ICT Seventh Framework Programme (ICT FP7)

Grant Agreement No: 318497

Data Intensive Techniques to Boost the Real – Time Performance of Global Agricultural Data Infrastructures

D3.4.2: Techniques for Heterogeneous Distributed Semantic Querying

<table>
<thead>
<tr>
<th>Deliverable Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Reference No.</td>
</tr>
<tr>
<td>Deliverable No.</td>
</tr>
<tr>
<td>Relevant Workpackage:</td>
</tr>
<tr>
<td>Nature:</td>
</tr>
<tr>
<td>Dissemination Level:</td>
</tr>
<tr>
<td>Document version:</td>
</tr>
<tr>
<td>Date:</td>
</tr>
<tr>
<td>Authors:</td>
</tr>
</tbody>
</table>

Document description: This deliverable: a) describes the problems and questions faced regarding the Heterogeneous Distributed Semantic Querying aspect of SemaGrow; b) explains and supports the resulting research and implementation decisions; c) examines the parameters affecting the expected performance of the method; d) states the next steps based on the SemaGrow vision beyond project’s end.


<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author (Partner)</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final v1.0</td>
<td></td>
<td>NCSR-D, UAH</td>
<td>Delivered as D3.4.1</td>
</tr>
<tr>
<td>Draft v1.1</td>
<td>30/10/2014</td>
<td>NCSR-D</td>
<td>Query Decomposition (Section 3)</td>
</tr>
<tr>
<td>Draft v1.2</td>
<td>7/11/2014</td>
<td>SWC</td>
<td>Execution engine (Section 4)</td>
</tr>
<tr>
<td>Draft v1.3</td>
<td>14/11/2014</td>
<td>UAH</td>
<td>Internal review</td>
</tr>
<tr>
<td>Intermediate v1.4</td>
<td>21/11/2014</td>
<td>NCSR-D</td>
<td>Delivered as D3.4.2(i)</td>
</tr>
<tr>
<td>Draft v1.5</td>
<td>20/6/2015</td>
<td>NCSR-D and SWC</td>
<td>Updates of Section 3 and Section 4.</td>
</tr>
<tr>
<td>Draft v1.6</td>
<td>3/7/2015</td>
<td>NCSR-D</td>
<td>Evaluation (section 5)</td>
</tr>
<tr>
<td>Draft v1.7</td>
<td>20/7/2015</td>
<td>UAH</td>
<td>Internal review</td>
</tr>
<tr>
<td>Final v2.0</td>
<td>24/7/2015</td>
<td>NCSR-D</td>
<td>Delivered as D3.4.2</td>
</tr>
</tbody>
</table>
EXECUTIVE SUMMARY

Within the highly volatile and dynamic setting of Linked (Open) Data, the collection of knowledge can be both critical and challenging. In the SemaGrow vision, the process of gathering data from a multitude of sources should be as simple as querying a single, universal data source. In the T3.4 on Heterogeneous Distributed Semantic Querying we study and implement techniques that can realize this vision through state-of-the-art practices and methods.
# TABLE OF CONTENTS

LIST OF TERMS AND ABBREVIATIONS ........................................................................................................ 5

1.  INTRODUCTION ......................................................................................................................................... 6
    1.1 Purpose and Scope ................................................................................................................................. 6
    1.2 Approach ............................................................................................................................................... 6
    1.3 Relation to other Work Packages and Deliverables ............................................................................. 6
        1.3.1 Interactions of Query Decomposition with the Resource Discovery ............................................. 8
        1.3.2 Interactions of Execution Engine with the Query Transformation Source Metadata Access .... 8
    1.4 Objectives ............................................................................................................................................. 8
    1.5 Big Data Aspects ................................................................................................................................. 8

2.  BACKGROUND ............................................................................................................................................ 10
    2.1 The Query Language ............................................................................................................................ 10
        2.1.1 Relational Algebra ......................................................................................................................... 11
    2.2 Query Optimization ............................................................................................................................... 11
        2.2.1 Dynamic Programming ................................................................................................................ 12
        2.2.2 Distributed Query Optimization .................................................................................................. 13
    2.3 Federated Linked Data Querying .......................................................................................................... 13
        2.3.1 Federated LOD Querying Systems ................................................................................................ 14

3.  QUERY DECOMPOSITION .......................................................................................................................... 16
    3.1 Overview ............................................................................................................................................. 16
    3.2 Cardinality and Cost Estimation .......................................................................................................... 18
    3.3 Query Planning ..................................................................................................................................... 20
        3.3.1 Plan Properties and Operators ..................................................................................................... 21
        3.3.2 Plan Generation ........................................................................................................................... 22
        3.3.3 Plan Pruning ................................................................................................................................... 22
    3.4 Implementation ..................................................................................................................................... 23

4.  EXECUTION ENGINE ................................................................................................................................... 24
    4.1 Reactive Execution Engine ................................................................................................................... 24
    4.2 Join Implementation ............................................................................................................................... 24
    4.3 Implementation ..................................................................................................................................... 25
        4.3.1 Query Results Interception .......................................................................................................... 26

5.  EVALUATION ............................................................................................................................................... 27
    5.1 Experimental Setup ............................................................................................................................... 27
    5.2 Query Execution .................................................................................................................................... 27
    5.3 Query Optimization ............................................................................................................................... 30
    5.4 Large RDF Benchmark ......................................................................................................................... 31

6.  CONCLUSION ............................................................................................................................................... 38

REFERENCES .................................................................................................................................................. 39
# LIST OF TERMS AND ABBREVIATIONS

<table>
<thead>
<tr>
<th>Term/Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data source</td>
<td>In the context of this document, one of the SPARQL endpoints federated by the SemaGrow Stack. Besides their location, the SemaGrow Stack also retains metadata about its data sources' contents and schema.</td>
</tr>
<tr>
<td>Heterogeneous data sources</td>
<td>Data sources the data of which follow different schemas.</td>
</tr>
<tr>
<td>Query triple pattern</td>
<td>SPARQL queries are made up of query patterns, subject-predicate-object triples of either specific values or variables. Triple patterns are sometimes also called triple patterns.</td>
</tr>
<tr>
<td>Query fragment</td>
<td>A query fragment is a part of a query that can be executed as a stand-alone query.</td>
</tr>
<tr>
<td>Querying</td>
<td>Querying retrieves a solution sequence that corresponds to the ways in which the query's pattern matches the data that is being queried.</td>
</tr>
<tr>
<td>Query Execution Plan (QEP)</td>
<td>An ordered set of executable steps used to access and combine data.</td>
</tr>
<tr>
<td>SemaGrow Stack</td>
<td>The <em>SemaGrow Stack</em> integrates the <em>SemaGrow</em> components needed in order to offer a single SPARQL endpoint that federates a number of large-scale, heterogeneous data sources, also exposed as SPARQL endpoints.</td>
</tr>
<tr>
<td>SPARQL</td>
<td>SPARQL is a W3C Recommendation for expressing queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. The results of SPARQL queries can be result sets or RDF graphs.</td>
</tr>
<tr>
<td>SPARQL endpoint</td>
<td>A SPARQL endpoint is a machine-friendly (but also human-useable) service that enables clients to query an RDF knowledge base via the SPARQL language.</td>
</tr>
<tr>
<td>SPARQL endpoint federation</td>
<td>A SPARQL endpoint federation provides a single endpoint through which to query distributed remote SPARQL endpoints. Federations are queried in the SPARQL Federated Query Extension, a W3C Proposed Recommendation.</td>
</tr>
<tr>
<td>Triple store</td>
<td>A database that is optimized for the storage and retrieval of RDF triples.</td>
</tr>
<tr>
<td>Principle of optimality</td>
<td>The property that states that an optimal solution of a problem cannot be consisted of suboptimal solutions of its subproblems.</td>
</tr>
<tr>
<td>Cardinality</td>
<td>The cardinality of a set (or a bag) is the number of elements of the set (or a bag).</td>
</tr>
<tr>
<td>Selectivity</td>
<td>The ratio of the elements of a relation that satisfy a predicate to the elements of the relation. Indicates how selective is a predicate applied to a relation.</td>
</tr>
<tr>
<td>OpenRDF Sesame</td>
<td>A de-facto standard framework for processing RDF data.</td>
</tr>
<tr>
<td>Sesame SAIL</td>
<td>A set of programming interfaces of OpenRDF Sesame that enable the implementers to create stackable layers of functionality.</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

1.1 Purpose and Scope

This report documents the Query Decomposition and Execution Engine components of the SemaGrow Stack. The Query Decomposition component decomposes the original query to appropriate query fragments that are passed as an execution plan to the Execution Engine component in order to be executed. The decomposition takes into account knowledge from the Resource Selector (T3.1) about the data sources that are likely to have results related to the query and selects an optimal execution plan. The execution plan is passed to the Execution Engine component that is responsible to (a) translate the query fragments into valid SPARQL queries that will be issued to the appropriate SPARQL endpoints, (b) collect the results and (c) transform, merge and join them and return the final result to the user.

In this context, the work carried out in Task 3.4 and reported in this document pertains to:

- Developing the Query Decomposition component, the component that uses metadata from the Resource Selector in order to select an optimal execution plan that can be executed by the Execution Engine component.
- Developing the Execution Engine component, the component that will execute a plan by distributing parts of the plan to different data sources and efficiently merge the query results into a homogeneous response to the original user.
- Measuring the efficiency and optimality of the Query Decomposition and the performance of the Execution Engine.

The rest of the document is structured as follows. In the rest of the current section we discuss the relation of this report with other deliverables and illustrate the interactions of the implemented components with others on the SemaGrow Stack. In Section 2 we provide a brief introduction about relevant concepts and notions and discuss well-known techniques for query optimization. Furthermore, we discuss related systems on the Federated Linked Data querying context. In Section 3 we present the main techniques used for the implementation of the Query Decomposition component and in Section 4 we discuss implementation issues for the Execution Engine components.

1.2 Approach

The aim of this work package (WP3 - Robust and Scalable Techniques) is to carry out the fundamental research needed for the development of new optimization algorithms or the extension and adaptation of existing ones, for scaling up data intensive techniques and the analysis of their complexity.

1.3 Relation to other Work Packages and Deliverables

The SemaGrow Stack integrates the components needed in order to offer a single SPARQL endpoint that federates a number of heterogeneous data sources, also exposed as SPARQL endpoints. The main difference between the SemaGrow Stack and most existing distributed querying solutions is that SemaGrow targets the federation of heterogeneous and independently provided data sources. In other words, SemaGrow aims to offer the most efficient distributed querying solution that can be achieved without controlling the way data is distributed among sources and, in general, without having the responsibility to centrally manage the data sources of the federation.
Figure 1: Position of the Query Decomposition and Query Execution Engine components in the overall architecture of the SemaGrow Stack
We see two distinct challenges in querying a federation of big data sources: support different types of queries with answer sets of varying sizes.

Query Decomposition techniques. Firstly, SemaGrow setting synergy with the source selection mechanisms, this will allow queries in any schema to be executed at all and only the original responses of the data sources are intercepted and logged in order to be passed to the Source Metadata Access component for further analysis. These results will later update the metadata store in order to produce more accurate estimations for future queries.

1.4 Objectives
The objectives of the Task 3.4 is to extend and develop query decomposition techniques that will enable complex queries in one schema to be broken down into sub-queries, each of which can be in a possibly different schemas. In synergy with the source selection mechanisms, this will allow queries in any schema to be executed at all and only those repositories that might hold relevant information, regardless of the schema these repositories use. The SemaGrow setting also poses an important set of restrictions related to the nature and expected performance of the Query Decomposition techniques. Firstly, we cannot access the internals of the data sources and the data sources may not be even aware that they are part of the federation. Moreover, the availability of the individual data source is unknown a priori and thus runtime information may be critical. The system needs to be scalable and efficient, to support different types of queries with answer sets of varying sizes.

1.5 Big Data Aspects
We see two distinct challenges in querying a federation of big data sources:

- Small results from big data: in this “needle in a haystack” situation we are joining properties from different big datasets in order to retrieve a result set that does not constitute big data by itself. The challenge here is to have an intelligent query execution planner that guides the executor engine along a query execution plan that never retrieves large quantities of results that will fail to join with subsequent query patterns. This is discussed in Chapter 3 of this deliverable.

- Big results from big data: in this situation the result set constitutes big data, and no amount of query plan optimization can avoid this since this is what has been requested by the client. In this case the execution
Engine must efficiently use its computational, memory, and disk resources in order to compute and dispatch results as quickly as they are coming in from the distributed data sources and, thus, avoid retaining large partial results. This is discussed in Chapter 4 of this deliverable.
2. BACKGROUND

In this section we will give a brief introduction on the basic notions of the query optimization, related work and background.

2.1 The Query Language

A query is usually a form of expressing in a declarative way the data to be retrieved from a database abstracting away from the user the physical location and arrangement of the raw data. Queries can be formally expressed by query languages, such as SQL, which is the de-facto query language in the Relational Database Management Systems (RDBMS). The SPARQL language is similar to SQL and is well-known as the de-facto query language to query RDF graphs.

SPARQL is a W3C Recommendation for expressing queries across data that is stored natively as RDF or viewed as RDF via middleware. Since RDF data can be considered to form a graph, SPARQL is a language to query for graph patterns, i.e. subgraphs that satisfy certain restrictions. On the other hand, RDF data can also be viewed as relational and hence SPARQL and SQL have many features in common. In fact, SQL and SPARQL share the same underlying theoretical foundations [1], [2].

We consider three disjoint sets, namely B (blank nodes), U (URIs) and L (Literals). We also write BLU as an abbreviation for the union of the sets. Blank nodes are nodes that are named locally and cannot be used outside of that scope. An RDF triple (s,p,o) in BUxUxBLU, relates subject s through predicate p to object o. Subjects can be resources namely either URIs or blank nodes, predicates can be URIs and objects can be either URIs, blank nodes or literals. A set of RDF triples form a RDF graph and is stores in a RDF triple store.

Let V be a set of variables disjoint from BLU. We distinguish variables from resources and literals by a leading question mark. We write, for example, ?x, ?y, ?name.

A triple pattern is a triple that can occur variables in either subject, predicate or object.

A SPARQL expression is an expression that is defined recursively as follows:

- A triple pattern is a SPARQL expression
- If E1 and E2 are SPARQL expressions, then (E1 JOIN E2) is a SPARQL expression. We will also write (E1. E2) to denote the same expression.
- If E is a SPARQL expression and R is a filter condition then (Q FILTER R) is a SPARQL expression.
- If E1 and E2 are SPARQL expressions, then (E1 OPTIONAL E2) is a SPARQL expression.
- If E1 and E2 are SPARQL expressions, then (E1 UNION E2) is a SPARQL expression.

We will also call basic pattern group a conjunction of triples patterns with optionally a set of filters.

A simple SPARQL query is defined as (SELECT ?x1, ... ?xn FROM G WHERE { E }) where E is a SPARQL expression, ?x1, ... , ?xn are variable that occur in E and G is an RDF graph. A SPARQL query can also have an ORDER BY or GROUP BY or LIMIT directive as in SQL. The full syntax kai capabilities can be found in [?].

Consider the following SPARQL query that requests for all situations where a potato crops yielded more than 5 while precipitation was less than 5”, expressed in SPARQL:

```
SELECT ?X WHERE {
  FILTER (?Y > 5 && ?P < 5) }
```
2.1.1 Relational Algebra
Relational algebra [3] provide the theoretical foundation of both SQL and SPARQL [1]. Every query that is expressed in SQL or SPARQL have an algebraic representation in the relational algebra and can be viewed as an algebraic tree where (a) the internal nodes of the tree are relational operators, (b) leaves of the tree are relations (or triple patterns respectively) and (c) subtrees are subexpressions of relational algebra.

We can convert an SPARQL query to each algebraic representation as follows:

- A triple pattern t is considered as a base relation [t].
- A SPARQL expression (E1 JOIN E2) is considered as a join of the two relations [E1] and [E2].
- A SPARQL expression (E1 OPTIONAL E2) is considered as a left outer join of the two relations [E1] and [E2].
- A SPARQL expression (E FILTER R) is considered as a selection of relation [E] based on predicate R.
- A SPARQL expression (E1 UNION E2) is considered as a union of the two relations [E1] and [E2].
- A SPARQL query that have a (SELECT ?x1, .. ?xn { E }) is considered as a projection of the relation [E] on the attributes ?x1, ..., ?xn.

Note that the algebraic representations carry no information about the methods to be used for retrieving the data. Also, note that the algebraic representation of SQL and SPARQL does not differ significantly on the internal nodes of the algebraic tree. Therefore, since query optimization is performed on the algebraic representations of the queries, the same basic ideas and techniques can be employed to optimize a SPARQL query.

2.2 Query Optimization
Given a query, there are many plans that a database system can consider to process it and retrieve the answer. All those plans are equivalent in the sense that will return the same answer, but they vary in their cost, that is the resources and the amount of time that they need to run. This is stemmed from:

a) The algebraic representation of a query can be transformed into many other logical equivalent algebraic representations of the same query. This is mainly due to the associativity, commutativity and distributivity of the join operator.

b) For a given algebraic representation there are many trees of physical operators that can implement the algebraic expression. Typically there exist many implementations of the join operator, for example the nested loop join, the hash join and the merge join.

Consider the following simple SPARQL query that requests for the titles of the publications that are annotated with the keyword called “Potato”:

```
PREFIX dct:<http://purl.org/dc/terms>
PREFIX dc:<http://purl.org/dc/elements/1.1>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>

SELECT ?title {
    ?keyword rdfs:label “Potato”.
}
```

Assume that there are 1000 publications and 10 total keywords that can be used to annotate every publication. Each publication is related with 3 keywords in average. A potential execution plan will fetch all the titles, join them with all the potential keywords and then join them with the keywords that have the label “Potato”. Then first join will yield approximately 3000 tuples that will be later filtered out by a significant factor. On the other hand, an execution plan that will first fetch the keywords that are labelled “Potato”, and use that intermediate result to fetch only the relevant publication seems more reasonable.
Such *query optimization* is absolutely essential for a database system. The cost difference between two equivalent plans can be enormous. The task of translating a user query into an execution plan is undertaken by the component that is usually called, in the database literature, *query optimizer*.

The area of query optimization has been a popular subject of research for many years in the database community. It has been studied in a variety of contexts and from many different angles. The interested reader can consult [4], [5] for a detailed presentation of the different approaches on the query optimization.

Query optimization is considered a difficult problem and can be formalized as a search problem that requires:

- a space of query plans (*search space*);
- a *costing estimation* to assign a cost to each plan in the search space. Intuitively, that represents the estimations of the resources need to execute the plan;
- an *enumeration algorithm* that searches among the possible query plans and selects the one with the least cost.

In fact, even for conjunctive queries (i.e. queries that consist only of inner joins) with a single physical implementation of join, the query optimization problem is NP-Hard [6]. To overcome the exponential nature of the problem other approaches use greedy heuristics that are tailor-made on specific assumptions and cannot guarantee finding an optimal solution for the problem.

In the following sections we introduce the most popular algorithm for query optimization that is currently implemented in many commercial relational database systems. Moreover, we discuss how this algorithm can be extended to be used in parallel or distributed environment.

### 2.2.1 Dynamic Programming

Dynamic Programming was first proposed as an enumeration algorithm for query optimization in the context of the implementation of System R [7]. Commercial systems have then used it successfully in various forms and with various extensions. The advantage of dynamic programming is that is produces the best possible plan given that the cost function is sufficiently accurate. The essence of Dynamic Programming approach is based on the assumption that the cost function respects the *principle of optimality*. Specifically, it assumes that in order to take an optimal plan for a query consisting of k relations, it suffices to consider optimal plans of (k-1) relations and extend those plans with an additional join. In other worlds, suboptimal plans of (k-1) relations do not need to be further considered in determining the optimal plan, and thus can be pruned. The algorithm view the SPJ query as a set of n relations. It proceeds bottom-up constructing *access plans* for the single relations first. Access plans will enumerate all alternative ways to access the base relation (e.g. index scan, table scan) that are known to the optimization. Then, the algorithm enumerates all the two-way joins using access plans as building blocks. Again, the algorithm will enumerate all the alternative join plans considering all the available join algorithms (e.g. nested loop join, merge join, hash join) to the system. At the end of the j-th step the algorithm produces intermediate plans for expressions consisting up to j relations. The algorithm will assign a cost to each of those j-way join plans and keep only those with the minimum cost, drastically reducing the enumeration space for more complex plans. The algorithm continues in this way until all n-way join plans are enumerated which are complete plans for the query.

One of the join methods that is usually implemented by a database system and therefore consider by the Dynamic programming algorithm is *Merge Join*. Merge Join first sorts the two input relations on the corresponding join attributes and then perform a synchronized scan on both inputs joining tuples with the same join attributes and the output stream is also sorted on the join attributes. If any of the two relations are already sorted (either by an access plan that maintain the data of a relation in an ordered sequence, or as a result of a previous merge join) then, the first step of the merge join can be omitted and the cost of the merge join become linear with respect to the cardinality of the input relations. Hence, given two intermediate plans, cannot compare them based on their cost only and prune the more expensive one. The algorithm must also consider the sorted order (if any) in which their results come out. This is clearly a violation of the principal of optimality, since plans that are suboptimal but have a sorting order can be later used as parts of an optimal more complex plan. To overcome this problem the original algorithm introduce the notion
of an interesting order and does not prune plans with different interesting orders. A variation of the original algorithm proposed in [8] also relaxes the restriction the cost function must respect the principle of optimality. The proposed algorithm can handle plans that are partial ordered, and prune only plans that are genuinely suboptimal.

The original algorithm proposed in [7] considers only left-deep trees; trees that for each join node their right operand is always a relation. The reason behind this restriction is the reduction of the search space of the possible execution plans. However, bushy trees (trees that both join operands can be complex subexpressions) can also be considered as viable alternative plans and usually have better execution time especially when the plan can be executed in parallel. A variant of Dynamic Programming algorithm has been proposed in [9] for bushy trees with cartesian products. DPCCP [10] is also a variant of Dynamic Programming algorithm, that is optimal in terms of the lower bound of the problem. The main idea of this algorithm is to enumerate only pairs of connected subsets and thus avoiding enumerate subsets that lead to Cartesian products, since plans with Cartesian products are always suboptimal.

2.2.2 Distributed Query Optimization

In distributed databases, the data are typically distributed among different sites interconnected over the network in a share-nothing architecture. The user submits a query into a central point without been aware of where the data resides and the data are shipped to convenient sites, processed and the final result is presented to the user at the site of preference. The query is optimized centrally and the optimization decides not only the execution order of the operations but also the locations of those executions. For example, assume that the original query is (A ⨝ B ⨝ C) and A resides at site S1 and B and C at site S2 with a request for the result to be delivered at site S1. The optimizer may decide based on the cost (both communication cost and workload cost) to perform the join of B and C locally at site S2 and then ship the result to site S1 to be joined with the relation A and present the answer at site S1.

The dynamic programming algorithm can be smoothly extended to accommodate such decisions [11]. Specifically, the dynamic algorithm must take into account the site where the plan will be executed. Moreover, there is also the notion of interesting site, similar to the one of interesting orders. Plans with different interesting sites cannot be compared since joins are less expensive if both relations are on the same location. The enumeration space the algorithm depends on both the number of the relations and the number of the potential sites and can be prohibited for a large number of sites. An extension of dynamic programming, known as iterative dynamic programming [12], uses dynamic programming along with a greedy algorithm in order to produce optimal plans when possible, or suboptimal plans based on the heuristics when dynamic programming is not viable.

Another interesting subject of research for distributed query optimization is the case where different heterogeneous data sources are consolidated under a central query optimization. By heterogeneous sources we mean sources that have different query capabilities and access APIs. The difference of the standard distributed scenario is that an extended query optimization algorithm is needed to be used. In [13] proposed an extension of dynamic programming algorithm that handle heterogeneous data sources. The main idea is to encapsulate the details of component databases with a wrapper, a module that translates every request of the central execution engine so that the request can be understood by the component database’s API. The wrapper also translates the results returned so that the results can be compliant with the global schema of the distributed database.


2.3 Federated Linked Data Querying

Federated data sources are loosely coupled and typically controlled by independent third party providers which means that details about the statistics and physical organization of the data might be unavailable to the federation. In many cases the data sources can be unaware that they are part of a federation. The federation of heterogeneous data sources has been a popular topic in database research for a long time. In this section we focus on the federation of (LOD) Linked Open Data sources. In fact the challenges for federated database are quite similar to the ones in the federated Linked Data scenario. However, there are also differences. Federated and distributed databases typically use wrappers to abstract away the communication details and the diverse schemata used in different database instances.
Such wrappers are not essential in the LOD scenario, since data can be provided by a standard SPARQL endpoint. However, wrappers are typically involved in the estimation of cardinalities and processing costs.

In the following we provide a brief comparison among the state-of-the-art Linked Open Data federation systems and techniques that they use to perform query optimization.

### 2.3.1 Federated LOD Querying Systems

There are numerous systems and approaches for federating Linked Open data. In this section, we focus on the most well-known systems that federate SPARQL endpoints that are closely related to the SemaGrow case. For a more thorough comparison between state-of-the-art federated querying systems, please consult [14], [15].

#### 2.3.1.1 FedX

The FedX system [16] is a well-established system of federated querying. It follows the Sesame allows querying several SPARQL endpoints, without any a priori knowledge of their internal implementation and data. FedX does not use any statistics to estimate the result cardinalities of the query. It uses coarse-grained heuristics as a function to provide a rank on joins to a greedy optimization algorithm. Although the algorithm can potentially produce suboptimal plans its complexity is linear and thus can return fast a query decomposition. Furthermore, it groups query fragments that are identified to be executed on the same source in order to push down the join to the source. During the query execution phase it implements bind join algorithm. Roughly speaking, it combines the result of one fragment to filter the results of a different fragment on a different source by passing these results as bindings to the SPARQL query. Moreover FedX developed a threaded execution engine in order to handle many SPARQL queries in parallel maximizing the efficiency of the query. Overall, FedX has been shown to perform well [15] as compared to other state-of-the-art systems, like Sesame (Alibaba) [17] and SPLENDID [18]. However, it seems that the performance gain over the other system are mainly due to its execution engine efficiency while the query optimization phase is irrelevant due to the low cost queries of the benchmarks.

#### 2.3.1.2 SPLENDID

SPLENDID [18] is also developed on the Sesame interfaces as FedX. However, it departs from the techniques used by FedX. Specifically, SPLENDID employs VoID descriptions for the estimation of the cardinality and the cost of the query. Moreover, it uses VoID descriptions to identify candidate data sources; sources that have a possibility to contain matching triples. In case of nonexistence of such descriptions it uses ASK queries, similar to the FedX approach, as a fallback mechanism. SPLENDID uses the dynamic programming algorithm to find the optimal query decomposition based on the aforementioned cost estimation. Moreover it groups heuristically triple patterns of the same source together as a single access plan. Overall, SPLENDID has competitive performance compared to FedX. This is mainly due to selecting the optimal query execution plan and a more educated estimation of cardinalities and costs in comparison with FedX.

#### 2.3.1.3 LHD

LHD [19] implementation is focused on developing a cost model based on VoID descriptions that enables parallel execution of plans. As in SPLENDID, the query optimization phase uses the dynamic programming algorithm, however due to the different cost model it favours bushy execution plans that can be executed in parallel. The performance of LHD is moderate in comparison with FedX and SPLENDID and possible this is due to the simple implementation of the execution engine that cannot exploit the needed parallelism.

#### 2.3.1.4 ANAPSID

The ANAPSID system [20] departs from the standard ideas developed in the aforementioned systems. It develops an adaptive execution engine that is keen to hide unexpected delays from the sources. Its query execution engine adapts to the data availability and runtime condition of each SPARQL endpoint. This framework provides physical join operators that detect when a source becomes blocked and they produce results as quickly as they arrive. ANAPSID make use of both a VoID catalogue and ASK queries to perform the source selection step.
2.3.1.5 QUETSAL

By contrast to the aforementioned systems, Quetsal\(^1\) leverages the additional SPARQL 1.1. functionality for accessing federated endpoints using an explicit SERVICE keyword. The basic idea is to automatically rewrite an ordinary SPARQL query without into an enhanced query with SERVICE constructs that can be executed by a SPARQL 1.1 enabled system. In other words, the scope of Quetsal is restricted to source selection and not full optimization. In its distribution, Quetsal is bundled with FedX to carry out query execution.

\(^1\) Please cf. https://code.google.com/p/quetsal
3. QUERY DECOMPOSITION

In this section we will discuss the algorithms and the techniques used for developing the Query Decomposition component that is responsible for the quality of the execution plans. The Query Decomposition component is positioned at the core of the SemaGrow Stack receiving original queries and proposing an optimal execution plan to the execution engine. In a high-level the Query Decomposition component receive input from the Resource Discovery component and sends output to the Execution Engine component (see Section 4).

3.1 Overview

In order to efficiently process a query, federated querying systems narrow the scope of query execution and 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 from the FedBench benchmark.

```
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)
}
```

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.

First, the query string is parsed into a tree where the leaves correspond to triple patterns (P1 through P5) 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 reordering 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.

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 3 for our example query.
In this structure we have introduced the following elements to the standard syntax used to represent the parsed query tree:

- 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 **Cardinality** annotation that gives the estimated cardinality of the results of an operator.
- The **Cost** annotation that gives the estimated cost of an operator.

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

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), 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 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 (or 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.

Before defining the cost function that we will use to evaluate the execution plans we need to define the cardinality estimation of an expression. The cardinality can be estimated using the source metadata maintained by the Resource Discovery component. Given a triple pattern the Resource Discovery component will return a list of candidate data sources that is believed may contain matching triples along with statistics about this data sources. These statistics is:

a) the estimated number of the triples that match the triple pattern in the given data source,

b) the estimated distinct subjects matching the triple pattern,

c) the estimated distinct predicates matching the triple pattern,

d) the estimated distinct objects matching the triple pattern.

The rest of the information retrieved from the Resource Discovery component will be used by the plan generation functions and will be discussed later.

The cardinality of a triple pattern is provided by the Resource Discovery component. On the other hand, for more complex expressions we need to make an estimation based on our available statistics. In order to estimate complex expressions based on the aforementioned basic statistics we adopt the formulas described by Wang and Davies [19].

First, we need to define the notion of the selectivity factor. Given a generic predicate p and a relation R the selectivity factor is defined as the ratio of the number of the tuple that satisfy predicate p over the total cardinality of R. By estimating this factor we can estimate the cardinalities of more complex expressions such as joins. For our approach we distinguish two kinds of selectivity factors: the join selectivity that is applied to join expressions and the filter selectivity that is applied to filter expressions.

Join Selectivity Estimation

The Join Selectivity estimation JoinSel(E1 \cdot E2) of a join expression E is defined as follows:

- \text{JoinSel}(E1 \cdot E2) = \min\{\text{JoinSel}(E1), \text{JoinSel}(E2)\}
- \text{JoinSel}(T) = \min\{1/d_i\}

where E1 and E2 are any pattern or join expression, T is a simple triple pattern, and d_i is the number of distinct values for the i-th join attribute in triple pattern T.

Note that for the computation of the join selectivity we assume:

- **Independence** between the join arguments
- **Containment** of the smallest (in number of tuples that satisfy it) pattern into the other, so that selectivity is determined by the number of distinct values
• **Uniformity** in the distribution of distinct values, so that for a given pattern, the specific values for the positions in the pattern (subject, predicate, or object) that are binded do not influence the computation.

It is obvious that these assumptions do not hold in the general case. Therefore, the success of optimization strategies depends on the quality of the metadata. Specifically, it depends on the availability of distinct values counts for *subsets* of the complete dataset, where these subsets partition the dataset in such a way that the assumptions above hold.

**Filter Selectivity Estimation**

The Filter Selectivity estimation $Sel(C, E)$ of a filter expression $(E \text{ FILTER } C)$ is defined as follows:

- $Sel(C1 \text{ AND } C2, E) = Sel(C1, E) \times Sel(C2, E)$
- $Sel(C1 \text{ OR } C2, E) = \max\{Sel(C1, E), Sel(C2, E)\}$
- $Sel(\neg C, E) = 1 - Sel(C, E)$
- $Sel(E) = 1/2$.

We can now define the cardinality estimation:

**Cardinality Estimation**

Given a source $S$, the cardinality estimation $Card(E, S)$ of an expression $E$ is defined recursively as follows:

- $Card(s \text{ p o }, S) = Card_{spo}$ given by the Resource Discovery
- $Card(E1 \bowtie E2, S) = Card(E1, S) \times Card(E2, S) \times JoinSel(E1 \bowtie E2)$
- $Card(E1 \text{ OPTIONAL } E2, S) = Card(E1, S) \times Card(E2, S) \times JoinSel(E1 \bowtie E2) - Card(E1) \times Card(E2) \times (1 - JoinSel(E1 \bowtie E2))$
- $Card(E1 \text{ UNION } E2, S) = Card(E1, S) + Card(E2, S)$
- $Card(E \text{ FILTER } R, S) = Card(E, S) \times Sel(R, E)$

We write $Card(E)$ to denote the sum of the cardinality estimations over all the candidate data sources for expression $E$.

Consider as an example the query in the previous section and assume that in DrugBank there are the following statistics:

- for the pattern $?X \text{ db:drugCategory } ?Y$ there are 4602 matching triples with 1879 distinct subjects and 584 distinct objects;
- for the pattern $?X \text{ db:casRegistryNumber } ?Y$ there are 2240 matching triples with 2240 distinct subjects and 2218 distinct objects.

By assuming independence and homogeneous distribution of values we can calculate the estimated matching triples of the pattern $?X \text{ db:drugCategory db:micronutrient}$ to be 8 by dividing the total triples with the distinct number of objects. Moreover, the join selectivity of $P1\bowtie P2$ on the variable $?X$ can be estimated as $\min(ds-1, ds-1)$ where $ds$ is the distinct 12 values of $?X$ in each of the triple patterns. In our example, the selectivity factor is estimated to be $\min(1/1879, 1/2240) = 1/2240$. The estimated cardinality of $P1\bowtie P2$ is then the product of their estimated cardinalities and the selectivity factor, thus $8 \times 2240/2240$ equals 8. However, the actual cardinality of $P1\bowtie P2$ 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 cost of each expression is evaluated based on the cost estimates for the individual operands. Since there are various approaches on defined cost functions we used a traditional formulas in distributed databases [11], [18]. This simple cost function models the overall effort spent to compute the expression under consideration. Let the constants $C_{\text{transfer}}$ and $C_{\text{process}}$ be the communication and computation unit cost for a single tuple respectively. Moreover, $C_{\text{query}}$ is the cost involving the execution of a query to a remote source. These constants can be given by the system configuration and/or on a data source basis.
Cost Estimation

The cost function \( \text{Cost}(E) \) of an expression \( E \) is defined recursively as follows:

- \( \text{Cost}(s \ p \ o) = \text{Card}(s \ p \ o) \times \text{C\_process} \)
- \( \text{Cost}(E1 \text{ merge join } E2) = \text{Cost}(E1) + \text{Cost}(E2) + (\text{Card}(E1) + \text{Card}(E2)) \times \text{C\_process} \)
- \( \text{Cost}(E1 \text{ hash join } E2) = \text{Cost}(E1) + \text{Cost}(E2) + \text{Card}(E1) \times \text{C\_process} + \text{Card}(E1 \Join \ E2) \times \text{C\_process} \)
- \( \text{Cost}(E1 \text{ bind join } E2) = \text{Cost}(E1) + \text{C\_transfer} \times \text{Card}(E1 \Join \ E2) + \text{C\_process} \times \text{Card}(E1) + \text{C\_query} \times [\text{Card}(E1)/\text{batch}] \)

where \( \text{batch} \) is the number of bindings of \( E1 \) dispatched to the source of \( E2 \) by a single query, so that \( \text{Card}(E1)/\text{batch} \) is the number of queries needed in order to dispatch all bindings.

- \( \text{Cost}(E1 \text{ union } E2) = \text{Cost}(E1) + \text{Cost}(E2) \)
- \( \text{Cost}(E \text{ filter } R) = \text{Cost}(E) + \text{C\_process} \times \text{Cost}(E) \)
- \( \text{Cost}(\text{Sort}(E)) = \text{Cost}(E) + \text{C\_process} \times \text{Card}(E) \times \log(\text{Card}(E)) \)
- \( \text{Cost}(\text{SourceQuery}(Q)) = \text{C\_query} + \text{C\_transfer} \times \text{Card}(Q, \text{Site}(Q)) \)

Note that the SPARQL operators for which the cost function is defined above, covers all expressions for which meaningful optimization can be performed by the planner. SPARQL constructs, such as GROUP, that are not covered are left as they appear in the original query, although inner expressions will get optimized. The planner optimizes the maximal sub-queries that are fully covered, leaving the relationship between such sub-queries outside the scope of optimization. For example, in an expression such GROUP \( G \) (\( T1, T2 \)) the inner join \( T1 \Join T2 \) might get re-ordered although the cost of the overall expression is not defined and the relationship of GROUP \( G \) (\( T1, T2 \)) to the rest of the query will not be altered.

Note also that, in contrast to the cardinality estimation function, the cost function is dependent on the specific join algorithm. We assume that the decomposer can select between bind join, merge join and hash join each having a different cost for given \( E1, E2 \). For example, consider the Bind join operator (that will be discussed in details in Section 4.2.1.1). An intuitive description of the evaluation of \( (E1 \text{ bind join } E2) \) is: The execution engine first evaluates \( E1 \), and the results of \( E1 \) will be served as bindings for the evaluation of \( E2 \). If the evaluation of \( E1 \) yields \( n \) different tuples, then the evaluation of \( E2 \) will be performed \( n \) times, each time with a different binding. It should be noted here that certain optimizations can be applied in a bind join implementation that will group a set of bindings into a single batch evaluation. Therefore, if the batch size is \( b \), the actual number of the \( E2 \) evaluation is reduced to \( n/b \). Since its evaluation of \( E2 \) results to a separate query to the underlying data source it is easy to verify that \( E1 \) with smaller cardinalities will result to a lower cost expression. This intuitive observation is depicted to the cost formula of bind join. Notice that the cost of the bind join is asymmetric, namely the cost of \( (E1 \text{ bind join } E2) \) can differs vastly from the cost of \( (E2 \text{ bind join } E1) \). On the other hand, the cost of merge join is symmetric since the algorithm must evaluate both \( E1 \) and \( E2 \) independently.

3.3 Query Planning

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. It takes as input a basic pattern group, i.e. a part of the algebraic tree of a query that consists only of joined triple patterns and optionally a set of filters. It returns as output a potential reordering of the join operators and each operator is annotated with the specific algorithm to be used. The pseudocode of the generic Dynamic Programming algorithm is given in Figure 4.
1. **Input:** a BPG query q with triple patterns \{T1, ..., Tn\} and filters \{C1, ..., Ck\}
2. **Output:** An execution plan for q.
3. **For** i = 1 to n **do** {
   4. \( \text{optPlans}(\{T_i\}) = \text{accessPlans}(T_i) \)
   5. \( \text{prunePlans}(\text{optPlans}(\{T_i\})) \)
5. **}**
6. **For** all subsets \( S1, S2 \) such that \( S = S1 \cup S2 \) i = 2 **do** {
7. \( \text{optPlans}(S) = \text{optPlans}(S) \cup \text{joinPlans}(\text{optPlans}(S1), \text{optPlans}(S2)) \)
8. \( \text{prunePlans}(\text{optPlans}(S)) \)
9. **}**
10. **}**
11. **FinalizePlans** (optPlans(\{T1, ..., Tn\}))
12. OptimalPlan = bestPlan(optPlans(\{T1, ..., Tn\}))

**Figure 4 The generic Dynamic Programming algorithm**

In the first phase (lines 3-6) the algorithm considers all the single-relation plans and for each relation create a set of access plan using the procedure accessPlans. In the second phase (lines 7-10), dynamic programming considers all the possible ways to join the plans. First, it considers 2-way joins by using the access plans produced in the first phase and calling joinPlans procedure to build a join plan for these plans. The joinPlans considers all the possible join algorithms that can be used. From the 2-way joins and the access plans the algorithm then produces 3-way join plans. In the same way, dynamic programming continue to produce more complex plans up to n-way join plans. Then, the finalizePlans is called to create complete plans from the n-way joinPlans. At each iteration the inferior plans are pruned. This process is carried out by the prunePlans procedure. Lastly, since the n-way optimal plans can be more than one (due to plans that are incompatible), the last step selects one of the optimal plans as the plan to return.

A point to notice is that the algorithm is generic enough to be applied in many environments. All the details are hidden on the procedures of accessPlans, joinPlans, prunePlans, finalizePlans and bestPlan. The input filters are applied by the accessPlans and joinPlans as soon as possible (i.e. a filter can be applied to an expression if all the variables that occur in the filter also occur in the expression). This means that filters are applied as early as possible during the execution. This heuristic does not always produce optimal plans, especially in cases where the filter predicates are expensive. However, in cases where the predicates are built in this heuristic is proven to produce satisfactory plans. Another point to notice is that the algorithm enumerates all the bushy trees instead of only the so-called left-deep trees as the original algorithm [7] does.

### 3.3.1 Plan Properties and Operators

Recall that execution plans are annotated trees of physical operators that contain information that is useful for the query execution phase. Each plan is annotated with a set of properties. We consider the following properties:

- **Filters** property maintains the filters applied to the plan.
- **Cost** property maintains the computed cost of the plan.
- **Cardinality** property maintains the estimated cardinality of the output of the plan.
- **Site** property maintains the location of the execution of the plan. This property can be either the remote endpoint or local.
- **Ordering** property maintains the ordering (if any) of the plan output.
- **Vocabularies** property maintains the set of Vocabularies for each variable of the plan.
1. **Input:** A collection of plans $P$.
2. **Output:** The reduced collection of plans $P_{best}$.
3. $P_{best} = \emptyset$
4. **For** all plans $p$ in $P$ **do** {
5. a. If there is no $p'$ in $P\setminus\{p\}$ such that $p' \preceq p$ **Then** {
6. b. $P_{best} = P_{best} \cup \{p\}$
7. }
8. }
9. **Return** $P_{best}$

Figure 5 Identifying minimal plans

### 3.3.2 Plan Generation

To handle the plan properties the following operators must be considered by the plan generation:

- The **Sort** operator will change the Ordering property of a plan and
- The **SourceQuery** operator will change the Site property of a plan.
- The **Transform** operator will change the Vocabularies property of a plan.

In **Figure 4** we used procedures to hide the specificities around optimizing federated SPARQL queries from the generic enumeration algorithm. These procedures are:

The **access plans** procedure takes as input a triple pattern and returns an execution plan. The procedure consults the Resource Discovery component in order to get a list of candidate sources. For every data source, the triple pattern is translated (if needed), the applicable filters are applied, and a plan is created for each data source. The plans are annotated with the location that will be executed. In case of multiple data sources, all the resulting plans are combined with a union that is returned as the access plan of the triple pattern. If two data sources is known to have mirror of the dataset then alternative access plans are created. Finally, the Cost and Cardinality properties of the returned plan are updated.

The **join plans** procedure considers all the possible join algorithms and for each type of join create an alternative plan. Some join request certain properties for the underlying plans. For example, the merge-join request for its input to be sorted on the join variables. Another example is the remote join can only be applied when both plans reside on the same location. If the subplans does not comply with the requirements of the operator, the procedure “enforces” (if possible) the requested properties. For example, the procedure insert a Sort operator before the merge join operator. Again, filters are applied whenever possible and the Cost and Cardinality properties are updated with respect to the cost and cardinality estimations.

The **finalize plans** procedure applies any remaining filters and enforces extra operators if needed (e.g. Sort, SourceQuery).

### 3.3.3 Plan Pruning

The plan pruning procedure will reduce the plans that are produced in the current iteration, pruning the inferior ones. In order to specify which plan is inferior we establish a partial order between the plans.

A plan $p_1$ is less than $p_2$ if and only if for every plan property $x$, $p_1.x \preceq p_2.x$. In case of informative properties, for example the cardinality property, the order relates with all the possible cardinalities, thus becoming irrelevant to the order of the plans. The property ordering are the following:

- $p_1.Cost \preceq p_2.Cost$ is the usual numerical ordering.
- $p_1.Ordering \preceq p_2.Ordering$ holds if $p_2.Ordering$ is a prefix of $p_1.Ordering$
- $p_1.Site \preceq p_2.Site$ if $p_1.Site = p_2.Site$
• \( p_1.\text{Filters} \leq p_2.\text{Filters} \) if \( p_2.\text{Filters} \) is a subset of \( p_1.\text{Filters} \)

The algorithm in Figure 5 will keep the plans that are the minimal with respect to the partial order defined.

### 3.4 Implementation

The Query Decomposition and the Execution Engine components being at the very core of the SemaGrow Stack are implemented in Java, based on the open source project OpenRDF Sesame [17]. Moreover, Sesame API is well-known to many researchers and many of the mainstream system expose their functionality as a Sesame SAIL API.

The Query Decomposer component reuses some of the boilerplate code Sesame such as:

- The Parser of SPARQL Queries that parses a string into an intermediate tree representation.
- The Query Response writer that transforms a stream of query bindings into a compliant SPARQL response of the appropriate format (e.g. JSON, RDF/XML, CSV).

The Semagrow Stack is implemented as a Repository that provide to the user a repository connection. The user can use the connection to evaluate queries. The SPARQL queries are firstly parsed to the Sesame internal representation, namely, into an instance of a TupleQuery. Once the query is parsed, the user can “evaluate” the query using the method `TupleQuery.evaluate`. This method triggers the actual Query Decomposition component and the result of the query decomposer is passed to the Execution Engine. The Query Decomposer accepts a `TupleExpr` which is actually the algebraic tree of the query and identifies subexpressions that constitute basic group patterns. Then, the `QueryDecomposer` is called for each such basic pattern group. The basic pattern group is then replaced by the output of the Dynamic Programming algorithm. Moreover, certain physical operators have been introduced to the `TupleExpr` tree, for example, the alternative join operators `BindJoin`, `HashJoin` and the `SourceQuery` operator that performs the remote querying.

The “decomposed” `TupleExpr` is then passed to the execution engine via the `EvaluationStrategy` interface (see Section 4.1).
4. EXECUTION ENGINE

The Execution Engine component is responsible for evaluating an execution plan that is received by the Query Decomposition component (see Section 3). In a high-level, the Execution Engine component receives input from the Query Decomposition component and sends output to the Resource Discovery component. Moreover, it consults the Query Transformation Service for potential transformations of the query results.

In the following we discuss the most important implementation issues for the Execution Engine component.

4.1 Reactive Execution Engine

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 model [21] in order to realize this dataflow. In that model of execution, operators (i.e. 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 sequential systems, since data are generated only if and when needed and no computational effort is spent without contributing to the final result. However, in a parallel system, where data can be available at different rates, the iterator model is not suitable. Requests that are propagated down the stream can be blocked by other slower operations and potentially stall the overall query evaluation.

For example, consider an 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, it might happen 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 (Section 5). FedEx still uses the iterator model and circumvents this problem by a multithreaded 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 overheads [11].

SemaGrow on the other hand uses the reactive paradigm that 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 call-back 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, i.e. a consumer can send a request downstream 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.

4.2 Join Implementation

Sesame execution engine provides out-of-the box a non-optimized implementation of the nested loop join operator and a hash join implementation. We have also implemented two new join algorithms, namely the well-known Merge Join [22] and the Bind Join [13].
4.2.1.1 Bind Join

Bind Join [13] 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. However, the difference is that bind join can also work for remote queries since it substitutes the results of the outer relation as bindings to the 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 a grouping proposed in [23] 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 to the ChEBI dataset for the bind join processing of the P4 pattern \(^2\) using bindings from the outer result (P1\(\times\)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 in the ChEBI SPARQL endpoint:

```sparql
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 }
}
```

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, not only we can support SPARQL 1.0 endpoints, but also avoid a slower execution which can be a resulted from a possible inefficient implementation of the VALUES operator of an endpoint (which actually occurs in our case).

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 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. However, the batch size can be set as a configuration option, but we set 8 as the default value.

4.3 Implementation

The Execution Engine component is based on the Sesame default execution engine. Specifically, SemaGrow Execution Engine implements the interface EvaluationStrategy that is used to evaluate a TupleExpr given a set of bindings. The result of the evaluation is an iteration that can be consumed by the host application. Moreover, the

\(^2\) Specifically, we used the LS queries from the FedBench suite, cf Section 5.
Execution Engine also implements an additional method that can evaluate a TupleExpr given an iteration of bindings. This is particularly useful if we can evaluate efficiently an expression on a collection of binding sets at once. For example, Bind join that is discussed in Section 4.2.1.1 exploits this idea.

The EvaluationStrategy class handles only the local processing. When a remote query is encountered by the EvaluationStrategy the control flow is handed to the QueryExecutor evaluates the sub-query against a certain data sources. The QueryExecutor is responsible to convert an abstract TupleExpr into a valid SPARQL query, manage the connections to the data sources and intercepting the incoming results for metadata analysis by the Source Metadata Access component (see Section 1.3.1). Furthermore, the QueryExecutor also inserts provenance data (ie. the endpoint of the stream) to the resulting iteration for tracking purposes.

### 4.3.1 Query Results Interception

The Execution Engine component provides also a detailed query log to the Resource Discovery component for analysis. This query log is populated during the execution of the individual sub queries. The result sets returned by the data source are intercepted by the query log interceptor and written among other query metadata to the query log store. Typical query metadata is the data source endpoint, the total cardinality of the results and the total execution time.

<table>
<thead>
<tr>
<th>Implementation of Bind Join</th>
<th>Time in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNION construct</td>
<td>12</td>
</tr>
<tr>
<td>VALUES clause</td>
<td>218</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>Time in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6351</td>
</tr>
<tr>
<td>7</td>
<td>5955</td>
</tr>
<tr>
<td>8</td>
<td>5934</td>
</tr>
<tr>
<td>9</td>
<td>6025</td>
</tr>
<tr>
<td>10</td>
<td>6231</td>
</tr>
<tr>
<td>12</td>
<td>6427</td>
</tr>
<tr>
<td>15</td>
<td>7465</td>
</tr>
</tbody>
</table>
5. EVALUATION

In this section we evaluate SemaGrow and we compare it with other federation systems. For our experiments we used the FedBench suite and its LargRDF extension. 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. We present results for SemaGrow, FedX, and SPLENDID since the latter two systems have been shown to be the most performant [8]. The LargeRDF extension adds large-scale datasets to the FedBench datasets as well as queries over the overall FedBench/LargeRDF federation.

5.1 Experimental Setup

FedBench [24] 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 3 for the query characteristics).

In the evaluation setting\(^3\) 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 is generated by extracting statistics directly from the actual data. For every experiment, all queries are executed 5 times following a single warm-up run.

5.2 Query Execution

Table 5 shows the total query processing time (average) for both query collections, while Table 6 shows only the query execution times.

SemaGrow outperforms SPLENDID in most of the queries. The reason for our small execution time is due to the efficient query execution engine of SemaGrow. For example, notice the difference in query execution time (Table 6) for queries CD6, LS5, LS6 and LS7, where the execution plans are the same for both SemaGrow and SPLENDID.

With the exception of a vast difference in the execution time of LS6 query and some smaller differences in LS3, LS5 and LS7, both FedX and SemaGrow exhibit similar behaviour. 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.

---

\(^3\) The complete configuration and code of our experimental setup can be retrieved at https://bitbucket.org/dataengineering/semagrow-fedbench.
Table 3: FedBench Dataset Characteristics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of triples (millions)</th>
<th>Number of subjects (thousands)</th>
<th>Number of Predicates</th>
<th>Number of objects (thousands)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBpedia subset</td>
<td>43.60</td>
<td>9500</td>
<td>1063</td>
<td>13600</td>
</tr>
<tr>
<td>GeoNames</td>
<td>108.00</td>
<td>7480</td>
<td>26</td>
<td>35800</td>
</tr>
<tr>
<td>LinkedMDB</td>
<td>6.15</td>
<td>694</td>
<td>222</td>
<td>2.050</td>
</tr>
<tr>
<td>Jamendo</td>
<td>1.05</td>
<td>336</td>
<td>26</td>
<td>441</td>
</tr>
<tr>
<td>NY Times</td>
<td>0.34</td>
<td>22</td>
<td>36</td>
<td>192</td>
</tr>
<tr>
<td>SW Dog Food</td>
<td>0.10</td>
<td>12</td>
<td>118</td>
<td>38</td>
</tr>
<tr>
<td>KEGG</td>
<td>1.09</td>
<td>34</td>
<td>21</td>
<td>939</td>
</tr>
<tr>
<td>ChEBI</td>
<td>7.33</td>
<td>51</td>
<td>28</td>
<td>772</td>
</tr>
<tr>
<td>Drugbank</td>
<td>0.77</td>
<td>20</td>
<td>119</td>
<td>276</td>
</tr>
</tbody>
</table>

Table 4 FedBench Query Characteristics

<table>
<thead>
<tr>
<th>Query</th>
<th>Type</th>
<th>Number of Triple Patterns in query</th>
<th>Number of result tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD1</td>
<td>Complex</td>
<td>3</td>
<td>90</td>
</tr>
<tr>
<td>CD2</td>
<td>Star</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>CD3</td>
<td>Chain-Star</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>CD4</td>
<td>Complex</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>CD5</td>
<td>Chain-Star</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>CD6</td>
<td>Chain-Star</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>CD7</td>
<td>Chain-Star</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>LS1</td>
<td>Complex</td>
<td>2</td>
<td>1159</td>
</tr>
<tr>
<td>LS2</td>
<td>Chain</td>
<td>3</td>
<td>333</td>
</tr>
<tr>
<td>LS3</td>
<td>Chain-Star</td>
<td>5</td>
<td>9054</td>
</tr>
<tr>
<td>LS4</td>
<td>Complex</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>LS5</td>
<td>Complex</td>
<td>6</td>
<td>393</td>
</tr>
<tr>
<td>LS6</td>
<td>Complex</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td>LS7</td>
<td>Complex</td>
<td>5</td>
<td>144</td>
</tr>
</tbody>
</table>
**Table 5 Comparison of the total query processing times**

<table>
<thead>
<tr>
<th>Query</th>
<th>FedX</th>
<th>SPLENDID</th>
<th>SemaGrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD1</td>
<td>23</td>
<td>76</td>
<td>18</td>
</tr>
<tr>
<td>CD2</td>
<td>14</td>
<td>34</td>
<td>12</td>
</tr>
<tr>
<td>CD3</td>
<td>32</td>
<td>81</td>
<td>38</td>
</tr>
<tr>
<td>CD4</td>
<td>35</td>
<td>43</td>
<td>50</td>
</tr>
<tr>
<td>CD5</td>
<td>31</td>
<td>58</td>
<td>87</td>
</tr>
<tr>
<td>CD6</td>
<td>400</td>
<td>8119</td>
<td>385</td>
</tr>
<tr>
<td>CD7</td>
<td>484</td>
<td>1644</td>
<td>404</td>
</tr>
<tr>
<td>LS1</td>
<td>36</td>
<td>54</td>
<td>63</td>
</tr>
<tr>
<td>LS2</td>
<td>31</td>
<td>189</td>
<td>36</td>
</tr>
<tr>
<td>LS3</td>
<td>3821</td>
<td>24468</td>
<td>2935</td>
</tr>
<tr>
<td>LS4</td>
<td>32</td>
<td>70</td>
<td>84</td>
</tr>
<tr>
<td>LS5</td>
<td>2287</td>
<td>295147</td>
<td>1061</td>
</tr>
<tr>
<td>LS6</td>
<td>95983</td>
<td>19670</td>
<td>107</td>
</tr>
<tr>
<td>LS7</td>
<td>1683</td>
<td>18634</td>
<td>1186</td>
</tr>
</tbody>
</table>

**Table 6 Comparison of the query execution times**

<table>
<thead>
<tr>
<th>Query</th>
<th>FedX</th>
<th>SPLENDID</th>
<th>SemaGrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD1</td>
<td>15</td>
<td>28</td>
<td>17</td>
</tr>
<tr>
<td>CD2</td>
<td>9</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>CD3</td>
<td>28</td>
<td>45</td>
<td>27</td>
</tr>
<tr>
<td>CD4</td>
<td>29</td>
<td>12</td>
<td>37</td>
</tr>
<tr>
<td>CD5</td>
<td>24</td>
<td>28</td>
<td>82</td>
</tr>
<tr>
<td>CD6</td>
<td>394</td>
<td>8079</td>
<td>380</td>
</tr>
<tr>
<td>CD7</td>
<td>480</td>
<td>1598</td>
<td>398</td>
</tr>
<tr>
<td>LS1</td>
<td>34</td>
<td>35</td>
<td>62</td>
</tr>
<tr>
<td>LS2</td>
<td>27</td>
<td>169</td>
<td>35</td>
</tr>
<tr>
<td>LS3</td>
<td>3817</td>
<td>24441</td>
<td>2927</td>
</tr>
<tr>
<td>LS4</td>
<td>27</td>
<td>31</td>
<td>19</td>
</tr>
<tr>
<td>LS5</td>
<td>2283</td>
<td>295095</td>
<td>1035</td>
</tr>
<tr>
<td>LS6</td>
<td>95977</td>
<td>19642</td>
<td>98</td>
</tr>
<tr>
<td>LS7</td>
<td>1678</td>
<td>18595</td>
<td>1182</td>
</tr>
</tbody>
</table>
5.3 Query Optimization

Table 7 shows the total query planning times for both query collections. FedEx planning times are smaller than those of SPLENDID, due to the dynamic programming optimization of SPLENDID, which is slower than the greedy optimizer of FedEx. 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 result to 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. FedEx and SemaGrow ask every source, while SPLENDID perform sameAs groupings. This heuristic works for most of FedBench queries, but in CD7 misses one answer.

An interesting case is query LS6, in which SemaGrow outperforms both SPLENDID and FedEx 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 FedEx 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, FedEx 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 (P11&P2) at drugBank dataset and P3 at KEGG dataset is equal to 100%. This decision of the FedEx 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.

<table>
<thead>
<tr>
<th>Query</th>
<th>FedEx</th>
<th>SPLENDID</th>
<th>SemaGrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD1</td>
<td>8</td>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>CD2</td>
<td>5</td>
<td>26</td>
<td>2</td>
</tr>
<tr>
<td>CD3</td>
<td>4</td>
<td>36</td>
<td>11</td>
</tr>
<tr>
<td>CD4</td>
<td>6</td>
<td>31</td>
<td>13</td>
</tr>
<tr>
<td>CD5</td>
<td>7</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td>CD6</td>
<td>6</td>
<td>40</td>
<td>5</td>
</tr>
<tr>
<td>CD7</td>
<td>4</td>
<td>46</td>
<td>6</td>
</tr>
<tr>
<td>LS1</td>
<td>2</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>LS2</td>
<td>4</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>LS3</td>
<td>4</td>
<td>27</td>
<td>8</td>
</tr>
<tr>
<td>LS4</td>
<td>5</td>
<td>39</td>
<td>65</td>
</tr>
<tr>
<td>LS5</td>
<td>4</td>
<td>52</td>
<td>26</td>
</tr>
<tr>
<td>LS6</td>
<td>6</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>LS7</td>
<td>5</td>
<td>39</td>
<td>4</td>
</tr>
</tbody>
</table>
5.4 Large RDF Benchmark

In addition to the well-known FedBench queries, in order to evaluate the performance of SemaGrow we also used queries from the LargeRDFBench extension to FedBench. LargeRDFBench contains 32 queries: 14 simple queries (which is the same queries as of FedBench), 10 complex queries (C), and 8 large data queries (B). We present results of the 10 complex queries for SemaGrow and FedX.

The datasets used by the LargeRDFBench queries are presented in Table 8. The statistics for the Complex Queries of the benchmark are presented in Table 9.

For every experiment, all queries are executed 5 times. For every system we measured the average of the total query processing time of all 5 runs, the average query execution time and the average decomposition time (Table 10). We also give the number of successful runs, since some queries do not terminate but end with a timeout or a run-time error. Table 11 gives more details about the failed runs, with timeout set to 20 minutes for each query.

### Table 8: LargeRDFBench Dataset Characteristics. The top part of the table gives the datasets that are identical to FedBench and the bottom part the datasets that are the LargeRDFBench extension

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of triples (millions)</th>
<th>Number of subjects (thousands)</th>
<th>Number of Predicates</th>
<th>Number of objects (thousands)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBpedia subset</td>
<td>43.60</td>
<td>9500</td>
<td>1063</td>
<td>13600</td>
</tr>
<tr>
<td>GeoNames</td>
<td>108.00</td>
<td>7480</td>
<td>26</td>
<td>35800</td>
</tr>
<tr>
<td>LinkedMDB</td>
<td>6.15</td>
<td>694</td>
<td>222</td>
<td>2.050</td>
</tr>
<tr>
<td>Jamendo</td>
<td>1.05</td>
<td>336</td>
<td>26</td>
<td>441</td>
</tr>
<tr>
<td>NY Times</td>
<td>0.34</td>
<td>22</td>
<td>36</td>
<td>192</td>
</tr>
<tr>
<td>SW Dog Food</td>
<td>0.10</td>
<td>12</td>
<td>118</td>
<td>38</td>
</tr>
<tr>
<td>KEGG</td>
<td>1.09</td>
<td>34</td>
<td>21</td>
<td>939</td>
</tr>
<tr>
<td>ChEBI</td>
<td>7.33</td>
<td>51</td>
<td>28</td>
<td>772</td>
</tr>
<tr>
<td>Drugbank</td>
<td>0.77</td>
<td>20</td>
<td>119</td>
<td>276</td>
</tr>
<tr>
<td>LinkedTCGA-M</td>
<td>415.00</td>
<td>83000</td>
<td>6</td>
<td>166000</td>
</tr>
<tr>
<td>LinkedTCGA-E</td>
<td>344.00</td>
<td>54400</td>
<td>7</td>
<td>84400</td>
</tr>
<tr>
<td>LinkedTCGA-A</td>
<td>35.00</td>
<td>5700</td>
<td>384</td>
<td>8300</td>
</tr>
<tr>
<td>Affymetrix</td>
<td>44.20</td>
<td>44200</td>
<td>105</td>
<td>13200</td>
</tr>
</tbody>
</table>

### Table 9 LargeRDF query characteristics

<table>
<thead>
<tr>
<th>Query</th>
<th>Type</th>
<th>#Triple Patterns</th>
<th>#Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Complex</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>C2</td>
<td>Complex</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>C3</td>
<td>Complex</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>C4</td>
<td>Complex</td>
<td>12</td>
<td>50</td>
</tr>
<tr>
<td>C5</td>
<td>Complex</td>
<td>8</td>
<td>500</td>
</tr>
<tr>
<td>C6</td>
<td>Complex</td>
<td>9</td>
<td>148</td>
</tr>
<tr>
<td>C7</td>
<td>Complex</td>
<td>9</td>
<td>112</td>
</tr>
<tr>
<td>C8</td>
<td>Complex</td>
<td>11</td>
<td>3067</td>
</tr>
<tr>
<td>C9</td>
<td>Complex</td>
<td>9</td>
<td>100</td>
</tr>
<tr>
<td>C10</td>
<td>Complex</td>
<td>10</td>
<td>102</td>
</tr>
</tbody>
</table>

---

4 Available at https://code.google.com/p/bigrdfbench
Table 10 Queries C3, C8 and C10

```sparql
#QUERY C3
SELECT DISTINCT ?artist ?name ?location ?anylocation
WHERE
{
  ?artist a mo:MusicArtist ;
    foaf:name ?name ;
    foaf:based_near ?location .
  ?location geonames:parentFeature ?locationName .
  ?locationName geonames:name ?anylocation .
  ?nytLocation owl:sameAs ?location. #7
  OPTIONAL
  {
    ?locationName geonames:name 'Islamic Republic of Afghanistan' . #8
  }
}

#QUERY C8
SELECT DISTINCT * WHERE
{
  ?paper swc:isPartOf iswc:proceedings . #1
  iswc:proceedings swrc:address ?proceedingAddress. #2
  ?paper swrc:author ?author . #3
  ?author swrc:affiliation ?affiliation . #4
  ?author rdfs:label ?fullnames . #5
  ?author foaf:based_near ?place. #6
  OPTIONAL
  {
    #7
    ?place dbpedia:populationDensity ?populationDensity . #8
    ?place dbpedia:governmentType ?governmentType . #9
    ?place dbpedia:language ?language . #10
    ?place dbpedia:leaderTitle ?leaderTitle. #11
  }
}
```
# QUERY C10

WHERE
{
?uri tcga:bcr_patient_barcode ?patient .  #1
?patient tcga:gender ?gender.          #2
?patient dbpedia:country ?country.      #3
?country dbpedia:populationDensity ?popDensity. #4
?patient tcga:bcr_drug_barcode ?drugbcr. #5
?drugbcr tcga:drug_name ?drugName.     #6
?drgBnkDrg drugbank:genericName ?drugName. #7
?drgBnkDrg drugbank:indication ?indication. #8
?drgBnkDrg drugbank:chemicalFormula ?formula. #9
?drgBnkDrg drugbank:keggCompoundId ?compound . #10
}

Table 11 Querying times for the LargeRDF queries

<table>
<thead>
<tr>
<th>Query</th>
<th>Successful runs</th>
<th>SemaGrow</th>
<th>FedEx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Planning</td>
<td>Execution</td>
<td>Total</td>
</tr>
<tr>
<td>C1</td>
<td>5</td>
<td>183</td>
<td>20378</td>
</tr>
<tr>
<td>C2</td>
<td>5</td>
<td>269</td>
<td>6171</td>
</tr>
<tr>
<td>C3</td>
<td>5</td>
<td>284</td>
<td>17656</td>
</tr>
<tr>
<td>C4</td>
<td>5</td>
<td>281</td>
<td>52668</td>
</tr>
<tr>
<td>C5</td>
<td>1</td>
<td>8215</td>
<td>407069</td>
</tr>
<tr>
<td>C6</td>
<td>5</td>
<td>871</td>
<td>50583</td>
</tr>
<tr>
<td>C7</td>
<td>5</td>
<td>114</td>
<td>478</td>
</tr>
<tr>
<td>C8</td>
<td>5</td>
<td>1477</td>
<td>22345</td>
</tr>
<tr>
<td>C9</td>
<td>3</td>
<td>1476</td>
<td>13219</td>
</tr>
<tr>
<td>C10</td>
<td>5</td>
<td>1885</td>
<td>2125</td>
</tr>
</tbody>
</table>
Notice that in C1 and C4 queries, 4 of 5 FedX runs resulted in timeout, while all corresponding SemaGrow runs terminated without any problem. Also, in C5 and C9 queries, while some SemaGrow runs resulted in timeout, there exists at least one which terminates successfully. In the case of FedX though, all corresponding runs failed to deliver any results due to timeouts or runtime errors.

SemaGrow outperforms FedX in all queries except C8. FedX optimizer in this query selects a better plan, which leads to a faster execution. Notice though that one of the five runs in this case resulted in runtime error.

In the cases that SemaGrow decomposer and FedX optimizer yield the same plan, the performance of SemaGrow is better. This happens in queries C1, C6 and C9, where the execution plans are the same. In the remaining queries (C2, C3, C5, C7 and C10), SemaGrow selects a slightly or greatly better plan. We will consider the execution plans for the queries C3 and C10.

The C3 query returns all Jamendo artists who are based in locations which appear in the news of the New York Times. All triple patterns are occurring in various sources, therefore each source query contains a single triple pattern for both systems. In this situation, the efficiency of an execution plan can be determined by the order in which the triple patterns are evaluated. Notice that patterns P1 and P7 are the basic patterns of the join operation (the Jamendo artists and the NYT news respectively), patterns P3 and P6 are used for the interlinking between the two datasets, and the remaining patterns are providing additional information (i.e. the name of the artist etc).

<table>
<thead>
<tr>
<th>Query</th>
<th>Timeout</th>
<th>SemaGrow</th>
<th>Total</th>
<th>Timeout</th>
<th>FedX</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>C2</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>C3</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>C4</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>C5</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>C6</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>C7</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>C8</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>C9</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>C10</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>
Figure 6 Execution plan of C3 produced by FedX

Figure 7 Execution plan of C3 produced by SemaGrow
A good planning strategy for this query could be the following: Firstly, the planner should make a decision on which of the P1 and P7 must be evaluated first. Then, the “interlinking” patterns P3 and P6 should be placed between P1 and P7. Lastly, the remaining patterns that provide additional information should be placed after P7, in order to minimize unnecessary intermediate results. SemaGrow cost estimator makes a good estimation of the joins (as shown in the figure), and as a result it pushes the evaluation of P7 early in the execution plan. On the other hand, FedEx, which uses a greedy approach, doesn’t make any rearrangement on the order of the patterns, and as a result leaves the evaluation of P7 in the end of the evaluation, which results in a much slower execution time.

Another example of a query that SemaGrow performs better than FedEx is the query C10. The SemaGrow plan is more efficient, since the first triple pattern returns fewer results that the first triple pattern of the corresponding FedEx plan, and the number of the intermediate bindings is kept relatively small during the evaluation of the query.

Query C8 asks for basic information about papers of the ISWC conference together with optional information about the location of the authors of the paper. In that query the related sources are the SWDF and Jamendo. However, due to the based_near predicate, LinkedMDB and Jamendo must be also checked for matching triples. Moreover, since the rdf:label is so common, both SemaGrow and FedEx assume that all data sources should be asked for matching triples. The actual query plans produced are depicted in Figure 9 and Figure 10. FedEx manages to create a more efficient plan, mainly because it groups patterns of the same source into a single query. On the other hand, SemaGrow decides to split the group into two separate sub-queries, and ask for P4 after joining with P5. SemaGrow erroneously estimates the selectivity of P5 as it expects all sources with different selectivities for P5 to contribute to the join result. Moreover, SemaGrow also estimates that an early join of P4 will increase the intermediate results making the cost of the joining P4 even greater. Thus, it prefers to perform the join of P5 earlier than P4.

In that case, metadata that describe join selectivities between triple patterns in different sources will be helpful. For example, if it is known that only some sources can contribute to the result (i.e. sources with non-zero join selectivity), then, selectivities of the remaining sources will not obscure the estimation of the join result. Such metadata is more detailed than the commonly-used VoID metadata used in these experiments for the sake of comparability; the more detailed Senvod data source description vocabulary developed in SemaGrow foresees selectivity (cf. D3.1.2). Senvod experiments will be reported in D4.4.

Figure 8: The execution plans of C10 produced by FedEx (left) and SemaGrow (right)
Figure 9 SemaGrow execution plan for query C8

Figure 10 FedEx execution plan for query C8
6. CONCLUSION

In this document we have described the techniques and algorithms employed for the implementation of the Query Decomposition and the Execution Engine component, essential components of the SemaGrow Stack.

The well-known Dynamic Programming algorithm yields in general quality execution plans. It has been adapted for the SemaGrow environment by defining a cost function based on the total work needed for computing the answer of a given query. Moreover, we have discussed the interconnections and data dependencies with the rest of the SemaGrow Stack modules. A new non-blocking data-driven reactive Execution Engine has been proposed and implemented. The execution engine can handle distributed querying and merging the partial results using various join implementations. We have conducted extensive experiments on the FedBench and Large RDF Benchmark suites in order to compare the performance of SemaGrow Stack with state of the art systems FedEx and SPLENDID. In most cases SemaGrow outperform both systems especially in cases where the queries are complex.

Future work within SemaGrow (WP4) will test the Stack on the datasets and queries from the SemaGrow use cases. This work will be reported in D4.4 (due M36).

Future research plans are to extend the query planning component with research produced in the relational database field regarding techniques for optimizing queries with outer joins [25] and user defined predicates in filters [26].

Moreover, since our execution engine operates in an asynchronous and non-blocking way we expect an even larger margin in performance when executing over remote sources with substantial differences in network throughput and latency. As a future plan we would like 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 time-outs or unavailable sources.
REFERENCES


