Querying Distributed RDF Data Sources with SPARQL

Bastian Quilitz and Ulf Leser

Humboldt-Universität zu Berlin
{quilitz,leser}@informatik.hu-berlin.de

Abstract. Integrated access to multiple distributed and autonomous RDF data sources is a key challenge for many semantic web applications. As a reaction to this challenge, SPARQL, the W3C Recommendation for an RDF query language, supports querying of multiple RDF graphs. However, the current standard does not provide transparent query federation, which makes query formulation hard and lengthy. Furthermore, current implementations of SPARQL load all RDF graphs mentioned in a query to the local machine. This usually incurs a large overhead in network traffic, and sometimes is simply impossible for technical or legal reasons. To overcome these problems we present DARQ, an engine for federated SPARQL queries. DARQ provides transparent query access to multiple SPARQL services, i.e., it gives the user the impression to query one single RDF graph despite the real data being distributed on the web. A service description language enables the query engine to decompose a query into sub-queries, each of which can be answered by an individual service. DARQ also uses query rewriting and cost-based query optimization to speed-up query execution. Experiments show that these optimizations significantly improve query performance even when only a very limited amount of statistical information is available. DARQ is available under GPL License at *http://darq.sf.net/*.

1   Introduction

Many semantic web applications require the integration of data from distributed, autonomous data sources. Until recently it was rather difficult to access and query data in such a setting because there was no standard query language or interface. With SPARQL [1], a W3C Recommendation for an RDF query language and protocol, this situation has changed. It is now possible to make RDF data available through a standard interface and query it using a standard query language. The data does not need be stored in RDF but can be created on the fly, e.g. from a relational databases or other non-RDF data sources (see D2R Server[1] and SquirrelRDF[2]). We expect that more and more content provider will make their data available via a SPARQL endpoint. Nevertheless, it is still difficult to integrate data from multiple data sources. RDF data integration is

---

[1] D2R Server: *http://www.wiwiss.fu-berlin.de/suhl/bizer /d2r-server/*
[2] SquirrelRDF: *http://jena.sf.net/SquirrelRDF/*

often done by loading all data into a single repository and querying the merged data locally. In many cases this will not be feasible for legal or technical reasons. Often it will not be allowed to create copies of the whole data source due to copyright issues. Possible technical reasons are that local copies are not up-to-date if the data sources change frequently, that data sources are too big, or that the RDF instances are created on-the-fly from non-RDF data, like relational databases, web services, or even websites. This clearly shows the need for virtual integration of RDF datasets.

In this paper, we present DARQ[3], a query engine for federated SPARQL queries. It provides transparent query access to multiple, distributed endpoints as if querying a single RDF graph. We introduce *service descriptions* that describe the capabilities of SPARQL endpoints and a *query optimization algorithm* that builds a cost-effective query plan considering limitations on access patterns [2]. Sources with limited access patterns require some some variables in a query to be bound or fail to answer the query.

Related work. Data integration has been a research topic in the field of database systems for a long time. Systems providing a single interface to many underlying data sources are generally called *federated information systems* [3]. Solutions range from multi-database query languages (MDBQL) such as SchemaSQL [4] to federated databases [5] to mediator based information systems (MBIS) [6]. While multi-database query languages require that the user explicitly specifies the used data sources in the query MBIS hide the federation from the user by providing a single, unified schema. In this notation, SPARQL currently can be considered as a MDBQL for RDF allowing the user to specify the graphs to be used in the query. In contrast, DARQ offers source transparency to the user, but unlike MBIS it does not assume an integrated schema.

In [7] Stuckenschmidt et. al theoretically describe how to extend the Sesame RDF repository to support distributed SeRQL queries over multiple Sesame RDF repositories. They use a special index structure to determine the relevant sources for a query. To this end, they restricted themselves to path queries. In [8] the authors describe a system for SPARQL queries over multiple relational databases. To our best knowledge there exists no system that supports SPARQL query federation for multiple regular SPARQL endpoints. Also, none of the described systems uses service descriptions to declaratively describe the data sources nor do they support limitations on access patterns. A special characteristic of DARQ is that it strongly relies on standards and is compatible with any endpoint that supports the SPARQL standards. There is no other need for cooperation except of the support of the SPARQL protocol.

Research on query optimization for SPARQL includes query rewriting [9] or basic reordering of triple patterns based on their selectivity [10]. Optimization for queries on local repositories has also focused on the use of specialized indices for RDF or efficient storage in relational databases, e.g. [11,12]. However, none of the approaches targets SPARQL queries across multiple sources. There has been

---

[3] Distributed ARQ, as an extension to ARQ (*http://jena.sourceforge.net/ARQ/*)

```
SELECT ?name ?mbox WHERE    {
    ?x foaf:name ?name .
    ?x foaf:mbox ?mbox .
    FILTER regex(?name, "^Tim") && regex(?mbox, "w3c")
} ORDER BY ?name LIMIT 5
```

Listing 1.1. Example SPARQL Query

a lot of research on query optimization in the context of databases and federated information systems. An excellent overview of distributed query processing techniques can be found in [13]. In this paper, we show that existing techniques from relational systems, such as query rewriting and cost based optimization for join ordering can be adopted to federated SPARQL. We also propose a way to estimate the result sizes of SPARQL queries with only very few statistical information.

Structure of this paper. The rest of the paper is structured as follows. Section 2 gives a brief introduction to the SPARQL query language. In Section 3 we show the architecture of DARQ, give an introduce service descriptions and describe the used query planning and optimization algorithms we use in our current implementation. We show initial results of the evaluation of the system in Section 4 and conclude and discuss future directions in Section 5.

2   Preliminaries

Before we describe our work on federated queries we give a short introduction to the SPARQL query language and the operators of a SPARQL query that are considered for this report. For a more detailed introduction to RDF and SPARQL we refer the interested reader to [14,1,15]. In the following we use the definitions from the SPARQL Recommendation in [1].

A SPARQL query Q is defined as tuple $Q = (E, DS, R)$. Basis of SPARQL query is an algebra expression E that is evaluated with respect to a RDF graph in a dataset DS. The results of the matching process are processed according to the definitions of the *result form* R (SELECT, CONSTRUCT, DESCRIBE, ASK). The algebra expression E is build from different *graph patterns* and can also also include *solution modifiers*, such as PROJECTON, DISTINCT, LIMIT, or ORDER BY.

The simplest graph pattern defined for SPARQL is the *triple pattern*. A triple pattern t is similar to a RDF triple but allows the usage of variables for subject, predicate, and object:

$$t \in TP = (RDF\text{-}T \cup V) \times (I \cup V) \times (RDF\text{-}T \cup V)$$

with *RDF-T* being the set of RDF Terms (RDF Literals and Blank Nodes), I being a set of all IRIs, and V a set of all variables [1].

A *basic graph pattern BGP* is defined as a set of triple patterns $BGP = \{t_1..t_n\}$ with $t_1..t_n \in TP$. It matches a subgraph if all contained triple patterns

match. Basic graph patterns can be mixed with value constraints (FILTER) and other graph patterns. The evaluation of basic graph patterns and value constraints is order independent. This means, a structure of two basic graph patterns BGP_1 and BGP_2 separated by a constraint C can be transformed into one equivalent basic graph pattern followed by the constraint. We refer to a basic graph pattern followed by one or more constraints as *filtered basic graph pattern* ($FBGP$).

Example 1. Listing 1.1 shows a SPARQL query with one filtered basic graph pattern that retrieves the names and email addresses of persons whose name start with "Tim" and email address contains "w3c". The results are ordered by the name, the number of results is limited to five.

SPARQL furthermore defines other types of graph patterns such as GRAPH, UNION, or OPTIONAL. We omit these patterns here, because DARQ works on basic graph patterns as we will see in Section 3.2. Note however, that the engine is able to process all other patterns by distributing the FBGPs contained in these patterns and doing local post-processing. This means that the FBGPs in every of these pattens are handled separately, i.e. the scope for distribution and cost-based optimization is always limited to one FBGP. DARQ correctly handles the order-dependent OPTIONAL pattern, but may waste resources transferring unnecessary results when OPTIONAL is used to express negation as failure.

3   DARQ: Federated SPARQL Queries

To provide transparent query access to multiple data sources we adopt an architecture of mediator based information systems [6] as shown in Figure 1. The DARQ query engine has the role of the mediator component. Non-RDF data sources can be wrapped with tools such as D2R and SquirrelRDF. A DARQ query engine itself can work as SPARQL endpoint and may be integrated by another instance of DARQ. Data sources are described by service descriptions (see Section 3.1). The query engine uses this information for query planning and optimization. In contrast to MBIS the schema is not fixed and does not need to be specified, but is determined by the underlying data sources.

A query is processed in 4 stages:

1. **Parsing.** In the first stage the query string is parsed into a tree model of SPARQL. The DARQ query engine reuses the parser shipped with ARQ.
2. **Query Planning.** In the second stage the query engine decomposes the query and builds multiple sub-queries according to the information in the service descriptions, each of which can be answered by one known data source (see Section 3.2).
3. **Optimization.** In the third stage, the query optimizer takes the sub-queries and builds an optimized query execution plan (see Section 3.3).
4. **Query Execution.** In the fourth stage, the query execution plan is executed. The sub-queries are sent to the data sources and the results are integrated.
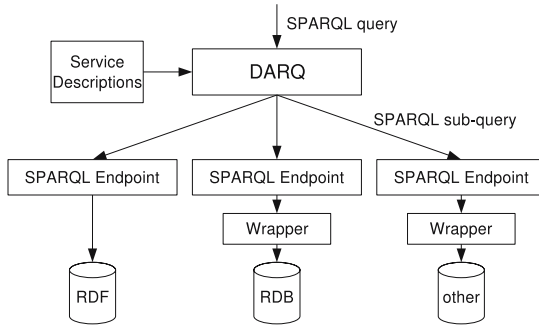
Fig. 1. DARQ - integration architecture

3.1   Service Descriptions

To find the relevant information sources for the different triples in a query and to decompose the query into sub-queries the query engine needs information about the data sources. To this end, we introduce *service descriptions* which provide a declarative description of the data available from an endpoint and allow the definition of limitations on access patterns. Furthermore, service descriptions can include statistical information used for query optimization. Service Descriptions are represented in RDF.

Data Description. A service description describes the data available from a data source in form of capabilities. Capabilities define what kind of triple patterns can be answered by the data source. The definition of capabilities is based on predicates. The capabilities of a data source D are a set C_D of tuples $c = (p, r) \in C_D$, where p is a predicate existing in D and r is a constraint on subjects and objects. This constraint is a regular SPARQL filter expression that enables a more precise source selection, e.g. we can express that a data source only stores data about specific types of resources. We denote the constraint as function $r(subject, object)$ with $r : (RDF\text{-}T \cup V) \times (RDF\text{-}T \cup V) \rightarrow \{true, false\}$. For example, the constraints can be used for horizontal partitioning. It is possible to define a constraint that says that a Service A can only answer queries for names starting with a letter from A to R, whereas another service can answer queries for names starting with a letter from Q to Z.

Limitations on Access Patterns. Some data sources have limitations on access patterns [2]. For example, a wrapper that transforms results from a web form into RDF may require some input values that can be entered into the form to compute the results. Another example is a wrapper for an LDAP server may require that the name of a person or their email address is always included in the query because the server owner does not allow other queries.

DARQ supports the definition of limitations on access patterns in the service descriptions in form of patterns that must be included in a query. Because

predicates must be bound we use them as basis for the pattern definition. Let L_D be a set of limitations on access patterns for data source D and $(S, O) \in L_D$ be one pattern with S and O being sets of predicates that must have bound subject (S) or bound objects (O).

Source D could contribute to the query answer of a query with graph pattern P if it satisfies at least one of the defined access patters for D. Let $bound(x)$ be a function that returns $false$ if x is a variable and $true$ otherwise. An access pattern (S, O) is satisfied if

$$(\forall p_s \in S \backslash O : \exists (s, p_s, o) \in P : bound(s))$$
$$\wedge (\forall p_o \in O \backslash S : \exists (s, p_o, o) \in P : bound(o))$$
$$\wedge (\forall p_b \in S \cap O : \exists (s, p_b, o) \in P : bound(s) \wedge bound(o))$$

Example 2. To come back to the example of the LDAP server, the service description in this example would contain two access patterns, (S_1, O_1) and (S_2, O_2), with $S_1 = S_2 = \emptyset$ and $O_1 = \{foaf : name\}$ $O_2 = \{foaf : mbox\}$

Statistical Information. Defining statistical information about the data available from a data source helps the query optimizer to find a cost-effective query execution plan. Service descriptions include the total number of triples N_s in data source D and optionally information for each capability $(p, r) \in C_D$: (1) The number of triples $n_D(p)$ with the predicate p in D, (2) the selectivity $ssel_D(p)$ of a triple pattern with predicate p if the subject is bound (default=$\frac{1}{n_D(p)}$), and (3) the selectivity $osel_D(p)$ of a triple pattern with predicate p if the object is bound (default=1). We deliberately use only these simple statistics because we expect every data source to be able to provide them, or at least rough estimations. More precise statistics such as histograms would be preferable but will not be available from many sources. Future work should explore what other statistics are required for more complex cost-models and how they can be estimated. In this context, aggregate functions, such as *count*, could be a valuable addition to future SPARQL version.

RDF Representation. Service Descriptions are represented in RDF. Listing 1.2 shows an example service description for a FOAF data source, e.g. an LDAP Server. The data source defined in the example can answer queries for foaf:name, foaf:mbox and foaf:weblog. Objects for a triple with predicate *foaf:name* must always start with a letter from A to R. In total it stores 112 triples. The data source has limitations on access patters, i.e. a query must at least contain a triple pattern with predicate *foaf:name* or *foaf:mbox* with a bound object. More detailed examples of service descriptions can be found at *http://darq.sf.net/*.

3.2   Query Planning

When querying multiple data sources it is necessary to decide which data source can contribute to answer a query. The process of finding relevant sources and

```
[]     a        sd:Service ;
    sd:capability [ sd:predicate foaf:name ;
                    sd:objectFilter "REGEX(?object,"^[A–R]")";
                    sd:triples 51 ] ;
    sd:capability [ sd:predicate foaf:mbox ;
                    sd:triples 51 ] ;
    sd:capability [ sd:predicate foaf:weblog ;
                    sd:triples 10 ] ;
    sd:totalTriples "112" ;
    sd:url   "EndpointURL" ;
    sd:requiredBindings [ sd:objectBinding foaf:name ] ;
    sd:requiredBindings [ sd:objectBinding foaf:mbox ] .
```

Listing 1.2. Example Service Description

feasible sub-queries is referred to as query planning. In this section we describe the query planning algorithm used by DARQ. Query planning is based on the information provided in the service descriptions. In the following let $R = \{(d_1, C_1), .., (d_n, C_n)\}$ be a set of data sources $d_1..d_n$ and their capabilities $C_1..C_n$, where $C_i = \{(p_{i,1}, r_{i,1})..(p_{i,m}, r_{i,m})\}$.

Source Selection. A SPARQL query contains one or more filtered basic graph patterns each containing the actual triple patterns. Query Planning is performed separately for each filtered basic graph pattern. The algorithm for finding the relevant data sources for a query simply matches all triple patterns against the capabilities of the data sources. The matching compares the predicate in a triple pattern with the predicate defined for a capability and evaluated the constraint for subject and object. Because matching is based on predicates, DARQ currently only supports queries with bound predicates.

Let BGP be a set of triple patterns in a filtered basic graph pattern. The result of the source selection is a set of data sources D_j for each triple pattern $t_j = (s_j, p_j, o_j) \in BGP$ with

$$D_j = \{d | (d, C) \in R \wedge \exists (p_j, r) \in C : r(s_j, o_j) = true\}$$

Building Sub-Queries. The results from source selection are used to build sub-queries that can be answered by the data sources. Sub-queries consist of one filtered basic graph pattern per data source. We represent a sub-query as triple (T, C, d), where T is a set of triple patterns, C is a set of value constraints and d is the data source that can answer the sub-query. Algorithm 1 shows how the sub-queries are generated. If a triple pattern matches exactly one data source $(D_i = \{d\})$ the triple will be added to the set of a sub-query for this data source. All triples in this set can later be sent to the data source in one sub-query. If a triple matches multiple data sources the triple must be sent individually to all matching data sources in separate sub-queries.

Example 3. Let data source A and B be two data sources with the capabilities (name,*true*) and (mbox,*true*). A stores the triple (a, name,"Tim"), B stores the triple (a ,mbox,"Tim@x.y"). The query shown in Listing 1.1 will return no results if sent to A and B with both triple patterns or the correct result if triple patterns are sent in separate sub-queries and the results are joined afterwards.

Algorithm 1. Sub-query generation

Require: $T = \{t_1, .., t_n\}$, // set of triple patterns
   $D = \{D_1, .., D_n\}$ // sets of data sources matching to the triple patterns
1: $queries = \emptyset$, $separateQueries = \emptyset$
2: **for** each $t_i \in T$ **do**
3:      **if** $D_i = \{d\}$ **then**
4:          $q = queries.getQuery(d)$
5:          **if** q not null **then**
6:              $q.T = q.T + t_i$
7:          **else**
8:              $queries = queries + (\{t_i\}, \{\}, d)$
9:          **end if**
10:      **else**
11:          **for** each $d_j \in D_i$ **do**
12:              $separateQueries = separateQueries + (\{t_i\}, \{\}, d_j)$
13:          **end for**
14:      **end if**
15: **end for**
16: **return** $queries \cup seperateQueries$ // Return all queries

3.3 Optimization

After query planning the query plan consists of multiple sub-queries. The task of the query optimizer is to build a feasible and cost-effective query execution plan considering limitations on the access patterns. To build the plan we use logical and physical query optimization.

Logical Optimization. Logical query optimization uses equalities of query expressions to transform a logical query plan into an equivalent query plan that is likely to be executed faster or with less costs. The current implementation of DARQ uses logical query optimization in two ways. First, we use rules based on the results in [15] to rewrite the original query before query planning so that basic graph patterns are merged whenever possible and variable are replaced by constants from filter expressions.

Example 4. Listing 1.3 shows the original query submitted by the user. There are two separate Basic Graph Patterns, each with one triple pattern. In the rewritten query that is shown in Listing 1.4 the two patterns are merged. Also, variables that occur in filters with an *equal* operator are substituted. In our example, `?name` is substituted by `"Tim"`.

```
SELECT ?mbox WHERE   {
  { ?x foaf:name ?name . }
  FILTER (?name = "Tim")
     && regex(?mbox, "w3c")
  { ?x foaf:mbox ?mbox . }
}
```

```
SELECT ?mbox WHERE   {
  ?x foaf:name "Tim"  .
  ?x foaf:mbox ?mbox .
  FILTER regex(?mbox, "w3c")
}
```

Listing 1.3. Query before rewriting   **Listing 1.4.** Query after rewriting

Second, we move possible value constraints into the sub-queries to reduce the size of intermediate results as early as possible. Let $Q = (T, C, d)$ be a sub-query and $FGP = (T', C')$ a filtered basic graph pattern. The value constraint C' can be moved to the sub-query if all variables in the constraint are also used in the triple patterns in the sub-query. Filters that contain variables from more than one sub-query and that cannot be split using a limited set of rules are applied locally inside the DARQ query engine.

Example 5. Listing 1.1 shows a query with a conjunctive filter on two attributes. Let us assume that the two triple patterns are split into two sub-queries for services A and B. In this case, the single filter cannot be moved into the sub-queries because one of the variables would be unbound. However, to benefit from filtering at the remote site the conjunction can be split into two filters `FILTER regex(?name, "^Tim")` and `FILTER regex(?mbox, "w3c")` that can then be moved into the sub-queries. If the optimizer is not able to split a filter using its limited set of rewriting rules, it will apply the filter locally, inside DARQ, as soon as all used variables are bound.

Physical Optimization. Physical query optimization has the goal to find the 'best' query execution plan among all possible plans and uses a cost model to compare different plans. In case of federated queries with distributed sources network latency and bandwidth have the highest influence on query execution time. Thus, the main goal in our system is to reduce the amount of transferred data and to reduce the number of transmissions, which will lead to less transfer costs and faster query execution. We use the expected result size as the cost factor of sub-queries.

We use iterative dynamic programming for optimization considering limitations on access patterns. Currently, we support two join implementations:

- **nested-loop join** ($\bowtie$) The nested-loop join is the simplest join implementation. For every binding in the outer relation, we scan the inner relation and add the bindings that match the join condition to the result set.
- **bind join** ($\bowtie_B$) The bind join was introduced in [16]. Basically it is a nested loop join where intermediate results from the outer relation are passed to the inner to be used as filter. This means that DARQ sends out the sub-query for the inner relation multiple times with the join variables bound. We use the bind join for data sources with limitations on access patterns. Furthermore, it can help to drastically reduce the transfer costs if the unbound query would return a large result set.

We calculate the result size of joins with

$$|R(q_1 \bowtie q_2)| = |R(q_1)| \, |R(q_2)| \, sel_{12}$$

where q_1 and q_2 are the joined query plan elements, i.e sub-query or join, $|R(q)|$ is the result size of q, and sel_{12} is a selectivity factor for the join attributes. For DARQ, we currently set $sel_{12} = 0.5$ because the current statistics in the service descriptions do not provide enough information for a better estimation.

The (transfer) costs of a nested loop join is estimated as

$$C(q_1 \bowtie q_2) = |R(q_1)|\, c_t + |R(q_2)|\, c_t + 2c_r$$

while the costs of a bind join are estimated as

$$C(q_1 \bowtie_B q_2) = |R(q_1)|\, c_t + |R(q_1)|\, c_r + |R(q_2')|\, c_t$$

with c_t and c_r being the transfer costs for one result tupel[4] and one query, respectively, and q_2' being the query with variables bound with values of a result tuple from q_1.

Query result size estimation The result size estimation for a sub-query is based on the statistics provided in the service descriptions. Currently, service descriptions include for each capability $(p, r) \in C_d$ of service d: (1) the number of triples $n_d(p)$ with the predicate p in data source d, (2) the average selectivity $ssel_d(p)$ if the subject is bound, and (3) the average selectivity $osel_d(p)$ if the object is bound. With this information we estimate the result size of a query with a single triple pattern (s, p, o) that is sent to a service d using the function $cost_d : TP \times V \to N$ with

$$costs_d((s, p, o), b) = \begin{cases} n_d(p) & \text{if } \neg bound(s, b) \wedge \neg bound(o, b), \\ n_d(p) * osel_d(p) & \text{if } \neg bound(s, b) \wedge bound(o, b), \\ n_d(p) * ssel_d(p) & \text{if } bound(s, b) \wedge \neg bound(o, b), \\ 0.5 & \text{if } bound(s, b) \wedge bound(o, b). \end{cases}$$

where b is a set of previously bound variables and $bound(x, b)$ is a function that returns *true* if x is bound given the bound variables in b and *false* otherwise.

Estimating the result size of a combination of two or more triple patterns is more complex. Note that adding a triple pattern to a query can restrict the result size or introduce new results because of a join. Adding more triple patterns with the same subject will not introduce new results, but rather reduce the result size. In contrast, adding triple pattern with another subject potentially increases the result size. Thus, we start with estimating the result size for all triple patterns with the same subject or subject variable. Let $T = \{t_1, ..., t_n\}$ be a set of triple patterns where $t_1, ...t_n$ all have the same subject. Triple patterns with a bound object restrict the possible solutions. We use the minimum function over all triple patters with a bound object to estimate an upper bound for the number of subjects. Note that this is different from the *attribute independence assumption* that is widely used in SQL query optimization [17]. Triple patterns with an unbound object can introduce new bindings for the used object variable. The overall result size for the set of triple patterns is the product of number of subjects and the result sizes of all triple patterns with unbound object. Using the cost function for a single triple pattern we estimate the result size of as follows:

---

[4] For simplicity, we currently disregard the specific tuple size.

$$costs_d(T, b) = \min_{v \in T_{bound}} (costs_d(v, b)) * \prod_{u \in T_{unbound}} costs_d(u, b)$$

with

$$T_{bound} = \{t | t = (s, p, o) \in T \wedge bound(o)\} \text{ and}$$
$$T_{unbound} = \{t | t = (s, p, o) \in T \wedge \neg bound(o)\}$$

Finally, we must combine the groups of triples with one subject to to compute the estimated result size for the complete sub-query. The result sizes of the single triple groups strongly depend of the already bound variables. Algorithm 2 builds groups of triple patterns with same subjects and then incrementally selects the group with the minimal result size considering the variables bound by the previously selected groups. We calculate the overall costs of the query as the product of the result sizes of all groups.

Algorithm 2. Result size estimation for a general basic graph pattern

---

Require: $T = \{t_1, .., t_n\}$ // basic graph pattern
1: $result = 1$, $bindings = \emptyset$, $groups = \{g_1, .., g_m\} = buildGroups(T)$
2: **while** $groups \neq \emptyset$ **do**
3:     $g = null$, $costs = positiveInfinity$
4:     **for** each $g_i \in groups$ **do**
5:         $c = costs(g_i, bindings)$
6:         **if** $c < costs$ **then**
7:             $g = g_i$, $costs = c$
8:         **end if**
9:     **end for**
10:    $groups = groups - \{g\}$, $bindings = bindings \cup var(g)$
11:    $result = result * costs$
12: **end while**
13: **return** $result$

---

4   Evaluation

In this section we evaluate the performance of the DARQ query engine. The prototype was implemented in Java as an extension to ARQ[5]. We used a subset of DBpedia[6]. DBpedia contains RDF information extracted from Wikipedia. The dataset is offered in different parts. The names of the parts we used can be found in the description column of Table 1(a).

The dataset has about 31.5 million triples in total. For our experiments we split the dataset into multiple parts located at different endpoints as shown in Table 1(a). To make sure that the endpoints are not a bottleneck in our setup we split all data over two Sun-Fire-880 machines (8x sparcv9 CPU, 1050Mhz, 16GB

---

[5] *http://jena.sf.net/ARQ/*
[6] *http://dbpedia.org* (Version 2.0)

RAM) running SunOS 5.10. The SPARQL endpoints were provided using Virtuoso Server 5.0.3[7] with an allowed memory usage of 8GB . Note that, although we use only two physical servers, there were five logical SPARQL endpoints. DARQ was running on Sun Java 1.6.0 on a Linux system with Intel Core Duo CPUs, 2.13 GHz and 4GB RAM. The machines were connected over a standard 100Mbit network connection.

Table 1. Overview on data sources and queries

<table>
<tr><td colspan="3">(a) data sources</td><td colspan="3">(b) queries</td></tr>
<tr><td>**No.**</td><td>**Description**</td><td>**#triples**</td><td>**No.**</td><td>**Used sources**</td><td>**#results**</td></tr>
<tr><td>S1</td><td>Articles</td><td>7.6M</td><td>Q1</td><td>S4, S5</td><td>452</td></tr>
<tr><td>S2</td><td>Categories</td><td>6.4M</td><td>Q2</td><td>S4, S5</td><td>452</td></tr>
<tr><td>S3</td><td>Yago</td><td>2M</td><td>Q3</td><td>S2, S4, S5</td><td>6</td></tr>
<tr><td>S4</td><td>Infoboxes</td><td>14.6M</td><td>Q4</td><td>S1, S3, S4, S5</td><td>1166</td></tr>
<tr><td>S5</td><td>Persons</td><td>0.6M</td><td></td><td></td><td></td></tr>
<tr><td></td><td>**Total**</td><td>**31.5M**</td><td></td><td></td><td></td></tr>
</table>

We run four example queries and evaluated the runtime with and without optimization. For queries without optimization we used the ARQ 1.5 default execution strategies without any changes, i.e. bind joins of all sub-queries in order of appearance. The queries can be found in Listings 1.5-1.8. The queries use different numbers of sources and have different result sizes. An overview is given in Table 1(b). For all queries we had a timeout of 10 minutes. Q1 and Q2 demonstrate the effect of pushing filters into the sub-queries. Q3 and Q4 are rather complex queries, involving three to four sources, but have very different result sizes. The results shown in the following are the average values over four runs.

```
/* Find all movies of actors
born in Paris */

SELECT ?p ?m WHERE {
?p dbpedia2:birthPlace :Paris.
?p foaf:name ?name .
?m dbpedia2:starring ?p.
}
```
Listing 1.5. Q1

```
/* Find all movies of actors
born in Paris */
SELECT ?p ?m WHERE {
?p foaf:name ?name .
?p dbpedia2:birthPlace ?paris.
?m dbpedia2:starring ?p.
FILTER (?paris=
<http://dbpedia.org/resource/
 Paris>)
}}
```
Listing 1.6. Q2

```
/*Find name,birthday and image of
german musicians born in Berlin*/
SELECT ?n ?b ?p ?img WHERE {
?p foaf:name ?n .
?p dbpedia2:birth ?b .
?p dbpedia2:birthPlace :Berlin .
?p skos:subject
        cat:German_musicians.
OPTIONAL { ?p foaf:img ?img }}
```
Listing 1.7. Q3

```
/* Find all Movies with actors
born in London with an image */
SELECT * WHERE { ?n rdf:type
 yago:MotionPictureFilm103789400.
?n dbpedia2:starring ?p.
?p dbpedia2:birthPlace :London.
?p foaf:name ?name .
?n rdfs:label ?label.
?n foaf:depiction ?img.
FILTER (LANG(?label) = 'en') .}
```
Listing 1.8. Q4

---

[7] http://virtuoso.openlinksw.com/

Figure 2(a) shows the query execution times. The experiments show that our optimizations significantly improve query evaluation performance. For query Q1 the execution times of optimized and unoptimized execution are almost the same. This is due to the fact that the query plans for both cases are the same and bind joins of all sub-queries in order of appearance is exact the right strategy. For queries Q2–Q4 the unoptimized queries took longer than 10 min to answer and timed out, whereas the execution time of the optimized queries is quiet reasonable. The optimized execution of Q1 and Q2 takes almost the same time because Q2 is rewritten into Q1.
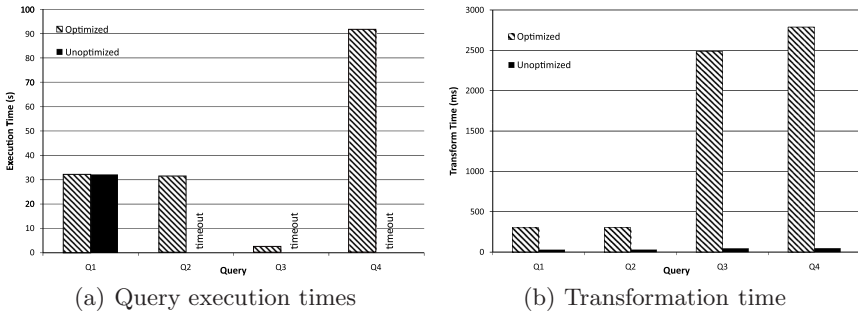

(a) Query execution times   (b) Transformation time

Fig. 2. Benchmark Results

Figure 2(b) shows the time needed for query planning and optimization (transformation time). We can see that the transformation times for optimized queries increase with query complexity from around 300 ms to 2800ms. Compared to this transformation times for unoptimized queries (query planning only) are negligible, around 30–40 ms. However, in comparison to the performance gains for query execution, the transformation times including optimization still remain very small.

Our evaluations show that even with a very limited amount of statistical information it is possible to generate query plans that perform relatively well. All queries where answered within less than one and a half minutes. Of course it would be possible for the user to write down triple patterns in exact the right order for ARQ, but this is in conflict with the declarative nature of SPARQL. Note that optimized queries in DARQ will be less performant if all the sub-queries are very unselective, e.g. contain no values for subject and object or a very unselective filter. In this case DARQ has few possibilities to improve performance by optimization.

5   Conclusion and Future Work

DARQ offers a single interface for querying multiple, distributed SPARQL endpoints and makes query federation transparent to the client. One key feature of

DARQ is that it solely relies on the SPARQL standard and therefore is compatible to any SPARQL endpoint implementing this standard. Using service descriptions provides a powerful way to dynamically add and remove endpoints to the query engine in a manner that is completely transparent to the user. To reduce execution costs we introduced basic query optimization for SPARQL queries. Our experiments show that the optimization algorithm can drastically improve query performance and allow distributed answering of SPARQL queries over distributed sources in reasonable time. Because the algorithm only relies on a very small amount of statistical information we expect that further improvements are possible using techniques as described in [16,13].

An important issue when dealing with data from multiple data sources are differences in the used vocabularies and the representation of information. In further work, we plan to work on mapping and translation rules between the vocabularies used by different SPARQL endpoints. Also, we will investigate generalizing the query patterns that can be handled and blank nodes and identity relationships across graphs.

References

1. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (January 2008), *http://www.w3.org/TR/rdf-sparql-query/*
2. Florescu, D., Levy, A., Manolescu, I., Suciu, D.: Query optimization in the presence of limited access patterns. In: International conference on Management of data (SIGMOD), pp. 311–322. ACM, New York (1999)
3. Busse, S., Kutsche, R.D., Leser, U., Weber, H.: Federated information systems: Concepts, terminology and architectures. Technical Report Forschungsberichte des Fachbereichs Informatik 99-9, Technische Universität Berlin (1999)
4. Lakshmanan, L.V.S., Sadri, F., Subramanian, I.N.: SchemaSQL - a language for interoperability in relational multi-database systems. In: Vijayaraman, T.M., Buchmann, A.P., Mohan, C., Sarda, N.L. (eds.) 22th International Conference on Very Large Data Bases (VLDB), Mumbai (Bombay), India, September 1996, pp. 239–250 (1996)
5. Sheth, A.P., Larson, J.A.: Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Comput. Surv. 22(3), 183–236 (1990)
6. Wiederhold, G.: Mediators in the architecture of future information systems. Computer 25(3), 38–49 (1992)
7. Stuckenschmidt, H., Vdovjak, R., Houben, G.-J., Broekstra, J.: Index structures and algorithms for querying distributed rdf repositories. In: WWW 2004 (2004)

8. Chen, H., Wang, Y., Wang, H., Mao, Y., Tang, J., Zhou, C., Yin, A., Wu, Z.: Towards a semantic web of relational databases: a practical semantic toolkit and an in-use case from traditional chinese medicine. In: 4th International Semantic Web Conference (ISWC), Athens, USA. LNCS, pp. 750–763. Springer, Heidelberg (2006)
9. Hartig, O., Heese, R.: The sparql query graph model for query optimization. In: 4th European Semantic Web Conference (ESWC), pp. 564–578 (2007)
10. Bernstein, A., Christoph Kiefer, M.S.: OptARQ: A SPARQL Optimization Approach based on Triple Pattern Selectivity Estimation. Technical Report ifi-2007.03, Department of Informatics, University of Zurich (2007)
11. Harth, A., Decker, S.: Optimized index structures for querying rdf from the web. In: Third Latin American Web Congress (LA-WEB), Washington, DC, USA, p. 71. IEEE Computer Society, Los Alamitos (2005)
12. Harris, S., Gibbins, N.: 3store: Efficient bulk rdf storage. In: PSSS - Practical and Scalable Semantic Systems (2003)
13. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. 32(4), 422–469 (2000)
14. Manola, F., Miller, E.: RDF Primer, W3C Recommendation (2004), http://www.w3.org/TR/rdf-primer/
15. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: 4th International Semantic Web Conference (ISWC), Athens, GA, USA, pp. 30–43 (November 2006)
16. Haas, L.M., Kossmann, D., Wimmers, E.L., Yang, J.: Optimizing queries across diverse data sources. In: 23rd Int. Conference on Very Large Data Bases (VLDB), pp. 276–285. Morgan Kaufmann Publishers Inc., San Francisco (1997)
17. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: International conference on Management of data (SIGMOD), pp. 23–34. ACM Press, New York (1979)