View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Servicio de Coordinación de Bibliotecas de la Universidad Politécnica de Madrid

Federating Queries to RDF repositories

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. Currently large amounts of RDF data are being published in the Web. These data is commonly accessed by means of SPARQL endpoints. However to query a set of SPARQL endpoints new mechanisms are needed due to neither the SPARQL protocol nor the language provide any norms or guidelines about how to proceed. In this paper we present an approach for federating queries to a set of SPARQL endpoints, using relational database distributed query processing techniques and part of the WS-DAI specification for web-service based access to relational and XML databases.

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

1 Introduction

Currently there is a high increase of RDF data available in the Web. The list of RDF datasets accessible, among other ways, by Linked Data-enabled URLs and SPARQL endpoints is increasing every day. In the W3C Wiki[1] there are listed 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. In the same wiki there is an estimation that there exist more than 13 billion RDF triples available on the Web[2] .

This proliferation of RDF datasets brings a new problem: how to query them in a way we can obtain useful information. In some situations, to query a single dataset might be enough. However if we want to query a set of RDF datasets and link their data some difficulties arise. Let's imagine that we work in a genome project and we want to obtain the information related to a specific protein (orf5F[3]), together with additional information, such as the gene that codifies it, in which species the gene participates and some features of the taxonomy to which the species belong. In this scenario we need to query four SPARQL endpoints: Iproclass[4] (to obtain the protein and its information), GeneId[5] (gene information), Taxon[6] (taxonomy related

[1] http://esw.w3.org/topic/SparqlEndpoints
[2] http://esw.w3.org/topic/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics
[3] http://www.uniprot.org/uniprot/Q8KKD2.html
[4] http://iproclass.bio2rdf.org/sparql
[5] http://geneid.bio2rdf.org/sparql
[6] http://taxon.bio2rdf.org/sparql

information about species) and Genbank[7] (more gene related information). Finally we have to join together these query results.

In this paper we propose to federate SPARQL queries to a set of SPARQL endpoints. We base this proposal on existing studies that show the relationship between SPARQL and SQL [4]. We also base our proposal in the need of efficiently querying RDF datasets as it was proved in [6]. In there, the authors state that SPARQL access and optimization techniques are still in their infancy. We 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]. The indirect access model, described in the WS-DAI recommendation allows to leave the queries' results in the server which can later on be iteratively accessed. Using the WS-DAI resource properties is also possible to generate better statistics to improve the access generated by the underlying query engine.

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, SPARQL-DQP, detailing the design decisions we took and a brief description of some implementation details. Section 4 describes the existing solutions to SPARQL query federation. Section 5 presents an evaluation comparing some of the existing similar systems with our solution. Section 6 presents conclusions of this paper and our future workplan.

2 Background: SPARQL and Relational Algebra

2.1 SPARQL and Relational Algebra

We based our proposal to federated RDF querying in the application of existing data query processing techniques. These techniques come from highly used relational database systems which include distribution and optimization techniques. Previously to apply such techniques we have to verify the validity of our solution, in terms of the

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

[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.

preservation of the query language semantics in the transformation between SPARQL and SQL.

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.

In [4] the author claims that SQL Logical Query Plans (LQPs) may be used to represent the most common SPARQL queries. As a result, most of the SPARQL queries can be actually transformed to SQL without losing any expressivity and preserving the query semantics. Not all SPARQL queries can be translated directly to SQL LQPs. In [4] the author also describes some limitations and mismatches between SPARQL and Relational Algebra. These mismatches are:

- Relational Algebra and SPARQL behaves different with unbound variables. While SPARQL does not take into account null values - they are left in blank -, the relational model specifies the null value, which is ignored.
- join behaviour differs when there is missing information (due to the previous problem of null values and blank nodes),
- nested optional problems (for the same reason)
- 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. 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

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

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 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.

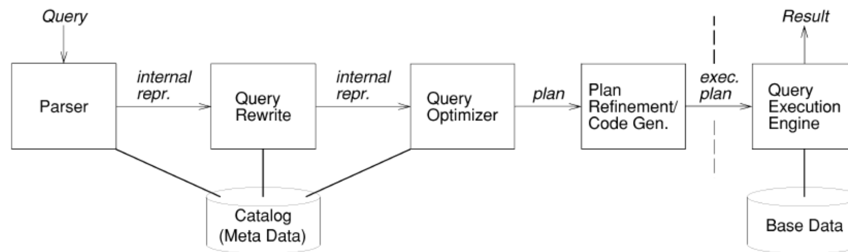

Figure 1: Most common phases of query processing.

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 *Cost estimation* which use cost estimations based on resource consumption or on response time estimating a cost model for each query. *Plan enumeration with dynamic programming*, in which 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. *Site selection,* in which 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.

These are the most common architectural components and optimizations that are applied in classical DQP systems. We base our system in the architecture described here and we plan to apply some of the described optimisers (cost based optimizations

and site selection at least), which, as it was described in the previous subsection, might need some adaptations to work with SPARQL.

3 A SQL-based distributed query processor for SPARQL

In this section we describe our solution 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 how 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 is the identification of the datasets to which the SPARQL queries will be sent, which is basic for query partitioning. Different possibilities are to specify where triples may be coming from in this distributed setting, extend SPARQL and use configuration files to describe the RDF datasets to access each namespace, use a pure Linked Data approach, considering that URIs should be dereferenceable, or use a registry of data sources with a summary of their content so that queries can be partitioned adequately taking into account this information.

In our approach we 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. We do not aim for a new extension to SPARQL, neither extending its syntax nor its semantics. Here we describe a simple means for querying a set of RDF datasets using a simple extension of the SPARQL query language. 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[10], Sindice[11], 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.

We will use throughout the rest of this section the following query about the orf5F protein and its related information. The query asks Iproclass for those genes that are involved in the codification of the protein, together with the species' taxonomy in which the gene is present, if any, according to the Geneid endpoint. The query has two main parts: the first triple pattern retrieves all the information about the protein.

[10] http://watson.kmi.open.ac.uk/
[11] http://sindice.com/

The second triple pattern is optional, and asks for the taxonomical information about the genes that participated in the codification of this protein, if any. 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 ipr: <http://bio2rdf.org/ns/iproclass>
PREFIX gn: <http://bio2rdf.org/ns/bio2rdf>
SELECT ?iproclass.protein ?geneid.taxon
FROM iproclass: <http://iproclass.bio2rdf.org/sparql>
FROM geneid: <http://geneid.bio2rdf.org/sparql>
WHERE{
      ?iprcolass.protein ipr:xGeneid ?iproclass.gene .
      OPTIONAL {?geneid.gene gn:xTaxon  ?geneid.taxon}}
```

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.

SPARQL-D Query Optimizer. The SPARQL-D query optimizer receives the previous LQP and generates an optimised query plan. As we described in the background section, the majority of optimisers 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 optimisations available in the underlying query management infrastructure: OGSA-DQP. We benefit from this infrastructure (service based distributed architecture and OGSA-DAI data services) using its parallel query distribution and the multi-phase query plan optimisation (physical and logical optimisations). Query evaluation in OGSA-DQP is based on the algorithm described in [16]. This algorithm describes two types of parallelisms in query evaluation, which are pipelined and partitioned parallelisms. Partitioned parallelism separates instances of an operator that exist in different nodes while pipelined parallelism processes sequentially different sets of data in the same node. Other optimisations applied by OGSA-DQP are heuristic based optimisations, cost based optimisations (using two step optimizations and mixing them with other cost-based optimisations) and those based on pushing the select clauses as next to the data sources as possible. In the SPARQL-DQP extension we apply the parallel and pipelined optimisations while other optimisations are left for future work due to the missing component in the SPARQL query federation state of the art (e.g., cost-based optimisations).

Besides, we add a new optimiser: the RDFTableScanImplosion optimiser. 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 optimiser 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 "<http://bio2rdf.org/iproclass:Q8KKD2> <http://bio2rdf.org/ns/iproclass#xGeneid> ?gene>". The translation into a LQP is RDF Scan (select * where {?s ?p ?o}), Select (p<-p: <http://bio2rdf.org/iproclass:Q8KKD2>, o<- ipr:xGeneid) and next the Project

operator (?o <- `?gene`). Using the optimisation the query that is done to the triple store is `select * where {<http://bio2rdf.org/iproclass:Q8KKD2> ipr:xGeneid ?gene>. }`.

Besides the RDFTableScanImplosion optimiser, 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 optimiser, 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, optimisation 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[12] 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 2:

[12] https://forge.gridforum.org/projects/dais-wg

- **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.

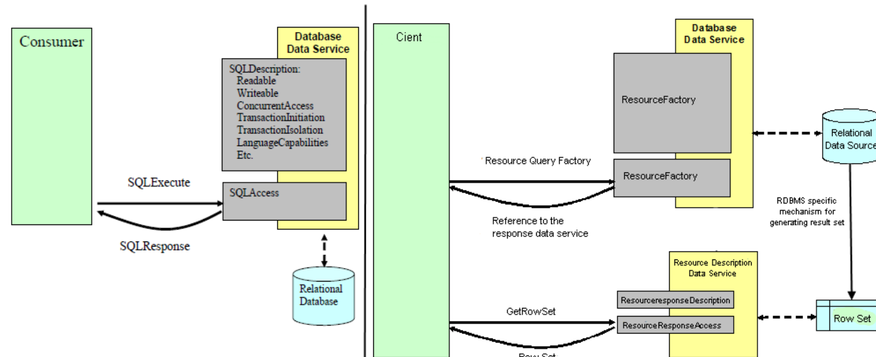

Figure 2: Direct and Indirect access to data resources respectively

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[13] and within the Globus Toolkit[14], and is used in OMII-UK[15], 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 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 optimises the access to distributed OGSA-DAI data resources. We take advantage of the WS-DAI used in OGSA-DAI/DQP indirect access by using its parallelisation system. This query parallelism system, as it was previously stated, is based in [16], which can hardly be done without a WS*/OGSA approach and its indirect access functionality. We also benefit from the security and resource management available in the WS-DAI specification plus the different set of OGSA-DAI activities for transforming and delivery of data.

---

[13] http://tomcat.apache.org/
[14] http://www.globus.org/toolkit
[15] http://www.omii.ac.uk/

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 these 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 Related Work

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 different endpoints (e.g. in our example, we would first query Iproclass endpoint to obtain the bacterium information, then we may let users filter these results, then each of these results/URIs would be used in follow up queries to the proteins 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[16], 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 catalogue 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. Other important drawback of DARQ is that it can only execute queries with bound predicates. Unfortunately DARQ is no longer maintained.

Networked Graphs [15] also follows a SPARQL query federation approach (Distributed SPARQL[17]), 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.

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

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

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

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.

6 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 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. Systems like RDFStats [10] or examineRDF[18] which produce statistics about large datasets should be used in our system. 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.

[18] http://www.zaltys.net/examineRDF/

7 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 Marcelo Arenas 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.

8 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.

[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. Proceedings of the 8th International Semantic Web Conference

[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.

[13] Pérez, J., Arenas, M., and Gutierrez, C. 2009. Semantics and complexity of SPARQL. ACM Trans. Database Syst. 34, 3 (Aug. 2009), 1-45

[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
[16] Goetz Graefe, Encapsulation of parallelism in the Volcano query processing system, Proceedings of the 1990 ACM SIGMOD international conference on Management of data, p.102-111,
[17] S. Schenk, C. Saathoff, A. Baumesberger, F. Jochum, A. Kleinen, S. Staab, and A. Scherp. Semaplorer - interactive semantic exploration of data and media based on a federated cloud infrastructure. In Billion Triple Challenge at ISWC, 2008.