Multi-query Optimization in Federated RDF Systems

Peng Peng[1(✉)], Lei Zou[2,3], M. Tamer Özsu[4], and Dongyan Zhao[2]

[1] Hunan University, Changsha, China
hnu16pp@hnu.edu.cn
[2] Peking University, Beijing, China
{zoulei,zhaodongyan}@pku.edu.cn
[3] Beijing Institute of Big Data Research, Beijing, China
[4] University of Waterloo, Waterloo, Canada
tamer.ozsu@uwaterloo.ca

Abstract. This paper revisits the classical problem of multiple query optimization in federated RDF systems. We propose a heuristic query rewriting-based approach to share the common computation during evaluation of multiple queries while considering the cost of both query evaluation and data shipment. Furthermore, we propose an efficient method to use the interconnection topology between RDF sources to filter out irrelevant sources and share the common computation of intermediate results joining. The experiments over both real and synthetic RDF datasets show that our techniques are efficient.

1 Introduction

Now, many data providers publish, share and interlink their datasets using open standards such as RDF and SPARQL [3]. In RDF, data is represented as triples of the form ⟨subject, property, object⟩. By deeming subjects and objects as the vertices, and properties as the directed edges from the corresponding subjects to objects, an RDF dataset can be viewed as a directed labeled graph G. On the other hand, SPARQL is a query language to retrieve and manipulate data stored in RDF format. When many data providers publish their RDF data, they often store the actual triple files at their own *autonomous* sites some of which are *SPARQL endpoints* that can execute SPARQL queries. An autonomous site with a SPARQL endpoint is called an *RDF source* in this paper.

To integrate and provide transparent access over many RDF sources, federated RDF systems have been proposed [1,5,14,15,18], in which, a control site is introduced to provide a common interface for users to issue SPARQL queries. However, existing federated RDF systems only consider query evaluation for a single query and miss the opportunity for multiple query optimization. Real SPARQL query workloads reveal that many SPARQL queries are often posed simultaneously. For example, in a real SPARQL query workload over DBPedia[1],

[1] http://aksw.org/Projects/DBPSB.html.

J. Pei et al. (Eds.): DASFAA 2018, LNCS 10827, pp. 745–765, 2018.
https://doi.org/10.1007/978-3-319-91452-7_48

there are in average more than six SPARQL queries per second. Furthermore, 97% queries in the workload are isomorphic to one of the 163 frequent patterns. There is room for sharing computation when executing these queries. Therefore, it is desirable to design a multiple SPARQL query optimization strategy.

Consider a batch of queries (e.g., Q_1, Q_2 and Q_3 in Fig. 1) that are posed simultaneously over the federated RDF system in Fig. 1. Specifically, Q_1 is to find out all news about Canada; Q_2 retrieves all people who graduated from Canadian universities; and Q_3 is to retrieve all semantic web-related workshops held in Canada. Obviously, there are some common substructures over these three queries, which suggests the possibility of some sharing computation. This motivates us to revisit the classical problem of multiple query optimization in the context of federated RDF systems.

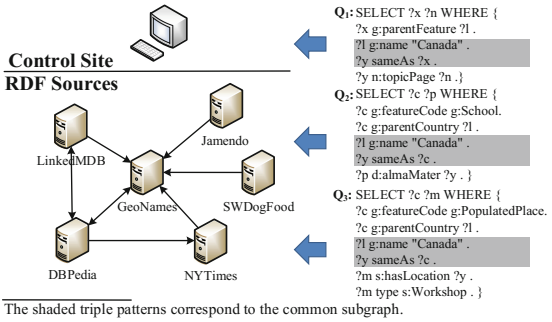


Fig. 1. Multiple federated SPARQL queries

Although multiple query optimization has been well studied in distributed relational databases [12], some techniques commonly referred to as data movement and data/query shipping [10] are not easily applicable in federated RDF systems. In federated RDF systems, we cannot require one source to send intermediate results directly to another source for join processing [7]. Meanwhile, there is only one proposal about multiple SPARQL query optimization in literature [11], but *only in the centralized environment*, where all RDF datasets are collected in one physical database. It is a method based on query rewriting. However, there is no data shipment in the centralized environment, and rewriting multiple queries as [11] in federated RDF systems may generate many remote requests for data shipment as shown in the experiments.

To the best of our knowledge, this is the first study of multiple SPARQL query optimization over federated RDF systems, with the objective to reduce the query response time and the number of remote requests. In this paper, we propose a cost model-driven greedy approach based on query rewriting for multiple query optimization in federated RDF systems. We use both "OPTIONAL" and "FILTER" clauses of SPARQL to rewrite multiple queries with commonalities while considering the cost for both query evaluation and data shipment.

In addition, we also study relevant source selection and partial match joins in federated RDF systems, which do not arise in the centralized counterpart. We propose a topology structure-based source selection and study how to share common computation in joining partial matches in federated RDF systems.

2 Background

In this section, we review the terminology that we use throughout this paper. First, in the context of federated RDF systems, an RDF graph G is a combination of many RDF graphs located at different source sites.

Definition 1 ***(Federated RDF System).*** *A* federated RDF system *is defined as $W = (S, g, d)$, where (1) S is a set of source sites that can be obtained by looking up URIs in an implementation of W; (2) $g : S \rightarrow 2^{E(G)}$ is a mapping that associates each source with a subgraph of RDF graph G; and (3) $d : V(G) \rightarrow S$ is a partial, surjective mapping where looking up URI of vertex u results in the retrieval of the source represented by $d(u) \in S$. $d(u)$ is called the* host *source of u, and is unique for a given URL of vertex u.*

Obviously, RDF graph G is formed by collecting all subgraphs at different sources, i.e., $\bigcup_{s \in S} g(s) = G$. Note that, although there may be multiple RDF sources that describe an entity identified by vertex u, u can be only dereferenced by the host source $d(u)$.

On the other hand, SPARQL is a structured query language over RDF where primary building block is the basic graph pattern (BGP).

Definition 2 ***(Basic Graph Pattern).*** *A* basic graph pattern *is denoted as $Q = \{V(Q), E(Q), L\}$, where $V(Q) \subseteq V(G) \cup V_{Var}$ is a set of vertices, where $V(G)$ denotes vertices in RDF graph G and V_{Var} is a set of variables; $E(Q) \subseteq V(Q) \times V(Q)$ is a set of edges in Q; each edge e in $E(Q)$ either has a property in L or the property is a variable.*

In federated RDF systems, a BGP match may span over different sources as follows.

Definition 3 ***(BGP Match over Federated RDF System).*** *Consider an RDF graph G, a federated RDF system $W = (S, g, d)$ and a BGP Q that has n vertices $\{v_1, \ldots, v_n\}$. For $S' \subseteq S$, a subgraph M of $\bigcup_{s \in S'} g(s)$ with n vertices $\{u_1, \ldots, u_n\}$ is said to be a* match *of Q if and only if there exists a* function *μ from $\{v_1, \ldots, v_n\}$ to $\{u_1, \ldots, u_n\}$, where the following conditions hold: (1) if v_i is not a variable, $\mu(v_i)$ and v_i have the same URI or literal value ($1 \leq i \leq n$); (2) if v_i is a variable, there is no constraint over $\mu(v_i)$ except that $\mu(v_i) \in \{u_1, \ldots, u_n\}$; (3) if there exists an edge $\overrightarrow{v_i v_j}$ in Q, there also exists an edge $\overrightarrow{\mu(v_i)\mu(v_j)}$ in $\bigcup_{s \in S'} g(s)$; furthermore, $\overrightarrow{\mu(v_i)\mu(v_j)}$ has the same property as $\overrightarrow{v_i v_j}$ unless the label of $\overrightarrow{v_i v_j}$ is a variable.*

The set of matches for Q over S' is denoted as $[\![Q]\!]_{S'}$.

Our notion of a SPARQL query can be defined recursively as follows by combining BGPs using the following standard SPARQL algebra operations.

Definition 4 ***(SPARQL Query).*** *Any BGP is a SPARQL query. If Q_1 and Q_2 are SPARQL queries, then expressions $(Q_1\ AND\ Q_2)$, $(Q_1\ UNION\ Q_2)$, $(Q_1\ OPT\ Q_2)$ and $(Q_1\ FILTER\ F)$ are also SPARQL queries.*

The results of a query Q over sources S' are defined as follows.

Definition 5 ***(SPARQL Result over Federated RDF System).*** *Given a federated RDF system $W = (S, g, d)$, the result of a SPARQL query Q over a set of sources $S' \subseteq S$, denoted as $[\![Q]\!]$, is defined recursively as follows:*

1. *If Q is a BGP, $[\![Q]\!]_{S'}$ is defined in Definition 3.*
2. *If $Q = Q_1\ AND\ Q_2$, then $[\![Q]\!]_{S'} = [\![Q_1]\!]_{S'} \bowtie [\![Q_2]\!]_{S'}$*
3. *If $Q = Q_1\ UNION\ Q_2$, then $[\![Q]\!]_{S'} = [\![Q_1]\!]_{S'} \cup [\![Q_2]\!]_{S'}$*
4. *If $Q = Q_1\ OPT\ Q_2$, then $[\![Q]\!]_{S'} = ([\![Q_1]\!]_{S'} \bowtie [\![Q_2]\!]_{S'}) \cup ([\![Q_1]\!]_{S'} \setminus [\![Q_2]\!]_{S'})$*
5. *If $Q = Q_1\ FILTER\ F$, then $[\![Q]\!]_{S'} = \Theta_F([\![Q_1]\!]_{S'})$*

If $S' = S$, i.e., the whole federated RDF system W, we call $[\![Q]\!]_S$ the results of Q over federated RDF system W.

The problem to be studied in this paper is defined as follows:

Given a set of SPARQL queries $\mathcal{Q}$ and a federated RDF system $W = (S, g, d)$, our problem is to find the results of each query in $\mathcal{Q}$ over W.

3 Framework

A federated RDF system consists of a control site as well as some RDF sources. The control site is amenable to receive the SPARQL queries, decompose them into several subqueries on their relevant sources and do some global optimizations, while the RDF sources actually store the RDF graphs. Generally speaking, our approach consists of five steps: *query decomposition and source selection*, *query rewriting*, *local evaluation*, *postprocessing* and *partial match join* (see Fig. 2). We briefly review the five steps before we discuss them in details in upcoming sections. Note that only *local evaluation* is conducted over the remote sources and the other four steps work at the control site.

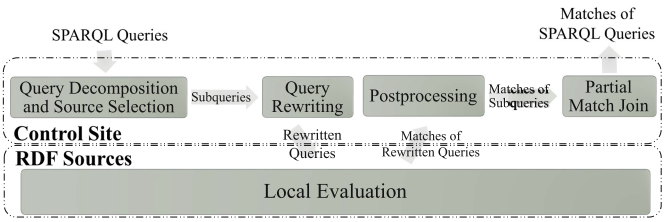


Fig. 2. Scheme for federated SPARQL query processing

First, in the step of query decomposition and source selection, we decompose the input queries into a set of subqueries expressed over relevant sources. Given a batch of SPARQL queries $\{Q_1, \ldots, Q_n\}$ posed simultaneously, we decompose each query Q_i into $\{q_i^1@S(q_1^1), \ldots, q_i^{m_i}@S(q_i^{m_i})\}$ where $S(q_i^j)$ is the set of relevant sources for subquery q_i^j (see Sect. 4). Then, in the step of query rewriting, we use FILTER and OPTIONAL operators to rewrite these subqueries as fewer queries to reduce the number of remote requests and improve the query performance. After query rewriting, we obtain a set of rewritten queries $\hat{\mathcal{Q}}_s$ that will be sent to source s (see Sect. 5). In the step of local evaluation, we send the set of rewritten queries to their relevant sources and evaluate them there, and the results will be returned back to the control site (see Sect. 6). In the step of post-processing, the control site checks each local evaluation result against each query in $\mathcal{Q}_s$ (see Sect. 6). Last, in the step of partial match join, for each subquery q_i^j, we collect and join all the matches at each relevant source in $S(q_i^j)$. Considering the context of multiple SPARQL queries over a federated RDF system, we propose an optimized solution to avoid duplicate computation in join processing (see Sect. 7).

4 Query Decomposition and Source Selection

Most existing solutions [1,5,6,13–15,18] decompose the input query based on the triple patterns. Given a query, each triple pattern maps to a subquery, and they select the relevant sources based on the values of its subject, property and object. If a source is exclusively selected for some connected triple patterns, they can be combined together to form a larger subquery. Note that, if a group of triple patterns shares exactly the same set of multiple RDF sources, they cannot be combined, because combining them together may miss some matches crossing different RDF sources.

For example, the existing solutions decompose the query Q_1 in Fig. 1 into three subqueries q_1^1, q_1^2 and q_1^3, as shown in Fig. 3. q_1^1 has a relevant source "GeoNames"; q_1^2 has five sources except for "Jamendo" and q_1^3 has a relevant source "NYTimes".

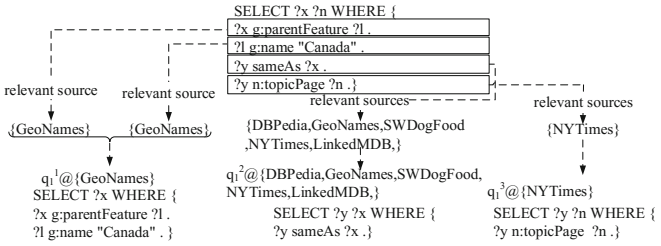
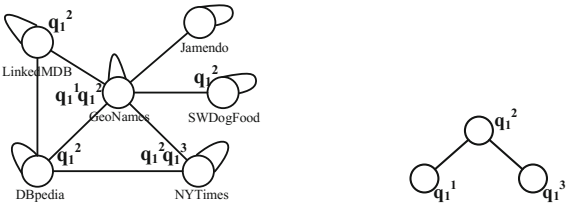


Fig. 3. Existing query decomposition and source selection result for Q_1

However, the existing solutions may overestimate the set of relevant sources. To filter out more irrelevant sources, we employ the interconnection structures among sources. A crawler proposed in [16] can be used for the federated RDF system to figure out crossing edges between different sources, and then we define the *source topology graph* (maintained in the control site) as follows:

Definition 6 ***(Source Topology Graph).*** *Given a federated RDF system* $W = (S, g, d)$*, the corresponding* source topology graph $T = (V(T), E(T))$ *is an* undirected *graph, where (1) each vertex in* $V(T)$ *corresponds to a source* $s_i \in S$*; (2) there is an edge between vertices* s_i *and* s_j *in* T*, if and only if there is at least one edge* $\overrightarrow{u_i u_j} \in g(s_i)$ *(or* $\overrightarrow{u_j u_i} \in g(s_i)$ *or* $\overrightarrow{u_i u_j} \in g(s_j)$ *or* $\overrightarrow{u_j u_i} \in g(s_j)$*), where* $d(u_i) = s_i$ *and* $d(u_j) = s_j$*.*



(a) Annotated Source Topology Graph for Q_1 (b) Join Graph $JG(Q_1)$ of Q_1

Fig. 4. Example join graph and annotated source topology graph for Q_1

We propose a source topology graph (STG)-based pruning rule to filter out irrelevant sources. We firstly annotate each source in STG with its relevant decomposed subqueries. For example, "NYtimes" is a relevant source to q_1^3, we annotate "NYTimes" in STG with q_1^3. Then, each query leads to an annotated source topology graph. Figure 4(a) shows the annotated STG T^* for query Q_1. Meanwhile, for a BGP Q, we build a *join graph* (denoted as $JG(Q)$), where each vertex indicates a subquery of Q and an edge between two vertices in the join graph if and only if the corresponding subqueries are connected in the original SPARQL query. Figure 4(b) shows the join graph $JG(Q_1)$.

Given a join graph $JG(Q)$ and the annotated source topology graph T^*, we find all homomorphism matches of $JG(Q)$ over T^* (SPARQL semantic is based on homomorphism). If a subquery q does not map to a source s in any match, s is not a relevant source of subquery q. We formalize this observation in Theorem 1.

Theorem 1. *Given a join graph* $JG(Q)$ *and its corresponding annotated source topology graph* T^**, for a subquery* q*, if there exists a homomorphism match* m *of* Q^* *over* T^* *containing* q*, then* $m(q)$ *is the relevant source of* q*.*

Proof. Given a source s pruned by the theorem for subquery q, there do not exist any homomorphism matches of q over T^* that contains s. Then, there exists another subquery q' of which relevant sources do not contain triples that

can join with results of q in s through $JG(Q)$. Hence, s cannot contribute any final results and can be pruned. □

Based on the above pruning rule, for example, sources "LinkedMDB" or "SWDogFood" do not match q_1^2, so both of them can be pruned from the relevant sources of q_1^2.

5 Query Rewriting

As mentioned in Sect. 1, when multiple SPARQL queries are posed simultaneously, there is room for sharing computation when executing these queries. This section proposes a cost-driven query rewriting scheme to rewrite them (at the control site) into fewer SPARQL queries. To simplify presentation, we assume that the SPARQL queries originally issued at the control site are BGPs, since BGPs are the building block of SPARQL queries. Our solution is easily extensible to handle general SPARQL queries.

5.1 Intuition

We first discuss how to rewrite multiple queries with the common substructure into a single SPARQL query. Specifically, we utilize "OPTIONAL" and "FILTER" operators to make use of common structures among different queries for rewriting.

FILTER-Based Rewriting. Let us consider two subqueries $q_2^1@\{GeoNames\}$ and $q_3^1@\{GeoNames\}$ in Fig. 5, which are decomposed from Q_2 and Q_3. Although they distinguish from each other at the first triple pattern, the only difference is constant bounded to objects in the first triple pattern. We can rewrite the two queries using FILTER, as shown in Fig. 5. In other words, if some subqueries issued at the same source have the common structure except the constants on some vertices (subject or object positions), they can be rewritten as a single query with FILTER.

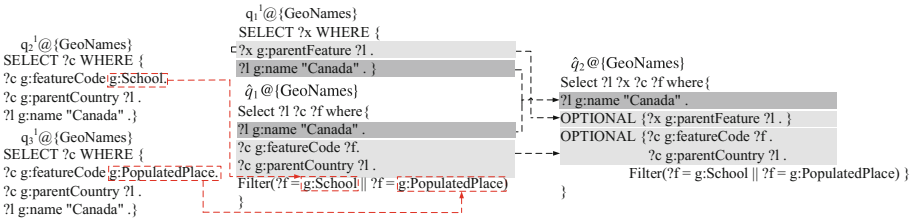


Fig. 5. Rewriting queries using OPTIONAL and FILTER operators

Formally, if a set of subqueries $\{q_1@\{s\}, q_2@\{s\}, \ldots, q_n@\{s\}\}$ has the same structure p except for some vertex labels (i.e., constants on vertices), we rewrite them as follows.

$$\hat{q}@\{s\} = p\ FILTER(\bigvee_{1 \le i \le n} (\bigwedge_{v \in V(q_i)} f_i(v) = v))@\{s\}$$

where f_i is a bijective isomorphism function from q_i to p.

For the FILTER operators, we can perform selection over main pattern results. Thus, the result cardinality of SPARQL with FILTER operator is upper bounded by result cardinality of its main graph pattern. In addition, unlike OPTIONAL operators, FILTER operators do not introduce extra joins and only generate a little more partial matches. Hence, the cost of data shipment will increase little.

OPTIONAL-Based Rewriting. Let us consider two subqueries over GeoNames, $q_1^1@\{GeoNames\}$ and $\hat{q}_1@\{GeoNames\}$, as shown in Fig. 5. They share a common substructure, i.e., triple pattern "?l g:name "Canada"". Therefore, they can be rewritten to a single query, where "?l g:name "Canada"" maps to the main pattern. The subgraphs that q_1^1 and q_2^1 minus "?l g:name "Canada"" map to two OPTIONAL clauses, respectively. The query rewriting is illustrated in Fig. 5. The rewritten query can avoid one remote request for GeoNames.

Formally, given a set of subqueries $\{q_1@\{s\}, q_2@\{s\}, \ldots, q_n@\{s\}\}$ over a source s, if p is the common subgraph among $q_1,\ldots,q_n$, we rewrite these subqueries as a query with OPTIONAL operators as follows.

$$\hat{q}@\{s\} = p\ OPT\ (q_1 - p)\ OPT\ (q_2 - p)\ \ldots\ OPT\ (q_n - p)@\{s\}$$

Because existing RDF stores implement OPTIONAL operators using left-joins, the result cardinality of a SPARQL query with OPTIONAL operator is also upper bounded by result cardinality of its main graph pattern. Thus, the cardinality of the rewritten query does not increase much. Meanwhile, using OPTIONAL operators adds extra left-joins for each optional clause which results in larger local evaluation cost. This needs a trade off between many queries being rewritten together and many optional clauses. We propose a cost model to measure the rewriting benefits in Sect. 5.2.

5.2 Cost Model

The cost of a rewriting strategy can be expressed with respect to the total time. The total time is the sum of all time (also referred to as cost) components. There are two time components for evaluating a rewriting strategy: time for local evaluation and time for data shipment. We discuss the two components respectively in the following sections.

Cost Model for BGPs. As mentioned in [11], selective triple patterns in BGP have higher priorities in evaluation. We verify the principle in two real and synthetic RDF repositories, DBpedia and WatDiv. For DBpedia, we randomly sample 10000 queries from the real SPARQL workload; for WatDiv, we generate 12500 queries from 125 templates provided in [2]. Given a triple pattern e, its selectivity is defined as $sel(e) = \frac{|[\![e]\!]|}{|E(G)|}$, where $|[\![e]\!]|$ denotes the number of

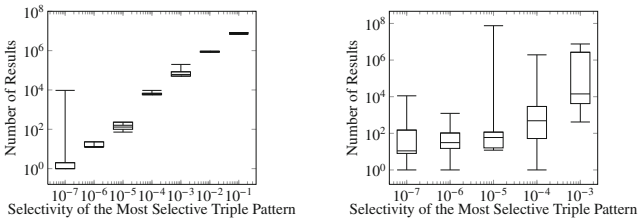
matches of e and $|E(G)|$ denotes the number of edges in RDF graph G. The experimental results are shown in Fig. 6. As the experimental results show, the cardinality of a query is positively associated with the selectivity of the most selective triple pattern for both real and synthetic datasets.

Based on the above observation, we define the cardinality of evaluating a basic graph pattern Q as follows.

$$card(Q) = min_{e \in E(Q)}\{sel(e)\}$$

where $sel(e)$ is the selectivity of triple pattern e in Q.

In real applications, for estimating the selectivity of triple pattern, we can employ the heuristics introduced by [19]. Meanwhile, the relative coefficients, T_{CPU} and T_{MSG}, are greatly influenced by the resources of each RDF sources and the network topology of the federated RDF system, which can be estimated offline.



(a) DBPedia (b) WatDiv 10M

Fig. 6. Relationship between the cardinality and the most selective triple pattern

Cost Model for General SPARQLs. Then, we extend the cost model to handle general SPARQLs. The design of our cost model is motivated by the way in which a SPARQL query is evaluated on popular RDF stores. This includes a well-justified principle that the graph patterns in OPTIONAL clauses and the expressions in FILTER operators are evaluated on the results of the main pattern (for the fact that the graph pattern in the OPTIONAL clause is a left-join and the FILTER operator is selection) [11]. Hence, given a SPARQL Q, its cardinality $card(Q)$ is defined as follows.

$$card(Q) = \begin{cases} \min_{e \in E(Q)}\{sel(e)\} & if\ Q\ is\ a\ BGP; \\ \min\{card(Q_1), card(Q_2)\} & if\ Q = Q_1\ AND\ Q_2; \\ card(Q_1) + card(Q_2) & if\ Q = Q_1\ UNION\ Q_2; \\ card(Q_1) + \Delta_1 & if\ Q = Q_1\ OPT\ Q_2; \\ card(Q_1) + \Delta_2 & if\ Q = Q_1\ FILTER\ F; \end{cases} \quad (1)$$

where Δ_1 and Δ_2 are empirically trivial values [11].

Then, given a set of subqueries $\mathcal{Q} = \{q_1@S, q_2@S, \ldots, q_n@S\}$ over a source S using OPTIONAL and FILTER, if p is their common subgraph among $q_1, \ldots, q_n$,

we rewrite them into a SPARQL query $\hat{q}$. The cost of evaluating $\hat{q}$ is defined as follows.

$$cost_{DS}(\hat{q}) = card(\hat{q}) \times T_{MSG} = (\min_{e \in p}\{sel(e)\} + \Delta_1 + \Delta_2) \times T_{MSG}$$

Because Δ_1 and Δ_2 are trivial values, we ignore the trivial variables Δ_1 and Δ_2, and have the following cost function of data shipment.

$$cost_{DS}(\hat{q}) = \min_{e \in p}\{sel(e)\} \times T_{MSG}$$

In addition, since the OPTIONAL and FILTER operators are evaluated on the result of main pattern on popular RDF stores, the time for local evaluation is also based on the cardinality of the main pattern. Furthermore, since each OPTIONAL operator adds simply a left-join with the results of the main pattern on popular RDF stores, the time for local evaluation of multiple OPTIONAL operators is approximately equal to the time of multiple execution of main pattern. We verify the above principle by using WatDiv on three popular RDF stores, Jena, Sesame and Virtuoso. We generate some queries with multiple OPTIONAL operators by using the query rewriting algorithm proposed in Sect. 5.3. The results in Fig. 7 show that the query response time of a query with multiple OPTIONALs is proportional to the number of OPTIONALs on all RDF stores.

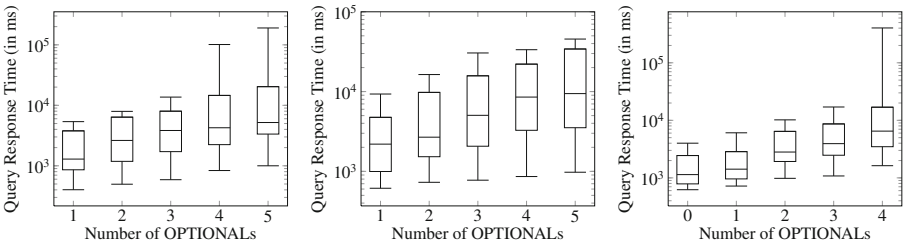


Fig. 7. Experiment results of the relationship between the local evaluation cost and the number of OPTIONALs

Thus, given a SPARQL Q, its time for local evaluation can be estimated as follows.

$$cost_{LE}(Q) = \begin{cases} \min_{e \in E(Q)}\{sel(e)\} \times T_{CPU} & if\ Q\ is\ a\ BGP; \\ \min\{cost_{LE}(Q_1), cost_{LE}(Q_2)\} & if\ Q = Q_1\ AND\ Q_2; \\ cost_{LE}(Q_1) + cost_{LE}(Q_2) & if\ Q = Q_1\ UNION\ Q_2; \\ cost_{LE}(Q_1) \times (|OPT_{Q_2}| + 1) + \Delta_3 & if\ Q = Q_1\ OPT\ Q_2; \\ cost_{LE}(Q_1) + \Delta_4 & if\ Q = Q_1\ FILTER\ F; \end{cases} \quad (2)$$

where $|OPT_{Q_2}|$ is the number of OPTIONALs in Q_2, and Δ_3 and Δ_4 are also empirically trivial values. Here, the definition of the local evaluation cost of

a query containing OPTIONAL operators is recursive, which means that the local evaluation cost of a query is proportional to the number of OPTIONAL operators.

Thus, the local evaluation cost of evaluating a rewritten query, $\hat{q}$, is as follows.

$$cost_{LE}(\hat{q}) = (|OPT_{\hat{q}}| \times \min_{e \in p}\{sel(e)\} + \Delta_3 + \Delta_4) \times T_{CPU}$$

where $|OPT_{\hat{q}}|$ is the number of OPTIONAL operators in $\hat{q}$.

Similar to the cost function of data shipment, since Δ_3 and Δ_4 are trivial values can also be ignored, the cost function of local evaluation is as follows.

$$cost_{LE}(\hat{q}) = |OPT_{\hat{q}}| \times \min_{e \in p}\{sel(e)\} \times T_{CPU}$$

In summary, given a set of subqueries $\mathcal{Q}$ over a source S, we define the cost of a specific query rewriting as follows.

Definition 7 ***(Rewriting Cost).*** *Given a set of subqueries $\mathcal{Q}$ on a source s using OPTIONAL and FILTER, if p is their common subgraph among queries in $\mathcal{Q}$, we rewrite them into a SPARQL query $\hat{q}$. The cost of the rewriting is the cost of the rewritten query $\hat{q}$ with main basic graph pattern p as shown in the following formula:*

$$cost(\mathcal{Q}, \hat{q}) = cost(\hat{q}) = cost_{LE}(\hat{q}) + cost_{DS}(\hat{q}) = \min_{e \in p}\{sel(e)\} \times (|OPT_{\hat{q}}| \times T_{CPU} + T_{MSG})$$

5.3 Query Rewriting Algorithm

The problem of query rewriting is that given a set $\mathcal{Q}$ of subqueries $\{q_1, \ldots, q_n\}$, we compute a set $\hat{\mathcal{Q}}$ of rewritten queries $\{\hat{q_1}, \ldots, \hat{q_m}\}$ $(m \leq n)$ with the smallest cost. Note that each rewritten query $\hat{q_i}$ $(i = 1, \ldots, m)$ comes from rewriting a set of original subqueries in $\mathcal{Q}$, where these subqueries share the same main pattern p_i.

Generally speaking, we find the optimal set of common patterns P, where each subquery contains at least one patterns in P. According to Sect. 5.1, if a set of subqueries can be rewritten as a rewritten query $\hat{q}$, they must share one common main pattern p. Therefore, we have the following equation.

$$cost(\mathcal{Q}, \hat{\mathcal{Q}}) = \sum_{\hat{q} \in \hat{\mathcal{Q}}} cost(\hat{q}) = \sum_{p \in P} \min_{e \in p}\{sel(e)\} \times (|OPT_{\hat{q}}| \times T_{CPU} + T_{MSG}) \quad (3)$$

where $|OPT_p|$ is the number of OPTIONAL operators that the rewritten query contains.

Given a set of original queries $\mathcal{Q}$, Eq. 3 is a set-function with respect to set P, i.e., a set of main patterns. Unfortunately, finding the optimal rewriting is a NP-complete problem as discussed in the following theorem.

Theorem 2. *Given a set of subqueries $\mathcal{Q}$, finding an optimal rewriting $\hat{\mathcal{Q}}$ to minimize the cost function in Eq. 3 is a NP-complete problem.*

Proof. We prove that by reducing the weighted set cover problem into the problem of selecting the optimal set of patterns. We map each set in S to a pattern and each element in the universe U to a subquery. A set s in S containing an element e in U maps to the pattern corresponding to s hitting the subquery corresponding to e. The weight of a set s in S is the cost of its corresponding pattern. Hence, finding the smallest weight collection of sets from S whose union covers all elements in U is equivalent to the problem of selecting the optimal set of patterns. Since the weighted set cover problem is NP-complete [4], selecting the optimal set of patterns is also NP-complete. □

Then, we propose a greedy algorithm that iteratively selects the locally optimal triple pattern in Algorithm 1. Let $\mathcal{Q}$ denote all original subqueries. At each iteration, we select a triple pattern e_{max} with the largest value $\frac{|\mathcal{Q}'|}{sel(e_{max})\times(|OPT_{\hat{q}}|\times T_{CPU}+T_{MSG})}$, where $\mathcal{Q}'$ denote all subqueries containing e_{max}. Then, we extract the largest common pattern p of queries in $\mathcal{Q}'$. It should be noted that the common pattern contains the triple pattern e_{max}, so we can find the largest common pattern by exploring from e_{max}, which is much cheaper. We divide $\mathcal{Q}'$ into several equivalence classes, where each class contains subqueries with the same structure except for some constants on subject or object positions. Subqueries in the same equivalence class can be rewritten to a query pattern with FILTER operators as discussed in Sect. 5.1, and all queries in $\mathcal{Q}'$ can be rewritten into SPARQL $\hat{q}$ with OPTIONAL operator using e_{max} as the main pattern. We remove queries in $\mathcal{Q}'$ from $\mathcal{Q}$ and iterate until $\mathcal{Q}$ is empty.

Algorithm 1. Query Rewriting Algorithm

Input: A set of subqueries $\mathcal{Q}$.
Output: A set of rewritten queries sets $\mathcal{Q}_{OPT}$.

```
while Q ≠ ∅ do
    Select the triple pattern e_max with the largest value
    |Q'| / (sel(e_max)×(|OPT_q̂|×T_CPU+T_MSG)), where Q' is the set of subqueries containing
    e_max;
    Extract the largest common pattern p of queries in Q' by exploring from
    e_max, since all queries in Q' contain e_max;
    Initialize a rewritten query q̂, where p is its main pattern;
    Divide Q' into a collection of equivalence classes C, where each class
    contains subqueries isomorphic to each other;
    for each class C ∈ C do
        Generalize a pattern p' isomorphic to all patterns in C;
        Build a query pattern with p';
        Add FILTER operators by mapping p' to patterns in C;
        Add the pattern into q̂ as an OPTIONAL pattern;
    Add q̂ into Q_OPT;
    Q = Q − Q';
Return Q_OPT;
```

For example, given subqueries $q_1^1@\{GeoNames\}$, $q_2^1@\{GeoNames\}$ and $q_3^1@\{GeoNames\}$ in Fig. 5, we select the triple pattern "?l g:name "Canada"" in the first step. It is contained by the three subqueries. We divide them into two equivalence classes $\{q_1^1\}$, $\{q_2^1, q_3^1\}$ according to the query structure. Then, we rewrite $\{q_2^1, q_3^1\}$ using FILTER operator. Finally, we rewrite the three queries using OPTIONAL operator using "?l g:name "Canada"" and obtain the result rewritten query as shown in Fig. 5.

Theorem 3. *The total cost of patterns selected by Algorithm 1 is no more than* $(1 + ln|\cup_{q\in\mathcal{Q}} E(q)|) \times cost_{opt}$*, where* $\cup_{q\in\mathcal{Q}} E(q)$ *is the set of triple patterns of all subqueries in* $\mathcal{Q}$ *and* $cost_{opt}$ *denotes the smallest cost of patterns that are contained by all subqueries.*

Proof. According to Eq. 3, selecting patterns to hit subqueries is equivalent to selecting triple patterns to hit subqueries. Thus, although we only select the most beneficial triple pattern in Algorithm 1 (Line 2), it is equivalent to selecting the most beneficial pattern graph to hit subqueries. A result in [4] shows that the approximation ratio of the greedy algorithm for the weighted set-cover problem is $(1 + ln|\cup_{q\in\mathcal{Q}} E(q)|)$. □

6 Local Evaluation and Postprocessing

A set of subqueries $\mathcal{Q}$ that will be sent to source s are rewritten as queries $\hat{\mathcal{Q}}$ and evaluated at source s. Let $[\![\hat{q}]\!]_{\{s\}}$ denote the result set of $\hat{q}$ $(\in \hat{\mathcal{Q}})$ at source s, and $\hat{q}$ is obtained by rewriting a set of original subqueries in $\mathcal{Q}$. Thus, $[\![\hat{q}]\!]_{\{s\}}$ is always the union of the results of the subqueries that are rewritten, and we track the mappings between the variables in the rewritten query and the variables in the original subqueries. The result of a rewritten query might have empty (null) columns corresponding to the variables from the OPTIONAL operators. Therefore, a result in $[\![\hat{q}]\!]_{\{s\}}$ may not conform the description of every subquery in $\mathcal{Q}$.

We should identify the valid overlap between each result in $[\![\hat{q}]\!]_{\{s\}}$ and each subquery in $\mathcal{Q}$, and check whether a result in $[\![\hat{q}]\!]_{\{s\}}$ belongs to the relevant sources of a subquery. We return to each query the result it is supposed to get. Specifically, we perform an intersection between each result in $[\![\hat{q}]\!]_{\{s\}}$ and each subquery. The algorithm distributes the corresponding part of this result to $q@\{s\}$ as one of its query results, if the result meet the following two conditions: (1) the columns of this result corresponding to those columns of a subquery $q@s \in \mathcal{Q}$ are not null; and (2) the columns of the result meet the constraints in the FILTER operators rewritten from q. This step iterates over each row and each subquery in $\mathcal{Q}$. The checking on $[\![\hat{q}]\!]_{\{s\}}$ only requires a linear scan on $[\![\hat{q}]\!]_{\{s\}}$, and can be done on-the-fly as the results of $\hat{q}@\{s\}$ are streamed out from the evaluation.

7 Joining Partial Matches

After obtaining the matches of subqueries, we need to join them together to form complete results. Assume that an original query Q_i $(i = 1, \ldots, n)$ is decomposed into a set of subqueries $\{q_i^1@S(q_i^1), \ldots, q_i^{m_i}@S(q_i^{m_i})\}$. We need to obtain query result $[\![Q_i]\!]$ by joining $[\![q_i^1]\!]_{S(q_i^1)}, \ldots, [\![q_i^{m_i}]\!]_{S(q_i^{m_i})}$ together. In the following, we abbreviate $[\![q_i^{m_i}]\!]_{S(q_i^{m_i})}$ to $[\![q_i^{m_i}]\!]$.

The straightforward method is to join subquery matches for each original SPARQL query independently. However, considering multiple queries, there may exist some common computation in joining partial matches. Taking advantage of these common join structures, we can speed up the query response time for multiple queries.

Formally, given subqueries $q_i^{i_1}$ and $q_i^{i_2}$ for query Q_i and subqueries $q_j^{j_1}$ and $q_j^{j_2}$ for query Q_j, if $q_i^{i_1}$ has the same structure to $q_i^{i_2}$, $q_j^{j_1}$ has the same structure to $q_j^{j_2}$ and the join variables between $q_i^{i_1}$ and $q_i^{i_2}$ are the same to the join variables between $q_j^{j_1}$ and $q_j^{j_2}$, then we can merge $[\![q_i^{i_1}]\!] \bowtie [\![q_i^{i_2}]\!]$ and $[\![q_j^{j_1}]\!] \bowtie [\![q_j^{j_2}]\!]$ into $([\![q_i^{i_1}]\!] \cup [\![q_j^{j_1}]\!]) \bowtie ([\![q_i^{i_2}]\!] \cup [\![q_j^{j_2}]\!])$.

The use of the above optimization technique is beneficial if the cost to merge the same two joins is less than the cost of executing two joins separately. To illustrate the potential benefit of the above optimization technique, let us compare the costs of the two alternatives: $[\![q_i^{i_1}]\!] \bowtie [\![q_i^{i_2}]\!]$ and $[\![q_j^{j_1}]\!] \bowtie [\![q_j^{j_2}]\!]$ versus $([\![q_i^{i_1}]\!] \cup [\![q_j^{j_1}]\!]) \bowtie ([\![q_i^{i_2}]\!] \cup [\![q_j^{j_2}]\!])$.

The cost of executing $[\![q_i^{i_1}]\!] \bowtie [\![q_i^{i_2}]\!]$ and $[\![q_j^{j_1}]\!] \bowtie [\![q_j^{j_2}]\!]$ separately is the sum of the costs of two joins, $min\{card([\![q_i^{i_1}]\!]), card([\![q_i^{i_2}]\!])\} + min\{card([\![q_j^{j_1}]\!]), card([\![q_j^{j_2}]\!])\}$. On the other hand, the cost of executing $([\![q_i^{i_1}]\!] \cup [\![q_j^{j_1}]\!]) \bowtie ([\![q_i^{i_2}]\!] \cup [\![q_j^{j_2}]\!])$ is $min\{card([\![q_i^{i_1}]\!] \cup [\![q_j^{j_1}]\!]), card([\![q_i^{i_2}]\!] \cup [\![q_j^{j_2}]\!])\}$. Then, our optimization technique is better if it acts as a sufficient reducer, that is, if $[\![q_i^{i_1}]\!]$ and $[\![q_j^{j_1}]\!]$ overlap a lot and $[\![q_i^{i_2}]\!]$ and $[\![q_j^{j_2}]\!]$ overlap a lot. Otherwise, we do two joins separately. It is important to note that neither approach is systematically the best; they should be considered as complementary.

We can find common join substructures by using frequent subgraph mining technique [20]. Specifically, we can first find a common substructure among the join graphs of all queries, where vertices (i.e., the subqueries) in the common substructure have the largest benefit. We perform the join for this common substructure; and iterate the above process. Obviously, we can do this part only once to avoid duplicate computation.

8 Experimental Evaluation

In this section, we evaluate our federated multiple query optimization method (FMQO) over both real (FedBench) and synthetic datasets (WatDiv). We compare our system with two existing federated SPARQL query engines: FedX [18] and SPLENDID [5].

8.1 Setting

WatDiv. WatDiv [2] is a benchmark that enables diversified stress testing of RDF data management systems. We generate a WatDiv dataset of 10 million triples. WatDiv provides its own workload generator, so we directly use WatDiv's own workload generator of WatDiv to generate different workloads for testing.

FedBench. FedBench [17] is a comprehensive benchmark suite for federated RDF systems. It includes 6 real cross-domain RDF datasets and 4 real life science domain RDF datasets with 7 federated queries for each RDF dataset. To enable multiple query evaluation, we use these 14 queries as seeds and generate the workload of queries isomorphic to the benchmark queries in our experiments.

We conduct all experiments on a cluster of machines running Linux, each of which has one CPU with four cores of 3.06 GHz, 16 GB memory and 150 GB disk storage. The prototype is implemented in Java. At each site, we install Sesame 2.7 to build up an RDF source. Each source can only communicate with the control site through HTTP requests and cannot communicate with each other. For FedBench, we assume that each dataset is resident at a source site. For WatDiv, we first use METIS [8] to divide the schema graph of the collection into 4 connected subgraphs. Then, we place all instances of the same type in a source that subgraphs of the schema graph.

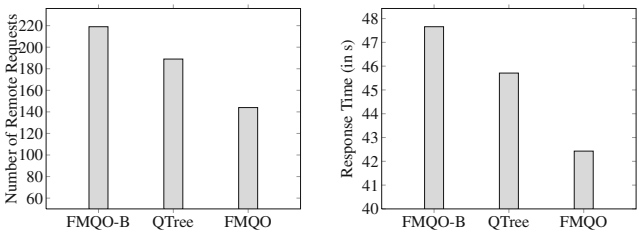
8.2 Evaluation of Proposed Techniques

In this section, we use WatDiv and a query workload of 150 queries of 10 templates to evaluate each proposed technique in this paper. In other words, 150 queries are posed simultaneously to the federated RDF systems storing WatDiv.

Effect of the Query Decomposition and Source Selection Technique. First, we evaluate the effectiveness of our source topology-based technique proposed in Sect. 4. In Fig. 8, we compare our technique with the baseline without any optimizations during source selection (denoted as FMQO-B). We also compare the source selection method proposed in [6,13], which is denoted as QTree and uses the neighborhood information in the source topology to prune some irrelevant sources for each triple patterns.

As shown in Fig. 8, FMQO-Basic does not prune any sources, so it leads to the most number of remote requests and the largest query response time. QTree only uses the neighborhood information and does not consider the whole topology of relevant sources, so the effectiveness of its pruning rule is limited. In contrast, many queries contain triple patterns containing constants with high selectivity, so these triple patterns can be localized to a few sources. Then, for other triple patterns, if the relevant sources are far from relevant sources of the selective triple patterns in the source topology graph, they can be filtered out by our method. Thus, our method leads to the smallest numbers of remote requests and the least query response time.

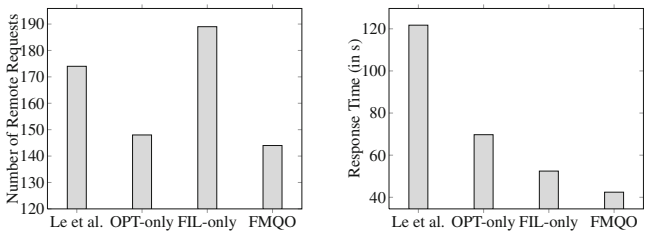
Effect of the Rewriting Strategies. In this experiment, we compare our SPARQL query rewriting techniques using only OPTIONAL operators (denoted



(a) Number of Remote Requests (b) Response Time

Fig. 8. Evaluating source topology-based source selection technique

as OPT-only) and only FILTER operators (denoted as FIL-only). We also re-implement the rewriting strategies proposed in [11] (denoted as Le et al.) to rewrite subqueries. Our query rewiring technique is denoted as FMQO. Figure 9 shows the experimental results.



(a) Number of Remote Requests (b) Response Time

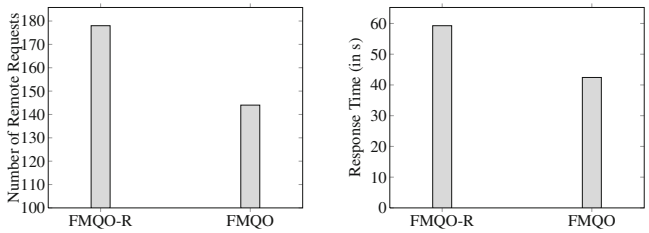
Fig. 9. Evaluating different rewriting strategies

Given a workload, since the number of subqueries sharing common substructures is more than the number of subqueries having the same structure, FIL-only leads to the largest number of rewritten queries, which indicates most remote requests. Le et al. first cluster all subqueries into some groups, and then find the maximal common edge subgraphs of the group of subqueries for query rewriting. In contrast, OPT-only and FMQO use some triple patterns to hit subqueries. In real applications, most maximal common edge subgraphs found by Le et al. also contain our selected triple patterns. Hence, Le et al. generate more rewritten queries which means more remote requests. Finally, FMQO obtains the smallest number of rewritten queries.

Since OPT-only generates smaller number of rewritten queries and share more computation than Le et al., OPT-only can result in faster query response time. If two queries have the same main pattern, the OPTIONAL operator is slower than the FILTER operator, since the former is based on left-join and the latter is based on selection. Hence, although FIL-only generate more rewritten queries, it takes about half time of Le et al., as shown in Fig. 13(b) and two thirds

of OPT-only. Furthermore, our rewriting technique using both OPTIONAL and FILTER operators has the best performance, because it takes advantages of both OPTIONAL and FILTER operators.

Evaluation of the Cost Model. In this section, we evaluate the effectiveness of our cost model and cost-aware rewritten strategy in Sect. 5.2. We design a baseline (FMQO-R) that randomly select triple patterns to rewrite subqueries. As shown in Fig. 10(a), because the patterns with lower cost are shared by more subqueries and can result in fewer rewritten queries, our cost-based selection causes fewer remote requests. In addition, in our cost-based rewriting strategy, we prefer selective query patterns, resulting in lower query response times, as shown in Fig. 10(b). Generally speaking, the cost model-based approach can provide speed up of twice.



(a) Number of Remote Requests (b) Response Time

Fig. 10. Evaluating cost model

Effect of Optimization Techniques for Joins. We evaluate our optimized join strategy proposed in Sect. 7. We design a baseline that runs multiple federated queries with only rewriting strategies but not our optimization techniques for joins (denoted as FMQO-QR). Although this technique does not affect the number of remote requests, it reduces the join cost by making use of common join structures. In general, it reduces join processing time by 10%, as shown in Fig. 11.

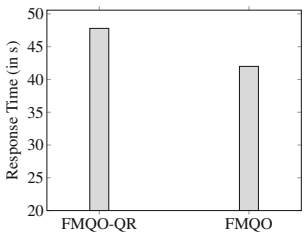
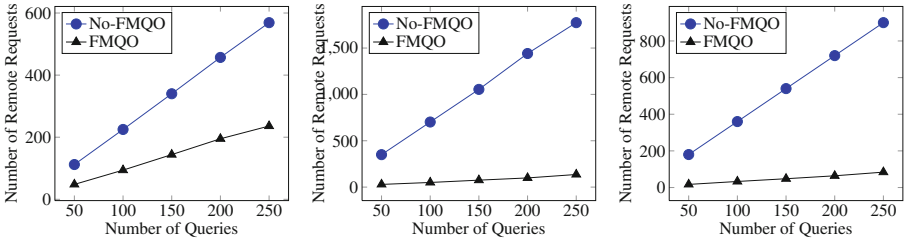


Fig. 11. Effect of optimization techniques for joins

8.3 Performance Comparison

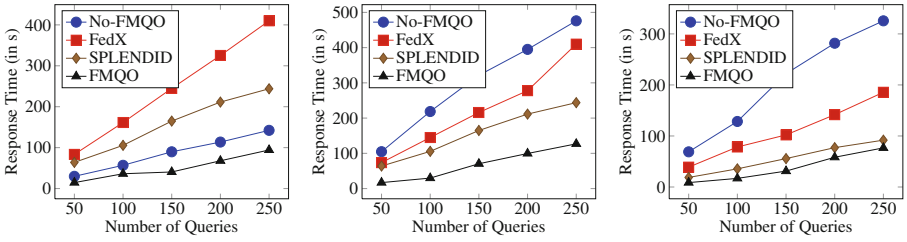
In this experiment, we using both WatDiv and FedBench to test the performance of our method in Figs. 12 and 13. We design a baseline that runs multiple federated queries sequentially (denoted as No-FMQO). This baseline does not employ any optimizations proposed in this paper. We also compare our method with FedX and SPLENDID.

Due to query rewriting, FMQO can merge many subqueries into fewer rewritten queries, which results in smaller number of remote requests, as shown in Fig. 12. FMQO can reduce the number of remote accesses by 1/2–2/3, compared with No-FMQO. Since FedX and SPLENDID do not provide their numbers of remote requests, we do not compare FMQO with FedX and SPLENDID. In terms of evaluation times, the cost-driven rewriting strategy in our method guarantees that rewritten queries are always faster than evaluating them sequentially, as shown in Fig. 13. Note that, FedX and SPLENDID always employ the semijoin to join partial matches. In WatDiv, since almost all partial matches participate in the join, the semijoin is not always efficient.



(a) WatDiv (b) FedBench (Cross Domain) (c) FedBench (Life Science)

Fig. 12. Number of remote requests



(a) WatDiv (b) FedBench (Cross Domain) (c) FedBench (Life Science)

Fig. 13. Response time

9 Related Work

There are two threads of related work: SPARQL query processing in federated RDF systems and multiple SAPRQL queries optimization.

Federated Query Processing. Many approaches [1,5,6,13–15,18] have been proposed for federated SPARQL query processing. Since RDF sources in federated RDF systems are autonomous, the major differences among existing approaches are the query decomposition and source selection approaches.

First, most papers find the relevant RDF sources for all triple patterns based on the metadata. In particular, the metadata in DARQ [14] is named service descriptions, which describes the data available from a data source in form of capabilities. SPLENDID [5] uses Vocabulary of Interlinked Datasets (VOID) as the metadata. QTree [6,13] is a variant of RTree, where its leaf stores a set of source identifiers, including one for each source of a triple approximated by the node. HiBISCuS [15] relies on capabilities to compute the metadata. For each source, HiBISCuS defines a set of capabilities which map the properties to their subject and object authorities. ANAPSID [1] dynamically generates the query plan while considering data availability and run-time conditions. Moreover, ANAPSID implement a memory-based caching mechanism.

Besides the metadata-assisted methods, FedX [18] performs the source selection by using ASK queries. FedX sends ASK queries for each triple pattern to the RDF sources. Based on the results, It annotates each pattern in the query with its relevant sources.

Multiple SPARQL Queries Optimization. Le et al. [11] first discuss how to optimize multiple SPARQL queries evaluation, but only in a centralized environment. It first finds out all maximal common edge subgraphs (MCES) among a group of query graphs, and then rewrite the queries into queries with OPTIONAL operators. In the rewritten queries, the MCES constitutes the main pattern, while the remaining subquery of each individual query generates an OPTIONAL clause. Konstantinidis et al. [9] discuss how to optimize multiple SPARQL queries evaluation over multiple views. They find out some atomic join operations among multiple queries and compute them just once.

10 Conclusion

In this paper, we study multiple query optimization over federated RDF systems. Our optimization identifies common subqueries with a cost model and rewrites queries into equivalent queries. We also discuss how to efficiently select relevant sources and join intermediate results. Experiments show that our optimizations are effective.

Acknowledgement. This work was supported by The National Key Research and Development Program of China under grant 2016YFB1000603, NSFC under grant 61702171, 61622201 and 61532010, and the Fundamental Research Funds for the Central Universities. Özsu's work was supported in part by Natural Sciences and Research Council (NSERC) of Canada.

References

1. Acosta, M., Vidal, M.-E., Lampo, T., Castillo, J., Ruckhaus, E.: ANAPSID: an adaptive query processing engine for SPARQL endpoints. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011. LNCS, vol. 7031, pp. 18–34. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25073-6_2
2. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C., Vrandečić, D., Groth, P., Noy, N., Janowicz, K., Goble, C. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 197–212. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_13
3. Berners-Lee, T.: Linked Data - Design Issues. W3C (2010)
4. Chvatal, V.: A greedy heuristic for the set-covering problem. Math. Oper. Res. **4**(3), 233–235 (1979)
5. Görlitz, O., Staab, S.: SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In: COLD (2011)
6. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K., Umbrich, J.: Data summaries for on-demand queries over linked data. In: WWW, pp. 411–420 (2010)
7. Hose, K., Schenkel, R., Theobald, M., Weikum, G.: Database foundations for scalable RDF processing. In: Polleres, A., d'Amato, C., Arenas, M., Handschuh, S., Kroner, P., Ossowski, S., Patel-Schneider, P. (eds.) Reasoning Web 2011. LNCS, vol. 6848, pp. 202–249. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23032-5_4
8. Karypis, G., Kumar, V.: Multilevel graph partitioning schemes. In: ICPP, pp. 113–122 (1995)
9. Konstantinidis, G., Ambite, J.L.: Optimizing query rewriting for multiple queries. In: IIWeb, pp. 7:1–7:6 (2012)
10. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. **32**(4), 422–469 (2000)
11. Le, W., Kementsietsidis, A., Duan, S., Li, F.: Scalable multi-query optimization for SPARQL. In: ICDE, pp. 666–677 (2012)
12. Li, J., Deshpande, A., Khuller, S.: Minimizing communication cost in distributed multi-query processing. In: ICDE, pp. 772–783 (2009)
13. Prasser, F., Kemper, A., Kuhn, K.A.: Efficient distributed query processing for autonomous RDF databases. In: EDBT, pp. 372–383 (2012)
14. Quilitz, B., Leser, U.: Querying distributed RDF data sources with SPARQL. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 524–538. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68234-9_39
15. Saleem, M., Ngonga Ngomo, A.-C.: HiBISCuS: hypergraph-based source selection for SPARQL endpoint federation. In: Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8465, pp. 176–191. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07443-6_13
16. Schmachtenberg, M., Bizer, C., Paulheim, H.: Adoption of the linked data best practices in different topical domains. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C., Vrandečić, D., Groth, P., Noy, N., Janowicz, K., Goble, C. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 245–260. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_16

17. Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., Tran, T.: FedBench: a benchmark suite for federated semantic data query processing. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011. LNCS, vol. 7031, pp. 585–600. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25073-6_37
18. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: optimization techniques for federated query processing on linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011. LNCS, vol. 7031, pp. 601–616. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25073-6_38
19. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL basic graph pattern optimization using selectivity estimation. In: WWW, pp. 595–604 (2008)
20. Yan, X., Han, J.: gSpan: graph-based substructure pattern mining. In: ICDM, pp. 721–724 (2002)