FedX: Optimization Techniques for Federated Query Processing on Linked Data

Conference Paper · January 2011

DOI: 10.1007/978-3-642-25073-6_38 · Source: DBLP

Andreas Schwarte

10 PUBLICATIONS **686** CITATIONS

SEE PROFILE

Peter Haase

metaphacts GmbH

168 PUBLICATIONS **5,684** CITATIONS

SEE PROFILE

Katja Hose

Aalborg University

154 PUBLICATIONS **3,215** CITATIONS

SEE PROFILE

Michael Schmidt

metaphacts GmbH

46 PUBLICATIONS **2,344** CITATIONS

SEE PROFILE

FedX: Optimization Techniques for Federated Query Processing on Linked Data

Andreas Schwarte[1], Peter Haase[1], Katja Hose[2], Ralf Schenkel[2], and Michael Schmidt[1]

[1]fluid Operations AG, Walldorf, Germany
[2]Max-Planck Institute for Informatics, Saarbrücken, Germany

Abstract. Motivated by the ongoing success of Linked Data and the growing amount of semantic data sources available on the Web, new challenges to query processing are emerging. Especially in distributed settings that require joining data provided by multiple sources, sophisticated optimization techniques are necessary for efficient query processing. We propose novel join processing and grouping techniques to minimize the number of remote requests, and develop an effective solution for source selection in the absence of preprocessed metadata. We present FedX, a practical framework that enables efficient SPARQL query processing on heterogeneous, *virtually integrated* Linked Data sources. In experiments, we demonstrate the practicability and efficiency of our framework on a set of real-world queries and data sources from the Linked Open Data cloud. With FedX we achieve a significant improvement in query performance over state-of-the-art federated query engines.

1 Introduction

In recent years, the Web more and more evolved from a Web of Documents to a Web of Data. This development started a few years ago, when the Linked Data principles [3] were formulated with the vision to create a globally connected data space. The goal to integrate semantically similar data by establishing links between related resources is especially pursued in the Linking Open Data initiative, a project that aims at connecting distributed RDF data on the Web. Currently, the Linked Open Data cloud comprises more than 200 datasets that are interlinked by RDF links, spanning various domains ranging from Life Sciences over Media to Cross Domain data.

Following the idea of Linked Data, there is an enormous potential for integrated querying over multiple distributed data sources. In order to join information provided by these different sources, efficient query processing strategies are required, the major challenge lying in the natural distribution of the data. A commonly used approach for query processing in this context is to integrate relevant data sets into a local, centralized warehouse. However, accounting for the decentralized structure of the Semantic Web, recently one can observe a paradigm shift towards federated approaches over the distributed data sources with the ultimate goal of *virtual integration* [8,13,14,16]. From the user perspective this means that data of multiple heterogeneous sources can be queried transparently as if residing in the same database.

While there are efficient solutions to query processing in the context of RDF for local, centralized repositories [5,15,23], research contributions and frameworks for federated query processing are still in the early stages. Available systems offer poor performance, do not support the full SPARQL standard, and/or require local preprocessed metadata and statistics. The problem we deal with in this work is to find optimization techniques that allow for efficient SPARQL query processing on federated Linked Data. Our goal is to provide optimizations that do not require any preprocessing – thus allowing for on-demand federation setup – and that are realizable using SPARQL 1.0 language features. Given that in a distributed setting communication costs induced by network latency and transfer of data are a considerable factor, we claim that reducing the number of (remote) requests that are necessary to answer a query must be minimized. Thus, join processing strategies as well as other sophisticated optimization approaches are needed to find an appropriate solution.

In summary, our contributions are:

- We propose novel optimization techniques for federated RDF query processing (Section 3), including new join processing strategies for query processing targeted at minimizing the number of requests sent to federation members, mechanisms to group triple patterns that can be exclusively evaluated at single endpoints, and an effective approach for source selection without the need of preprocessed metadata.
- We present FedX (Section 4), a practical framework allowing for *virtual integration* of heterogeneous Linked Open Data sources into a federation. Our novel sophisticated optimization techniques combined with effective variants of existing approaches constitute the FedX query processing engine and allow for efficient SPARQL query processing. Linked Data sources can be integrated into the federation on-demand without preprocessing.
- We evaluate our system (Section 5) using experiments with a set of real-world queries and data sources. We demonstrate the practicability and efficiency of our framework on the basis of real data from the Linked Open Data cloud and compare our performance to other competitive systems.

2 Related Work

Driven by the success of Linked Data, recently various solutions for federated query processing of heterogeneous RDF data sources have been discussed in the literature. A recent overview and analysis of federated data management and query optimization techniques is presented in [6]. [9,10] discuss the consumption of Linked Data from a database perspective. Federated query processing from a relational point of view has been studied in database research for a long time

[11,21]. Although the architectures and optimization approaches required in the context of RDF query processing have the same foundation, several problems arise due to differences in the data models.

Generally, in the context of Linked Data query processing, we can distinguish (a) bottom-up strategies that discover sources during query processing by following links between sources, and (b) top-down strategies that rely on upfront knowledge about relevant sources [7,12]. Several bottom-up techniques including active discovery of new sources based on Linked Data HTTP lookups have been proposed in the literature [8,13]. New relevant sources are discovered at runtime by following URIs of intermediate results using an iterator-based pipelining approach [8] or using the novel *Symmetric Index Hash Join* operator [13].

In our work, we focus on top-down strategies, where the relevant sources are known, hence guaranteeing sound and complete results over a *virtually integrated* data graph. In the research community various systems implementing these strategies have been proposed. *DARQ* [16] is a query engine allowing for SPARQL query processing on a number of (distributed) SPARQL endpoints. DARQ uses so-called *service descriptions* to summarize capabilities and statistics of data providers. This information is used in the optimization steps for source selection, i.e. sources for a triple pattern are determined based on predicate index lookups. Consequently, DARQ restricts query processing to queries in which all predicates are bound. A similar approach is employed in *SemWIQ* [14]. SemWIQ uses a concept-based approach and performs source selection based on type information of RDF entities available in a local dynamic catalog. SemWIQ requires that all subjects in a SPARQL query are variables. In addition, the type of each subject must be explicitly or implicitly known. In contrast to previous systems, our solution does not need any local preprocessed metadata since a different technique is employed for source selection. This makes it suitable for on-demand federation setup and practical query processing. Moreover, there is no limitation with respect to the SPARQL query language.

The W3C's SPARQL Working Group started to work on language extensions targeting the requirements and challenges arising in the context of distributed SPARQL processing. In a recent working draft[1], they propose the `SERVICE` operator, which allows for providing source information directly within the SPARQL query. In addition, `BINDING` clauses are introduced, which make it possible to efficiently communicate constraints to SPARQL endpoints. [2] provides a formal semantics for these features and presents a system called *SPARQL DQP*, which is capable of interpreting the new `SERVICE` keyword. SPARQL DQP does not need any preprocessed metadata, however, requires the endpoint to interpret SPARQL 1.1, which is typically not implemented in existing endpoints, as SPARQL 1.1 is currently available as a W3C working draft only.

In contrast to SPARQL DQP, FedX does not require any SPARQL 1.1 extensions and achieves automatic source selection over a set of defined sources (which can be dynamically extended) without additional input from the user. Thus, query formulation is more intuitive for the user, while query processing

[1] http://www.w3.org/TR/sparql11-federated-query/

in most cases is as efficient as with manual specification of service providers. In fact, this is not a restriction: when implementing the SPARQL 1.1 federation extensions in a future release, FedX can exploit the `SERVICE` keyword for improved source selection and use the `BINDING` clauses to further optimize queries.

Statistics can influence performance tremendously in a distributed setting. The VoID vocabulary (Vocabulary of Interlinked Datasets) [1] allows to specify various statistics and features of datasets in a uniform way at the endpoint. In addition, the SPARQL Working Group proposes the SPARQL 1.1 service descriptions[2], which allow discovery of basic information about the SPARQL service. Although these (remote) statistics are a good foundation for various optimizations, the expressiveness is limited to basic statistics, such as the number of triples or distinct subjects. Currently, we focus on optimizations without these statistics, yet we are planning to incorporate them in a future release.

3 Optimization Techniques for Federated Linked Data

In a federated setting with distributed data sources it is important to optimize the query in such a way that the number of intermediate requests is minimized, while still guaranteeing fast execution of the individual requests. While we support full SPARQL 1.0, our optimization techniques focus on conjunctive queries, namely basic graph patterns (BGPs). A BGP is a set of triple patterns, a triple pattern being a triple `(subject, predicate, object)` with variables in zero or more positions.

Given that the SPARQL semantics is compositional, our strategy is to apply the optimizations to all conjunctive subqueries independently (including, e.g., BGPs nested inside OPTIONAL clauses) to compute the intermediate result sets. Since we aim at a practical federation framework capable of on-demand configuration, we additionally focus on optimizations that do not require preprocessed metadata and that are realizable using SPARQL 1.0.

In practice, there are two basic options to evaluate a SPARQL query in a federated setting: either (1) all triple patterns are individually and completely evaluated against every endpoint in the federation and the query result is constructed locally at the server or (2) an engine evaluates the query iteratively pattern by pattern, i.e., starting with a single triple pattern and substituting mappings from the pattern in the subsequent evaluation step, thus evaluating the query in a nested loop join fashion (NLJ). The problem with (1) is that, in particular when evaluating queries containing non-selective triple patterns (such as e.g. `(?a,sameAs,?b)`), a large amount of potentially irrelevant data needs to be shipped from the endpoints to the server. Therefore, we opt for the second approach. The problem with (2), though, is that the NLJ approach causes many remote requests, in principle one for each join step. We show that, with careful optimization, we can minimize the number of join steps (e.g., by grouping triple patterns) and minimize the number of requests sent in the NLJ approach.

[2] http://www.w3.org/TR/sparql11-service-description/

3.1 Federated Query Processing Model

In our work, we focus on top-down strategies, where a set of user-configured sources is known at query time, hence guaranteeing sound and complete results over a virtually integrated data graph. Figure 1 depicts our federated query processing model, which closely follows the common workflow for general distributed query processing [11]. First, the SPARQL query is parsed and transformed into an internal representation (cf. Figure 2). Next, the relevant sources for each triple pattern are determined from the configured federation members using SPARQL `ASK` requests in conjunction with a local cache (Section 3.2). The remaining optimization steps include join order optimization (Section 3.3) as well as forming *exclusive groups* (Section 3.4). The outcome of the optimization step is the actual query execution plan. During query execution, subqueries are generated and evaluated at the relevant endpoints. The retrieved partial results are aggregated locally and used as input for the remaining operators. For iterative join processing the *bound joins* technique (Section 3.5) is applied to reduce the number of remote requests. Once all operators are executed, the final query result is returned to the client.

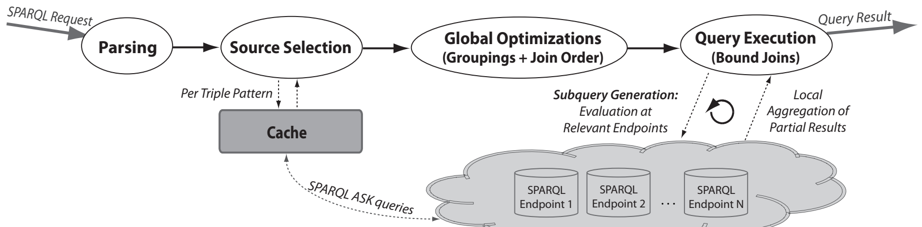
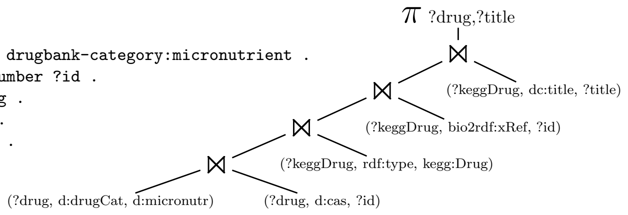


Fig. 1: Federated Query Processing Model

As a running example, Figure 2 depicts Life Science query 6 from our benchmark collections (Section 5) and illustrates the corresponding unoptimized query plan. The query computes all drugs in Drugbank[3] belonging to the category "Micronutrient" and joins computed information with corresponding drug names from the KEGG dataset[4]. A standard SPARQL query processing engine implementing the NLJ technique evaluates the first triple pattern in a single request, while the consecutive joins are performed in a nested loop fashion meaning that intermediate mappings of the left join argument are fed into the right join pattern one by one. Thus, the number of requests directly correlates with the number of intermediate results. In a federation, it must additionally be ensured that the endpoints appear *virtually integrated* in a combined RDF graph. This can in practice be achieved by sending each triple pattern to all federation members, using the union of partial results as input to the next operator.

[3] http://www4.wiwiss.fu-berlin.de/drugbank/
[4] http://kegg.bio2rdf.org/sparql

```
SELECT ?drug ?title WHERE {
   ?drug drugbank:drugCategory drugbank-category:micronutrient .
   ?drug drugbank:casRegistryNumber ?id .
   ?keggDrug rdf:type kegg:Drug .
   ?keggDrug bio2rdf:xRef ?id .
   ?keggDrug purl:title ?title .
}
```



π ?drug,?title
⋈
(?keggDrug, dc:title, ?title)
⋈
(?keggDrug, bio2rdf:xRef, ?id)
⋈
(?keggDrug, rdf:type, kegg:Drug)
⋈
(?drug, d:drugCat, d:micronutr)
(?drug, d:cas, ?id)

Fig. 2: Life Science Query 6 and the Corresponding Unoptimized Query Plan.

3.2 Source Selection

Triple patterns of a SPARQL query need to be evaluated only at those data sources that can contribute results. In order to identify these *relevant sources*, we use an effective technique, which does not require preprocessed metadata: before optimizing the query, we send SPARQL `ASK` queries for each triple pattern to the federation members and, based on the results, annotate each pattern in the query with its relevant source(s). Although this technique possibly overestimates the set of relevant data sources (e.g., for `(?s, rdf:type, ?o)` any data source will likely match during source selection, however, during join evaluation with actual mappings substituted for `?s` and `?o` there might not be results), in practical queries many triple patterns are specific to a single data source. Note also that FedX uses a cache to remember binary provenance information (i.e., whether source S is relevant/irrelevant for a triple pattern) in order to minimize the number of remote `ASK` queries.

Source selection has been discussed in previous works, e.g., [7, 14, 16]. However, existing approaches either require extensive local metadata or are too restrictive with respect to the SPARQL query language. In DARQ [16], for instance, relevant sources are determined using predicate lookups in so-called preprocessed *service descriptions*, hence requiring all predicates to be bound in a SPARQL query. The SPARQL 1.1 federation extension requires to specify sources in the query using the `SERVICE` keyword. In our approach we do not oblige the user to specify sources, while still offering efficient query computation.

3.3 Join Ordering

The join order determines the number of intermediate results and is thus a highly influential factor for query performance. For the federated setup, we propose a rule-based join optimizer, which orders a list of join arguments (i.e., triple patterns or groups of triple patterns) according to a heuristics-based cost estimation. Our algorithm uses a variation of the variable counting technique proposed in [22] and is depicted in Algorithm 1. Following an iterative approach it determines the argument with lowest cost from the remaining items (line 5-10) and appends it to the result list (line 13). For cost estimation (line 6) the number of free variables is counted considering already bound variables, i.e., the variables that are bound through a join argument that is already ordered in the result list.

Additionally, we apply a heuristic that prefers *exclusive groups* (c.f. Section 3.4) since these in many cases can be evaluated with the highest selectivity.

Algorithm 1 Join Order Optimization

```
order(joinargs: list of n join arguments) {
 1: $left \leftarrow joinargs$
 2: $joinvars \leftarrow \emptyset$
 3: for $i = 1$ to $n$ do
 4:     $mincost \leftarrow MAX\_VALUE$
 5:     for all $j \in left$ do
 6:         $cost \leftarrow$ estimateCost($j$, $joinvars$)
 7:         if $cost < mincost$ then
 8:             $arg \leftarrow j$
 9:             $mincost \leftarrow cost$
10:         end if
11:     end for
12:     $joinvars \leftarrow joinvars \cup$ vars($arg$))
13:     $result[i] \leftarrow arg$
14:     $left \leftarrow left - arg$
15: end for
16: return $result$ }
```

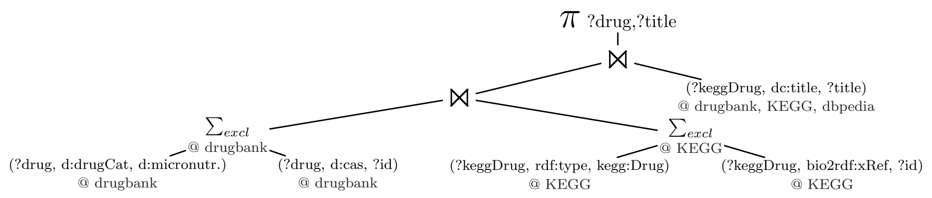
3.4 Exclusive Groups

High cost in federated query processing results from the local execution of joins at the server, in particular when joins are processed in a nested loop fashion. To minimize these costs, we introduce so-called *exclusive groups*, which play a central role in the FedX optimizer:

Definition 1. *Let $t_1 \ldots t_n$ be a set of triple patterns (corresponding to a conjunctive query), $S_1 \ldots S_n$ be distinct data sources, and S_t the set of relevant sources for triple pattern t. For $s \in \{S_1, \ldots, S_n\}$ we define $E_s := \{t \mid t \in \{t_1..t_n\}$ s.t. $S_t = \{S\}\ \}$ as the exclusive groups for source S, i.e. the triple patterns whose single relevant source is S.*

Exclusive groups with size ≥ 2 can be exploited for query optimization in a federated setting: instead of sending the triple patterns of such a group sequentially to the (single) relevant source, we can send them together (as a conjunctive query), thus executing them in a single subquery at the respective endpoint. Hence, for such groups only a single remote request is necessary, which typically leads to a considerably better performance because the amount of data to be transferred through the network and the number of requests often can be minimized by evaluating the subquery at the endpoint. This is because in many cases triple patterns that are not relevant for the final result are filtered directly at the endpoint, and on the other hand because the communication overhead of sending subqueries resulting from a nested loop join is avoided entirely. Correctness is guaranteed as no other data source can contribute to the group of triple patterns with further information.

In Figure 3, we illustrate the optimized query execution plan for our running example. During source selection, we annotate each triple pattern with its relevant sources and identify two exclusive groups, denoted as $\sum_{excl}$. For this query, we can reduce the number of local joins from four to just two.



π ?drug,?title
$\bowtie$
$\bowtie$
(?keggDrug, dc:title, ?title)
@ drugbank, KEGG, dbpedia
$\sum_{excl}$
@ drugbank
$\sum_{excl}$
@ KEGG
(?drug, d:drugCat, d:micronutr.)
@ drugbank
(?drug, d:cas, ?id)
@ drugbank
(?keggDrug, rdf:type, kegg:Drug)
@ KEGG
(?keggDrug, bio2rdf:xRef, ?id)
@ KEGG

Fig. 3: Execution Plan of Life Science Query 6 (Including Optimizations)

3.5 Bound Joins

By computing the joins in a block nested loop fashion, i.e., as a distributed semijoin, it is possible to reduce the number of requests by a factor equivalent to the size of a *block*, in the following referred to as an *input sequence*. The overall idea of this optimization is to group a set of mappings in a single subquery using SPARQL `UNION` constructs. This grouped subquery is then sent to the relevant data sources in a single remote request. Finally, some post-processing is applied locally to retain correctness. We propose the *bound join* technique and discuss the technical insights below.

In the following, we illustrate bound join processing for the triple pattern `(?S, name, ?O)`. For the example, assume that values for `?S` have been computed yielding the input sequence I:=`[?S=Person1,?S=Person2,?S=Person3]`. Further, let us assume that the database (where we evaluate the triple pattern) contains the RDF triples t_1=`(Person1, name, 'Peter')` and t_2=`(Person3, name, 'Andreas')`. When evaluating the query sequentially for the bindings in the input sequence I, we obtain the result depicted in Figure 4 a). While the naive NLJ approach requires distinct subqueries for each input mapping substituted into the triple pattern (e.g., `Person1, name, ?O`), our bound join solution allows to evaluate the complete input sequence in a single grouped subquery. The concrete subquery for this example is depicted in Figure 4 b).

a) Expected Result

?S	?O
Person1	Peter
Person3	Andreas

b) SPARQL subquery

```
SELECT ?O_1 ?O_2 ?O_3 WHERE {
  { Person1 name ?O_1 } UNION
  { Person2 name ?O_2 } UNION
  { Person3 name ?O_3} }
```

c) Subquery result

?O_1	?O_2	?O_3
Peter		
		Andreas

Fig. 4: Sample execution for bound join processing of `(?S, name, ?O)`

To guarantee correctness of the final result, we have to address three issues within the subquery: (1) we need to keep track of the original mappings, (2) possibly not all triple patterns yield results, and (3) the results of the subquery may be in arbitrary order. Our solution to this is an effective renaming technique: we annotate variable names in the subquery with the index of the respective mapping from the input sequence, e.g., for the first input mapping the constructed bound triple pattern is `(Person1, name, ?O_1)`. This renaming technique allows to implicitly identify correspondences between partial subquery results and input mappings in a post-processing step. Figure 4 c) depicts the results of this subquery evaluated against our sample database. In the post-processing step the final result is reconstructed by matching the retrieved partial results to the corresponding input mapping using the index annotation in the variable name, and then performing the inverse renaming. In our running example, for instance, variable `?O_1` is linked to the first position in the input sequence; therefore, the binding from `?O_1` to 'Peter' is combined with the first binding for ?S in the input sequence, yielding the first result in Figure 4 a). Bound join processing can be trivially generalized to an input sequence of N mappings. For a detailed formalization and a technical discussion we refer the interested reader to [19].

A similar technique is discussed in [6, 24]. The authors propose to use a distributed semijoin sending the buffered mappings as additional conditions in a SPARQL `FILTER` expression. Although the theory behind this technique is similar to *bound joins*, in practice it is far less efficient than using `UNION`s. We observed that for many available SPARQL endpoints the whole extension for a triple pattern is evaluated prior to applying the `FILTER` expressions. In the working draft for SPARQL 1.1 federation extensions, the W3C proposes the `BINDINGS` keyword to efficiently communicate constraints in the form of mappings to SPARQL endpoints, allowing to process sets of mappings corresponding to a block in a single subquery. We achieve a distributed semijoin without requiring this feature, using only SPARQL 1.0 language constructs. Clearly, our approach can easily be extended to utilize SPARQL 1.1 `BINDINGS` in the future.

4 FedX - Implementation

Having introduced various optimizations for distributed federated query processing on Linked Data, in this section we present FedX[5], a solution implementing the previously discussed techniques. FedX represents a practical solution for efficient federated query processing on heterogeneous, *virtually integrated* Linked Data sources. The practicability of FedX has been demonstrated in various scenarios in the Information Workbench[6] [20].

4.1 Architecture & Design

FedX has been developed to provide an efficient solution for distributed query processing on Linked Data. It is implemented in Java and extends the Sesame framework with a federation layer. FedX is incorporated into Sesame as `SAIL` (Storage and Inference Layer), which is Sesame's mechanism for allowing seamless integration of standard and customized RDF repositories. The underlying Sesame infrastructure enables heterogeneous data sources to be used as endpoints within the federation. On top of Sesame, FedX implements the logics for efficient query execution in the distributed setting utilizing the basic Sesame infrastructure (i.e., query parsing, Java mappings, I/O components) and adding the necessary functionality for data source management, endpoint communication and – most importantly – optimizations for distributed query processing.

[5] http://www.fluidops.com/FedX
[6] http://www.fluidops.com/information-workbench/

In FedX, data sources can be added to a federation in the form of any implementation of a Sesame repository. Standard implementations are provided for local, native Sesame repositories as well as for remote SPARQL endpoints. Furthermore, customized mediators can be integrated by implementing the appropriate Sesame interface. With these mediators different types of federations are possible: SPARQL federations integrating (remote) SPARQL endpoints, local federations consisting of native, local Sesame repositories, or hybrid forms. In the SPARQL federation, communication with the endpoints is done using HTTP-based SPARQL requests, while in the local case the native Java interfaces are employed. In the remainder of this paper, we focus on SPARQL federations.

4.2 Parallelization

Query processing in a federated, distributed environment is highly parallelizable meaning that different subqueries can be executed at the data sources concurrently. FedX incorporates a sophisticated parallelization infrastructure, which uses a multithreaded worker pool to execute the joins, i.e., *bound joins* (Section 3.5), and union operators in a highly parallelized fashion. In addition, we employ a pipelining approach such that intermediate results can be processed in the next operator as soon as they are ready – yielding higher throughput.

The parallelization architecture in FedX is realized by means of a `Scheduler` implementation managing a set of `ParallelTask`s and a pool of `WorkerThread`s. A `ParallelTask` refers to a prepared subquery to be executed at a particular data source. As an example, consider a task representing a single step of a nested loop join. In the scheduler, all tasks are maintained in a FIFO queue that the workers *pull* new tasks from. To reduce synchronization costs, worker threads are *paused* when they are idle, and notified when there are new tasks available. Note that only a single worker thread is notified if a new task arrives to avoid unnecessary synchronization overhead. Moreover, worker threads only go to sleep when there are no further tasks in the queue to avoid context switches. After experimenting with different configurations, we defined 25 worker threads for the scheduler as default.

4.3 Physical Join and Union Operators

For the physical `JOIN` operator, we tested with two variants: (1) parallel execution using a simple nested loop join and (2) our *bound join* technique, which we call *controlled worker join* (CJ) and *controlled bound worker join* (CBJ), respectively. Both variants generate tasks for each (block) nested loop iteration and submit them to the scheduler (Section 4.2). The scheduler then takes care of the controlled parallel execution of the tasks. For both, the CJ and CBJ implementation, synchronization is needed because the partial results of tasks belonging to the same join are merged, i.e., all partial results of a particular join are added to the same result set. In SPARQL federations, where (remote) requests cause a certain base cost, the CBJ operator improves performance significantly (see Section 5.2 for details) because the number of requests can be reduced tremendously. This is also the default implementation used in a SPARQL federation.

Note that in local federations with native Sesame stores the first approach, i.e., the CJ operator, outperforms bound joins because simple subqueries with a single triple pattern only, can be evaluated faster. This is because the data source can be accessed through native Java interfaces using Sesame's `getStatements` method, i.e., without prior SPARQL query construction.

Similarly, we provide two implementations for the `UNION` operator: a *synchronous union* (SU) and a *controlled worker union* (CU). The *synchronous union* executes its operands in a synchronous fashion, i.e., one union task after the other, thus avoiding synchronization overhead. In contrast, the *controlled worker union* executes the particular operands using the above described parallelization infrastructure (Section 4.2). The decision which implementation to use in a particular setup depends on the tradeoff between synchronization overhead and execution cost of an operand. In a remote setup, for instance, FedX benefits from parallel execution of a union since network latency and HTTP overhead typically outweigh synchronization costs. Note that union in this context does not solely refer to the SPARQL `UNION` operator but also to subqueries, which have to be evaluated at several relevant data sources resulting in a union of intermediate results. Consequently, for a SPARQL federation the *controlled worker union* is the implementation of choice, and for a local federation unions are evaluated using the *synchronous union* implementation. Note that SPARQL `UNION`s are always executed in the parallelization architecture described above.

5 Evaluation

In this section, we evaluate FedX and analyze the performance of our optimization techniques. With the goal of assessing the practicability of our system, we run various benchmarks and compare the results to state-of-the-art federated query processing engines. In our benchmark, we compare the performance of FedX with the competitive systems DARQ and AliBaba[7] since these are comparable to FedX in terms of functionality and the implemented query processing approach. Unfortunately, we were not able to obtain a prototype of the system presented in [2] for comparison.

5.1 Benchmark Setup

As a basis for our evaluation we use FedBench[8] [17], a comprehensive benchmark suite, which in contrast to other SPARQL benchmarks [4, 18] focuses on analyzing the efficiency and effectiveness of *federated* query processing strategies over semantic data. FedBench covers a broad range of scenarios and provides a benchmark driver to perform the benchmark in an integrative manner.

We select the Cross Domain (CD) as well as the Life Science (LS) data collections from the FedBench benchmark. The reason for our choice lies in the nature of the queries and data sets: both query sets implement realistic

[7] http://www.openrdf.org/

[8] FedBench project page: http://code.google.com/p/fbench/

queries on top of real-world data from the Linked Open Data cloud. The queries focus on aspects relevant for query processing over multiple sources and vary in join complexity, query result size, the number of data sources involved, and structure (i.e., star shaped, chain, or hybrid). Figure 2 in Section 3.1 depicts Life Science query 6 as an example; for space reasons we refer the interested reader to the FedBench project page for the complete query set. To give a better understanding of the queries, we summarize some characteristics in Table 1 a). In particular, we depict the number of triple patterns, reference the number of results on the domain's data sets, and an estimate of the relevant data sources (possibly overestimated).

The used data sources in the two scenarios are part of the Linked Open Data cloud. Table 1 b) summarizes the included data collections. Details to the datasets and various advanced statistics are provided at the FedBench project page. To ensure reproducibility and reliability of the service, we conducted our experiments on local copies of the SPARQL endpoints using the infrastructure provided by FedBench, i.e. for each data source a local process is started publishing the respective data as individual SPARQL endpoint; we did not introduce an additional delay to simulate network latency. All federation engines access the data sources via the SPARQL protocol.

For the respective scenarios, we specify the relevant data sources as federation members upfront (Cross Domain: DBpedia, NYTimes, LinkedMDB, Jamendo, GeoNames; Life Sciences: KEGG, Drugbank, ChEBI, DBpedia). Note that DARQ required additional preprocessing of the *service descriptions*, which are needed for their source selection approach. The duration of this preprocessing is depicted in Table 1 b). Even with 32GB RAM provided, a service description for GeoNames could not be generated with DARQ's tools. Hence, we had to omit the evaluation of DARQ for queries CD6 and CD7 (which require data from GeoNames). Thus, the federation for the Cross Domain scenario had one member less for DARQ.

a) Query Characteristics

Cross Domain (CD)			
	#Tp.	*#Src*	*#Res*
1	3	2	90
2	3	2	1
3	5	5	2
4	5	5	1
5	4	5	2
6	4	4	11
7	4	5	1

Life Science (LS)			
	#Tp.	*#Src*	*#Res*
1	2	2	1159
2	3	4	333
3	5	3	9054
4	7	2	3
5	6	3	393
6	5	3	28
7	5	3	144

b) Datasets

	#Triples	DARQ *SD*
DBpedia	43.6M	01:05:46
NYTimes	335k	00:00:09
LinkedMDB	6.15M	01:07:39
Jamendo	1.05M	00:00:20
GeoNames	108M	n/a
KEGG	1.09M	00:00:18
Drugbank	767k	00:00:12
ChEBI	7.33M	00:01:16

Table 1: Query characteristics of benchmark queries (a) and datasets used (b): Number of triple patterns (#Tp.), data sources (#Src) and results (#Res); number of triples included in datasets (#Triples) and preprocessing time for DARQ Service Description (SD) in hh:mm:ss

All experiments are carried out on an HP Proliant DL360 G6 with 2GHz 4Core CPU with 128KB L1 Cache, 1024KB L2 Cache, 4096KB L3 Cache, 32GB 1333MHz RAM, and a 160 GB SCSI hard drive. A 64bit Windows 2008 Server operating system and the 64bit Java VM 1.6.0_22 constitute the software environment. Sesame is integrated in version 2.3.2 and AliBaba's 2.0b3 build was used. In all scenarios we assigned 20GB RAM to the process executing the query, i.e. the query processing engine that is wrapped in the FedBench architecture. In the SPARQL federation we additionally assign 1GB RAM to each individual SPARQL endpoint process. For all experiments we defined a timeout of 10 minutes and all queries are executed 5 times, following a single warm-up run. All systems are run in their standard configurations.

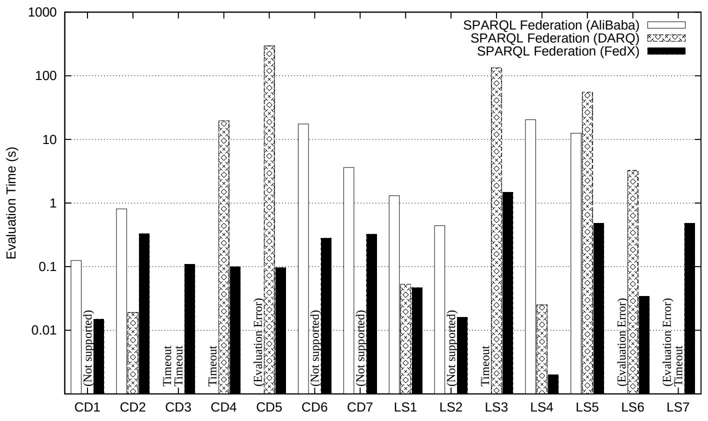
5.2 Experimental Results

Figure 5 summarizes our experimental results of the Cross Domain and Life Science scenarios in a SPARQL federation. We depict the average query runtimes for AliBaba, DARQ, and FedX in Figure 5 a). As an overall observation, we find that FedX improves query performance significantly for most queries. Only in Query CD2 DARQ outperforms FedX. The reason is that FedX' exclusive group optimization in this query is more expensive than using simple triple patterns because the used SPARQL endpoint is more efficient for simple triple patterns for very small intermediate result sets (which is the case in this query as each triple pattern yields only a single result). For many queries the total runtime is improved by more than an order of magnitude. Moreover, timeouts and evaluation errors for this set of realistic queries are removed entirely. The improvement is best explained by the reduction in the number of requests, for which we provide a detailed analysis below. With our optimization techniques, we are able to reduce the number of requests significantly, e.g., from 170,579 (DARQ) and 93,248 (AliBaba) to just 23 (FedX) for query CD3. Such a reduction is made possible by the combined use of our optimization approaches, in particular source selection, exclusive groups, join reordering, and bound joins. Note that query CD2 and LS2 are not supported in DARQ since the query contains an unbound predicate, and that CD6 and CD7 are omitted since we were not able to generate the service description with 32GB RAM.

To measure the influence of caching the results of `ASK` requests during source selection, we performed a benchmark with activated and deactivated cache. The results are summarized in Figure 5 b). We observe that there is a slight overhead due to the additional communication. However, even with these `ASK` requests FedX significantly outperforms the state-of-the art systems for most queries. Our source selection technique – which in contrast to DARQ does not need preprocessed metadata – thus is effective in the federated setting.

Figure 5 c) summarizes the total number of requests sent to the data sources during query evaluation in the SPARQL federation. In particular, we indicate the results for AliBaba and DARQ, as well as for FedX with a nested loop implementation of the *controlled worker join* (CJ) and in the bound join variant using the *controlled worker bound join* (CBJ). These numbers immediately explain the improvements in query performance of FedX. With our optimization techniques, FedX is able to minimize the number of subqueries necessary to process the queries. Consider as an example query CD5: FedX is able to answer this query

a) Benchmark Results



	AliBaba	DARQ	FedX
CD1	0.125	x	0.015
CD2	0.807	0.019	0.330
CD3	>600	>600	0.109
CD4	>600	19.641	0.100
CD5	#	294.890	0.097
CD6	17.499	x	0.281
CD7	3.623	x	0.324
LS1	1.303	0.053	0.047
LS2	0.441	x	0.016
LS3	>600	133.414	1.470
LS4	20.370	0.025	0.001
LS5	12.504	55.327	0.480
LS6	#	3.236	0.034
LS7	#	>600	0.481

x	not supported
#	evaluation error

b) Caching in FedX

	No Caching	Caching
CD1	0.044	0.015
CD2	0.374	0.330
CD3	0.219	0.109
CD4	0.134	0.100
CD5	0.131	0.097
CD6	0.508	0.281
CD7	0.449	0.324
LS1	0.062	0.047
LS2	0.038	0.016
LS3	2.202	1.470
LS4	0.018	0.001
LS5	0.633	0.480
LS6	0.063	0.034
LS7	0.686	0.481

c) Number of Requests

	AliBaba	DARQ	FedX CJ	FedX CBJ
CD1	27	x	7	7
CD2	22	5	2	2
CD3	(93,248)	(170,579)	63	23
CD4	(372,339)	22,331	69	38
CD5	(117,047)	247,343	35	18
CD6	6,183	x	2,457	185
CD7	1,883	x	1,508	138
LS1	13	1	1	1
LS2	61	x	38	18
LS3	(410)	101,386	14,221	2059
LS4	21,281	3	3	3
LS5	16,621	2,666	6,537	458
LS6	(130)	98	315	45
LS7	(876)	(576,089)	5,027	485

d) Join Operators

	CJ	CBJ
CD1	0.016	0.015
CD2	0.349	0.330
CD3	0.203	0.109
CD4	0.134	0.100
CD5	0.115	0.097
CD6	1.560	0.281
CD7	1.336	0.324
LS1	0.053	0.047
LS2	0.025	0.016
LS3	5.435	1.470
LS4	0.001	0.001
LS5	2.146	0.480
LS6	0.103	0.034
LS7	1.763	0.481

Fig. 5: Experimental Results of Cross Domain (CD) and Life Science (LS) Queries in SPARQL Federation: a) Benchmark Results of AliBaba, DARQ, and FedX. b) Influence of Caching `ASK` Requests for Source Selection. c) Total Number of Requests sent to Endpoints; Parentheses Indicate Timeouts after 10min or Evaluation Errors. d) Comparison of Join Operator Implementations in the SPARQL Federation: *Controlled Worker Join* (CJ), *Controlled Worker Bound Join* (CBJ). All Runtimes in Seconds.

in just 18 requests, while DARQ needs 247,343. This is obviously immediately reflected in query runtime, which is just 0.097s in the case of FedX and 294.890s for DARQ. Note that the timeouts and the long runtimes of AliBaba and DARQ are easily explained with the number of requests sent to the endpoints.

In Figure 5 d) we compare the physical join operators of the *controlled worker join* (CJ) and *controlled worker bound join* (CBJ), which use the NLJ and bound joins (BNLJ) technique, respectively. We observe that the CBJ implementation significantly improves performance over the simple CJ variant since in a SPARQL federation we tremendously benefit from the reduction in the number of requests due to bound joins.

6 Conclusion and Outlook

In this paper, we proposed novel optimization techniques for efficient SPARQL query processing in the federated setting. As revealed by our benchmarks, bound joins combined with our grouping and source selection approaches are effective in terms of performance. By minimizing the number of intermediate requests, we are able to improve query performance significantly compared to state-of-the-art systems. We presented FedX, a practical solution that allows for querying multiple distributed Linked Data sources as if the data resides in a *virtually integrated* RDF graph. Compatible with the SPARQL 1.0 query language, our framework allows clients to integrate available SPARQL endpoints on-demand into a federation without any local preprocessing. While we focused on optimization techniques for conjunctive queries, namely basic graph patterns (BGPs), there is additional potential in developing novel, operator-specific optimization techniques for distributed settings (in particular for OPTIONAL queries), which we are planning to address in future work. As our experiments confirm, the optimization of BGPs alone (combined with common equivalence rewritings) already yields significant performance gains.

Important features for federated query processing are the federation extensions proposed for the upcoming SPARQL 1.1 language definition. These allow to specify data sources directly within the query using the `SERVICE` operator, and moreover to attach mappings to the query as data using the `BINDINGS` operator. When implementing the SPARQL 1.1 federation extensions for our next release, FedX can exploit these language features to further improve performance. In fact, the SPARQL 1.1 SERVICE keyword is a trivial extension, which enhances our source selection approach with possibilities for manual specification of new sources and gives the query designer more control.

Statistics can influence performance tremendously in a distributed setting. Currently, FedX does not use any local statistics since we follow the design goal of on-demand federation setup. We aim at providing a federation framework, in which data sources can be integrated ad-hoc, and used immediately for query processing. In a future release, (remote) statistics (e.g., using VoID [1]) can be incorporated for source selection and to further improve our join order algorithm.

Acknowledgments

Research reported in this paper was partially supported by the German BMBF in the project CollabCloud. `http://collabcloud.de/`

References

1. Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. Describing linked datasets - on the design and usage of void. In *In Linked Data on the Web Workshop (LDOW 09), in conjunction with WWW '09*, 2009.
2. Carlos Buil Aranda, Oscar Corcho, and Marcelo Arenas. Semantics and optimization of the SPARQL 1.1 federation extension. In *ESWC*. Springer, 2011.

3. Tim Berners-Lee. Linked data - design issues. Retrieved August 25th, 2011, http://www.w3.org/DesignIssues/LinkedData.html, 2006.
4. Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
5. Orri Erling and Ivan Mikhailov. RDF support in the virtuoso DBMS. In *CSSW*, volume 113 of *LNI*, pages 59–68. GI, 2007.
6. Olaf Görlitz and Steffen Staab. Federated Data Management and Query Optimization for Linked Open Data. In *New Directions in Web Data Management*. Springer, 2011.
7. Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jürgen Umbrich. Data summaries for on-demand queries over linked data. In *WWW*, 2010.
8. Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing SPARQL Queries over the Web of Linked Data. In *ISWC 2009*. Springer, 2009.
9. Olaf Hartig and Andreas Langegger. A database perspective on consuming linked data on the web. *Datenbank-Spektrum*, 10:57–66, 2010.
10. Katja Hose, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Database Foundations for Scalable RDF Processing. In *Reasoning Web*. Springer, 2011.
11. Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
12. Günter Ladwig and Duc Tran Thanh. Linked Data Query Processing Strategies. In *ISWC*, 2010.
13. Günter Ladwig and Duc Tran Thanh. SIHJoin: Querying Remote and Local Linked Data. ESWC, 2011.
14. Andreas Langegger, Wolfram Wöß, and Martin Blöchl. A semantic web middleware for virtual data integration on the web. In *ESWC*, pages 493–507. Springer, 2008.
15. Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19:91–113, 2010.
16. Bastian Quilitz and Ulf Leser. Querying Distributed RDF Data Sources with SPARQL. In *ISWC*, volume 5021, pages 524–538. Springer, 2008.
17. Michael Schmidt, Olaf Görlitz, Peter Haase, Günter Ladwig, Andreas Schwarte, and Thanh Tran. FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In *ISWC*. Springer, 2011.
18. Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP2Bench: A SPARQL Performance Benchmark. In *ICDE*, pages 222–233, 2009.
19. Andreas Schwarte. FedX: Optimization Techniques for Federated Query Processing on Linked Data. Master's thesis, Saarland University, Germany, 2011.
20. Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. FedX: A Federation Layer for Distributed Query Processing on Linked Open Data. In *ESWC Poster and Demo Session Proceedings*. Springer, 2011.
21. Amit P. Sheth. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. In *VLDB '91*, page 489, 1991.
22. Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, pages 595–604. ACM, 2008.
23. Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.
24. Jan Zemanek, Simon Schenk, and Vojtech Svatek. Optimizing SPARQL Queries over Disparate RDF Data Sources through Distributed Semi-Joins. In *ISWC 2008 Poster and Demo Session Proceedings*. CEUR-WS, 2008.