

## Chapter 5

# Federated Data Management and Query Optimization for Linked Open Data

Olaf Görlitz and Steffen Staab

Institute for Web Science and Technologies,  
University of Koblenz-Landau, Germany  
{goerlitz, staab}@uni-koblenz.de

**Abstract.** Linked Open Data provides data on the web in a machine readable way with typed links between related entities. Means of accessing Linked Open Data include crawling, searching, and querying. Search in Linked Open Data allows for more than just keyword-based, document-oriented data retrieval. Only complex queries across different data source can leverage the full potential of Linked Open Data. In this sense Linked Open Data is more similar to distributed/federated databases, but with less cooperation between the data sources, which are maintained independently and may update their data without notice. Since Linked Open Data is based on standards like the RDF format and the SPARQL query language, it is possible to implement a federation infrastructure without the need for specific data wrappers. However, some design issues of the current SPARQL standard limit the efficiency and applicability of query execution strategies. In this chapter we consider some details and implications of these limitations and presents an improved query optimization approach based on dynamic programming.

## 1 Introduction

The automatic processing of information from the World Wide Web requires that data is available in a structured and machine readable format. The *Linking Open Data* initiative<sup>1</sup> actively promotes and supports the publication and interlinking of so called *Linked Open Data* from various sources and domains. Its main objective is to open up data silos and to publish the contents in a semi-structured format with typed links between related data entities. As a result a growing number of Linked Open Data sources are made available which can be freely browsed and searched to find and extract useful information.

The network of Linked Open Data (or simply Linked Data) is very similar to the World Wide Web's structure of web pages connected by hyperlinks. Linked Data entities are identified by URIs. A relationship between two data entities is commonly expressed with the *Resource Description Framework* (RDF) [37] as a triple consisting of *subject*, *predicate*, and *object*, where the predicate denotes the type of the relation between subject and object. Linking a data entity from one data

---

<sup>1</sup> <http://esw.w3.org/SweoIG/TaskForces/CommunityProjects/LinkingOpenData>

source to a data entity in a different data source is very simple. It only requires a new RDF triple to be placed in one data source with the URI of the referenced data entity in the triple's object position. Additionally, the *linked data principles* [5] require that the referenced URI is resolvable, i.e. a common HTTP GET request on the URI returns *useful information* about the referenced data entity. A good overview about the foundations of Linked Open Data and current research directions is given in [8].

A Linked Open Data infrastructure can have different characteristics, namely central or distributed data storage, central or distributed data indexing, and independent or cooperative data sources, which influence how query processing can be implemented on top of it. Not all of the eight potential combinations are reasonable, in fact, three main paradigms can be identified (c.f. Fig. 1).

**Central Repository.** RDF data can be obtained from data sources, either by crawling them or by using data dumps, and put into a single RDF store. A centralized solution has the advantage that query evaluation can be implemented more efficiently due to optimized index structures. The original data sources are not involved in the query evaluation. Hence, it is irrelevant if they are cooperative or not. Note also that the combination of a central data storage with a distributed index does not make sense at all.

**Federation.** The integration of distributed data sources via a central mediator implies a federation infrastructure. The mediator maintains a global index with statistical information which are used for mapping queries to data sources and also for query optimization. Cooperation between data source allows for reducing the query processing overhead at the mediator, as certain query evaluation steps can be moved to the data sources. Otherwise, the whole query execution, including join computations, has to be done at the mediator.

**Peer-to-peer Data Management.** RDF data and index data can both be maintained in a distributed fashion if all data sources are cooperative. The result is a peer-to-peer network where all data source share the responsibility for managing parts of the RDF data and the index data. Hence, no central mediator is necessary and the data storage and processing load can be better balanced across all involved data sources. Without cooperation among the data source it is not possible to realized a distributed index. Peer-to-peer RDF Management is not further discussed in this chapter.

	Central Data Storage		Distributed Data Storage	
Independent Data Sources	n/a	Central Repository	Federation	n/a
Cooperative Data Sources	n/a			P2P Data Management
	Distr. Index	Central Index		Distr. Index

**Fig. 1.** Combinations of characteristics which yield different infrastructure paradigms

This chapter focuses on a federation infrastructure for Linked Open Data and the optimization of federated SPARQL queries. Section 2 illustrates our running example. Section 3 motivates why a federation infrastructure is necessary for Linked Open Data and presents some basic requirements. The related work is presented in Sect. 4 before the detailed architecture of our federation infrastructure is discussed in Sect. 5. The elaboration of optimization strategies follows in Sect. 6. Further improvements for federating Linked Open Data are shortly mentioned in Sect. 7. Finally, Sect. 8 discusses the evaluation of federation infrastructures for Linked Open Data and Sect. 9 concludes the chapter.

## 2 Example

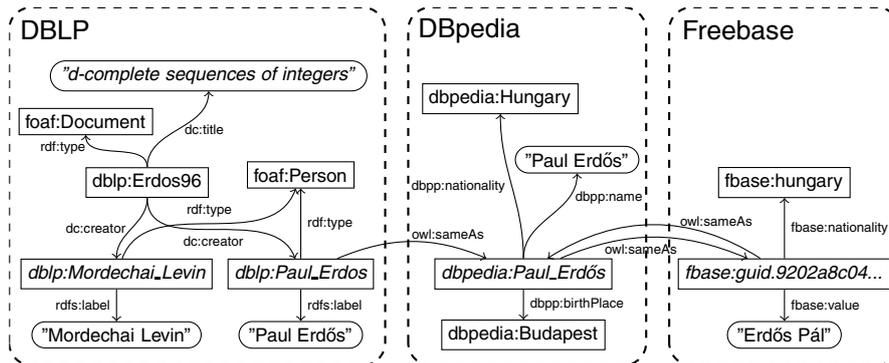
A common albeit tedious task is to get a general overview of an arbitrary research area exploring the most important research questions and the current state-of-the-art. It requires (i) finding important publications and good scientific overviews of the research area, (ii) identifying the most relevant conferences and journals, (iii) finding influential researchers based on number of publications or activity in conference committees etc., (iv) taking social aspects into account like personal relations and joint work on research projects, and (v) filtering and ranking all the information based on individual criteria, like recent activities and hot topics. Especially, students often struggle with this task due to the vast amount of information which can be found on the World Wide Web that has to be collected and connected to form the big picture. Ideally, an easy to use tool takes the description of a research topic and appropriate constraints as input and automatically gathers and merges all relevant information from the different data sources. The results are also valuable when looking for a new job or just to rank the impact or activity of researchers from different institutions/countries.

For the area of computer science, much of this information is already publicly available on the web, in semi-structured and machine readable formats. General information about research areas can be obtained from Wikipedia and, in a semi-structured format, from *DBpedia* [4]. Over 1.3 million publications can be found in the *DBLP Computer Science Bibliography*<sup>2</sup>, including information about authors, conferences, and journals. Additionally, some conference web sites also provide semi-structured data about their conference program, like accepted papers, speakers, and members of the program committee. Other information like the acquisition of projects by research institutions and the amount of funding can be retrieved from funding agencies, e.g. from the EU and the *Community Research and Development Service* (CORDIS), or national funding programs. Last but not least, individual researchers offer information about their affiliation, research interests, and social relations in so called *friend-of-a-friend* (FOAF) profiles [12]. The *CS AKTive Space*<sup>3</sup> project, for example, already integrates some of this information for the UK Computer Science research domain.

<sup>2</sup> <http://dblp.uni-trier.de/>

<sup>3</sup> <http://www.aktors.org/technologies/csaktivespace/>

The example in Fig. 2 illustrates three Linked Open Data sources, i.e. DBLP, DBpedia, and Freebase, which contain RDF data about the mathematician Paul Erdős. The DBLP data describes a publication written by Paul Erdős and a coauthor. DBpedia and Freebase contain information about his nationality. The similarity of data entities is expressed via `owl:sameAs` relations.



**Fig. 2.** Example RDF data located at three different Linked Open Data sources, namely DBLP, DBpedia, and Freebase (namespaces are omitted for readability)

### 3 Linked Open Data Search

Browsing is a natural way of exploring Linked Open Data. Starting with an initial URI that identifies a data entity outgoing links can be followed in any direction, to any data source, to discover new information and further links. However, there is no guarantee that the desired information can be found within a certain number of steps and that all relevant data is reached along the browsing path. Moreover, since links are directed the usefulness of results found by browsing depends heavily on the choice of a good starting point and the followed path.

In contrast, search based on complex queries provides a high flexibility in terms of expressing the desired properties and relations of RDF data entities to be retrieved. However, query evaluation is based on finding exact matches. Hence the user has to have good knowledge about the structure of the data sources and the used vocabularies. Obviously, this is not feasible for a large number of diverse linked data sets. Instead, queries can be formulated based on some standard vocabulary and the query processor applies query relaxation, approximate search, inferencing, and other techniques to improve the quality and quantity of search results. In the ideal case one benefits from higher expressiveness of queries and an abstraction from the actual data sources, i.e. a query does not need to specify which linked data sources to ask for collecting the results.

### 3.1 Requirements

A federation infrastructure for Linked Open Data needs basic components and functionalities:

**A declarative query language** is required for concise formulation of complex queries. Constraints on data entities and relations between them need to be expressible in a flexible way. For RDF there is SPARQL [48], a query language similar to SQL, based on graph pattern matching.

**A data catalogue** is required in order to map query expressions to Linked Open Data sources which contain data that satisfies the query. Moreover, mappings are also needed between vocabularies in order to extend queries with similar terms.

**A query optimizer** is required, like in (distributed) databases, to optimize the query execution in order to minimize processing cost and the communication cost involved when retrieving data from the different Linked Data sources.

**A data protocol** is required to define how queries and results are exchanged between all involved Linked Data sources. SPARQL already defines such a protocol [15] including result formats.

**Result ranking** should be used for Linked Open Data search but is not directly applicable on RDF data since query evaluation is based on exact match. Additional information has to be taken in to account to rank search results.

**Provenance information** should be integrated, especially for ranking results. With a growing number of data sources it becomes more important to trace the origin of result items and establish trust in different data sources.

### 3.2 Architecture Variations

The query-based search on Linked Open Data can be implemented in different ways, all of which have certain advantages and disadvantages.

**Centralized repositories** are the common approach for querying RDF triples. All available datasets are retrieved (e.g. crawled) and stored in a central repository. This approach has the advantage that optimized index structures can be created locally for efficient query answering. However, the local copies and the index data need to be updated whenever something changes in the original data source. Otherwise, old and inconsistent results may be returned.

**Explorative query processing** is based on the browsing principle. A query is first evaluated on an initial data set to find matching data entities and also interesting links pointing to other data sets which may contain more data entities satisfying the query. In an iterative manner, the links are resolved and newly discovered data is fed as a stream into the query evaluation chain. Results are also returned in a streamed fashion as soon as they are available. The search terminates when there are no more links with potential results to follow. This approach evaluates the queries directly

on the linked data and does not require any data copies or additional index structures. However, this also implies an increased communication effort to process all interesting links pointing to other data sources. Moreover, the choice of the starting point can significantly influence the completeness of the result.

**Data source federation** combines the advantages of both approaches mentioned above, namely the evaluation of queries directly on the original data source and using data indices and statistics for efficient query execution and returning complete results. A federated Linked Open Data infrastructure only maintains the meta-information about available data sources, delegates queries to data sources which can answer at least parts of them, and aggregates the results. Storing the data statistics requires less space than keeping data copies. Moreover, changes in the original data have less influence on the metadata since the structure of the data source, i.e. the used vocabulary and the interlinks between them, does not change much. Thus, data changes are more likely to affect the statistics about data entities which influences mostly the quality of the query optimization than the correctness or completeness of the result.

**Table 1.** Comparison of the three architecture variations for querying Linked Open Data with respect to the most relevant characteristics

	central repository	link exploration	data source federation
data location	local copies	at data sources	at data sources
meta data	data statistics	none	data statistics
query processing	local	local+remote	local+remote
requires updates	yes (data)	no	yes (index)
complete results	yes	no	yes
up-to-date results	maybe	yes	yes

### 3.3 Federation Challenges

A federation infrastructure offers a great flexibility and scalability for querying Linked Data. However, there are some major differences to federated and distributed databases. Linked Data sources are loosely coupled and typically controlled by independent third party providers which means that data schemata usually differ and the raw data may not be directly accessible. Hence, the base requirement for the federation infrastructure is that all data sources offer a standard (SPARQL) query interface to retrieve the desired data in RDF format. Additionally, we assume that each Linked Data source also provides some data statistics, like the number of occurrences of a term within the dataset, which are used to (i) identify suitable data sources for a given query and (ii) to optimize the query execution. Hence, the responsibility of the federation infrastructure is to maintain the data statistics in a *federation index* and to coordinate the interaction with the Linked Data sources.

Scalability is without doubt the most important aspect of the infrastructure due to the large and growing number of Linked Data source. That implies two main challenges – an efficient statistics management and an effective query optimization and execution.

### 3.3.1 Statistics Management

Data statistics are collected from all known Linked Data sources and stored in a combined index structure.

**Accuracy vs. index size.** The best query optimization results can be achieved with detailed data statistics. However, the more statistics are collected the larger the required size for storing the index structure. Hence the challenge is to find the right tradeoff between rich statistical data and low resource consumption.

**Updating statistics.** Linked Data source will change over time. Hence the stored statistical information needs to be updated. However, such changes may not be detected easily if not announced. Sophisticated solutions may perform updates on the fly based on statistical data extracted from query results.

### 3.3.2 Query Optimization and Execution

The execution order of query operators significantly influences the overall query evaluation cost. Besides the important query execution time there are also other aspects in the federated scenario which are relevant for the query optimization:

**Minimizing communication cost.** The number of contacted data sources directly influences the performance of the query execution due to the communication overhead. However, reducing the number of involved data source trades off against completeness of results.

**Optimizing execution localization.** The standard query interfaces of linked data sources are generally only capable of answering queries on their provided data. Therefore, joins with other data results usually need to be done at the query issuer. If possible at all, a better strategy will move parts of the result merging operations to the data sources, especially if they can be executed in parallel.

**Streaming results.** Retrieving a complete result when evaluating a query on a large dataset may take a while even with a well optimized execution strategy. Thus one can return results as soon as they become available, which can be optimized by trying to return relevant results first.

## 4 Related Work

The federation of heterogeneous data sources has been a popular topic in database research for a long time. A large variety of optimizations strategies has been proposed for federated and distributed databases [33,55,30,32]. In fact, the challenges for federated databases are very similar to the ones in the federated Linked Data scenario. But there are also significant differences. Distributed databases typically use wrappers to abstract from diverse schema used in different database instances. Such

wrappers are not required for Linked Data as Linked Data source provide a SPARQL endpoint which returns the results in one data format, i.e. RDF. However, database wrappers are typically involved in the estimation of result cardinalities and processing cost. Without them, the estimation has to be handled differently. Optimizations strategies used in federated and distributed databases rely on the cooperation of the individual database instances, e.g. parts of the query execution can be delegated to specific instances in order to make use of data locality or to improve load balancing. Query evaluation in Linked Data sources can not be optimized in the same way since the SPARQL protocol only defines how a query and the results are exchanged with the endpoints. It does not allow for cooperation between them.

Semantic web search engines like Sindice [45], Watson [16], SWSE [26], Falcons [14] allow for document-oriented, keyword-based search. Like typical web search engines, RDF data is crawled from the web and indexed in a central index structure. Frequent re-crawling is necessary to keep the index up-to-date. Support for complex queries is limited or not available at all.

Some general investigations on the complexity of SPARQL query optimization, i.e. identifying the most complex elements and proposing specific rewriting rules, have been done by [53,46]. A recent trend is the development of highly scalable RDF repositories. Implementation like RDF3X [41], Hexastore [60], and BitMatrix [3] focus on optimal index structures and efficient join ordering which allows answering queries directly from the indexed data. Other systems, e.g. YARS2 [28], 4Store [24], and Virtuoso [17] use clustering techniques or federation in order to achieve high scalability, but in an environment with full control over the data storage. Federation of distributed RDF data source has been investigated in systems like DARQ [49] and SemWIQ [35]. Details of these and likewise approaches are presented in Sect. 5.

## 5 Federation Infrastructure for Linked Open Data

The architecture of a federation infrastructure for Linked Data differs not much from the architecture of a federation system for relational data sources, like Garlic or Disco [23,57]. In fact, it is simplified due to the use of SPARQL as common query protocol. First, customized data source wrappers that provide a common data format are not needed. Second, the increasing reuse of ontologies, such as FOAF [12], SIOC [11], and SKOS [38], lessens the need for conceptual mappings (c.f. [51] for ways of integrating conceptual mappings ex post). Figure 3 depicts the main components of a generic federation infrastructure for Linked Open Data.

All data sources are accessible via a SPARQL endpoint, i.e. a web interface supporting the SPARQL protocol. The actual data does not necessarily need to be stored in a native RDF repository. The data source may also use a relational database with a RDF wrapper like the D2R-Server [7].

**The Resource Description Framework (RDF)** is a widely accepted standard in the Semantic Web for semi-structured data representation. RDF defines a graph structure where data entities are represented as nodes and relations between them as edges.

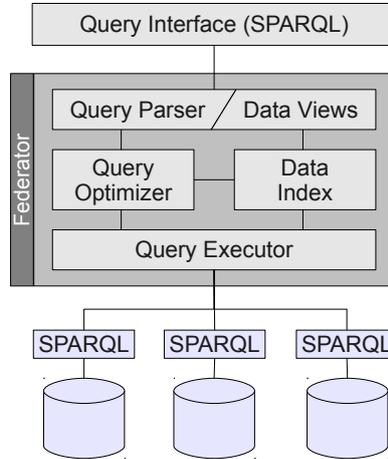


Fig. 3. Architecture of the federation infrastructure

**Definition 1 (RDF Graph<sup>4</sup>).** Let  $U, L, B$  be the pairwise disjoint sets of URIs, Literals, and Blank Nodes. Let  $T = U \cup L \cup B$  be the set of RDF terms. A triple  $S = (s, p, o) \in T \times U \times T$  is called a statement, where  $s$  is the subject,  $p$  is the property, and  $o$  is the object of the statement.

The example in Fig. 4 depicts a set of RDF triples describing a publication with the title "d-complete sequences of integers" written by Paul Erdős and Mordechai Levin (c.f. Fig. 2 for the graph representation). The prefix definitions in line 1-4 simplify the URI notation of the RDF triples in lines 6-13 and improve the readability.

```

1 @prefix dc: <http://purl.org/dc/elements/1.1/>.
2 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 @prefix dblp: <http://dblp.13s.de/d2r/resource/>
4 @prefix foaf: <http://xmlns.com/foaf/0.1/>
5
6 dblp:ErdoSL96 rdf:type foaf:Document.
7 dblp:ErdoSL96 dc:title "d-complete sequences of integers".
8 dblp:ErdoSL96 dc:creator dblp:Paul.Erdos.
9 dblp:ErdoSL96 dc:creator dblp:Mordechai.Levin.
10 dblp:Paul.Erdos rdf:type foaf:Person.
11 dblp:Paul.Erdos foaf:name "Paul.Erdos".
12 dblp:Mordechai.Levin foaf:name "Mordechai.Levin".
13 dblp:Mordechai.Levin rdf:type foaf:Person.
    
```

Fig. 4. Example RDF data from DBLP in Turtle notation

<sup>4</sup> This definition slightly differs from the original W3C Recommendation [37] but is in line with the SPARQL query protocol [48] in allowing literals in subject position.

**The SPARQL query language** [48] defines graph patterns which are matched against an RDF graph such that the variables from the graph patterns are bound to concrete RDF terms.

**Definition 2 (SPARQL Query).** A query  $Q : \{A\}$  is a set of query expressions. Let  $V$  be the set of query variables which is disjoint from  $T$  and let  $\mathcal{P} \in (T \cup V) \times (U \cup V) \times (T \cup V)$  be a triple pattern. A triple pattern  $\mathcal{P}$  is a query expression. If  $A$  and  $B$  are query expressions, then the compositions

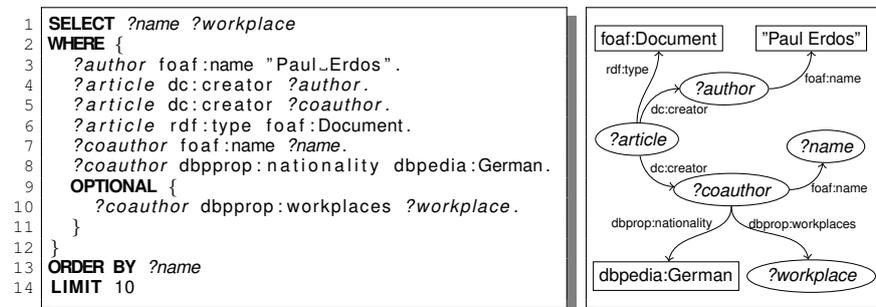
- (i)  $A . B$  (ii)  $A \text{ UNION } B$  (iii)  $A \text{ OPTIONAL } B$  (iv)  $A \text{ FILTER } (exp)$

are query expressions, too. The dot operator in (i) denotes a join in SPARQL.

A function, such as  $(a = b)$ ,  $(a < b)$ , or  $(a > b)$ , with  $a$  and  $b$  being variables or constants, is a filter expression. If  $e_1$  and  $e_2$  are filter expressions, then following expressions are filter expressions, too.

- (i)  $e_1 \parallel e_2$  (ii)  $e_1 \&\& e_2$  (iii)  $!e$

The SPARQL example in Fig. 5 selects German co-authors of Paul Erdős. The graph pattern, which needs to be matched, is defined in the WHERE clause in lines two to twelve and also depicted as a graph on the right side. Co-authorship is defined via the creator relation between people and articles. The German nationality is a property of a person. Line 13 defines an order on the results and line 14 restricts the number of results to 10. The first line specifies the final projection of variables. Namespaces are omitted for readability.



**Fig. 5.** SPARQL example: Find 10 German co-authors of Paul Erdős

Some common elements of SQL, like insert, aggregates, sub-queries, and grouping, are not part of the original SPARQL standard but only supported in version 1.1 [25].

## 5.1 Federator

The main component of the infrastructure is the *federator*. It is responsible for maintaining meta data about known data sources and for managing the whole query evaluation process. The federator offers an interface to the user which allows for keyword-based search and for submitting SPARQL queries. The keyword-based search is ideal for users without a good knowledge about the federated data sets. Although keyword-based search is usually entity centric, it can also be used to derive complex SPARQL queries, as shown in Hermes [59].

The actual query processing includes query parsing, query adaptation, query mapping, query optimization, and query execution. Query adaptation, which will not be further discussed here, is about modifying and extending a query, e.g. by including similar or broader terms and relations from other vocabularies, in order to broaden the search space and to obtain more results. The query mapping is about selecting data sources which can return results for the expressions contained in a query. During query optimization different join strategies for combining results are evaluated. Finally, the query execution implements the communication with the data sources and the processing of the optimized query execution plans.

### 5.1.1 Data Source Selection

Queries may describe complex data relations across different domains. Hence, it is very likely that a single data source may only be able to return results for parts of the query, as it is the case in the example query in Fig. 5. The first part about the co-authorship can be answered by DBLP, the second part about the nationality only by DBpedia. Thus, the respective query fragments have to be sent to different data sources and the individual results have to be merged. The SemaPlover [50], an application for location-based search on top of Linked Open Data, employs explicit data source mappings. This works well for the application's specific scenario but is not flexible enough for a general federation of Linked Open Data.

There have been discussions on how to implement federation in SPARQL, but it has not been included (yet) in the SPARQL standard. The proposed solution [47] is an extension of the SPARQL syntax with the *SERVICE* keyword, which should be used for specifying the designated data source. But an explicit definition of query endpoints requires that the user knows where the data is located. Hence, that solution is impractical for a flexible Linked Data infrastructure. Instead, the query federation should remain transparent, i.e. the federator should determine which data sources to contact based on a data source catalog. Such a catalog is an index which maps RDF terms or even sub-graph structures of a query to matching data sources. The catalog information is tightly coupled with the maintained data statistics (c.f. Sect. 5.3).

```

1 SELECT ?name ?workplace
2 WHERE {
3   SERVICE <http://dblp.uni-trier.de> {
4     ?author foaf:name "Paul.Erdos".
5     ?article dc:creator ?author.
6     ?article dc:creator ?coauthor.
7     article rdf:type foaf:Document.
8     ?coauthor foaf:name ?name.
9   }
10  SERVICE <http://dbpedia.org> {
11    ?coauthor dbpprop:nationality dbpedia:German.
12    OPTIONAL {
13      ?coauthor dbpprop:workplaces ?workplace .
14    }
15  }
16 }
17 ORDER BY ?name
18 LIMIT 10

```

Fig. 6. Example SPARQL query with explicit service endpoints

### 5.1.2 Join Strategies

In a federated system, the main cost factor is the communication overhead for contacting data sources and transferring data over the network. Not only the execution order of join operators, as discussed in Sect. 6, influences the processing cost, but also the execution strategy for transferring the query and the result data. In the following the different approaches and their limitations are discussed in more detail.

**Remote Join.** Executing joins directly at the data sources is the most efficient way as the processing overhead for the federator is minimized and the communication costs are reduced because of smaller intermediate result sets. However, remote joins are only applicable if all parts of the join can be satisfied by the data source. A problem arises when one data source can satisfy more parts of a join expressions than another data source. In that case, the data source which satisfies less join parts may be able to generate results when joining with partial results from the first data source. Hence, even if the first data source evaluates a join completely, it also has to return the results for its intermediate join parts.

**Mediator Join.** Executing the join in the federator (or mediator) after receiving the intermediate results from the data sources is a common approach in existing RDF federation systems [49,56,35]. The join is typically realized as nested-loop-join. Since the SPARQL protocol allows for streaming result sets, i.e. by reading the HTTP response stream, it is possible to start joining a large result set in a nested loop join just after the smaller result set has been received completely. Other join variants like *merge-join* can only be used when the intermediate results are ordered, which is usually not the case. However, the mediator join can significantly increase the communication cost and also the processing cost at the federator if a data source returns a large intermediate result set which is part of a highly selective join.

**Semi Join.** The communication cost and the processing cost of the federator can be significantly reduced with semi-joins [6]. First, the federator retrieves the smaller

intermediate result set of the two joins parts. Then a projection of the join variables is done and the extracted variable bindings are attached to the query fraction which is sent to the second data source. The intermediate results of the second data source are filtered with the transmitted bindings and only the reduced result set is returned to the federator. Thus, the federator has to join smaller intermediate results and also less data has to be transmitted over the network. Essentially, a semi-join realizes a pipelined execution, in contrast to the centralized join where each join part can be executed in parallel.

Unfortunately, the current SPARQL standard does not support the inclusion of variable bindings in a query. The SPARQL federation draft [47] proposes a solution by extending the syntax with the `BINDINGS` keywords. Since this extension is not yet accepted or supported by current SPARQL endpoints the only alternative is to include variable bindings as filter expressions in a query. Both variants are shown in Fig. 7. Although the second approach can be realized with standard SPARQL it is not optimal as the query can be blown up for a large number of bindings, thus increasing the communication cost. DARQ [49] realizes bind joins by sending a query multiple times with all the different bindings. However, this increases the communication overhead as for each binding a separate query and result message has to be sent.

Finally, the proposed `SERVICE` keyword for referencing other datasets may be used within a query fragment to define a bind join. When resolved by a SPARQL endpoint the intermediate result could be directly transferred between the data source without involving the federator. But if multiple data sources include the same sub query it will be sent multiple times to the remote data source and the intermediate results will also be generated more than once if no caching is applied.

<pre> 1 SELECT ?article ?author ?coauthor 2 WHERE { 3   ?article dc:creator ?author. 4   ?article dc:creator ?coauthor. 5 } 6 BINDINGS ?author ?coauthor { 7   ("Paul_Erdos" "Andreas_Blass") 8   ("Paul_Erdos" "Walter_Deuber") 9 10 }</pre>	<pre> 1 SELECT ?article ?author ?coauthor 2 WHERE { 3   ?article dc:creator ?author. 4   ?article dc:creator ?coauthor. 5   FILTER ( 6     (?author = "Paul_Erdos" &amp;&amp; 7      ?coauthor = "Andreas_Blass")    8     (?author = "Paul_Erdos" &amp;&amp; 9      ?coauthor = "Walter_Deuber")) 10 }</pre>
---	---

**Fig. 7.** Variable bindings in SPARQL: via `BINDINGS` syntax extension or as `FILTER` expression

**Bind Join.** The bind-join [23] is an improvement of the semi-join. It is executed as a nested loop join that passes bindings from the intermediate results of the outer part to the inner part, i.e. to the other data source, which uses them to filter its results. This is comparable to a prepared query where variable bindings are provided after the query template has already been optimized. However, SPARQL is also missing a suitable mechanism for defining prepared queries and does not support the streaming of bindings. A common SPARQL endpoint will only start evaluating a query after all query data has been transmitted as the query can only be optimized when all query information is available.

**Filter Chain.** A completely different approach is implemented in [29]. The presented query engine operates in a pipelined fashion by resolving linked data references on-the-fly. Thus, it follows the exploration scheme of linked data. Specific optimizations for speeding up the retrieval are implemented, namely cascaded iterators which operate on the data stream and return results as soon as they become available.

## 5.2 Data Catalog

The data catalog stores two different kinds of data mappings. The first mapping captures relations between RDF terms, like similarity defined with the `owl:sameAs` and `rdfs:seeAlso` predicate. Such information can be used to adapt a query to different data schemata, or simply to broaden the search space. The second mapping associates RDF terms or even complex graph structures with data sources. During the data source selection phase this information is used to identify all relevant data sources which can provide results for query fragments.

The data catalog may be combined with the data statistics described below. Additional statistical information is indeed quite useful for ranking data sources and mappings between RDF terms. Popular predicates, like `rdf:type` and `rdfs:label`, occur in almost every data source. Thus, a ranking helps to avoid querying too many less relevant data sources.

The most common constants in query patterns are predicates. The number of different predicates in a data set is usually limited since predicates are part of the data schema. In contrast, the number of data entities, i.e. RDF terms occurring in a RDF triple's subject or object position, can be quite large. However, they may have just one occurrence. Hence, there is a trade-off between storing many item counts for detailed mappings and minimizing the catalog size. It should also be noted that literals are best stored in a full text index. They will usually occur only a few times in a data set. Moreover, this also allows for searching the data based on string matching.

## 5.3 Data Statistics

Statistical information is used by the query optimizer to estimate the size of intermediate result sets. The cost of join operations and the amount of data which needs to be transmitted over the network is estimated based on these statistics. Very accurate and fine grained data statistics allow for better query optimization results but also require much more space for storing them. Therefore, the main objective is to find the optimal trade-off between accuracy of the statistics and the required space for storing them. However, precise requirements will depend on the application scenario.

**Item counts.** The finest granularity level with the most exact statistical information is implemented by counting data items. In RDF, such counts typically comprise the overall number of triples as well as the number of individual instances of subject, predicate, and object. Counts for combinations of subject, predicate, and

object are useful statistics, too. State-of-the-art triple store implementations like RDF3X[41,42], Hexastore[60], and BitMatrix [3] employ full triple indexing, i.e. all triple variations (S, P, O, SP, PO, SO) are indexed, which allows for generating query answers directly from the index.

**Full text indexing.** The RDF graph of a data source can also be seen as a document which can be indexed with techniques known from information retrieval, like stop word removal and stemming. As a result, data entities may be searched efficiently via keyword-based search. The difference is that typically only literals, as objects in RDF triples, are really suitable for indexing. If URIs should be indexed as well building a prefix-tree is usually a good choice.

**Schema level indexing.** Instead of maintaining statistics for individual data instances one may also restrict the index to the data schema, i.e. the type of instances and relations. This can reduce the overall index size it but also has disadvantages. Certain types and properties, like `foaf:Person` and `rdfs:label`, are widely used in RDF data sets and can be a bad choice for discrimination. Moreover, queries must contain the indexed types. Otherwise, all data sources have to be contacted.

Flesca et al.[18] built equivalence classes for RDF terms using a similarity measure based on types and properties. Counts and references to data sources with equivalent entities are attached to the equivalence classes. This approach can resolve identical data instances across data sources. However, there is no information about the scalability of this approach.

**Structural indexing.** Join statistics contain information about the combination of certain triple patterns. They are much better for estimation the join cardinality since the existence of results for two individual triple pattern in a data source does not automatically imply that there are also results for the joined pattern. However, since there is an exponentially large number of join combinations, not all of them can be stored in an index.

Therefore, as RDF data represents a graph and SPARQL queries describe graph patterns, it makes sense to identify and index only the most common graph structures found in the Linked Data sets. An early proposal for federating SPARQL [56] was based on indexing path structures extracted from data graphs. However, since star-shaped queries are very common for SPARQL queries the path-based approach is not optimal. Instead, generic or frequent sub graph indexing [36,58] can be used but requires sophisticated algorithms for efficiently identifying sub graphs patterns.

A major limitation of structural indexing is its restriction to a single data source. Query federation would benefit significantly from the identification of graph structures across data sources, as certain combinations of data sources could be excluded from the query execution. Moreover, structural indexing is costly and a typical of-line pre-processing step. Hence it is not easily applicable for life Linked Data.

### 5.3.1 Index Size Reduction

Ideally, all required index data should fit into main memory to avoid frequent disk access. Hence, for a large data sets it is necessary to reduce the size of the index data.

Histograms are commonly used for restricting index data to a fixed size while still retaining a high accuracy. Similar data items are put into so called buckets which count the number of items they contain. Space reduction is achieved through a fixed number of buckets. The typically used histogram type (in the database world) is the equi-depth histogram [40], since it provides a good balance of items per bucket – even for skewed data distributions. QTrees, which are a combination of histograms and R-Trees, are used in [27] with three dimensions representing the three components of an RDF triple. Buckets are used as leaf nodes to store item counts and the list of respective data sources which fall into the region covered by the bucket.

Alternatively, Bloom Filters [10] are also suitable for reducing index data to a fixed size. Items are hashed to a bit vector which represents an item set. Membership of items in the set can be efficiently checked with low error rate. However, Bloom filters are not keeping track of the number of items in the represented set. If such information is needed extensions as presented in [44] are necessary.

If a high accuracy of statistics is necessary, it will not be possible to avoid large index structures and disk access. An optimal index layout and common data compression techniques, e.g. as applied in RDF3X [41], can be employed to reduce the required disk space and the frequency of disk access.

### 5.3.2 Obtaining and Maintaining Data Source Statistics

In order to build a federation index with detailed statistics about all involved data sources it is necessary to first pull the statistical details from the data sources. But not all data sources are capable or willing to disclose the desired information. Hence, additional effort may be necessary to acquire and maintain the data source statistics.

**Data Dump Analysis.** Several popular Linked Open Data Sources provide RDF dumps of their data. Analyzing the dumps is the easiest way to extract data statistics and if a new version of the dataset becomes the analysis on the new data dump is simply redone. However, new data versions are not always advertised and only a few large datasets provide dumps. In the future it will be probably more common to have many small frequently changing datasets, e.g. on product information.

**Source Descriptions.** A Linked Data Source may publish additionally some information about the data it contains. For the federator it is necessary to obtain at least some basic statistics like the overall number of triples, the used vocabulary, and individual counts for RDF properties and entities. A suitable format for publishing a data source's statistical information is the *Vocabulary of Interlinked Datasets* (voID) [2]. Additionally, voID also allows to express statistics about data subsets, like the interlinks to other datasets, which is quite useful when retrieving data from different data source. However, complex information, like frequent graph patterns, can not be expressed with voID.

DARQ [49] employs so called *service descriptions* describing data source capabilities. A service description contains information about the RDF predicates found in the data source and it can also include additional statistical information like counts and the average selectivity for predicates in combination with bound subjects or objects. However, the explicit definition of service description for each involved data

source is not feasible for a large number of Linked Open Data sources. Moreover, DARQ restricts the query expressiveness as predicates always need to be bound.

**Source Inspection.** If no dump nor data description is offered by a data source but only a SPARQL endpoint it is still possible to retrieve some statistics by sending specifically crafted queries. But this can be tedious work as initially no knowledge about data set size and data structure is available. So the first step is to explore the data structure (schema) by querying for data type and properties. Additionally, SPARQL aggregates are required to retrieve counts for individual data instances, which are not supported before SPARQL 1.1. But even with counts, crawling data is expensive and SPARQL endpoints typically restrict the number of requests and the size of the returned results.

**Result-based Refinement.** As an alternative for data inspection, it is possible to extract data statistics from the results returned for a query. This approach has little extra processing overhead and it is non-intrusive. However, the lack of exact statistics in the beginning results in increased communication cost and longer execution time as more or even all data sources have to be queried. But with every returned result the statistics will be refined. Moreover, changes in the remote data sets are adapted automatically. Ideally, this approach is combined with some initial basic statistics setup which minimizes the number of inefficient queries in the beginning.

### 5.3.3 Index Localization

The index management can also be viewed from the perspective of where an index is located, i.e. at the data source or at the federator, and which statistical information it maintains, i.e. local data source statistics or global federation statistics.

**Data Source Index.** A data source usually maintains its own statistics for local query optimization. These statistics are not accessible from the outside due to their integration with the data source's query engine. Publicly exposed data statistics, e.g. as *voID* descriptions, are explicitly generated and need to be updated when the actual data changes.

**Virtual Data Source Index.** If a data source does not offer any data statistics publicly they have to be collected by other means, as mentioned above. The result is essentially a virtual index which is created and managed at the federator as part of the federation index.

**Federation index.** The federator maintains a centralized index where the statistical information of all known data sources is collected and updated. The stored statistical information ranges from basic item counts to complex information like frequent graph patterns. All statistical data is connected to the data source where the data can be found.

**Distributed Federation Index.** In a cooperative environment, a federation index may be partitioned and distributed among all data sources to improve scalability. Like in peer-to-peer systems, a data source would additionally keep a part of the

global index and references to data sources with similar information. In such an environment, the query execution can take advantage of the localized index information and less central coordination is needed. The interlinks in Linked Data Sources can already be seen as step into that direction. To support cooperation, a new protocol would be required to exchange information among linked data sources about mutual links and data statistics. The usage of a distributed index for data federation is outlined in [18] but it involves a large number of update messages when data changes. An efficient solution for updating a distributed index is presented in [20].

## 6 Query Optimization

The objective of the query optimization is to find a query execution plan which minimizes the processing cost of the query and the communication cost for transmitting query and results between mediator and Linked Data sources. In the following, basic constraints for the federation of RDF data sources and the structure of query execution plans will be presented before optimization strategies are discussed.

Existing federation approaches for RDF mainly use centralized joins for merging intermediate results at the mediator. The application of semi-joins has not yet been considered for the optimization of distributed RDF queries. We will show that optimization based on dynamic programming, which is presented in more detail, can easily be applied for semi-join optimizations of federated SPARQL queries.

### 6.1 Data Source Mappings

The mapping of multiple data sources to different query fragments implies specific constraints for the query optimization. Consider the SPARQL query in Fig. 8 with two triple patterns which can be answered by three data sources, i.e. `foaf:name` is matched by `{http://dblp.uni-trier.de/, http://dbpedia.org/}` and the combination of `dbprop:nationality` and `dbpedia:German` by `{http://dbpedia.org/, http://rdf.freebase.com/}`.



**Fig. 8.** Query fragment with data source mappings

Although DBpedia could answer the whole query it is not possible to just merge the result from DBpedia with the joined results of the other two sources since a partial result set of DBpedia, i.e. results for a single triple pattern, may also be joined with an intermediate result from `http://dblp.uni-trier.de/` or `http://rdf.freebase.com/`. Hence, each triple pattern has to be evaluated individually at the respective data sources. The results for every pattern are combined via UNION and finally joined. For a SPARQL endpoint, which could resolve remote graphs, the rewritten SPARQL query would look like in Fig. 9.

```

1 SELECT ?coauthor ?name
2 WHERE {
3   {
4     SERVICE <http://dblp.uni-trier.de/> {
5       ?coauthor foaf:name ?name
6     }
7     UNION
8     SERVICE <http://dbpedia.org/> {
9       ?coauthor foaf:name ?name
10    }
11  }.
12  {
13    SERVICE <http://dbpedia.org/> {
14      ?coauthor dbprop:nationality dbpedia:German
15    }
16    UNION
17    SERVICE <http://rdf.freebase.com/> {
18      ?coauthor dbprop:nationality dbpedia:German
19    }
20  }
21 }

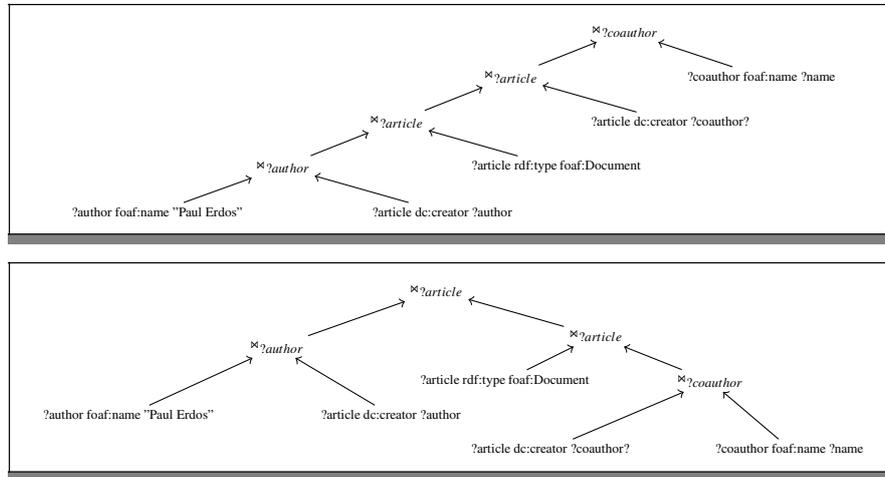
```

**Fig. 9.** Rewritten SPARQL query which maps triple patterns to different data graphs and merges the results via UNION

## 6.2 Query Execution Plans

The output of the query parser is an *abstract syntax tree* containing all logical query operators, e.g. join, union, triple pattern, that make up the query. The tree structure defines the order in which the query operators have to be executed, i.e. child nodes have to be evaluated first. A *query execution plan* is an executable query plan where logical operators are replaced by physical operators, e.g. a join may be implemented as *nested loop join*, *sort-merge join*, *hash-join*, or, as explained in Sect. 5.1.2, as *semi-join* or *bind-join*.

The structure of the query execution plan is also important. The two main types are *left-deep trees* and *bushy trees*. Figure 10 depicts both variations for the running example. Left-deep trees imply basically a pipelined execution. Starting with the leftmost leaf node operators are evaluated one after another and results are passed as input to the parent node until the root node is reached. In contrast, bushy trees allow for parallel execution as sub trees can be evaluated concurrently.



**Fig. 10.** Left deep and bushy query execution plan

The choice of physical operators also affects the execution characteristics. Although the processing and communication cost can be reduced significantly with semi-join it also implies longer query execution times as operators have to wait for the input of preceding operators. Streaming the data between operators can speed up the execution but with the SPARQL standard streaming is only possible for query results. Queries including variable bindings have to be propagated completely before the query execution can start.

The leaf node operators represent *access plans*, i.e. they are wrappers which send the query fragments to the data sources and retrieve and merge the results. If semi-joins are used, the access plan will include the propagated variable bindings in the query fragments. Additionally, a semi join operator needs to project the join variables from the results returned by the child operator, which is executed first, before the variable bindings can be passed on.

Filters are not explicitly mentioned in the execution plans but will be considered during executing. Generally, filters are pushed down as far in the tree as possible, i.e. a filter will be attached to a leaf node or join node if the node satisfies all variables in the filter and there is no other child node that does. Filters attached to leaf nodes are included in the query fragment sent to remote data sources. In case of a semi-join, the filter will also be propagated down to child nodes along with the variable bindings. Otherwise, the filter will be applied by the federator on the (joined) result set.

### 6.3 Optimization Fundamentals

The objective of the query optimization is to find a query execution plan with minimal cost in terms of processing cost and communication cost. Typically, the *query execution time* is the main cost measure. It combines both processing cost and communication cost.

The join order has a significant influence on the query execution time. Small intermediate result sets reduce the communication cost as well as the join processing cost. Thus, the join order optimization is often the main focus of the query optimization. Join order optimization in SPARQL is mainly about optimizing *basic graph patterns*, i.e. a set of conjunctively connected triple patterns. Other operators, like OPTIONAL and UNION, usually have additional constraints which complicate the optimization. They are not further considered here.

There are different optimization strategies which will be discussed shortly. All of them rely on the same two basic measures for estimating the cost of a query execution plan, namely *cardinality* and *selectivity*. Cardinality is the estimated number of elements in a result set which are returned for a query expression. Selectivity defines the estimated fraction of elements which match a query expression. Selectivity values are in the range  $[0..1]$  where a selectivity of 0 means most selective and 1 means least selective.

The cardinality and selectivity for RDF is based on triples and formally defined as follows:

**Definition 3 (RDF graph cardinality).** Let  $|G| = |\{S_i \in G\}|$  be the cardinality of a graph, i.e. the graph size in terms of the overall number of triple statements contained in the graph.

**Definition 4 (RDF term selectivity).** Let  $sel_G(t) = \frac{|\{S_i \in G\}|}{|G|}$  be the selectivity of term  $t \in T$  in graph  $G$  with  $t = subj(S_i) \vee t = pred(S_i) \vee t = obj(S_i)$ .

**Definition 5 (Triple Pattern Selectivity).** The selectivity of a triple pattern  $\mathcal{P}$  is the product of the selectivities of the contained RDF terms:

$$sel_G(\mathcal{P}) = \prod_i^n sel_G(t_i); t_i \in const(\mathcal{P})$$

Consequently, the pattern cardinality  $|\mathcal{P}_G| = |G| \times sel_G(\mathcal{P})$  is the estimated number of matching statements in graph  $G$ . This assumes that the RDF terms in triple patterns are independent.

## 6.4 Optimization Strategies

There are different approaches for query optimization. Usually there is a trade-off between finding the optimal query plan and finding a query plan quickly. Optimization strategies can be classified by static and dynamic optimization. Static optimizers generate one query plan and sticks to it during the whole query execution. Dynamic optimizers may change a query plan during execution due to updated statistics.

Applying heuristics is a common approach to find a good solution fast. Popular heuristics are pushing down filters and sorting query expression by their estimated selectivity. The optimization approach presented in [56] uses iterative improvement and simulated annealing.

Although heuristics can provide good results they will often produce sub optimal execution plans. A guaranteed optimal solution can be found with the *dynamic programming* approach, which is commonly used for query optimization in databases. The SPARQL federation implementation in [49] also uses dynamic programming but details are not given. Dynamic programming will be discussed in more detail in the following section.

Query optimization is a complex topic and there are a lot more approaches, some of which have already been applied for SPARQL. For example, there is RCQ-GA [31], a genetic algorithm for optimizing chain queries. An evolutionary algorithm for approximate querying with anytime behavior is presented in [21].

## 6.5 Dynamic Programming

Dynamic programming [54] is an optimization strategy in traditional relational databases which ensures to find the optimal query execution plan for any given query. All possible query execution plans are iterated and inferior plans are pruned based on a the calculated cost estimates. A cost function is used to estimate the execution cost for each operator based on the cardinality and selectivity of intermediate results.

### 6.5.1 Query Plan Generation

In Dynamic programming query execution plans are generated in a bottom up fashion. The initialization is done by creating an access plan for each query pattern. Then in each iteration step *n-ary joins* are created by combining partial plans from previous iterations. Joins which yield cross products are deferred until the end. An optimized algorithm for the generation of bushy trees is presented in [39]. If many joins and different alternatives for physical operator are involved in the query plan generation the number of plan variations can rapidly grow too large for the available memory. *Iterative Dynamic Programming* [34] can be used in such a situation to iterate the plans in a divide and conquer fashion.

### 6.5.2 Query Plan Evaluation

The result of each iteration step is a set of execution plans which includes equivalent plans with different operator order. The plan evaluation step computes for each plan the execution cost, in order to prune inferior plans. The execution cost is computed recursively based on a cost model which uses the cardinality of each query operator to estimate the individual processing and communication cost. The cardinality of query expressions is defined as follows.

**Definition 6 (Query expression cardinality).** Let  $A_G$  and  $B_G$  be query expressions applied on graph  $G$ . Then,  $|A_G|$  is the expression cardinality and  $sel_G(A) = \frac{|A_G|}{|G|}$  is the selectivity of expression  $A_G$ . Under the assumption that terms in expressions are independent, we define the cardinality of complex expressions as

$$\begin{aligned}
|A_{G_i} \cdot B_{G_j}| &= |A_{G_i}| \times |B_{G_j}| \times \min(sel_{G_i}(A), sel_{G_j}(B)) \\
|A_{G_i} \text{ UNION } B_{G_j}| &= |A_{G_i}| + |B_{G_j}| \\
|A_G \text{ FILTER } (exp)| &= |A_G| \times sel_G(exp)
\end{aligned}$$

**Definition 7 (RDF filter selectivity).** Similar to [54] the selectivity of a filter is defined as:

$$\begin{aligned}
sel_G(a = x) &= sel_G(x) \\
sel_G(a > x) &= \frac{max_a - x}{max_a - min_a} \text{ or } \frac{1}{3} \text{ if not comparable.} \\
sel_G(a || b) &= sel_G(a) + sel_G(b) - sel_G(a) \times sel_G(b). \\
sel_G(a \&\& b) &= sel_G(a) \times sel_G(b) \text{ assuming that } a \text{ and } b \text{ are independent.} \\
sel_G(!a) &= 1 - sel_G(a).
\end{aligned}$$

### 6.5.3 Cost Model

Each query operator is evaluated based on the cost estimates for the individual operations. The cost of a query execution plan  $c(\mathcal{Q})$  is the sum of the cost of all its operators. The cost  $c_{op}$  of an operator applied on a query fragment  $\mathcal{Q}^*$  and a set of variable bindings  $B$  is defined based on a cost model. The constants  $c_{connect}$ ,  $c_{compare}$ ,  $c_{hash}$ , and  $c_{transmit}$  define the cost for establishing a connection to a data source and the cost for comparing, hashing, and transmitting a binding. The cost  $c_{eval}$  is the cost for evaluating a query fragment at a data source. It depends on the actual implementation which is usually not known and may employ index lookups or full table scans. Hence,  $c_{eval}$  is based on rough estimates.

$$\begin{aligned}
c_{remote\_eval}(\mathcal{Q}^*, B) &= c_{connect} + c_{send}(\mathcal{Q}^*, B) + c_{eval}(\mathcal{Q}^*, B) + c_{send}(B') \\
c_{send}(B) &= |B| \cdot c_{transmit} \\
c_{filter}(B, f) &= |B| \cdot c_{compare} \\
c_{union}(B, \hat{B}) &= |B| + |\hat{B}| \\
c_{nested-loop-join}(B, \hat{B}) &= |B| \cdot |\hat{B}| \cdot c_{compare} \\
c_{hash-join}(B, \hat{B}) &= \min(|B|, |\hat{B}|) \cdot c_{compare} + \max(|B|, |\hat{B}|) \cdot c_{hash} \\
c_{sort-merge-join}(B, \hat{B}) &= (|B| \cdot \log|B| + |\hat{B}| \cdot \log|\hat{B}| + |B| + |\hat{B}|) \cdot c_{compare}
\end{aligned}$$

**Parallel Execution Cost.** For parallel query execution plans the overall cost is the maximum cost of all individual plans.

$$c(\mathcal{Q}) = \max(\mathcal{Q}'_1, \dots, \mathcal{Q}'_n)$$

## 7 Improvements for Federation

The presented federation infrastructure and query optimization covers the basic requirements for the federation of Linked Open Data sources. However, there is still

room for improvements. Not every optimization technique, which works for distributed and federated databases, may be applied to federated linked data. Some constraints are due to limitations of the SPARQL standard, as pointed out earlier.

### 7.1 Streaming Results

The execution chain of operators can be a critical bottleneck if large intermediate results are produced or if some data sources have bad response times. The standard SPARQL protocol and its implementation in typical SPARQL endpoints requires that a query, including all filters expressions, must be completely available before the query optimization can be performed. That implies that each query stage in the chain has to be completed before the next one can be executed. In order to speed up the query execution, partial query results may be propagated as soon as they become available. However, such data streaming is not (yet) supported by the SPARQL standard.

### 7.2 Result Ranking

A SPARQL query does not define an order for a result set, unless it is explicitly defined with the keyword `ORDER_BY`. Hence, the result items have to be considered unordered. Nevertheless, some result items may be more relevant than others (from a user's perspective) and should be returned first. However, the criteria for relevance in a federated infrastructure may also include trust and other factors, like response time and data quality. Existing ranking algorithms for RDF data, like RSS [43] or TripleRank [19], are not directly applicable because they are working on the link structure and do not take other aspects into account.

Ranking is also important for the query optimization. The dynamic programming approach [54] considers so called interesting orders, i.e. orders which are required for the final result and can minimize the join processing cost. Such information is not yet considered for federated queries.

### 7.3 Views

Views are a common concept in the relational database world. They allow for data abstraction and simplify the querying of complex data relations. For RDF there is no standardized definition of views. With so called named graphs [13] it is possible to define a context for RDF graphs. But this is rather limited and not flexible enough for managing a large number of RDF graphs, as all RDF triples in a graph context have to be explicitly listed.

Networked RDF Graphs [51] extend named graphs with a SPARQL based view mechanism. They allow users to define RDF graphs both, by extensionally listing statements describing the graph or by using views which are defined as SPARQL queries on other graphs. These views can be used to include parts of other graphs, to transform data before including it and to denote rules. Networked Graphs can be evaluated in a distributed setting using existing protocols. The benefits of networked

graphs is the easy reuse and exchange of graphs, recursive view definitions and the application for data integration from distributed data sources. Especially the last point is interesting for Linked Open Data.

Views are basically an adequate way to establish an abstraction for underlying data schema. They also provide transparency concerning data distribution. If data is moved or merged only the respective view definition needs to be adapted while everything else remains unchanged.

## 8 Performance Evaluation

In order to compare different federation infrastructures, an evaluation scenario is required which can measure the performance based on different criteria. Different benchmarks like LUBM [22], the MIT Barton dataset benchmark [1], or the SP2 benchmark [52] have been developed in recent years, but primarily for evaluating query processing performance of local repositories on a single large data set. Hence, they are not applicable for a distributed infrastructure. Unfortunately, there is no suitable benchmark for evaluating an infrastructure for (federated) Linked Open Data sources. So the problem is to find an evaluation scenario with several linked data sets and a number of complex queries spanning these data sets. Essentially, one has the option to choose between real world and artificial data sets which both have their advantages and disadvantages.

### 8.1 Real World Datasets

The number of available linked data sets has grown significantly in recent months with DBPedia [9] being one of the most popular ones. Thus, there should be lots of interesting information to be queried. However, formulating meaningful queries involving multiple data sources requires a good understanding of the information provided by the data sources in the first place. A good set of queries should cover different query types and should also produce results of different sizes. Due to the large number and diversity of linked data source, plus the constantly changing data, it requires a lot of effort to create such a consistent set of benchmark queries. But more importantly, the possibility to reproduce results is questionable.

### 8.2 Artificial Datasets

Most of the above mentioned benchmarks use artificial data sets. The design objective of such artificial datasets is to cover all typical characteristics of data relations and queries that can be evaluated on top of them. Hence, they allow for comparable evaluations of different systems. The only problem with existing artificial benchmarks is that they are not directly applicable for evaluating data federation which requires the existence of multiple data sources. The obvious solution is to split one large data set into several smaller partitions.

### 8.3 Data Partitioning

The SP2 benchmark [52] is a good basis for creating a data set for benchmarking the federated scenario. It covers a wide range of SPARQL query types and reproduces the characteristics of the DBLP bibliography dataset. Its data generator can be used to create data sets of arbitrary size.

In order to resemble the characteristics of linked data sources the partitioning should be applied vertically and horizontally and also retain a certain overlap between the partitions. Vertical partitioning means splitting the data schema, i.e. different partitions should only share a few common RDF types and predicates to mimic different domains. Horizontal partitioning implies a separation at the instance level, e.g. RDF triples with the same subject are placed in the same partition. Overlap can be realized by placing data instances in multiple different partitions. This usually happens automatically when data instances occurs in subject and object position of RDF triples.

## 9 Summary

A federated infrastructure was presented in this chapter which allows for transparent querying of distributed Linked Open Data sources. The main components of the architecture, namely the federator, the data catalog, and the data statistics were discussed in details. The SPARQL standard does not support all requirements for an efficient processing of federated queries. Specifically, semi-joins, which can significantly reduce the processing and communication cost, are not well supported.

The optimization of SPARQL queries is mainly focusing on join order optimization. A new optimization strategy using semi-joins and dynamic programming was explained in more detail. There is still room for improving the federation of Linked Open Data, e.g. with data streaming, ranking, and the support for data views. Especially, the efficiency of the query processing is not optimal yet.

## References

1. Abadi, D., Marcus, A., Madden, S., Hollenbach, K.: Using the Barton libraries dataset as an RDF benchmark. Tech. rep., Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory (2007)
2. Alexander, K., Cyganiak, R., Hausenblas, M., Zhao, J.: Describing Linked Datasets – On the Design and Usage of void, the “Vocabulary Of Interlinked Datasets”. In: Proceedings of the Linked Data on the Web Workshop. CEUR Workshop Proceedings, Madrid, Spain (2009); ISSN 1613-0073
3. Atre, M., Chaoji, V., Zaki, M., Hendler, J.: Matrix “Bit” loaded: A Scalable Lightweight Join Query Processor for RDF Data. In: Proceedings of the 19th International World Wide Web Conference, Raleigh, NC, USA, pp. 41–50 (2010)
4. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: A Nucleus for a Web of Open Data. In: Proceedings of the 6th International Semantic Web Conference, Busan, Korea, pp. 722–735 (2007)

5. Berners-Lee, T.: Linked Data Design Issues, <http://www.w3.org/DesignIssues/LinkedData.html>
6. Bernstein, P., Chiu, D.: Using Semi-Joins to Solve Relational Queries. *Journal of the ACM* 28(1), 25–40 (1981)
7. Bizer, C., Cyganiak, R.: D2R Server – Publishing Relational Databases on the Semantic Web, <http://www4.wiwiss.fu-berlin.de/bizer/d2r-server/>
8. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data – The Story So Far. *International Journal on Semantic Web and Information Systems* 5(3), 1–22 (2009)
9. Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia – A Crystallization Point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web* 7(3), 154–165 (2009)
10. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13(7), 422–426 (1970)
11. Breslin, J., Decker, S., Harth, A., Bojars, U.: SIOC: an approach to connect web-based communities. *International Journal of Web Based Communities* 2(2), 133–142 (2006)
12. Brickley, D., Miller, L.: FOAF Vocabulary Specification 0.97, Namespace Document (January 1, 2010), <http://xmlns.com/foaf/spec/>
13. Carroll, J., Bizer, C., Hayes, P., Stickler, P.: Named graphs. *Web Semantics: Science, Services and Agents on the World Wide Web* 3(4), 247–267 (2005)
14. Cheng, G., Qu, Y.: Searching Linked Objects with Falcons: Approach, Implementation and Evaluation. *International Journal on Semantic Web and Information Systems* 5(3), 49–70 (2009)
15. Clark, K.G., Feigenbaum, L., Torres, E.: SPARQL Protocol for RDF, W3C Recommendation (January 15, 2008), <http://www.w3.org/TR/rdf-sparql-protocol/>
16. D’ Aquin, M., Baldassarre, C., Gridinoc, L., Angeletou, S., Sabou, M., Motta, E.: Characterizing Knowledge on the Semantic Web with Watson. In: *Proceedings of the 5th International Workshop on Evaluation of Ontologies and Ontology-based Tools (EON)*, Busan, Korea, pp. 1–10 (2007)
17. Erling, O., Mikhailov, I.: RDF Support in the Virtuoso DBMS. In: Pellegrini, T., Auer, S., Tochtermann, K., Schaffert, S. (eds.) *Networked Knowledge - Networked Media*, pp. 7–24. Springer, Heidelberg (2009)
18. Flesca, S., Furfaro, F., Pugliese, A.: A Framework for the Partial Evaluation of SPARQL Queries. In: *Proceedings of the 2nd International Conference on Scalable Uncertainty Management*, Naples, Italy, pp. 201–214 (2008)
19. Franz, T., Schultz, A., Sizov, S., Staab, S.: TripleRank: Ranking SemanticWeb Data By Tensor Decomposition. In: *Proceedings of the 8th International Semantic Web Conference*, Chantilly, VA, USA, pp. 213–228 (2009)
20. Görlitz, O., Sizov, S., Staab, S.: PINTS: Peer-to-Peer Infrastructure for Tagging Systems. In: *Proceedings of the 7th International Workshop on Peer-to-Peer Systems (IPTPS)*, Tampa Bay, Florida, USA (2008)
21. Gueret, C., Oren, E., Schlobach, S., Schut, M.: An Evolutionary Perspective on Approximate RDF Query Answering. In: *Proceedings of the 2nd International Conference on Scalable Uncertainty Management*, Naples, Italy, pp. 215–228 (2008)
22. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3(2-3), 158–182 (2005)
23. Haas, L., Kossmann, D., Wimmers, E.L., Yang, J.: Optimizing Queries across Diverse Data Sources. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, pp. 276–285 (1997)

24. Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2009), Chantilly, VA, USA, pp. 94–109 (2009)
25. Harris, S., Seaborne, A.: SPARQL Query Language 1.1, W3C Working Draft (January 26, 2010), <http://www.w3.org/TR/sparql11-query/>
26. Harth, A., Hogan, A., Delbru, R., Umbrich, J., O’Riain, S., Decker, S.: SWSE: Answers Before Links! In: Proceedings of Semantic Web Challenge (2007)
27. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K.U., Umbrich, J.: Data Summaries for On-Demand Queries over Linked Data. In: Proceedings of the 19th International World Wide Web Conference, Raleigh, NC, USA, pp. 411–420 (2010)
28. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data From The Web. In: Proceedings of the 6th International Semantic Web Conference, Busan, Korea, pp. 211–224 (2007)
29. Hartig, O., Bizer, C., Freytag, J.C.: Executing SPARQL Queries over the Web of Linked Data. In: Proceedings of the 8th International Semantic Web Conference, Chantilly, VA, USA, pp. 293–309 (2009)
30. Heimbigner, D., McLeod, D.: A Federated Architecture for Information Management. *ACM Transactions on Information Systems* 3(3), 253–278 (1985)
31. Hogenboom, A., Milea, V., Frasinca, F., Kaymak, U.: RCQ-GA: RDF Chain Query Optimization Using Genetic Algorithms. In: Proceedings of the 10th International Conference on E-Commerce and Web Technologies, Linz, Austria, pp. 181–192 (2009)
32. Josifovski, V., Schwarz, P., Haas, L., Lin, E.: Garlic: A New Flavor of Federated Query Processing for DB2. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, pp. 524–532 (2002)
33. Kossmann, D.: The State of the Art in Distributed Query Processing. *ACM Computing Surveys* 32(4), 422–469 (2000)
34. Kossmann, D., Stocker, K.: Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems (TODS)* 25(1), 43–82 (2000)
35. Langegger, A., Wöß, W., Blöchl, M.: A Semantic Web Middleware for Virtual Data Integration on the Web. In: Proceedings of the 5th European Semantic Web Conference, Tenerife, Canary Islands, Spain, pp. 493–507 (2008)
36. Maduko, A., Anyanwu, K., Sheth, A., Schliekelman, P.: Graph Summaries for Subgraph Frequency Estimation. In: Proceedings of the 5th European Semantic Web Conference, Tenerife, Canary Islands, Spain (2008)
37. Manola, F., Miller, E.: RDF Primer, W3C Recommendation (February 10, 2004), <http://www.w3.org/TR/rdf-primer/>
38. Miles, A., Matthews, B., Wilson, M., Brickley, D.: SKOS Core: Simple Knowledge Organisation for the Web. In: Proceedings of the 3rd European Semantic Web Conference, Budva, Montenegro, pp. 95–109 (2006)
39. Moerkotte, G., Neumann, T.: Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In: Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, pp. 930–941 (2006)
40. Muralikrishna, M., DeWitt, D.: Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries. In: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, pp. 28–36. ACM Press, Chicago (1988)
41. Neumann, T., Weikum, G.: RDF-3X: a RISC-style Engine for RDF. In: Proceedings of the 34th International Conference on Very Large Data Bases, Auckland, New Zealand, pp. 647–659 (2008)

42. Neumann, T., Weikum, G.: Scalable Join Processing on Very Large RDF Graphs. In: Proceedings of the 35th SIGMOD International Conference on Management of Data, Providence, RI, USA, pp. 627–640 (2009)
43. Ning, X., Jin, H., Wu, H.: RSS: A framework enabling ranked search on the semantic web. *Information Processing and Management* 44(2), 893–909 (2007)
44. Ntarmos, N., Triantafyllou, P., Weikum, G.: Counting at Large: Efficient Cardinality Estimation in Internet-Scale Data Networks. In: Proceedings of the 22nd International Conference on Data Engineering, Atlanta, Georgia, USA (2006)
45. Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., Tummarello, G.: Sindice.com: A Document-oriented Lookup Index for Open Linked Data. *International Journal of Metadata, Semantics and Ontologies* 3(1), 37–52 (2008)
46. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems* 34(3), 1–45 (2009)
47. Prud'hommeaux, E.: SPARQL Federation Extensions 1.1, Editor's Draft (March 25, 2010), <http://www.w3.org/2009/sparql/docs/fed/service>
48. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF, W3C Recommendation (January 15, 2008), <http://www.w3.org/TR/rdf-sparql-query/>
49. Quilitz, B., Leser, U.: Querying Distributed RDF Data Sources with SPARQL. In: Proceedings of the 5th European Semantic Web Conference, Tenerife, Canary Islands, Spain, pp. 524–538 (2008)
50. Schenk, S., Saathoff, C., Staab, S., Scherp, A.: SemaPlorer – Interactive Semantic Exploration of Data and Media based on a Federated Cloud Infrastructure. *Journal on Web Semantics: Science, Services and Agents on the World Wide Web* 7(4), 298–304 (2009)
51. Schenk, S., Staab, S.: Networked Graphs: A Declarative Mechanism for SPARQL Rules, SPARQL Views and RDF Data Integration on the Web. In: Proceeding of the 17th International World Wide Web Conference, Beijing, China, pp. 585–594 (2008)
52. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP<sup>2</sup>Bench: A SPARQL Performance Benchmark. In: Proceedings of the 25th International Conference on Data Engineering, Shanghai, pp. 222–233 (2009)
53. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL Query Optimization (2008); Arxiv preprint arXiv:0812.3788
54. Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., Price, T.: Access Path Selection in a Relational Database Management System. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Boston, MA, USA, pp. 23–34 (1979)
55. Sheth, A., Larson, J.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys* 22(3), 183–236 (1990)
56. Stuckenschmidt, H., Vdovjak, R., Houben, G.J., Broekstra, J.: Index Structures and Algorithms for Querying Distributed RDF Repositories. In: Proceedings of the 13th International World Wide Web Conference, New York, NY, USA, pp. 631–639 (2004)
57. Tomasic, A., Raschid, L., Valduriez, P.: Scaling Heterogeneous Databases and the Design of Disco. In: Proceedings of the 16th International Conference on Distributed Computing Systems, Hong Kong, pp. 449–457 (1996)
58. Tran, T., Haase, P., Studer, R.: Semantic Search – Using Graph-Structured Semantic Models for Supporting the Search Process. In: Proceedings of the 17th International Conference on Conceptual Structures, Moscow, Russia, pp. 48–65 (2009)
59. Tran, T., Wang, H., Haase, P.: Hermes: Data Web search on a pay-as-you-go integration infrastructure. *Web Semantics: Science, Services and Agents on the World Wide Web* 7(3), 189–203 (2009)
60. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. In: Proceedings of the 34th International Conference on Very Large Data Bases, Auckland, New Zealand, pp. 1008–1019 (2008)

