SPARQL Query Optimization for Federated Linked Data

Desared Osmanllari

Chair of Computer Science 5 “Information Systems and Database Technology”
at RWTH Aachen University,
Ahornstr. 55, 52056 Aachen, Germany
desared.osmanllari@rwth-aachen.de

Abstract. The Web has evolved from a system of internet servers supporting formatted documents into a web of linked data. In the last years, the Web of Data is constantly growing. Consequently, it has developed a large collection of interlinked data sets from multiple domains. To exploit the diversity of all available data, federated queries are needed. However, many problems such as processing power, query response time, high workload or outdated information are hindering the query processing. In this paper, I am aiming to explain various optimization techniques which have the potential to lead a significant improvement on the final query runtime. I will start by briefly introducing recent approaches of federation and show why SPARQL federation endpoints are mostly in my focus. Specifically, I will compare state-of-the-art SPARQL query federation engines and analyze respective optimization approaches. The main federation engines I will analyze in terms of query optimization are FedX, DARQ and SPLENDID. As the result I provide concrete examples and conclude which of the engines has the best performance based on the query execution time as key criterion.

Keywords: Linked Data, Semantic Web, Query Optimization, Query Processing, SPARQL Query, FedX, DARQ, SPLENDID, SPARQL endpoints, Federated Queries

1 Introduction

The fast transition from the Web of Documents to the Web of Linked Data has increased the amount of available RDF data sources. Many semantic web applications retrieve interlinked information from diverse domains. Currently, the Linked Open Data[1] (LOD) Cloud contains more than 124 billion triples from 2005 different datasets and this number is constantly increasing. LOD has a decentralized and linked architecture. To exploit the diversity of all available data, federated SPARQL queries are needed. SPARQL [1] is recommended from W3C[2] as a RDF query language and protocol. Now, it is possible to access linked data through a standard interface and query language. However, it is still difficult to properly integrate data from various data sources.

[1] https://stats.lod2.eu/
[2] https://www.w3.org/

In this paper, I will shortly introduce the basic federation mechanisms used to get access to the Linked Data. Two different paradigms are applied for this purpose. *Data Warehousing* is one of the most well-known mechanisms because it can provide connection to its data even when the network connection is lacking [2]. On the other hand, this mechanism doesn't support up-to-date information, since all the data sources are stored and physically loaded from a given repository. Consequently, the whole information must be downloaded from the servers before it can be used. This process increases the data workload and saves abundant but useless data. In this particular case, query processing will require a high response time and might reduce engines' performance.

My intention is to cover the so called *Federated Query Processing* [3] and more specifically focus in SPARQL endpoints implementation. Federated Query Processing intend to split the query into several parts and federate it against multiple data sources. In this mechanism, there is no need for data synchronization since the federation is directly made upon updated sources. The main approaches to federate queries are: Triple Pattern Fragments, Live Linked Data Streams and SPARQL Endpoint Federation. I have to emphasize that my analyze will be in terms of optimization and evaluation. I will start by pointing out the main problems in each approach, evaluating their performance and finally comparing their main differences.

Next, I will go more deeply analyzing query optimization processes in SPARQL endpoint implementation, which is one of the most popular approaches of Federated Query Processing. SPARQL endpoints are supported by most of the Linked Data servers. On the other hand, many obstacles are produced because of the increasing number of semantic web applications operating upon this approach. In the latter case, SPARQL servers have to deal not only with the amount of users, but also must process all the data for the requested queries.

A SPARQL endpoint is a RESTful web service which permits users to run a SPARQL query over the data set that the endpoint provides [2]. In this case, multiple repositories are accessed via a federation layer. Loading of data is not required since an ad hoc federation can be built by simply adding an additional SPARQL endpoint to the federation [3]. But, what we are really concerned about is the query processing, which becomes more complex. The access via SPARQL endpoint is read-only and statistical information needed for query optimization is not accessible. Furthermore, we incur the performance penalty by the network communication.

These obstacles and others, I am going to study by introducing three SPARQL federation engines: FedX, DARQ and SPLENDID. Each of these engines applies different types of optimization techniques, whose purpose is to increase query performance based on the execution time as key criterion. Furthermore, I will consider some additional criteria for query evaluation such as: number of source selected, total number of SPARQL ASK requests used and source selection time [4]. A special focus will be dedicated to FedX, an engine that outperforms the others in terms of the performance when a cache is involved. The use of this cache will considerably reduce the source selection and query execution time. At the end, I will conclude by providing a comparison among the engines in terms of execution time and optimization techniques.

Structure of this paper. The rest of this paper is structured as follows. Section 2 gives a short introduction to Federated Query Processing and its main approaches.

In section 3, I will cover in more depth SPARQL Endpoint Federation and introduce the engines I am going to analyze. In section 4, I show optimization techniques used in each engine and analyze their performance. Also, I will give a comparison among these engines in terms of query runtime performance as key criterion. In section 5, I summarize the main features implemented in each engine and discuss possible future improvements. The last section will be a conclusion of my report where I state my own opinion.

2 Federated Query Processing

In this chapter, I am going to introduce the main approaches to Federated Query Processing. As I mentioned in the introduction, federated queries have been developed to access, retrieve and combine information from multiple data sources. The main concept behind federated queries is to split a single query among multiple SPARQL endpoints and then combine the results. Anyway, they support different design alternatives which affect the practicality of query processing and the performance. The main approaches analyzed in this section will be: Triple Pattern Fragments, Live Linked Data Streams and SPARQL endpoints.

2.1 Triple Pattern Fragments

The main advantage supported by Triple Pattern Fragment approach is the server cost minimization. This is done by offloading the query execution from the server to the client side [5]. Consequently, an interface to RDF data is created. This technique balances the server cost with the availability of the interface. As proposed in [5], a Triple Pattern Fragment consist of the following components: data formatted as triple patterns, some metadata, hypermedia controls and a selector used for triple patterns selection.

In this approach, the user sends a triple pattern to the server. Then the server responds with Triple Pattern Fragments. A browser in the role of a client gets and displays the results in a listed form (data & metadata) . The server just needs to send the triple patterns matching to the user request and compress it into a Triple Pattern Fragment. This reduces the server cost and loads the remaining process into the client side.

Van Herwegen has explained in [6] how the query execution optimization for clients in Triple Pattern Fragments works. Firstly, the client splits the query into separate triple patterns and sends them as requests to the Triple Pattern Fragment servers. As introduced in the previous paragraph, the server sends back multiple triple patterns based in the requested triple pattern. The client retrieves the results from the server and merges them. The algorithm provided in [6] selects the triple pattern with the fewest results in order to process it further. After accumulating the results of one pattern, all the bindings will be applied to the other patterns. The algorithm will restart in the new context of triple patterns.

On the other side, the server should provide a stable availability to the clients. For this reason, the server is responsible to fetch data from existing SPARQL endpoints and consequently to provide the Triple Pattern Fragments. Those patterns are created with high performance, as the endpoint is not busy while the server fetches these data. This approach is very beneficial for federated query processing, because the whole query execution part is shifted to the client side [5]. When using the Triple Pattern Fragments, the provider of the data does not need to worry about his server resources or capabilities. In terms of evaluation, this approach provides higher scalability and lower processing time compared to SPARQL endpoints.

2.2 Live Linked Data Streams

Live Linked Data Streams is another approach to process federated queries. In contrast to the other two approaches analyzed in this chapter, Streaming Linked Data is still a new area of research. In this area, optimizing queries is not the hardest problem yet. First, accessing the data is difficult, since a continuous stream can not be stored as static data. Secondly, the data is produced by multiple sensors and should be stored as RDF streams. After surpassing this two steps, a query language to access and get information over streams must be invented.

One proposal how to publish Data Streams as Linked Data is given by Barbieri in [7]. Moreover, a new language(C-SPARQL) responsible for continuous querying is proposed in [8]. This query language offers the opportunity to query RDF stream data. In difference to SPARQL, this language computes an interval in which the results must be updated, since the stream is not static.

To better understand how Live Linked Data Streams might work in the near future, let's consider a simple example. FlightRadar[3] is a service which provides real-time streams over plane flights. This application makes use of streaming data and visualizes them in real time. The purpose of Linked Data Streams is to build similar applications by using federated queries over RDF sources.

Even though Live Linked Data Streams is a specifically unique approach to deal with, it offers some advantages. Firstly, it expands the scope of data sources from where we can access Linked Data. Secondly, this approach guarantees live and up-to-date information, since the data is live-streamed and recently it can be queried. By this approach, we can surpass one of the problems presented in abstract: outdated information. Anyway, this approach is new and still undeveloped, so I am not going to analyze it further in terms of query optimization.

2.3 SPARQL Endpoint Federation

SPARQL Endpoint Federation is the most distributed approach to federate queries. Today most of the servers are running against SPARQL endpoints. Also, the development of SPARQL endpoints has started much earlier than the distribution of Triple Pattern Fragments. This makes it more popular and widespread. For this reason, I am going to analyze this approach more deeply in the next chapters.

A SPARQL endpoint is a RESTful web service which permits users to run a SPARQL query over the data set that the endpoint provides [2]. The communication is provided through HTTP. The client sends a SPARQL query to the server using the endpoint interface. The server executes the query and returns results in various formats: CSV, JSON,

[3] https://www.flightradar24.com/

HTML or XML. The endpoint is specific for each data set. I am mostly familiar with BIO2RDF project, which is accessible via **`http://bio2rdf.org/sparql`** endpoint.

Version 1.1 of SPARQL federates subqueries to different endpoints by using the keyword SERVICE[4]. By using this keyword, users can federate their own queries over endpoint interface. This technique is practical in the first sight, but sometimes it can become problematic as explained in [4]. In this case, the whole processing power will be offloaded in the server side, which has to deal with unlimited number of requests. Thus optimization of SPARQL queries is unavoidable, since the complexity of requests might be irrational. Also, servers can stop responding, since the endpoints are available to everyone and consequently requests can be generated simultaneously.

Example 1. In Listing 1.1, there is given a SPARQL query which asks for all the metabolites in Wikipedia dataset having InChlKeys from Wikidata. So, the users' aim is to retrieve information from two different data sources: Wikipedia and Wikidata. Inside the query pattern, a connection with the remote SPARQL endpoint is created. In this specific case, we are running the query under Wikipedia endpoint. On the other side, we need to send the subquery which identifies the InChlKeys into Wikidata endpoint. SERVICE keyword is used to send this subquery into the remote SPARQL endpoint identified by the URI: *https://query.wikidata.org/sparql*.

Listing 1.1. SPARQL Endpoint Federation via the SERVICE keyword

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>

SELECT ?metabolite ?wikidata ?inchikey
WHERE {
  ?metabolite a wp:Metabolite ;
    wp:bdbWikidata ?wikidata .
  SERVICE <https://query.wikidata.org/sparql> {
    ?wikidata wdt:P235 ?inchikey .
  }
}
```

Another possible way to query against multiple SPARQL endpoints is by executing a Data Source Selection. In this technique, the usage of SERVICE keyword is not needed, but the user must determine the data source for the given query. In the next chapter, I am going to introduce some engines applying this specific approach. Later, I will focus mostly in optimization techniques used in each engine and study their effects upon the overall performance.

In Figure 1, there is a visualized comparison of required processing(*filled bars*) and data transfers(*dotted lines*) between SPARQL endpoints and Linked Data Fragments. SPARQL endpoints perform all the processing on the server, while Linked Data Fragment servers support only simple requests. This leads to a fast query execution with low bandwidth and an overloaded server in SPARQL endpoints approach. Linked Data Fragment server can load higher loads, while clients perform querying. Because the processing part is offloaded to the client side, the caching of Fragments can be performed. Compared to Live Linked Data Streams, SPARQL endpoints do not provide

[4] https://www.w3.org/TR/sparql11-service-description/

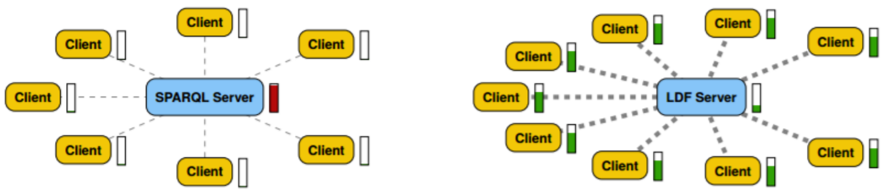

Fig. 1. SPARQL Endpoint vs Triple Pattern Fragment [5]

live data, since the data is stored in the server. For a live reference to the world, Linked Data Streams is the best approach among all of them.

3 SPARQL Endpoint Engines

In this chapter, I am going to further analyze SPARQL Endpoint Federation by introducing three federation engines built upon this approach. In all these engines you can find an unique element: a federation layer as shown in Figure 2. This layer is used to access multiple data sets. More specifically, it computes an execution plan for the queries, fetches and combines the results of all requested endpoints.

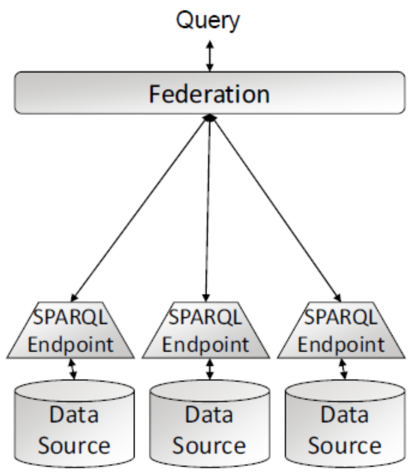

Fig. 2. Federation over multiple SPARQL endpoints[3]

By using SPARQL endpoints, there is no need to download the data in advance. Users have to query specific information only against the relevant SPARQL endpoint. As stated in the previous paragraph, multiple repositories are accessed via a federation layer. Still, the access is completed by SPARQL endpoints of data providers. Loading in federated sources can be done by adding an ad hoc federation, which on the other side can be built by simply adding an additional SPARQL endpoint to the federation[3].

In my interest stands the query processing, which in this case becomes more difficult. The access via a SPARQL endpoint is restricted to read-only, although there are some proposals for a SPARQL-Update approach. Also, statistical information needed for query optimization is not accessible. On the other hand, servers in this approach are usually overloaded and sometimes they stop due to arbitrary complexity of requests. All these reasons create a necessity to optimize SPARQL queries. In the next sub-chapters, I will deal with the architecture design and main components in each of the following engines: FedX, DARQ and SPLENDID. Query processing and optimization techniques applied in each engine will be explained in chapter 4.

3.1 FedX

FedX is implemented in Java and extends the Sesame[5] framework with a federation layer[11]. FedX is a framework that enables querying on “heterogeneous, virtually integrated Linked Data Sources”[10].

FedX can be downloaded from the website[6]. The SPARQL endpoint is directly accessed via the terminal. This is done by introducing a *Data Config* file, which consist of URIs for the SPARQL endpoints. The user must know the data source’ URI in order to configure this connection. The Data Config is extensible, so the user can add new endpoints manually. With several URIs stored in Config File, FedX can federate queries against multiple data sources.

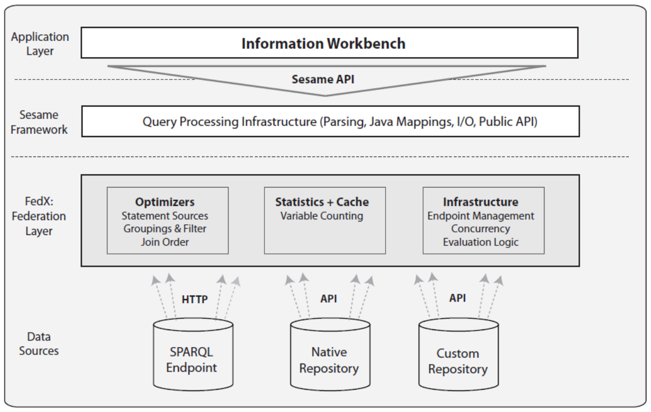

Fig. 3. FedX System Overview[11]

[5] https://rdf4j.org/
[6] https://www.fluidops.com/en/company/training/

In Figure 3, there is a visualized architecture of an application built on top of FedX. The application layer is necessary for any kind of interaction with the federation. This layer provides the frontend to the query processing engine. The second layer provides the frontend to the query processing engine and is composed of Sesame framework. This layer includes infrastructure for the following processes: query parsing, I/O components and the API for the client. The third layer, known as federation layer, is implemented as extension of Sesame. Besides basic features described above, FedX adds through this layer additional functionality for endpoint communication, data source management and optimization for distributed query processing[10][11]. Data Sources can be added to federation layer through repository mediator as visualized below.

3.2 DARQ

Distributed ARQ (DARQ) is a system to distribute SPARQL queries among multiple data sources[3][9]. DARQ gives the user an impression to query one single RDF graph despite the real data is being distributed in different sources. It was developed as an extension of ARQ[7] query engine[12]. DARQ executes a query against endpoints that support the SPARQL protocol[8]. The query federation process is hidden from the user.

A service description language enables DARQ to decompose the query into several sub-queries. Each of these sub-queries can be answered by an individual service. Service Descriptions contain needed information about the data sources, which will be used from the federated queries. The service description also inputs statistical information which is not a feature of other SPARQL endpoint engines. Defining statistical information about the data available can help the query optimizer to find a cost-effective query execution plan. Service Descriptions are represented in RDF. The use of *Service Description* is visualized in Figure 4.

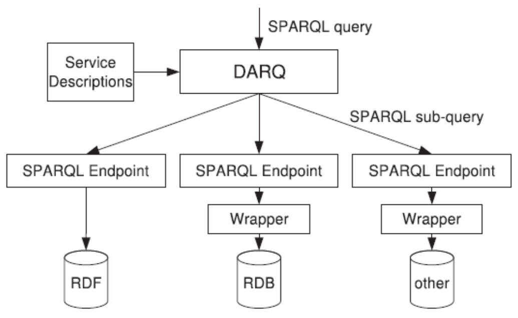

Fig. 4. DARQ-integration architecture[9]

[7] http://jena.apache.org/documentation/query/index.html
[8] https://www.w3.org/TR/sparql11-protocol/

Service Descriptions describe data in form of capabilities[4]. The definition of capabilities is based on predicates. Capabilities define what kind of triple patterns can be answered by the data source. Using this information, it is possible to say whether a query can be executed against the endpoint of this data set or not. Moreover, the Service Description has the the ability to limit the query. This is done through some access patterns which are included in the query. These access patterns are defined by *requiredBindings* property inside the query. DARQ federates a query against a data set if it satisfies all these requirements. Otherwise no result will be returned.

3.3 SPLENDID

SPLENDID is an engine, which federates queries based on voiD descriptions[15]. VOID (Vocabulary of Interlinked Datasets) incorporates statistical information and contains information about the URI of the SPARQL endpoint, triples and specific predicates in the data set. Most of existing federation engines assume an arbitrary level of details for statistics-based source selection, query optimization and query execution. On the contrary, SPLENDID relies only on the VOID statistics[13]. These statistics are created via a generator[9].

SPLENDID uses voiD description to determine which data set is relevant for a given triple pattern in the query. To achieve that, SPLENDID makes use of an a-priori knowledge about the data set. Since most endpoints offer a low availability, voiD descriptions are a better approach to get the required information.

As shown in Figure 5, SPLENDID consist of three main components:
a) Index Manager interacts with voiD descriptions. All the necessary information is saved for each endpoint.
b) Query Optimizer optimizes the query before it gets executed. Moreover, the relevant data sets for the triple patterns and the execution plan are determined.
c) Query Executor processes the data, retrieves and joins the results for different triple patterns.

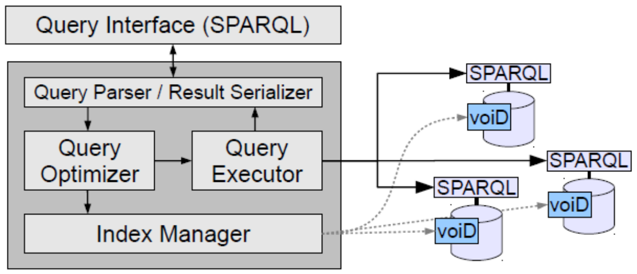

Fig. 5. Architecture of SPLENDID Federator[13]

[9] http://void.rkbexplorer.com/sparql

4 SPARQL Query Optimization

Optimizing SPARQL queries in federated setting with distributed data sources is crucial. It is important to guarantee a fast execution of the individual requests and to minimize the number of intermediate requests. There are two basic strategies to evaluate a SPARQL query in federated linked data: 1) all triple patterns are completely evaluated against every endpoint in federation and the query result is constructed locally in the server, and 2) an engine evaluates the query iteratively pattern by pattern, by evaluating the query in a nested loop join fashion. Below, I will cover these strategies and focus especially on the second one, based on three main engines: FedX, DARQ and SPLENDID. What optimization techniques are used specifically in each engine and which is the most effective one? Moreover, a comparison among these engines and an evaluation upon their performance is given in the last part.

4.1 Query Optimization in FedX

FedX [10] is an index-free SPARQL query federation system. The source selection is based on SPARQL ASK queries and a cache. This cache is used to save the recent SPARQL ASK operations for data source selection. The use of this cache significantly reduces the source selection and query execution time [4]. FedX implies the nested loop join fashion strategy for optimizing the queries. The problem with this approach is the number of remote requests caused for each join step. The intention of FedX is to minimize the number of join steps by grouping triple patterns and by reducing the number of sent requests [10].

According to [11], FedX processes a query in the following order:

1. Statement sources: All statements of SPARQL queries are examined for their relevant data sources to avoid unnecessary communication.
2. Filter pushing: SPARQL filter expressions are pushed down for early evaluation.
3. Parallel processing: Multithreaded execution of join and union computations.
4. Join order: Joins are important because they significantly influence the performance and thus the overall query runtime. FedX uses various heuristics to calculate the cost of each join and then executes them in ascending order of cost.
5. Bound joins: Joins are computed in a block nested loop fashion to reduce the number of requests and the overall time.
6. Groupings: Statement with relevant data source are executed in a single SPARQL query to push joins to the corresponding endpoint.

Figure 6 visualizes the FedX query processing model. First, the SPARQL query is parsed and transformed into an internal representation. Then, the **source selection** process takes place. The relevant sources are determined from the configured federation members using SPARQL ASK requests in conjunction with a cache. Triple patterns of a SPARQL query must evaluate only those data sources that are relevant to the results. In order to find relevant sources, FedX sends SPARQL ASK queries for each triple pattern to the federation members [4]. Based on the results, each pattern in the query is annotated with its relevant sources. Moreover, FedX uses a cache to remember whether source S is relevant or not for a triple pattern. This will minimize the number of remote ASK requests.

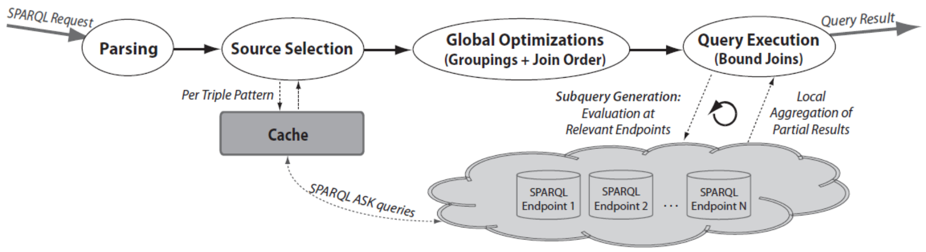
SPARQL Request
Parsing
Source Selection
Global Optimizations (Groupings + Join Order)
Query Execution (Bound Joins)
Query Result
Per Triple Pattern
Cache
Subquery Generation: Evaluation at Relevant Endpoints
Local Aggregation of Partial Results
SPARQL ASK queries
SPARQL Endpoint 1
SPARQL Endpoint 2
SPARQL Endpoint N

Fig. 6. Federated Query Processing Model[10]

The next optimization steps are **join order optimization** and forming of **exclusive groups**. These techniques have an important impact on the final execution plan. First, it is important to determine a suitable join order on the results of the different triple patterns. Determination of join order affects directly the computation time of the query. This is done by **heuristics-based cost estimation** [22]. A variation of algorithm presented in [22] is introduced in [10]. This algorithm implies an iterative approach to determine the argument with the lowest cost for remaining items and appends it to the result list.

Moreover, FedX extends this optimization technique by introducing exclusive groups. **Exclusive Groups** play a central role in FedX optimizer. Local execution of nested loop joins at the server causes a high cost in federated query processing. The role of Exclusive Groups is to minimize such a cost. An exclusive group of a data source is defined as a set of triple patterns, which only have this data source assigned [11][12]. Such a group is considered as one argument of the join operation and is preferred in the selection of the join order. Using the exclusive groups, all included triple patterns are merged in one query and then executed against the relevant endpoint. Without the application of such groups, each of the triple patterns could be distributed over the join order and consequently would increase the number of required requests. In paper [10], there is a concrete running example where exclusive groups are used.

After determining the join order, the query execution takes place. The join order shows the execution plan, meaning the order that different triples are executed against their relevant data sources. FedX uses an extended approach of **Block Nested Loop Join** [11]. According to this approach, FedX generates the result of one pattern and then matches the corresponding bindings to the second pattern.

Computing the joins in a block nested loop approach reduces the number of requests by a factor equivalent to the size of a *block*. The idea of this optimization is to group a set of mappings in a single sub-query using SPARQL UNION constructs. This sub-query is later sent to the relevant data sources in a single remote request. This technique is called **Bound Join** [10].

In addition, FedX provides even parallelization infrastructure [23], which uses multi-threading techniques to execute the joins, i.e, *bound joins* and union operators. This is possible because query processing in federated systems is highly parallelizable, meaning that different subqueries can be executed concurrently. Also, FedX has employed a pipelining approach which processes the intermediate results in the next operator as soon as they are ready. FedX and LHD are the only engines implementing parallelization as an optimizing technique.

4.2 Query Optimization in DARQ

DARQ [9] query engine can work as a SPARQL endpoint itself. Data sources are described by service descriptions as mentioned in the previous chapter. The query engine uses such information for query planning and optimization. According to [9], DARQ processes a query in the following steps:

1. Parse the query into a tree model of SPARQL.
2. Query Planning. The engine decomposes the query. Multiple sub-queries are built according to the information in the service descriptions.
3. Optimization. In this phase, the optimizer uses the sub-queries and constructs a optimized query execution plan.
4. Query Execution. The sub-queries are federated to different endpoints and the results are merged.

I have briefly explained how the first phase works while introducing DARQ engine. In the second phase, the information provided in the Service Description gets relevant. The purpose is to determine which sub-query gets distributed to a given data source. This is done by *Source Selection* process. An algorithm for finding the relevant data sources for a query is given in [9]. DARQ currently supports only queries with bound predicates, since the matching is based on predicates. The results retrieved from the *Source Selection* are used to build sub-queries that can be answered by the data sources.

DARQ engine provides some optimization techniques in order to improve the performance while federating queries over multiple data sets. To build a cost-effective query execution plan, the query optimizer uses logical and physical query optimization [12]. The aim of query processing is data minimization. Minimizing the federated data reduces the query processing time and improves the response time [16].

The logical optimization uses the rules given in [17] to rewrite the original query. This technique uses equalities of query expression to convert a logical query into an equivalent query plan that is executed faster. The original query is rewritten before query planning so the basic graph patterns can be merged. Variables will be replaced by constants from filter expressions.

Example 2. To better explain rewriting rules using FILTER expression, I am considering an example from BIO2RDF project. Listing 1.2 shows the original query submitted by the user. The user asks for a chemical named 2,6-xylidine, which is identified by the ontology *mesh: C007766*. This chemical is stored in Comparative Toxicogenomics Database. Using the dataset vocabulary, the user wants to know the functions of this chemical. So, the query asks for the functions that the 2,6-xylidine has in Comparative Toxicogenomics dataset.

Listing 1.2. Query before rewriting

```
PREFIX ctd_vocab:<http://bio2rdf.org/ctd_vocabulary:>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mesh: <http://bio2rdf.org/mesh:>

SELECT distinct ?chemical (str(?label) as ?Label)
WHERE {
 ?chemID ctd_vocab:has-function ?goFunction;
         rdfs:label ?chemical .
   FILTER (?chemID = mesh:C007766)
 ?goFunction rdfs:label ?label .
}
```

In Listing 1.2, there are two separate Basic Graph Patterns, each with one triple pattern. In Listing 1.3, there is given the same query, which is rewritten. In this case, the two patterns are merged. The variables that occur in filters with an *equal* operator are replaced. In this specific case, **?chemID** is replaced by the chemical URI (http://bio2rdf.org/mesh:**C007766**).

Listing 1.3. Query after rewriting

```
PREFIX ctd_vocab:<http://bio2rdf.org/ctd_vocabulary:>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT distinct ?chemical (str(?label) as ?Label)
WHERE {
 <http://bio2rdf.org/mesh:C007766>
                        ctd_vocab:has-function ?goFunction;
                        rdfs:label ?chemical .
 ?goFunction rdfs:label ?label .
}
```

Secondly, it is possible to move value constraints into the sub-queries to reduce the size of intermediate results[9]. Filters that have variables from more than one sub-query and that can not be splitted by a set of rules are applied locally in DARQ engine.

Example 3. In Listing 1.4, there is a query with a conjunctive filter on two attributes. I am considering a similar example as given previously. In this case, the user asks for *xylidine* chemical which has a *heme binding* function. Presumably, two triple patterns are split in two sub-queries for two different services. In this specific case, a single filter can not be moved into the sub-queries since one of the variables would be unbound. For this reason, the conjunction can be split into two filters: *FILTER (regex (?chemical, "xylidine")* and *FILTER regex (str(?label), "heme binding")*. These two filters can later be moved into the sub-queries. In case that the query optimizer is not able to split a filter using some sets of rewriting rules, a local filter will be applied. Such filtering will happen locally, inside DARQ query engine, as soon as all variables are bound.

Listing 1.4. Example SPARQL query

```
PREFIX ctd_vocab:<http://bio2rdf.org/ctd_vocabulary:>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT distinct ?chemical (str(?label) as ?Label)
WHERE {
   ?chemID ctd_vocab:has-function ?goFunction ;
            rdfs:label ?chemical .
   ?goFunction rdfs:label ?label .
  FILTER (regex (?chemical, "xylidine") &&
                   regex (str(?label), "heme binding"))
}
```

Physical Optimization focuses on a cost-based query optimization technique [12]. Network latency and bandwidth have the highest influence on query execution time, when we consider federated queries. Consequently, our goal is to reduce the transfer cost by reducing the amount of transferred data and the number of transmissions. This will considerably improve the query execution time.

Physical query optimization has the purpose to find the best query execution plan among all possible plans and uses a cost model to compare different plans [9]. This cost model is designed based on some statistical information in Service Descriptions [3]. The number of triples with a predicate shown in capability property of an endpoint is used to predict the size of a result for a given query [3].

Considering limitations in access patterns, we use iterative dynamic programming for optimization. Two join implementations are support in this approach:

- nested-loop join For every binding in the outer relation, we check the inner relation and add the bindings that match the join condition to the result set.
- bind join Introduced in [14]. It is a version of nested loop join, where intermediate results from the outer relation are passed to the inner to be used as filter. DARQ sends out the sub-query for the inner relation multiple times with the join variables bound[9][14]. We use such a binding join for data sources with limitations on access patterns. This approach reduces considerably the transfer costs if the unbound query returns a large result set.

4.3 Query Optimization in SPLENDID

SPLENDID [19] makes use of VOID descriptions to determine which data source is relevant for a specific triple pattern in the query. As mentioned in chapter 3, the main components of SPLENDID are *Index Manager, Query Optimizer and Query Executor* (see Figure 5). Before executing the main components, the query in SPLENDID gets converted into an abstract syntax tree[14]. Query Optimizer can handle such a query better.

Index Manager accumulates the statistics of VOID descriptions in a local index. General information such as triple count, the number of distinct predicates, subjects, and objects are stored as attributes for every SPARQL endpoint [19].

Query Optimizer optimizes the query before it gets executed. The SPLENDID query optimizer focuses on transforming the given query into a semantically equivalent query. This provides a lower cost in terms of processing time and communication overhead. Three steps are applied during the optimization: query rewriting, data source selection and cost-based join order optimization. The full algorithm is deeply explained in [13]. First, the *query rewriting* optimizes the logical tree structure of the query. As stated in [19], complex filter expressions are split and relocated close to operators which produce bindings for the filtered variables.

The *source selection* is done by using structures I_p: {(p, {d_i, c_i})} and I_t: {(t, {d_i, c_i})}. These structures are inverted indexes data structures[10], which are managed by the *Index Manager*. I_p includes the data set *d* and the triple count *c* for each predicate *p*. The voiD description of a data set *d* contains such information. Multiple data sets can have the same predicate, hence the structure will contain more tuples. This is done in I_t for the types like *rdf:type*.

In this paragraph, I will explain how the optimization algorithm works as it is stated in [13]. Each triple pattern with a bounded predicate can be mapped to the corresponding data set in I_p. Both data structures contain information only for bounded predicates. Consequently, patterns with no bounded predicate need to be mapped to all known data sources. SPLENDID added a new process to minimize the relevant data sources for a pattern. If a pattern has only one bounded predicate, the data set will return a result. In this case, SPLENDID uses ASK queries to check if the pattern gets results from the endpoint[19]. If it does not, the data set gets pruned from the corresponding data sets[13]. To reduce the amount of requests, patterns with common data sources are sent in a single query. The dynamic programming approach[20] is applied to determine the query execution plan (join order) for the Query Executor.

Join Order Optimization is a traditional optimization strategy which is used in relational databases, but can also be used in SPARQL basic graph patterns. Using the sub queries created by the source selection, all possible query execution plans are iterated, while the unwanted plans are pruned based on the overall cost estimation[19]. Query execution plans might have different tree structures, but the best choice for SPARQL queries are bushy trees[21].

Cost Function is used to compare two equivalent execution plans with different join orders and different join operators. Two formulas for calculating the transfer cost for hash join and bind join are given in [19]. The network communication has the highest influence in the overall cost. For this reason, the cost model basically includes the cost for sending queries to a SPARQL endpoint and the cost for receiving the results.

The execution plan with a minimal cost is passed for processing to the **Query Executor**. As provided in [19], SPLENDID uses two different join execution strategies:

1) "use the results of the first join argument to substitute unbound variables in the second join argument with a repeated evaluation for every binding."

[10] https://www.quora.com/Information-Retrieval-What-is-inverted-index

2) “request result tuples for the join arguments in parallel from the SPARQL endpoints to join them locally.”

In the first approach, the bind joins algorithm SPLENDID_B is used to join the results[14]. It is suited to reduce the network overhead if the selectivity of the join variable is high and the result set is large. The second approach executes the two arguments in parallel and then joins them locally by using the hash join algorithm[13] SPLENDID_H. It is very good for retrieving small result sets that can be joined locally.

4.4 Evaluation

In this subsection, I will briefly make an evaluation review based on four metrics: 1) total triple pattern-wise sources selected, 2) total number of SPARQL ASK requests used during source selection, 3) source selection time, and 4) query execution time. In paper [4], you can find a detailed overview how these criteria affect the overall performance. My aim is to make a comparison among the engines explained above and show how they respond to each criteria.

In Table 1, there are given the most important characteristics of each engine explained previously. As a result of these observations, I can state that DARQ and SPLENDID consist on many similarities. They use dynamic programming for finding the best execution plan and a source selection approach without preprocessing. However the preprocessed information is retrieved from different components, respectively Service Descriptions and Void Descriptions. On the contrary, FedX provides an index-free source selection approach by applying a user generated Data Config file to determine the relevant data sources. FedX uses a heuristic-based cost estimation instead of dynamic programming. All the engines apply grouping techniques such as Exclusive Groups to optimize the query execution process. Using these features, I will explain how each engine responds to the metrics I am interested in.

Table 1. Comparison among the engines

	FedX	DARQ	SPLENDID
Source Selection	Data Config + ASK queries + Cache	Service Descriptions	VOID Descriptions + ASK queries
Source Selection approach	Index-free	Index-only	Index-based
Preprocessing	No	Yes	Yes
Query execution plan	Heuristic-based cost estimation	Dynamic programming	Dynamic programming
Query rewriting	No	Yes	Yes
Join order optimization	Nested-loop + Bind join	Nested-loop + Bind join	Hash-loop + Bind join

My scope of evaluation stands only on three engines: FedX, DARQ and SPLENDID. Considering the *triple pattern-wise selected sources*, FedX and SPLENDID overpass DARQ in terms of source selection accuracy. This happens because both FedX and SPLENDID make use of ASK queries when any of subject or object is bound in a triple pattern. On the other side, DARQ is an index-only approach which does not use SPARQL ASK queries when any of the subject or object is bound. Thus, DARQ tends to overestimate the triple pattern sources per individual triple pattern.

In terms of *SPARQL ASK requests*, DARQ as an index-only approach can only make use of its index to perform the source selection. Therefore, DARQ does not need any ASK request to process queries. FedX on the other side, uses only ASK requests along with a cache to perform source selection. If FedX uses a full cached approach, the complete source selection is performed by using cache entries. Consequently, the use of a cache improves significantly the source selection time and the overall query execution time. SPLENDID is the most efficient engine if we consider the SPARQL ASK requests consumed during source selection. It overpasses FedX, since the latter one is closely dependent by the use of the cache, which is not always secured.

The number of SPARQL ASK requests grows and increases the overall *source selection time*. Thus, we can expect a trade-off between the source selection process and the time required to perform it. DARQ needs less time compared to the other engines, since it does not send any SPARQL ASK queries during the source selection process. The runtime in index-only approaches will be minimal because the index is usually pre-loaded in memory before the query execution happens. FedX uses a cache to store the results of the recent SPARQL ASK operations. This reduces the source selection time in FedX, but makes it highly dependent on the cache. In SPLENDID, the source selection time is high, since it overestimates the number of triple pattern sources.

Based on the criteria mentioned so far, FedX outperforms the others in terms of *query execution time.* FedX uses parallel execution of SPARQL ASK queries and caching to perform the source selection process. This parallelization of processes is very time-efficient. Other factors such as the join type, join order selection and the buffer sizes can affect the query execution time. But the most influential one is the level of source selection caching which is used only in FedX.

5 Summary and Future Work

In this section, I summarize the main points discussed in the report and state some possible future improvements. Many mechanisms accessing Linked Data are invented lately. I basically covered the so called Federated Query Processing and studied the main approaches used to federate queries. SPARQL Endpoint Federation is the most popular approach for accessing federated queries. I introduced the main obstacles and showed why query optimization is crucial to overpass such problems. My intention was focused on three well-known SPARQL Endpoint engines: FedX, DARQ and SPLENDID. The purpose of this paper was to explain the main optimization techniques implied in each engine. I tried to analyze the performance of each engine and compare them in terms of query execution time as key criterion. In the next paragraphs, I shortly recapitulate the main features of each engine and discuss possible future improvements.

FedX is a practical solution for querying multiple distributed Linked Data sources. Various optimization techniques are implemented by this engine in order to reduce the overall query runtime. Bound joins combined with grouping and source selection increase the performance. Compared to the other state-of-the-art systems, FedX increases the query performance by minimizing the number of intermediate requests. FedX is compatible with SPARQL 1.0 query language, allowing clients to integrate SPARQL endpoints into a federation without any preprocessing.

By the integration of SPARQL 1.1, further improvements are expected. For example, federation extensions by using the SERVICE operator will allow the engine to directly specify the data sources. Also, statistics may influence the performance drastically. FedX has not yet implemented any local statistics. Inspired by the SPLENDID experience, FedX might incorporate remote statistics using voiD descriptions [15] in the future.

DARQ is compatible to any SPARQL endpoint. Using Service Descriptions provides a way to dynamically add and remove endpoints to the query engine in a transparent way. I introduced basic query optimization for SPARQL, which help reducing the execution costs. The techniques explained above can drastically improve query performance and allow distributed answering of SPARQL queries over multiple sources in a reasonable time. The optimization algorithm relies on small amount of statistical information. Some future improvements can be done in mapping and vocabulary translation. This is important because multiple data sources are using different vocabularies and non-unique representation. Further improvements are also possible as explained in [14][18].

SPLENDID can achieve a good query performance compared to the other state-of-the-art federation implementations. Data source selection and query optimization is based on basic statistical information which is obtained from VOID descriptions. Hash join and bind join can reduce the processing time for certain types of queries.

Anyway, this approach can be further improved. It is possible to extend VOID descriptions with more detailed statistics in order to better query the execution plans. Moreover, the query execution can be optimized by using the UNION operator, an optimization technique used in FedX[13]. Also, the adoption of SPARQL 1.1 federation extension will improve the query execution efficiency.

6 Conclusion

To sum up, I will give my personal opinion about the topic. Also, I will conclude by suggesting the most important factors that must be considered while building the next federation engines.

Many approaches to federate Linked Data are developed lately. In my opinion, Triple Pattern Fragments must be considered as an innovative approach, since it alleviates the server from the whole processing effort. On the other side, SPARQL endpoints require additionally processing power per each new client requesting data from the endpoint. By using Triple Pattern Fragments, the client will take care of its own processing. Live Linked Data Streams are a unique approach to federate streaming data over various data sets. Personally, I think that this approach can positively overpass the problem of outdated information.

Considering the federation engines, I think that a smart source selection, in terms of triple pattern sources and execution time is the key element to be considered while building next SPARQL Endpoint federation engines. Inspired by FedX, I can sum up that a combination of caching with ASK queries to perform the source selection will improve the overall runtime of SPARQL query processing system. Other factors such as the join type, join order selection and the buffer sizes can affect the query execution time and must be considered too.

References

1. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (January 2008) http://www.w3.org/TR/rdf-sparql-query/.
2. Hartig O.: An Overview on Execution Strategies for Linked Data Queries (2013)
3. Haase, P., Math, T., Ziller, M.: An Evaluation of Approaches to Federated Query Processing over Linked Data
4. Saleem, M., Ngonga Ngomo, A., Khan, Y., Hasnain, A., Hauswirth M., Ermilov, I.: A Fine-Grained Evaluation of SPARQL Endpoint Federation Systems
5. Verborgh, R., Vander Sande, M., Colpaert, P., Coppens, S., Mannens, E., Van de Walle, R.: Web-Scale Querying through Linked Data Fragments
6. Van Herwegen, J., Vborgh, R., Mannens, E., Van de Walle R.: Query Execution Optimization for Clients of Triple Pattern Fragments
7. F. Barbieri, D., Della Valle, E.: A proposal for Publishing Data Streams as Linked Data
8. Francesco Barbieri, D., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for Continuous Querying
9. Quilitz, B., Leser, U.: Querying Distributed RDF Data Sources with SPARQL
10. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization Techniques for Federated Query Processing on Linked Data
11. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: A Federation Layer for Distributed Query Processing on Linked Open Data
12. Buil-Aranda, C.: Federated Query Processing for the Semantic Web
13. Grlitz, O., Staab, S.: SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions
14. M. Haas, L., Kossmann, D., L. Wimmers, E., Yang, J.: Optimizing Queries across Diverse Data Sources. In: 23rd Int. Conference on Very Large Data Bases (VLDB), San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1997)
15. L. Alexander, R. Cyganiak, M. Hausenblas, J. Zhao. 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*, Madrid, Spain, 2009.
16. Saleem, M., Ngonga Ngomo, A., Xavier Parreira, J., F. Deus, H., Hauswirth M.: DAW: Duplicate-Aware Federated Query Processing over the Web of Data
17. Perez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: 4th International Semantic Web Conference (ISWC), Athens, GA, USA (November 2006) 30-43
18. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. 32(4) (2000) 422-469
19. Gorlitz, O., Staab, S.: SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions
20. P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. in *Proceedings of the 13th International Conference on Management of Data*, pages23-24, Boston, MA, USA, 1979
21. M.E. Vidal, E. Ruckhaus, T. Lampo, A. Martinez, J. Sierra, and A. Pollers. Efficiently Joining Group Patterns in SPARQL Queries. In *7th Extended Semantic Web Conference*, pages 228-242, Heraklion, Crete, Greece, 2010. Springer.
22. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation
23. Wang, X., Tiropanis, T., C. Davis, H.: LHD: Optimizing Linked Data Query Processing Using Parallelisation