A Cost Model to Optimize Queries over Heterogeneous Federations of RDF Data Sources

Sijin Cheng[1,*,†], Olaf Hartig[1]

[1]Linköping University, SE-58183 Linköping, Sweden

Abstract

Federated processing of queries over RDF data sources offers significant potential when a SPARQL query cannot be answered by a single data source alone. However, finding efficient plans to execute a query over a federation is challenging, especially if different federation members provide different types of data access interfaces. Different interfaces imply different request types, different forms of responses, and different physical algorithms that can be used, each of which consumes varying amounts of resources during query execution. This heterogeneity poses additional obstacles to the task of planning query executions, in addition to the inherent complexity arising from numerous possible join orderings and various physical algorithms. As a first step to address these challenges, we propose a cost model that captures the resource requirements of different operators depending on the type of federation member, allowing us to estimate cost of a given query execution plan without actually executing it. To evaluate our approach, we conduct experiments on FedBench with our cost model and compare it to the current state-of-the-art approach to query planning for heterogeneous federations of RDF data sources.

Keywords

Heterogeneous Federations, Cost Model, Query Optimization

1. Introduction

For queries that cannot be answered by accessing a single data source, a federation of RDF data sources becomes essential to collectively answer a query. Processing such a query is challenging as not all parts of the query need to be issued to each federation member. Effectively grouping the subqueries and determining which parts are to be issued to which of the federation members may impact the overall query performance significantly, and the order in which the results of the subqueries are joined is another important performance-related factor. Over the past decade, a number of approaches have been proposed to improve the performance of processing such queries (e.g., [1, 2, 3, 4, 5, 6]). One line of work is focused on the source selection phase, which aims to split a given SPARQL query into subqueries that can be assigned to federation members. The objective is to ensure that each subquery is assigned to only those federation members that can provide a nonempty result (e.g., [5, 7]). Another research direc-

*Corresponding author.

†These authors contributed equally.

sijin.cheng@liu.se (S. Cheng); olaf.hartig@liu.se (O. Hartig)

https://www.ida.liu.se/~sijch63/ (S. Cheng); http://olafhartig.de/ (O. Hartig)

0000-0003-4363-0654 (S. Cheng); 0000-0002-1741-2090 (O. Hartig)

tion involves query planning heuristics, which are typically used to determine the join order in logical plans. Furthermore, some studies propose cost models to determine optimal physical query plans (e.g., [3]).

All these approaches focus only to homogeneous federations, assuming that all federation members provide a SPARQL endpoint interface. In recent years, however, other types of interfaces were proposed, including the Triple Pattern Fragment (TPF) interface [8], the Bindings-Restricted TPF (brTPF) interface [9], the SaGe interface [10], and the smart-KG interface [11]. In the federated setting, each federation member may support a different type of interface to access its RDF dataset. The resulting heterogeneity of federations poses extra challenges that render many of the existing source selection approaches, heuristics, and cost models inadequate [12]. These challenges are primarily due to the features of different interfaces. For instance, different interfaces may require (or enable!) federation engines to leverage specific physical operators, not all forms of subqueries can be answered directly by every interface, and different interfaces may provide different forms of metadata relevant for query processing.

Recently, Heling and Acosta proposed a query planner that is designed to address these challenges of query processing over heterogeneous data sources [13]. This query planner builds left-deep query plans, for which it first determines a join order based on cardinality estimation. Thereafter, the planner selects join algorithms, choosing between a symmetric hash join and bind join, where the choice is made based on the estimated number of requests needed to execute the join. We argue that deciding the join order first and determining the physical algorithm afterwards might overlook the optimal plan, as the order becomes fixed without considering any features of federation members. Moreover, we believe that, when determining the physical algorithm, it is crucial to consider not only the number of requests, but also factors such as the size of data to be transferred and the amount of work that is processed by the engine.

In this paper, we propose a cost model that enables to estimate resource usage in physical plans, considering factors such as the number of requests, the size of data transferred, and the amount of work need to be done by the federation engine. Our cost model considers different features that federation members have when using different types of interfaces. To evaluate the effectiveness of the proposed cost model, we employ it in a greedy plan-enumeration algorithm similar to the one used by Heling and Acosta's approach [13] and, then, compare the approaches experimentally based on the FedBench benchmark. The experiments show that applying our cost model can result in a plan that requires less data to be transferred, without a significant increase in query execution time. In particular, for queries with UNION operators in subqueries after source selection, our cost model is able to find a plan that requires significantly less data to be transferred, resulting in less query execution time with higher completeness of results.

2. Preliminary

This section introduces concepts, notations and a running example used in the rest of the paper. To represent query plans in this paper we use the FedQPL language as introduced in our earlier work [14]. The syntax of FedQPL expressions is defined as follows.

Definition 1. A **FedQPL expression** φ is constructed from the following grammar, where req, tpAdd, bgpAdd, join, union, mj, mu, (,), are terminal symbols. ρ is an expression in the request language L_{req} of some interface [14] (e.g., triple patterns, whole SPARQL patterns), *fm* is a federation member, tp is a triple pattern, B is a BGP, and Φ is a nonempty set of FedQPL expressions.

$$\varphi ::= \mathsf{req}^{\rho}_{fm} \mid \mathsf{tpAdd}^{tp}_{fm}(\varphi) \mid \mathsf{bgpAdd}^{B}_{fm}(\varphi) \mid \mathsf{join}(\varphi,\varphi) \mid \mathsf{union}(\varphi,\varphi) \mid \mathsf{mj}\,\Phi \mid \mathsf{mu}\,\Phi \mid$$

The first operator, req, captures the intention to retrieve the result of a certain (sub)query ρ from a given federation member. The unary operator tpAdd captures the intention to access a federation member to obtain solution mappings for a single triple pattern that must be compatible with solution mappings obtained from the plan represented by the given subexpression. The operator bgpAdd is a BGP-based variation of tpAdd. In contrast to these operators, join is a binary operator that joins two inputs, capturing the intention to get the input sets of solution mappings independently, and then join them in the query federation engine. As for the remaining operators, union lifts the standard SPARQL algebra operator union into the FedQPL language, whereas mj and mu are multiway variations of join and union to capture the intention to apply a multiway algorithm that can combine an arbitrary number of inputs. For a formal definition of the semantics of these operators, refer to our earlier work [14].

Example 1. As our running example, consider a basic graph pattern (BGP) $B_{\text{ex}} = \{tp_1, tp_2\}$ with $tp_1 = (?x, \texttt{foaf:name}, \texttt{"Alice"})$ and $tp_2 = (?x, \texttt{dbo:country}, \texttt{dbr:Canada})$, and a heterogeneous federation F_{ex} with three members, denoted by fm_1, fm_2, and fm_3. While federation members fm_1 and fm_2 provide a SPARQL endpoint interface, fm_3 provides a TPF interface.

The following fragment of FedQPL captures the output of source selection processes [14].

Definition 2. A **source assignment** is a FedQPL expression that is constructed using only the operators req, mj, and mu such that, for each subexpression of the form req^{ρ}_{fm}, it holds that ρ is a triple pattern or a BGP.

Example 2. For the running example (in Example 1), we assume that members fm_1 of our example federation F_{ex} can contribute matching triples for tp_1 of the example BGP B_{ex} (cf. Example 1), whereas tp_2 is matched in the data of fm_2 and fm_3. Hence, we may use the following source assignment a_{ex} for B_{ex} over F_{ex}.

$$\mathsf{mj}\{\ \mathsf{req}^{tp_1}_{fm_1}\ ,\ \mathsf{mu}\{\mathsf{req}^{tp_2}_{fm_2}, \mathsf{req}^{tp_2}_{fm_3}\}\ \} \tag{1}$$

A left-deep logical plan (see Figure 1a) can be converted from source assignment (Equation 1) by converting multiway join (mj) to binary join directly. Further from a logical plan to a physical plan, physical algorithms for each operator can be established directly based on a logical plan. Alternatively, the physical plan can be reconstructed by considering the possible physical algorithms for each operator in a combined manner. The physical algorithms that can be used for each logical operator of FedQPL depend on the type of interface. Table 1 (left-hand side) lists the physical algorithms that we consider in this work. For the unary operators $\mathsf{tpAdd}^{tp}_{fm}(\varphi)$ and $\mathsf{bgpAdd}^{B}_{fm}(\varphi)$, the operator accesses federation members with requests where input solution mappings are shipped as part of the requests. A straightforward algorithm to implement

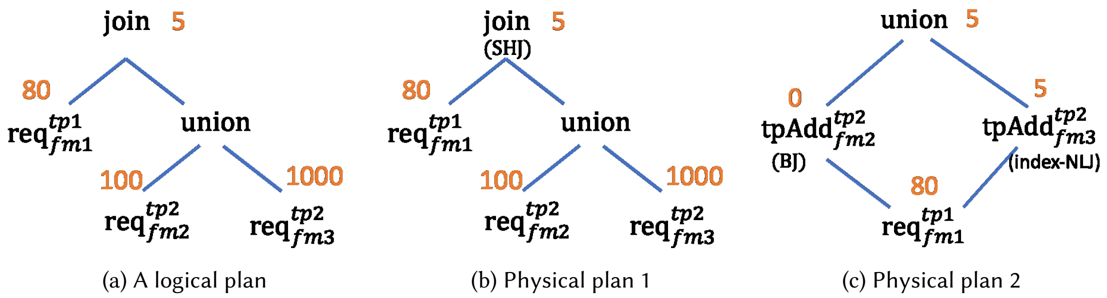

Figure 1: A left-deep logical plan (a) and possible physical plans (b, c) for the running example. The annotations in parentheses indicate the physical algorithm used by each operator (SHJ for symmetric hash join, BJ for bind join, and index-NLJ for index-nested-loops join). The orange numbers present a hypothetical size of intermediate results as used for illustrating the cost calculation in Example 4.

a unary operator is to use an index-nested loop join (index-NLJ). The input solution bindings are iterated for creating requests with updated tp or B, where the variables are replaced with obtained variable bindings. If the federation member provides a TPF interface, only the index-NLJ can be applied as TPF interface restricts the type of queries that can be answered by the server to single triple patterns. If the federation member uses a brTPF interface, a set of input solution mappings (variable bindings for the shared variable) can be attached to a brTPF request. While federation members support the SPARQL endpoint, as further alternatives, the FILTER, UNION and FILTER operators can be used for shipping the input bindings along with the query pattern as requests [15], denoted as bind join (BJ). The binary operator join(φ_1, φ_2) joins input solution mappings in the query federation engine, so the physical algorithm is not affected by the interface type that the federation member uses. In this paper, the symmetric hash join (SHJ) (see [15]) is used for implementing join by default, while there are other options, such as hash join and merge join.

Example 3. For the running example, the logical operator join can be implemented via SHJ (Figure 1b) or rewriting the logical plan to push join into the union (Figure 1c). In the latter case, $\text{tpAdd}^{tp_2}_{fm_2}$ can use an index-NLJ or any variations of the bind join, while $\text{tpAdd}^{tp_2}_{fm_3}$ can only apply index-NLJ as fm_3 provides a TPF interface. Regarding query execution, subplans of binary operators (e.g., SHJ or union) may be executed in parallel, for example, $\text{tpAdd}^{tp_2}_{fm_2}$ and $\text{tpAdd}^{tp_2}_{fm_3}$ of Figure 1c can consume the intermediate results of $\text{req}^{tp_1}_{fm_1}$ at the same time.

3. Cost Model

Each physical query plan consumes a specific amount of different kinds of resources when it is executed, and these resource requirements can differ significantly for different possible plans for the same query. In this section, we present a cost model that is designed to capture these resource requirements and can be used to estimate the cost of a given plan without actually executing the plan. In this cost model, the (estimated) cost of a query plan is calculated as a

Table 1
Physical algorithms for each logical operator (left),
Number of requests for each combination of physical algorithms and interface types (right)

Logical Operator	Interface of *fm*	Physical Algorithm	$\#requests^{\text{est}}$
$\text{tpAdd}^{tp}_{fm}(\varphi)$	SPARQL endpoint	index-NLJ	$\lvert \text{sols}(\varphi) \rvert$
		BJ(UNION) BJ(FILTER) BJ(VALUES)	$\left\lceil \frac{\lvert \text{sols}(\varphi) \rvert}{blockSize} \right\rceil$
	TPF	index-NLJ	$\lvert \text{sols}(\varphi) \rvert \cdot \left\lceil \frac{totalPageNum}{\lvert \text{sols}(\varphi) \rvert} \right\rceil$
	brTPF	index-NLJ	
		brTPF-based BJ	$\left\lceil \frac{\lvert \text{sols}(\varphi) \rvert}{blockSize} \right\rceil \cdot \left\lceil \frac{totalPageNum}{\left\lceil \frac{\lvert \text{sols}(\varphi) \rvert}{blockSize} \right\rceil} \right\rceil$
$\text{bgpAdd}^{B}_{fm}(\varphi)$	SPARQL endpoint	index-NLJ	$\lvert \text{sols}(\varphi) \rvert$
		BJ(UNION) BJ(FILTER) BJ(VALUES)	$\left\lceil \frac{\lvert \text{sols}(\varphi) \rvert}{blockSize} \right\rceil$
req^{tp}_{fm}	SPARQL endpoint	request	1
	TPF, brTPF	TPF request	$\left\lceil \frac{card(tp,fm)}{pageSize} \right\rceil$
req^{B}_{fm}	SPARQL endpoint	request	1
$\text{join}(\varphi_1, \varphi_2)$		SHJ	0
$\text{union}(\varphi_1, \varphi_2)$		Union	0

weighted sum of multiple metrics, and each of these metrics is calculated by summing up corresponding measures from all operators in the plan. In the following, we first define the metrics based on which the cost of a physical plan is captured in our cost model, and introduce functions for measuring each of these metrics for each physical operator. Thereafter, we present how these cost metrics are used to calculate the overall cost for an entire physical plan.

3.1. Cost Metrics

The metrics of our cost model are defined in terms of the costs of communication between the federation engine and the federation members and the processing costs induced at the federation engine. The communication cost considers both the request process and the response process, including the number of requests ($\#requests^{\text{est}}$), the size of shipped request data for all requests together ($\#reqData^{\text{est}}$), and the size of relevant responses all together ($\#respRDFterms^{\text{est}}$). The metric for capturing the processing cost by the federation engine, denoted as $fedProcess^{\text{est}}$, is estimated based on the cardinality of intermediate results of the query plan. Since binary operators, join and union, do not interact with the federation members, there is no associated communication cost involved. Therefore, the primary cost incurred by binary operations is the processing cost by the federation engine.

3.1.1. Number of requests ($\#requests^{\text{est}}$)

To estimate the number of requests involved during query execution, it is important to note that this number depends not only on the physical algorithm but also on the interface type of the accessed federation member. In other words, the same physical algorithm may necessitate a different number of requests if the federation member uses different interfaces. Therefore, we

Table 2
The total size of shipped request data

Logical Operator	Physical Algorithm	$\#reqData^{\text{est}}$	$fedProcess^{\text{est}}$
$\text{tpAdd}_{fm}^{tp}(\varphi)$	index-NLJ BJ(UNION)	$3 \cdot \lvert\text{sols}(\varphi)\rvert$	$\text{joinCard}(tp, fm, \text{sols}(\varphi))$
	BJ(VALUES)	$3 + \lvert\text{sols}(\varphi)\rvert \cdot \text{joinVars}(tp, \varphi)$	
	BJ(FILTER) brTPF algorithm	$3 + 2 \cdot \lvert\text{sols}(\varphi)\rvert \cdot \text{joinVars}(tp, \varphi)$	
$\text{bgpAdd}_{fm}^{B}(\varphi)$	index-NLJ BJ(UNION)	$3 \cdot \text{countTP}(B) \cdot \lvert\text{sols}(\varphi)\rvert$	$\text{joinCard}(B, fm, \text{sols}(\varphi))$
	BJ(VALUES)	$3 \cdot \text{countTP}(B) + \lvert\text{sols}(\varphi)\rvert \cdot \text{joinVars}(B, \varphi)$	
	BJ(FILTER)	$3 \cdot \text{countTP}(B) + 2 \cdot \lvert\text{sols}(\varphi)\rvert \cdot \text{joinVars}(B, \varphi)$	
req_{fm}^{tp}	request	3	$\text{card}(tp, fm)$
req_{fm}^{B}	request	$3 \cdot \text{countTP}(B)$	$\text{card}(B, fm)$
$\text{join}(\varphi_1, \varphi_2)$	SHJ	0	$\text{joinCard}(\text{sols}(\varphi_1), \text{sols}(\varphi_2))$
$\text{union}(\varphi_1, \varphi_2)$	Union	0	$\text{sols}(\varphi_1) + \text{sols}(\varphi_2)$

Table 3
The total size of shipped response data

Logical Operator	Interface of fm	$\#respRDFterms^{\text{est}}$
$\text{tpAdd}_{fm}^{tp}(\varphi)$	SPARQL endpoint	$\lvert vars(tp)\rvert \cdot \text{joinCard}(tp, fm, \text{sols}(\varphi))$
	TPF, brTPF	$3 \cdot \text{joinCard}(tp, fm, \text{sols}(\varphi))$
$\text{bgpAdd}_{fm}^{B}(\varphi)$	SPARQL endpoint	$\lvert vars(B)\rvert \cdot \text{joinCard}(B, fm, \text{sols}(\varphi))$
req_{fm}^{tp}	SPARQL endpoint	$\lvert vars(tp)\rvert \cdot card(tp, fm)$
	TPF, brTPF	$3 \cdot card(tp, fm)$
req_{fm}^{B}	SPARQL endpoint	$\lvert vars(B)\rvert \cdot card(B, fm)$

provide a set of functions for measuring $\#requests^{\text{est}}$ for each combination of physical algorithm and interface type of federation members (illustrated in the right part of Table 1)

For the req_{fm}^{ρ} operator with ρ a BGP B, the number of requests is fixed to one, since this operator is considered valid only when the federation member employs an interface that supports BGP requests (e.g., SPARQL endpoints). For the req_{fm}^{ρ} operator with ρ a triple pattern tp, the federation member can use any type of interface that supports triple pattern requests (e.g., SPARQL endpoints, TPF servers, and brTPF servers). If the federation member provides a SPARQL endpoint, the number of requests is one whereas, for TPF servers and brTPF servers, the number of requests depends on both the cardinality of the results of req_{fm}^{tp} and the page size since TPF and brTPF interfaces employ a paging mechanism. The cardinality of the results of req_{fm}^{tp}, denoted as $\text{card}(tp, fm)$, represents the number of triples in the RDF dataset of fm that match the given triple pattern tp. The page size (*pageSize*) refers to the maximum number of triples that can be retrieved per request from a TPF server or a brTPF server and is typically set to 100. Consequently, the number of requests can be calculated as the ceiling value of the division between the cardinality and the page size: $\lceil \text{card}(tp, fm)/pageSize \rceil$.

The $\text{bgpAdd}_{fm}^{B}(\varphi)$ operator can be implemented using some physical operators: the index-nested loop join (index-NLJ) operator or the bind join (BJ) operator. If the bgpAdd operator is implemented via an index-NLJ algorithm, the number of requests ($\#requests^{\text{est}}$) is primarily influenced by the number of solution mappings that the operator receives as input from its subplan φ, represented as $|\mathsf{sols}(\varphi)|$. During execution, for each input solution mapping μ, the join variables (i.e. $\mathsf{joinVars}(B,\varphi)$) of the BGP B are substituted with the bindings from μ, and then a new BGP B' is sent to *fm*. Hence, each solution mapping causes one request and, thus, the number of requests $\#requests^{\mathrm{est}}$ is equivalent to $|\mathsf{sols}(\varphi)|$. Alternatively, the BJ algorithm can be applied to reduce the number of requests by consolidating multiple bindings together with the BGP B into a single request. The number of bindings that can be attached for one request is determined based on the server capabilities, denoted as *blockSize*. Hence, the number of requests depends on the total number of input solution mappings and page size: $\lceil|\mathsf{sols}(\varphi)|/blockSize\rceil$.

For the $\mathrm{tpAdd}_{fm}^{tp}(\varphi)$ operator, the number of requests is similar to the bgpAdd operator in case the federation member provides a SPARQL endpoint. However, if the federation member is a TPF server, only the index-NLJ algorithm can be used. Given that TPF servers adopt a paging mechanism, the total number of pages is at least equal to the number of input solution mappings, as each such solution mapping results in at least one request. In cases in which all matching triples cannot be retrieved with a single page, the total number of pages with all responses together can be estimated based on the join cardinality (joinCard) divided by the page size (*pageSize*).

$$totalPageNum = \max\left\{|\mathsf{sols}(\varphi)|, \left\lceil \frac{\mathsf{joinCard}(tp, fm, \mathsf{sols}(\varphi))}{pageSize} \right\rceil\right\} \tag{2}$$

Furthermore, the number of input solution mappings is $|\mathsf{sols}(\varphi)|$ and each such solution mapping results in at least one request. In our analysis, we make an assumption that the matching triples are evenly distributed for each of such input solution mapping, implying that each request is expected to yield an equivalent number of pages. Consequently, the estimated number of pages for each input solution mapping can be calculated as Equation 3. Hence, the total number of requests for $\mathrm{tpAdd}_{fm}^{tp}(\varphi)$ with a TPF server *fm* is estimated as Equation 4.

$$avgPageNum = \left\lceil \frac{totalPageNum}{|\mathsf{sols}(\varphi)|} \right\rceil \tag{3}$$

$$|\mathsf{sols}(\varphi)| \cdot avgPageNum \tag{4}$$

If federation members provide a brTPF interface, an index-NLJ algorithm or brTPF algorithm can be applied. In case index-NLJ is applied, the total number of requests is the same as for the TPF interface. If the brTPF algorithm is applied, a set of input solution mappings can be attached to a brTPF request. The number of bindings that can be attached to each request is defined as block size (*blockSize*). Under the even distribution assumption, the total number of requests for the brTPF algorithm is estimated as follows.

$$\left\lceil \frac{|\mathsf{sols}(\varphi)|}{blockSize} \right\rceil \cdot \left\lceil \frac{totalPageNum}{\left\lceil \frac{|\mathsf{sols}(\varphi)|}{blockSize} \right\rceil} \right\rceil \tag{5}$$

3.1.2. Total size of shipped request data ($\#reqData^{\mathrm{est}}$)

Regarding the triple pattern request operator, the total size of shipped request data is estimated as the number of RDF terms and variables in all responses together (listed in the third column of Table 2). Specifically, for a single triple pattern request, the value of $\#reqData^{\text{est}}$ is determined as 3, taking into account the three components of the triple pattern. In the case of a basic graph pattern (B), the value of $\#reqData^{\text{est}}$ becomes 3 multiplied by the number of triple patterns (countTP(B)) within the basic graph pattern B.

Concerning the $\mathsf{tpAdd}^{tp}_{fm}(\varphi)$ operator, the size of shipped request data depends on both the number of input solution mappings and variables that are common across the given subquery with input solution mappings, denoted by $\mathsf{joinVars}(tp, \varphi)$. Different physical algorithms exhibit different total sizes of shipped request data. Specifically, for the index-NLJ algorithm, each request contains a single triple pattern where the join variables are substituted by bindings of the input solution mappings. Hence, the size of shipped request data of all requests is three times the number of input solution mappings, $3 \cdot |\mathsf{sols}(\varphi)|$. For the BJ, the size of shipped request data is different for different implementations of BJ (i.e., UNION, FILTER or VALUES). When using UNION to implement bind join, triple patterns with bound variables are consolidated into one request with the UNION operator, resulting in a total size of shipped request data equivalent to that of index-NLJ, despite the size for individual requests being different. In case FILTER is applied, a new request is constructed based on the original triple pattern along with attached join variables as well as corresponding values. Consequently, the size of shipped request data for all queries is three plus $2 \cdot |\mathsf{sols}(\varphi)| \cdot \mathsf{joinVars}(tp, \varphi)$. Using VALUES is similar to using FILTER, but VALUES allows for specifying multiple values for given variables, to which only the bound values in the input solution mappings would be attached. As a result, the overall size of shipped request data decreases to three plus $|\mathsf{sols}(\varphi)| \cdot \mathsf{joinVars}(tp, \varphi)$.

3.1.3. Total size of shipped response data ($\#respRDFterms^{\text{est}}$)

The overall size of shipped response data mainly depends on two factors: the estimated number of solution mappings and the type of interface. Different types of interfaces have different response types. For instance, TPF and brTPF servers return matching triples in their responses, whereas SPARQL endpoints directly respond with solution mappings. For this reason, the size of shipped data in TPF/ brTPF responses depends only on the total number of matching triples, while the size of shipped response data for SPARQL endpoints also depends on the number of variables in the corresponding subquery (as illustrated in Table 3), where $|vars(tp)|$ and $|vars(B)|$ represents the number of distinct variables in the triple pattern tp or BGP B, respectively).

3.1.4. Processing cost by federation engines ($fedProcess^{\text{est}}$)

The amount of work that needs to be done by the federation engine is hard to measure accurately. In an effort to obtain a rough estimation, we adopt a simplified approach by considering the size of intermediate results that need to be processed by the federation engine. The size of intermediate results is listed in the fourth column of Table 2.

3.2. Cost Calculation

In our proposed model, the cost of a query plan is calculated as a weighted sum of multiple metrics. Each metric is computed by aggregating the corresponding measures from all operators encompassed within the plan. For example, the total number of requests for the physical plan is obtained by summing the number of requests for each operator in the plan (see Equation 6).

$$\mathit{total\#requests}^{\text{est}} = \sum_{\text{op} \in \mathit{subPlan}} \mathit{\#requests}^{\text{est}}(\text{op}) \tag{6}$$

In addition, our cost model accommodates parallel evaluation of operators, thereby capturing the effects of parallelism. If the root operator of a physical plan is an SHJ or a union, the work performed by the corresponding two subplans (referred to as $\mathit{subPlan}_1$ and $\mathit{subPlan}_2$) can occur in parallel. The overall work processed by the federation engine ($\mathit{totalfedProcess}^{\text{est}}$) is calculated using the Equation 7, which applies specifically for SHJ and union operators. In addition, this formula for parallelism is only used for $\mathit{fedProcess}^{\text{est}}$ but not for any communication costs.

$$\mathit{fedProcess}^{\text{est}}(\mathit{rootOp}) + \mathit{max}\left\{ \sum_{\text{op} \in \mathit{subPlan}_1} \mathit{fedProcess}^{\text{est}}(\text{op}), \sum_{\text{op} \in \mathit{subPlan}_2} \mathit{fedProcess}^{\text{est}}(\text{op}) \right\} \tag{7}$$

Example 4. To illustrate these concepts, Table 4 presents the metric values for two physical plans (1b and 1c) for our running example. The total number of requests is computed by summing up values for all operators in each plan. Furthermore, the total work processed by the federation engine is evaluated with considering parallelism. For instance, the total amount of work by the federation engine for the physical plan (1b) is 2105, which is calculated by: $5 + \mathit{max}(80, 1100 + \mathit{max}(100, 1000))$.

Table 4
Cost calculation for plan(b) and plan(c) of the running example (in Figure 1)

operators	$\mathit{\#requests}^{\text{est}}$	$\mathit{\#reqData}^{\text{est}}$	$\mathit{\#respRDFterms}^{\text{est}}$	$\mathit{fedProcess}^{\text{est}}$
join	0	0	0	5
$\text{req}^{tp_1}_{fm_1}$	1	3	$80 \cdot 1$	80
union	0	0	0	100+1000
$\text{req}^{tp_2}_{fm_2}$	1	3	$100 \cdot 1$	100
$\text{req}^{tp_2}_{fm_3}$	1000/100	3	$3 \cdot 1000$	1000
sum	12	9	3180	2105

operators	$\mathit{\#requests}^{\text{est}}$	$\mathit{\#reqData}^{\text{est}}$	$\mathit{\#respRDFterms}^{\text{est}}$	$\mathit{fedProcess}^{\text{est}}$
union	0	0	0	5
$\text{tpAdd}^{tp_2}_{fm_2}(\varphi)$	$\lceil 80/30 \rceil$	$3+2 \cdot 80 \cdot 1$	0	0
$\text{tpAdd}^{tp_2}_{fm_3}(\varphi)$	80	$3 \cdot 80$	$3 \cdot 5$	5
$\text{req}^{tp_1}_{fm_1}$	1	3	$80 \cdot 1$	80
sum	84	406	95	90

To determine the total cost of a physical plan while considering the combination of different cost metrics, a weighted sum can be employed (see Equation 8). The weights for each metric may be adjusted based on the specific scenario and user requirements. For instance, in a scenario where the network delay is substantial and the number of requests dominates the total processing time, using a greater value for ω_i may be beneficial in identifying a suitable plan for the scenario. In the evaluation of our cost model presented in this paper, a uniform weight of 1 is assigned to all metrics, maintaining equal importance across the board.

$$\begin{aligned} \mathit{totalCost} = {} & \omega_i \cdot \mathit{total\#requests}^{\text{est}} + \omega_j \cdot \mathit{total\#reqData}^{\text{est}} + \\ & \omega_m \cdot \mathit{total\#respRDFterms}^{\text{est}} + \omega_n \cdot \mathit{totalfedProcess}^{\text{est}} \end{aligned} \tag{8}$$

3.3. Cardinality Estimation

As indicated in the preceding section, the metrics of our cost model rely on the expected cardinality of intermediate results. It is noteworthy that our cost model is not tied to any specific approach for estimating such cardinalities. Instead, our cost model is designed to accommodate any approach that can be utilized to obtain such cardinality estimates.

For the evaluation of our cost model in this paper, we are adopting a comparatively simple method for cardinality estimation [13]. This method is based on issuing requests to the federation members. For each subquery with only one req operator, the number of triples that match the given graph pattern can be obtained by issuing a request to the corresponding federation member. The concrete type of this request depends on the type of interface employed by the federation member. If the federation member provides a SPARQL endpoint, the cardinality of a graph pattern (tp or B) can be obtained by constructing a SELECT COUNT query for each subquery. If the federation member uses a TPF interface or a brTPF interface, a request can be issued to retrieve the first page of matching triples, and the count estimates can be obtained from the metadata of the retrieved pages.

In the case of subqueries involving operators other than req operator, the expected cardinality of the subquery can be determined by recursively applying a cardinality estimation function to each subquery. Specifically, the cardinality of join operations involving two subqueries in the approach that we adopt for the evaluation in this paper is estimated as the minimum of their respective cardinalities, while the cardinality of union operations is computed by summing the individual cardinalities [13]. However, as mentioned above, our cost model can also be used in combination with other cardinality estimation approaches.

4. Evaluation

This section presents an empirical evaluation in which we study the effectiveness of our cost model by using Heling and Acosta's approach [13] as a baseline. We first describe the implementation that we use and the experiment setup, including the datasets, queries, and evaluation metrics. Thereafter, we present the measurements and discuss our observations.

4.1. Implementation

We implemented the proposed cost model in our query federation engine HeFQUIN[1] in combination with a simple greedy plan-enumeration algorithm. This algorithm produces left-deep execution plans for source assignments consisting of joins over subplans that may be individual request operators or unions of requests. In particular, after picking a first such subplan based on the estimated cost, and using that subplan as the starting point for building up a left-deep plan, the algorithm iterates over the remaining subplans that can be joined with the partial left-deep plan that has been built so far (i.e., ignoring subplans that would introduce cross products, unless there are no other subplans left). In each step of this iteration, the algorithm considers all available subplans in combination with all possible algorithms for joining each such subplan with the current left-deep plan, and then greedily picks the least costly of these options. Additionally, we extended HeFQUIN with an implementation of the approach that Heling and Acosta proposed [13] to produce left-deep execution plans for the aforementioned

[1]https://github.com/LiUSemWeb/HeFQUIN

types of source assignments. For the experiments, such source assignments are given to the engine in the form of SPARQL queries with SERVICE clauses.

4.2. Experiment Setup

All experiments described in this paper have been performed on a server machine with two 8-core Intel Xeon E5-2667 v3@3.20GHz CPUs and 256 GB of RAM. The machine runs a 64-bit Debian GNU/Linux 10 server operation system. On this machine, we use KOBE [16] (a benchmarking system based on Kubernetes infrastructures) to containerize and configure federations of RDF datasets, queries, federation engines and experiments. For setting up heterogeneous federations, we use Virtuoso v7.20 for configuring a SPARQL endpoint, the Server.js (v3.3.0)[2] and Java brTPF server[3] to deploy TPF and brTPF servers with HDT backends. The source code and experimental results are provided online.[4]

Datasets and Queries: The datasets and queries we use for the evaluation are from FedBench [4], which involves 9 datasets containing a total number of 10M triples. Each dataset can be exposed using different types of interfaces. We use the same types of federations as used by Heling and Acosta. These two federations differ in terms of the types of interfaces used by the federation members (illustrated in Table 5). We use a total of 25 queries from Cross Domain (cd1–7), Life Science (ls1–7) and Linked Data (ld1–11). For these queries, we manually applied the source selection approach of FedX [5] in order to produce source assignments as assumed as input for the tested approaches (cf. Section 4.1).

Table 5
Two Heterogeneous Federations of RDF Data Sources (differ in terms of the types of interfaces)

	GeoNames	DBpedia	ChEBI	Jamendo	Drugbank	SWDF	LinkedMDB	Kegg	NYTimes
Fed I	SPARQL	SPARQL	SPARQL	TPF	TPF	TPF	brTPF	brTPF	brTPF
Fed II	TPF	TPF	TPF	SPARQL	SPARQL	SPARQL	brTPF	brTPF	brTPF

Evaluation Metrics We report performance metrics based on the following definitions: i) *Query execution time* (**QET**) is the amount of time elapsed since the evaluation of the query plan starts until the complete answer has been received. ii) *Number of requests* (*#requests*) is the number of requests that are issued to federation members during the execution of the query. iii) *Size of data transferred* (*#respRDFterms*) is represented as the total number of RDF terms that are transferred from federation members to the federation engine. iv) *Local work* (*fedProcess*) is the amount of work that needs to be done by the federation engine.

4.3. Experimental Results

Figures 2a and 2b illustrate the average query execution time per query for federations Fed I and Fed II, respectively; Figures 2c and 2d illustrate the respective number of requests sent per query, and Figures 2e and 2f illustrate the total size of data transferred from federation members to the federation engine. We observe that there is not much difference between the two

[2]https://github.com/LinkedDataFragments/Server.js
[3]https://github.com/LiUSemWeb/Server.Java/tree/feature-brtpf
[4]https://github.com/LiUSemWeb/HeFQUIN-DMKG2023-Experiments

(a) Query execution time (Fed I)

(b) Query execution time (Fed II)

(c) Overall number of requests (Fed I)

(d) Overall number of requests (Fed II)

(e) Size of data transferred (Fed I)

(f) Size of data transferred (Fed II)

Figure 2: Measurement results for each query in both federations

approaches for about half of queries (11 out of 24 queries). Compared to the baseline approach, it also shows up that for some queries, the plans selected using our cost model require less data to be retrieved from federation members without a significant increase in query execution time (some queries have a significant reduction in query execution time). Table 6 illustrates these queries for which there is a significant difference in the measurements (i.e., where the respective greater value is at least double the smaller value). All these are queries for which the plans selected by the two approaches were different.

Based on Table 6, we observe that the number of queries that shows improvement varies across the federations since the interface's capabilities affect different queries. We find that, for these queries, even though the plans selected using our cost model issue more requests to the federation members, the query execution times are reduced or without significant increase, but the total size of data transferred is significantly decreased! The best reduction in query execution time occurs on the query ld8 over Fed I, which is nearly 300 times. Further, the plans selected using our cost model produces more complete result than those using the baseline approach, which is caused by the SPARQL endpoint as the Virtuoso server does not produce

Table 6
Measurements for the queries for which there is a significant difference between the two approaches (i.e., where the respective greater value is at least double the smaller value). The blue highlighting indicates cases in which our cost model is better than the baseline, while orange highlighting indicates cases in which it is the other way around. Red-colored numbers (additionally marked with an asterisk *) are cases in which the produced query result was incomplete.

Federation	Measurement	Approach	cd3	ld1	ld3	ld4	ld7	ld8	ls3	ls4	ls5	ls6	ls7
Fed I	QET	baseline	26	491	621	156	41,752*	47,521*	2,647	18	1,075	168	496
		cost model	27	476	862	238	253	161	2,906	20	1,733	67	539
	#requests	baseline	26	912	1,273	249	20*	280*	703	10	229	36	210
		cost model	26	958	1,945	465	83	323	5,051	11	2,887	61	210
	#dataTransferred	baseline	32	13,806	13,719	1,464	2,103,708*	2,132,372*	337,108	51	105,414	7,434	24,818
		cost model	32	1,974	2,460	1,545	3,648	792	98,818	54	20,091	855	24,818
	#resultSize	baseline	2	309	162	50	592*	18*	9,054	3	393	28	1,620
		cost model	2	309	162	50	1,216	22	9,054	3	393	28	1,620
Fed II	QET	baseline	37,473	378	429	11	672	106	14,040	17	3,533	71	5,178
		cost model	1,400	409	664	10	707	153	2,643	15	2,099	75	993
	#requests	baseline	3,158	874	979	1	1,266	249	962	5	1,709	85	1,106
		cost model	2,232	872	1,557	1	1,263	300	4,805	10	3,981	85	1,367
	#dataTransferred	baseline	951,954	1,146	1,088	200	7,296	419	346,984	36	515,706	754	316,369
		cost model	28,683	1,146	1,470	200	7,296	557	55,884	36	12,891	754	16,504
	#resultSize	baseline	2	309	162	50	1,216	22	9,054	3	393	28	1,620
		cost model	2	309	162	50	1,216	22	9,054	3	393	28	1,620

complete results (at most 2^{20}), but this issue does not happen for the plans selected using our cost model as this cost model not only considers the number of requests but also the size of data transferred. This query is a case like the one illustrated in the running example 1c. Due to this rewriting, the number of requests increases a bit on the number of requests (from 20 to 80), but the size of data transferred is reduced by more than 2600 times.

5. Summary and Future Work

In this paper we propose a cost model designed for query optimization over heterogeneous federations of RDF data sources. This cost model can be seamlessly integrated with a greedy algorithm or a dynamic programming algorithm. With experimental evaluation, we have demonstrated that the query plan selected using the cost model requires less data to be transferred from federation members to the federation engine compared to the baseline approach. Furthermore, for certain queries, leveraging our cost model achieves a better plan that not only delivers complete results but also achieves shorter execution times.

To attain a more comprehensive understanding of the model's capabilities and limitations, we intend to conduct a more extensive evaluation of the cost model in our future work. This evaluation shall involve testing with a diverse range of federation setups, exploring additional types of interfaces, and accounting for network latency. Additionally, we aim to enhance our cost model by also incorporating the processing cost incurred by the federation members. Furthermore, we will explore opportunities to accommodate more expressive forms of query patterns to further enhance the applicability of our cost model.

Acknowledgments

This work was funded by the National Graduate School in Computer Science, Sweden (CUGS), and by the Swedish Research Council (Vetenskapsrådet, project reg. no. 2019-05655).

References

[1] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, E. Ruckhaus, ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints, in: Proc. of the 11th International Semantic Web Conference (ISWC), 2011.

[2] A. Charalambidis, A. Troumpoukis, S. Konstantopoulos, SemaGrow: Optimizing Federated SPARQL Queries, in: Proc. of the 11th Int. Conf. on Semantic Systems, 2015.

[3] M. Saleem, A. Potocki, T. Soru, O. Hartig, A. N. Ngomo, CostFed: Cost-Based Query Optimization for SPARQL Endpoint Federation, in: Proc. of the 14th Int. Conf. on Semantic Systems, 2018.

[4] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, T. Tran, FedBench: A Benchmark Suite for Federated Semantic Data Query Processing, in: Proceedings of the 10th International Semantic Web Conference (ISWC), 2011.

[5] A. Schwarte, P. Haase, K. Hose, R. Schenkel, M. Schmidt, FedX: Optimization Techniques for Federated Query Processing on Linked Data, in: Proceedings of the 10th International Semantic Web Conference (ISWC), 2011.

[6] M. Vidal, S. Castillo, M. Acosta, G. Montoya, G. Palma, On the Selection of SPARQL Endpoints to Efficiently Execute Federated SPARQL Queries, Trans. Large-Scale Data- and Knowledge-Centered Systems 25 (2016).

[7] M. Saleem, A.-C. Ngonga Ngomo, HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation, in: Proc. of the European Sem. Web Conf. (ESWC), 2014.

[8] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, P. Colpaert, Triple Pattern Fragments: A Low-Cost Knowledge Graph Interface for the Web, Journal of Web Semantics 37 (2016).

[9] O. Hartig, C. Buil-Aranda, Bindings-Restricted Triple Pattern Fragments, in: 15th Ontologies, Databases, and Applications of Semantics (ODBASE), 2016.

[10] T. Minier, H. Skaf-Molli, P. Molli, SaGe: Web Preemption for Public SPARQL Query Services, in: Proceedings of the Web Conference (WWW), 2019.

[11] A. Azzam, J. D. Fernández, M. Acosta, M. Beno, A. Polleres, SMART-KG: Hybrid Shipping for SPARQL Querying on the Web, in: Proc. of The Web Conference (WWW), 2020.

[12] S. Cheng, O. Hartig, Source Selection for SPARQL Endpoints: Fit for Heterogeneous Federations of RDF Data Sources?, in: Proceedings of the 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs (QuWeDa), 2022.

[13] L. Heling, M. Acosta, Federated sparql query processing over heterogeneous linked data fragments, in: Proceedings of the ACM Web Conference, 2022.

[14] S. Cheng, O. Hartig, FedQPL: A Language for Logical Query Plans over Heterogeneous Federations of RDF Data Sources, in: Proc. of the 22nd Int. Conf. on Information Integration and Web-based Applications & Services (iiWAS), 2020.

[15] C. B. Aranda, A. Polleres, J. Umbrich, Strategies for executing federated queries in SPARQL1.1, in: 13th International Semantic Web Conference, 2014.

[16] C. Kostopoulos, G. Mouchakis, A. Troumpoukis, N. Prokopaki-Kostopoulou, A. Charalambidis, S. Konstantopoulos, KOBE: Cloud-native Open Benchmarking Engine for Federated Query Processors, in: European Semantic Web Conference, 2021.