SemaGrow: Optimizing Federated SPARQL queries

Angelos Charalambidis
Institute of Informatics and Telecommunication
NCSR 'Demokritos'
Ag. Paraskevi, Greece
acharal@iit.demokritos.gr

Antonis Troumpoukis
Institute of Informatics and Telecommunication
NCSR 'Demokritos'
Ag. Paraskevi, Greece
antru@iit.demokritos.gr

Stasinos Konstantopoulos
Institute of Informatics and Telecommunication
NCSR 'Demokritos'
Ag. Paraskevi, Greece
konstant@iit.demokritos.gr

ABSTRACT

Processing SPARQL queries involves the construction of an efficient *query plan* to guide query execution. Alternative plans can vary in the resources and the amount of time that they need by orders of magnitude, making planning crucial for efficiency. On the other hand, the construction of optimal plans can become computationally intensive and it also operates upon detailed, difficult to obtain, metadata. In this paper we present *Semagrow*, a federated SPARQL querying system that uses metadata about the federated data sources in order to optimize query execution. We balance 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 as many situations as possible. Semagrow also exploits non-blocking and asynchronous stream processing technologies to achieve query execution efficiency and robustness. We also present and analyse empirical results using the FedBench benchmark to compare *Semagrow* against FedX and SPLENDID. Semagrow clearly outperforms SPLENDID and it is either on a par or much faster than FedX.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing; H.2.4 [**Systems**]: Distributed databases; I.2.8 [**Problem Solving, Control Methods, and Search**]: Plan execution, formation, and generation

General Terms

Algorithms

Keywords

Federated query optimization, SPARQL query processing

1. INTRODUCTION

Efficient query processing depends on the construction of an efficient *query plan* to guide query execution. More specifically, given a query in a declarative language such as SPARQL, there are many plans that a query processing system can consider to retrieve the answer. All those plans are *equivalent* in the sense that they return the same answer, but they can vary by orders of magnitude in their *cost* in computational resources and time. The implication is that detailed *instance-level metadata* about the data sources can be used to estimate cost and select among alternative plans. Although this is true of any query processing system offering a declarative query language, it is in *federated querying* where we feel that the state of the art has not reached the level of maturity where declarative queries can be processed transparently and efficiently.

Distributed triple stores and federated querying are key technologies for the efficient and scalable deployment of semantic technologies. Distributed triple stores exercise full control over a cluster of storage nodes to efficiently scale out without exposing to client applications the internal organization of data among the nodes of the cluster. Federated querying, on the other hand, accommodates the more dynamic combination of query endpoints that are maintained and populated independently of their membership in one (or multiple) federations.

This dynamic, loose integration of endpoints is more characteristic of the kind of scalability envisaged for the Semantic Web and the Linked Data cloud than the tighter integration of centralized distributed stores. But it leaves open questions in query optimization, since it introduces factors that are not relevant to single-node or tightly integrated distributed databases and triple stores: that endpoint metadata might not be available or not be reliable; endpoints might get updated, might be temporarily unavailable; or might have varying cost parameters depending on load other than the load generated by the federation service.

Two systems that stand out among the federated querying literature for their low query processing time, are also characteristic of different approaches to efficient federated querying. The FedX system [15] plans query execution without relying on any metadata about the endpoints it federates. FedX issues ASK queries to identify candidate data sources among the federated endpoints and then applies a coarse heuristics to optimize the query plan. Although this algorithm can potentially produce suboptimal plans, it introduces minimal overhead to query execution. Together with a very efficient execution engine, this strategy makes FedX extremely fast executing low-cost queries and, more generally, in situations where more sophisticated optimization has little impact. SPLENDID [5], on the other hand, em-

SEMANTiCS '15, September 15-17, 2015, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2814864.2814886

```
SELECT ?drug ?title {
  ?drug db:drugCategory dbc:micronutrient. (#P1)
  ?drug db:casRegistryNumber ?id.          (#P2)
  ?keggDrug rdf:type kegg:Drug.            (#P3)
  ?keggDrug bio2rdf:xRef ?id.              (#P4)
  ?keggDrug purl:title ?title.             (#P5)
}
```

Figure 1: Query LS6 of the FedBench benchmark.

ploys data source metadata to estimate the cost of executing query plans in order to select the most advantageous one to execute. This introduces query planning overhead and the requirement that such metadata is available. To counterbalance this requirement, SPLENDID relies on endpoint descriptions using the VoID vocabulary [1], an independently motivated and widely used vocabulary for describing linked datasets.

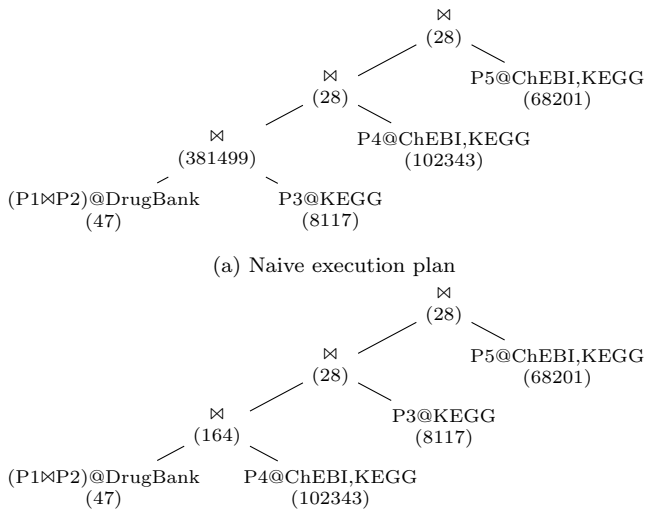
Although SPLENDID is generally slower than FedX, its plans can sometimes require an order of magnitude less data to be fetched from the remote endpoints in order to prepare the same final results. Besides the execution efficiency improvement, this is particularly important when federating publicly available endpoints that often apply a limit in the number of rows in a result set. The observation that optimized planning often does not pay off but can sometimes have a huge impact has prompted us to investigate how to optimize query plans while at the same time performance remains competitive despite the optimization overhead in situations where optimization does not yield considerable performance improvements.

In *Semagrow*, the federated querying system presented here, we balance 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 most (although not theoretically all) possible situations (Section 2). We have also developed an efficient non-blocking query execution engine that takes advantage of state-of-the-art stream processing technologies to achieve efficiency and robustness to the size of the result set (Section 3). We present and analyse empirical results over the FedBench benchmark, demonstrating that our system outperforms both SPLENDID and FedX (Section 4) and close the paper with conclusions and future research plans (Section 5).

2. QUERY OPTIMIZATION

In order to minimize query processing time, federated querying systems first perform *source selection* to narrow the scope of query execution and then formulate a *querying plan*. The first step ensures that only relevant data sources are considered and the second provides to the execution engine a detailed specification of the sub-query to be executed at each (relevant) endpoint and the ordering in which such sub-queries will be executed and joined.

In order to illustrate the steps performed during the query optimization, we will consider Query LS6 (Figure 1) from the FedBench benchmark. This query is taken from the life sciences domain and combines information from the *DrugBank* database of drugs and from the *Kyoto Encyclopedia of Genes and Genomes (KEGG)* in order to retrieve the KEGG title of drugs in the Micronutrient category. Besides these two datasets, the *Chemical Entities of Biological Interest (ChEBI)* dataset is also a candidate as it contains triples matching P4 and P5 but, unlike KEGG, these triples do not join with the DrugBank triples matching P1 and P2.



(a) Naive execution plan

(b) Optimized execution plan

Figure 2: Alternative execution plans for Query LS6. The numbers in parentheses give the cardinality of the result of each operation.

First, the query string is parsed into a tree where the leaves correspond to triple patterns (P1 through P5 in Figure 1) and the intermediate nodes represent relational operators such as join, union, etc. Query optimization restructures and augments query parse trees into execution plans where (a) besides the retrieval of triples matching a pattern, leafs can also represent complex sub-queries to be executed by an endpoint; and (b) the order or the execution is changed. Finally, execution trees are annotated with the endpoint (or endpoints) where each sub-query should be executed.

In our example, a more naive plan would execute the query in the order specified in the query string (Figure 2a) whereas an optimized plan would avoid the premature join with P3 (Figure 2b). Note in this example the dramatic difference in the cardinality of the intermediate results (and the associated processing time) effected by a re-ordering of the processing that does not alter the final result.

Execution plan construction is generally approached as a search through the space of possible plans guided by a cost function that estimates the efficiency of each solution. We will now proceed to present the Semagrow instance of this overall methodology and to discuss its relation to the methods used by FedX and SPLENDID.

2.1 Assigning Queries to Data Sources

FedX and SPLENDID use similar techniques to restrict the sources that must be considered during query optimization. For each individual pattern in the query, FedX issues ASK queries to all the sources in the federation to identify data sources that contain matching triples. SPLENDID uses VoID descriptions to exclude sources that cannot possibly contain matching triples, and then proceeds to issue ASK queries over the remaining candidates. In the absence of available metadata, SPLENDID falls back to issuing ASK queries to all the sources in the federation. Semagrow uses

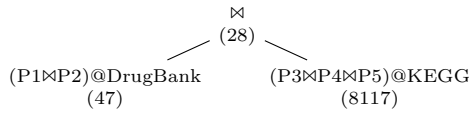


Figure 3: Optimal execution plan for Query LS6 in the absence of the ChEBI dataset.

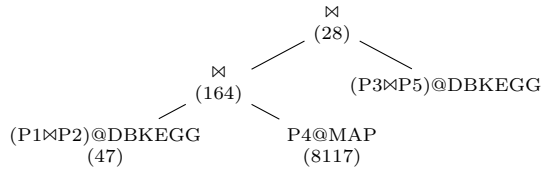


Figure 4: Optimal execution plan for Query LS6 under a different distribution of triples to datasets.

the same hybrid, pattern-wise source selection method as SPLENDID does.

Relevant to identifying these sources is also the decision regarding how to decompose the query into the sub-queries that will be executed at each source. An effective heuristics is to group together all patterns to be executed at the same source in order to maximize the computation that is (more efficiently) performed closer to the data. Consider, for example, how P1⋈P2 is executed at DrugBank rather than fetching P1 and P2 results and joining them at the federation node (Figure 2). If it were possible to exclude ChEBI from the data source candidates, then P3⋈P4⋈P5 would also be grouped together into the more efficient execution plan shown in Figure 3.

By implementing this heuristic the scope of optimization is restricted to combining the resulting sub-queries. This greatly reduces the optimizer's search space and has a large impact to the time it takes for the optimizer to execute. This heuristics is, however, not guaranteed to give optimal results.

Consider, for example, a situation where all the data in DrugBank and KEGG where available from a single dataset *DBKEGG* except for the triples that match our pattern P4 that only exist in a (hypothetical) dataset *MAP*. Then the heuristics above would query (P1⋈P2⋈P3⋈P5)@DBKEGG and join that with P4@MAP. This plan would be very inefficient: without the triples from P4, the variables in the remaining patterns do not form a chain and the join has a multiplicative effect on the cardinality of the results. The alternative plan that starts with P4@MAP and then executes (P1⋈P2⋈P3⋈P5)@DBKEGG with prior bindings for ?keggDrug and ?id is not optimal either, because P4 also has a very large cardinality when neither of its variables has a prior binding.

The optimal plan (Figure 4) can only be formed if grouping together patterns at the same source is a soft preference rather than a hard constraint. On the other hand, the hard constraint reduces the optimization overhead. In Semagrow we have left this as a configuration option. It is possible to allow the search for an optimal plan to consider any decomposition into sub-queries, but the default behaviour is to force all patterns executing at the same source to be grouped together. This default was chosen after considering both the FedBench suite and the various use cases that drive the development of the system [10] and observing that there was no situation where the optimal plan is not the one that places all patterns at the same source under the same leaf.

Either way, the end-result of source selection is that patterns are annotated with relevant data sources. In case of multiple data sources a UNION node is added to combine the results from each data source. If data sources that are known to mirror another source then alternative plans are created, rather than a single plan with union nodes. The resulting query parse tree has a structure like the one shown in Figure 5 for our example query.

In this structure we have introduced the following elements to the syntax used by Sesame to represent the parsed query tree:[1]

- The SourceQuery operator that issues a sub-query to a remote endpoint.
- The SITE annotation that gives the endpoint where the operator is executed, or 'LOCAL' for internal computations.
- The CARD annotation that gives the estimated cardinality of the results of an operator.
- The COST annotation that gives the estimated cost of an operator.

At the source selection phase discussed here, the tree is annotated with the possible sites for each query pattern and the cardinality of the matching triples, as provided in the data source metadata. The cost for SourceQuery operators is estimated by applying a communication overhead factor to the cardinality of the results. In the costs shown in Figure 5 this is a uniform 10% for all sources. It is possible to apply different overhead to different data sources, assuming that such information is available in the metadata.

We will now proceed to present how Semagrow estimates the cardinality and cost of the non-leaf operators in the tree.

2.2 Cost Estimation

The cost of complex expressions is estimated recursively using a *cost model* over statistics about its sub-expressions. Different cost models are defined for each operator, so that, for example, the SourceQuery operator applies a communications overhead to the cardinality of the results but the Union operator sums the costs of its sub-expressions. Due to space considerations, please find the complete definition in the relevant SemaGrow project deliverable [4, Section 3.2].

It should also be noted in Figure 5 that the local join operators have been replaced by specific implementations of the generic join operator (in this example the BIND JOIN $\bowtie_b$), each with their own cost model. Semagrow considers multiple join algorithms (bind join, merge join) and creates alternative plans to be evaluated.

Some join implementations request certain properties for the underlying plans. For example, MERGE JOIN requests for its input to be sorted on the join variables, so the appropriate ORDERING operator is added to the tree *if not already there to satisfy the original query.* In this manner the optimizer can evaluate if it is worth sorting (or if it can

[1]Semagrow is developed within the Sesame framework, so we assume the output of the Sesame SPARQL parser as our base syntax. Please cf. `http://rdf4j.org` for more details.

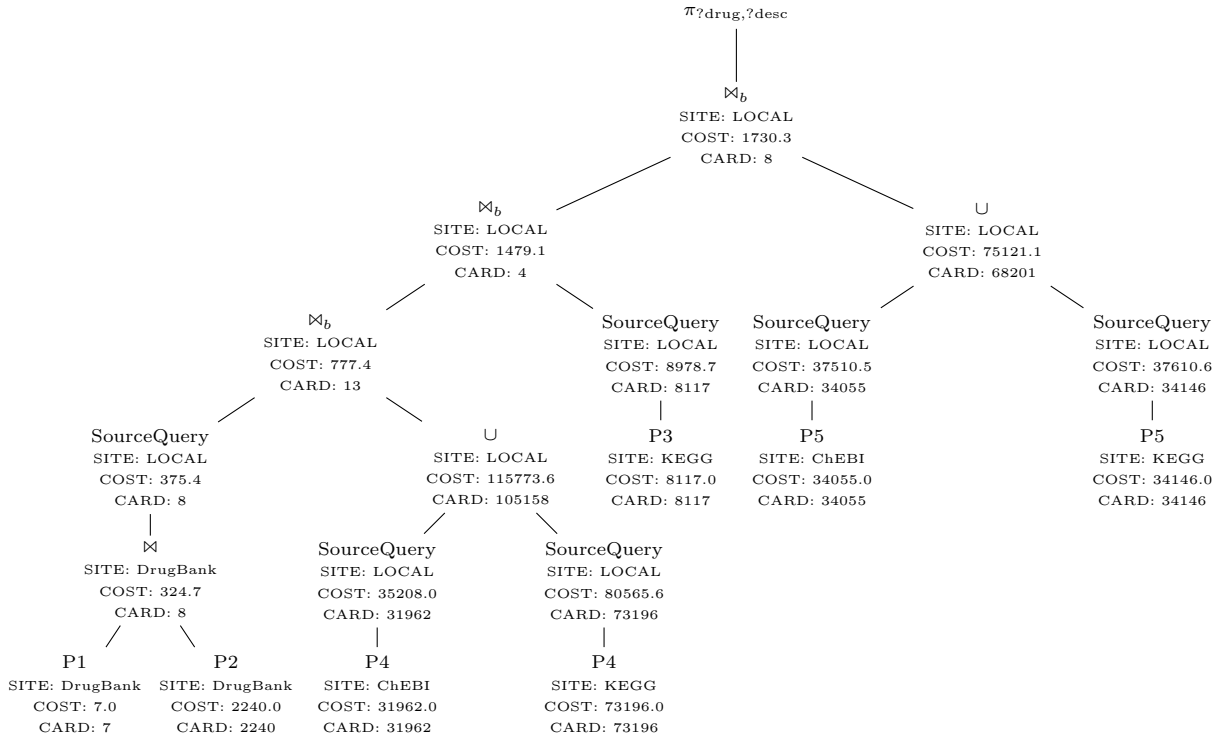


Figure 5: Execution plan with annotations

exploit a sorting that was required anyway) in order to apply the MERGE JOIN implementation that has a lower cost than the BIND JOIN implementation.

These statistics are either provided by source metadata as seen above, or (on internal nodes) estimated from the cardinality of the sub-expressions' results, as well as the number of *distinct* subjects, predicates, and objects appearing in these results. The number of distinct entities is used to estimate the *selectivity* of JOIN and FILTER expressions: the ratio of the number of tuples that satisfy the expression over the number of tuples in the complete relation.

For example, assume that in DrugBank there are:

- 4602 triples with 1879 distinct subjects and 584 distinct objects matching ?X db:drugCategory ?Y
- 2240 triples with 2240 distinct subjects and 2218 distinct objects matching ?X db:casRegistryNumber ?Y

For the sake of the argument, let us assume independence and homogeneous distribution of values. We estimate that pattern ?X db:drugCategory db:micronutrient has 8 matching triples, by dividing the total triples with the distinct number of objects. Moreover, the join selectivity of P1⋈P2 on the variable ?X can be estimated as $\min(ds_1^{-1}, ds_2^{-1})$ where ds_i is the distinct values of ?X in each of the triple patterns. In our example, the selectivity factor is estimated to be $\min(1879^{-1}, 2240^{-1})$ equals 2240^{-1}. The estimated cardinality of P1⋈P2 is then the product of their estimated cardinalities and the selectivity factor, thus $8 * 2240/2240$ equals 8. However, the actual cardinality of P1⋈P2 is 47. The difference between the actual cardinality and its estimate comes from the assumption that triples are distributed evenly over the values of different drug categories.

The above shows the importance of not only accurate, but also *detailed* source metadata for the construction of a good query execution plan. VoID is a popular vocabulary for describing datasets. In general, and omitting the details, a VoID metadata description partitions datasets in subsets based on (a) the property of the triples in each subset and (b) on regular expressions specifying the URIs of the subjects and objects. For each such subset, the cardinality and the number of distinct instances can be specified. This allows for arbitrarily detailed hierarchical descriptions that can capture heterogeneity in the density of triples involving given instances.

Coming back to our concrete example, the Semagrow estimator would have accurately predicted the cardinality of P1⋈P2 in the presence of statistics that give the density of the subset defined by ?X db:drugCategory db:micronutrient as different from its ?X db:drugCategory ?Y superset.

Finally, the purposes of resource discovery, VoID is adequate for finding sources for grounding star patterns in the query, where the properties of a known 'central' individual need to be retrieved. It is less well-suited for finding sources for path patterns, since VoID makes no assertions regarding filler instances.

2.3 Plan Generation

With cost estimations defined, we can proceed to evaluate different plans in order to identify the one that is optimal with respect to our cost model. To do this we use *dynamic programming*, the standard enumeration algorithm for join ordering optimization that has been used successfully in many database systems.

We identify subtrees of the query that consists of triple patterns connected with inner joins and optimize them sep-

Algorithm 1 Query planning using Dynamic programming

```
procedure GetBestPlan({r_1, ..., r_n})
    for all relations r_i do
        plans({r_i}) ← AccessPlans({r_i})
        PrunePlans(plans({r_i}))
    for all subsets S_1, S_2 such that S = S_1 ∪ S_2 do
        plans(S) ← JoinPlans(plans(S_1), plans(S_2))
        PrunePlans(plans(S))
    FinalizePlans(plans({r_1, ... r_n}))
    return MinimumPlan(plans({r_1, ... r_n}))
```

arately (i.e. find an equivalent join tree with the minimum cost). The algorithm (Algorithm 1) proceeds bottom-up constructing execution plans for the leaves of the tree (i.e. triple patterns) first. *Access plans* enumerate all alternative ways to access that triple pattern consulting the source selection. Then, the algorithm enumerates all the two-way joins using the access plans as building blocks. The algorithm will enumerate all the alternative join plans considering all the join implementations available. At the end of the j-th step the algorithm produces intermediate plans for expressions consisting up to j relations. The problem reduces to the enumeration and cost assignment of all the possible n-way joins and finally selecting the one with the minimum cost, where n is the number of triple patterns. However, the enumeration of all possible plans is exponential, the algorithm prunes in each step the inferior plans in order to keep the enumeration space as small as possible.

3. QUERY EXECUTION

3.1 Reactive Query Execution

The most popular paradigm for evaluating tree-based execution plans like the ones produced by SemaGrow is to form a dataflow tree where results flow from the leaves to the root. Most of the state-of-the-art database system use the popular iterator [6] model in order to realize this dataflow. In that model of execution, operators (ie. intermediate nodes of the query plan) are encapsulated under a common iterator interface that provide basic operations over a stream of query results; i.e. open(), close() and next() operations. Typically, higher-level nodes request the evaluation of lower-level elements by using the next() operation. This *demand-driven* paradigm suites perfectly for sequential systems, since data are generated only if and when needed and almost no extra effort is added. However, in a parallel system, where data can be available in different rates, the iterator model is not suitable. Requests that are propagated down the stream can be blocked by other slower operations and potentially stalling the overall query evaluation.

For example, consider a iterator implementation of the union operator that merges a pair of results into a single stream of results without a predefined order. In such an implementation the request for the next element of the union will propagate to one of the underlying streams. However, in a parallel query execution since both streams can work independently and produce results in a non deterministic way, there might occur that the union operator will block waiting for the slow stream to yield a result, and in the same time data from the other stream can be already available.

SPLENDID uses the iterator model forcing the evaluation to be performed sequentially and as a result exhibits poor performance (see Section 4). FedX still uses the iterator model, but in order to circumvent that problem, has developed a multithreading execution engine, in which every operator executes in a separate thread. This obviously increases the parallelism of the execution engine but on the other hand threads must be synchronized resulting in additional costs [9]. Another approach to circumvent the blocking nature of iterators is the non-blocking iterators [8] which provide a non-blocking call to next() where the next() can either return an element or nothing if the next element is not yet available.

SemaGrow uses the *reactive* paradigm, that also operates in an asynchronous and non-blocking way. The basic idea of a reactive implementation is that operators subscribe to a stream and are notified when data become available. Meanwhile the operator can produce useful work without being blocked waiting data to arrive. In other words, the dataflow can be perceived as a chain of callback handlers that are activated as soon as data are available in the input working in a *data-driven* fashion. However, since data can be produced in a faster pace than can be consumed, the reactive paradigm employs a backpressure mechanism, ie. a consumer can send a request down stream to inform for slower processing.

The reactive model of computation seems ideal for use cases such as federation over wide area networks where data are produced in different rates and unexpected delays can occur and in the same time, the federated sources can work truly in parallel.

3.2 Join implementations

Bind join [7] is especially attractive for distributed and federated environments because it is designed to drastically reduce the communication costs of joining two relations. The idea, similar to the nested loop join, is that if we have a very selective outer relation of a join, we can pass the results as bindings to the inner relation in order to filter out a large number of tuples. The difference is that bind join can also work for remote queries since it substitutes the results of the outer relation as bindings to he query of the inner relation.

A naive implementation of a bind join can be highly inefficient since it will create and execute a different query for each result of the outer relation. A more elaborate solution for reducing the overall number of queries produced is to group multiple bindings into a single query. Fortunately, SPARQL 1.1 specification foresees a VALUES keyword as a mechanism for passing multiple variable bindings at once. However, in order to support legacy SPARQL 1.0 endpoints we have also implemented the grouping proposed by Schwarte et al. [15] using a more complex UNION expression. An example of the corresponding transformations for the two approaches can be seen in Figure 6. In this example we illustrate the source query that the query executor sends

```
SELECT * WHERE { ?keggDrug bio:xRef ?id . }
VALUES (?id) { (cas:70-26-8) (cas:56-85-9) (cas:61-19-8) }

SELECT * WHERE {
  { ?keggDrug_1 bio:xRef cas:70-26-8 } UNION
  { ?keggDrug_2 bio:xRef cas:56-85-9 } UNION
  { ?keggDrug_3 bio:xRef cas:61-19-8 } }
```

Figure 6: Alternative queries to the data sources for the BIND JOIN implementation.

Table 1: Bind join implementations comparison

Type of query	Time in ms
UNION construct	12
VALUES clause	218

Table 2: Query time (msec) for different batch sizes

Size	5	7	8	9	10	12	15
Time	6351	5955	5934	6025	6231	6427	7465

to the ChEBI dataset for the bind join processing of the P4 pattern using 3 bindings from the outer result (P1⋈P2 at drugBank). Note that the second query requires some additional post-processing in order to rename the binding variables back to their original names.

We tested both approaches by measuring the average execution time for 5 runs of each of the two queries of Figure 6 in the ChEBI SPARQL endpoint. For each query we attached 8 bindings for the variable ?id (instead of 3 that is illustrated in Figure 6). The results are illustrated in Table 1, where we can clearly see that the query with the UNION expression is faster than the corresponding query with the VALUES expression. Therefore, by selecting the first approach, we support SPARQL 1.0 endpoints and also avoid the slower execution which we have observed, possibly resulting from the inefficient implementation of the VALUES operator.[2]

Another crucial factor for the performance of our query executor is the size of the batching, i.e., the maximum number of bindings that are passed in each of the UNION construct. A small batch will result to a larger number of remote queries, while a large batch will result to fewer but larger and slower queries. In order to identify the optimal batch size, we measured the execution time for a set of federated queries[3] using different batch sizes. The results of this experiment are shown in Table 2, and suggest the use of a batch size equal to 8. Although we used FedBench queries for this tuning, there is nothing specific to the FedBench suite in the batch size decision since it only relates to the trade-off between opening a connection and processing a large UNION of different bindings for the same query pattern. Consequently, although the batch size can be set as a configuration option, we recommend that the default value is used without need for tuning to specific use cases.[4]

4. EXPERIMENTS AND EVALUATION

In this section we evaluate SemaGrow and we compare it with FedX 3.0 and SPLENDID,[5] which have been shown to be the most performant systems in the state of the art [11]. For our experiments we used the FedBench suite. FedBench has been used extensively to evaluate federated SPARQL query processing systems, so using it allowed us to directly compare SemaGrow with the current state of the art.

[2]In our experiments we have used Virtuoso v. 06.01.3127 to deploy the federated sources.

[3]Specifically, the LS queries from the FedBench suite.

[4]Naturally, this depends on the federated endpoints and might change with future versions of the major triple stores. We will continue updating the SemaGrow default whenever it becomes necessary. We expect that support for SPARQL 1.1 will soon be ubiquitous and efficient enough to allow us to replace the UNION implementation of the bind join with the VALUES implementation.

[5]The most recent version available at `https://github.com/goerlitz/rdffederator`

Table 3: Dataset Statistics

Dataset	#triples	#subj.	#pred.	#obj.
DBpedia subset	43.60m	9500k	1063	13600k
GeoNames	108.00m	7480k	26	35800k
LinkedMDB	6.15m	694k	222	2050k
Jamendo	1.05m	336k	26	441k
NY Times	0.34m	22k	36	192k
SW Dog Food	0.10m	12k	118	38k
KEGG	1.09m	34k	21	939k
ChEBI	7.33m	51k	28	772k
Drugbank	0.77m	20k	119	276k

Table 4: Query characteristics

Query	Type	# Triple Patterns	# Results
CD1	Complex	3	90
CD2	Star	3	1
CD3	Chain-Star	5	2
CD4	Complex	5	1
CD5	Chain-Star	4	2
CD6	Chain-Star	4	11
CD7	Chain-Star	4	1
LS1	Complex	2	1159
LS2	Chain	3	333
LS3	Chain-Star	5	9054
LS4	Complex	7	3
LS5	Complex	6	393
LS6	Complex	5	28
LS7	Complex	5	144

4.1 Experimental setup

FedBench [14] is commonly used to evaluate the performance of the SPARQL query federation systems. The benchmark is explicitly designed to represent SPARQL query federation on real-world datasets. FedBench contains two suites of datasets and queries over them: a more varied *cross-domain (CD)* suite and a larger-scale suite from *life sciences (LS)*. The benchmark queries resemble typical requests on these datasets and their structure ranges from simple star and chain queries to more complex graph patterns (see Table 4 for the query characteristics).

In the evaluation setting[6] of this experimental setup we assume the SPARQL endpoints are reliable and fast. Thus, the endpoints are deployed in a single machine (with 2x3.1 GHz CPU and 4 GB RAM) and in different Virtuoso servers (Virtuoso version 06.01.3127). All the federation engines can access the data sources via the SPARQL protocol. All experiments were performed on a Linux Desktop PC (Ubuntu 14.04 LTS) with Intel(R) Core(TM) i7-4790 CPU, 8 GB RAM and a Gigabit Ethernet connection.

The statistics of each dataset used in the evaluation is presented in Table 3. SPLENDID and SemaGrow use VoID metadata that are generated by extracting statistics directly from the actual data. For every experiment, all queries are executed 5 times following a single warm-up run.

4.2 Execution

Table 5 shows the total query processing time (average)

[6]The complete configuration and code of our experimental setup can be retrieved at `https://bitbucket.org/dataengineering/semagrow-fedbench`

Table 5: Query planning, execution only and total processing times (msec)

Query	Planning			Execution			Total		
	FedX	SPLENDID	SemaGrow	FedX	SPLENDID	SemaGrow	FedX	SPLENDID	SemaGrow
CD1	8	48	1	15	28	17	23	76	*18*
CD2	5	26	2	9	8	10	14	34	*12*
CD3	4	36	11	28	45	27	*32*	81	38
CD4	6	31	13	29	12	37	*35*	43	50
CD5	7	30	5	24	28	82	*31*	58	87
CD6	6	40	5	394	8079	380	400	8119	*385*
CD7	4	46	6	480	1598	398	484	1644	*404*
LS1	2	19	1	34	35	62	*36*	54	63
LS2	4	20	1	27	169	35	*31*	189	36
LS3	4	27	8	3817	24441	2927	3821	24468	*2935*
LS4	5	39	65	27	31	19	*32*	70	84
LS5	4	52	26	2283	295095	1035	2287	295147	*1061*
LS6	6	28	9	95977	19642	98	95983	19670	*107*
LS7	5	39	4	1678	18595	1182	1683	18634	*1186*

for both query collections, while Table 5 shows only the query execution times.

SemaGrow outperforms SPLENDID in most queries, including when executing identical plans (CD6, LS5, LS6 and LS7, Table 5). This is mainly due to the fact that SPLENDID does not group the bindings while evaluating a bind join operator.

FedX and SemaGrow exhibit similar behaviour in smaller queries, but Semagrow performs better in longer-running quries (LS3, LS5, LS6 and LS7). In particular, the query execution time is similar for both engines in the situations where both optimizers produce the same plan, which happens for all queries except LS5 and LS6. Therefore, we conclude that our execution engine performs on a par with that of FedX. In LS5 and LS6 Semagrow is substantially faster because it executes a more efficient plan.

4.3 Optimization

Table 5 shows the total query planning times for both query collections. FedX planning times are smaller than those of SPLENDID, due to the dynamic programming optimization of SPLENDID, which is slower than the greedy optimizer of FedX. Even though SPLENDID and SemaGrow use a similar approach in planning, we notice that the planning of SemaGrow is faster than that of SPLENDID. This is due to the caching we have deployed in our decomposer for performance issues.

The use of ASK queries in resource selection and the dynamic programming decomposition of the SemaGrow optimizer results in the construction of high quality plans for all queries for the FedBench experiment. However, the optimization process does not incorporate any use of heuristics, as it is in the case of SPLENDID. These heuristics helps SPLENDID to produce an efficient plan, but are not safe in terms of completeness of the query. For example, consider the triple patterns of the form of ?x owl:sameAs ?y, that occur in every source. FedX and SemaGrow ask every source, while SPLENDID performs sameAs groupings. This heuristic works for most of FedBench queries, but in CD7 misses one answer.

An interesting case is query LS6 (Figure 1), in which SemaGrow outperforms both SPLENDID and FedX by an order of magnitude. SPLENDID and SemaGrow yield the same execution plan, so the difference in the performance relies on the more efficient execution engine of SemaGrow. SemaGrow outperforms FedX because our decomposer yields an execution plan of better quality. The reason for that is twofold: first, SemaGrow uses the dataset statistics and second the query decomposer guarantees the optimality of the plan with respect to the SemaGrow's cost function. On the other hand, FedX selects a suboptimal plan due to its greedy query decomposer, resulting in a vast difference of cost. The comparison of the plans is illustrated in Figure 2 (FedX uses the first plan while SemaGrow the second plan). Notice that the selectivity of the join between the source queries (P1⋈P2) at drugBank dataset and P3 at KEGG dataset is equal to 100%. This decision of the FedX optimizer leads not only to slower execution time (due to the high cardinality of the operation) but can also lead to loss of results, since many public endpoints apply a limit in the number of rows in a result set (each of the source queries in the KEGG dataset will return more than 120000 results due to the UNION transformation for bind join). This situation is a concrete example that the optimality of the execution plan has a crucial role in the overall performance of the systems.

Overall, results demonstrate that Semagrow and FedX are clearly more efficient than SPLENDID. By comparison to FedX, Semagrow's more sophisticated planner needs more time, but this time is not dramatically longer while the produced plans can be dramatically better. Consider how FedX outperforms Semagrow in fewer queries (6 out of the 14 FedBench queries) and only in smaller queries (all under 100msec) and by a smaller margin as at its best (CD5) FedX needs 36% of the time needed by Semagrow. On the other hand, when Semagrow outperforms FedX in the remaining 8 queries, including all the larger one (over 1sec) and the margin can get dramatic as Semagrow completes LS6 in 0.1% of the time needed by FedX.

5. CONCLUSIONS AND FUTURE WORK

In this paper we present the query processing methods developed for the *Semagrow* federated querying system. We have assumed as the starting point the comparison of the behaviour of the two federated querying systems that have been shown to be the most performant ones in benchmarks: the sophisticated execution planning of SPLENDID can have a huge impact on execution time, but very often does not manage to recover the overhead introduced by the optimizer;

FedX, on the other hand, relies on the speed of query execution to perform well and can fall into pitfalls where its naive execution plan can lead to execution times that are orders of magnitude slower than what they could have been. Another important distinction is that SPLENDID depends on the existence of VoID metadata about the data sources it federates. SPLENDID provides the tools necessary to automatically extract the metadata it needs, but applying such tools requires access to data dumps.

Semagrow, by comparison, features 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 most (although not theoretically all) possible situations. Furthermore, we have developed a new execution engine that is shown to be as efficient as that used by FedX but with the added benefit of being based on the *reactive* paradigm to operate in an asynchronous and non-blocking way. The benefits of the Semagrow system are already visible in the relatively small scale FedBench suite, where Semagrow clearly outperforms SPLENDID and it is either on a par (in smaller queries) or much faster than FedX (especially in longer-running queries). We expect an even larger margin in performance when executing over remote sources with substantial differences in network throughput and latency.

This leads into one of our future research plans being to develop a federated querying benchmark that more realistically simulates querying over the internet. We plan to base this benchmark on existing data sources and query sets, such as FedBench, but to develop the software infrastructure for simulating latency, limited throughput, and occasional timeouts or unavailable sources. We will calibrate this simulator using the statistics compiled by Buil-Aranda et al. [2] on endpoints registered in Datahub.

Following up on the point made about the importance of detailed metadata (Section 2.2), we are planning on investigating the effect of using more detailed metadata on query planning time as well on the plans produced. Planning time can be reduced by integrating more sophisticated source selection algorithms that prune data sources by leveraging metadata regarding how query patterns join across datasets, and not only the presence in datasets of matching triples [12, 13]. Regarding the actual plan produced, further experiments are needed in order to analyse the conditions under which more detailed metadata result in different plans and substantial query time efficiency gains.

In the experiments presented here we have used the exact same metadata as those used by SPLENDID. The Semagrow cost model, however, is aware of more detailed VoID descriptions as well as of descriptions in Sevod [3], an extension of the VoID vocabulary that allows or detailed descriptions inspired by histograms in the relational databases literature. The experiments and analyses of the effect of more detailed metadata are important in order to make a solid argument in favour of Sevod.

Semagrow is an open source project developed on Github. The version used for the experiments in this paper is tagged as `https://github.com/semagrow/semagrow/tree/1.4.0`

6. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 318497. More details at `http://www.semagrow.eu`

7. REFERENCES

[1] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing linked datasets with the VoID vocabulary. W3C Interest Group Note, 3 March 2011.

[2] C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche. SPARQL web-querying infrastructure: Ready for action? In *Proc. 12th Intl Semantic Web Conference (ISWC 2013), Sydney, Australia, October 21-25, 2013, Part II*, LNCS 8219. Springer, 2013.

[3] A. Charalambidis, S. Konstantopoulos, and V. Karkaletsis. Dataset descriptions for optimizing federated querying. In *24th Intl World Wide Web Conference Companion Proceedings (WWW 2015), Poster Session, Florence, Italy, 18-22 May 2015*, 2015.

[4] A. Charalambidis, A. Troumpoukis, and J. Jakobitsch. *Techniques for heterogeneous distributed semantic querying.* Semagrow Public Deliverable D3.4, 2015.

[5] O. Görlitz and S. Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *Proc. 2nd Intl Workshop on Consuming Linked Data (COLD 2011), Bonn, Germany*, CEUR 782, 2011.

[6] G. Graefe. Iterators, schedulers, and distributed-memory parallelism. *Softw., Pract. Exper.* 26(4), 1996.

[7] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. 23rd Intl Conference on Very Large Data Bases (VLDB'97), Athens, Greece*. 1997.

[8] O. Hartig, C. Bizer, and J. C. Freytag. Executing SPARQL queries over the web of linked data. In *Proc. of 8th Intl Semantic Web Conference (ISWC 2009), Chantilly, VA, USA*. LNCS 5823. Springer, 2009.

[9] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[10] R. Lokers, S. Konstantopoulos, A. Stellato, R. Knapen, and S. Janssen. Exploiting innovative linked open data and semantic technologies in agro-environmental modelling. In *Proc. of the 7th Intl Congress on Environmental Modelling and Software (iEMSs 2014)*, San Diego, USA, 15-19 June 2014.

[11] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. Ngonga Ngomo. A fine-grained evaluation of SPARQL endpoint federation systems. Accepted to *Semantic Web Journal*. 2014.

[12] M. Saleem and A.-C. Ngonga Ngomo. Hibiscus: Hypergraph-based source selection for SPARQL endpoint federation. In *Proc. 11th ESWC Conference, Anissaras, Crete, Greece*, LNCS 8465. Springer, 2014.

[13] M. Saleem, A.-C. Ngonga Ngomo, J. Xavier Parreira, H. F. Deus, and M. Hauswirth. DAW: duplicate-aware federated query processing over the web of data. In *Proc. 12th Intl Semantic Web Conference (ISWC 2013), Sydney, Australia, Part I*, LNCS 8218, 2013.

[14] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig *et al.* Fedbench: A benchmark suite for federated semantic data query processing. In *Proc. of the 10th Intl Semantic Web Conference (ISWC 2011), Bonn, Germany, Part I*, LNCS 7031. Springer, 2011.

[15] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: A federation layer for distributed query processing on Linked Open Data. In *Proc. 8th Extended Semantic Web Conference (ESWC 2011), Heraklion, Crete, Greece*, LNCS 6644. Springer, 2011.