Distributed Querying to RDF Repositories using SPARQL-DQP

Carlos Buil-Aranda[1], Oscar Corcho[1]

[1] Ontology Engineering Group, Departamento de Inteligencia Artificial, Facultad de Informática, Universidad Politécnica de Madrid. Boadilla del Monte, Spain
{cbuil, ocorcho}@fi.upm.es

Abstract. One of the most common mechanisms to access RDF data is by means of SPARQL endpoints. However, when we need to execute queries that involve accessing RDF data from multiple SPARQL endpoints, neither the protocol nor the language provide any norms or guidelines about how to proceed. In this paper we propose an approach for federating queries to a set of SPARQL endpoints, using relational database distributed query processing techniques, and exploiting the added value of OGSA-DAI, a de facto standard for web-service based access to relational and XML databases.

Keywords: SPARQL, query federation, RDF, Distributed Query Processing.

1 Introduction

The amount of RDF data available in the Web of Data is constantly increasing. New RDF datasets are being published regularly, offering several means to be accessed, mainly including Linked Data-enabled URLs and SPARQL endpoints. The W3C Wiki[1] lists more than 40 SPARQL Endpoints and more than 70 RDF datasets and wrappers offering RDF data and there are even more that are not listed there. There is an estimation that there are more than 13 billion RDF triples available on the Web[2] .

Answering queries against the data coming from one single dataset may be enough in a large number of situations. However, there may be also an added value in the combination of data from different datasets. Let's imagine that we want to query DBPedia for those bands in the Madchester movement in the 1990s, together with additional information such as BBC programs related to those bands, and album information from MusicBrainz. In this scenario we need to query three SPARQL endpoints: DBpedia[3], BBC[4] and MusicBrainz[5] and join together these query results.

Several approaches to access distributed RDF datasets using SPARQL endpoints have been described in the literature. Some of them use follow-up queries to the

[1] http://esw.w3.org/topic/SparqlEndpoints
[2] http://esw.w3.org/topic/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics
[3] http://dbpedia.org/snorql
[4] http://api.talis.com/stores/bbc-backstage/services/sparql
[5] http://dbtune.org/musicbrainz/sparql

different endpoints (e.g. in our example, we would first query DBPedia to obtain the band names, then we may let users filter these results, then each of these results/URIs would be used in follow up queries to the BBC SPARQL endpoint, the results would be joined in the presentation layer, and so on), others are based on querying a central collection of datasets where all the data is stored in a single location[6], others use query mediation and federation systems (e.g. SemWIQ [11], DARQ [14], Networked Graphs [15]) and other more recent approaches follow an automated link traversal approach over the Web of Linked Data (e.g. Hartig and colleagues' proposal [8]).

We will now describe briefly some of these approaches: SemWIQ, DARQ, Networked Graphs and Hartig and colleagues'. We will not focus on those operating over partial or complete stored copies of existing datasets, since our assumption is that distributed datasets may change at their own pace and we are not interested in devising synchronization mechanisms, caches, etc., even if a pure distributed approach to querying will obviously have more performance constraints.

SemWIQ [11] is a mediator-wrapper based system, where heterogeneous data sources (available as CSV files, RDF datasets or relational databases) are accessed by a mediator through wrappers. Queries are expressed in SPARQL and consider OWL as the vocabulary for the RDF data. SemWIQ uses the Jena's SPARQL processor ARQ to generate query plans and it applies its own optimizers. These optimizers mainly consist in rules to move down filters or unary operators in the query plan, together with join reordering based on the application of an iterative dynamic programming algorithm. The system has a registry catalog that indicates where the sources to be queried are and the vocabulary to be used. Currently the system does not handle SPARQL endpoints but this is being updated at the time of writing this paper.

DARQ [14] is a SPARQL query federation system, which also extends the Jena's SPARQL processor ARQ. This extension requires attaching a configuration file to the SPARQL query, with information about the SPARQL endpoint, vocabulary and statistics. DARQ applies logical and physical optimizations, focused on using rules for rewriting the original query before query planning (so as to merge basic graph patterns as soon as possible) and moving value constrains into subqueries to reduce the size of intermediate results. Unfortunately DARQ is no longer maintained.

Networked Graphs [15] also follows a SPARQL query federation approach (Distributed SPARQL[7]), based on the creation of graphs for representing views, content or transformations from other RDF graphs, and allowing the composition of sets of graphs to be queried in an integrated manner. The implementation considers optimizations such as the application of distributed semi-join optimization algorithms.

Finally, Hartig and colleagues [8] propose a more novel model that tries to exploit the navigational structure of the Web of Data, by incrementally executing queries over it. They discover new URIs from the initial SPARQL query and populate a local RDF repository, which is queried again for new answers to the initial query. Although this approach looks promising, the optimizations that are being applied for the time being are still naïve, and there are inherent limitations related to the fact that it is focused on exploiting the navigational nature of the Web of Data.

---

[6] http://lod.openlinksw.com/sparql

[7] http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IFI/AGStaab/Research/systeme/DistributedSPARQL

The aforementioned systems are the most relevant in terms of distributed RDF dataset querying. All of them apply some form of optimization, mainly based in join reordering algorithms and push down filters (SemWIQ and DARQ also implement cost based optimizations). In the case of SemWIQ and Networked Graphs, optimizations are inspired by existing work in the area of distributed query processing (DQP) in relational databases, adapting them to RDF databases.

Based on existing studies that show the relationship between SPARQL and SQL [4] and on the claims made in [6], where the authors state that SPARQL access and optimization techniques are still in their infancy, our proposal is to use relational database DQP techniques and SQL optimization techniques to generate and optimize query plans to be executed against RDF datasets available as SPARQL endpoints. Hence in this paper we describe a SPARQL query federation system based on the transformation of a subset of SPARQL queries[8] into their equivalent SQL queries, the extension of an existing relational database DQP system (OGSA-DQP [12]) to generate optimized query plans across distributed RDF datasets, and the use of the OGSA-DAI [2] framework for the robust execution of those queries and for managing direct and indirect access to datasets following the WS-DAI recommendation [3]. In the indirect access model, the result of querying a data source is a new data source that will be incrementally containing the results of the query execution as they are being generated by the underlying query engine, available as a stream of data tuples (or triples, in the case of RDF-based querying).

Our approach is the first that combines relational database DQP techniques with the use of indirect access modes to data sources, which can be useful for the creation of complex data workflows, such as those generated in many e-Science applications.

This paper is structured as follows. Section 2 presents the background needed for a better understanding of this paper, describing previous work on relating SPARQL with the relational model, and some background on relational distributed query processing. Section 3 describes our solution, SparqlDQP, detailing the design decisions we took and a brief description of some implementation details. Section 4 presents an evaluation comparing some of the existing similar systems with our solution. Section 5 presents conclusions of this paper and our future workplan.

2 Background: SPARQL and Relational Algebra

2.1 SPARQL and Relational Algebra

As pointed out in the introduction, our approach to federated RDF querying is based on the application of relational database distribution and optimization techniques. Obviously, one of the first considerations to be taken into account before moving further with the description of how we have actually performed this distribution and optimization is to assess the validity of the approach, in terms of the preservation of the query language semantics in the transformation between SPARQL and SQL.

[8] Some SPARQL features like unions or filters have been left temporarily outside the implementation of the system, but they will be implemented in the short term.

Such analysis can be already found in the literature. In [1] the authors demonstrate that Relational Algebra under bag semantics and the W3C SPARQL specification have the same expressive power. The authors base this claim in the fact that Relational algebra has the same expressive power than non-recursive Datalog with negation (nr-Datalog). Together with the previous demonstration, they show that SPARQL with compositional semantics is equivalent to nr-Datalog with negation. In [13] the authors demonstrate that SPARQL and SPARQL with compositional semantics are equivalent to the W3C SPARQL specification. Therefore we can claim that using a relational algebra representation in our system we will not have a relevant impact in terms of losing expressivity.

Previously, in [4] the author also claims that SQL Logical Query Plans (LQPs) may be used to represent the most common SPARQL queries. These LQPs use equi-joins for grouping triple patterns and left outer joins for handling optionals in queries. As a result, most of the SPARQL queries can be actually transformed to SQL without losing any expressivity and preserving the query semantics. The author specifies as well a set of limitations and mismatches between SPARQL and Relational Algebra. These mismatches can be summarized as follows: the different behavior of Relational Algebra and SPARQL with unbound variables (SPARQL does not take into account null values - they are left in blank -, while the relational model specifies the null value), different join behavior with missing information (due to the previous problem of null values and blank nodes), nested optional problems (for the same reason) and different filter scope (in which FILTER may not affect the right triple pattern). These limitations will be addressed in further iterations of our solution, as specified in the future work section of this paper.

Finally in [13] the authors describe the semantics of SPARQL and its complexity, which is not addressed in the aforementioned references. They define the concept of well-designed patterns for SPARQL queries, which impose restrictions on how to write SPARQL queries in order to ensure a low complexity in their treatment and the possibility of applying optimizations in query plans[9]. Well-designed patterns are those where every variable occurring in the first part of a query occurs in both the first part of the pattern and in the last. For instance, an AND-FILTER-OPT pattern is well–designed if for every OPT in the pattern (...(A OPT B)...) if a variable occurs inside B and anywhere outside the OPT operator, then the variable also occurs inside A. Following this approach, we limit the queries to be handled in our system to those following well-designed patterns.

2.2 Distributed Query Processing

Once we have analyzed the relationship between SPARQL and relational algebra, we move into describing the most important components of a DQP system [9]. Figure 1 shows a generic architecture for a DQP system, which considers the following components: query parser, query rewriter, query optimizer, plan refinement component and query execution engine. The parser reads the query and transforms it into the system's internal representation. Next, the query rewriter creates a Logical

[9] The complexity of well-designed patterns is coNP-complete [13].

Query Plan from this internal representation. The query optimizer is in charge of applying different optimizations depending on the type of DQP system, the physical state of the system, which indices to use, which nodes to send the query to, etc. As a result, the query optimizer generates an optimized query plan that specifies how the query is going to be executed. This plan is refined and transformed into an executable plan by the plan refinement component. This plan will be executed by the query execution engine in each local node. The query execution engine provides generic implementations for every operator in the query plan. Finally, the catalog (or metadata) component stores information about the databases (schema, tables, views or physical information about it), which can be used during parsing, query rewriting and query optimization.

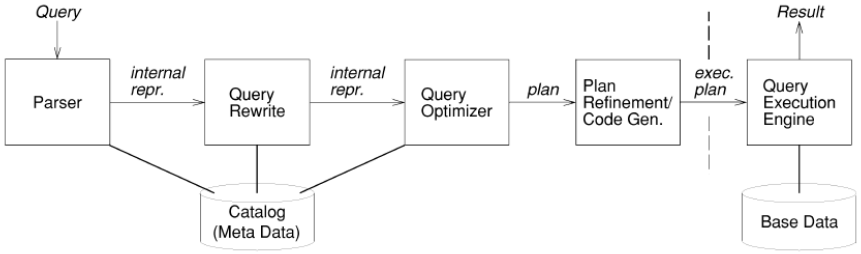

Figure 1: Most common phases of query processing.

This is a generic architecture, which can be adapted depending on the types of data sources that are handled, on the type of metadata available, etc. One of the most important elements in this architecture, since it heavily influences the quality of the DQP system, is the query optimizer. Some of the most common optimizations that may be performed by this component are the following:

- *Cost estimation*. These optimizers may use cost estimations based on resource consumption or on response time. The classic cost model estimates the cost of every individual operator of the plan and sums the cost of all the operators that are in the LQP. The cost of a plan is then defined as the resource consumption of the operator and the total cost of the plan is the cost of executing all operators in the LQP. A problem of this cost model is that it does not take into account intra-query parallelism. Another cost estimation method is the response time model. In this cost model the response between data nodes is taken into account. The model considers if a query can be parallelized in two different data nodes, taking into account the cost of the communications in the network.
- *Plan enumeration with dynamic programming*. Iterative dynamic programming algorithms build complex subplans from simpler plans. In each iteration, the algorithm selects the plan with minimal cost and updates the existing plan list. The result is the plan with minimal cost. The main drawbacks of this algorithm are related to its computational complexity (which is exponential in terms of time and space) and to its dependence on the quality of the cost function used to calculate the cost of each operation.
- *Site selection*. Every site has annotations that indicate where the operator is to be executed, and operators can be executed either at the client side or at the server

side. Based on this information the optimizer chooses the best operator/site to execute each part of the query plan.

- *Two step optimization*. This technique is based on two optimization cycles, done at compile and execution time. At compile time the initial query plan is generated, specifying joins, selections, projections and other operators. At the time of actual execution, the query plan is optimized again using annotations available at each data source site, taking into account their current status.

3 A SQL-based distributed query processor for SPARQL

In this section we describe our approach for executing SPARQL queries over distributed RDF datasets, based on the theoretical results discussed in section 2.1 and the usual DQP architecture described in section 2.2. We describe first our simple extension to SPARQL (SPARQL-D), so as to support distributed dataset querying. Then we describe our approach to generating basic query plans, the optimizations selected and the execution of the final and optimized data workflow, focusing on the most relevant components from Figure 1. For an easier understanding of our approach we show the walkthrough of a sample SPARQL-D query in our system.

3.1 SPARQL extension for distributed data querying

One of the first issues that we have had to tackle in our approach is the identification of the datasets to which the SPARQL queries will be sent, which is basic for query partitioning. Different systems follow different approaches to specify where triples may be coming from in this distributed setting. Some of them extend SPARQL and use configuration files to describe the RDF datasets to access each namespace, others use a pure Linked Data approach, considering that URIs should be dereferenceable, and others use a registry of data sources with a summary of their content so that queries can be partitioned adequately taking into account this information, which may be also available in the voID[10] data attached to the dataset.

We have decided to follow an approach similar to the one proposed in DARQ: extend the SPARQL language to allow specifying the source of each namespace. This extension (SPARQL-D) consists in allowing several FROM clauses in the SPARQL query, where each of these FROM clauses identifies the RDF resources to be accessed. While this is a restricted approach, we consider that it is not too relevant for the time being for our approach, since the major contributions are on the query transformation and optimization steps. In the future we plan to provide more flexibility, considering the use of an ad-hoc registry of sources or a general-purpose search engine (e.g., Watson[11], Sindice[12], etc.) to locate the sources that can provide results for query parts, and considering that URIs belonging to a namespace may be coming from different data sources.

[10] http://rdfs.org/ns/void-guide
[11] http://watson.kmi.open.ac.uk/
[12] http://sindice.com/

We will use throughout the rest of this section the following query about music bands and their members. The query asks DBPedia for those bands that were involved in (i.e., whose genre is) the Madchester movement, together with the current members that they have, if any, according to the BBC SPARQL endpoint (which in fact replicates a large part of the information from DBPedia, although this is not relevant for the sake of presenting our approach). The query has two main parts: the first triple pattern retrieves all the bands that were involved in the Madchester movement. The second triple pattern is optional, and asks for the current members of the band. The results of both patterns are merged with a left outer join. Our SPARQL extensions are represented in bold font, including the datasets to access and the variables to bind from each of them.

```
PREFIX p: <http://dbpedia.org/property/>
SELECT ?dbpedia.band ?bbc.member
FROM dbpedia: <http://dbpedia.org/sparql>
FROM bbc: <http://api.talis.com/stores/bbc-backstage/services/sparql>
WHERE{
      ?dbpedia.band p:genre <http://dbpedia.org/resource/Madchester>.
      OPTIONAL {?dbpedia.band p:currentMembers ?bbc.member}}
```

3.2 SPARQL-D Distributed Query Processor

We will now describe the characteristics of each of the components of our distributed query processor, according to the set of components identified in Figure 1.

SPARQL-D parser. It is the module in charge of creating an Abstract Syntax Tree (AST) from the initial SPARQL-D query.

SPARQL-D Logical Query Plan (LQP) Builder. The SPARQL LQP builder receives the previous AST as an input and generates an operator representing the SPARQL-D LQP (as shown in Figure 2). A SPARQL-D logical query plan is a directed graph whose nodes are a mix of relational and SPARQL operators. The processing of the query is done in two stages: first we process the prologue section of the SPARQL query, obtaining the SPARQL prefixes and the variables to be retrieved from the RDF repository, as specified in the FROM list; then we process the WHERE clause, considering two major blocks: graph-matching triples and OPTIONAL clauses. Solution modifiers like DISTINCT are also processed here. However, in our current implementation we leave out FILTER clauses.

The processing of the WHERE clause is based on the equivalence between SPARQL and SQL described in [4]: any two triples are translated as equi-joins if they share a variable, and OPTIONAL clauses are treated as SQL left outer joins. Besides, we apply the well-designed pattern concept [13] to OPTIONAL clauses.

For the special case of blank nodes and results without any value we assign the value “BlankNode” (which is adequately escaped in case that this specific string appears as a normal result of a query). Attributes that do not contain any value will have this special value, so that it will be possible to make joins with them. Otherwise errors and problems like the ones commented in the background section may arise.

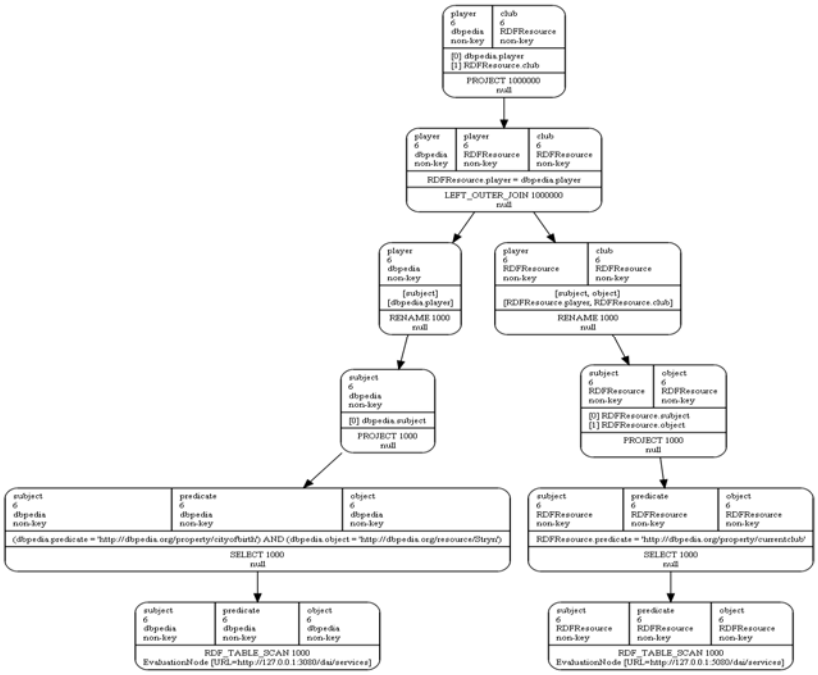

Figure 2: Generated LQP from our sample query

SPARQL-D Query Optimizer. The SPARQL-D query optimizer receives the previous LQP and generates an optimized query plan. As we described in the background section, the majority of optimizers are based on relational database information, such as the schema of the underlying database or the estimated number of tuples that the query will retrieve. In RDF datasets the schema can be always considered conceptually the same (subject, predicate and object), although different implementations have varying schemas. Therefore, optimization-wise it is more important to know how many properties of a certain type exist between subjects and objects, or the number of instances of certain concepts. This helps determining the cost of a specific SPARQL query, which can be measured as the estimated number of RDF triples that will be retrieved from each of the RDF datasets to be accessed. Another consideration related to the number of triples retrieved from each RDF dataset is the cost of joining them. In this case it is possible to apply cost based algorithms and other SQL optimizations, since the operators are the same. Real or estimated cost plans that select the operators with the minimal cost will be applied.

In our approach we apply some of the default optimizations available in our underlying query infrastructure: OGSA-DQP. These are heuristic based optimisations, cost based optimizations (using two step optimizations and mixing them with other cost-based optimizations) and those based on pushing the select clauses as next to the data sources as possible.

Besides, we add a new optimizer: the RDFTableScanImplosion optimizer. As aforementioned, SPARQL-D queries are translated into LQPs, which represent the

original query using SQL operators. Normally, the first operator to be applied is the RDF Scan, which retrieves all the triples from an RDF dataset (this would be in general very inefficient, since a data set may contain a huge amount of RDF triples); then the Select operator would be applied (which selects the triples from the triples initially retrieved from the Scan operator), and finally a Project operator would be applied (which contains the variable to be retrieved from the RDF triples). The RDFTableScanImplosion optimizer unifies these operations into a single one in order to perform at the RDF dataset the most restricting query. An example is the following: the SPARQL query contains a triple pattern `"?band p:genre <http://dbpedia.org/resource/Madchester>"`. The translation into a LQP is RDF Scan (`select * where {?s ?p ?o}`), Select (`p<-p:genre, o<-<http://dbpedia.org/resource/Madchester>`) and next the Project operator (`?p <-?band`). Using the optimization the query that is done to the triple store is `select * where {?band p:genre <http://dbpedia.org/resource/Madchester>. }`.

Besides the RDFTableScanImplosion optimizer, we also use a normalizer (which normalizes the LQP removing unnecessary operators) and a query partitioner (explained in the next section).

SPARQL-D Query Partitioner. The previous query plan is partitioned into subqueries addressed to the nodes where they will be executed. OGSA-DQP provides an algorithm in charge of partitioning the logical query plan according to the data nodes from which the data is retrieved. If a query plan contains a join or product of two data streams that are located on different data nodes, the partitioning algorithm detects this and transforms the LQP by inserting exchange operators that represent data transfers between two remote data nodes. The output from the partitioner is a set of partitions and the LQP where every operator is annotated with the partition to which it belongs. In our approach we have directly used the OGSA-DQP partitioner optimizer, since the LQP partitioning logic is the same in both situations.

3.3 SPARQL-DQP implementation

In this section we describe the implementation details of the SPARQL-DQP system. Besides the previously described work on query parsing, logical query plan generation, optimization and partitioning, following a classical approach, we also base our implementation in the use of Web Service-based access to data sources, as a result of our choice of implementation. A reason for selecting a WS-based approach for accessing RDF data sources is the availability of indirect access modes, which are not common in the current state of the art in SPARQL centralized and distributed querying. We will first provide some background on this type of access to data sources, and will then describe more details about our implementation.

3.3.1 WS-based access to RDF repositories

WS-DAI (Web Service Data Access and Integration) [3] is a recommendation of the Open Grid Forum that defines interfaces to access data resources as web services. The general WS-DAI specification has two extended realizations, one for accessing relational databases (WS-DAIR) and another for accessing XML databases (WS-DAIX), and work is being done in providing another extended realization for RDF data (WS-DAI-RDF [5]). The key elements of the WS-DAI specification are data services. A data service is a Web service that implements one or more of the DAIS-WG[13] specified interfaces to provide access to data resources (relational or XML databases, file systems, RDF datasets, etc.).

In WS-DAI, there are two access modes to data resources, as shown in Figure 3:

- **Direct access.** Data resources are accessed like a regular service: a request containing a query is sent to the data resource and the web service returns a rowset with the requested data.
- **Indirect access**. It implements a factory pattern for data requests. When a data resource is queried the data resource creates a new data resource where the query results will be populated incrementally when they are available.

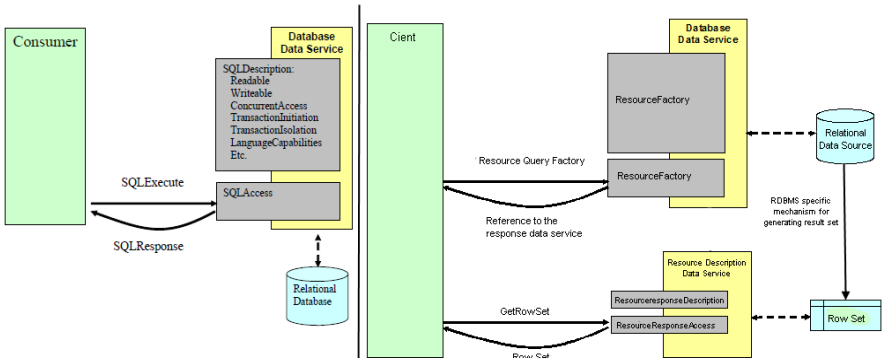

Figure 3: Direct and Indirect access to data resources respectively

OGSA-DAI [2] is a framework that was primarily intended as the WS-DAI reference implementation but which evolved differently, extending it. It executes data-centric workflows involving heterogeneous data resources for the purposes of data access, integration, transformation and delivery. OGSA-DAI is integrated in Apache Tomcat[14] and within the Globus Toolkit[15], and is used in OMII-UK[16], the UK e-Science platform.

OGSA-DAI relies on two key elements: data resources, which implement some of the WS-DAI methods, and activities, which are operations or named units of functionality (data goes in, something is done, data comes out) that can be combined to create workflows, by combining inputs and outputs from the activities that access the different resources. OGSA-DAI uses a tuple-based format for the internal

[13] https://forge.gridforum.org/projects/dais-wg
[14] http://tomcat.apache.org/
[15] http://www.globus.org/toolkit
[16] http://www.omii.ac.uk/

representation of data types. If a query returns two values, the internal representation is <value1, value2>. This is encoded as a data stream for a faster transfer of data between OGSA-DAI nodes.

OGSA-DQP [12] is the Distributed Query Processing extension of OGSA-DAI, which optimizes the access to distributed OGSA-DAI data resources.

3.3.2 SPARQL-DQP OGSA-DAI and OGSA-DQP implementation

From a high level point of view, SPARQL-DQP can be defined as an extension of OGSA-DQP that considers an additional query language: SPARQL. The design of SPARQL-DQP follows the idea of adding a new type of resource to the standard data resources provided by OGSA-DAI (relational databases, XML databases and file systems), and extending the parsers, planners, operators and optimizers that are handled by OGSA-DQP in order to handle this new type of resource.

Therefore our first extension to OGSA-DAI consists in adding a new type of data resource that accesses RDF datasets. This RDF data resource provides access to thse RDF stores that offer their data by means of SPARQL endpoints. This resource is configured with the URL of the SPARQL endpoint to which the query is addressed, together with other lifetime properties.

Queries are sent to this new type of RDF data resource by means of OGSA-DAI activities. The implementation of the RDF data resource sends the query to the corresponding SPARQL endpoint and waits for the results. These results can be directly returned to the requester or kept at the server wrapped as a new data resource, following direct and indirect access modes respectively. These results are provided as RDF data streams, using the internal data representation used by OGSA-DAI, what allows faster communication between nodes in the distributed settings.

Once it is possible to access RDF datasets using OGSA-DAI, it is also possible to extend OGSA-DQP to accept, optimize and distribute SPARQL queries across different data nodes. SPARQL-DQP extends OGSA-DQP with the new parsers, LQP builders, operators and optimizers described in the previous section, so as to read the query, create a basic query plan, optimize it, partition it and send it to the different nodes in which the different parts of the query will be processed. The extension follows the recommendations described in [7] and [9].

Once the query plan has been created, optimized and distributed across the nodes it is time for executing it. From the original planning a set of OGSA-DAI workflows is created. Each of these workflows represents a partition of the logical query plan created and these workflows are connected through their inputs and outputs to produce the result of the query. Once the workflow has been created the generated remote requests and local sub-workflows are executed and the results collected and returned by the activity.

4 Preliminary evaluation

In this section we present some preliminary comparisons of the performance of our tool and DARQ, which has been the only system that we have managed to work with[17]. The testbed that we have used cannot be considered as the definitive testbed, since it does not explore sufficiently the whole range of types of distributed SPARQL queries that may be considered in usual applications. However, we wanted to make a first test on the behavior of these systems. For these experiments, we used a PC with an Intel Core 2 Duo, 2.5GH and 3GB of RAM memory running on Windows 7. In terms of query response time, we executed the selected queries 10 times for each system, and obtained the median of all executions.

The first query is the one described in section 3 (bands from the Madchester movement and, optionally, their members). DARQ does not return the whole set of correct answers since it skips the optional part of the query, meanwhile SPARQL-DQP provides correct answers to this query. SPARQL-DQP requires 4 seconds to execute the query, while DARQ does it in 5 seconds.

Our next query involves the Bio2RDF[18] set of SPARQL endpoints. In this query we ask for the taxonomy, the gene and related elements of the protein Q8KKD2. To retrieve this information we have to query four SPARQL endpoints: Iproclass[19], GeneId[20], Taxon[21], and GenBank[22]. The query is the following:

```
SELECT ?iproclass.gene ?taxon.x ?genbank.genBank
FROM iproclass: <http://iproclass.bio2rdf.org/sparql>
FROM geneid: <http://geneid.bio2rdf.org/sparql>
FROM taxon: <http://taxon.bio2rdf.org/sparql-taxon>
FROM genbank: <http://genbank.bio2rdf.org/sparql-genebank>
WHERE
{
   <http://bio2rdf.org/iproclass:Q8KKD2>
               <http://bio2rdf.org/ns/iproclass#xGeneid>  ?iproclass.gene .
   ?iproclass.gene <http://bio2rdf.org/ns/bio2rdf#xTaxon>  ?geneid.taxon .
   ?geneid.taxon <http://bio2rdf.org/ontology/bio2rdf:xGI> ?taxon.x .
   ?taxon.x <http://bio2rdf.org/ontology/ncbi:featureID>  ?genbank.genBank .
}
```

The previous query has been subdivided and executed in three different stages, so as to test different configurations:

- Q1. It obtains only the information from the Iproclass and Geneid SPARQL endpoints.
- Q2. It obtains the information from the Iproclass, Geneid and Taxon endpoints.
- Q3. It obtains the information from the four endpoints.

[17] Only Networked Graphs could be also comparable. However, we have not been able to configure it properly to execute it, and hope to be able to do it in the future.

[18] http://bio2rdf.org/

[19] http://iproclass.bio2rdf.org/sparql

[20] http://geneid.bio2rdf.org/sparql

[21] http://taxon.bio2rdf.org/

[22] http://genbank.bio2rdf.org/sparql

	Q1	Q2	Q3
SPARQL-DQP	6s	6s	9s
DARQ	2s	6s	5s

Table 1: DARQ and SPARQL-DQP comparison.

As can be seen in Table 1, there are important execution time differences between both systems in queries 1 and 3, while they behave similarly for query 2. In terms of the logical query plans generated, they are similar, hence the main differences between both systems rely on the efficiency of the implementation, which is better in general in the case of DARQ since it does not have to deal with the overhead imposed by the marshalling and unmarshalling of Web service based interactions. In the future we will explore better these overheads and also the differences in terms of throughput, which can show better the advantages of the OGSA-DAI-based approach.

5 Conclusions and Future Work

We have presented a Distributed Query Processing (DQP) system, SPARQL-DQP, which is able to process SPARQL-D queries across distributed RDF datasets. We follow the approach of classical DQP and we base our solution on an existing relational framework like OGSA-DAI and OGSA-DQP. This allows us to reuse SQL optimizers already implemented in this framework, although we also create new ones that are specific for the types of queries that are handled in SPARQL and that can be attached to the optimization chain.

Besides, our choice of implementation also provides us with additional advantages that we will be exploiting in the future. The use and extension of OGSA-DAI provides us with the possibility of creating data workflows that make use of several data resources available in heterogeneous formats (e.g., relational databases, XML databases and RDF repositories). It is possible to create a workflow that access a SQL database, use these results to merge them with a SPARQL query to a RDF repository and send the results from this merge to another store or print them on the screen. Since the results of queries to RDF resources are provided in a tuple-based format, which is the basic data format handled in OGSA-DAI, it is also possible to integrate these results easily with queries performed to relational or XML databases.

Our preliminary evaluation has been based on the comparison of query results and query execution time between our system and DARQL. Other similar systems described in the introduction have not been compared because of different reasons: SemWIQ does not provide yet access to SPARQL endpoints, although it implement several optimization mechanisms that would be interesting to test; it has not ben possible to evaluate Networked Graphs due to configuration problems; and we have not tested Hartig and colleagues' approach since it is not available online yet. We understand that more comparisons and tests with other existing systems are necessary, and that an exhaustive testbed should be built as part of our future work, in order to

provide more fine-grained detail about the advantages and disadvantages of each approach.

Our future work will be devoted to the extension of the current SPARQL-DQP expressivity, so that it covers more aspects of the SPARQL query language (more query types like ASK, CONSTRUCT and DESCRIBE, result modifiers and SPARQL operators), and to the creation of additional optimizers specialized for the type of data that we are handling, while trying to understand better which other SQL-related optimizers we can apply in combination with those ones.

As it is stated in [4] and previously mentioned in the Preliminaries section, the transformation of SPARQL query plans to SQL query plans has certain limitations. The problems described with null/blank values, nested optionals and filters require a more detailed study.

Finally, our approach would clearly benefit from the existence of statistics about the RDF datasets to query, so that more optimized query plans can be created. It is important to know how many specific properties, classes or instances of a class an RDF repository contains. Statistics are widely used in the database world to perform optimizations over the logical query plans created due to the same reason. Two main approaches currently exist in this sense. The first one, RDFStats [10], crawls existing RDF repositories creating statistics and histograms about them. These statistics can be accessed with the RDFStas API. The second approach, examineRDF[23], produces statistics about large datasets: UniProt[24], DBpedia, CIA World Factbook[25] and the 230M triple BSBM[26] dataset. Our current implementation should be extended by considering some of these existing statistics in order to generate more optimized query plans.

8 Acknowledgments

This work has been performed in the context of the ADMIRE project (FP7 ICT-215024). We would like to thank the OGSA-DAI team (Ally, Bartek, Amy, Tilaye, Mario, Alastair, Mike and Elias) for the help provided during the system implementation, and to Claudio Gutiérrez and Renzo Angles for their collaboration in the understanding of the relationship between SPARQL and SQL. Finally we thank Alexander de León for his support on the generation of the testbed for bio2rdf.org.

9 References

[1] Angles R, Gutiérrez C (2008) The Expressive Power of SPARQL. In Proc. of the 7th International Semantic Web Conference (ISWC 2008). LNCS 5318:114-129.

[23] http://www.zaltys.net/examineRDF/
[24] http://www.uniprot.org/
[25] https://www.cia.gov/library/publications/the-world-factbook/
[26] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/

[2] Antonioletti M, Chue Hong NP, Hume AC, Jackson M, Karasavvas K, Krause A, Schopf JM, Atkinson MP, Dobrzelecki B, Illingworth M, McDonnell N, Parsons M, Theocharopoulos E (2007) OGSA-DAI 3.0 - The Whats and the Whys, Proceedings of the UK e-Science All Hands Meeting 2007, pp. 158-165
[3] Antonioletti M, Krause A, Paton NW, Eisenberg A, Laws S, Malaika S, Melton J, Pearson D. The WS-DAI family of specifications for web service data access and integration, ACM SIGMOD Record 35(1), March 2006
[4] Cyganiak, R. (2005) A relational algebra for SPARQL. Technical Report. HP Laboratories Bristol. HPL-2005-170
[5] Esteban-Gutiérrez M, Kojima I, Pahlevi SM, Corcho O, Gómez-Pérez A (2009) Accessing RDF(S) data resources in service-based Grid infrastructures. Concurrency and Computation: Practice and Experience 21(8):1029-1051
[6] Gray A J, Gray N, Ounis I (2009) Can RDB2RDF Tools Feasibly Expose Large Science Archives for Data Integration? In Proceedings of the 6th European Semantic Web Conference. LNCS 5554:491-505. Springer-Verlag.
[7] Haas LM, Freytag JC, Lohman GM, Pirahesh H (1989) Extensible Query Processing in Starburst. SIGMOD Conference 1989:377-388
[8] Hartig O, Bizer C, Freytag JC (2009) Executing SPARQL Queries over the Web of Linked Data. In Proceedings of the 8th International Semantic Web Conference (ISWC), Washington, DC, USA
[9] Kossmann D (2000) The state of the art in distributed query processing. ACM Comput. Surv. 32(4):422-469.
[10] Langegger A, Wöß W (2009) RDFStats - An Extensible RDF Statistics Generator and Library. DEXA 2009 Workshop on Web Semantics
[11] Langegger A, Wöß W, Blöchl M (2008) SemWIQ – Semantic Web Integrator and Query Engine. In Lecture Notes in Informatics, International Applications of Semantic Web Workshop (AST'08), Gesellschaft für Informatik, Bonn.
[12] Lynden S, Mukherjee A, Hume AC, Fernandes AAA, Paton NW, Sakellariou R, Watson P (2009) The design and implementation of OGSA-DQP: A service-based distributed query processor. Future Generation Computer Systems 25(3):224-236.
[13] Pérez J, Arenas M, Gutiérrez C (2006) Semantics and Complexity of SPARQL. In: Proc. of ISWC 2006. LNCS 4273:30–43. Springer-Verlag
[14] Quilitz B, Leser U (2008) Querying distributed RDF data sources with SPARQL. In: Proceedings of the 5th European Semantic Web Conference (ESWC2008). LNCS 5021:524-538. Springer-Verlag.
[15] Schenk S, Staab S (2008) Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the Web. In Proc. of the International World Wide Web Conference (WWW2008), pp. 585–594