Optimizing Multi-Query Evaluation in Federated RDF Systems

Peng Peng, Qi Ge, Lei Zou, M. Tamer Özsu, Zhiwei Xu, and Dongyan Zhao

Abstract—This paper revisits the classical problem of multiple query optimization in federated RDF systems. We propose a heuristic query rewriting-based approach to optimize the evaluation of multiple queries. This approach can take advantage of SPARQL 1.1 to share the common computation of multiple queries while considering the cost of both query evaluation and data shipment. Although we prove that finding the optimal rewriting for multiple queries is NP-complete, we propose a heuristic rewriting algorithm with a bounded approximation ratio. Furthermore, we propose an efficient method to use the interconnection topology between RDF sources to filter out irrelevant sources, and utilize some characteristics of SPARQL 1.1 to optimize multiple joins of intermediate matches. The extensive experimental studies show that the proposed techniques are effective, efficient and scalable.

Index Terms—Federated RDF Systems, SPARQL Query Processing, Multiple Query Optimization.

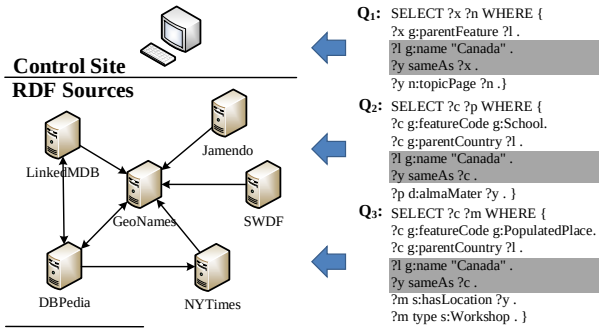
1 Introduction

RDF is a data model that is proposed by the W3C to model information on the Web, and represents data as triples of the form ⟨subject, property, object⟩. An RDF dataset can be represented as a graph, where subjects and objects are vertices and each triple is a labelled edge. To manipulate RDF data, W3C has also designed a standard query language, named SPARQL. The latest version of SPARQL is SPARQL 1.1, which extends SPARQL 1.0 by introducing some new operators. Many popular RDF stores, like Jena[1], Sesame[2] and Virtuoso[3], support SPARQL 1.1.

Increasingly, data providers have been publishing their datasets using the RDF model. These datasets are often stored at the producers' own 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 multiple RDF sources, federated RDF systems have been proposed [11], [30], [32], [36], where a control site is introduced to provide a common interface for users to issue SPARQL queries. Especially in the filed of life science, some efforts have been made to integrate different RDF sources into a federated system. For example, the European Bioinformatics Institute has built up a uniform platform[4] supporting federated queries over multiple bioinformatics SPARQL endpoints, including BioModels, Biosamples, ChEMBL, Ensembl, ExpressionAtlas, Reactome and Uniprot. To support federated queries over the data of The Cancer Genome Atlas (TCGA) project for the cancer related molecular analysis, the researchers from University of Leipzig and NUI Galway have built a federated RDF system, named TopFed [33]. Fig. 1 shows an example federated RDF system involving six RDF sources, such as NYTimes, GeoNames, SWDF and DBPedia.

Existing federated RDF systems only evaluate single queries and miss the opportunities for multiple query optimization. Real SPARQL query workloads reveal that many SPARQL queries are often posed simultaneously [7]. This provides significant multi-query optimization opportunities. 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. 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. It is easy to identify some common subgraphs over these three queries. All the three queries will be used as running examples throughout this paper.



The shaded triple patterns correspond to the common subgraph.

Fig. 1. Multiple Federated SPARQL Queries

1.1 Challenges & Our Solution

Although multi-query optimization has been well studied in distributed relational databases [21], some techniques commonly referred to as data movement and data/query shipping [19] are not easily applicable to federated RDF systems. For example, we cannot require one source to send intermediate results directly to another source. Moreover, moving data from one source to another for join processing is also infeasible [19].

- *Peng Peng, Qi Ge and Zhiwei Xu are with Hunan University, China and their email addresses are hnu16pp@hnu.edu.cn, kathy_gq@hnu.edu.cn and zhiweixu@hnu.edu.cn.*
- *Lei Zou and Dongyan Zhao are with Peking University, China and their email addresses are zoulei@pku.edu.cn and zhaodongyan@pku.edu.cn.*
- *M. Tamer Özsu is with University of Waterloo, Canada and his email address is tamer.ozsu@uwaterloo.ca.*

The corresponding author is Lei Zou.

1. http://jena.apache.org/
2. http://rdf4j.org/
3. https://virtuoso.openlinksw.com/rdf-quad-store/
4. https://www.ebi.ac.uk/rdf/services/sparql

The only multiple SPARQL query optimization work [20] only considers the centralized environment. It [20] finds out all maximal common edge subgraphs (MCES) among a group of query graphs and rewrite each group of queries into one query by only using OPTIONAL operators. Adapting this approach to federated RDF systems is difficult, because rewriting multiple queries as proposed may generate many intermediate matches and result in high data shipment cost. Furthermore, the approach does not consider some new operators introduced by SPARQL 1.1 like VALUES.

Our method has two novel characteristics:

1) We consider some characteristics of SPARQL 1.1 including operators such as "OPTIONAL", "UNION" and "VALUES". This allows us great rewrite opportunities.
2) We propose a cost model-driven greedy solution for multi-query rewriting. Our cost model considers the cost of both query evaluation and data shipment, while our solution considers both the common subgraphs and the selectivity of queries. Also, the linear time complexity of our greedy algorithm guarantees the scalability of our rewriting strategy.

In addition, we also study relevant source selection and intermediate match joining in federated RDF systems, which obviously do not arise in the centralized counterpart. First, we propose a topology structure-based source selection, while existing approaches only consider the properties in each source. Experiments confirm that our approach can reduce remote requests by 20% compared with the existing data localization techniques. Second, we discuss how to to optimize multiple joins of intermediate matches and propose a rewriting-based optimization for joins by using some new operators in SPARQL 1.1. Experiments show that the optimized join approach can improve query performance by 40% compared with the naive join techniques.

To the best of our knowledge, this is the first study of multiple SPARQL query optimization over federated RDF systems while considering the characteristic of SPARQL 1.1, with the objective to reduce the query response time and the number of remote requests.

2 Related Work

There are two threads of related work: SPARQL query processing in federated RDF systems and multi-query optimization.

Federated SPARQL Query Processing. Many methods [11], [13], [23], [29], [30], [32], [36] have been proposed for federated SPARQL query processing. The main differences among the methods are their query decomposition and source selection strategies.

For data localization, the metadata-assisted methods are popular. These find the relevant RDF sources for a query based on the metadata. The metadata in DARQ [30] (called service descriptions) describes the data available from a data source in the form of capabilities. SPLENDID [11] uses Vocabulary of Interlinked Datasets (VOID) as the metadata. QTree [13], [29] is another kind of metadata. It is a variant of RTree, and its leaf stores a set of source identifers, including one for each source of a triple approximated by the node. HiBISCuS [32] 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. TopFed [33] is a biological federated SPARQL query engine. Its metadata comprises of an N3 file and a Tissue Source Site to Tumour hash table, which is based on the data distribution. UPSP [25] extends HiBISCuS by adding the indices for a special kind of properties that can only be found in one RDF source and is not involved in subject-subject, subject-object, object-subject or object-object joins. Odyssey [23] maintains metadata that allows for a more accurate cost estimation to produce better query execution plans.

There are few works that do not require the metadata. FedX [36] selects sources by using ASK queries. It sends ASK queries for triple patterns to the RDF sources. Based on the results, it annotates each triple pattern in the query with its relevant sources.

Furthermore, in SPARQL 1.1, a new keyword, SERVICE, is introduced, which can explicitly instruct a federated query processor to invoke a portion of a SPARQL query against a remote SPARQL endpoint. Some papers [4], [5], [6] discuss the syntax of this SPARQL 1.1 federation extension and formalize its semantics. Buil-Aranda et al. [3], [5] also discuss how to optimize the evaluation for the queries in the presence of both SERVICE and OPTIONAL operators. The keyword SERVICE needs users to explicitly specify the remote SPARQL endpoints, while the federate RDF system can support querying without specifying the remote SPARQL endpoints.

In addition, there are many distributed RDF systems [9], [12], [24], [34] based on query decomposition. In [24], the authors first decompose the input query into star subqueries and compute their matches. Then, the matches are joined by using the MapReduce framework. CliqueSquare [9] decompose the input query into multiple star subqueries and builds an optimized flat query plan in Hadoop. S2RDF [34] partitions the RDF data vertically and materializes some extra join results. The partitioning results and extra join results are stored using SPARK SQL. For query processing, S2RDF translates the query into SQL queries. DREAM [12] maintains the whole RDF dataset at each site. It divides the input query into subqueries and executes each subquery at one site. The intermediate results are merged to produce the final matches.

Multiple SPARQL Query Optimization. Le et al. [20] were the first to discuss how to optimize multiple SPARQL queries, but only in a centralized environment. It first finds out all maximal common edge subgraphs (MCES) among a group of query graphs, and then rewrites these queries into a query 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. [18] discuss how to optimize multiple SPARQL queries over multiple views. They first find out some atomic join operations among multiple queries. Then, they map each atomic join operation to a view and compute it just once to avoid redundant work. Peng et al. [27] propose a multi-query optimization over federated RDF systems which only uses "OPTIONAL" and "FILTER" but does not consider "UNION" and "VALUES".

There are a few papers on multi-query processing and optimization on Hadoop. HadoopSPARQL [22] discusses how to translate a set of join operators into one Hadoop job. Anyanwu et al. [2] extend the "multi-starjoin" processing in multiple SPARQL queries to "multi-OPTIONAL" processing, which reduces the number of MapReduce cycles.

3 Background

We build on the definition of the RDF graph, SPARQL query and federated RDF systems given in Section 1.

In RDF, resources in the Web are identified by Uniform Resource Identifiers (URIs). In the context of our federated RDF

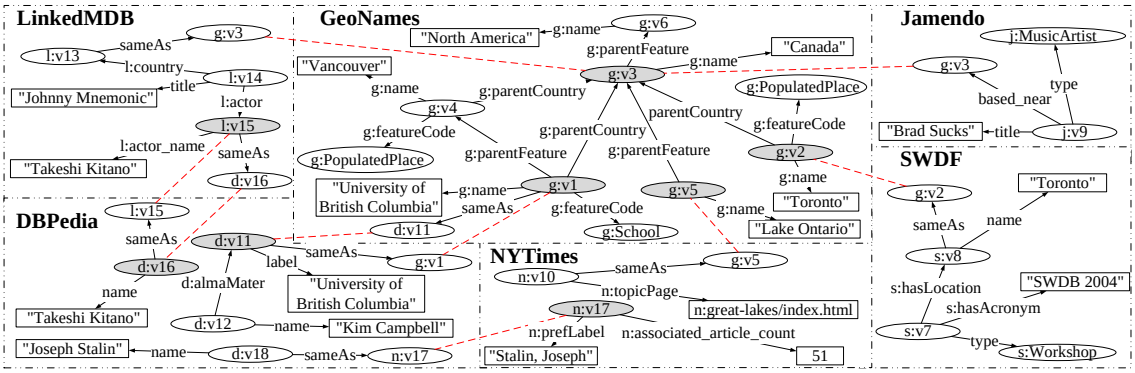


Fig. 2. Example Federated RDF System

system, we extend the idea of authoritative source [16]. A key characteristic of our federated RDF system is the correspondence between an URI and a source; i.e., the name for a resource is associated with the host source where the information of the resource is contained.

Definition 1. **(Resource in RDF, Lookup)**. Let $\mathcal{U}$ be the set of all resources in the Web. The function $d : \mathcal{U} \rightarrow S$ is a partial, surjective mapping that models a lookup and returns the source containing the information of URI.

Based on the definition of URI, we can define the RDF graph as follows. We stay close to similar definitions as found in [4], [14], [15], [28].

Definition 2. **(RDF Terms, Triple and Graph)**. The set of RDF terms consists of the set of URIs $\mathcal{U}$, the set of blank nodes $\mathcal{B}$ and the set of literals $\mathcal{L}$. A triple $(s, p, o) \in \mathcal{T} = (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ is called an RDF triple, where s is the subject, p is the property and o is the object. A set $\mathcal{T}$ of triples is called an RDF graph.

In the context of federated RDF systems, G is a combination of many RDF graphs located at different source sites.

Definition 3. **(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) $g : S \rightarrow 2^{\mathcal{T}}$ is a mapping that associates each source with a graph of RDF graph $\mathcal{T}$; and (3) $d : \mathcal{U} \rightarrow S$ is a partial, surjective mapping that models the fact that looking up URI of resource u matches 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.
Then, an RDF graph $\mathcal{T}$ is the union of all subgraphs at different sources, i.e., $\bigcup_{\tau \in S} g(\tau) = \mathcal{T}$.

An RDF graph can also be represented as a graph, where subjects and objects are vertices and each triple is a labelled edge between vertices. Fig. 2 shows a federated RDF system as a graph distributed among six different sources. Given a resource with the URI "g:$v2$", d("g:$v2$")=GeoNames, where "g" is abbreviation of "GeoNames". This means that "g:$v2$" is dereferenced by the host GeoNames. Although the URI of a resource u may be contained in multiple sources, u is only dereferenced by the host source $d(u)$; the other copies are called *mirrors*. In Fig. 2, the URIs at their host sources are denoted as the grey ovals and they connect to the corresponding mirrors at other sources by dashed lines. For the URI "g:$v2$", it is distributed among two sources, GeoNames and SWDF, and only GeoNames is its host source.

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

Definition 4. **(Basic Graph Pattern)** Let $\mathcal{V}$ be a set of variables; variables bind to RDF terms from $\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$. A triple $e \in (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V} \cup \mathcal{L})$ is called triple pattern. We omit blank nodes from triple patterns for ease of exposition. $\mathcal{P}$ is the set of all triple patterns. A basic graph pattern (BGP) is a set $Q \subset \mathcal{P}$. A set of queries is $\mathcal{Q} \subset 2^{\mathcal{P}}$.

A match distributed over a set of sources $S' \subseteq S$ is a function μ from variables in Q to RDF terms in $\bigcup_{\tau \in S'} g(\tau)$. In federated RDF systems, a match of BGP Q may span over different sources.

Definition 5. **(BGP Match over Federated RDF System)** Here, we denote the partial function $\mu : \mathcal{V} \rightarrow \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ as a *mapping* μ from $\mathcal{V}$ to $\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$. Abusing notation, for a triple pattern e, we denote by $\mu(e)$ the triple obtained by replacing the variables in e according to μ.
Consider a federated RDF system $W = (S, g, d)$ and a BGP Q. Given $S' \subseteq S$, a *mapping* μ is said to be a *match* of Q if and only if $\mu(tp) \in \bigcup_{\tau \in S'} g(\tau)$ for each triple pattern e in Q.

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

Definition 6. **(Compatibility)** Given two BGP queries Q_1 and Q_2, μ_1 and μ_2 define two matching functions $\mathcal{V}_1 \rightarrow \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ and $\mathcal{V}_2 \rightarrow \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$, respectively. μ_1 and μ_2 are *compatible* when for all $x \in \mathcal{V}_1 \cap \mathcal{V}_2$, $\mu_1(x) = \mu_2(x)$ (denoted as $\mu_1 \sim \mu_2$); otherwise, they are incompatible, denoted as $\mu_1 \nsim \mu_2$.

The semantics of a general SPARQL query is recursively defined based on BGP.

Definition 7. **(SPARQL Query)** Any BGP is a SPARQL query. If Q_1 and Q_2 are SPARQL queries, then expressions $(Q_1\ AND\ Q_2)$, $(Q_1\ UNION\ Q_2)$, $(Q_1\ OPT\ Q_2)$, $(Q_1\ FILTER\ F)$ and $(VALUES\ \vec{W}\ D)$ are also SPARQL queries, where F is a constraint, $\vec{W}$ is a list of distinct variables in $\mathcal{V}$ and D is a set of vectors of constants.

Fig. 3 shows an example SPARQL query.

```
Select ?l ?x ?c ?f where{
?l g:name "Canada" .
OPTIONAL {{?x g:parentFeature ?l . }UNION
          {?c g:featureCode ?f.
          ?c g:parentCountry ?l .
          VALUES (?f) {(g:School) (g:PopulatedPlace)} } }
```

Fig. 3. Example SPARQL Query

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

Definition 8. **(SPARQL Match over Federated RDF System)** Given a federated RDF system $W = (S, g, d)$, the match of a SPARQL query Q over a set of sources S' ($S' \subseteq S$), denoted as $[\![Q]\!]_{S'}$, is defined recursively as follows:

1) If Q is a BGP, $[\![Q]\!]_{S'}$ is defined in Definition 5;
2) If $Q = Q_1$ *AND* Q_2, then $[\![Q]\!]_{S'} = [\![Q_1]\!]_{S'} \bowtie [\![Q_2]\!]_{S'} = \{\mu_1 \cup \mu_2 \,|\, \mu_1 \in [\![Q_1]\!]_{S'} \wedge \mu_2 \in [\![Q_2]\!]_{S'} \wedge (\mu_1 \sim \mu_2)\}$;
3) If $Q = Q_1$ *UNION* Q_2, then $[\![Q]\!]_{S'} = [\![Q_1]\!]_{S'} \cup [\![Q_2]\!]_{S'} = \{\mu \,|\, \mu \in [\![Q_1]\!]_{S'} \vee \mu \in [\![Q_2]\!]_{S'}\}$;
4) If $Q = Q_1$ *OPT* Q_2, then $[\![Q]\!]_{S'} = ([\![Q_1]\!]_{S'} \bowtie [\![Q_2]\!]_{S'}) \cup ([\![Q_1]\!]_{S'} \backslash [\![Q_2]\!]_{S'}) = \{\mu_1 \cup \mu_2 \,|\, \mu_1 \in [\![Q_1]\!]_{S'} \wedge \mu_2 \in [\![Q_2]\!]_{S'} \wedge (\mu_1 \sim \mu_2)\} \cup \{\mu_1 \,|\, \mu_1 \in [\![Q_1]\!]_{S'} \wedge (\forall \mu_2 \in [\![Q_2]\!]_{S'}, \mu_1 \nsim \mu_2)\}$;
5) If $Q = Q_1$ *FILTER* F, then $[\![Q]\!]_{S'} = \Theta_F([\![Q_1]\!]_{S'}) = \{\mu_1 | \mu_1 \in [\![Q_1]\!]_{S'} \wedge \mu_1$ *satisfies* $F\}$;
6) If $Q =$ *VALUES* $\overrightarrow{W}$ D, then $[\![Q]\!]_{S'} = \{\mu | dom(\mu) = \overrightarrow{W} \wedge \mu(\overrightarrow{W}) \in D\}$.

If $S' = S$, i.e., the whole federated RDF system W, we call $[\![Q]\!]_S$ the match 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 Q_{in} and a federated RDF system $W = (S, g, d)$, our problem is to find the matches *of each query in Q_{in} over W.*

4 Framework

A federated RDF system consists of a control site as well as some RDF sources. All queries are submitted to the control site. The control site decomposes a query into several subqueries that are sent to relevant sources. If many subqueries sent to the same source share common subgraphs, the control site rewrites them as few queries as possible. Matches of subqueries are returned to the control site for joins to form complete matches.

Our framework consists of five steps: *query decomposition and source selection*, *query rewriting*, *local evaluation*, *postprocessing* and *intermediate match join* (see Fig. 4). We briefly review the five steps before we discuss them in detail in upcoming sections. Note that only *local evaluation* is conducted over the remote sources and the other four steps work at the control site.

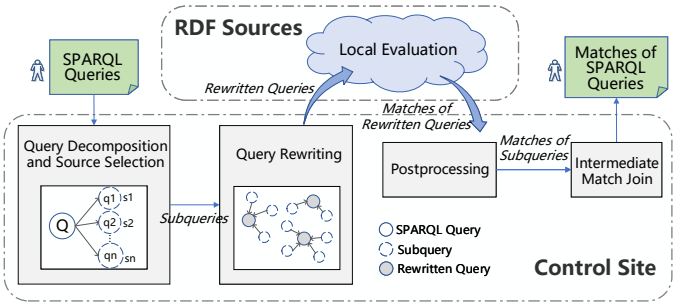


Fig. 4. Scheme for Federated Multiple SPARQL Queries Processing

Query Decomposition and Source Selection. We first decompose a query Q into a set of subqueries expressed over relevant sources. In this paper, we propose to utilize the interconnection between different RDF sources to further filter out irrelevant sources. For a SPARQL query Q_i in Q_{in}, we obtain a set Q_i of subqueries $\{q_i^1@S(q_i^1), ..., q_i^{m_i}@S(q_i^{m_i})\}$, where $q_i^j@S(q_i^j)$ is subquery of query Q_i whose relevant sources are $S(q_i^j)$. Then, Q_{in} is decomposed into $\bigcup_i Q_i$.

Query Rewriting. The set of subqueries that are planned to be sent to the same source provides an opportunity for multiple query optimization. The control site uses UNION, OPTIONAL and VALUES operators to rewrite them into fewer queries that will be sent to the relevant source. Our query rewriting technique is based on a cost model that considers the time for both local evaluation and data shipment. We will discuss this in Section 6.

Local Evaluation. The set of rewritten queries are sent to their relevant sources and evaluated there. Local evaluation matches are returned to the control site.

Postprocessing. The union of local evaluation results from evaluating rewritten queries is a superset of results obtained by evaluating the original subqueries. Therefore, the control site needs to perform some postprocessing to check each local evaluation match against each original query. In this paper, we propose a postprocessing method that only requires a linear scan of the local results in Section 7.

Intermediate Match Join. For each subquery q_i^j, collecting the matches at each relevant source in $\tau \in S(q_i^j)$, we obtain all its matches. Assume that an original query is decomposed into a set of subqueries, we can obtain matches of the original query by joining matches of the subqueries. For multiple SPARQL queries over a federated RDF system, we propose an optimized solution to avoid duplicate computation in join processing (see Section 8).

5 Query Decomposition and Source Selection

Each single-triple pattern in a SPARQL query corresponds to a subquery that maps to a set of relevant sources by using ASK queries [36]. If a triple pattern is variables-only, the triple pattern can map to all sources in the federated RDF system as its relevant source and corresponds to one subquery. Consider triple pattern "?y sameAs ?x" in Q_1 of Fig. 1. Since RDF sources "GeoNames", "NYTimes", "DBPedia", "SWDF" and "LinkedMDB" contain the property "sameAs", this triple pattern has five relevant sources. However, triple pattern "?x g:parentFeature ?l" in Q_1 only maps to source "GeoNames", because that is the only source which has property "g:parentFeature".

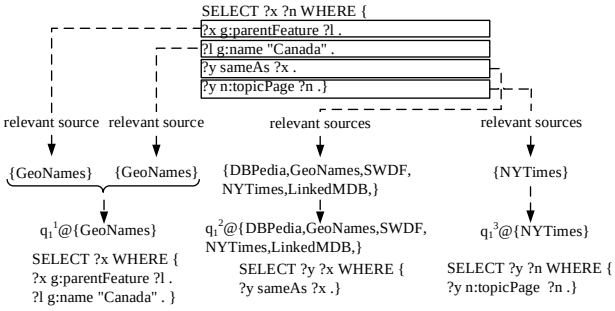


Fig. 5. Basic Query Decomposition and Source Selection Result for Q_1

To reduce the number of subqueries that are sent to each source, some single-triple pattern queries $q^1, ..., q^m$ can be combined to form a larger subquery if and only the following two conditions hold: (1) all single-triple pattern queries q^i ($i = 1, .., m$) have the same *single* relevant source; and (2) the subquery induced by these single-triple patterns forms a *connected* subquery graph. Note that, if two single-triple patterns q^i and q^j have the same *multiple* (not a single one) relevant sources, they cannot be merged. For example, q^i and q^j have the same relevant sources $\{\tau_1, \tau_2\}$. If we merge q^i with q^j together and send the merged query to both τ_1 and τ_2, we may miss the crossing match where one triple in τ_1 matches q^i and another triple in τ_2 matches q^j.

Based on the above analysis, Fig. 5 illustrates that query Q_1 is decomposed into three subqueries q_1^1, q_1^2 and q_1^3. We also show the relevant sources of each subquery.

The existing strategies (e.g., [36]) may overestimate relevant sources, leading to unnecessary joins that do not produce results. In this paper, we employ the interconnection structures among sources to further filter out irrelevant sources. Based on the crossing edges, a *source topology graph* can be defined and maintained at the control site.

Definition 9. (**Source Topology Graph**) Given a federated RDF system $W = (S, g, d)$, the corresponding *source topology graph* (STG) $T = (V(T), E(T))$ is an *undirected* graph, where (1) each vertex in $V(T)$ corresponds to a source $\tau_i \in S$; (2) there is an edge between vertices τ_i and τ_j in T if and only if there is at least one triple $(s, p, o) \in g(\tau_i)$ (or $(o, p, s) \in g(\tau_i)$ or $(s, p, o) \in g(\tau_j)$ or $(o, p, s) \in g(\tau_j)$), where s and o are URIs of two resources, $d(s) = \tau_i$ and $d(o) = \tau_j$.

Note that, unlike graph summarisation techniques like bisimulation as surveyed in [8], the source topology graph maintains the interconnection structure among RDF sources. In the source topology graph, the RDF graphs in one source is as a whole, while the graph summarisation techniques summary the structure of the whole RDF graph.

Building up the source topology graph is challenging. However, many RDF sources provide their own schemas, and the source topology graph can be built by merging the schemas of different RDF sources. Furthermore, we can utilize the crawler, LDspider proposed in [17], for a federated RDF system to figure out edges that cross between different sources, since we can look up the URIs to retrieve their host sources. We use the SPARQL endpoint to sample some URIs, and perform crawls to retrieve resources from every dataset using a breadth-first crawling strategy. Based on the crawled data, we can build up the source topology graph.

Based on this source topology graph, we propose a pruning rule to filter out irrelevant sources. Each source in STG is first annotated with its relevant decomposed subqueries. Specifically, according to the baseline solution, query Q is decomposed into several subqueries and each of them is associated with a set of relevant sources. For example, "NYtimes" is a relevant source to q_1^3, we annotate "NYTimes" in STG with q_1^3. Fig. 7(a) shows the annotated STG T^* for query Q_1. Then, each query results in an annotated source topology graph.

Meanwhile, for a BGP Q, a *linkage graph of subqueries* $LG(Q)$ is defined as follows. Fig. 7(b) shows the linkage graph of subqueries $LG(Q_1)$.

Definition 10. (**Linkage Graph of Subqueries**) Assume that a BGP query Q is decomposed into a set of subqueries $\{q^1, ..., q^m\}$. The *Linkage Graph of Subqueries* of Q is defined as a graph, where each vertex represents a subquery q^i $(i = 1, ..., n)$ of Q, an edge exists if and only if the corresponding subqueries share some common URIs or variables. The common variable must occur in "subject" position in at least one triple pattern of Q. The common URIs and variables are labels of the edge connecting their corresponding vertices.

Given a linkage graph of subqueries $LG(Q)$ and the annotated source topology graph T^*, we find all the homomorphic matches of $LG(Q)$ over T^*. Then, if a subquery q does not map to a source τ in any match, τ is not a relevant source of subquery q. We formalize this observation in Theorem 1.

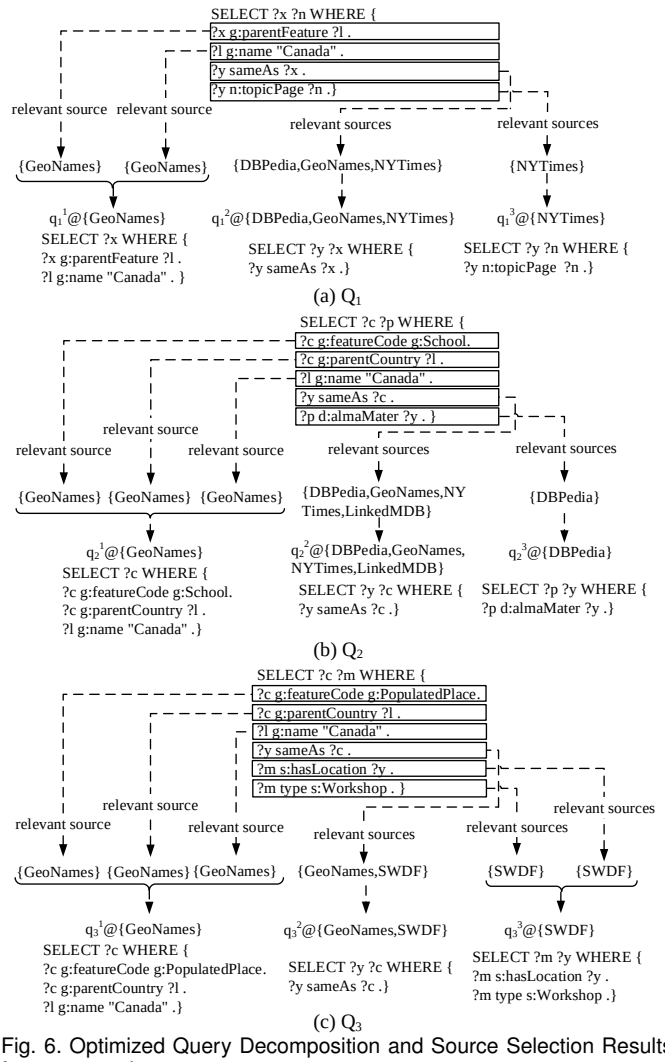
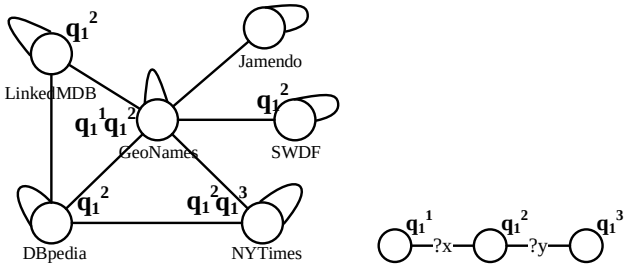


Fig. 6. Optimized Query Decomposition and Source Selection Results for Q_1, Q_1 and Q_3

Theorem 1. Given a linkage graph of subqueries $LG(Q)$ and its corresponding annotated source topology graph T^* for a subquery q, after we omit the the edges in $LG(Q)$ that their labels are literals and variables only occurring as the objects, if there exists a match m of Q^* over T^* containing q, then $m(q)$ is a relevant source of q.

Proof: The proof is given in the appendix. □



(a) Annotated Source Topology Graph for Q_1 (b) Linkage Graph of Subqueries $LG(Q_1)$ of Q_1

Fig. 7. The Linkage Graph of Subqueries and Annotated Source Topology Graph for Q_1

For example, there are three homomorphic matches of $LG(Q_1)$ over T^* in Fig. 8. Sources "LinkedMDB" and "SWDF" do not match q_1^2, so both of them can be pruned from the relevant sources of q_1^2. Fig. 6(a) shows that $q_1^2@\{DBpedia, GeoNames, NYTimes\}$, which means that q_1^2 only has three relevant sources. Existing solutions that only consider the "property" would assign the

subquery to five relevant sources. Thus, the STG-based strategy can reduce the number of relevant sources.

Match 1:	**Match 2:**	**Match 3:**
$q_1^1 \longrightarrow$ GeoNames	$q_1^1 \longrightarrow$ GeoNames	$q_1^1 \longrightarrow$ GeoNames
$q_1^2 \longrightarrow$ DBpedia	$q_1^2 \longrightarrow$ GeoNames	$q_1^2 \longrightarrow$ NYTimes
$q_1^3 \longrightarrow$ NYTimes	$q_1^3 \longrightarrow$ NYTimes	$q_1^3 \longrightarrow$ NYTimes

Fig. 8. Homomorphism Matches for $LG(Q_1)$ over the Annotated Source Topology Graph

Analogously, Q_2 and Q_3 can be decomposed into subqueries over their relevant sources. Fig. 6 shows the query decomposition and source selection results of all three example queries.

6 Query Rewriting

As mentioned in Section 1, when multiple SPARQL queries are posed simultaneously, there is room for sharing computation when executing these queries. Assume that multiple decomposed subqueries that are expected to be sent to the same source share some *common substructures*. A possible optimization is to rewrite them (at the control site) into fewer SPARQL queries based on these common substructures and then send them to relevant sources, which can save both the number of remote accesses and query response time. Obviously, different query rewritings may lead to different performances; thus, this section proposes a cost-driven query rewriting scheme. To simplify presentation, we assume that the SPARQL query originally issued at the control site is a BGP. Our solution is easily extended to handle general SPARQL queries as discussed in appendix. Note that, the following discussion focuses on the subqueries sent to the same source τ.

6.1 Intuition

We utilize "OPTIONAL", "UNION" and "VALUES" operators to utilize common subgraphs among different queries for rewriting.

6.1.1 OPTIONAL-UNION-based Rewriting

Consider two subqueries $q_1^1@\{GeoNames\}$ and $q_2^1@\{GeoNames\}$ in Fig. 6 that are decomposed from Q_1 and Q_2. They share a common subgraph: triple pattern "?l g:name "Canada" ". Therefore, a straightforward rewriting strategy in [20], [27] is to rewrite them into a single query, where the subqueries $q_1^1@\{GeoNames\}$ and $q_2^1@\{GeoNames\}$ minus "?l g:name "Canada" " map to two OPTIONAL clauses, and "?l g:name "Canada" " is the left-hand-side of the OPTIONAL.

However, simply rewriting each subquery as one OPTIONAL clause may result in some redundant matches that can match the union of multiple subqueries. For example, there may be some triples matching both $q_1^1@\{GeoNames\}$ and $q_2^1@\{GeoNames\}$. Then, the straightforward rewriting strategy produces matches of both $q_1^1@\{GeoNames\}$ and $q_2^1@\{GeoNames\}$, but these are redundant.

We can rewrite a set of subqueries with a common subgraph as a query $\hat{q}$ with an OPTIONAL clause that contains multiple UNION clauses, where the main graph pattern of $\hat{q}$ is the common subgraph among these subqueries and each subquery corresponds to one UNION clause in the OPTIONAL. For example, $q_1^1@\{GeoNames\}$ and $q_2^1@\{GeoNames\}$ are rewritten into a single query with one OPTIONAL clause, where "?l g:name "Canada" " is the main pattern, and the subgraphs $q_1^1@\{GeoNames\}$ and $q_2^1@\{GeoNames\}$ minus "?l g:name "Canada" " map to two UNION clauses in the OPTIONAL clause, as shown in Fig. 9.

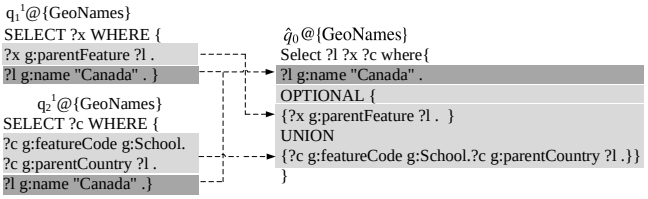


Fig. 9. Rewriting Queries using OPTIONAL and UNION Operators

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

$$\hat{q} = p\ OPT\ ((q_1 - p)\ UNION\ (q_2 - p)\ ...\ UNION\ (q_n - p))$$

Most existing RDF stores implement OPTIONAL operators using left-joins, so the cardinality of a SPARQL query with one OPTIONAL operator often does not much exceed the cardinality of the left-hand-side of the OPTIONAL [20], [27]. Thus, the cardinality of the rewritten query with one OPTIONAL operator does not increase much.

6.1.2 VALUES-based Rewriting

Consider two subqueries $q_2^1@\{GeoNames\}$ and $q_3^1@\{GeoNames\}$ in Fig. 10 decomposed from Q_2 and Q_3. Although their first triple patterns are different, the only difference is constant bounded to objects in the first triple pattern. We can rewrite the two queries using VALUES, as shown in Fig. 10. In other words, if some subqueries issued at the same source have the common subgraph except the constants on a vertex (subject or object positions), they can be rewritten as a single query with VALUES. For example, q_2^1 and q_3^1 share the same structure except for two entities "g:School" and "g:PopulatedPlace" (highlighted in Fig. 10). We introduce a new variable $?f$ and restrict the variable using VALUES operator.

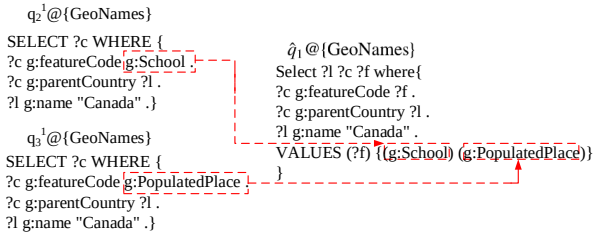


Fig. 10. Rewriting Queries using VALUES Operator

Formally, if some subqueries $\{q_1, q_2, ..., q_n\}$ that are sent to the same source τ employ the same query structure p except for some vertex constants, we rewrite them as follows.

$$\hat{q} = p\ VALUES\ (?v)\ \{(v_1)\ (v_2)\ ...\ (v_n)\}$$

where $v_1, v_2, ..., v_n$ are the different constants in the same position of p and $?v$ is the variable that replaces the constants.

The VALUES operator provides inline data as a solution sequence that are combined with the results of the main pattern. Thus, the cardinality of the rewritten query with VALUES operator is still equal to the total cardinality of the subqueries. Unlike OPTIONAL operators adding extra columns, VALUES operators do not introduce extra joins and extra intermediate matches. Hence, the cost of data shipment will increase little.

6.1.3 Hybrid Rewriting

A hybrid rewriting strategy is also feasible by using OPTIONAL, UNION and VALUES. Let us consider the three subqueries issued at the same source GeoNames. Fig. 11 illustrates a hybrid rewiring strategy, using OPTIONAL followed by UNION and VALUES.

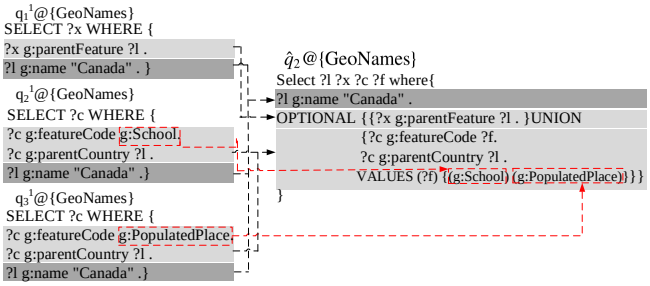


Fig. 11. Rewriting Queries using OPTIONAL, UNION and VALUES Operators

6.2 Cost Model

Given a set of subqueries, we may have different rewriting strategies with different costs. To select a better one, we need a model to quantify the cost, which includes two components: time for local evaluation and time for data shipment. The former is the CPU time of evaluating subqueries, while the latter one refers to the data (subquery results) shipment cost.

6.2.1 Cost Model for BGPs

It has been claimed that selective triple patterns in BGP have higher priorities in evaluation [20], [27]. As verified in the appendix, the cardinality of a query is positively associated with the selectivity of the most selective triple pattern, so 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.

As mentioned earlier, all matches of subqueries are sent to the control site to join for the final matches after local evaluation. Hence, the cost of data shipment for a BGP Q can be defined based on the number of matches as follows.

$$cost_{DS}(Q) = card(Q) \times T_{MSG} = min_{e \in E(Q)}\{sel(e)\} \times T_{MSG}$$

where T_{MSG} is the unit time to transmit a data unit.

The time for evaluating a query is proportional to the number of matches, so the cost of local evaluation for a BGP Q is defined as follows.

$$cost_{LE}(Q) = card(Q) \times T_{CPU} = min_{e \in E(Q)}\{sel(e)\} \times T_{CPU}$$

where T_{CPU} is the CPU unit time to construct a match.

For estimating the selectivity of a triple pattern, we can employ heuristics that can estimate the selectivity without pre-computed statistics about the RDF source [37]. The selectivity of a triple pattern is computed according to the type and number of unbound components and is characterized by the ranking rule that subjects are more selective than objects and objects more selective than properties. Furthermore, the relative coefficients, T_{CPU} and T_{MSG}, in the cost model are greatly influenced by the resources of each RDF store and the network topology of the federated RDF system. They can be estimated offline as the metadata.

6.2.2 Cost Model for General SPARQL Queries

We design the cost model to handle general SPARQL queries based on the above discussion. The design of our cost model is motivated by the way in which a SPARQL query is evaluated in popular RDF stores. As discussed in previous studies [20], [27], the graph patterns in OPTIONAL clauses are evaluated on the results of the main pattern (for the fact that the graph pattern in the OPTIONAL clause is a left-join), which suggests that a good optimization should keep the cardinality from the common subgraph as small as possible.

In addition, the VALUES operator specifies the variable with some constants. The constants in the VALUES operator can greatly increase the selectivity of the triple patterns containing the variable, and the cardinality of a query relies heavily on the selectivity of the most selective triple pattern. As a result, the cardinality of a query with VALUES often relies heavily on the number of constants in the VALUES operator.

Consequently, the cardinality of a SPARQL query Q, $card(Q)$, is defined as follows:

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

where Δ_1 and Δ_2 are empirically trivial values [20] and $|D|$ is the size of D.

When we rewrite the subqueries by using VALUES, all constants added in VALUES are extracted from the subqueries. Because VALUES provides inline data to the variables, the rewriting is nearly equivalent to merging the subqueries without extra cost. Thus, the cardinality of the rewritten query with VALUES operator is still equal to the total cardinality of the subqueries.

Thus, given a set of subqueries $\mathcal{Q} = \{q_1@\{\tau\}, q_2@\{\tau\}, ..., q_n@\{\tau\}\}$ over a source τ, if p is the common subgraph among them and we rewrite them into a query $\hat{q}$ by using UNION, OPTIONAL and VALUES, the cost of evaluating $\hat{q}$ is defined as follows:

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

Here, as discussed before, Δ_1 is empirically trivial, so $card(\hat{q})$ is mostly determined by the cardinality of p. Hence, we ignore Δ_1 and have the following cost function for data shipment.

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

Similarly, the cost function for local evaluation is also mostly determined by the cardinality of p:

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

In summary, for a rewritten query $\hat{q}$ with the main pattern p, its total cost is defined as follows.

$$\begin{aligned} cost(\hat{q}) &= cost_{LE}(\hat{q}) + cost_{DS}(\hat{q}) \\ &= \min_{e \in p}\{sel(e)\} \times (T_{CPU} + T_{MSG}) = cost(p) \end{aligned} \quad (2)$$

6.2.3 Cost Model for Rewriting

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

As mentioned in Equation 2, the cost of a rewritten query is mostly determined by the cost of its main pattern. Thus, the problem of finding out a set $\hat{\mathcal{Q}}$ of rewritten queries is equivalent to finding out a set P of patterns, where each subquery contains at least one pattern $p \in P$ and can be rewritten by using p as the main pattern. To find out the optimal set of patterns for query rewriting, we need to measure the unit benefit of selecting a pattern for rewriting a set of subqueries. The benefit is defined as follows.

Definition 11. **(Benefit of Selecting a Pattern for Rewriting Subqueries)** Given a common subgraph p over a set of

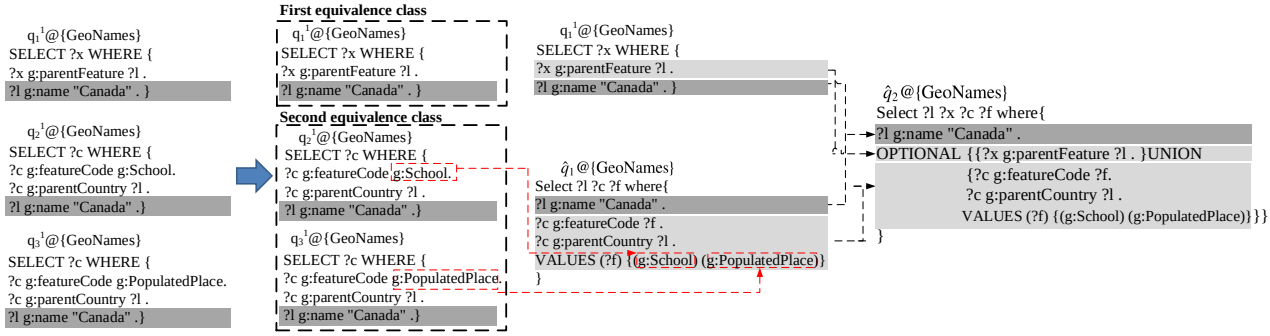


Fig. 12. Example of Rewriting Subqueries

subqueries Q, the benefit of using p to rewrite Q, $B(p, Q)$, is denoted as follows:

$$B(p,Q) = (\sum_{q\in Q} cost(q)) - cost(p)$$
$$= ((\sum_{q\in Q} \min_{e\in q}\{sel(e)\}) - \min_{e\in p}\{sel(e)\}) \times (T_{CPU} + T_{MSG})$$

In practice, there may be no common subgraph over all the subqueries in Q. Thus, we may need to decompose Q into multiple subsets $\{Q_1, ..., Q_m\}$ such that subqueries in each subset Q_i share a common subgraph p_i and they can be rewritten based on p_i. In this case, we define the benefit of rewriting the whole Q using patterns $\{p_1, ..., p_m\}$ as follows.

Definition 12. **(Benefit of Selecting a Pattern Set for Rewriting a Subquery Set)** Given a set of subqueries Q, assume that Q can be partitioned into a collection $\{Q_1, ..., Q_m\}$ of subsets of Q and p_i is the most beneficial common subgraph over Q_i. Then, the benefit of rewriting Q using $P = \{p_1, ..., p_m\}$ is defined as.

$$B(P,Q) = \sum_{i=1}^{m} B(p_i, Q_i)$$

Then, the problem of query rewriting is equivalent to finding the set of patterns P and its corresponding partitioned collection of subsets of Q that leads to the maximal benefit.

6.3 Query Rewriting Algorithm

Given a set of subqueries Q, Definition 12 is a set-function with respect to set P, a set of patterns. Unfortunately, the function is submodular. Hence, finding the optimal set P_{max} for rewriting is a NP-hard problem as discussed in the following theorem.

Theorem 2. Given a set of subqueries Q, finding the optimal set P_{max} of patterns to rewrite all subqueries in Q while maximizing the benefit function in Definition 12 is NP-hard.

Proof: The proof is given in the appendix. □

We propose a greedy algorithm that iteratively selects the locally optimal triple pattern in Algorithm 1. Generally, at each iteration, there are three steps:

1) **Step 1: Determining the set of subqueries for rewriting (Lines 3-4)**. We select a triple pattern e_{max} with the largest discrete derivative $\frac{\triangle_{Benefit}(e_{max}|P)}{sel(e_{max})\times(T_{CPU}+T_{MSG})} = \frac{\sum_{q\in Q'} cost(q)-cost(e_{max})}{sel(e_{max})}$, where P is the set of selected patterns, Q' denote the set of subqueries hit by e_{max} and all patterns of P hitting the subqueries in Q' are less selective than e_{max}. This ensures that all subqueries in Q' contains the common triple pattern e_{max}. We can start the search from the common triple pattern to check whether different subqueries have the same common substructure except for some constants on subject or object positions. Furthermore, the average number of triple patterns in the queries of most real workloads are less than 4 [7]. Because of the common triple pattern and the limited sizes of queries, we can divide Q' into several equivalence classes, where each class contains subqueries with the same structure except for some constants on subject or object positions, within a reasonable amount of time.
2) **Step 2: VALUES-based rewriting (Lines 5-8)**. Subqueries in the same equivalence class can be rewritten to a query pattern with VALUES operators.
3) **Step 3: OPTIONAL-UNION-based rewriting (Lines 9-10)**. All subqueries in Q' can be rewritten into query $\hat{q}$ with an OPTIONAL operator using p as the main pattern. All the returning variables in the subqueries of Q' and the join variables are specified in the select clauses of $\hat{q}$.

Then, we remove queries in Q' from Q and iterate the above process until Q is empty.

Algorithm 1: Query Rewriting Algorithm

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

```
1  Initialize a pattern set P with ∅;
2  while Q ≠ ∅ do
3      Select the triple pattern e_max with the largest value
       (Σ_{q∈Q'} cost(q) − cost(e_max)) / sel(e_max), where Q' is the set of subqueries
       hit by e_max and all patterns of P hitting the
       subqueries in Q' are less selective than e_max;
4      Divide Q' into a collection of equivalence classes EC,
       where each class contains subqueries isomorphic to
       each other;
5      for each class ec ∈ EC do
6          Generalize a pattern p' isomorphic all patterns in
           EC, where p' does not contain any constants;
7          Build a query pattern with p';
8          Add VALUES by mapping p' to patterns in EC;
9      for each class ec ∈ EC do
10         Add the pattern into q̂ as a UNION pattern in the
           OPTIONAL clause;
11     All the returning variables in the queries of Q' and the
       join variables are specified in the select clauses of q̂;
12     Add q̂ into Q̂;
13     Q = Q − Q', and P = P ∪ {p};
14 Return Q̂;
```

Given subqueries $q_1^1@\{GeoNames\}$, $q_2^1@\{GeoNames\}$ and $q_3^1@\{GeoNames\}$ in Fig. 12, we select the triple pattern "?l

g:name "Canada"" in the first step. It hits the three subqueries. We divide them into two equivalence classes $\{q_1^1@\{GeoNames\}\}$, $\{q_2^1@\{GeoNames\}, q_3^1@\{GeoNames\}\}$ according to the query structure. Then, we rewrite $\{q_2^1@\{GeoNames\}, q_3^1@\{GeoNames\}\}$ using VALUES operator. Finally, we rewrite the three queries using OPTIONAL and UNION operators using the "?l g:name "Canada"" as the main pattern.

Theorem 3. The total benefit of patterns selected by using Algorithm 1 is no less than $(1-\frac{1}{e})\times benefit_{opt}$, where $benefit_{opt}$ is the largest benefit of patterns that rewrite all subqueries.

Proof: The proof is given in the appendix. □

7 Local Evaluation, Postprocessing and Intermediate Match Join

A set of subqueries Q that will be sent to source τ are rewritten as queries $\hat{Q}$ and evaluated at source τ. Let $[\![\hat{q}]\!]_{\{\tau\}}$ denote the match set of $\hat{q}$ ($\in \hat{Q}$) at source τ; $\hat{q}$ is obtained by rewriting a set of original subqueries in Q; Thus, $[\![\hat{q}]\!]_{\{\tau\}}$ is the union of the matches 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 match of a rewritten query might have empty (null) bindings corresponding to the variables from the OPTIONAL operators. Thus, a match in $[\![\hat{q}]\!]_{\{\tau\}}$ may not conform the description of every subquery in Q. We need identify the valid overlap between each match in $[\![\hat{q}]\!]_{\{\tau\}}$ and each subquery in Q, and check whether a match in $[\![\hat{q}]\!]_{\{\tau\}}$ is a match of a subquery. Each subquery receives the match that it is supposed to get.

To achieve the above objective, we perform an intersection between each match in $[\![\hat{q}]\!]_{\{\tau\}}$ and each subquery. We distribute the corresponding part of this match to $q@\{\tau\}$ as one of its query matches, if the match meet two conditions: 1) the bindings of this match corresponding to those bindings of a subquery $q@\{\tau\} \in Q$ are not null; and 2) the bindings of the match meet the constraints in the VALUES operators rewritten from q. This step iterates over each row and each subquery in Q. The checking on $[\![\hat{q}]\!]_{\{\tau\}}$ only requires a linear scan on $[\![\hat{q}]\!]_{\{\tau\}}$. Therefore, it can be done on-the-fly as the matches of $\hat{q}@\{\tau\}$ are streamed out from the evaluation. We also give an example to illustrate how we distribute $[\![\hat{q}_1]\!]_{\{GeoNames\}}$ to $q_1^1@\{GeoNames\}$, $q_2^1@\{GeoNames\}$ and $q_3^1@\{GeoNames\}$, after evaluating rewritten query $\hat{q}_2$ in appendix.

After we postprocess all matches of rewritten queries and find out the matches of all subqueries, we need to join the subquery matches for each input query. The straightforward method to obtain matches of all input queries in Q is to join subquery matches for each original SPARQL query independently. For a BGP query Q, we define the join graph $JG(Q)$, where each vertex represents a subquery of Q and an edge exists if and only if the corresponding subqueries share some common constants (including URIs, blank nodes and literals) or variables in the original SPARQL query. The common constants and variables of two subqueries are labels of the edge connecting their corresponding vertices. Then, we find the optimal execution plan by directly extending the join ordering algorithm proposed in [26]. Last, the matches of the subqueries join together according to the optimal execution plan.

8 Multi-Join Optimization

For multiple subqueries, there may exist common computation in joining intermediate matches. In this section, we discuss an optimization to share some common computations of multi-join. Here, we assume that the query Q_i $(i = 1, ..., n)$ is decomposed into a set of subqueries $\{q_i^1@S(q_i^1), ..., q_i^{m_i}@S(q_i^{m_i})\}$. We need to obtain query match $[\![Q_i]\!]$ by joining $[\![q_i^1]\!]_{S(q_i^1)}$,..., $[\![q_i^{m_i}]\!]_{S(q_i^{m_i})}$. In the following, for simplicity, we abbreviate $q_i^{m_i}@\{S(q_i^{m_i})\}$ and $[\![q_i^{m_i}]\!]_{S(q_i^{m_i})}$ to $q_i^{m_i}$ and $[\![q_i^{m_i}]\!]$.

8.1 Multi-Join Graph

Considering multiple subqueries, there may exist common computation in joining intermediate matches. Especially when some subqueries have been rewritten as described in Section 6.1, we can merge multiple joins by directly joining the matches of rewritten queries. For example, q_1^1, q_2^1 and q_3^1 are rewritten to one query $\hat{q}_2$ as shown in Fig. 12, while q_1^2, q_2^2 and q_3^2 can be rewritten to $\hat{q}_3$ that only contains one triple pattern "?y sameAs ?c". As a result, we can directly join $[\![\hat{q}_2]\!]$ with $[\![\hat{q}_3]\!]$ to compute out the combination of matches of $q_1^1 \bowtie q_1^2$, $q_2^1 \bowtie q_2^2$ and $q_3^1 \bowtie q_3^2$. Then, some postprocessings can be applied in the control site to refine the matches of $[\![\hat{q}_2 \bowtie \hat{q}_3]\!]$ and find out the matches of $q_1^1 \bowtie q_1^2$, $q_2^1 \bowtie q_2^2$ and $q_3^1 \bowtie q_3^2$.

Based on the above observation, a multi-join optimization is proposed to combine multiple joins together. Here, for a set Q of queries, we extend the definition of the join graph in Section 5 to combine multiple join graphs as a *multi-join graph* $MJG(Q)$.

In a multi-join graph, one vertex indicates a subquery or a rewritten query. We introduce an edge between two vertices in the multi-join graph if and only if the corresponding subqueries or rewritten queries share some common variables or constants. The common variables or constants of two subqueries or rewritten queries are labels of the edge connecting their corresponding vertices. The multi-join graph is a multigraph that two vertices may be connected by more than one edge.

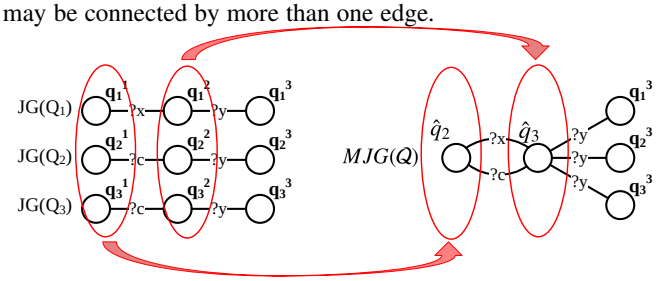


Fig. 13. Example Multi-Join Graph

Fig. 13 shows the example multi-join graph, $MJG(Q)$, combined the three join graphs of $JG(Q_1)$, $JG(Q_2)$ and $JG(Q_3)$. As discussed before, q_1^1, q_2^1 and q_3^1 are rewritten to one query $\hat{q}_2$, while q_1^2, q_2^2 and q_3^2 can be rewritten to $\hat{q}_3$. Therefore, $MJG(Q)$ replaces the vertices of q_1^1, q_2^1 and q_3^1 with one vertex $\hat{q}_2$ and q_1^2, q_2^2 and q_3^2 with $\hat{q}_3$. There are two edges between $\hat{q}_2$ and $\hat{q}_3$, because the common join variable of q_1^1 and q_1^2 is different from the common join variable of q_2^1 and q_2^2 (and q_3^1 and q_3^2).

Based on the multi-join graph, we can find out the opportunity of merging multiple joins by directly joining the matches of rewritten queries. For example, in Fig. 13, directly joining $\hat{q}_2$ with $\hat{q}_3$ is the combination of $q_1^1 \bowtie q_1^2$, $q_2^1 \bowtie q_2^2$ and $q_3^1 \bowtie q_3^2$.

8.2 Rewriting-based Join Method

Joining the matches of all subqueries or rewritten queries may be costly. Given a set of subqueries or rewritten queries, there often exist both subqueries or rewritten queries with high selectivity and subqueries or rewritten queries with low selectivity in the multi-join graph. We propose to generate the intermediate matches only for selective subqueries or rewritten queries and use them to optimize the evaluation of unselective ones.

The overall idea of our rewriting-based join method is to group a set of join variables' matches in the unselective subqueries and

rewritten queries using VALUES operators. This grouped query is then sent to the relevant sources in a single remote request. Finally, some postprocessings are applied in the control site to retain correctness. For example, as shown in Fig. 14, evaluating $\hat{q}_2$ can get the matches $[\![\hat{q}_2]\!]$. Then, $\hat{q}_3$ can be optimized by using VALUES and evaluated in its relevant sources.

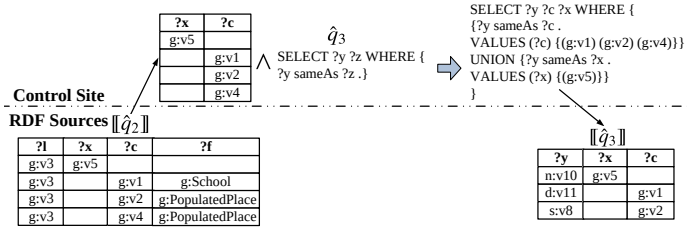


Fig. 14. VALUES-based Rewriting for Joining

Note that, as the number of intermediate matches increases, the values added to a query using VALUES also increase and the performance of evaluating the query decreases. When the performance drops to a certain level, it may be better to use the basic join method discussed in Section 7. In practice, because most RDF stores can only support the SPARQL query of the limited length, so the use of the above optimization technique is beneficial if the length of the query adding intermediate matches in one VALUES operator is not beyond the length limit.

The effect of our join strategy is similar to the effect of a semi-join [26]. However, it reduces the number of intermediate results in a different way. Semi-join still needs to generate the matches of two joining subqueries and requires an additional inner join operation following the semi-join. However, the federated RDF system cannot directly send intermediate matches from one source to another source, so sending the projected results to the other join operand is not applicable in federated RDF systems.

Two similar techniques are discussed in literature [10], [36], [38]. FedX [36] proposes a join strategy called bound join that groups a set of matches in a single query using UNION constructs, while the second technique [10], [38] proposes to send the buffered mappings as additional conditions in a FILTER expression. The VALUES-based rewriting strategy can rewrite queries in a more compact way than other rewriting strategies, and the VALUES operator can outperform the FILTER and UNION.

8.3 Multi-Join Optimization Algorithm

In this section we illustrate the use of the techniques presented in earlier subsections. We describe an optimization algorithm based on our VALUES-based rewriting strategy in Algorithm 2. Generally, there are two steps Algorithm 2 at each iteration.

1) **Step 1: Selecting the next subquery or rewritten query for evaluation (Line 3)**. We select the query q_{min} of the smallest cost in Q as the next query for evaluation.
2) **Step 2: Joining the selected query with other evaluated queries (Lines 4-17)**. If the selected query q_{min} can join with an evaluated query q^* according to $MJG(Q)$, we determine whether we use the VALUES-based join method based on the number of intermediate matches. If the use of the VALUES-based join method is not beyond the length limit, we rewrite q_{min} by using VALUES to join q^* and q_{min}; otherwise, we use the basic join method to join q^* and q_{min}. If q_{min} cannot join with any other evaluated queries, we directly send q_{min} to its relevant sources and evaluate it.

For example, let us consider the example multi-join graph as shown in Fig. 13. We assume that $\hat{q}_2$ and $\hat{q}_3$ are the first two selected rewritten queries to be evaluated and adding $\hat{q}_3$ with the bindings of common variables ($?x$ and $?c$) in $[\![\hat{q}_2]\!]$ by using VALUES is not beyond the length limit of a query. According the example multi-join graph, $\hat{q}_3$ need join with $\hat{q}_2$, so we rewrite $\hat{q}_3$ by using VALUES as shown in Fig. 14, send the new rewritten query to the relevant sources, evaluate it, and postprocess the evaluation results to get the matches of $q_1^1 \bowtie q_1^2$, $q_2^1 \bowtie q_2^2$ and $q_3^1 \bowtie q_3^2$.

Algorithm 2: Multi-Join Optimization Algorithm

Input: A set Q of subqueries and rewritten queries, and its multi-join graph $MJG(Q)$.
Output: The matches set RS of all queries in Q_{in}.

```
1  Initialize an empty set Q_eva for evaluated subqueries and rewritten queries;
2  while Q ≠ ∅ do
3      Select the subquery or rewritten query q_min ∈ Q of the smallest cost;
4      for each query q* ∈ Q_eva do
5          if q_min is connected with q* in MJG(Q) then
6              if Adding q_min with the common variables' bindings in [[q*]] by using VALUES is not beyond the length limit of a query then
7                  Rewriting q_min by using VALUES to q̂_min;
8                  Send q̂_min to its relevant sources and evaluate it;
9                  if All subqueries or rewritten queries from Q_i ∈ Q_in has been evaluated then
10                     Postprocess [[q̂_min]] to get [[Q_i]];
11                     Add [[Q_i]] in RS;
12             else
13                 Send q_min to its relevant sources and evaluate it;
14                 Join [[q_min]] with [[q*]];
15                 if All subqueries or rewritten queries from Q_i ∈ Q_in has been evaluated then
16                     Postprocess [[q_min ⋈ q*]] to get [[Q_i]];
17                     Add [[Q_i]] in RS;
18     if q_min cannot join with any queries in Q_eva then
19         Send q_min to its relevant sources and evaluate it;
20     Q = Q − {q}, and Q_eva = Q_eva ∪ {q};
21 Return RS;
```

9 Handling General SPARQL Queries

So far, we only considered BGP (basic graph patterns) over federated RDF systems. In this section, we discuss how to extend our method to general SPARQL queries with UNION, OPTIONAL, FILTER and VALUES statements. Generally, any type of query can be transformed to the operations on a set of BGPs.

Queries with UNION/OPTIONAL operators. A query with UNION operator (Q_1 *UNION* Q_2) can be directly decomposed into two BGPs Q_1 and Q_2, while a query with a OPTIONAL operator Q_1 *OPTIONAL* Q_2 can be rewritten into two BGPs Q_1 and (Q_1 *AND* Q_2). Then, we can pass the batch of BGPs for our multi-query optimization, and the matches to the original query with UNION operators can be generated through the matches from the transformed BGPs after our multi-query optimization.

Queries with FILTER/VALUES operators. For queries with FILTER or VALUES operators, during data localization, we move possible FILTER or VALUES constraints into the subqueries to

reduce the size of intermediate matches as early as possible. For example, the query with FILTER and VALUES operators in Fig. 15(a) is decomposed to two subqueries as in Fig. 15(b).

In addition, when we use VALUES-based rewriting strategy to rewrite a subquery, we merge the original VALUES operators and the rewritten VALUES operators by using the intersection operators. For example, the subquery in Fig. 15(b) can be rewritten to the query in Fig. 16.

```
SELECT ?c ?n WHERE {
?c g:featureCode g:School.
?c g:name ?n
?c g:parentCountry ?l .
?l g:name x .
?y sameAs ?c
FILTER (regex(str(?n), "Toronto", "i"))
VALUES (?x) {("Canada")}}
```

(a) Query with FILTER/-VALUES Operator

```
SELECT ?c ?n WHERE {
?c g:featureCode g:School.
?c g:name ?n
?c g:parentCountry ?l .
?l g:name ?x .
FILTER (regex(str(?n), "Toronto", "i"))
VALUES (?x) {("Canada")}}
```

```
SELECT ?x ?n WHERE {
?y sameAs ?c }
```

(b) Subqueries

Fig. 15. Query with FILTER/VALUES Operator to Its Subqueries

```
SELECT ?c ?n WHERE {
?c g:featureCode g:School.
?c g:name ?n
?c g:parentCountry ?l .
?l g:name ?x .
FILTER (regex(str(?n), "Toronto", "i"))
VALUES (?x) {("Canada")}
VALUES (?f) {(g:School)} }
```

Fig. 16. Rewritten Query for Subquery with VALUES

10 Experimental Evaluation

In this section, we evaluate the proposed federated multiple query optimization method (FMQO) over both real and synthetic datasets, WatDiv and LargeRDFBench. We also evaluate FMQO over the real dataset FedBench [35] and the results are given in appendix. We compare our system with four state-of-the-art federated SPARQL query engines: FedX [36], SPLENDID [11], HiBISCuS [32] and our previous study [27]. The codes of FedX, SPLENDID and HiBISCuS have been released in GitHub[5]. We also release our codes in GitHub[6].

10.1 Setting

WatDiv. WatDiv [1] is a benchmark that enables diversified stress testing of RDF data management systems. In WatDiv, instances of the same type can have the different sets of attributes. We generate three datasets varying sizes from 100 million to 300 million triples. WatDiv provides its own workload generator and we directly use it to generate different workloads for testing.

LargeRDFBench. LargeRDFBench [31] is a comprehensive benchmark extended from a well-known benchmark FedBench [35] for testing and analyzing both the efficiency and effectiveness of federated RDF systems. There are 13 datasets in different domains, and the number of triples is more than one billion. It provides 40 benchmark queries, and we use these 40 queries as seeds and generate different kinds of workloads in our experiments. For each benchmark query, we remove all constants (strings and URIs) at subjects and objects and replace them with variables as a template. Then, we instantiate these templates with actual RDF terms from the dataset. By default, we generate 150 queries.

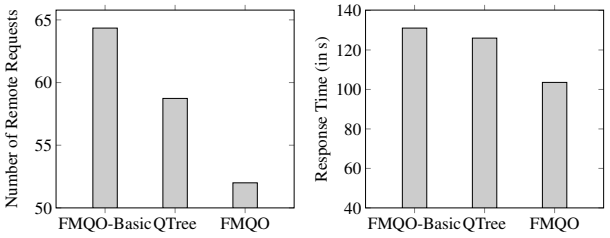
We conduct all experiments on a cluster of machines running Linux, each of which has one CPU with four cores of 3.06GHz. Each site has 16GB memory and 150GB disk storage. The prototype is implemented in Java. At each site, we install Sesame 2.7 to build up an RDF source. Each source can only communicate with the control site through HTTP requests and cannot communicate with each other. For LargeRDFBench, we assume that each dataset is resident at a source site. For WatDiv, we cluster by the types occurring in the dataset, finally obtaining m subdatasets, where parameter m varies from 2 to 8. The default value for m is 6.

5. https://github.com/dice-group/LargeRDFBench
6. https://github.com/QiGe57/MultiQueryOptimization

10.2 Evaluation of Proposed Techniques

In this section, we use WatDiv 100M and a query workload of 150 queries to evaluate each proposed technique in this paper. In other words, 150 queries are posed simultaneously to the federated RDF systems storing WatDiv 100M.

Effect of the Query Decomposition and Source Selection Technique. First, we evaluate the effectiveness of our source topology-based technique proposed in Section 5. In Fig. 17, we compare our technique with the baseline that does not utilize any topological information to prune irrelevant sources during source selection (denoted as FMQO-Basic). We also compare the source selection method proposed in [13], [29], which is denoted as QTree. It only uses the neighborhood information in the source topology to prune some irrelevant sources for each triple patterns.



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

Fig. 17. Evaluating Source Topology-based Source Selection Technique

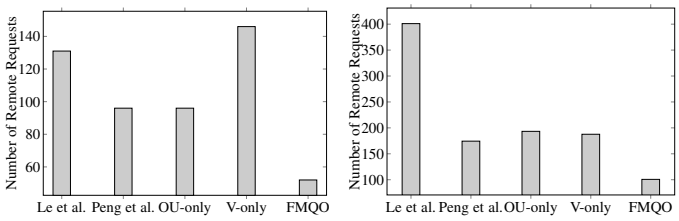
Obviously, FMQO-Basic does not prune any sources, so it leads to the most number of remote requests and the largest query response time. QTree only uses the neighborhood information and does not consider the whole topology of relevant sources. Hence, the effectiveness of its pruning rule is limited. Many queries contain triple patterns containing constants with high selectivity that can be localized to a few sources. Then, for other triple patterns, if some of their 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 (as shown in Fig. 17(a)) and the least query response time (as shown in Fig. 17(b)).

Effect of the Rewriting Strategies. In this experiment, we compare our SPARQL query rewriting strategies with a baseline using only the OPTIONAL-UNION-based rewriting strategy (denoted as OU-only) and a baseline only using only the VALUES-based rewriting strategy (denoted as V-only). We also re-implement the rewriting strategies proposed in [20] (denoted as Le et al.) and [27] (denoted as Peng et al.) to rewrite subqueries. Our query rewiring technique is denoted as FMQO. Fig. 18 shows the experiments by using the five rewriting strategies.

For a workload, since the number of subqueries sharing common subgraphs is often more than the number of subqueries of the same structure, V-only leads to the largest number of rewritten queries which results in most remote requests. Le et al. first cluster all subqueries into groups, and then find the maximal common edge subgraphs (MCESs) of the group. Thus, the number of rewritten queries generated by Le et al. is no less than the number of the groups. In contrast, OU-only, Peng et al. and FMQO use some triple patterns to hit subqueries. Hence, the number of rewritten queries they generate is the number of selected triple

patterns. In practice, most MCESs found by Le et al. also contain most of 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.

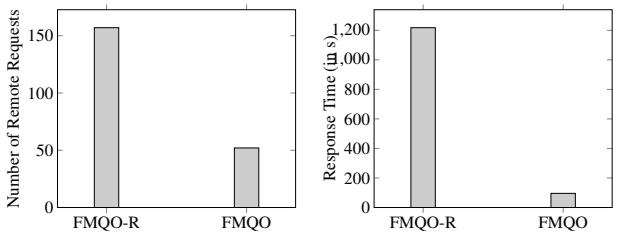
Since OU-only generates smaller number of rewritten queries and share more computation than Le et al., it can result in faster query response time. A query with OPTIONAL operators is slower than a query with VALUES operators, assuming they have the same main pattern, since the former is based on left-join and the latter is based on the inline mappings to the variable. Hence, although more queries are generated by using V-only rewriting strategy, V-only takes about half the time of Le et al. and nine-tenths of OU-only, as shown in Fig. 18(b). Furthermore, because our rewriting technique takes advantages of both the rewriting strategy using OPTIONAL and UNION and the rewriting strategy only using VALUES, FMQO can outperform others.



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

Fig. 18. Evaluating Different Rewriting Strategies

Evaluation of the Cost Model. In this section, we evaluate the effectiveness of our cost model and cost-aware rewriting strategy in Section 6.2 in Fig. 19. We design a baseline (FMQO-R) that does not selects the locally optimal triple patterns as Algorithm 1 but randomly select triple patterns to rewrite subqueries.

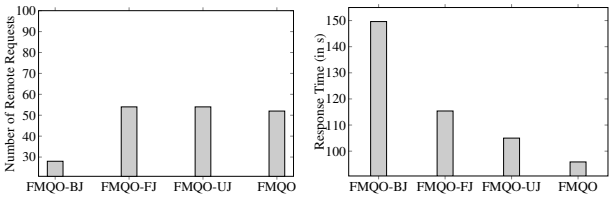


(a) Number of Remote Requests (b) Response Time

Fig. 19. Evaluating Cost Model

As shown in Fig. 19(a), we find out that cost-based selection causes fewer remote requests than FMQO-R. This is because the patterns with lower cost are shared by more subqueries, which results in fewer rewritten queries. In addition, in our cost-based rewriting strategy, we prefer selective query patterns, resulting in lower query response times, as shown in Fig. 19(b). Generally, the cost model-based approach can provide tenfold speed up.

Effect of Join Optimization Techniques. We evaluate our optimized join strategy proposed in Section 8, by comparing it with three baselines. The first baseline runs multiple federated queries with only rewriting strategies but no any optimization techniques for joins (denoted as FMQO-BJ), the second one uses the FILTER-based optimization for joins [10], [38] (denoted as FMQO-FJ) and the third uses the UNION-based optimization for joins [36] (denoted as FMQO-UJ). Our proposed VALUES-based optimization for joins is denoted as FMQO. Although the optimization techniques of FMQO-FJ, FMQO-UJ and FMO cause some extra remote requests for joins due to the rewriting, they can reduce the high join cost by avoiding the evaluation of unselective subqueries. Especially, compared with FMQO-BJ, FMQO reduces join processing time by 40%, as shown in Fig. 21(b). Note that, the number of remote requests of FMO is a little smaller than FMQO-FJ and FMQO-UJ, because the VALUES-based optimization can rewrite the subqueries in a more compact way and more intermediate results can be rewritten into one rewritten query for joins, as shown in Fig. 21(a).



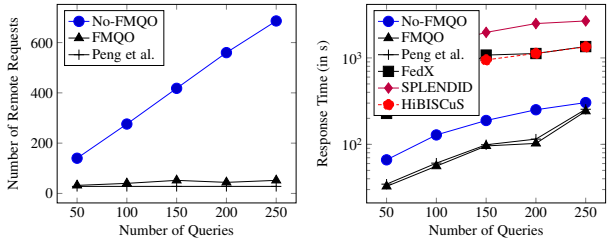
(a) Number of Remote Requests (b) Response Time

Fig. 20. Effect of Optimization Techniques for Joins

10.3 Evaluating Scalability

In this section, using WatDiv, we test the scalability of our method in four aspects: varying the number of queries, varying the number of query templates, varying the dataset sizes and varying the number of sources. We design a baseline that runs multiple federated queries sequentially (denoted as No-FMQO), which does not employ the source topology graph and any optimizations for multiple queries. We compare our method with FedX [36], SPLENDID [11], HiBISCuS [32] and our previous multiple query optimization technique [27] (denoted as Peng et al.). By default, the dataset is WatDiv 100M, the number of sources is 6, the number of queries is 150 and the number of templates is 10.

Varying Number of Queries. We study the impact of the size of the query set, which we vary from 100 to 250 queries, in increments of 50. Fig. 21 shows the experimental results.



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

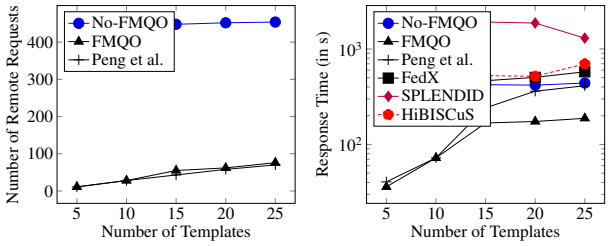
Fig. 21. Varying Number of Queries

Due to query rewriting, Peng et al. and FMQO can rewrite many subqueries into fewer queries, which results in smaller number of remote requests than No-FMQO, as shown in Fig. 21(a). FMQO can reduce the number of remote accesses by 4/5-9/10, compared with No-FMQO. For evaluation times as shown in Fig. 21(b), since No-FMQO does not share any computation, it takes a third more time than FMQO. Because FMQO further rewrites some intermediate matches into some unselective subqueries by using VALUES operators, the number of remote requests in Peng et al. is a little fewer than FMQO (as shown in Fig. 21(a)) and the cost-driven rewriting-based join method can ensure the better performance of FMQO (as shown in Fig. 21(b)).

FedX, SPLENDID and HiBISCuS do not provide their numbers of remote requests, so we do not compare FMQO with them in Fig. 21(a). In addition, FedX, SPLENDID and HiBISCuS always employ a semijoin algorithm to join intermediate results. Since almost all partial matches of subqueries participate in the join in WatDiv, the semijoin algorithm is not always efficient for

this dataset. Hence, FMQO is twice as fast as FedX, SPLENDID and HiBISCuS (as shown in Fig. 21(b)).

Varying Number of Query Templates. We study the impact of the number of templates. We vary the number of templates from 5 to 25, in increments of 5. The results are shown in Fig. 22. As before, we do not compare FMQO with FedX, SPLENDID and HiBISCuS in Fig. 22(a).



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

Fig. 22. Varying Number of Query Templates

As the number of queries is kept constant, more templates mean that fewer queries have the common subgraphs. Since FMQO and Peng et al. use the common subgraphs to rewrite queries, fewer queries containing the common subgraphs result in more number of rewritten queries. More rewritten queries mean that their number of remote requests increases and less computation is shared by different queries, so the performance of FMQO and Peng et al. become worse as shown in Fig. 22(b) and the number of remote requests increases with the number of templates as shown in Fig. 22(a). However, the response time of FMQO increases slowly, so it is still less than 30% of FedX, SPLENDID and HiBISCuS, and two thirds less than No-FMQO.

Varying Dataset Size. We investigate the impact of dataset size. We generate three WatDiv datasets varying the sizes from 100 million to 300 million triples. Fig. 23 shows the results. While the dataset size has little effect on the number of remote requests, it clearly affects evaluation times. As the size of RDF datasets gets larger, the response time of all six methods increases. However, the rate of increase for FMQO is smaller than other competitors. The response time of FMQO decreases from 50% of No-FMQO to 30% of No-FMQO, is always smaller than Peng et al. and less than 25% of SPLENDID, FedX and of HiBISCuS.

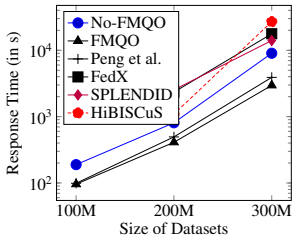
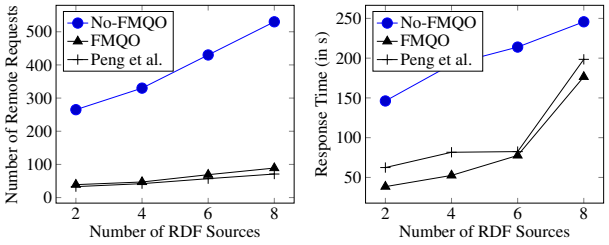


Fig. 23. Varying Size of Datasets

Varying Number of Sources. In this experiment, we vary the number of sources from 2 to 8. Fig. 24 presents the scalability of our solution adapting to different number of RDF sources. As the number of sources increases, a query may be relevant to more sources and it is decomposed into more subqueries. Thus, more rewritten queries and more joins are generated to evaluate the input queries. However, FMQO grows much slower than No-FMQO in both the number of remote accesses and query response time. Although FMQO further rewrites some intermediate matches by using VALUES for joins which results more remote requests than Peng et al., its cost-driven rewriting-based join method can ensure the better performance. Finally, the experiments confirms that FMQO has better scalability with the number of sources.

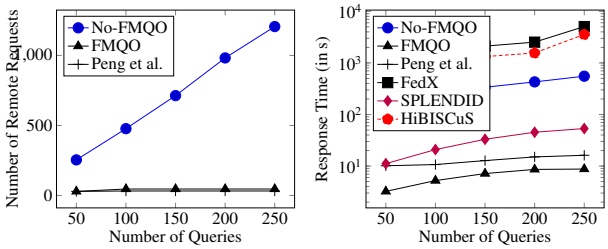


(a) Number of Remote Requests (b) Response Time

Fig. 24. Varying Number of RDF Sources

10.4 Performance over Real Datasets

In this experiment, we test FMQO using the real RDF dataset, LargeRDFBench. Since real datasets do not allow changing data sizes and the number of sources, we only test the methods by varying the number of queries in Fig. 25.



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

Fig. 25. Experimental Results over LargeRDFBench

Experiments over LargeRDFBench confirm that our method lead to fewer remote requests and better query performance than FedX [36], SPLENDID [11], HiBISCuS [32], Peng et al. [27] and our proposed baseline, since FMQO can rewrite many queries into fewer rewritten queries and the cost-driven rewriting strategy guarantees that rewritten queries are always faster than evaluating them sequentially. In addition, our rewriting-based join method can rewrite unselective subqueries in a more selective way. Although our join method may result in more remote requests than others, it avoids the high cost of executing the unselective subqueries, which improves the performance.

11 Conclusions and Future Work

In this paper, we study the problem of multiple query optimization over federated RDF systems. Our optimization framework, which integrates a novel algorithm to identify common subqueries with a cost model, rewrites queries into fewer queries while considering some characteristics of SPARQL 1.1. We also discuss how to efficiently selection relevant sources and join intermediate results by using the operators in SPARQL 1.1. Experiments show that our optimizations are effective.

Although we consider some new operators introduced in SPARQL 1.1 (e.g. VALUES), there are other features in SPARQL 1.1 that are still not studied in federated RDF systems, like aggregate functions and property paths. Studying these features in federated RDF systems is our future research plan. Moreover, how to access the federated RDF systems in more flexible ways, like keyword search and natural language question answering, is also a open problem that we intend to sutdy.

Acknowledgement. This work was supported by The National Key Research and Development Program of China under grant 2018YFB1003504, NSFC under grant 61702171, 61932001, 61961130390, 61622201 and 61532010, and Hunan Provincial Natural Science Foundation of China under grant 2018JJ3065. This work is also supported by Beijing Academy of Artificial Intelligence (BAAI). Tamer Ozsu's work has been supported by a Discovery Grant from Natural Sciences and Engineering Research Council (NSERC) of Canada. Lei Zou is the corresponding author of this paper.

REFERENCES

[1] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *ISWC*, pages 197–212, 2014.

[2] K. Anyanwu. A Vision for SPARQL Multi-Query Optimization on MapReduce. In *Workshops of ICDE*, pages 25–26, 2013.

[3] C. B. Aranda, M. Arenas, and Ó. Corcho. Semantics and Optimization of the SPARQL 1.1 Federation Extension. In *ESWC*, pages 1–15, 2011.

[4] C. B. Aranda, M. Arenas, Ó. Corcho, and A. Polleres. Federating Queries in SPARQL 1.1: Syntax, Semantics and Evaluation. *J. Web Semant.*, 18(1):1–17, 2013.

[5] C. B. Aranda, A. Polleres, and J. Umbrich. Strategies for Executing Federated Queries in SPARQL1.1. In *ISWC*, pages 390–405, 2014.

[6] M. Arenas and J. Pérez. Federation and Navigation in SPARQL 1.1. In *Reasoning Web*, pages 78–111, 2012.

[7] A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *PVLDB*, 11(2):149–161, 2017.

[8] S. Cebiric, F. Goasdoué, H. Kondylakis, D. Kotzinos, I. Manolescu, G. Troullinou, and M. Zneika. Summarizing Semantic Graphs: A Survey. *VLDB J.*, 28(3):295–327, 2019.

[9] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis. CliqueSquare: Flat Plans for Massively Parallel RDF Queries. In *ICDE*, pages 771–782, 2015.

[10] O. Görlitz and S. Staab. *Federated Data Management and Query Optimization for Linked Open Data*, pages 109–137. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[11] O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *COLD*, 2011.

[12] M. Hammoud, D. A. Rabbou, R. Nouri, S. Beheshti, and S. Sakr. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *PVLDB*, 8(6):654–665, 2015.

[13] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich. Data Summaries for On-demand Queries over Linked Data. In *WWW*, pages 411–420, 2010.

[14] A. Harth and S. Speiser. On Completeness Classes for Query Evaluation on Linked Data. In *AAAI*, 2012.

[15] O. Hartig. SPARQL for a Web of Linked Data: Semantics and Computability. In *ESWC*, pages 8–23, 2012.

[16] A. Hogan, A. Harth, and A. Polleres. Scalable Authoritative OWL Reasoning for the Web. *Int. J. Semantic Web Inf. Syst.*, 5(2):49–90, 2009.

[17] R. Isele, J. Umbrich, C. Bizer, and A. Harth. LDspider: An Open-source Crawling Framework for the Web of Linked Data. In *ISWC*, 2010.

[18] G. Konstantinidis and J. L. Ambite. Optimizing Query Rewriting for Multiple Queries. In *IIWeb*, pages 7:1–7:6, 2012.

[19] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[20] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable Multi-query Optimization for SPARQL. In *ICDE*, pages 666–677, 2012.

[21] J. Li, A. Deshpande, and S. Khuller. Minimizing Communication Cost in Distributed Multi-query Processing. In *ICDE*, pages 772–783, 2009.

[22] C. Liu, J. Qu, G. Qi, H. Wang, and Y. Yu. HadoopSPARQL: A Hadoop-Based Engine for Multiple SPARQL Query Answering. In *ESWC (Satellite Events)*, pages 474–479, 2012.

[23] G. Montoya, H. Skaf-Molli, and K. Hose. The Odyssey Approach for Optimizing Federated SPARQL Queries. In *ISWC*, pages 471–489, 2017.

[24] C. Nomikos, M. Gergatsoulis, E. Kalogeros, and M. Damigos. A Map-Reduce Algorithm for Querying Linked Data based on Query Decomposition into Stars. In *EDBT/ICDT Workshops*, pages 224–231, 2014.

[25] E. C. Ozkan, M. Saleem, E. Dogdu, and A. N. Ngomo. UPSP: Unique Predicate-based Source Selection for SPARQL Endpoint Federation. In *PROFILES@ESWC*, 2016.

[26] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[27] P. Peng, L. Zou, M. T. Özsu, and D. Zhao. Multi-query Optimization in Federated RDF Systems. In *DASFAA*, pages 745–765, 2018.

[28] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[29] F. Prasser, A. Kemper, and K. A. Kuhn. Efficient Distributed Query Processing for Autonomous RDF Databases. In *EDBT*, pages 372–383, 2012.

[30] B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *ESWC*, pages 524–538, 2008.

[31] M. Saleem, A. Hasnain, and A.-C. N. Ngomo. LargeRDFBench: A Billion Triples Benchmark for SPARQL Endpoint Federation. *Journal of Web Semantics*, 48(0), 2018.

[32] M. Saleem and A. N. Ngomo. HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation. In *ESWC*, pages 176–191, 2014.

[33] M. Saleem, S. S. Padmanabhuni, A. N. Ngomo, A. Iqbal, J. S. Almeida, S. Decker, and H. F. Deus. TopFed: TCGA Tailored Federated Query Processing and Linking to LOD. *J. Biomedical Semantics*, 5:47, 2014.

[34] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF Querying with SPARQL on Spark. *PVLDB*, 9(10):804–815, 2016.

[35] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In *ISWC*, pages 585–600, 2011.

[36] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *ISWC*, pages 601–616, 2011.

[37] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *WWW*, pages 595–604, 2008.

[38] J. Zemánek and S. Schenk. Optimizing SPARQL Queries over Disparate RDF Data Sources through Distributed Semi-Joins. In *International Semantic Web Conference (Posters & Demos)*, 2008.

Peng Peng received his BS degree and Ph.D. degree in Computer Science at Beijing Normal University and Peking University in 2009 and 2016, respectively. Now, he is an assistant professor of Hunan University. His research interests include graph database, distributed RDF system.

Qi Ge is currently pursuing a MS degree advised by Dr. Peng Peng in Computer Science and Technology at Hunan University. Her research interests include graph database, data mining and distributed RDF system.

Lei Zou received his B.S. degree and Ph.D. degree in Computer Science at Huazhong University of Science and Technology (HUST) in 2003 and 2009, respectively. Now, he is a professor in Institute of Computer Science and Technology of Peking University. His research interests include graph database and semantic data management.

M. Tamer Özsu is a professor of computer science and an associate dean (research) of the Faculty of Mathematics at the Cheriton School of Computer Science, University of Waterloo. His current research focuses on large-scale data distribution and management of unconventional data. He is a fellow of the IEEE and the ACM, an elected member of Turkish Academy of Science, and a member of Sigma Xi.

Zhiwei Xu is currently a senior student pursuing BE degree in software engineering in Hunan University. He is a research assistant advised by Dr. Peng Peng. His research interest include data mining, distributed RDF system and deep learning.

Dongyan Zhao received the B.S. degree, M.S. degree and Ph.D. degree from Peking University in 1991, 1994 and 2000, respectively. Now, he is a professor in Institute of Computer Science and Technology of Peking University. His research interest is on information processing and knowledge management.