FedSearch: efficiently combining structured queries and full-text search in a SPARQL federation

Andriy Nikolov, Andreas Schwarte, Christian Hütter

fluid Operations AG, Walldorf, Germany
{andriy.nikolov, andreas.schwarte, christian.huetter}@fluidops.com

Abstract. Combining structured queries with full-text search provides a powerful means to access distributed linked data. However, executing hybrid search queries in a federation of multiple data sources presents a number of challenges due to data source heterogeneity and lack of statistical data about keyword selectivity. To address these challenges, we present FedSearch – a novel hybrid query engine based on the SPARQL federation framework FedX. We extend the SPARQL algebra to incorporate keyword search clauses as first-class citizens and apply novel optimization techniques to improve the query processing efficiency while maintaining a meaningful ranking of results. By performing on-the-fly adaptation of the query execution plan and intelligent grouping of query clauses, we are able to reduce significantly the communication costs making our approach suitable for top-k hybrid search across multiple data sources. In experiments we demonstrate that our optimization techniques can lead to a substantial performance improvement, reducing the execution time of hybrid queries by more than an order of magnitude.

1 Introduction

With the growing amount of Linked Data sources becoming available on the Web, full-text keyword search is becoming more and more important as a paradigm for accessing Linked Data. Already today the majority of triple stores support both full-text search and structured SPARQL queries, allowing for hybrid queries that combine these approaches. Given the distributed nature of Linked Data, efficient processing of user queries in a federated environment with multiple data sources has become a central research area in the Semantic Web Community [1, 2].

In practice there are many use cases where hybrid search is required. Consider as an example a scenario involving a text-based database (e.g., a Semantic Wiki) that offers access to its data via a SPARQL interface (e.g., through LuceneSail). In addition, there might be one or more external RDF databases required to fulfill the information needs of the user.

However, execution of hybrid search queries presents several challenges at different levels. The first class of problems is caused by *data source heterogeneity*: Because there is no formal representation of full-text index search included in

a) OWLIM[2]

```
SELECT ?page WHERE {
  ?id rdfs:label ?val .
  ?val luc:luceneIndex "obama" .
  ?nytId owl:sameAs ?id .
  ?nytId nyt:topicPage ?page .
}
```

b) Virtuoso[3]

```
SELECT ?page WHERE {
  ?id rdfs:label ?val .
  ?val bif:contains "obama" .
  ?nytId owl:sameAs ?id .
  ?nytId nyt:topicPage ?page .
}
```

c) LuceneSail[4]

```
SELECT ?page WHERE {
  ?id search:matches ?m .
  ?m search:query "obama" .
  ?m search:property rdfs:label .
  ?nytId owl:sameAs ?id .
  ?nytId nyt:topicPage ?page .
}
```

Triple store vendors use custom vocabularies to express keyword search: OWLIM uses the *http://www.ontotext.com/owlim/lucene#* namespace (luc), Vituoso uses a predefined *bif* prefix and LuceneSail uses the *http://www.openrdf.org/contrib/lucenesail#* namespace (search).

Table 1. Hybrid Search Queries for Different Selected Triple Stores

the standard SPARQL syntax[1], triple store manufacturers model keyword search clauses using proprietary vocabularies. Table 1 shows how a search for the term "obama" and an associated news page is specified for three selected sample repositories. The consequence of this heterogeneity is that hybrid queries written for a particular triple store are system-specific, making it hard to define such a query in a federated environment. Additionaly, a system has to deal with semantic heterogeneity such as, for instance, different scoring schemes for result ranking.

The second challenge concerns *efficient runtime processing* of hybrid queries in order to minimize the execution time. Optimal ordering of operators and the choice of processing techniques (e.g., nested loop join and symmetric hash join) depend on the selectivity of graph patterns and characteristics of the federated environment (e.g., hardware equipment and network latency of repositories). As a federation may include external data sources, collecting statistical information about remote sources may be infeasible (especially, if data is frequently updated). While there are heuristics for estimating the selectivity of SPARQL graph patterns using only static information (e.g., number of free variables, number of relevant data sources), estimating the selectivity of keyword search requests can be particularly difficult.

Finally, given that full-text and hybrid search queries often require only a subset of most relevant results, they represent a special case of top-k queries. Optimal processing techniques for such queries can be different from the ones retrieving complete result sets.

With this work we make the following novel contributions:

- We propose an extension to the SPARQL query algebra that allows to represent hybrid SPARQL queries in a triple-store-independent way (Section 3). On the basis of this algebra extension, we propose query optimization techniques to match keyword search clauses to appropriate repositories, combine retrieved results seamlessly, and reduce the processing time.

[1] http://www.w3.org/TR/sparql11-query/
[2] http://www.ontotext.com/owlim
[3] http://virtuoso.openlinksw.com/rdf-quad-store/
[4] http://dev.nepomuk.semanticdesktop.org/wiki/LuceneSail

- We propose novel runtime query execution techniques for optimized scheduling of tasks (Section 4), supporting on-the-fly adaptation of the query execution plan based on a cost model. These mechanisms allow for time-effective and robust execution of hybrid queries even in the absence of statistical data about federation members.
- We present and evaluate FedSearch (Section 5), which allows to process hybrid SPARQL queries efficiently in heterogeneous federations. Our evaluation based on two benchmarks shows substantial performance improvements achieved with static and runtime optimization mechanisms of FedSearch, sometimes reducing execution time by more than an order of magnitude.

2 Related Work

Processing queries in a federation of data sources has been studied for a long time in the database community [3, 4]. Although this research forms the basis for approaches tackling distributed Linked Data sources, differences in data representation formats and access modes require special handling mechanisms. Existing systems divide into two categories depending on the assumed data access protocol: link traversal [5, 6], where new sources are added incrementally by dereferencing URIs, and endpoint querying [1, 2], which assume a set of known sources providing SPARQL endpoint services. While the former approach is targeted at open scenarios involving public Linked Data sources, the latter is more suitable for enterprise use cases that involve a set of internal repositories and combine their data with selected third-party ones.

The tasks of a query processing engine involve matching query clauses to relevant data sources, query optimization to find an optimal execution plan, and query execution aimed at minimizing the processing time. Default federation support in SPARQL 1.1[5] assumes explicit specification of graph patterns in a SERVICE clause, which are evaluated at the specified endpoint. Some systems go further and automatically determine relevant data sources for different query parts. For this purpose, SPLENDID [7] uses VoID[8] descriptors of federation members, while systems such as DARQ [9] utilize custom source profiles. Avalanche [10] does not require having data source statistics in advance, but gathers this information as part of the query optimization workflow. To avoid the need for statistical data about federation members, FedX [1] uses ASK queries to endpoints, while ANAPSID [2] utilises only schema-level information for source selection and uses sampling-based techniques to estimate selectivity and adaptive query processing to adjust the execution process on the fly. A substantial body of related work already exists on the topic of general SPARQL query optimization: e.g., in [11] an optimizer efficiently combining left-linear and bushy query plans is proposed. A good empirical comparison of the behavior of systems utilizing different join strategies is given by [12].

Keyword-based entity search over structured data represents a special case of semantic search and has been studied in parallel to structured query process-

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

ing (a survey of methods can be found in [13]). A natural evolution of purely keyword-based search involves hybrid search combining both paradigms. For processing such queries, Wang et al. [14] propose an extended ranking schema taking into account features from both full-text and structured data. Although existing approaches already provide complex and efficient query processing models, these techniques usually rely on detailed statistical information about both structured and unstructured data stored in the repositories. For this reason, we consider these methods complementary to our approach, which does not require such apriori information.

Finally, full-text and hybrid queries often require results to be ranked according to their relevance, while typically only the highest ranked ones are of interest for the user. For this reason, hybrid search queries represent a special case of top-k queries, in which the ranking function has to aggregate the ranking scores associated with keyword search results. The $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra [15] was proposed to enable static optimization of query plans containing ORDER and LIMIT modifiers. Our approach extends this algebra to incorporate full-text search clauses. A complementary approach proposed in [6] focuses on top-k query answering using link traversal for data access. This method features push-based processing of algebra operators instead of traditional pull-based techniques to reduce the effect of network latency issues and slow data sources.

3 Hybrid Search in SPARQL

Different triple stores use different syntax to express hybrid search SPARQL queries. In order to process such queries in a federation of heterogeneous data sources, a given query has to be tailored to the standards expected by each federation member. Given that keyword search clauses produce ordered result sets, the query engine must be able to adjust the query plan to retrieve top-k ranked query results in the most efficient way. To achieve this, our proposed approach involves abstracting from repository-specific syntax and expressing keyword search clauses in the query algebra in a uniform way. This section provides the necessary background information and discusses our extension of the SPARQL query algebra to represent hybrid queries and static query optimization techniques aimed at minimizing processing costs.

3.1 Basic Definitions

In a SPARQL query, the WHERE clause defines a *graph pattern* to be evaluated on an RDF graph G. An atomic graph pattern is a *triple pattern* defined as a tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$, where I, L, and V correspond to the sets of IRIs, literals, and variables respectively. Arbitrary graph patterns are constructed from triple patterns by means of JOIN, UNION, FILTER, and OPTIONAL operators. A *mapping* is defined as a partial function $\mu : V \rightarrow (I \cup L \cup B)$ (B is a set of blank nodes) [16], and the domain of the mapping $dom(\mu)$ expresses a subset of V on which the mapping is defined. Then, the semantics of SPARQL queries is expressed by means of a function $[\![P]\!]_G$, which takes as input a graph pattern P and produces a set of mappings from the set of variables $var(P)$ mentioned in P to elements of the graph G. The binding of the variable $?x$ according to the mapping μ is denoted as $\mu(?x)$. The basic query algebra then defines the standard operations (Selection σ, Join $\bowtie$, Union $\cup$, Difference $\setminus$, and Left Join $⟕$) over the sets of mappings, and query evaluation involves translating the query into a *query tree* composed of these operations. For simplicity, in this paper we use the notation $P_1 \bowtie P_2$ to refer to the join operation over sets of mappings produced by the patterns P_1 and P_2.

In order to allow efficient processing of *top-k* queries, the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra [15] introduces a new rank operator $\rho(P)$ which orders the set of input mappings according to some *scoring function* $\mathcal{F}$. The function $\mathcal{F}(b_1, \dots, b_n)$ is defined over the set B of *ranking criteria* $b_i(?x_1, \dots, ?x_m)$, where each ranking criterion specifies a function over the set of variables $var(P)$. Based on the semantics of the rank operator, the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra proposes the rank-aware modifications of the standard combination operators (RankJoin $\bowtie^{\rho}$ and RankUnion U^{ρ}) and defines algebraic equivalences which can be used to reformulate and optimize the algebraic query tree, such as rank splitting, rank commutative law, and propagation of rank over union and join operations.

3.2 Background: FedX Federated SPARQL Query Engine

FedX [1] provides a framework for transparent access to data sources through a federation. It establishes a federation layer which employs several static and runtime optimization techniques. Static optimization includes reordering join operands with the aim of evaluating selective query parts first and executing filters early to reduce the size of intermediate results. At runtime FedX utilizes sophisticated join execution strategies based on distributed semijoins. One such strategy is the *Bind Nested Loop Join* (BNLJ) algorithm denoted by $\bowtie_{BNLJ}$ – a variation of the block nested loop join, in which each subquery sent to a remote data source probes it at once for several partial mappings pulled from the left operand. This significantly reduces the number of required remote requests. In addition, FedX applies pipelining to compute results as fast as possible: a special scheduler maintains a queue of atomic operations, and processes them in parallel. Instead of waiting for execution of each subquery in sequence, the system sends them in parallel and collects results as soon as they arrive, which further improves the execution performance.

The system further identifies situations where a query can be partitioned into so-called exclusive groups Σ_{excl}, which combine several triple patterns that can be evaluated together on the same data source. All these optimization techniques are applied automatically and do not require any interaction with the user. An important feature of FedX is its independence from statistical data about the federation members. Instead of relying on indexes or catalogs to decide on the relevance of a source, FedX uses caching in combination with SPARQL ASK queries. In this way it allows for on-demand federation setup (meaning that data sources can be added and removed from the federation at query time). Our extension of FedX – FedSearch – maintains this property.

3.3 Hybrid Search in SPARQL Algebra

To enable hybrid queries without modifying the SPARQL syntax, existing triple stores express keyword search using special graph patterns which use proprietary vocabularies. At evaluation time, the query engine recognizes these special terms, extracts the search parameters (keywords and projected variables), evaluates the keyword search using its full-text index, and returns a set of mappings binding the projected variables to search answers and their properties (related resource, matched value, relevance score). Thus, graph patterns defining search parameters do not follow the SPARQL semantics, as their result sets are in general not equivalent to the result of algebra operations combining the mapping sets of their constituting triple patterns. This has strong implications for federated query processing, as triple patterns related to keyword search cannot be evaluated separately either on the same or different federation members. Such proprietary graph patterns have to be recognized by the query engine, isolated, and evaluated as whole blocks.

For this purpose, FedSearch introduces the notion of a *keyword search group* as a special graph pattern in the query tree.

Definition 1*: A keyword search group* Σ^{KS} *is a tuple* (q, v, r, s, p, sn) *defined as follows:*

- $q \in L$ *– a literal value representing the keyword query*
- $v \in (V \cup \{nil\})$ *– a variable bound to a literal value matching the keyword*
- $s \in (I \cup V)$ *– a subject resource connected to* v*.*
- $p \in (I \cup V \cup \{nil\})$ *– a property connecting* s *to* v
- $r \in (V \cup \{nil\})$ *– a variable bound to a literal value between 0 and 1 representing a normalized keyword search score (1 corresponding to the highest degree of relevance)*
- $sn \in (V \cup \{nil\})$ *– a value snippet highlighting the matching keywords*

The value *nil* provided for a tuple element implies that the corresponding value or variable does not need to be included in the query: e.g., the queries shown in Table 1 do not explicitly project the relevance score.

Some of these elements are source-dependent: e.g., not all data repositories can provide the value snippet (a standard feature of LuceneSail, but not available in OWLIM), or, more importantly, returned score values cannot be compared across different data sources, even those of the same type. Traditionally, methods for combining ranked search results [17, 18] primarily rely on the analysis of matched values and re-estimation of their relevance to the query string. This procedure, however, is too costly in the context of SPARQL query processing, as it requires additional downloading, parsing, and processing of whole matched values. Thus, meaningful ranking of the combined result set according to some common relevance criterion is impossible without knowing the statistics of back-end repositories.

For this reason, FedSearch operates over normalized query scores lying in the interval $[0, \ldots, 1]$. It also applies the algebra operators *RankUnion* and *RankJoin*. The RankUnion operation over normalized scores (1) preserves the order of results retrieved from the same source and (2) ensures that results from one source do not suppress results from another source due to different scales. To combine ranking scores from different keyword search groups, the RankJoin operation applies the function $\mathcal{F}(r_1, r_2) = avg(r_1, r_2)$. This function preserves monotonicity of the result ranking with respect to the original scores (i.e., if $(r_1[\mu_1] < r_1[\mu_2])$ AND $(r_2[\mu_1] < r_2[\mu_2]) \Rightarrow (\mathcal{F}(r_1[\mu_1], r_2[\mu_1]) < \mathcal{F}(r_1[\mu_2], r_2[\mu_2]))$, while also taking both scores into account and maintaining the original scale.

3.4 Static Query Optimization

FedSearch assumes that the user's query is expressed using the vocabulary supported by one of the federation members. By default, the parsed query tree only consists of basic SPARQL operations applied to atomic triple patterns: for example, Figure 1 shows the initial plan for the example query from Table 1 expressed in LuceneSail syntax. The original FedX system applies static query optimization techniques aimed at adjusting the given query to the federated environment: matching triple patterns to relevant sources, combining together the exclusive groups of triple patterns, reordering join operands according to their estimated selectivity.

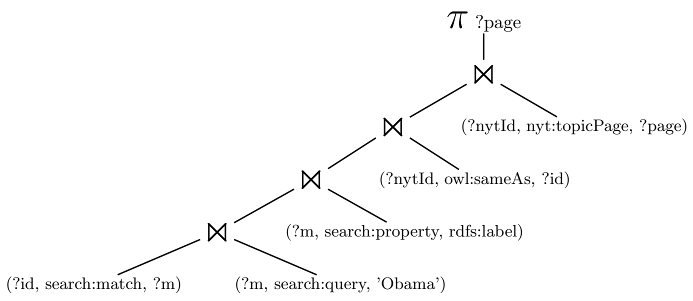

π ?page
⋈
(?nytId, nyt:topicPage, ?page)
⋈
(?nytId, owl:sameAs, ?id)
⋈
(?m, search:property, rdfs:label)
⋈
(?id, search:match, ?m)
(?m, search:query, 'Obama')

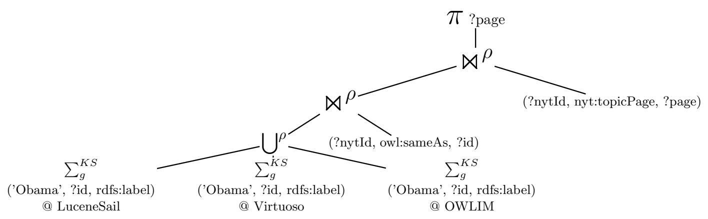
Fig. 1. Unoptimized hybrid search query tree

To process a hybrid query, the task of the static optimization stage includes three additional subtasks:

Detecting and isolating keyword search groups. At this stage, the query optimizer selects and groups triple patterns, which together form keyword search groups. In the query tree, these triple patterns are replaced with a single Σ^{KS} pattern. The result of this stage is an abstract query tree independent of concrete triple stores.

Mapping keyword search groups to relevant data sources. Unless the target is given in the SERVICE clause, each Σ^{KS} can potentially produce mappings from any data source supporting full-text search. Accordingly, the Σ^{KS} is replaced with the grounded repository-dependent graph pattern Σ_g^{KS}, which is associated with all endpoints of the same type (LuceneSail, Virtuoso, etc) and contains corresponding source-dependent triple patterns. The federation configuration contains the backend repository type of its members. If the federation includes repositories of several types, the keyword search group is replaced with a union of several grounded keyword search groups. The result of this stage is a grounded query tree.

Modifying the query tree to take result ranking into account. Each keyword search graph pattern Σ_g^{KS} is expanded to return the score value r_i, if it does not project the relevance score explicitly,



π ?page
$\bowtie^\rho$
$\bowtie^\rho$
(?nytId, nyt:topicPage, ?page)
$\bigcup^\rho$
(?nytId, owl:sameAs, ?id)
Σ_g^{KS} ('Obama', ?id, rdfs:label) @ LuceneSail
Σ_g^{KS} ('Obama', ?id, rdfs:label) @ Virtuoso
Σ_g^{KS} ('Obama', ?id, rdfs:label) @ OWLIM

Fig. 2. Grounded and optimized hybrid search query tree

The resulting expanded query tree is further processed to enforce the ordering of final results according to the combined score $\mathcal{F}(\{r_i\})$ and to minimize the query execution time. For this purpose, equivalence relations defined in the $\mathcal{S}$PARQL-$\mathcal{R}$ANK algebra are applied:

- Partial ranking criteria r_i of keyword search clauses are propagated towards the root of the query tree. This involves converting the standard Union and Join operations to corresponding *RankUnion* and *RankJoin* according to the rules defined in [15]. Relevance scores are combined using normalization and averaging, as discussed in section 3.3
- Top-level ordering criteria (if defined) are propagated down the query tree so that atomic clauses produce their mapping sets already ordered.
- LIMIT thresholds are moved towards the leaves of the tree using the relation $SLICE(P_1 \cup P_2, lim) = SLICE(SLICE(P_1, lim) \cup SLICE(P_2, lim), lim)$. This reduces the costs of local evaluation of keyword search clauses as well as network resources for transferring result sets.

Figure 2 shows the result of the static optimization operations applied to the example query from Table 1 for a federation including repositories of three types: OWLIM, Virtuoso, and LuceneSail.

4 Optimizing Top-k Hybrid Query Execution

Although static query optimization already helps to reduce the expected execution time, the actual performance strongly depends on the way the operators (primarily, joins) are processed. The *Bind Nested Loop Join* technique of the original FedX system significantly reduces the number of required remote requests by grouping together several binding sets in one probing request and using pipelining.

For processing top-k queries and hybrid queries in particular, however, this mechanism is insufficient for several reasons:

- Optimal scheduling of remote requests can differ for top-k queries and queries without a LIMIT modifier. For top-k queries it is important to produce the first complete results as soon as possible, even at the cost of some extra synchronization time, as it can possibly make processing of low-ranked partial results unnecessary.
- More importantly, performance strongly depends on the order of operands. In case if one operand is more selective than the other, reversing their order leads to big differences in execution time. While static optimization tries to sort the join operands according to the expected selectivity, there is no way to estimate selectivity of keywords in the general case.

To deal with these issues, we apply runtime join processing optimization techniques: *synchronization of loop join requests* and adaptive *parallel competing join processing* for queries containing several ordered clauses.

4.1 Synchronization of Loop Join Requests

As an example, let us consider a hybrid search query, which searches for all drugs interacting with aspirin and their side effects, while taking input in different languages:

```
  SELECT ?drugName ?sideEffect WHERE {
1   ?val luc:luceneIndex "acetylsalicylsäure" .        //DBpedia
2   ?id1 rdfs:label ?val .
3   ?id2 owl:sameAs ?id1 .                             //DrugBank
4   ?interaction drugbank:interactionDrug1 ?id2 .
5   ?interaction drugbank:interactionDrug2 ?id3 .
6   ?id3 rdfs:label ?drugName .
7   ?id3 owl:sameAs ?id4 .
8   ?id4 sider:sideEffect ?sideEffectId .              //SIDER
9   ?sideEffectId rdfs:label ?sideEffect .
  }
```

This query involves combining data from 3 sources: DBpedia[6] (triple patterns 1-2), DrugBank[7](3-7), and SIDER[8](8-9). During the static optimization stage

[6] http://dbpedia.org
[7] http://wifo5-04.informatik.uni-mannheim.de/drugbank/
[8] http://wifo5-04.informatik.uni-mannheim.de/sider/

these triple patterns are combined into 3 groups, which we denote as Σ_1^{KS} (DBpedia), Σ_2 (DrugBank), and Σ_3 (SIDER). When performing a bind nested loop join, the algorithm will iterate through the mapped tuples μ_i of the Σ_1^{KS} result set and probe the second operand Σ_2 binding the variable $?id1$. While gradually receiving results μ_{ij} from $(\Sigma_1^{KS} \bowtie_{BNLJ} \Sigma_2)$ and iterating through them, the last operand Σ_3 will be joined using the mappings $\mu_{ij}(?id4)$. As described in [1], this process is parallelized so that each probing subquery in the nested loop is scheduled in a processing queue and then sent in a separate thread. However, depending on the scheduling approach, the process can be performed in two ways:

– *Breadth-first*: In this way, all probing subqueries will be immediately added to the processing queue. Thus, the executor will first send all subqueries for $\Sigma_2(\mu_i(?id1))$ and only then, while results are arriving, send the subqueries for $\Sigma_3(\mu_{ij}(?id4))$.
– *Depth-first*: In this way, when the results from $\Sigma_2(\mu_i(?id1))$ begin to arrive, and subqueries for $\Sigma_3(\mu_{ij}(?id4))$ are added to the queue, the executor immediately moves them to the start of the queue, even if not all $\Sigma_2(\mu_i(?id1))$ requests have been sent yet.

Depending on the type of the query, FedSearch decides on using either of the two techniques. For top-k queries, the depth-first technique is applied. This involves additional synchronization costs to manipulate the task queue and maintain the result set ordering: results returned by probing subqueries to the operands Σ_2 and Σ_3 must be processed in the same order as they were sent. However, the depth-first approach allows receiving first complete results early and potentially terminate the processing early after k results are collected. On the contrary, the breadth-first approach gives an advantage when a complete result set is required: because all nested loops have to be executed completely, extra synchronization handling is unnecessary.

4.2 Adaptive Processing of Rank-Join Operators: Parallel Competing Joining

If a query contains more than one keyword search clause, it is impossible to determine the more selective one without possessing the distribution statistics of keywords. As a result, the join sequence determined at the static optimization stage can lead to a non-optimal execution plan. To avoid this, in the following we present *parallel competing rank join processing*, a novel technique which allows on-the-fly adaptation of the query plan at execution time.

Processing *N*-ary join The high-level idea of this technique is to use a subset of the join operands as *seeds* to allow adaptive query processing. In particular, competing join plans for those operands that determine the ordering are executed in parallel – thus competing against each other – while the other join operands are computed iteratively using the intermediate results from the seeds as soon as they arrive. Whenever an iteration completes with processing its partial result set, a re-evaulation of all query plans takes place to ensure that the next operand is joined to the most selective seed. Finally, the ordered intermediate result sets of competing join plans are combined using the N-ary Pull/Bound Rank Join algorithm (PBRJ) [19], which produces the results ranked according to the aggregated scores of its operands.

Algorithm 1 a) depicts our *Parallel Competing Rank Join* technique. Given the set of ranked join operands P^ρ (including all Σ^{KS} groups) – the *seeds* – and the set of unranked operands P^u, our algorithm first determines suitable competing join plans and then executes each competing *seed* P_i^ρ in parallel. The incoming intermediate results are processed and joined using the cost-based adaptive query technique explained below, yielding ordered results sets for each competing join plan. To combine the partial result sets produced by all seeds, FedSearch uses the N-ary PBRJ variant which modifies the symmetric hash join technique to process RankJoin in an efficient way.

Algorithm 1 Adaptive Processing of Rank Joins

a) Parallel Competing Rank Joins

```
$\mathcal{P}^\rho$: ranked operands (incl. all $\Sigma^{KS}$)
$\mathcal{P}^u$: unranked operands
$\mathcal{P}_{left} \leftarrow \mathcal{P}^u$
for all $P_i^\rho \in \mathcal{P}^\rho$ do
    $Q_i \leftarrow$ joinOrderSort($\mathcal{P}^u$)
    $P_i \leftarrow P_i^\rho$
    start($P_i$)
...
if $\mathcal{P}_{left} = \emptyset$ then
    return PBRJ($\{P_i\}$)
```

b) Processing Incoming Results

```
procedure pushResults($P_{curr}$, $[\![P_{curr}]\!]_G$)
    $P^u_{next} \leftarrow Q_{curr}.next$
    $c = cost(P_{curr} \bowtie_{BNLJ} P^u_{next})$
    for all $P_i \neq P_{curr}$ do
        $pos = Q_i.indexOf(P^u_{next})$
        $c_i = costLeft(P_i)$
            $+\sum_{j=1}^{pos} cost(P_i \bowtie_{BNLJ} P^u_j)$
            $+cost(P_i \bowtie_{BNLJ} P^u_{next})$
        if $c_i < c$ then
            return
    $P_{curr} \leftarrow P_{curr} \bowtie_{BNLJ} P^u_{next}$
    for all $Q_i$ do
        $Q_i \leftarrow Q_i \setminus \{P^u_{next}\}$
    $\mathcal{P}_{left} \leftarrow \mathcal{P}^{left} \setminus \{P^u_{next}\}$
    start($P_{curr}$)
```

The processing step of incoming intermediate results is depicted in Algorithm 1 b). Whenever a seed-computation has received the complete result set $[\![P_{curr}]\!]_G$ for its current operation, a re-evaluation of the execution plans takes place. The re-evaluation procedure estimates the cost c of executing the join $P_{curr} \bowtie P^u_{next}$ and compares it to the respective costs c_i of joining the operand P^u_{next} as part of other *"competitor"* query plans. The cost model used in Algorithm 1 b) is described in detail in the following.

Estimating join cost The basis for our cost model is the average request time of a *Bind Nested Loop Join* (BNLJ) operation at a remote SPARQL service. In general, these execution times can differ substantially for two different queries over the same endpoint. However, FedSearch only estimates the cost of BNLJ subqueries over single triple patterns, which have similar access times. For that reason, FedSearch keeps the statistics of executing BNLJ requests including the average execution time for the data source $\tau_{avg}(s)$ and the average execution time over all sources τ_{avg}. Note that the latter value is used instead of $\tau_{avg}(s)$ if for some source s there is no sufficient historical data.

Cost of a future join. In Algorithm 1, the cost of a join is estimated based on the average request time and the known cardinality of the received result set according to the following formula:

$$cost(P_{rec} \bowtie_{BNLJ} P^{u}_{next}) = \frac{(N(|[\![P_i]\!]_G|) - 1) * \tau_{avg}(s)}{N_{thread}/|\mathcal{P}^{\rho}|} + \tau_{avg}(s)$$

Here $N(|[\![P_i]\!]_G|)$ denotes the number of probing subqueries where according to BNLJ each subquery holds bindings for multiple mappings $\mu \in [\![P_i]\!]_G$. If the operand P^{u}_{next} has multiple data sources, we use the maximal average execution time of all sources, denoted by $max_s(\tau_{avg}(s))$. Finally, N_{thread} holds the number of parallel worker threads used by the system, which means that processing one of the competing query plans can utilize on average $N_{thread}/|\mathcal{P}^{\rho}|$ threads, where $\mathcal{P}^{\rho}$ denotes the set of ranked operands (i.e., the seeds).

For non-atomic operands involving other joins and unions, the cost is determined as follows:

- if the operand is a union, the cost is determined by the maximum cost of the individual union operands multiplied by a coefficient w that estimates the additional cost of combining the results.
- if the operand is a join, left-join or difference, the cost is determined by the sum of the individual costs of the join operands.

Estimating costs of competing branches. The cost c_i of joining the next operand P^{u}_{next} as part of a join sequence Q_i, which competes with the current sequence Q_{curr}, consists of two components:

1. the remaining cost of the current operation, denoted by $costLeft(P_i)$.
2. the cost of joining P_i with all operands in Q_i until P^{u}_{next} (inclusive): $\sum_{j=1}^{pos} cost(P_i \bowtie_{BNLJ} P^{u}_{j}) + cost(P_i \bowtie_{BNLJ} P^{u}_{next})$

The remaining cost is only considered if the operation is running, and can be estimated as $costLeft(P_k) = \tau_{passed} * (\frac{N_{total}}{N_{received}} - 1)$, where τ_{passed} is the time since the start of the operation, N_{total} is a total number of subqueries sent, and $N_{received}$ is the number of subqueries for which results have already been received.

Decision on joining the next operand. Depending on the computed costs for the competing execution sequences Q_i, FedSearch decides to either execute a sequence or reject it. If the cost c is found to be lower than all competitors' costs, the respective execution sequence continues. Otherwise, the execution of the current sequence Q_{curr} is considered rejected in favour of a competing plan Q_{comp}. Note that a rejected execution plan can be re-initiated, if during processing of Q_{comp} its cost is re-estimated to be higher than Q_{curr}, and the operand P^{u}_{next} has not been joined as a part of Q_{comp} yet.

In this way, multiple ranked operands are processed in an efficient way: due to the cost estimation process and the fact that more selective seed queries usually return results earlier, new operands are naturally joined to the more selective seed, thus minimizing the number of required nested loop join subqueries.

5 Evaluation

To validate the FedSearch approach, we performed experiments with two different benchmark datasets. First, we reused the LUBMft query benchmark proposed in [20] for evaluating full-text search performance of RDF triple stores. The benchmark extends the well-known LUBM university benchmark dataset [21] with full-text data and includes a comprehensive set of full-text and hybrid SPARQL queries testing various performance aspects. Second, we reused the set of Life Sciences data sources from the FedBench benchmark for federated query processing [22]. Since FedBench does not include hybrid search queries, we have extended the query set with 6 additional queries involving full-text search clauses. In both sets of experiments target endpoints were hosted as separate OWLIM repositories on a Windows server with two 3GHz Intel processors and 20GB of RAM. We compared the runtime query processing techniques of FedSearch with two other systems: the original FedX architecture and ARQ-$\mathcal{R}$ANK, the open source implementation of SPARQL-$\mathcal{R}$ANK algebra provided by its authors[9]. The original FedX architecture made use of the static optimization techniques described in section 3.4 (so that full-text search clauses could be matched to appropriate sources), but not the runtime optimization. For ARQ-$\mathcal{R}$ANK, which cannot automatically determine relevant data sources, the queries were expanded so that each graph pattern was explicitly targeted at relevant endpoints using SERVICE clauses. Each query was executed 10 times, out of which the first 5 queries were considered "warm-up" to fill the relevant endpoint caches, while the result was equivalent to the average over remaining 5 runs. Benchmark queries, the complete results of the tests, as well as a downloadable version of FedSearch are available online on our web site[10].

5.1 LUBMft benchmark

To perform tests with the LUBMft benchmark dataset, it has been split into 6 parts which represented different endpoints: generated dump files were distributed equally between endpoints resulting in a horizontal partitioning. The benchmark includes 24 queries aimed at testing different triple store capabilities related to keyword search. We used only the first 14 queries covering pure full-text search and hybrid search. Out of these, 8 queries contain only keyword search clauses, while 6 queries are hybrid: 3 queries containing 1 keyword search clause, 2 queries with 2 clauses, and 1 query with 3 keyword search groups. Table 2 shows the average query processing times achieved on the largest

[9] http://sparqlrank.search-computing.org/

[10] http://fedsearch.fluidops.net/resource/FedSearch

k	System	q1.1	q1.2	q3	q4	q6	q7	q8	q9	q10	q11	Geom. Mean
$N_{answers}$		1933	51257	52784	51	1933	1933	704	60	2783	15	
all	FedSearch	0.33	12.51	13.39	0.04	4.96	9.93	13.42	13.36	22.58	14.20	4.75
	FedX	0.29	12.54	13.22	0.02	5.53	6.58	18.87	110.48	455.19 (2)	396.79 (2)	10.24
	ARQ-$\mathcal{R}$ANK	0.62	12.72	13.01	0.22	32.79	66.65	169.34	142.75	Timeout	Timeout	13.63
100	FedSearch	0.08	0.11	0.10	0.04	0.80	1.23	5.01	10.22	26.71	15.48	0.96
	FedX	0.35	11.53	13.34	0.03	4.22	8.79	17.82	123.07	459.90 (3)	106.92 (3)	9.41
	ARQ-$\mathcal{R}$ANK	0.58	1.60	5.07	0.22	2.16	4.79	18.84	142.75	Timeout	Timeout	3.60
10	FedSearch	0.03	0.34	0.04	0.03	0.58	0.73	3.55	11.88	24.25	16.71	0.78
	FedX	0.29	11.56	12.51	0.27	4.31	7.76	18.26	144.22	456.18	153.83 (3)	12.15
	ARQ-$\mathcal{R}$ANK	0.57	1.59	1.52	0.22	0.67	0.98	1.66	142.75	Timeout	Timeout	1.62
1	FedSearch	0.02	0.04	0.04	0.02	0.74	0.72	1.82	11.47	23.95	15.09	0.54
	FedX	0.43	11.54	12.93	0.02	4.21	8.03	17.94	135.80	455.32 (3)	398.25 (3)	10.42
	ARQ-$\mathcal{R}$ANK	0.57	1.59	1.52	0.22	0.54	0.63	0.41	142.75	Timeout	Timeout	1.25

Table 2. Average execution time (sec) for LUBMft queries (N=50) taken over 5 query runs. Numbers in brackets indicate the number of runs which resulted in timeout.

LUBMft dataset ($N = 50$)[11]. For all values of k FedSearch achieved the best overall performance. For pure full-text queries (*q1.1 - q4*) performance of all three systems is similar when the complete result set is required. However, for top-k queries applying static optimization (pushing the limit modifier to atomic clauses) reduces the cost of remote evaluation and result set transfer over the network. FedSearch further improves on this due to parallelization. Hybrid queries with a single FTS clause (*q6*-*q8*) demonstrate respective benefits of depth-first and breadth-first N-ary join processing: the former gives an advantage when executing *top-k* queries (FedSearch and ARQ-$\mathcal{R}$ANK outperform FedX) while the latter is preferred when a complete result set is required. Finally, queries *q9*-*q11*, which contain two or more FTS clauses, illustrate the benefits of the parallel competing rank join algorithm. Results obtained for FedSearch and two other systems differ sometimes by more than one order of magnitude, while FedSearch delivers more robust performance: evaluation time does not depend on the join order produced at the static optimization stage.

5.2 FedBench Life Sciences benchmark

The Life Sciences module of the FedBench benchmark includes 4 datasets containing medicine-related data: KEGG[12], DrugBank[13], ChEBI[14], and a subset of DBpedia[15]. To test hybrid query performance we used a set of 6 queries, which we constructed with the following requirements:

– Each query requires accessing at least 3 datasets from the federation.
– Queries include different proportion of full-text vs graph clauses: 2 queries are full-text only, 2 queries are hybrid with 1 full-text search clause, and 2 queries are hybrid with 2 full-text search clauses.

[11] Queries 2.1, 2.2, 5.1, and 5.2 are skipped due to the lack of space, as they are largely redundant with respect to 1.1 and 1.2. However, they are included in our complete result set available online as well as results for $N = 1, 5, 10$.

[12] http://www.genome.jp/kegg/kegg1.html

[13] http://wifo5-04.informatik.uni-mannheim.de/drugbank/

[14] http://www.ebi.ac.uk/chebi/userManualForward.do

[15] http://dbpedia.org

– Full-text search clauses have different degrees of selectivity.

Evaluation results with these queries are shown in Table 3 (due to smaller size of result sets, we only performed experiments with $k = 10$). The scale of differences between processing engines is smaller than in case of horizontal partioning: mainly because triple patterns with bound predicates do not need to be evaluated on all endpoints, which reduces the overall number of required remote requests. However, the results are largely consistent with the LUBMft experiments. For pure full-text *top-k* search, applying static *top-k* optimization leads to substantial performance improvement if the overall result set is large. For hybrid queries with a single keyword search clause using depth-first N-ary join processing reduces execution time (ARQ-$\mathcal{R}$ANK even marginally outperforms FedSearch due to "fixed costs" of static optimization), while, however, it becomes a drawback when a complete result set is required. Finally, for hybrid queries with multiple FTS clauses the parallel competing bound join algorithm provides a clear advantage.

k	System	q1	q2	q3	q4	q5	q6	Geom. Mean
$N_{answers}$		8129	57	255	930	15	22	
all	FedSearch	0.50	0.09	0.55	6.20	2.33	7.40	1.17
	FedX	0.72	0.03	0.66	6.42	8.47	28.53	1.62
	SPARQL-RANK	0.95	0.24	3.24	32.10	4.56	21.54	3.64
10	FedSearch	0.06	0.03	0.74	0.81	2.36	7.85	0.50
	FedX	0.78	0.02	0.70	6.34	5.30	38.32	1.57
	SPARQL-RANK	0.07	0.01	0.12	0.42	3.73	21.41	0.39

Table 3. Average execution time (sec) for Life Science queries taken over 5 query runs.

6 Conclusion and Outlook

In this paper, we proposed novel static and runtime optimization techniques as a means to enable processing hybrid search queries in a federation of SPARQL endpoints. The evaluation of our implemented system, FedSearch, has shown that it allows for substantial reduction of processing time without relying on statistical data about the content of federation members.

One immediate practical benefit provided by FedSearch is the possibility to realize data access to a diverse set of sources including different triple stores and full-text indices through a common access interface. As a future direction of work, we are planning to utilize this ability to support practical use cases requiring end-user applications to consume data stored in multiple data sources in a seamless way.

While the ability to establish on-demand federation without significant additional effort is a requirement for our system, existing statistical data (e.g., VoID descriptors) can be utilized to further improve its performance. Employing additional techniques to estimate keyword selectivity (e.g., based on [23]) constitutes another promising direction.

References

1. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization techniques for federated query processing on linked data. In: ISWC 2011
2. Acosta, M., Vidal, M.E., Lampo, T., Castillo, J., Ruckhaus, E.: ANAPSID: An adaptive query processing engine for SPARQL endpoints. In: ISWC. (2011)
3. Sheth, A.P.: Federated database systems for managing distributed, heterogeneous, and autonomous databases. In: VLDB '91. (1991) 489
4. Kossmann, D.: The state of the art in distributed query processing. ACM Computing Surveys **32**(4) (2000) 422–469
5. Hartig, O.: Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In: ESWC 2011, Heraklion, Greece (2011) 154–169
6. Wagner, A., Tran, D.T., Ladwig, G., Harth, A., Studer, R.: Top-k linked data query processing. In: ESWC 2012, Heraklion, Crete (2012) 56–71
7. Görlitz, O., Staab, S.: Splendid: Sparql endpoint federation exploiting void descriptions. In: COLD2011, at ISWC 2011, Bonn, Germany (2011)
8. Alexander, K., Cyganiak, R., Hausenblas, M., Zhao, J.: Describing linked datasets - on the design and usage of void. In: LDOW'09. (2009)
9. Quilitz, B., Leser, U.: Querying Distributed RDF Data Sources with SPARQL. In: ISWC. Volume 5021., Springer (2008) 524–538
10. Basca, C., Bernstein, A.: Avalanche: Putting the spirit of the web back into Semantic Web querying. In: SSWS2010 Workshop. (2010)
11. Vidal, M.E., Ruckhaus, E., Lampo, T., Martínez, A., Sierra, J., Polleres, A.: Efficiently joining group patterns in SPARQL queries. In: ESWC 2010. (2010) 228–242
12. Montoya, G., Vidal, M.E., Corcho, Ó., Ruckhaus, E., Aranda, C.B.: Benchmarking federated sparql query engines: Are existing testbeds enough? In: ISWC 2012. (2012) 313–324
13. Tran, T., Mika, P.: Semantic search - systems, concepts, methods and the communities behind it. Technical report
14. Wang, H., Tran, T., Liu, C., Fu, L.: Lightweight integration of IR and DB for scalable hybrid search with integrated ranking support. Journal of Web Semantics **9**(4) (2011) 490–503
15. Magliacane, S., Bozzon, A., Valle, E.D.: Efficient execution of top-k SPARQL queries. In: ISWC 2012, Boston, USA (2012) 344–360
16. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM TODS **34**(3) (2009)
17. Craswell, N., Hawking, D., Thistlewaite, P.B.: Merging results from isolated search engines. In: Australasian Database Conference. (1999) 189–200
18. Si, L., Callan, J.: A semisupervised learning method to merge search engine results. ACM Transactions on Information Systems **21**(4) (2003) 457–491
19. Schnaitter, K., Polyzotis, N.: Optimal algorithms for evaluating rank joins in database systems. ACM Transactions on Database Systems **35**(1) (2008)
20. Minack, E., Siberski, W., Nejdl, W.: Benchmarking fulltext search performance of RDF stores. In: ESWC 2009, Heraklion, Greece (2009) 81–95
21. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics **3** (2005) 158–182
22. Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., Tran, T.: FedBench: A benchmark suite for federated semantic data query processing. In: ISWC 2011
23. Wagner, A., Bicer, V., Tran, T.D.: Selectivity estimation for hybrid queries over text-rich data graphs. In: EDBT 2013, New York, NY, USA, ACM (2013) 383–394