Multi-Query Optimization in Federated RDF Systems

Peng Peng[1], Lei Zou[1], M. Tamer Özsu[2], Dongyan Zhao[1]

[1]*Peking University, China;*
[2]*University of Waterloo, Canada;*
{ pku09pp,zoulei,zhaodongyan}@pku.edu.cn, tamer.ozsu@uwaterloo.ca

ABSTRACT

This paper revisits the classical problem of multi-query optimization in the context of federated RDF systems. We propose a cost-aware SPARQL rewriting-based approach to share the common computation during evaluation of multiple queries. 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 topological information to filter out irrelevant sources and join intermediate results during multi-query evaluation. The extensive experimental studies over both real and synthetic RDF datasets show that the proposed techniques are effective, efficient and scalable.

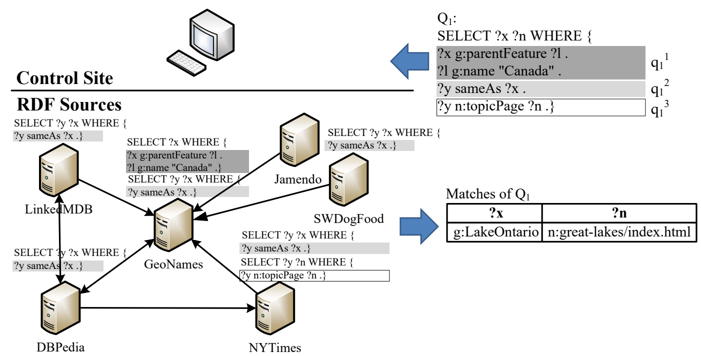


Figure 1: A Federated SPARQL Query

1. INTRODUCTION

Recently, many data providers have published, shared and interlinked their structured datasets using open standards such as RDF and SPARQL [5]. RDF is a self-describing data model that represents data as triples of the form ⟨subject, property, object⟩ for modelling information in the Web, while SPARQL is a query language to retrieve and manipulate data stored in RDF format. Although many data providers publish their RDF data, they often store their RDF triple files at their own *autonomous* sites some of which have *SPARQL endpoints* to enable applications to query RDF knowledge bases via SPARQL. An autonomous site with a SPARQL endpoint is called an *RDF source* in this paper.

To integrate and provide transparent access many RDF sources, federated RDF systems have been proposed [19, 23], in which, a control site is introduced to provide a common interface for users to issue SPARQL queries. Based on the metadata, the control site takes care of rewriting and optimizing the query. In particular, a SPARQL query Q is decomposed into several *local queries* that are evaluated on relevant sources; then, these local query results are joined together to form complete results that are returned to users.

Federated RDF systems have been widely used in many applications [11]. For example, European Bioinformatics Institute has provided a uniform platform[1] for users to query multiple bioinformatics RDF sources, including BioModels, Biosamples, ChEMBL, Ensembl, Atlas, Reactome and UniProt. It also provides a unified way to query across sources using the SPARQL query language.

So far, many federated RDF systems have been proposed [19, 23, 1, 8, 20] in the literature. A popular federated RDF benchmark—FedBench [22]—is often introduced to evaluate the performance of the federated RDF systems. Example 1 shows a sample federated query (in FedBench) involving multiple RDF sources.

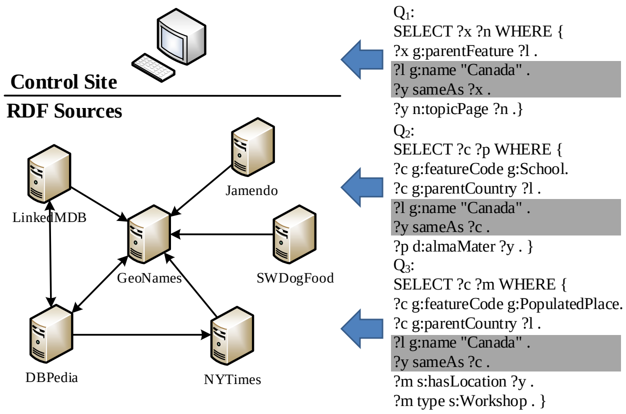
Example 1. ***(A Federated SPARQL Involving Multiple Sources)*** *There are six interconnected RDF sources of different domains, such as NYTimes, GeoNames, SWDogFood and DBPedia, in Fig. 1. Assume a person wants to find out all news about Canada. From NYTimes, s/he finds all news pages associated with news locations (such as cities or towns), but, NYTimes does not explicitly identify exhaustive Canadian places. Fortunately, the places in NYTimes are linked to the counterparts (using "sameAs" property) in GeoName, a worldwide geographical RDF database that states explicitly which places are located in Canada. Therefore, the federated SPARQL query Q_1 can be formulated (Fig. 1) over the federated RDF system.*

To evaluate query Q_1, it is decomposed into three local queries q_1^1, q_1^2 and q_1^3, which are sent to relevant sources. The details of query decomposition and relevant source selection are discussed in Section 4. These local query results are sent back to the query originating site and joined together to form complete results. □

However, all of existing federated RDF systems only consider query evaluation for a single SPARQL and miss the opportunity for multiple query optimization. Real SPARQL query workloads reveal that many SPARQL queries are often posed simultaneously. Let us consider a real SPARQL query workload over DBPedia[2].

[1]http://www.ebi.ac.uk/rdf/

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



The shaded triple patterns correspond to the common subgraph.

Figure 2: Multiple Federated SPARQL Queries

It records 8 millions SPARQL queries issued in 14 days. On average, there are more than six SPARQL queries per second. Take another example. According to the workload of SPARQL queries over Linked Geo Data[3], more than 600 queries are issued per second at the peak time. Therefore, it is desirable to design multiple SPARQL query optimization strategy.

Consider a batch of queries (e.g., Q_1, Q_2 and Q_3 in Example 2) that are posed simultaneously over federated RDF systems in Figure 1. The straightforward approach is to evaluate them sequentially. However, it is easy to identify some common substructures over these three queries, meaning that there are some sharing computation. This motivates us to revisit the classical problem of multi-query optimization in the context of federated RDF systems.

EXAMPLE 2. ***(Multiple Federated SPARQL Queries)*** *Fig. 2 shows two additional queries Q_2 and Q_3 besides Q_1. Q_2 retrieves all people who graduated from Canadian universities, such as Kim Campbell. Q_3 is to retrieve all semantic web-related workshops held in Canada, such as SWDB 2004. All the three queries will be used as running examples throughout this paper.* □

1.1 Challenges & Our Solutions

Although multi-query optimization have been well studied in distributed relational databases [15], federated RDF systems are more autonomous than classical distributed relational database systems and some techniques commonly referred to as data movement and data/query shipping [13] are not easily applicable. For example, we cannot require one source to send intermediate results directly to another source [11]. Moreover, moving data from one source to another one for join processing [15] is also infeasible.

To the best of our knowledge, only *one* proposal about multiple SPARQL query optimization exists in the literature [14], but *only in the centralized environment*, where all RDF datasets are collected in one physical database. Given a batch of SPARQL queries, [14] proposes to identify all maximal common edge subgraphs (MCES) among query graphs. Each MCES leads to a possible MCES-based rewritten query. Since [14] employs the exponential time complexity algorithm to identify MCES, which means that the rewriting approach in [14] is not scalable with respect to the size and the number of query graphs.

Our method is different from [14] in two major technical aspects besides that we consider the distributed RDF context while [14] focuses on the centralized one.

[3]http://aksw.github.io/LSQ/

1. [14] only considers the "OPTIONAL" clause in the SPARQL re-writing, while our method considers both "OPTIONAL", "FILTER" in the re-writing strategy. Obviously, the latter provides more optimization opportunities and experiments also confirm the superiority.
2. We propose a cost model-driven greedy solution for multiple query rewriting; meanwhile, our cost model considers both the common substructures and the selectivity of queries. Also, the linear time complexity of our greedy algorithms guarantees the scalability of our rewriting strategy with regard to the sizes and numbers of queries that are posed simultaneously.

Furthermore, we study relevant source selection and partial match joining in federated RDF systems, which do not arise in the centralized counterpart [14].

First, we optimize the query-decomposition and relevant source selection in the context of federated RDF systems, since existing techniques often overestimate the set of relevant sources. In this paper, 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 30% remote requests compared with the existing data localization techniques.

Second, we study how to optimize partial match joining during multiple query processing in federated RDF systems. In real applications, queries in a given workload queries often have similar joining structures. This common joining structures provide the potential of sharing common computation. In this paper, we merge multiple joining operations with the same joining structures together to share the common joining computation. Experiments show that the optimized joining approach can improve 10% query performance compared with the naive joining techniques.

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 a nutshell, we make the following contributions.

- We propose to employ the topology relation between sources in federated RDF systems to filter out irrelevant sources.
- We take into account the unique properties of SPARQL and federated RDF systems, and propose a cost-driven query-rewriting technique to reduce both the query response time and the number of remote requests.
- We propose to optimize joining local query results in the context of multiple SPARQL query over federated RDF systems, which avoids duplicate computation.
- We do experiments over both real and synthetic interconnected RDF sources and SPARQL query workloads to confirm the superiority of our approach.

2. BACKGROUND

2.1 RDF and Federated RDF System

DEFINITION 1. ***(RDF Graph)***. *An RDF graph is denoted as $G = \{V, E, L\}$, where $V(G)$ is a set of vertices that correspond to all subjects and objects in RDF data; $E \subseteq V \times V$ is a set of directed edges that correspond to all triples in RDF data; L is a set of edge labels. For each edge $e \in E$, its edge label is its property.*

In the context of federated RDF systems, RDF graph G is distributed over different source sites. We learn from the definition of LOD as found in [10] to define the federated RDF graph.

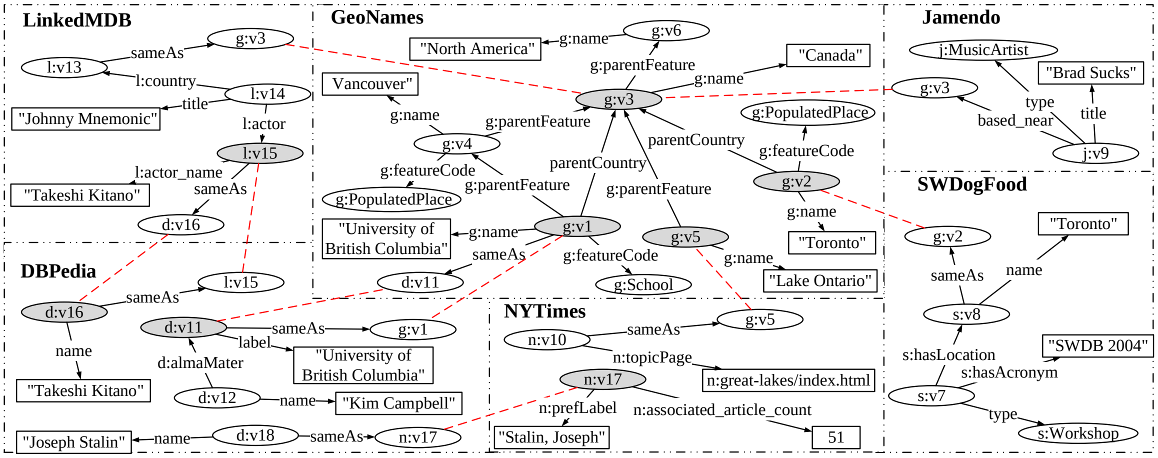


Figure 3: Example Federated RDF System

Definition 2. ***(Federated RDF System)*** *A* federated RDF graph *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 which models the fact that 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. $d(u)$ is unique for a given URL of vertex u.*

Obviously, the whole RDF graph G is formed by collecting all subgraphs in different sources, i.e., $\bigcup_{s \in S} g(s) = G$. Consider the RDF graph G distributed among six different sources in Fig. 3. Given a vertex "g:v2", d(g:v2)=GeoNames, where "g" is abbreviation of "GeoNames". This means that vertex "g:v2" is dereferenced by the host GeoNames.

Although vertex u may be contained in multiple sources, it is only dereferenced by the host source $d(u)$. In Fig. 3, the vertices at their host sources are denoted as the grey circles and they connect to the corresponding mirrors at other sources by dashed lines. Let us still consider the vertex "g:v2". It is distributed among two sources, GeoNames and SWDogFood, and only GeoNames is its host source.

2.2 SPARQL

SPARQL is a structured query language over RDF, where the basic graph pattern (BGPs) is its building block.

Definition 3. ***(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 an edge label in L (i.e., property) or the edge label is a variable.*

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

Definition 4. ***(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 $\{v_1, ..., v_n\}$. For $S' \subseteq S$, a subgraph M of $\bigcup_{s \in S'} g(s)$ with n vertices $\{u_1, ..., u_n\}$ is said to be a* match *of Q if and only if there exists a* function *μ from $\{v_1, ..., v_n\}$ to $\{u_1, ..., u_n\}$, 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 $\mu(v_i) \in \{u_1, ..., u_n\}$; (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 $[\![Q]\!]_{S'}$.

Definition 5. ***(Compatibility)*** *Given two BGP queries Q_1 and Q_2 over a set of sources S', μ_1 and μ_2 define two matching functions $V(Q_1) \rightarrow V$ and $V(Q_2) \rightarrow V$, respectively. μ_1 and μ_2 are* compatible *when for all $x \in V(Q_1) \cap V(Q_2)$, $\mu_1(x) = \mu_2(x)$, denoted as $\mu_1 \sim \mu_2$; otherwise, they are not compatible, denoted as $\mu_1 \nsim \mu_2$.*

Our notion of a SPARQL query can be defined recursively as follows by combining BGPs using the following standard SPARQL algebra operations [17].

Definition 6. ***(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 7. ***(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 4.*
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 $[\![Q]\!]_{S'} = ([\![Q_1]\!]_{S'} \bowtie [\![Q_2]\!]_{S'}) \cup ([\![Q_1]\!]_{S'} \backslash [\![Q_2]\!]_{S'}) = \{\mu_1 \cup \mu_2 \,|\, \mu_1 \in [\![Q_1]\!]_{S'} \wedge \mu_2 \in [\![Q_2]\!]_{S'} \wedge (\mu_1 \nsim \mu_2)\}$*
5. *If $Q = Q_1\ FILTER\ F$, then $[\![Q]\!]_{S'} = \Theta_F([\![Q_1]\!]_{S'}) = \{\mu_1 | \mu_1 \in [\![Q_1]\!]_{S'} \wedge \mu_1\ satisfies\ F\}$*

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:

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

3. FRAMEWORK

This section outlines our framework. For conciseness, as BGP is a build block of a SPARQL query, we only consider BGP queries in this paper. We include handling general SPARQLs in Appendix. Our framework consists of three steps: *query decomposition and source selection*, *local query evaluation* and *partial match joining* (see Fig. 4). We briefly review the three steps as follows.

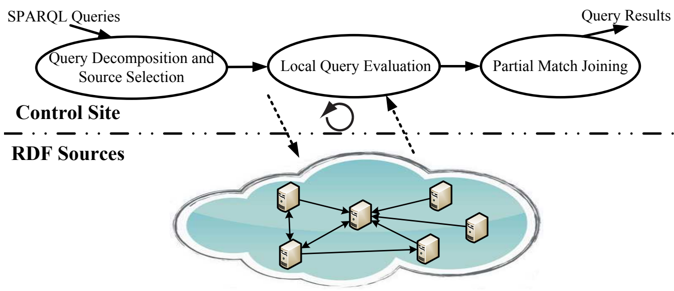


Figure 4: Scheme for Federated SPARQL Query Processing

Query Decomposition and Source Selection. Given a query Q, we first decompose Q into subqueries expressed on relevant sources. Existing solutions only consider the predicates in each source; thus, they often overestimate the set of relevant sources. In this paper, we propose to utilize the topology information to further filter out irrelevant sources. The details of the baseline and our optimized solution will be discussed in Section 4.

Specifically, given a batch of SPARQL queries $\{Q_1, ..., Q_n\}$, we obtain local queries $\mathcal{Q} = \{q_1^1@S(q_1^1), ..., q_1^{m_1}@S(q_1^{m_1}); q_2^1@S(q_2^1), ..., q_2^{m_2}@S(q_2^{m_2}); ...; q_n^1@S(q_n^1), ..., q_n^{m_n}@S(q_n^{m_n})\}$, where $\{q_i^1@S(q_i^1), ..., q_i^{m_i}@S(q_i^{m_i})\}$ come from original SPARQL query Q_i and $S(q_i^j)$ is the set of relevant sources for local query q_i^j.

Local Query Evaluation. After the first step, we figure out multiple local queries and assign them to their relevant sources for evaluation. A source site may be assigned with multiple local queries that share common computation. This provides an opportunity for multi-query optimization. In this paper, we rewrite the local queries using FILTER and OPTIONAL operators, based on the cost model derived from real experimental results. The rewritten queries help reduce both the response time and the number of remote accesses.

Assume that source s receives a set of local queries $\mathcal{Q}_s = \{q_s^1, ..., q_s^n\}$, where $\mathcal{Q}_s \subseteq \mathcal{Q}$. After query rewriting, we obtain a set of rewritten queries $\hat{\mathcal{Q}}_s$ that will be sent to source s. Each rewritten query comes from a subset of $\mathcal{Q}_s$. After the local query evaluation, we distribute the results of rewritten query to the corresponding local queries. We will discuss the details in Section 5.

Partial Match Joining. For each local query q_i^j in $\mathcal{Q}$, collecting the matches at each relevant source in $S(q_i^j)$, we obtain all its matches. Assume that an original query Q_i $(i = 1, ..., n)$ is decomposed into a set of local queries $\{q_i^1@S(q_i^1), ..., q_i^{m_i}@S(q_i^{m_i})\}$, we obtain query results $[\![Q_i]\!]_S$ by joining $[\![q_i^1]\!]_{S(q_i^1)}$,..., $[\![q_i^{m_i}]\!]_{S(q_i^{m_i})}$ together. Considering the context of multiple SPARQL over a federated RDF system, we propose an optimized solution to avoid duplicate computation in join processing (in Section 6).

4. QUERY DECOMPOSITION AND SOURCE SELECTION

4.1 Existing Solution

Given a SPARQL Q_1 in Fig. 1, each triple pattern corresponds to a local query. Each triple pattern maps to a set of relevant sources based on the values of its subject, property and object [23]. Consider triple pattern "?y sameAs ?x" in Q_1. Since RDF sources "GeoNames", "NYTimes", "DBPedia", "SWDogFood" and "LinkedMDB" contain the property "sameAs", "?y sameAs ?x" has five relevant sources. However, triple pattern "?x g:parentFeature ?l" in Q_1 only maps to source "GeoNames", because that only source "GeoNames" has property "g:parentFeature".

However, some one-triple patterns can be combined together to form a larger local query. If a source is exclusively selected for a set of connected triple patterns, they can be combined together to form a larger local query. For example, both "?x g:parentFeature ?l " and "?l g:name 'Canada' " of Q_1 correspond to the same source, i.e., "GeoNames". Thus, these two triple patterns can be combined together to form a larger local query q_1^1. Note that, if a group of triple patterns shares exactly the same set of more than one RDF sources, they cannot be combined together. This is because that results for individual triple patterns can be joined across RDF sources.

For query Q_1, the baseline solution decomposes it into three local queries q_1^1, q_1^2 and q_1^3, as shown in Fig. 5. q_1^1 has a single relevant source "GeoNames"; q_1^2 has five sources except for "Jamendo" and the relevant source q_1^3 is only "NYTimes".

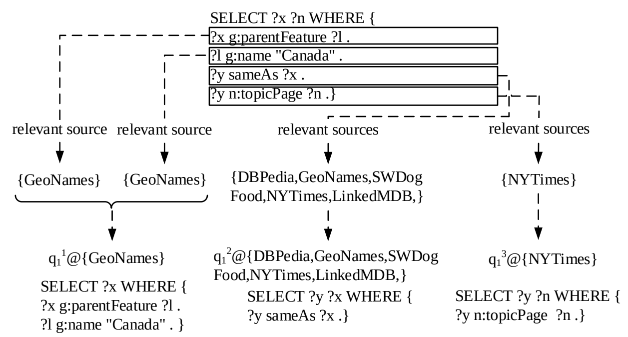


Figure 5: Basic Query Decomposition and Source Selection Result for Q_1

4.2 Source Topology Graph-Based Solution

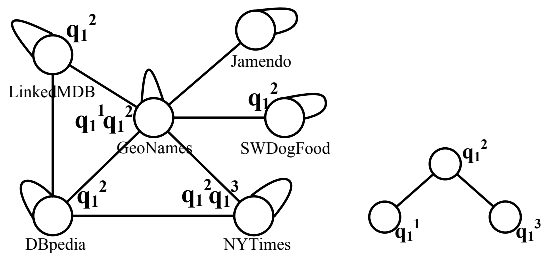
However, the existing solution [23] possibly overestimates the set of relevant sources. For example, for the triple pattern "?y sameAs ?x", there are five relevant sources in the baseline solution; but there is no result during join evaluation with actual mappings substituted in sources LinkedMDB and SWDogFood.

In this paper, we employ the topology structure between sources to filter out more irrelevant sources. A crawler for a federated RDF system can be used to figure out crossing edges between different sources. Based on the crossing edges, we define the *source topology graph* as follows. Note that the source topology graph is maintained in the control site.

DEFINITION 8. *(**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, such that (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 $\overrightarrow{u_iu_j} \in g(s_i)$ (or $\overrightarrow{u_ju_i} \in g(s_i)$ or $\overrightarrow{u_iu_j} \in g(s_j)$ or $\overrightarrow{u_ju_i} \in g(s_j)$), where $d(u_i) = s_i$ and $d(u_j) = s_j$.*

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 local queries. Specifically, according to the baseline solution, query Q is decomposed into several local queries 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. 6(a) shows the annotated STG T^* for query Q_1. Meanwhile, for a BGP Q, we build a *join graph* (denoted as $JG(Q)$) as follows. In a join graph, one vertex indicates a local query of Q. We introduce an edge between two vertices in the join graph if and only if the corresponding local queries are connected in the original SPARQL query. Fig. 6(b) shows the join graph $JG(Q_1)$.



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

Figure 6: Example Join Graph and Annotated Source Topology Graph for 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^*. If a local query q does not map to a source s in any homomorphism match, s is not the relevant source of subquery q. We formalize this observation in Theorem 1. For example, we can find three homomorphism matches of $JG(Q_1)$ over T^* in Fig. 7. However, sources "LinkedMDB" or "SWDogFood" does not match q_1^2 in any homomorphism match, so both of them can be pruned from the relevant sources of q_1^2.

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

PROOF. Given a source s that is pruned by the theorem for local query q, it means that there do not exist any homomorphism matches of q over T^* that contains s. Then, there exists another local query q' of which relevant sources do not contains triples that can join with results of q in s through $JG(Q)$. Hence, s cannot contribute any final results and can be filtered out. □

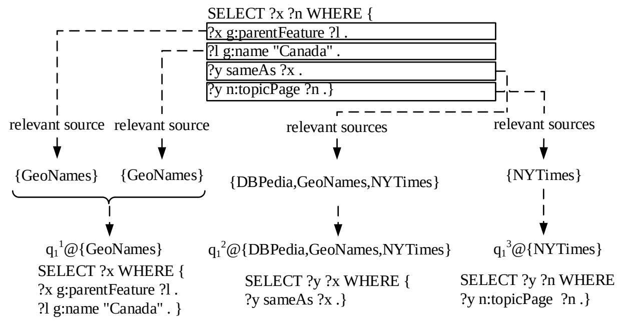
Match 1:	**Match 2:**	**Match 3:**
$q_1^1 \longrightarrow$ GeoNames	$q_1^1 \longrightarrow$ GeoNames	$q_1^1 \longrightarrow$ GeoNames
$q_1^2 \longrightarrow$ DBpedia	$q_1^2 \longrightarrow$ GeoNames	$q_1^2 \longrightarrow$ NYTimes
$q_1^3 \longrightarrow$ NYTimes	$q_1^3 \longrightarrow$ NYTimes	$q_1^3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

Figure 7: Homomorphism Matches for $JG(Q_1)$ over the Annotated Source Topology Graph

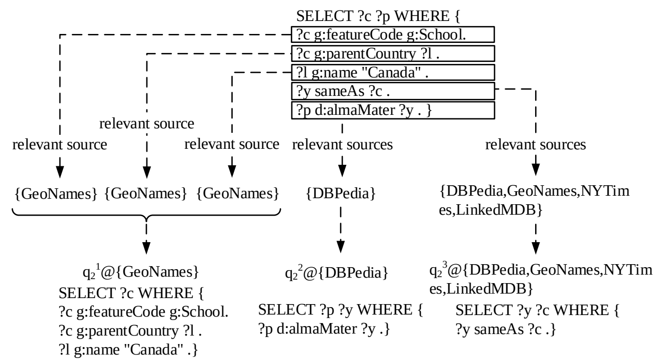
Analogously, we can also decompose Q_2 and Q_3 into local queries that are sent to their relevant sources. Fig. 8 shows the query decomposition and source selection results for all three queries in the running example.

5. LOCAL QUERY EVALUATION

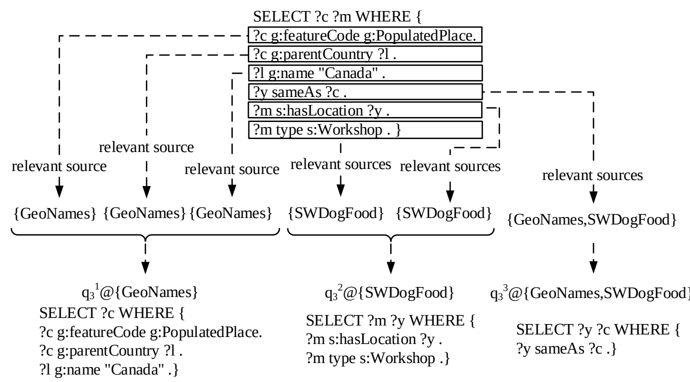
As mentioned in Section 1, multiple SPARQL queries may be posed simultaneously, and there is room for sharing computation when executing these queries.



(a) Q_1



(b) Q_2



(c) Q_3

Figure 8: Optimized Query Decomposition and Source Selection Results for Q_1, Q_1 and Q_3

Assume that a source site receives multiple local queries that share some common substructures. A possible optimization is to rewrite them as a single SPARQL and send it to relevant sources, which can save both the number of remote accesses and query response time. Obviously, different query rewriting may lead to different performances, thus, this section proposes a cost-driven query rewriting scheme.

5.1 Intuitions

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

5.1.1 OPTIONAL-based Rewriting

Given a set of local queries, we can rewrite them to a query $\hat{q}$ with multiple OPTIONAL clauses, where the main graph pattern of $\hat{q}$ is the common substructure among these local queries. Obviously, the rewriting can reduce the number of remote requests.

Formally, given a set of local queries $\{q_1@\{s\}, q_2@\{s\}, ..., q_n@\{s\}\}$ over the same source s, if p is their common subgraph among q_1,...,q_n, we rewrite these local queries to a query with OPTIONAL operator as follows.

$$\hat{q}@\{s\} = p\ OPT\ (q_1 - p)\ OPT\ (q_2 - p)\ ...\ OPT\ (q_n - p)@\{s\} \quad (1)$$

Let us consider two local queries over GeoNames, $q_1^1@\{GeoNames\}$ and $q_2^1@\{GeoNames\}$, as shown in Fig. 9. The two local queries are decomposed from Q_1 and Q_2. 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. 9. The rewritten query can avoid one remote request for GeoNames.

Because existing RDF stores implement OPTIONAL operators using left-joins, the result cardinality of a SPARQL query with OPTIONAL operator is upper bounded by result cardinality of its main graph pattern [14]. Thus, the query response time of the rewritten query does not increase much.

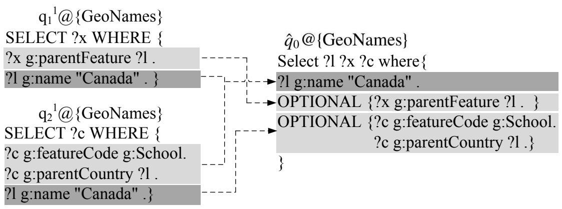


Figure 9: Rewriting Local Queries using OPTIONAL Operators

5.1.2 FILTER-based Rewriting

Let us consider two local queries $q_2^1@\{GeoNames\}$ and $q_3^1@\{GeoNames\}$ in Fig. 10, which is generated 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. Obviously, we can rewrite the two queries using FILTER, as shown in Fig. 10. In other words, if some local queries 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.

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

$$\hat{q}@\{s\} = p\ FILTER(\bigvee_{1\le i\le n}(\bigwedge_{v\in V(q_i)} f_i(v) = v))@\{s\} \quad (2)$$

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

Based on filter operator, we can perform selection over main pattern results. Thus, the result cardinality of SPARQL with FILTER operator is also upper bounded by result cardinality of its main graph pattern.

5.1.3 Hybrid Rewriting

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

5.2 Cost Model

Given a set of local queries, we may have a number of query rewriting strategies. It raises another problem that which one is better. Therefore, we propose a cost-driven approach, including the cost-model definition in Section 5.2 and the cost-driven query rewriting algorithm in Section 5.3.

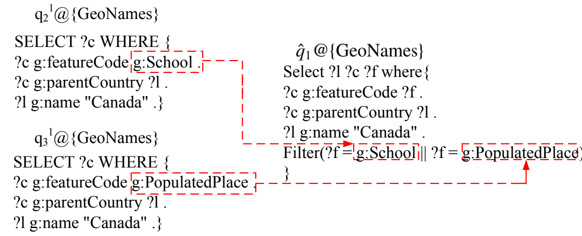


Figure 10: Rewriting Local Queries using FILTER Operator

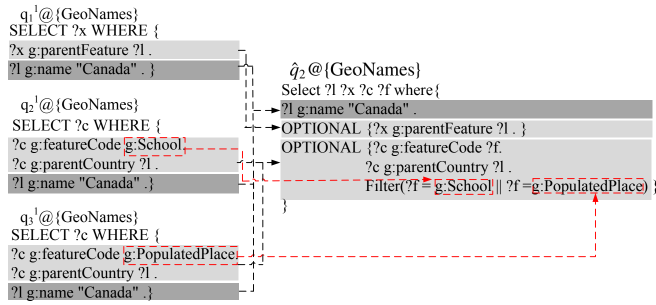
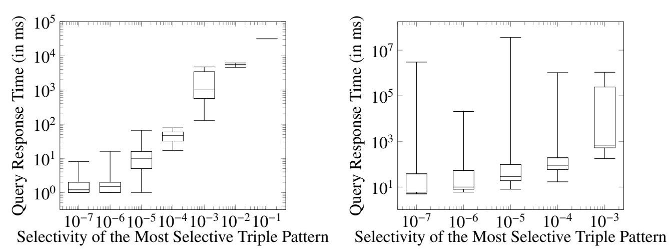


Figure 11: Rewriting Local Queries using OPTIONAL and FILTER Operators

5.2.1 Cost Model for BGPs

As mentioned in [14], selective triple patterns in BGP have higher priorities in evaluation. We verify the principle in both real and synthetic RDF repositories, such as DBpedia[4] and WatDiv [2], and experiment with a popular RDF store, Sesame 2.7[5]. For DBpedia, we download real SPARQL workload that records more than 8 millions SPARQL queries issued in 14 days of 2012 [6] and randomly sample 10000 queries to verify the principle; for WatDiv, to verify the principle, 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 show that query response time is positively associated with the selectivity of the most selective triple pattern e for both real and synthetic datasets on Sesame.



(a) DBPedia (b) WatDiv 10M

Figure 12: Experiment Results of the Relationship Between a Query and Its Most Selective Triple Pattern

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

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

[4] http://wiki.dbpedia.org/
[5] http://rdf4j.org/
[6] http://aksw.org/Projects/DBPSB.html

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

To estimate the selectivity of triple pattern, we employ the approach in [24], which estimates the selectivity without pre-computed statistics about the RDF source. Simply speaking, the heuristic is named *variable counting*. For this heuristics, the selectivity of a triple pattern is computed according to the type and number of unbound components and is characterized by the ranking $sel(S) < sel(O) < sel(P)$, i.e., subjects are more selective than objects and objects more selective than predicates [24]. More details about the selectivity estimation are given in [24].

5.2.2 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 pattern in the OPTIONAL clause and the expressions in the FILTER operator 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) [14]. This suggests that a good optimization should keep the result cardinality from the common subgraph as small as possible for two reasons: 1) the result cardinality of a SPARQL query with the OPTIONAL operators and FILTER operators is upper bounded by result cardinality of its main graph pattern clause since graph patterns in the OPTIONAL clause are simply left-joins and FILTER expressions are simply selection; 2) intermediate results from evaluating the main graph pattern is not well indexed, which implies that a non-selective main graph pattern will result in significantly more efforts in processing the corresponding rewriting graph patterns in the OPTIONAL clause and FILTER expressions.

According to the above reasons, we assume that the OPTIONAL clause and the FILTER expressions is evaluated on the the results of the main pattern. Specifically, we have the following cost model.

Given a SPARQL Q, its cost is defined as follows.

$$cost(Q) = \begin{cases} \min_{e \in E(Q)}\{sel(e)\} & if\ Q\ is\ a\ BGP; \\ \min\{cost(Q_1), cost(Q_2)\} & if\ Q = Q_1\ AND\ Q_2; \\ cost(Q_1) + cost(Q_2) & if\ Q = Q_1\ UNION\ Q_2; \\ cost(Q_1) + \Delta_1 & if\ Q = Q_1\ OPT\ Q_2; \\ cost(Q_1) + \Delta_2 & if\ Q = Q_1\ FILTER\ F; \end{cases} \tag{3}$$

where Δ_1 and Δ_2 are empirically trivial values [14].

5.2.3 Cost of Query Rewriting

Given a set of local queries $Q = \{q_1@S, q_2@S, ..., q_n@S\}$ over a source S using OPTIONAL and FILTER, if p is their common subgraph among q_1,...,q_n, we rewrite them into a SPARQL query $\hat{q}$. The cost of evaluating $\hat{q}$ is defined as follows.

$$cost(\hat{q}) = cost(p) + \Delta_1 + \Delta_2$$

As mentioned before, Δ_1 and Δ_2 are trivial values and $cost(\hat{q})$ is mostly credited to the evaluation of p. Hence, we ignore the trivial variables Δ_1 and Δ_2 assuming that the term $cost(p)$ is much larger.

$$cost(\hat{q}) = cost(p) = \min_{e \in p}\{sel(e)\}\} \tag{4}$$

Obviously, given a set local queries Q over a source S, we may have multiple query rewriting strategies. Thus, we need to define the cost of a specific query rewriting. Formally, we define the rewriting cost as follows.

DEFINITION 9. ***(Rewriting Cost)*** *Given a set of local queries $Q = \{q_1@s, q_2@s, ..., q_n@s\}$ on a source s using OPTIONAL and FILTER, if p is their common subgraph among q_1,...,q_n, 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:*

$$cost(Q, \hat{q}) = cost(\hat{q}) = \min_{e \in p}\{sel(e)\} \tag{5}$$

5.3 Local Query Rewriting Algorithm

The problem of query rewriting is that given a set Q of local queries $\{q_1, ..., q_n\}$, we compute a set $\hat{Q}$ of rewritten queries $\{\hat{q_1}, ..., \hat{q_m}\}$ ($m \leq n$) with the smallest cost (The cost function is defined in Equation 5). Note that each rewritten query $\hat{q_i}$ ($i = 1, ..., m$) comes from rewriting a set of original local queries in Q, where these local queries share the same main pattern p_i.

The cost of the whole rewriting is formatted as follows:

$$cost(Q, \hat{Q}) = \sum_{\hat{q} \in \hat{Q}} cost(\hat{q}) \tag{6}$$

Generally speaking, we find the set of common patterns P, where each local query contains at least one of patterns in P. Here, if a local query q contains a pattern $p \in P$, we call that p *hit* q. According to Section 5.1, if a set of local queries can be rewritten as a rewritten query $\hat{q}$, they must share one common main pattern p. Therefore, we have the following equation.

$$cost(Q, \hat{Q}) = \sum_{p \in P} cost(p) = \sum_{p \in P} \min_{e \in p}\{sel(e)\}\} \tag{7}$$

Given a set of original queries Q, Equation 7 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 local queries Q, finding an optimal rewriting $\hat{Q}$ to minimize the cost function in Equation 7 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. The weight set cover problem is defined as follows: given a set of elements $U = \{1, 2, ..., m\}$ (called the universe), a set S of n sets whose union equals the universe and a function w to mapping each set in S to a non-negative value, the set cover problem is to identify the least weighted subset of S whose union equals the universe.

To reduce the weighted set cover problem to the problem of selecting the optimal set of patterns, we map each set in S to a pattern and each element in the unverse U to a local query. A set s in S containing an element e in U maps to the pattern corresponding to s hitting the local query 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 [7], the problem of selecting the optimal set of patterns is also NP-complete. □

Then, we propose a greedy algorithm that iteratively selects the locally optimal triple pattern (see Algorithm 1).

Algorithm 1 is a greedy algorithm. Let Q denote all original local queries. At each iteration, we select a triple pattern e_{max} with the largest value $\frac{|Q'|}{sel(e_{max})}$, where Q' denote all local queries hit by e_{max}, i.e, these queries containing e_{max}. We propose a hybrid strategy to rewrite all quires in Q'. We divide Q' into several equivalence classes, where each class contains local queries with the same structure except for some constants on subject or object positions. Local queries in the same equivalence class can be rewritten to a query

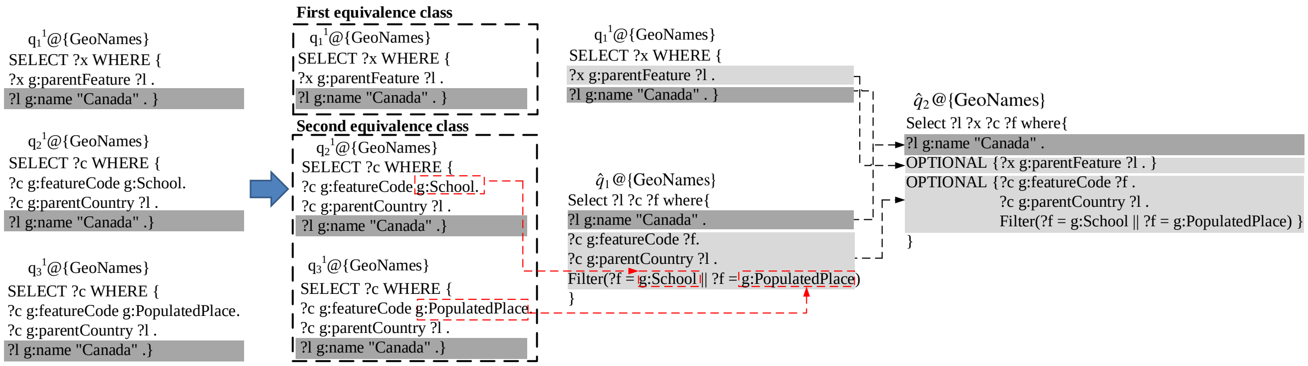


Figure 13: Example of Rewriting Local Queries

Algorithm 1: Local Query Rewriting Algorithm

Input: A set of local queries Q.
Output: A set of rewritten queries sets Q_{OPT}.
1 **while** $Q \neq \emptyset$ **do**
2 Select the triple pattern e_{max} with the largest value $\frac{|Q'|}{sel(e_{max})}$, where Q' is the set of local queries hit by e_{max};
3 Extract the largest common pattern p of queries in Q';
4 Build a rewritten query $\hat{q}$, where p is its main pattern;
5 Divide Q' into a collection of equivalence classes C, where each class contains local queries isomorphic to each other;
6 **for** *each class* $C \in \mathcal{C}$ **do**
7 Generalize a pattern p' isomorphic all patterns in C, where p' does not contain any constants;
8 Build a query pattern with p';
9 Add FILTER operators by mapping p' to patterns in C;
10 Add the pattern into $\hat{q}$ as an OPTIONAL pattern;
11 Add $\hat{q}$ into Q_{OPT};
12 $Q = Q - Q'$;
13 Return Q_{OPT};

pattern with FILTER operators as discussed in Section 5.1. Furthermore, all queries in Q' can be rewritten into SPARQL $\hat{q}$ with OPTIONAL operator using e_{max} as the main pattern. We remove queries in Q' from Q and iterate the above process until Q is empty.

Given local queries $q_1^1@\{GeoNames\}$, $q_2^1@\{GeoNames\}$ and $q_3^1@\{GeoNames\}$ in Fig. 13, we select the triple pattern "?l g:name "Canada"" in the first step. It hits the three local queries. 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"".

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

PROOF. According to Equation 7, selecting patterns to hit local queries is equivalent to selecting triple patterns to hit local queries. 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 local queries. A result in [6] shows that the approximation ratio of the greedy algorithm to the optimal solution of the weighted set-cover problem is $(1 + ln|\cup_{q\in Q} E(q)|)$. □

5.4 Postprocessing

Given a set of local queries Q that will be sent to source s, we rewrite them into $\hat{Q}$ and evaluate them at source s. Let $[\![\hat{q}]\!]_{\{s\}}$ denote the result set of $\hat{q}$ ($\in Q$) at source s. As we know, $\hat{q}$ is obtained by rewriting a set of original local queries in Q; thus, $[\![\hat{q}]\!]_{\{s\}}$ is always the union of the results of the local queries that are rewritten from, and we track the mappings between the variables in the rewritten query and the variables in the original local queries. 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 local query in Q. We should identify the valid overlap between each result in $[\![\hat{q}]\!]_{\{s\}}$ and each local query in Q, and check whether a result in $[\![\hat{q}]\!]_{\{s\}}$ belongs to the relevant sources of a local query. We return to each query the result it is supposed to get.

Figure 14 illustrates 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 $\hat{q}_1$. For the first row, we find that the columns of this result only correspond to those columns of a local query $q_1^1@\{GeoNames\}$, so we distribute the row to $[\![q_1^1]\!]_{\{GeoNames\}}$. The second row in $[\![\hat{q}_1]\!]_{\{GeoNames\}}$ corresponds to the columns of both local queries $q_2^1@\{GeoNames\}$ and $q_3^1@\{GeoNames\}$, but it only meet the constraints in the FILTER operators rewritten from q_2^1. Hence, we distribute the row to $[\![q_2^1]\!]_{\{GeoNames\}}$. We iterate this process until that all results are distributed.

$[\![\hat{q}_1]\!]_{\{GeoNames\}}$

?l	?x	?c	?f
g:v3	g:v5		
g:v3		g:v1	g:School
g:v3		g:v2	g:PopulatedPlace
g:v3		g:v4	g:PopulatedPlace

$[\![q_1^1]\!]_{\{GeoNames\}}$

?l	?x
g:v3	g:v5

$[\![q_2^1]\!]_{\{GeoNames\}}$

?l	?c
g:v3	g:v1

$[\![q_3^1]\!]_{\{GeoNames\}}$

?l	?c
g:v3	g:v2
g:v3	g:v4

Figure 14: Example for Distributing Results

6. JOINING PARTIAL MATCHES

Unlike the centralized environment in [14], multiple query evaluation over federated RDF systems requires joining local partial matches. In this section, we discuss how to join partial matches efficiently.

The straightforward method to obtain results of all original queries is that we join local query matches for each original SPARQL query independently. For each local query in Q, collecting the matches at each relevant source, we obtain all its matches. Assume that an original query Q_i ($i = 1, ..., n$) is decomposed into a set of local queries $\{q_i^1@S(q_i^1), ..., q_i^{m_i}@S(q_i^{m_i})\}$. We need to obtain query result $[\![Q_i]\!]$ by joining $[\![q_i^1]\!]_{S(q_i^1)}, ..., [\![q_i^{m_i}]\!]_{S(q_i^{m_i})}$ together. In the following, for the sake of simplicity, we abbreviate $[\![q_i^{m_i}]\!]_{S(q_i^{m_i})}$ to $[\![q_i^{m_i}]\!]$.

However, considering multiple queries, there may exist some common computation in joining partial matches. For example, let us consider the local queries in Figure 8. Figure 15 shows their join graphs. We can observe that q_2^1 is isomorphic to q_3^1, and q_2^3 is isomorphic to q_3^3. Meanwhile, the join variable between q_2^1 and q_2^3 is the same to the join variable between q_3^1 and q_3^3. Hence, we can merge $[\![q_2^1]\!] \bowtie [\![q_2^3]\!]$ and $[\![q_3^1]\!] \bowtie [\![q_3^3]\!]$ into $([\![q_2^1]\!] \cup [\![q_3^1]\!]) \bowtie ([\![q_2^3]\!] \cup [\![q_3^3]\!])$.

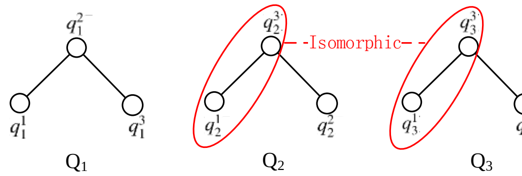


Figure 15: Example Join Graphs

Taking advantage of these common joining structures, we can speed up the query response time for multiple queries. Formally, given local queries $q_i^{i_1}$ and $q_i^{i_2}$ for query Q_i and local queries $q_j^{j_1}$ 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. Thus,

$$\begin{aligned} & cost([\![q_i^{i_1}]\!] \bowtie [\![q_i^{i_2}]\!]) + cost([\![q_j^{j_1}]\!] \bowtie [\![q_j^{j_2}]\!]) \\ & = min\{card([\![q_i^{i_1}]\!]), card([\![q_i^{i_2}]\!])\} + min\{card([\![q_j^{j_1}]\!]), card([\![q_j^{j_2}]\!])\} \end{aligned} \tag{8}$$

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 as follows.

$$\begin{aligned} & cost(([\![q_i^{i_1}]\!] \cup [\![q_j^{j_1}]\!]) \bowtie ([\![q_i^{i_2}]\!] \cup [\![q_j^{j_2}]\!])) \\ & = min\{card([\![q_i^{i_1}]\!] \cup [\![q_j^{j_1}]\!]), card([\![q_i^{i_2}]\!] \cup [\![q_j^{j_2}]\!])\} \end{aligned} \tag{9}$$

The 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.

It is easy to identify some common substructures between these join graphs. Obviously, we can do this part only once to avoid duplicate computation. We can find common substructures by using frequent subgraph mining technique [25]. Specifically, we can first find a common substructure among all join graphs, where vertices (i.e., the local queries) in the common substructure have the largest benefit. We perform the join for this common substructure; and then iterate the above process. We do not discuss this tangential issue any further.

7. EXPERIMENTAL EVALUATION

In this section, we evaluate our federated multi-query optimization method (FMQO) over both real (FedBench) and synthetic datasets (WatDiv). In addition, we compare our system with two state-of-the-art federated SPARQL query engines, such as FedX [23] and SPLENDID [8].

7.1 Setting

WatDiv. WatDiv [2] 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. Similar to the setting in [2], we generate three datasets varying sizes from 10 million to 100 millions triples. WatDiv provides its own workload generator which generates templates and instantiates some templates with actual RDF terms from the dataset. We directly use the own workload generator of WatDiv to generate different workloads and test our approaches.

FedBench. FedBench [22] is a comprehensive benchmark suite for testing and analyzing both the efficiency and effectiveness of federated RDF systems. It includes 6 real cross domain RDF datasets and 4 real life science domain RDF datasets. In this benchmark, 7 federated queries are defined for cross domain RDF datasets, and 7 federated queries are defined for life science RDF datasets. To enable multiple query evaluation, we use these 14 queries as seeds and generate different kinds of workloads in our experiment. In particular, for each benchmark query, we remove all constants (strings and URIs) at subjects and objects and replace them with variables. By doing this, we extract a general representation of a SPARQL query as a template. Then, we instantiates these templates from benchmark queries with actual RDF terms from the dataset. By default, we generate 150 queries for cross domain and life science domain RDF datasets, respectively.

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 a SPARQL endpoint as an RDF source. Each source can only communicate with the control site through HTTP requests and cannot communicate with each other.

To distribute the RDF datasets, for FedBench, we assume that each dataset is distributed over a source site. For WatDiv, we partition them into m parts located at different source sites, where parameter m varies from 4 to 16 in our experiment. The default value for m is 4. We first use METIS [12] to divide the schema graph of the collection into connected subgraphs. Then, we put each all vertices mapping to the same vertex of the schema graph into one RDF source.

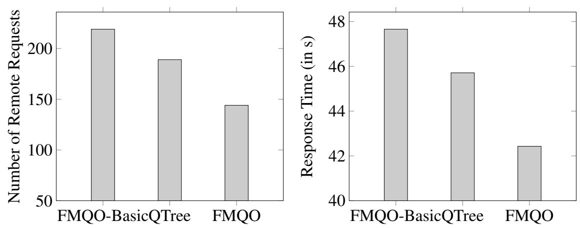
7.2 Evaluation of Proposed Techniques

In this section, we use WatDiv 10M 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 10M.

Effect of the Query Decomposition and Source Selection Technique. First, we evaluate the effectiveness of our source topology-based technique proposed in Section 4. In Figure 16, 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). Furthermore, we also compare the source selection method proposed in [9, 18], which is denoted as QTree. It only uses the neighborhood information in the source topology to prune some irrelevant sources for each triple patterns.

Obviously, FMQO-Basic does not prune any sources, so it leads to the most number of remote requests and the least 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.

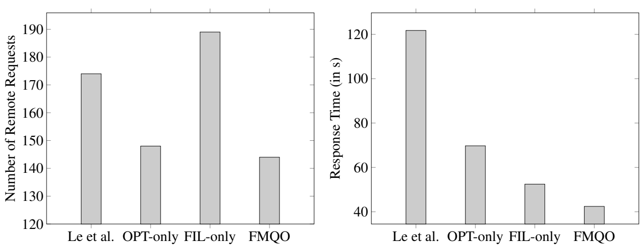
Many queries contain triple patterns having constants with high selectivity, so these triple patterns can be localized to a few sources. Then, for other triple patterns, if some of their relevant sources are far from relevant sources of the selective triple patterns in the source topology graph, they can be filtered out by our method, since our method considers the whole topology information of the federated RDF system. Thus, our method leads to the smallest numbers of remote requests (as shown in Figure 16(a)) and the least query response time (as shown in Figure 16(b)).



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

Figure 16: Evaluating Source Topology-based Source Selection Technique

Effect of the Rewriting Strategies. In this experiment, we compare our SPARQL query rewriting techniques with only using OPTIONAL operators (denoted as OPT-only) and only using FILTER operators for rewriting (denoted as FIL-only). We also reimplement the rewriting strategies proposed in [14] (denoted as Le et al.) to rewrite local queries. Our query rewiring technique is denoted as FMQO. Figure 17 shows the query response time and the number of remote requests for a workload by using the four rewriting strategies.



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

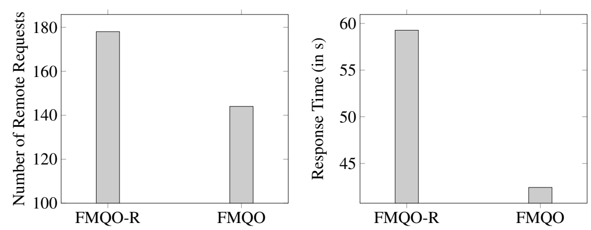
Figure 17: Evaluating Different Rewriting Strategies

Given a workload, because the number of local queries sharing common substructures is often more than the number of local queries having the same structure, FIL-only leads to the largest number of rewritten queries, meaning it results in more remote requests than other rewriting strategies. Le et al. first cluster all local queries into some groups, and then find the maximal common edge subgraphs of the group of local queries for query rewriting. Thus, the number of rewritten queries generated by Le et al. is no less than the number of the groups. In contrast, OPT-only and FMQO use some triple patterns to hit local queries. Hence, the number of rewritten queries generated by OPT-only and FMQO is the number of selected triple patterns. In real applications, most maximal common edge subgraphs found by Le et al. also contain most our selected triple patterns. Hence, Le et al. generate more rewritten queries than OPT-only and FMQO, which means more remote requests. Last, 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. A query with OPTIONAL operators is slower than a query with FILTER operators, assuming they have the same main pattern, since the former is based on left-join and the later is based on selection. Hence, although more queries are generated by using FIL-only rewriting strategy, their query response times are faster than Le et al. and OPT-only. Generally speaking, FIL-only takes about half time of Le et al., as shown in Figure 18(b) and two thirds of OPT-only. Furthermore, our proposed query rewiring take advantages of two rewriting strategies as confirmed in both real and benchmark datasets.

Effect of the Cost Model. In this section, we evaluate the effectiveness of our cost model and cost-aware rewritten strategy in Section 5.2. In Figure 18, we analyze the effect of our cost function to measure the cost of rewriting. We design a baseline (FMQO-R) that does not select the locally optimal triple patterns as presented in Algorithm 1 but randomly select triple patterns to rewrite local queries.

Compared to the baseline, we can find out that cost-based selection causes fewer remote requests, as shown in Figure 18(a). This is because that patterns with lower cost are shared by more local queries, which results fewer rewritten queries. In addition, in our cost-based rewriting strategy, we prefer to selecting selective query patterns, resulting in short query response time, as shown in Figure 18(b). Generally speaking, the cost model-based approach can speed up twice.



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

Figure 18: Evaluating Cost Model

Effect of Optimization Techniques for Joins. We evaluate our optimized join strategy proposed in Section 6. We design a baseline which 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. Generally speaking, it reduces 10% join processing time, as shown in Figure 19.

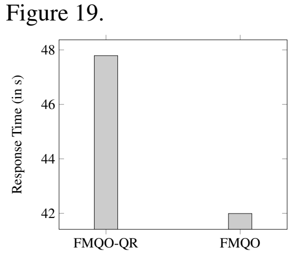
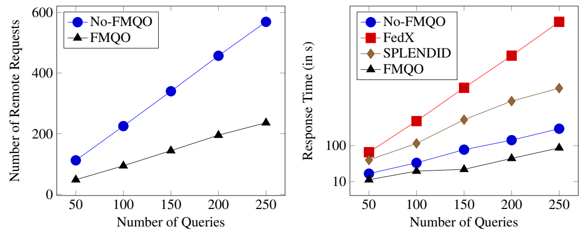


Figure 19: Effect of Optimization Techniques for Joins

7.3 Evaluating Scalability

In this subsection, 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 whole data sizes and varying the number of sources. We design a baseline that runs multiple federated queries sequentially (denoted as No-FMQO). This baseline uses the existing techniques for data localization without using the source topology graph and does not employ any optimizations for multiple queries. We also compare our method with FedX and SPLENDID. By default, the dataset is WatDiv 10M, the number of sources is 4, the number of queries is 150 and the number of templates is 10.

Varying Number of Queries. We study the impact on the number of the query set, for which we vary from 100 to 250 queries, by an increment of 50. Figure 20 shows the experimental results.



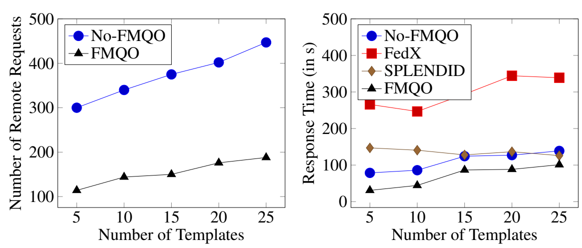
(a) Number of Remote Requests (b) Response Time

Figure 20: Varying Number of Queries

Due to query rewriting, our method (FMQO) can merge many local queries into fewer rewritten queries, which result in smaller number of remote requests, as shown in Figure 20(a). Generally speaking, FMQO can save the number of remote access by 1/2-2/3, compared with No-FMQO (evaluating multiple queries sequentially). Note that, since FedX and SPLENDID does not provides their numbers of remote requests, we do not compare FMQO with FedX and SPLENDID in Figure 20(a).

In terms of evaluation times (see Figure 20(b)), since the baseline method without any optimization (i.e., No-FMQO) does not share any replicate computation, it takes a third more times than FMQO approach in running time. Here, because both FedX and SPLENDID always employ a semijoin algorithm to join intermediate results and the semijoin algorithm is not always efficient for the synthetic dataset, No-FMQO and FMQO are also twice faster than FedX and SPLENDID.

Varying Number of Query Templates. Here, we study the impact on the number of templates. We vary the number of templates from 5 to 25, by an increment of 5. The results are shown in Figure 21. Similarly, we do not compare FMQO with FedX and SPLENDID in Figure 21(a).



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

Figure 21: Varying Number of Query Templates

Figure 21(a) shows that the number of remote requests increases as the number of templates increases. This is because, as the number of queries is kept constant, more templates mean that fewer queries have the same structures. Queries with more structures result in more number of rewritten queries. Therefore, the number of the remote requests increases by half (from about 100 to about 150). More rewritten queries mean that less computation is shared by different queries, so the performance of FMQO becomes worse as shown in Figure 21(b). However, the response time of FMQO is still 50% less than No-FMQO and SPLENDID, and two thirds less than FedX.

Varying Size of Datasets. Here, we investigate the impact of dataset size on the optimization results. We generate three WatDiv datasets varying the from 10 million to 100 million triples. Figure 22 shows the results.

While this does not affect the number of remote request, it clearly affects evaluation times, as shown in Figure 22. As the size of RDF datasets gets larger, the response time of all three methods increases. However, the rate of rise for FMQO is smaller than other competitors. The response time of FMQO decreases from 60% of No-FMQO to 50% of No-FMQO, while the response time of FMQO always is 30% of SPLENDID to 20% of FedX.

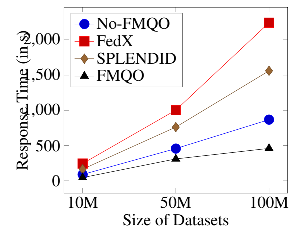
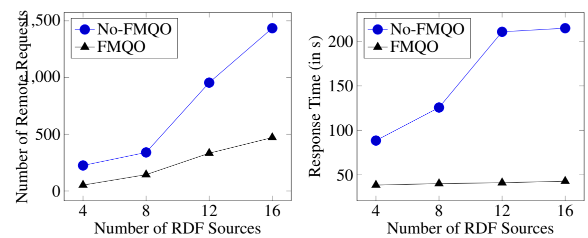


Figure 22: Varying Size of Datasets

Varying Number of Sources. In this experiment, we vary the number of sources from 4 to 16. Figure 23 presents the scalability of our solution adapting to different number of RDF sources. As the number of sources increases, a query may be relevant more sources and it is decomposed into more local queries. Thus, more rewritten queries 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. It confirms that FMQO has better scalability with the number of sources.

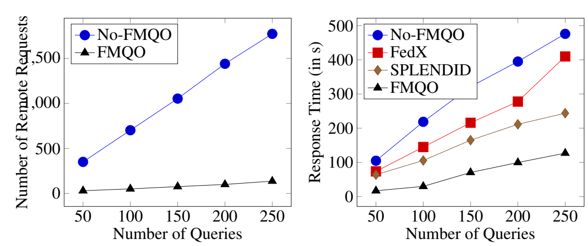


(a) Number of Remote Requests (b) Response Time

Figure 23: Varying Number of RDF Sources

7.4 Performance over Real Dataset

In this experiment, we test our FMQO method in the real RDF dataset FedBench. Since real datasets do not allow changing data sizes and the number of sources, we only test all methods by varying the number of queries in Figures 24 and 25. Experiments confirm that FMQO lead to fewest remote requests for FedBench and result in best query performance.



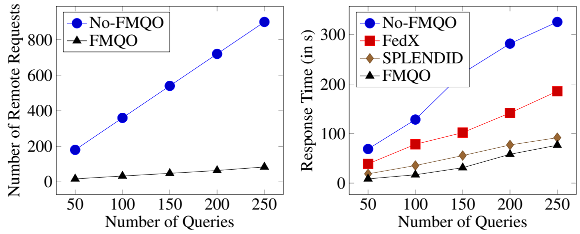
(a) Number of Remote Requests (b) Response Time

Figure 24: FedBench (Cross Domain)

8. RELATED WORK

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

Federated Query Processing. Recently, many approaches [19, 23, 8, 9, 18, 20, 21] have been proposed for federated SPARQL



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

Figure 25: FedBench (Life Science)

query processing. Since RDF sources in federated RDF systems are autonomous, they cannot be interrupted during query evaluation. Thus, the major challenges for processing queries in federated RDF systems are source selection and global query optimization. The challenges are the main differences among existing approaches.

For source selection, most papers propose the metadata-assisted methods. They find the relevant RDF sources for a query by simply matching all triple patterns based on the metadata. In particular, the metadata in DARQ [19] is named service descriptions, which describes the data available from a data source in form of capabilities. SPLENDID [8] uses Vocabulary of Interlinked Datasets (VOID) as the metadata. QTree [9, 18] 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 [20] 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 [21] 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.

Besides, there are still a few papers that do not require the metadata. FedX [23] sends ASK queries for each triple pattern to the RDF sources for source selections. Based on the results, it annotates each triple pattern with its relevant sources.

For global query optimization, the goal of this step is to find an efficient query execution plan. Most federated query engines employ existing optimizers, such as dynamic programming [4], for optimizing the join order of local queries. Furthermore, DARQ [19] and FedX [23] discuss how to use a semijoin algorithm to join intermediate results.

Multiple SPARQL Queries Optimization. Le et al. [14] 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 set 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.

There also have been a few papers on multi-query processing and optimization [16, 3] on Hadoop. HadoopSPARQL [16] discuss how to translate a set of join operators into one Hadoop job to share the computation of multiple SPARQL queries. Anyanwu et al. [3] extend the "multi starjoin" processing in multiple SPARQL queries to "multi-OPTIONAL" processing, which reduces the number of MapReduce cycles.

9. CONCLUSION

We studied the problem of multi-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 equivalent queries that are more efficient to evaluate. We also discuss how to efficiently select relevant sources and join intermediate results. Extensive experiments show that our optimizations are effective.

10. REFERENCES

[1] M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *ISWC*, pages 18–34, 2011.

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

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

[4] M. M. Astrahan, H. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1:97–137, 1976.

[5] T. Berners-Lee. Linked Data?Design Issues. *W3C*, 2010.

[6] V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, 1979.

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

[9] 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.

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

[11] K. Hose, R. Schenkel, M. Theobald, and G. Weikum. Database foundations for scalable RDF processing. In *Reasoning Web*, pages 202–249, 2011.

[12] G. Karypis and V. Kumar. Multilevel Graph Partitioning Schemes. In *ICPP*, pages 113–122, 1995.

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

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

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

[16] 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.

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

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

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *ICDM*, pages 721–724, 2002.

APPENDIX

A. HANDLING GENERAL SPARQL

So far, we only consider BGP (basic graph patterns) evaluation over federated RDF systems. In this section, we discuss how to extend our method to general SPARQLs with UNION, OPTIONAL and FILTER statements.

Queries with UNION operators. The query with UNION operators Q_1 $UNION$ Q_2 can be directly decomposed into two BGPs Q_1 and Q_2. Then, we can pass the batch of BGPs for multiple optimization. Finally, the result to the original query with UNION operators can be generated through the union of the results from the transformed BGPs after MQO.

Queries with OPTIONAL operators. To handle queries with OPTIONAL operators, it requires a preprocessing step on the input queries. Specifically, by the definition of OPTIONAL, a query with a OPTIONAL operator Q_1 $OPTIONAL$ Q_2 is rewritten into two BGPs, since our MQO algorithm only works on BGPs. The equivalent BGPs of a query Q_1 $OPTIONAL$ Q_2 are several BGPs: Q_1 and Q_1 AND Q_2. For example, after applying the above preprocessing, we can transform the query with OPTIONAL operators in Fig. 26(a) to a group of two BGPs as in Fig. 26(b).

```
SELECT ?x ?n WHERE {
?x g:parentFeature ?l .
?l g:name "Canada" .
OPTIONAL {?x g:name ?n .}}
```

(a) Query with OPTIONAL Operator

```
SELECT ?x WHERE {
?x g:parentFeature ?l .
?l g:name "Canada" .}
```

```
SELECT ?x ?n WHERE {
?x g:parentFeature ?l .
?l g:name "Canada" .
?x g:name ?n .}
```

(b) Equivalent BGPs

Figure 26: Query with OPTIONAL Operator to Its Equivalent BGPs

Queries with FILTER operators. For queries with FILTER operators, during data localization, we move possible value constraints into the local queries to reduce the size of intermediate results as early as possible. For example, for the query with FILTER operators in Fig. 27(a), it is decomposed to two local queries as in Fig. 27(b).

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

(a) Query with FILTER Operator

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

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

(b) Local Queries

Figure 27: Query with FILTER Operator to Its Local Queries

In addition, when we use FILTER-based rewriting strategy to rewrite a local query, we merge the original FILTER expressions and the rewritten FILTER expressions by using the intersection operators. For example, the local query in Fig. 27(b) can be rewritten to the query in Fig. A.

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

Figure 28: Rewritten Query with FILTER Operators