1 drugbank:drugCategory dbcategory:antibiotics .
    ?drug2 drugbank:drugCategory dbcategory:antiviralAgents .
    ?drug3 drugbank:drugCategory dbcategory:antihypertensiveAgents .
    ?I1 drugbank:interactionDrug2 ?drug1 .
    ?I1 drugbank:interactionDrug1 ?drug .
    ?I2 drugbank:interactionDrug2 ?drug2 .
    ?I2 drugbank:interactionDrug1 ?drug .
    ?I3 drugbank:interactionDrug2 ?drug3 .
    ?I3 drugbank:interactionDrug1 ?drug .
    ?drug drugbank:keggCompoundId ?cpd .
    ?enzyme kegg:xSubstrate ?cpd .
    ?enzyme rdf:type kegg:Enzyme .
    ?reaction kegg:xEnzyme ?enzyme .
    ?reaction kegg:equation ?equation .
    ?drug5 rdf:type dbowl:Drug .
    ?drug owl:sameAs ?drug5 .}
```

3 Related Work

Several approaches have considered the problem of selecting query data providers. Harth et al. [5] present a hybrid solution that combines histograms and R-trees; histograms are used for source ranking, while regions determine the best sources to answer a basic graph pattern. Li and Heflin [9] propose a bottom-up tree based technique to integrate data from multiple heterogeneous sources, where the predicates in the basic graph pattern are used to determine the data source that will execute the graph pattern. Kaoudi et al. [7] propose a P2P system running on top of Atlas for processing RDF documents distributed and implements a cost-based approach to identify an optimal plan; statistics are kept by peers. These approaches can effectively identify data providers for evaluating a triple pattern but they may fail identifying high selective sub-queries particularly when general predicates, such as `rdf:type`, `owl:sameAs`, are used in the query.

The Semantic Web community has been also very active proposing solutions to the problem of query processing on the Web of Data [1, 2, 5, 7–9]. Some of these approaches combine source selection techniques with query execution strategies [5, 8], while others have developed frameworks to retrieve and manage Linked Data [2, 6, 7, 9]. Additionally, adaptive query processing solutions have been proposed to overcome limitations of the traditional optimize-then-execute paradigm in presence of unpredictable data transfer delays and data bursty arrivals [1, 2, 8]. Recently, Schwarte et al. [11] have proposed FedX, a rule-based system able to decompose SPARQL 1.0 queries into SPARQL 1.1 queries comprised of sub-queries that can be completely executed by an endpoint or *exclusive groups*; then, it relies on the number of bounded variables to decide the query join order; joins are evaluated in a block-nested loop fashion to reduce the number of source requests. FedX uses no knowledge about mappings and statistics

associated with the sources; it may contact every source to determine where the predicates presented in a query are offered, and may save this information in cache for future queries of the same predicate. Similarly, SPLENDID [4] exploits information encoded in endpoint descriptions to select endpoints and to identify a set of triple patterns that comprise an exclusive group. Statistics are used to find the relevant sources for triple patterns in a query. Endpoints may be contacted to decide where non-exclusive groups can be executed; cost-based optimization techniques are used to identify bushy tree plans. SPARQL-DQP [3] and ARQ [7] exploit information on SPARQL 1.1 queries to decide where the subqueries of triple patterns will be executed; additionally, they rely on statistics or heuristics to identify query plans. WoDQA [8] is a tool built on top of ARQ to provide access to federations of endpoints. Finally, Avalanche [2] implements an inter-operator solution where heuristically a group of best plans is chosen. Statistics about cardinalities and data distribution are considered to identify possibly good plans. Avalanche follows a competition strategy where top-k plans are executed in parallel, the output of a query corresponds to the answers produced by the most promising plan(s) in a given period of time. These approaches may efficiently identify endpoints to execute a query; however, when queries are comprised of large number of triple patterns, sub-queries may be non-selective. Thus, performance can be affected even in presence of perfect networks with no or negligible connection latency.

4 Our Approach

4.1 Selecting SPARQL Endpoints

Our approach heuristically selects among the endpoints that can evaluate a triple pattern, those ones that likely provide relevant data for the joins in which the triple pattern participates in the query. Endpoints are modeled as the result of executing the SPARQL query: `SELECT DISTINCT ?p where {?s ?p ?o}`. Table 1 illustrates predicates of datasets Drugbank, KEGG, Chebi and DBpedia; each endpoint is able to answer triple patterns with the corresponding predicate. Based on these endpoint descriptions, triple patterns of query from Example 1 can be answered by the endpoints indicated in Table 2. As can be observed,

Table 1. Endpoint Descriptions- Datasets Drugbank, KEGG, Chebi and DBpedia

Endpoint	Predicates
Drugbank	rdf:type, drugbank:keggCompoundId, drugbank:genericName, drugbank:drugCategory, drugbank:interactionDrug1, drugbank:interactionDrug2, owl:sameAs
KEGG	rdf:type, bio2rdf:url, purl:title, kegg:xSubstrate, kegg:xEnzyme, kegg:equation, owl:sameAs, bio2rdf:urlImage
Chebi	rdf:type, bio2rdf:url, purl:title, bio2rdf:image
DBpedia	rdf:type, owl:sameAs

[7] `http://jena.sourceforge.net/ARQ`
[8] `http://23.23.193.63/seagent/wodqa/wodqa.seam`

there is not doubt where triple patterns in lines 7, 8 and 11 can be evaluated. Nevertheless, the other triple patterns are answered by several endpoints, and the query decomposer needs to identify which ones provide relevant answers. Our heuristics are defined as follows:

Heuristic 1 ***Star-Shaped Group Multiple endpoint selection (SSGM)*****:** given a triple pattern t of the form $\{s\ p\ o\}$ from a query Q. *a*) If p is not a general bound predicate, i.e., it is not from the general domains[9], e.g., RDF(S) or OWL, t must be evaluated by the endpoint that can answer more predicates in the same namespace of p. *b*) If s or o are URIs, t must be evaluated by the endpoint that can answer more predicates in the same namespace of s or o. *c*) If s is a variable, then t must be evaluated in the endpoint that can answer more triple patterns in Q which share the same variable as subject. Note that rules b and c can be applied even if p is a variable.

We can apply rule b of **Heuristic 1** to triple pattern in line 6 of query from Example 1. This is because the object is a URI whose namespace (drugbank) is shared by predicates in the Drugbank endpoint. Thus, triple pattern in line 6 will be also evaluated by the Drugbank endpoint. Furthermore, we can apply rule a of **Heuristic 1**, to decide where to evaluate triple pattern in line 9. In this case, the predicate is bounded and is not general; however, both KEGG and Chebi offer the same number of predicates in the namespace of the triple pattern predicate[10]; thus, the query decomposer assigns both endpoints as possible data providers. Finally, data provider of triple pattern in line 10 can be selected by applying rule c of **Heuristic 1**. This triple pattern shares the subject variable with triple pattern in line 11 suggesting that the predicate should be evaluated by the Drugbank endpoint. Regarding to Example 2, endpoints for evaluating triple patterns in lines 8-18 and 20-21 can be unambiguously selected. However, triple patterns in lines 19 and 22 should be evaluated in KEGG and DBpedia, respectively, as suggested by rule b of **Heuristic 1**. Additionally, based on rule c of **Heuristic 1**, triple pattern in line 23 should be evaluated in Drugbank. Table 2 illustrates the endpoints selected for the triple patterns of queries from Examples 1 and 2. **Heuristic 1** will be needed if data is partitioned into different endpoints. Finally, we propose a second heuristic that selects only one endpoint to evaluate a triple pattern. It may lead to an incomplete answer. However, If data is replicated or can be completely accessible from all relevant endpoints, this heuristic will lead to an efficient and complete solution.

Heuristic 2 ***Star Shaped Group Single endpoint selection (SSGS)*****:** given a triple pattern t of the form $\{s\ p\ o\}$ from a query Q. Let SE be the set of endpoints where t can be evaluated, i.e., SE is obtained from **Heuristic 1**. Then, for each endpoint e in SE, an ASK query is used to check if e can evaluate t and all the triple patterns *assigned* to e by **Heuristic 2**. The decomposer selects the endpoint that first answers TRUE and *assigns* this endpoint to t. Thus, it is relevant the order in which endpoints are considered during the application of

[9] `http://labs.mondeca.com/dataset/lov/`, July 2012.

[10] Both KEGG and Chebi offer 7 predicates in this namespace.

Table 2. Endpoints that can evaluate triples from Queries from Examples 1 and 2; endpoints selected by **Heuristic 1** are highlighted in **bold**.

Query Example 1		
Line	Triple Pattern	Endpoint
6	$drug rdf:type drugbank:drugs	**Drugbank**, KEGG Chebi, DBpedia
7	$drug drugbank:keggCompoundId $keggDrug	**Drugbank**
8	$drug drugbank:genericName $drugBankName	**Drugbank**
9	$keggDrug bio2rdf:url $keggUrl	**KEGG**, **Chebi**
10	$chebiDrug purl:title $drugBankName	KEGG, **Chebi**
11	$chebiDrug bio2rdf:image $chebiImage	**Chebi**
Query Example 2		
Line	Triple Pattern	Endpoint
8	?drug1 drugbank:drugCategory dbcategory:antibiotics	**Drugbank**
9	?drug2 drugbank:drugCategory dbcategory:antiviralAgents	**Drugbank**
10	?drug3 drugbank:drugCategory dbcategory:antihypertensiveAgents	**Drugbank**
11	?I1 drugbank:interactionDrug2 ?drug1	**Drugbank**
12	?I1 drugbank:interactionDrug1 ?drug	**Drugbank**
13	?I2 drugbank:interactionDrug2 ?drug2	**Drugbank**
14	?I2 drugbank:interactionDrug1 ?drug	**Drugbank**
15	?I3 drugbank:interactionDrug2 ?drug3	**Drugbank**
16	?I3 drugbank:interactionDrug1 ?drug	**Drugbank**
17	?drug drugbank:keggCompoundId ?cpd	**Drugbank**
18	?enzyme kegg:xSubstrate ?cpd	**KEGG**
19	?enzyme rdf:type kegg:Enzyme	Drugbank, **KEGG**, Chebi, DBpedia
20	?reaction kegg:xEnzyme ?enzyme	**KEGG**
21	?reaction kegg:equation ?equation	**KEGG**
22	?drug5 rdf:type dbowl:Drug	Drugbank, KEGG, Chebi, **DBpedia**
23	?drug owl:sameAs ?drug5	**Drugbank**, KEGG, DBpedia

Heuristic 2 as well as the order of the triple patterns in the query. For instance, triple pattern $keggDrug bio2rdf:url $keggUrl, from Example 1 could produce an empty answer whenever the Chebi endpoint is considered first. Nevertheless, the whole answer will be retrieved if KEGG is selected.

4.2 Decomposing SPARQL query

Once endpoints are selected for each triple pattern, the query is decomposed into sub-queries. Each subquery corresponds to a star-shaped group that can be an exact star, or a star with satellites. An exact star is a set of triples that share exactly one variable. A satellite is a triple pattern that shares a variable with one of the triple patterns in an exact star.

Definition 1 (Exact Star on ?X). *Let S be a BGP in a SPARQL 1.0 query Q, i.e., S is a set of triple patterns delimited by $\{\}$. An exact star of triple patterns in S on a variable ?X, $ES(S,?X)$, is as follows: a) ES(S,?X) is a triple pattern in S of the form $\{s\ p\ ?X\}$ or $\{?X\ p\ o\}$ such that, $s \neq ?X$, $p \neq ?X$ and $o \neq ?X$. b) ES(S,?X) is the union of two exact starts, ES1(S,?X) and ES2(S,?X), that they only share ?X, i.e., $var(ES1(S,?X)) \cap var(ES2(S,?X)) = \{?X\}$.*

Definition 2 (Star on ?X with Satellites). *Let S be a BGP in a SPARQL 1.0 query Q. A star of triple patterns in S on ?X with satellites, ESS(S,?X),*

is as follows: a) ESS(S,?X) is the union of an exact star ES(S,?X) and a triple pattern t={s' p' ?Y} or t={?Y p' o'} in S where, ?X ≠ ?Y and ?Y ∈ var(ES(S,?X))∩var(t). b) ESS(S,?X) is the union of two stars with satellites, i.e., ESS1(S,?X) ∪ ESS2(S,?X) .

Table 3 presents stars with satellites for triple patterns from Examples 1 and 2. Note that triple pattern in line 9 from the Example 1 must be evaluated in both endpoints (KEGG and Chebi) individually. In the corresponding SPARQL 1.1 query, stars will be represented as service blocks.

Table 3. Exact Star with Satellites for Queries from Examples 1 and 2

Query Example 1			
Line	Exact Star with Satellite	Endpoint	Star Variable
6	$drug rdf:type drugbank:drugs	Drugbank	$drug
7	$drug drugbank:keggCompoundId $keggDrug		
8	$drug drugbank:genericName $drugBankName		
10	$chebiDrug purl:title $drugBankName	Chebi	$chebiDrug
11	$chebiDrug bio2rdf:image $chebiImage		
9	$keggDrug bio2rdf:url $keggUrl	KEGG, Chebi	$keggDrug
Query Example 2			
Line	Exact Star with Satellite	Endpoint	Star Variable
8	?drug1 drugbank:drugCategory dbcategory:antibiotics	Drugbank	?I1
11	?I1 drugbank:interactionDrug2 ?drug1		
12	?I1 drugbank:interactionDrug1 ?drug		
9	?drug2 drugbank:drugCategory dbcategory:antiviralAgents	Drugbank	?I2
13	?I2 drugbank:interactionDrug2 ?drug2		
14	?I2 drugbank:interactionDrug1 ?drug		
10	?drug3 drugbank:drugCategory dbcategory:antihypertensiveAgents	Drugbank	?I3
15	?I3 drugbank:interactionDrug2 ?drug3		
16	?I3 drugbank:interactionDrug1 ?drug		
17	?drug drugbank:keggCompoundId ?cpd	Drugbank	?drug
23	?drug owl:sameAs ?drug5		
18	?enzyme kegg:xSubstrate ?cpd	KEGG	?enzyme
19	?enzyme rdf:type kegg:Enzyme		
20	?reaction kegg:xEnzyme ?enzyme		
21	?reaction kegg:equation ?equation		
22	?drug5 rdf:type dbowl:Drug	DBPedia	?drug5

4.3 Planning Queries Against SPARQL Endpoints

The optimizer generates a bushy tree plan that reduces intermediate results. The input is a SPARQL 1.1 query where service blocks correspond to sub-queries in the endpoints, and heuristic-based optimization techniques are followed to generate the tree plan. Leaves of the tree plan correspond to service blocks while internal nodes represent physical operators that will be used to merge intermediate results. Endpoints are contacted to retrieve the number of triples produced by each sub-query, and the optimizer uses this information to decide when to connect two sub-queries with a physical operator. The optimizer relies on a greedy-based algorithm to traverse the space of bushy plans, and outputs a bushy tree plan where the number of Cartesian products and the height of the tree are minimized. To reduce the tree height, sub-queries with the smallest

number of service blocks are connected with JOINs; then, those that are related with OPTIONAL or UNION operators are considered. Cartesian products are avoided and they are only placed at the end, if no other operator can be used.

5 Experimental Study

Datasets and Query Benchmarks: we ran the 25 FedBench queries against the collections [10]: cross-domain, linked data and life science. Further, since FedBench queries are composed of a relatively small number of triple patterns limiting the generation of different plans, we studied ten additional queries[11] (henceforth called *Complex Queries*, C1-C10). Extended setup evaluates the effects of selectivity of BGPs and number of SPARQL operators; they are comprised of between 6 and 48 triple patterns and can be decomposed into up to 8 sub-queries. FedBench collections[12]: DBpedia, NY Times, Geonames, KEGG, ChEBI, Drugbank, Jamendo, LinkedMDB, and SW Dog Food, were stored in 9 Virtuoso[13] endpoints; timeout was set up to 240 secs. or 71,000 tuples.
Evaluation Metrics: *i*) *Time First Tuple (TFT)* or elapsed time between query submission and the first answer[14]; *ii*) *Time Whole Answer (TWA)* or elapsed time between query submission and the last tuple of the answer; *iii*) *Endpoints' Answer Size (EAS)* or total number of tuples retrieved from the endpoints selected to evaluate a query; *iv*) *Percentage of the Answer (PA)* or percentage of the completeness of the query answer; and *v*) *Number of Endpoints' Calls (NEC)* or total number of requests submitted to the available endpoints during the execution of a plan. NEC sums up endpoints calls for: *1*) deciding if a triple can be answered by an endpoint, *2*) computing the size of a sub-query, and *3*) evaluating a sub-query. TFT and TWA correspond to the absolute wall-clock system time as reported by the Python `time.time()` function. Ground truths were computed by running each query against an endpoint that maintains all datasets in one unified Virtuoso endpoint. Experiments were executed on a Linux Mint machine with an Intel Pentium Core 2 Duo E7500 2.93GHz 8GB RAM 1333MHz DDR3.
Implementations: the decomposer was built on top of ANAPSID[1] using Python 2.6.5. It was equipped with capabilities to generate exclusive groups (EG), and stars with satellites (SSGM) and (SSGS). Python proxies were used to contact endpoints, compute statistics, and send tuples in messages of different sizes. Message size and execution timeout were 16KB and 1,800 secs, respectively.

5.1 Effectiveness and Efficiency of the Query Decomposition Techniques

This experiment aims to study the effects of EG, SSGS and SSGM on query performance; in all plans, data is merged in a bushy tree fashion. Table 4 reports

[11] `http://www.ldc.usb.ve/~mvidal/FedBench/ComplexQueries`
[12] `http://iwb.fluidops.com:7879/resource/Datasets`, November 2011.
[13] `http://virtuoso.openlinksw.com/`, November 2011.
[14] TFT reflects time required to produce the first answer; it is impacted by number of messages transferred from the endpoints and the selectivity of the sub-queries.

on NEC, TWA, EAS, and PA. We can observe that plans comprised of sub-queries produced by SSGS or SSGM are able to completely answer the queries in less time than the ones comprised of EGs. This is because stars with satellites produced by either SSGS or SSGM, commonly correspond to very selective sub-queries that can be efficiently executed by the endpoints. Contrary, the EG technique just focuses on identifying groups of triple patterns that can be exclusively executed by an endpoint, independently if the sub-query is selective or not. Thus, if the sub-queries are non-selective, these plans may contact endpoints multiple times and receive endpoints' responses that will lead to large intermediate results. All these factors negatively impact on the performance of the query engine as reported in Table 4. For example, this happens in LS6, and performance values of EG are up to four orders of magnitude worse than performance values of the SSGS plan. Regarding to completeness of the answer, plans produced by SSGM and EG generate more answers, i.e., almost 100% of completeness is achieved in the majority of the queries. Particularly, EG was able to answer query LD6 while the other techniques fail. This may happen when general predicates, such as `rdf:type`, `owl:sameAs`, are used in a query, and the proposed heuristics fail selecting the relevant endpoint(s).

Table 4. Performance and Quality of Different Query Decomposition: Number of Endpoints' Calls (NEC); Time Whole Answer (TWA) in secs; Endpoints' Answer Size (EAS); Percentage of the Answer (PA). Exclusive Groups (EG), Star Shaped Group Single endpoint selection (SSGS), Star Shaped Group Multiple endpoint selection (SSGM). (*) EG finishes with an error the execution of CD7. Relevant results are highlighted in **bold**.

Query	NEC			TWA (secs)			EAS			PA		
	EG	SSGS	SSGM	EG	SSGS	SSGM	EG	SSGS	SSGM	EG	SSGS	SSGM
CD1	32	2	21	1.45	0.07	0.20	62	48	61	100	78.69	100
CD2	17	3	6	1.05	0.16	0.07	3	0	2	100	0	100
CD3	**139**	**15**	18	**58.34**	**0.50**	0.52	**92,000**	**15**	15	**100**	**100**	100
CD4	**212**	16	**45**	**736.52**	0.63	**42.69**	**2,913,606**	11	**233,587**	**100**	0	**100**
CD5	10	3	4	46.7161	2.96	2.98	270,498	13,465	13,465	100	100	100
CD6	**216**	6	**17**	**375.49**	0.81	**11.02**	**1,971,154**	4,388	**81,297**	**100**	0	**100**
CD7	**≥250**	3	7	*	4.77	4.86	**≥171,216**	21,491	21,491	0	0	0
LD1	8	2	2	77.94	0.08	0.08	254,364	309	309	100	100	100
LD2	1	1	1	0.19	0.04	0.04	185	185	185	100	100	100
LD3	**15**	**2**	3	**107.53**	**0.06**	0.05	**511,804**	109	109	**100**	**100**	100
LD4	1	3	3	0.22	0.30	0.27	50	1,119	1,119	100	100	100
LD5	**21**	**1**	3	**24.42**	**0.03**	0.04	**128,986**	**28**	28	**78.57**	**100**	100
LD6	36	13	15	62.55	11.87	16.63	281,007	54,566	74,397	100	0	0
LD7	6	3	5	16.96	0.23	0.25	73,994	1,216	1,216	4.61	100	100
LD8	29	9	13	78.92	3.10	22.11	392,831	12,222	135,457	100	100	100
LD9	12	1	6	1.43	0.02	0.10	0	0	0	100	100	100
LD10	12	6	10	45.96	0.16	0.19	264,662	6	6	100	100	100
LD11	2	2	3	3.17	0.13	0.14	376	376	376	100	100	100
LS1	1	1	1	0.38	0.21	0.167	1,159	1,159	1,159	100	100	100
LS2	104	22	89	4.35	0.25	3.06	337	328	337	100	97.26	100
LS3	24	5	15	55.80	19.29	18.38	271,916	75,346	85,432	100	100	100
LS4	20	4	6	10.06	8.83	8.42	38,776	34,532	34,532	100	100	100
LS5	**2,896**	10	**4**	**221.40**	11.97	**15.82**	**1,437,488**	69,436	**1,331**	**0**	0	**100**
LS6	**24,393**	**5**	22	**697.87**	**8.04**	19.35	**543,105**	**32,562**	108,304	**100**	**100**	100
LS7	8	4	3	21.05	16.47	24.96	120,890	68,191	2,240	100	100	100

5.2 Effectiveness and Efficiency of Stars with Satellites in Complex Queries

Performance of bushy plans comprised of SSGS and SSGM sub-queries is compared to FedX [11] performance. FedX queries were run on cold cache, i.e., FedX did not record any information about the endpoints, and also in warm cache, i.e., each query was run five times and the best time was reported. We evaluated 25 queries of FedBench and ten additional complex queries that we have defined for the FedBench collections. FedX, SSGS and SSGM perform similarly in the 25 query of the FedBench benchmark in both cold and warm caches. Thus, we just focus on the comparison of these engines in complex queries. Table 5 reports on Time for First Tuple (TFT), Time Whole Answer (TWA), and Percentage of the Answer (PA). First, we can observe that FedX could only complete the execution of C4 and C5, and timed out during the execution of the rest of the complex queries after 1,800 secs (NA) or ended due to an evaluation error. This is because exclusive groups for these complex queries were costly to be evaluated by the endpoints. These sub-queries were comprised of a large number of triple patterns; in cases where they could be evaluated by the endpoints, they produced large results which led to larger intermediate results during the left-linear fashion execution of the plan. Contrary, SSGS and SSGM produced several very simple stars with satellites, which generated a small number of results that remain small during bushy fashion executions. Thus, less calls of endpoints were required and a small number of messages were transferred sooner; this has a positive impact on the values of TFT and TWA of SSGM, and on TFT of SSGS. Finally, C9 and C10 could not be executed by any of the studied engines before 1,800 secs. C9 is composed of 2 BGPs connected by an OPTIONAL; the first BGP has 40 triple patterns while the second has 8 triple patterns. C10 contains 3 BGPs and 2 nested OPTIONAL operators. SSGM and SSGS strategies produced complex bushy tree plans of up to 8 sub-queries, 4 of these sub-queries contain more than 9 triple patterns; these plans were so complex to evaluate that the execution engine was unable to produce the first tuple before 1,800 secs. The observed results suggest that SSGM, SSGS and FedX decomposition and execution techniques are competitive for FedBench-like queries. The first two are more appropriate if the queries are complex and comprised of a large number of triple patterns, while the latter is very efficient for queries with a small number of triple patterns that can be exclusively executed by single endpoints. Nevertheless, further study is required to extend current approaches for scaling up to very complex queries.

6 Conclusions and Future Work

We have devised a two-fold solution for the problem of finding plans against a federation of endpoints. SPARQL 1.0 queries are transformed into SPARQL 1.1 composed of simple sub-queries that are executed in bushy tree fashion. Results transferred from selected endpoints as well as produced during the execution of a plan, may be reduced. Experimental results suggest that the proposed techniques may overcome existing engines. Nevertheless, these techniques may also

Table 5. Performance and Quality of SSGS and SSGM Bushy Tree with FedX- Additional Complex Queries; Time First Tuple (TFT); Time Whole Answer (TWA); Percentage of the Answer (PA). NA represents No Answer. (*) FedX finishes with an error the execution of C3, C6, C9 y C10. Relevant results are highlighted in **bold**.

Query	TFT(secs)			TWA (secs)			PA		
	SSGM	SSGS	FedX	SSGM	SSGS	FedX	SSGM	SSGS	FedX
C1	14.11	4.30	NA	20.94	9.11	1,808.25	100	100	0
C2	18.87	1.99	NA	19.05	2.11	1,800.33	100	100	0
C3	47.50	6.43	NA	1,614.13	11.43	*	100	100	0
C4	12.58	10.36	NA	19.33	17.08	921.59	100	100	0
C5	**3.78**	**3.76**	**22.92**	**11.21**	**11.09**	**22.92**	**100**	**100**	**0.38**
C6	27.41	30.30	NA	27.41	30.30	*	100	100	0
C7	13.31	13.26	NA	13.37	13.32	1,800.38	100	100	0
C8	10.22	9.21	NA	10.23	9.22	1,815.60	100	100	0
C9	NA	NA	NA	1,800.19	1,800.09	*	0	0	0
C10	NA	NA	NA	1,800.11	1,800.47	*	0	0	0

fail executing very complex queries. In the future we plan to enhance our physical operators with new SPARQL 1.1. features, e.g., binding clauses, and merge in a more efficient fashion relevant data retrieved from the endpoints.

References

1. M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. Anapsid: an adaptive query processing engine for sparql endpoints. In *Proceedings of the 10th international conference on The semantic web - Volume Part I*, ISWC'11, pages 18–34, Berlin, Heidelberg, 2011. Springer-Verlag.
2. C. Basca and A. Bernstein. Avalanche: Putting the Spirit of the Web back into Semantic Web Querying. In *The 6th International Workshop on SSWS at ISWC*, 2010.
3. C. Buil-Aranda, M. Arenas, and O. Corcho. Semantics and optimization of the sparql 1.1 federation extension. In *ESWC (2)*, pages 1–15, 2011.
4. O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Proceedings of the 2nd International Workshop on Consuming Linked Data*, Bonn, Germany, 2011.
5. A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *WWW*, pages 411–420, 2010.
6. O. Hartig. Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *ESWC*, pages 154–169, 2011.
7. Z. Kaoudi, K. Kyzirakos, and M. Koubarakis. Sparql query optimization on top of dhts. In *ISWC*, pages 418–435, 2010.
8. G. Ladwig and T. Tran. Linked data query processing strategies. In *ISWC*, pages 453–469, 2010.
9. Y. Li and J. Heflin. Using reformulation trees to optimize queries over distributed heterogeneous sources. In *ISWC*, pages 502–517, 2010.
10. M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench: A benchmark suite for federated semantic data query processing. In *International Semantic Web Conference (1)*, pages 585–600, 2011.
11. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *International Semantic Web Conference*, pages 601–616, 2011.