The Knowledge Engineering Review, Vol. 30:5, 545–563.
doi:10.1017/S0269888915000107

Federated query processing on linked data: a qualitative survey and open challenges

DAMLA OGUZ[1,2,3], BELGIN ERGENC[1], SHAOYI YIN[3], OGUZ DIKENELLI[2] and ABDELKADER HAMEURLAIN[3]

[1]*Department of Computer Engineering, Izmir Institute of Technology, 35430 Izmir, Turkey;*
e-mail: damlaoguz@iyte.edu.tr, belginergenc@iyte.edu.tr;
[2]*Department of Computer Engineering, Ege University, 35100 Izmir, Turkey;*
e-mail: oguz.dikenelli@ege.edu.tr;
[3]*IRIT Laboratory, Paul Sabatier University, 31062 Toulouse, France;*
e-mail: shaoyi.yin@irit.fr, abdelkader.hameurlain@irit.fr

Abstract

A large number of data providers publish and connect their structured data on the Web as linked data. Thus, the Web of data becomes a global data space. In this paper, we initially give an overview of query processing approaches used in this interlinked and distributed environment, and then focus on federated query processing on linked data. We provide a detailed and clear insight on data source selection, join methods and query optimization methods of existing query federation engines. Furthermore, we present a qualitative comparison of these engines and give a complementary comparison of the measured metrics of each engine with the idea of pointing out the major strengths of each one. Finally, we discuss the major challenges of federated query processing on linked data.

1 Introduction

The term linked data is used to refer to the way of publishing and connecting structured data on the Web by using semantic Web technologies. Thus, it makes the Web as a huge global data space which is referred as Web of data. In order to create such a global data space with different data providers, data must be opened and linked according to some rules. These rules which are known as linked data principles are defined by Berners-Lee (2006). As understood from these rules, Uniform Resource Identifier (URI), Hypertext Transfer Protocol (HTTP), Resource Description Framework (RDF) and RDF Query Language (SPARQL) are the building blocks of linked data. These arguments are explained in detail in other papers (Bizer, 2009; Bizer *et al*., 2009; Hartig & Langegger, 2010).

Once a distributed data space is created, one important and difficult question arises: How can we automatically query this huge data space? According to the data source location, linked data querying infrastructure can be categorized as central repository and distributed repository (Rakhmawati *et al*., 2013). In central repository infrastructure, all the data in different sources are aggregated in a single repository and then query processing is executed. Although this infrastructure provides efficient query processing, the data is not always up-to-date and adding a new source is a difficult task. In distributed repository infrastructure, the query is executed on the distributed data sources and then the results are aggregated. As a result, the data is more up-to-date.

There are two approaches for query processing on distributed repository; link traversal and query federation. The first approach, link traversal (Hartig *et al*., 2009) is also called follow-your-nose and can be defined simply as discovering potentially relevant data by following links between data. Related data

sources are discovered during the query execution without any data knowledge. One of the well-known examples of link traversal approach is SQUIN (Hartig *et al*., 2009; Hartig, 2013). In this concept the data sources are RDF documents and the intermediate results are augmented with bindings for the common variables. Besides, Fionda *et al*. (2012) proposed a navigational declarative language for Web of data.

The major advantage of link traversal is providing up-to-date results and using the potential of the Web by discovering data sources at runtime. However, this approach has some remarkable weaknesses. The results can change according to the starting point, which is a triple pattern, and a wrong starting point can increase intermediate results. Although Hartig (2011) employed some heuristic query planning methods to SQUIN, the mentioned weaknesses cannot be solved. In other words, this approach cannot guarantee finding all results because the relevant data sources change according to the starting point. Furthermore, URI dereferencing is a time-consuming task.

The second approach is named as the query federation (Görlitz & Staab, 2011a). It is based on dividing a query into subqueries and distributing them to the related data sources. This process is employed via an engine. The infrastructure is similar to mediation system architecture (Wiederhold, 1992), thus the engine is also called the mediator. There are two main advantages of query federation. One of them is providing up-to-date results and the other one is the capability of guaranteeing finding all results. On the other hand, queries are executed on SPARQL endpoints. For this reason, in order to process a query, SPARQL endpoints of each data sources are needed. This can be accepted as a shortcoming of this approach. However, Rakhmawati *et al.* (2013) remarked that 68.14% of data sources provide linked data through their SPARQL endpoints. We think that this number is increasing day by day.

The common advantage of these approaches is providing up-to-date results due to processing queries on the actual data sources. However, link traversal does not guarantee complete results and has some performance problems. Because of these reasons, we turn our attention to the second approach. There are two surveys (Rakhmawati *et al.*, 2013; Saleem *et al.*, 2015) on this approach that discuss some of the systems in query federation. Saleem *et al.* (2015) also presented quantitative evaluations such as query execution time and data source selection time. However, neither of them covers query optimization strategies in detail.

As indicated above, to the best of our knowledge, there are not any detailed syntheses on query federation on linked data in the literature in which the query processing steps of different systems are introduced and compared. In this paper, we propose a survey of federated query processing on linked data. We synthesize the existing data source selection methods, join methods and query optimization methods in federated query processing through surveying the promising federated query engines such as DARQ (Quilitz & Leser, 2008), FedX (Schwarte *et al.*, 2011), SPLENDID (Görlitz & Staab, 2011b), ANAPSID (Acosta *et al.*, 2011), ADERIS (Lynden *et al.*, 2010, 2011), LHD (Wang *et al.*, 2013) and WoDQA (Akar *et al.*, 2012; Yönyül, 2014). We also provide a qualitative comparison of these studies. In addition, we identify the challenges of federated query processing on linked data.

The main objectives of this paper are to provide a literature survey about federated query processing on linked data by synthesizing the data source selection, join methods and query optimization methods of query federation engines; and identify the major challenges in this research area.

The rest of the paper is organized as follows: Section 2 presents the principles of query federation and the steps in federated query processing. Section 3 states the data source selection methods in the existing query federation engines. Section 4 synthesizes join methods in federated query processing while in Section 5 we review the query optimization methods. Section 6 provides a discussion on query federation with a qualitative comparison of the existing engines. Furthermore, it presents the challenges of federated query processing with future directions. Finally, we conclude the paper in Section 7.

2 Federated query processing

Query federation architecture is similar to mediator-wrapper architecture (Wiederhold, 1992) in integrating the information from different data sources via a mediator. However, they are different from each other in accessing the data sources. In mediator-wrapper architecture, wrappers are used to access the data sets due to the heterogeneous data models, whereas in query federation SPARQL endpoints are used

to access the data sources without wrappers due to the common data model (RDF). Actually, in query federation, each user query is decomposed into subqueries and directed to the SPARQL endpoints of the selected data sources to be executed. The results of the endpoints are combined and finally returned to the user. A subquery in this context is a triple pattern or a set of triple patterns.

A typical federated query processing engine contains four different steps with following functions: query parsing, data source selection, query optimization and query execution. Figure 1 shows the structure of a federated query engine. User sends the query to the engine and gets the results, whereas several steps are involved in query federation. Query parsing checks and transforms a SPARQL query into a query tree as an input to the query optimizer. In data source selection, the relevant data sources for each triple pattern or set of triple patterns of a query are determined. The subqueries and intermediate results are transmitted over the Web of data. Hence, query optimization is substantially important in query federation. The fundamental responsibilities of query optimization are grouping the triple patterns, deciding the join strategy and ordering the triple patterns. Query execution part is dedicated to the execution of the query operators defined by the optimizer and preparation of the result set.

In the following sections, we synthesize the data source selection, join methods and query optimization methods in query federation by surveying the promising engines in the literature. We do not include the details of parsing step as it is a generic process that aims to check syntax and semantic correctness of the user query and to transform it to a normalized algebraic query.

3 Data source selection

We have classified the data source selection methods in federated query processing as follows: (i) predicate-based selection, (ii) type-based selection, (iii) rule-based selection and (iv) SPARQL ASK queries. All the methods except the last method need metadata catalogs. For this reason, we first discuss the metadata catalogs.

3.1 Metadata catalogs

The existing query federation engines use three types of metadata catalogs: (i) service descriptions (Quilitz & Leser, 2008), (ii) VoID (Vocabulary of Interlinked Datasets) descriptions (Alexander & Hausenblas, 2009)

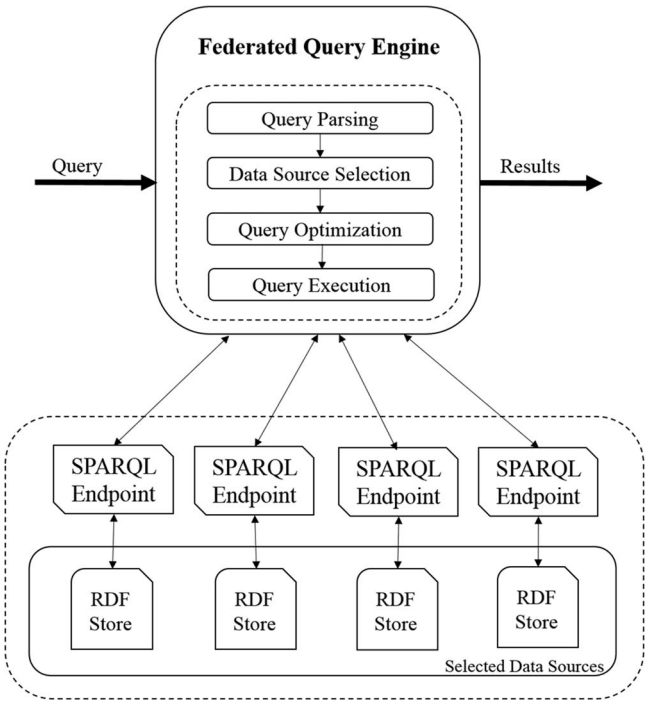

Figure 1 Federated query processing.

and (iii) list of predicates. Before we begin, we want to remark that data set and data source are used interchangeably.

- *Service description*: provides metadata about the RDF data and cover some statistics about this data. It states the potential capacity of a data source by containing statistics; such as number of triples with a predicate and total triples. In other saying, a service description takes into account the information about the data set, which means a set of RDF triples which is published by a single provider.
- *VoID description*: provides metadata about the RDF data and cover some statistics about this data like service descriptions. In addition to the information about data sets, there is another concept in VoID which is called linkset. A linkset describes a set of RDF triples where all subjects refer to one data set and all objects belong to another data set (Alexander & Hausenblas, 2009). Thus, VoID can be used to describe the metadata of RDF data sets with the interlinking to other data sets. In addition to these, statistics about the data sets can be defined in VoID descriptions as service descriptions. Number of triples and number of instances of a class or property are some examples of the statistics here. Cyganiak *et al.* (2011) proposed a VoID guide to data publishers or consumers.
- *List of predicates*: list of predicates is also used as a metadata catalog.

3.2 Data source selection methods

As indicated in the beginning of this section, according to our classification there are four data source selection methods of query federation engines as follows:

- *Predicate-based selection:* selects the relevant data sources of a triple pattern by matching its predicate with the covered predicates in the metadata catalog.
- *Type-based selection:* selects the relevant data sources according to the type definitions (*rdf:type*) in the metadata catalog.
- *Rule-based selection:* selects the relevant data sources according to the defined rules which are generated by analyzing the relations between the triple patterns of a query.
- *SPARQL ASK queries:* selects the relevant data sources by sending SPARQL ASK queries to the endpoints. If the result of the query is TRUE, this data source is selected as a relevant data source.

DARQ uses service descriptions for its metadata catalog and these descriptions must be created before using the query engine. It employs predicate-based data source selection. Therefore, the engine cannot support unbound predicate triple patterns.

SPLENDID uses VoID descriptions in data source selection. Data sources are indexed for every predicate and type by using VoID statistics. However, the statistics in VoID descriptions can be insufficient to select a triple pattern's relevant data source or data sources. This situation exists especially for the triples with common predicates such as *rdfs:label*. Almost all data sets use this predicate, thus SPLENDID sends SPARQL ASK queries for the triples with bound variables which are not covered in VoID descriptions. All data sources are selected for the triple patterns which have unbound predicates.

LHD (Wang *et al.*, 2013) is another federated query engine that uses VoID descriptions together with SPARQL ASK queries in data source selection. It first uses VoID descriptions and then sends SPARQL ASK queries to refine the selected data sources. Its data source selection is based on predicates as DARQ. However, it can support unbound predicates without eliminating irrelevant data sources such as SPLENDID.

WoDQA, which was first proposed by Akar *et al.* (2012) and then was enhanced by Yönyül (2014), also uses VoID descriptions and SPARQL ASK queries in data source selection. Akar *et al.* (2012) stated that predicate-based and type-based selection is not enough to eliminate all irrelevant data sources due to having common predicates or types. Hence, Akar *et al.* (2012) proposed different rules based on query pattern analysis. These rules are proposed with three perspectives; IRI-based analysis, linking analysis and shared variable analysis. IRI-based analysis selects the relevant data sources by matching the IRIs in the triple pattern with the *void:uriSpace* and *void:vocabulary* properties of VoID descriptions. Furthermore, IRI-based analysis includes the predicate-based selection and type-based selection methods as well. By this means, WoDQA does not only select the data sources according to the predicate or type of a query,

Table 1 Data source selection methods in query federation

	Predicate-based selection	Type-based selection	Rule-based selection	SPARQL ASK query
DARQ	+			
FedX				+
SPLENDID	+	+		+
ANAPSID	+			
ADERIS	+			
LHD	+			+
WoDQA			+	+

it considers all the IRIs in a query. In linking analysis, WoDQA takes into consideration the linkset definitions in the VoID descriptions. Lastly, in the shared variables analysis, WoDQA considers that triple patterns with shared variables can affect their related data sources. In other words, shared variables analysis aims to eliminate the irrelevant data sources.

As stated previously, list of predicates can also be used as a metadata catalog. ANAPSID (Acosta *et al.*, 2011) keeps a list of predicates, the execution timeout properties of the endpoints and the statistics as metadata catalog. The endpoints' execution timeouts and statistics are collected by an adaptive sampling technique (Vidal *et al.*, 2010; Blanco *et al.*, 2012) or during the query execution. It also updates the statistics on the fly. ANAPSID uses predicate-based selection and chooses the endpoints whose timeout is longer than the triple patterns estimated execution time. However, the details are not given in their publication so it is not clear how ANAPSID estimates the triple patterns' execution times. ADERIS (Lynden *et al.*, 2010, 2011) also uses list of predicates as metadata catalog and predicate-based selection for data source selection. It sends SPARQL SELECT queries with DISTINCT keyword to each endpoint to find out the unique predicates. Besides, ADERIS adds data sources manually when it is impossible to do that automatically[1].

FedX (Schwarte *et al.*, 2011) sends SPARQL ASK queries for each triple pattern in order to decide if the intended query can be answered by the endpoint or not and it also caches the relevance of each triple with each data source.

Table 1 shows the data source selection methods in the existing query federation engines. As seen from the table, predicate-based selection and sending SPARQL ASK queries are the common methods. FedX just sends SPARQL ASK queries to select the data sources. Although SPLENDID uses VoID descriptions to select the data sources based on predicates and types, it sends SPARQL ASK queries when the descriptions cannot help to select the relevant data sources. DARQ selects the data sources based on predicates via service descriptions. However, it does not send SPARQL ASK queries when its metadata catalog fails to select the data sources. LHD employs predicate-based selection via VoID descriptions and sends SPARQL ASK queries in order to eliminate the irrelevant data sources. ANAPSID and ADERIS use predicate-based selection as DARQ. However, ANAPSID also considers the execution timeout information of endpoints as well. Different from other engines, WoDQA uses rule-based selection by query pattern analysis and considers the linksets in VoID descriptions. Hence, it eliminates all the irrelevant data sources before the query execution by considering the relations between triple patterns. It also uses SPARQL ASK queries.

The data source selection methods in query federation show that using a metadata catalog is a need. Without a metadata catalog, data source selection becomes quite difficult and time consuming due to the need of preprocessing for each query. This time-consuming task usually is done by SPARQL ASK queries. However, first employing a metadata catalog and then when the selection cannot be done through the catalog, using SPARQL ASK queries seems a better strategy for data source selection in federated query processing.

[1] `http://code.google.com/p/sparql-aderis/`

In addition to the mentioned data source selection methods, Saleem and Ngomo (2014) proposed an approach for data source selection as well. We have not included them to our classification because it is a data source selection approach, not a query federation engine.

4 Join methods

In federated query processing, query optimization is done after data source selection. Query optimization covers subquery building, join method selection and join ordering. Following the query optimization, the last step in federated query processing is the query execution. In this step, subqueries are executed in the SPARQL endpoints of the selected data sources by using the chosen join methods in query optimization. In other words, selection of join methods for query execution is related to query optimization. In order to improve the coherence of the paper, we present the join methods in this section and discuss the query optimization in the next section.

Five different join methods are used in the existing federated query processing engines: (i) bound join, (ii) nested loop join, (iii) hash join, (iv) symmetric join and (v) multiple hash join.

4.1 Bind join

Bind join was introduced by Haas *et al*. (1997). The method passes the bindings of the intermediate results of the outer objects to the inner object in order to filter the result set. Its execution is similar to the nested loop join and substantially efficient when the intermediate results are small (Haas *et al*., 1997).

Bind join, also called bound join, is commonly used in federated query processing. Schwarte *et al*. (2011) proposed a bound join technique for FedX which uses SPARQL UNION[2] constructs to group a set of mappings in a subquery to be sent to the relevant data sources in a single remote request. The original mapping is kept in order to retain correctness of the final results. WoDQA (Yönyül, 2014) uses bound join method as well. However, FedX uses SPARQL UNION constructs whereas WoDQA prefers SPARQL FILTER[3] expression for bound join. In addition, SPARQL 1.1 Query Language[4] (March 2013) proposes SERVICE[5] keyword to explicitly execute certain subqueries on different SPARQL endpoints and WoDQA takes the advantage of SERVICE keyword in its bound join.

DARQ (Quilitz & Leser, 2008), ANAPSID (Acosta *et al*., 2011), ADERIS (Lynden *et al*., 2011), LHD (Wang *et al*., 2013) and SPLENDID (Görlitz & Staab, 2011b) employ bound join as well. We will talk about their usage later. Different from other engines, DARQ employs bound join when the data sources have limitations on access patterns. For this reason, DARQ keeps the definition of limitations on access patterns in service descriptions.

4.2 Nested loop join and hash join

Nested loop join method is used by Quilitz and Leser (2008) for query execution in DARQ when there is no limitation on access patterns. The inner relation is scanned for every binding in the outer relation and the bindings which provide the join condition are included in the result. ADERIS applies index nested loop join method in query execution which uses an index on join attributes.

Hash join is another join method used in federated query processing. Görlitz and Staab (2011a) implemented a hash join method for SPLENDID which requests the results of the join argument in parallel and joins them locally. Wilschut and Apers (1991) stated that although hash join method is symmetric conceptually, it is asymmetric in it operands.

4.3 Symmetric join

Symmetric hash join (Wilschut & Apers, 1991) is a non-blocking hash-based join which supports pipelining in parallel database systems. It maintains a hash table for each relation. In other words,

[2] http://www.w3.org/TR/rdf-sparql-query/#alternatives
[3] http://www.w3.org/TR/sparql11-query/#expressions
[4] http://www.w3.org/TR/sparql11-query/
[5] http://www.w3.org/TR/2013/REC-sparql11-service-description-20130321/

Table 2 Join methods in query federation

		NLJ				
	Bound join	Simple NLJ	Index NLJ	Hash join	Symmetric join	Multiple hash join
DARQ	+	+				
FedX	+					
SPLENDID	+			+		
ANAPSID	+				+	
ADERIS	+		+			
LHD	+					+
WoDQA	+					

NLJ = nested loop join.

symmetric hash join creates two hash tables instead of generating a single hash table as in hash join method. Thus, symmetric hash join is a non-blocking join method which produces the output of tuples as early as possible. However, it requires more memory due to keeping the hash tables in memory.

The double pipelined hash join (Ives *et al.*, 1999) and the XJoin (Urhan & Franklin, 2000) are the extended versions of symmetric hash join. Different from the symmetric hash join, double pipelined hash join adapts its execution when the memory is insufficient and XJoin moves some parts of hash tables to the secondary storage when the memory is full.

Acosta *et al.* (2011) proposed a non-blocking join method, called adaptive group join (agjoin), which is based on symmetric hash join and XJoin. This join method produces incremental results by presenting a data structure called resource join tuple (RJT), which records the join values and the tuples that are matched. By this means, ANAPSID can produce results even when an endpoint becomes blocked and can hide delays from users. Adaptive dependent join (adjoin) is the extended version of dependent join (Florescu *et al.*, 1999). It sends requests to the data sources in an asynchronous fashion and hides delays from the user. Specifically, it sends the request to the second data source when tuples from the first source are received. Actually, adjoin can be accepted as a bound join because it needs the bindings. Both agjoin and adjoin flush to the secondary memory when the memory is full as XJoin does. In addition, when the both endpoints become blocked, the RJTs in the first and second memory are joined.

4.4 Multiple hash join

The execution system of LHD (Wang *et al.*, 2013) is similar to double pipelined hash join (Ives *et al.*, 1999) and XJoin (Urhan & Franklin, 2000). It uses multiple hash tables at a node. Wang *et al.* (2013) stated that a node can be a subject or an object, while an edge refers to triple patterns. Also a data stream was defined as the results of triple patterns of an edge. LHD uses multiple hash tables in order to join more than one data stream at the same time. The result of a stream is stored in its hash table and it is probed against the hash tables of other streams simultaneously. On the other hand, LHD employs bound join when pre-computed bindings are used. Wang *et al.* (2013) proposed a join operator that separates the input bindings via a hash table on the dependent variable or variables. If there is only one binding in the query, the variables in the query is replaced by the values of the binding. Otherwise, the bindings are specified with VALUES[6] syntax.

Table 2 shows the join methods in federated query processing on linked data. All the engines employ bound join. DARQ, SPLENDID, ANAPSID, ADERIS and LHD provide two different join methods whereas FedX and WoDQA implement one join method. Different from others, ANAPSID uses a non-blocking join method which is an extended version of symmetric hash join and XJoin. However, it uses its own data structure instead of hash tables. For this reason, we name this method as symmetric join.

[6] `http://www.w3.org/TR/sparql11-query/#inline-data`

As the final remark of this section, we would like to mention three papers which are related to the efficiencies of query federation engines. Buil-Aranda *et al.* (2013) described and formalized the syntax of federation extension for SPARQL 1.1, whereas Buil-Aranda *et al.* (2014) focused on the limitations of SPARQL servers when using SPARQL 1.1. They showed that SPARQL servers can limit the amount of result sizes and these limitation can be overcome by using SERVICE patterns in queries. They also discussed different evaluation strategies and showed that results of all queries in all systems under test are returned by using only FILTER and 'symmetrical hash join with pagination of the remote results' strategies. Saleem *et al.* (2013) proposed a duplicate-aware approach, namely DAW, which can be adapted to query federation engines. They extended DARQ, SPLENDID and FedX with this approach, and compared the engines with their extended versions. They showed that query execution times are reduced in most cases by extending the engines with the proposed approach. The engines should consider the limitations mentioned by Buil-Aranda *et al.* (2014) and can use a duplicate-aware approach like DAW (Saleem *et al.*, 2013) to increase their efficiencies.

5 Query optimization

After synthesizing the data source selection and the join methods in query federation, in this section we will discuss the query optimization methods in query federation. The objective of the federated query engines can be stated as to minimize the response time which includes communication time, I/O time and CPU time in federated query processing. The communication time generally dominates the others and it is directly proportional to the amount of intermediate results. And the number of intermediate results are substantially affected by join order.

Query optimization is not only affected by the join order but also by the join method. Therefore, join method selection is one of the essential parts of query optimization in federated query processing. Furthermore, the number of sent HTTP requests affects the communication cost, too. For this reason, grouping the appropriate triple patterns and sending them together to the related endpoint decrease the communication time as well.

Consequently, there are three decisions in federated query optimization which are subquery building, join method selection and join ordering. In the following of this section, we discuss these decisions.

5.1 Subquery building

A SPARQL query is comprised of triple patterns. A subquery can be composed of a triple pattern or a set of triple patterns. Therefore, subquery building refers to grouping the triple patterns of a query in order to decrease the number of HTTP requests and intermediate results. We classify the subquery building methods used in query federation engines as follows: (i) exclusive grouping (EG), (ii) exclusive grouping considering shared variables (EGSV) and (iii) *owl:sameAs* grouping (SAG). We first define these methods and then explain their roles in the existing query federation engines.

- *EG*: a heuristic which groups the triple patterns that refer to the same data source with a provision that there must be one and only one relevant data source (Schwarte *et al.*, 2011). The grouped triple patterns are called exclusive groups. In other words, the triple patterns in an exclusive group must refer to a single data source. This heuristic aims to reduce both HTTP requests and intermediate results.
- *EGSV*: extended version of the EG heuristic which creates different exclusive groups for the triple patterns without shared variables. EG method can group triple patterns which do not have shared variables and it causes redundant intermediate results.
- *SAG:* creates a subquery for the triple pattern which has *owl:sameAs* predicate with an unbound subject variable and the triple pattern which has the same unbound variable. This method is used when there is an assumption that this predicate is used in order to indicate the internal resources of a data set.

The idea behind the EG was proposed by Quilitz and Leser (2008) for DARQ. However, the method was titled as exclusive grouping by Schwarte *et al.* (2011) for FedX. Although ANAPSID (Acosta *et al.*, 2011) does not use the name of EG, it groups the triple patterns which refer to the same endpoint.

SPLENDID (Görlitz & Staab, 2011b) uses both EG and SAG. The assumption about the SAG here is that all data sources describe *owl:sameAs* links for their data. This grouping can be realized manually when third-party data sets with external *owl:sameAs* links do not exist in the federation. WoDQA is the only engine which uses EGSV.

After data source selection, ADERIS (Lynden *et al.*, 2010, 2011) generates predicate tables which include subject and object of triples associated with each unique predicate in the intended query. In order to create these tables, the engine sends triple patterns to data sources by considering the predicates contained in each data source. In other words, ADERIS groups and sends triple patterns to the relevant endpoint together if their predicates are covered in the same data source. Actually, the main idea of this grouping is the same with EG.

To conclude this subsection, there are three methods for subquery building in query federation engines which are EG, EGSV and SAG. Although some engines group the triple patterns which refer to the same data source, they do not name this method as EG. On the other hand, SAG is used when there is an assumption that this predicate is used for the resources of one data set. Each triple pattern of a query is accepted as a subquery without using these methods.

5.2 Join method selection

The second decision in query optimization is join method selection. In this decision, the join method is selected if more than one exists. As stated in Section 4, DARQ, SPLENDID, ANAPSID, ADERIS and LHD implement two different join methods. We classify the methods for join method selection as follows: (i) binding limitation-based (BLB), (ii) cost-based (CB) and (iii) time constrained-based (TCB).

ANAPSID employs bound join when a binding is required by a data source. We refer to this selection method as BLB. DARQ employs bound join when the data sources have limitations on access patterns. Actually, it can be accepted as a binding limitation. However, when there is no binding limitation, DARQ employs cost models for join method selection as well. We refer to this selection method as CB.

After data source selection, for each predicate, ADERIS generates a predicate table with two columns (subject, object) by sending queries to the selected sources. Using these predicate tables, the join could be done with index nested loop join method. However, some endpoints may refuse to answer the queries for building predicate tables, if the estimated execution time is longer than a given threshold. If a predicate table is missing, the engine sends a subquery with bindings for the subject/object values to the corresponding endpoint, meaning that the join method becomes bound join. We refer to this selection method as TCB method.

SPLENDID and LHD use cost models for join method selection as DARQ. Equation (1) (Quilitz & Leser, 2008) and Equation (2) (Quilitz & Leser, 2008) are the cost functions of DARQ for nested loop join and bound join, respectively, where q and p are the relations, $R(q)$ the result size of q, c_t the transfer cost for one tuple and c_r the transfer cost for one query. p' is the relation with the bindings of q. We see that DARQ considers the transfer cost for each tuple and the transfer cost for each query in its cost functions. The transfer cost for one query is multiplied by two in nested loop join, whereas it is multiplied with the result size of the first relation in bound join. Besides, even though the transfer cost for one tuple is multiplied by result size for both two relations, the result size of the second relation is reduced by the bindings of the first join argument in bound join. SPLENDID uses the same cost functions. It uses Equation (1) for hash join and Equation (2) for bound join:

$$cost(q \bowtie_H p) = R(q) \cdot c_t + R(p) \cdot c_t + 2 \cdot c_r \tag{1}$$

$$cost(q \bowtie_B p) = R(q) \cdot c_t + R(q) \cdot c_r + R(p') \cdot c_t \tag{2}$$

LHD (Wang *et al.*, 2013) proposes a cost function assuming that response time of a query is proportional to the number of bindings. It first estimates the cardinalities of each operation by using statistics in VoID descriptions and then estimates the response time of query plans. It proposes two different plans according to the usage of bindings in query execution. The plan is named as plain access plan when a triple

pattern is executed directly whereas the plan is called as dependent access plan when intermediate bindings are used to execute the triple pattern.

Equations (3–6) (Wang *et al.*, 2013) show the cost functions where a plain access plan of triple pattern t is denoted as $acc(t)$ and dependent access plan with bindings of q is represented as $acc(q, t)$. Also rt_q is the time of sending a triple pattern or a pre-computed result to a data source, and rt_t is the time of receiving a result. Actually, these functions are quite similar to the cost models of DARQ and SPLENDID:

$$cost(q \bowtie_H p) = maximum(cost(q), cost(p)) \tag{3}$$

$$cost(q \bowtie_B p) = cost(q) + cost(acc(card(q), t)) \tag{4}$$

$$cost(acc(t)) = rt_q + card(t) \cdot rt_t \tag{5}$$

$$cost(acc(q, t)) = card(q) \cdot rt_q + card(q \bowtie t) \cdot rt_t \tag{6}$$

5.3 Join ordering

Ibaraki and Kameda (1984) stated that finding an optimization cost for a query is admitted as computationally intractable. Starting from this point of view, Gardarin and Valduriez (1990) specified that heuristics are necessary for optimizing the cost functions. Due to huge and distributed data space, query processing with SPARQL endpoints is a difficult task. Thus, using heuristic methods for join order optimization in federated query processing is an expected case.

FedX and WoDQA employ various heuristics for join ordering. We name these heuristics as follows:

- *Free variables heuristic (FVH)*: considers the number of free and bound variables. The number of free variables of triple patterns and groups are counted with considering the already bound variables from the earlier iterations. In other words, the free variables which have become bound from the earlier iterations are accepted as bound variables.
- *Exclusive group priority heuristic (EGPH)*: gives priority to exclusive groups.
- *Position and type-based selectivity heuristic (PTSH)*: calculates the heuristic selectivity value of each triple pattern as multiplying the coefficients of each node according to their positions with the coefficients of each node according to their types.
- *Shared variables heuristic (SVH)*: reorders the join order by considering the shared variables between triple patterns.

FedX orders both the triple patterns and groups of triple patterns by using FVH. The triple patterns and groups are chosen with the lowest cost iteratively.

WoDQA orders the triple patterns of each query by using PTSH after creating the exclusive groups of each query. The coefficient of position and types are assigned according to their selectivities. It considers that subjects are more selective than objects and objects are more selective than predicates. A similar strategy is used by Stocker *et al.* (2008) for the Jena ARQ optimizer in which they categorize this estimation as heuristics without pre-computed statistics and state that there are more triples matching with a predicate than a subject or an object in a typical data source. But at the same time Stocker *et al.* (2008) specify that making a distinction between subject and object is more difficult. On the other hand, WoDQA orders the selectivities of types as URIs, literals and variables.

After ordering the triple patterns in an exclusive group by employing PTSH, WoDQA uses SVH in order to order exclusive groups. The triple patterns which do not have shared variables are changed with the next triple pattern to process the related joins as early as possible. After finishing the ordering of triple patterns of each exclusive group, WoDQA orders the exclusive groups. It calculates the mean selectivity of each group by using PTSH. Lastly, this order is changed by employing SVH for exclusive groups. There is a difference between the usage of SVH for triple patterns and exclusive groups. The exclusive group which has more shared variables than the consequent group moves up in the order of exclusive groups. The aim of ordering the exclusive groups is to decrease the intermediate results as well.

Different from the above engines, DARQ, SPLENDID, LHD and ADERIS use cost-based (CB) methods for join ordering. DARQ, SPLENDID and LHD use dynamic programming (DP) whereas ADERIS employs greedy algorithm (GA) for the search strategy. The cost functions of DARQ and SPLENDID (Equations (1 and 2)), and LHD (Equations (3–6)) are explained in the previous subsection. Both of these engines consider the cardinality estimations, cost for sending a triple pattern and cost for receiving a result. Query result size estimation in DARQ is done by using the statistics in the service descriptions. On the other hand, SPLENDID and LHD use VoID statistics for the cardinality estimations in their cost functions.

As mentioned in the previous section, LHD classifies the execution of joins as follows: (i) joins which do not require input bindings (plain access plan) and (ii) joins which require pre-computed bindings (dependent access plan). The joins in the first class can be executed in a parallel fashion whereas the second one should be executed in a sequence due to the need of bindings. LHD uses plain access plans for the triple patterns which have concrete subject or object. We refer to that heuristic as concrete subject or object heuristic (CSOH). Therefore, it first executes these triple patterns and then uses DP to find the query plan with the minimum response time. Thus, we can say that LHD uses both heuristics and DP in query planning. Second, it determines the actual order of triple patterns to execute them in a parallel way by taking into consideration the type of access plans, bound variables and the already bound variables from the previous iterations. For example, it executes the triple patterns with plain access plan concurrently and the triple patterns with the dependent access are executed as soon as its bindings are ready. In addition, it proposes a concurrent control system to use the bandwidth and data sources efficiently by separating the execution of plans from communication from data sources.

Different from above engines, ADERIS supports adaptive query processing. The first version of ADERIS (Lynden *et al.*, 2010) builds predicate tables for federated query processing and adaptively joins two tables as they become complete while the other predicate tables are being generated. In the second version (Lynden *et al.*, 2011), the engine can employ non-materialized predicate tables and it uses an adaptive cost model for query optimization. Equation (7) (Lynden *et al.*, 2011) shows the cost model of ADERIS where *incard* is the estimated input cardinality for each iteration, *lookupTime* refers to the average time taken to probe a given table t and R is the remaining set of tables that need to be joined to the current plan. Furthermore, join ordering is based on a GA. Most importantly, the engine estimates cardinality at each stage for join ordering.

$$cost(t) = incard \cdot lookupTime(t) \cdot \sum_{i \in R} lookupTime(i) \cdot cardEst(t) \tag{7}$$

Table 3 shows the query optimization methods used in each decision in query federation engines. In order to decrease the HTTP requests, DARQ, FedX, SPLENDID, ANAPSID and ADERIS use EG. WoDQA employs EGSV. Thus, it aims to decrease the redundant intermediate results as well. SPLENDID employs SAG together with EG. However, this method cannot be used when other data sets use *owl:*

Table 3 Query optimization methods in query federation

	Subquery building			Join method selection			Join ordering						
	EG	EGSV	SAG	BLB	TCB	CB	FVH	EPGH	PTSH	SVH	CSOH	DP	GA
DARQ	+			+		+						+	
FedX	+						+	+					
SPLENDID	+		+			+						+	
ANAPSID	+			+									
ADERIS	+				+								+
LHD						+					+	+	
WoDQA		+							+	+			

EG = exclusive grouping; EGSV = exclusive grouping shared variables; SAG = *owl:sameAs* grouping; BLB = binding limitation-based; TCB = time constrained-based; CB = cost-based; FVH = free variables heuristic; EGPH = exclusive grouping priority heuristic; PTSH = position and type-based selectivity heuristic; SVH = shared variables heuristic; CSOH = concrete subject or object heuristic; DP = dynamic programming; GA = greedy algorithm.

sameAs predicate to define that the resource in their data sets indicates to the same resource in other data sets. Although LHD does not group the triple patterns, it sends them in a parallel fashion. The second decision and the third decision are the join method selection and the join ordering. Actually, they are substantially related with each other in query federation. FedX and WoDQA use some heuristics whereas DARQ, SPLENDID, ADERIS and LHD propose cost functions for join ordering. FedX orders the joins by using FVH and EGPH. WoDQA first orders the triple patterns by using PTSH and SVH. Second, it orders the exclusive groups by SVH. The different usages of SVH for the triple patterns and the exclusive groups are explained in the beginning of this subsection. DARQ and SPLENDID use CB method and DP for join method selection and join ordering. DARQ also uses BLB method for join method selection. ANAPSID selects the join method by employing BLB method as well. LHD first uses CSOH and then employs DP for join method selection and join ordering according to the proposed cost function. ADERIS employs TCB method for join method selection and uses GA for join ordering.

6 Discussion and future directions

We have already discussed the data source selection, query optimization and query execution processes of existing query federation engines. In this section, we first summarize the main results from the previous sections with a qualitative comparison and provide the measured metrics in the current engines. Then, we state the challenges in query federation.

6.1 Discussion on query federation on linked data

We provide a qualitative comparison of query federation engines according to the following criteria:

- No preprocessing per query: data source selection without using a metadata catalog might cause some performance problems due to the need of preprocessing per query.
- Unbound predicate queries: predicates are less selective than subjects and objects in a typical data source (Stocker *et al.*, 2008). Therefore, selecting the data source for a query with an unbound predicate is a difficult task. However, the data source selection methods of some engines are based on predicates. These query federation engines might have some problems to handle queries with unbound predicates.
- Parallelization: parallelization is another fact which improves the performances of engines due to the concurrent query processing. Parallelism can be achieved in two forms which are inter-operator parallelism and intra-operator parallelism. In inter-operator parallelism, more than one operation of a query are executed concurrently whereas a single operator is executed by multiple processors in intra-operator parallelism.
- Adaptive query processing: adaptive query processing (Deshpande *et al.*, 2007) is a form of dynamic query processing which reacts to the unforeseen variations of run-time environment (Ozsu & Valduriez, 2011). As the federated query processing is done on the Web, adaptive query processing is required in order to manage the changing conditions such as endpoint unavailability and timeouts.

Table 4 shows the qualitative comparison of engines with the mentioned criteria. All except FedX do not need preprocessing per query before query processing due to metadata catalogs. FedX sends SPARQL ASK queries to data sources for each query. Other engines primarily employ metadata catalogs. Actually, ADERIS sends SELECT DISTINCT queries to decide which predicates are covered by each data source. However, it does not send individual SPARQL ASK queries for each triple pattern in the query.

DARQ and ANAPSID cannot manage unbound predicate queries due to their predicate-based data source selections. SPLENDID, ADERIS and LHD handle queries with unbound predicates, although the data source selection methods of ADERIS and LHD are based on predicates. LHD and SPLENDID support unbound predicates without eliminating irrelevant data sources and ADERIS defines data sources manually when it cannot do that automatically. They might cause some performance problems but the queries with unbound predicates can be supported by this way.

FedX, ANAPSID and LHD execute the triple patterns in a parallel fashion. FedX integrates a parallelization infrastructure to execute subqueries at different endpoints concurrently. And also it uses a

Table 4 Comparison of query federation engines

	No preprocessing per query	Unbound predicate queries	Parallelization		Adaptive query processing
			Inter-operator	Intra-operator	
DARQ	+				
FedX		+	+		
SPLENDID	+	+			
ANAPSID	+			+	+
ADERIS	+	+			+
LHD	+	+	+	+	
WoDQA	+	+			

Table 5 Measured metrics of query federation engines

	A	B	C	D	E	F	G	H	I
DARQ					+		+		
FedX			+	+			+		+
SPLENDID	+	+		+			+		+
ANAPSID							+	+	+
ADERIS							+		
LHD						+			+
WoDQA	+	+	+	+			+		+

A = number of SPARQL ASK queries; B = number of selected data sources; C = data source selection time; D = number of HTTP requests; E = query planning and optimization time; F = number of queries executed per second; G = response time; H = first tuple time; I = comparison with other engines.

pipelining approach to send intermediate results to the next operator as they are ready. ANAPSID executes the triple patterns in parallel by proposing a join method based on symmetric hash join and XJoin. LHD separates the query plans and communication with data sources for parallelization. Several threads are used for sending triple patterns to a data source and for receiving results from a data source. It also considers the type of access plans, bound variables and the already bound variables from the previous iterations to adopt parallelization. Furthermore, it uses multiple hash joins. FedX provides inter-operator parallelism whereas ANAPSID employs intra-operator parallelism. LHD affords both inter-operator and intra-operator parallelism.

Only ANAPSID and ADERIS enable adaptive query processing. ANAPSID proposes adaptive join methods which produce new results when one of the endpoints becomes blocked. On the other hand, ADERIS provides adaptive query processing by changing the join order.

Table 5 shows the common measured metrics of the query federation engines. We think that these metrics are important as they show the main focus of query federation engines. The first three metrics are about the data source selection, the fourth and fifth one are about query optimization, and the sixth and seventh one are about the general performances of the engines. First tuple time and comparison with other engines are other measured metrics. Looking at these metrics, we can say that the studies in query federation mainly focus on the data source selection part so far. In order to make efficient query processing in this global and distributed data space, the first requirement is to select only the relevant data sources. As we have mentioned in the previous parts, SPARQL ASK queries can be used in data source selection. As the number of SPARQL ASK queries and the selected data sources increase, the network communication increases as well. Furthermore, data source selection time certainly gives idea about the performance of data source selection. For this reason, we can say that the first two metrics show the performances of engines in data source selection step. The number of HTTP requests is about query

optimization. This number changes according to the number of subqueries, join methods and join ordering. Query planning and optimization time is another measured metric which is certainly related to query optimization step. The next two metrics are the number of queries executed per second and the response time. The main objective of query federation engines is to minimize the response time. For this reason, the response time is the most common metric. Different from other engines, ANAPSID evaluated the first tuple time to show the advantages of non-blocking join methods. The last measured metric in the table is comparison with other engines. In addition to these metrics, LHD evaluated the size of outcoming and incoming data, the CPU usage per query and the memory usage per query.

6.2 Challenges

During surveying the studies in query federation on linked data, we have realized that there are some challenges and open research issues in this field, namely, metadata management, caching results and adaptive query processing. In this subsection, we state these challenges and suggest some ideas to handle them. Distributed query processing has been studied in database research for a long time. Therefore, we believe that some approaches in database field can be adapted to query federation on linked data to overcome the caching and adaptive query processing challenges.

6.2.1 Metadata management

Using a metadata catalog seems as an essential part of data source selection in query federation. As mentioned previously, in order to generate such metadata catalogs, service descriptions, VoID descriptions and predicate lists are used. Service description was proposed by Quilitz and Leser (2008) for DARQ, whereas VoID was proposed by Cyganiak *et al.* (2011) as a vocabulary which allows to define linked RDF data sets. In other words, VoID aims to provide a standard metadata publishing approach for RDF data. In addition, statistics about the entire data set or the linkset can be expressed in the VoID descriptions and these statistics can be used in query optimization. Therefore, VoID descriptions are more appropriate for generating metadata catalogs for query federation engines in general due to providing the metadata of the data sources and their relations with other data sources. However, there are some open research questions in this point. How is this metadata catalog generated? How often is this metadata catalog updated? More generally, how is metadata management supported?

Although most of the engines use metadata catalogs for data source selection, a few number of data sources provide their metadata descriptions in practice. Thus, existing engines should generate these descriptions before query processing. In other words, the engines which use metadata catalogs need to create them before query processing. Actually, VoID descriptions provide only a standardized vocabulary to express the metadata about the data set or the linkset and the statistics about the data set. However, how are the metadata information and the statistics obtained?

In addition, just generating a metadata catalog is not enough in practice. After generating the metadata catalog, the challenge of keeping it up-to-date arises when the data source is updated. The changes in the data set should be covered in the catalog as well. So we can say that there are two main challenges in metadata catalog usage. The first one is generating the metadata catalogs. And the other one is keeping the metadata catalogs up-to-date. In order to overcome these challenges, a metadata catalog management framework is a need. This framework should realize the following tasks: (i) gather and express the metadata of the data sets and maybe the statistics with VoID descriptions, (ii) monitor the changes in data sets and (iii) update the metadata catalog according to these changes. Thus, an up-to-date, standardized metadata catalog management can be provided and it can be both used in data source selection and query optimization steps in federated query processing.

6.2.2 Caching results

Caching has an important role in improving the performance in distributed query processing. Adali *et al.* (1996) proposed a query result caching mechanism by using the invariants which define the certain relationships between two different queries. In particular, a query can be covered by another query.

Therefore, the results of the covered query can be found from the cache by employing the invariants. The invariants should be decided through the knowledge about the data sources in the queries. Employing quite general invariants with few information about the data source is another option. Adali *et al.* (1996) stated that caching has provided savings in time and invariants is useful when the query is not explicitly cached. Furthermore, caching is also employed by search engines to improve the response time as well. Gan and Suel (2009) discussed the studies for caching in search engines, in addition Cambazoglu *et al.* (2012) classified the methods as result, similarity, semantic and rank caching.

Although caching has been studied in distributed query processing and search engines for a long time, caching in linked data has become popular recently. Martin *et al.* (2010) proposed an approach as a proxy layer between semantic web applications and the endpoint for caching results of querying triple stores. When the query is sent by the user, first the cache is checked and if the results of this query exists in the cache, the answer is returned without executing it in the endpoint. If the query is not cached previously, it is executed in its endpoint and then cached. In addition to query caching, this approach enables caching the arbitrary application objects and can associate these objects with the related queries. Although this study provides an efficient caching approach by keeping the results of a query only once, it is not useful for federated query processing due to working between the endpoint and the application. A study about caching at the HTTP level was presented by Williams and Weaver (2011), which uses the last modification date information and the up-to-dateness in the HTTP headings. Lorey and Naumann (2013) proposed another caching approach which aims to retrieve and cache potentially relevant data for subsequent requests via query matching. Although these two studies both employ query caching approaches for SPARQL queries, the first one cannot manage the cache size and the idea behind the second one can be a time-consuming task in real.

All the above studies provide caching query results in some way; however, a more comprehensive and efficient caching mechanism is a need for query federation engines.

Umbrich *et al.* (2012a) presented general strategies to implement a hybrid query processing approach based on link traversal which uses both caching and live querying. The aim of this study is to optimize the speed and freshness. Hence, the freshness of the cache is measured through coherence estimates. Umbrich *et al.* (2012b) discussed the trade-off between the up-to-date and fast results and thus proposed a hybrid approach which is based on the Umbrich *et al.* (2012a). However, to the best of our knowledge, there is no study with a hybrid query processing which employs a caching mechanism and live querying in query federation. It should be noted that FedX (Schwarte *et al.*, 2011) has a naive caching mechanism which caches only the results of the SPARQL ASK queries in the data source selection. We believe that combining caching results and live query results decrease the query processing time of query federation engines as in distributed mediator systems. This caching mechanism should cover the subsets of a query similar as Adali *et al.* (1996) ideally and should have an updating strategy in order to make the federated query processing more efficient. Besides, caching and live querying can be used by employing a hybrid query processing.

6.2.3 Adaptive query processing

The weaknesses of the plan-first-execute-then strategy have become clear as the database field has broadened to consider more general data management including autonomous remote data sources. Therefore, there has been great interest in adaptive methods due to providing solutions to dynamic data, missing statistics and unpredictable costs by using feedback to tune query processing.

In database community there is a wide panorama of research work for adaptive query processing with varying degree of adaptivity from evolutionary methods to revolutionary methods. Evolutionary methods focus on generating plans that can be switched during execution according to delays or estimation errors. Producing plans ready to be used with late binding mechanisms as to reduce the overhead of run-time planning and ensuring adaptivity in case of variation between compile-time and run-time conditions are the advantages of these methods. Their way of adaptation is inter-operator without interfering the execution of individual operator. Some known examples of evolutionary methods are query scrambling (Amsaleg *et al.*, 1998), mid-query re-optimization (Kabra & DeWitt, 1998), Tukwilla/ECA rules

(Ives *et al.*, 1999) progressive query optimization (Markl *et al.*, 2004; Kache *et al.*, 2006; Han *et al.*, 2007) and proactive re-optimization (Babu *et al.*, 2005).

On the other hand, revolutionary methods are more recent and their way of re-optimization is intra-operator. First group of intra-operator methods are adaptive operators like double hash join (Ives *et al.*, 1999), XJoin (Urhan & Franklin, 2000) and mobile join (Arcangeli *et al.*, 2004; Ozakar *et al.*, 2005), where the operator itself is able to adapt its way of execution according to variations encountered during its execution. Second group of intra-operator methods come up with the invention of eddies which enable researchers optimize the query processing from fine-grained to tuple level (Avnur & Hellerstein, 2000; Raman *et al.*, 2003; Deshpande, 2004; Deshpande & Hellerstein, 2004; Bizarro *et al.*, 2005; Zhou *et al.*, 2013).

When we look at the adaptive methods of query federation engines on linked data, we see only two adaptive methods, intra-operator approach of ANAPSID (Acosta *et al.*, 2011) and inter-operator approach of ADERIS (Lynden *et al.*, 2011). They showed that adaptable methods are well suited to unpredictable characteristics of linked data environment. Eddy-like operators specialized for linked data, which continuously re-optimize query blocks in response to changing run-time conditions, can be integrated to existing federated linked data engines. In order to construct an eddy operator; collaborating operators like select, symmetric joins, index join and finally routing operator (eddy) should be designed.

7 Conclusion

As the increase in the data providers of linked data, the Web of data becomes a single and global data space. Link traversal and query federation are the two approaches which provide automated query processing on this interlinked and distributed environment. In this paper, we focus on query federation approach and synthesize the data source selection, join methods and query optimization methods in the promising federated query processing engines. We provide a complementary comparison of the measured metrics for each query federation engine in order to refer the main strengths of each one. During studying on this survey, we have realized the challenges in this research area. We think that more complete, efficient and realistic query federation engines will be offered if the mentioned challenges can be addressed. Due to the similarities between the mediator-wrapper architecture in distributed heterogeneous databases and the federated query processing on linked data, we expect that some of the mentioned challenges can be overcome through adapting some approaches in the database field.

Acknowledgement

This work is partially supported by The Scientific and Technological Research Council of Turkey (TUBITAK).

References

Acosta, M., Vidal, M.-E., Lampo, T., Castillo, J. & Ruckhaus, E. 2011. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *The Semantic Web ISWC 2011*, Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N. & Blomqvist, E. (eds), Lecture Notes in Computer Science **7031**, 18–34. Springer.

Adali, S., Candan, K. S., Papakonstantinou, Y. & Subrahmanian, V. S. 1996. Query caching and optimization in distributed mediator systems. *ACM SIGMOD Record* **25**(2), 137–146.

Akar, Z., Halaç, T. G., Ekinci, E. E. & Dikenelli, O. 2012. Querying the web of interlinked datasets using VOID descriptions. In *Linked Data on the Web (LDOW2012)*.

Alexander, K. & Hausenblas, M. 2009. Describing linked datasets—on the design and usage of VoID, the 'Vocabulary of Interlinked Datasets'. In *WWW 2009 Workshop: Linked Data on the Web (LDOW2009)*.

Amsaleg, L., Franklin, M. J. & Tomasic, A. 1998. Dynamic query operator scheduling for wide-area remote access. *Distributed and Parallel Databases* **6**(3), 217–246.

Arcangeli, J., Hameurlain, A., Migeon, F. & Morvan, F. 2004. Mobile agent based self-adaptive join for wide-area distributed query processing. *Journal of Database Management (JDM)* **15**(4), 25–44.

Avnur, R. & Hellerstein, J. M. 2000. Eddies: continuously adaptive query processing. *ACM SIGMOD Record* **29**(2), 261–272.

Babu, S., Bizarro, P. & DeWitt, D. 2005. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD'05*, 107–118. ACM.

Berners-Lee, T. 2006. Linked data—design issues. `http://www.w3.org/DesignIssues/LinkedData.html`.

Bizarro, P., Babu, S., DeWitt, D. & Widom, J. 2005. Content-based routing: different plans for different data. In *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB'05*, 757–768. VLDB Endowment.

Bizer, C. 2009. The emerging web of linked data. *IEEE Intelligent Systems* **24**(5), 87–92.

Bizer, C., Heath, T. & Berners-Lee, T. 2009. Linked data—the story so far. *International Journal on Semantic Web and Information Systems* **5**(3), 1–22.

Blanco, E., Cardinale, Y. & Vidal, M.-E. 2012. Experiences of sampling-based approaches for estimating qos parameters in the web service composition problem. *IJWGS* **8**(1), 1–30.

Buil-Aranda, C., Arenas, M., Corcho, O. & Polleres, A. 2013. Federating queries in SPARQL 1.1: syntax, semantics and evaluation. *Web Semantics: Science, Services and Agents on the World Wide Web* **18**(1), 1–17.

Buil-Aranda, C., Polleres, A. & Umbrich, J. 2014. Strategies for executing federated queries in SPARQL 1.1. In *The Semantic Web—ISWC 2014—13th International Semantic Web Conference, 19–23 October. Proceedings, Part II*, 390–405.

Cambazoglu, B. B., Altingovde, I. S., Ozcan, R. & Ulusoy, O. 2012. Cache-based query processing for search engines. *ACM Transactions on the Web (TWEB)* **6**(4), 14.

Cyganiak, R., Zhao, J., Alexander, K. & Hausenblas, M. 2011. Describing linked datasets with the VoID vocabulary. `http://rdfs.org/ns/void/`.

Deshpande, A. 2004. An initial study of overheads of eddies. *ACM SIGMOD Record* **33**(1), 44–49.

Deshpande, A. & Hellerstein, J. M. 2004. Lifting the burden of history from adaptive query processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases—Volume 30, VLDB'04*, 948–959. VLDB Endowment.

Deshpande, A., Ives, Z. & Raman, V. 2007. Adaptive query processing. *Found Trends Databases* **1**(1), 1–140.

Fionda, V., Gutierrez, C. & Pirró, G. 2012. Semantic navigation on the web of data: specification of routes, web fragments and actions. In *Proceedings of the 21st International Conference on World Wide Web, WWW'12*, 281–290. ACM.

Florescu, D., Levy, A., Manolescu, I. & Suciu, D. 1999. Query optimization in the presence of limited access patterns. *ACM SIGMOD Record* **28**(2), 311–322.

Gan, Q. & Suel, T. 2009. Improved techniques for result caching in web search engines. In *Proceedings of the 18th International Conference on World Wide Web, WWW'09*, 431–440. ACM.

Gardarin, G. & Valduriez, P. 1990. *Relational Databases and Knowledge Bases*. Addison-Wesley Longman Publishing Co., Inc.

Görlitz, O. & Staab, S. 2011a. Federated data management and query optimization for linked open data. In *New Directions in Web Data Management 1*, Vakali, A. & Jain, L. C. (eds), Studies in Computational Intelligence **331**, 109–137. Springer.

Görlitz, O. & Staab, S. 2011b. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), 23 October*, Hartig, O., Harth, A. & Sequeda, J. (eds), CEUR Workshop Proceedings **782**, CEUR-WS.org

Haas, L. M., Kossmann, D., Wimmers, E. L. & Yang, J. 1997. Optimizing queries across diverse data sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB'97*, 276–285. Morgan Kaufmann Publishers, Inc.

Han, W.-S., Ng, J., Markl, V., Kache, H. & Kandil, M. 2007. Progressive optimization in a shared-nothing parallel database. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD'07*, 809–820. ACM.

Hartig, O. 2011. Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *Proceedings of the 8th Extended Semantic Web Conference on The Semantic Web: Research and Applications—Volume Part I, ESWC'11*, 154–169. Springer-Verlag.

Hartig, O. 2013. SQUIN: a traversal based query execution system for the web of linked data. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD'13*, 1081–1084. ACM.

Hartig, O., Bizer, C. & Freytag, J.-C. 2009. Executing SPARQL queries over the web of linked data. In *The Semantic Web—ISWC 2009*, Bernstein, A., Karger, D., Heath, T., Feigenbaum, L., Maynard, D., Motta, E. & Thirunarayan, K. (eds), Lecture Notes in Computer Science **5823**, 293–309. Springer.

Hartig, O. & Langegger, A. 2010. A database perspective on consuming linked data on the web. *Datenbank-Spektrum* **10**(2), 57–66.

Ibaraki, T. & Kameda, T. 1984. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems* **9**(3), 482–502.

Ives, Z. G., Florescu, D., Friedman, M., Levy, A. & Weld, D. S. 1999. An adaptive query execution system for data integration. *ACM SIGMOD Record* **28**(2), 299–310.

Kabra, N. & DeWitt, D. J. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. *ACM SIGMOD Record* **27**(2), 106–117.

Kache, H., Han, W.-S., Markl, V., Raman, V. & Ewen, S. 2006. POP/FED: progressive query optimization for federated queries in DB2. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB'06*, 1175–1178. VLDB Endowment.

Lorey, J. & Naumann, F. 2013. Caching and prefetching strategies for SPARQL queries. In *The Semantic Web: ESWC 2013 Satellite Events*, Cimiano, P., Fernndez, M., Lopez, V., Schlobach, S. & Vlker, J. (eds), Lecture Notes in Computer Science **7955**, 46–65. Springer.

Lynden, S., Kojima, I., Matono, A. & Tanimura, Y. 2010. Adaptive integration of distributed semantic web data. In *Proceedings of the 6th International Conference on Databases in Networked Information Systems, DNIS'10*, 174–193. Springer-Verlag.

Lynden, S., Kojima, I., Matono, A. & Tanimura, Y. 2011. ADERIS: an adaptive query processor for joining federated SPARQL endpoints. In *Proceedings of the 2011th Confederated International Conference on the Move to Meaningful Internet Systems—Volume Part II, OTM'11*, 808–817. Springer-Verlag.

Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H. & Cilimdzic, M. 2004. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD'04*, 659–670. ACM.

Martin, M., Unbehauen, J. & Auer, S. 2010. Improving the performance of semantic web applications with SPARQL query caching. In *Proceedings of the 7th International Conference on The Semantic Web: Research and Applications—Volume Part II, ESWC'10*, 304–318. Springer-Verlag.

Ozakar, B., Morvan, F. & Hameurlain, A. 2005. Mobile join operators for restricted sources. *Mobile Information Systems* **1**(3), 167–184.

Ozsu, M. & Valduriez, P. 2011. *Principles of Distributed Database Systems*, 3rd edition. Springer.

Quilitz, B. & Leser, U. 2008. Querying distributed RDF data sources with SPARQL. In *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, 524–538. Springer-Verlag.

Rakhmawati, N. A., Umbrich, J., Karnstedt, M., Hasnain, A. & Hausenblas, M. 2013. Querying over federated SPARQL endpoints—a state of the art survey. *CoRR abs/1306.1723*.

Raman, V., Deshpande, A. & Hellerstein, J. M. 2003. Using state modules for adaptive query processing. In *Proceedings of the 19th International Conference on Data Engineering, 5–8 March*, 353–364.

Saleem, M., Khan, Y., Hasnain, A., Ermilov, I. & Ngomo, A. N. 2015. A fine-grained evaluation of SPARQL endpoint federation systems. *Semantic Web Journal*, 1–26. `http://content.iospress.com/articles/semantic-web/sw186`.

Saleem, M. & Ngomo, A. N. 2014. HiBISCuS: hypergraph-based source selection for SPARQL endpoint federation. In *The Semantic Web: Trends and Challenges—11th International Conference, ESWC 2014, 25–29 May. Proceedings*, 176–191.

Saleem, M., Ngomo, A. N., Parreira, J. X., Deus, H. F. & Hauswirth, M. 2013. DAW: duplicate-aware federated query processing over the web of data. In *The Semantic Web—ISWC 2013—12th International Semantic Web Conference, 21–25 October, Proceedings, Part I*, 574–590.

Schwarte, A., Haase, P., Hose, K., Schenkel, R. & Schmidt, M. 2011. FedX: optimization techniques for federated query processing on linked data. In *The Semantic Web—ISWC 2011—10th International Semantic Web Conference, 23–27 October, Proceedings, Part I*, 601–616.

Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C. & Reynolds, D. 2008. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, 21–25 April*, 595–604.

Umbrich, J., Karnstedt, M., Hogan, A. & Parreira, J. X. 2012a. Freshening up while staying fast: towards hybrid SPARQL queries. In *Knowledge Engineering and Knowledge Management—18th International Conference, EKAW 2012, 8–12 October. Proceedings*, 164–174.

Umbrich, J., Karnstedt, M., Hogan, A. & Parreira, J. X. 2012b. Hybrid SPARQL queries: fresh vs. fast results. In *The Semantic Web—ISWC 2012—11th International Semantic Web Conference, 11–15 November, Proceedings, Part I*, 608–624.

Urhan, T. & Franklin, M. J. 2000. XJoin: a reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin* **23**(2), 27–33.

Vidal, M., Ruckhaus, E., Lampo, T., Martnez, A., Sierra, J. & Polleres, A. 2010. Efficiently joining group patterns in SPARQL queries. In *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, 30 May 30–3 June, Proceedings, Part I*, 228–242.

Wang, X., Tiropanis, T. & Davis, H. C. 2013. LHD: optimising linked data query processing using parallelisation. In *Proceedings of the WWW2013 Workshop on Linked Data on the Web, 14 May*.

Wiederhold, G. 1992. Mediators in the architecture of future information systems. *IEEE Computer* **25**(3), 38–49.

Williams, G. T. & Weaver, J. 2011. Enabling fine-grained HTTP caching of SPARQL query results. In *The Semantic Web—ISWC 2011—10th International Semantic Web Conference, 23–27 October, Proceedings, Part I*, 762–777.

Wilschut, A. N. & Apers, P. M. G. 1991. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems, PDIS'91*, 68–77. IEEE Computer Society Press.

Yönyül, B. 2014. *Performance Management in Federated Linked Data Query Engines*. Master's thesis, Ege University.

Zhou, Y., De, S. & Moessner, K. 2013. Implementation of federated query processing on linked data. In *2013 IEEE 24th International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC)*, 3553–3557.