

Multi-Query Optimization in Federated RDF Systems

Peng Peng¹, Lei Zou², M. Tamer Özsu³

¹Hunan University, China;

²Peking University, China;

³University of Waterloo, Canada;

hnu16pp@hnu.edu.cn, zoulei@pku.edu.cn, tamer.ozsu@uwaterloo.ca

ABSTRACT

This paper revisits the classical problem of multiple query optimization in the context of federated RDF systems. We propose a heuristic query rewriting-based approach to share the common computation during evaluation of multiple queries while considering the cost of both query evaluation and data shipment. 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 join intermediate results during multiple query evaluation. The extensive experimental studies over both real and synthetic RDF datasets show that the proposed techniques are effective, efficient and scalable.

1. INTRODUCTION

Many data providers publish, share and interlink their 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 the actual triple files at their own *autonomous* sites some of which are *SPARQL endpoints* that can execute SPARQL queries. An autonomous site with a SPARQL endpoint is called an *RDF source* in this paper.

To integrate and provide transparent access over many RDF sources, federated RDF systems have been proposed [21, 25], 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 subqueries that are evaluated on relevant sources; then, these subquery 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 platform¹ for users to query multiple bioin-

¹<http://www.ebi.ac.uk/rd/>

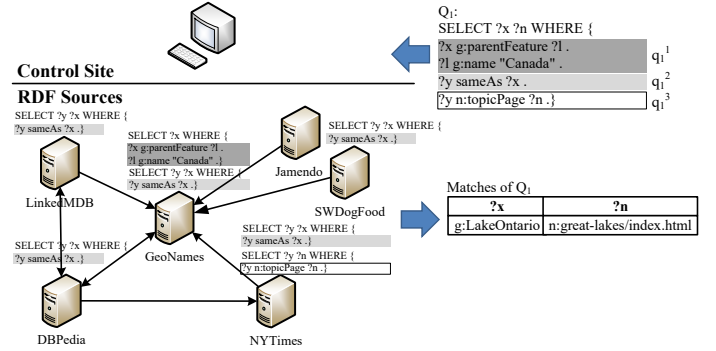


Figure 1: A Federated SPARQL Query

omatics 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 [21, 25, 2, 9, 22]. A popular federated RDF benchmark—FedBench [24]—is often used 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 all Canadian places. Fortunately, the places in NYTimes are linked to the counterparts (using “sameAs” property) in GeoNames, a worldwide geographical RDF database that states explicitly which places are located in Canada. Therefore, the federated SPARQL query Q_1 that ask for can be formulated over the federated RDF system.

To evaluate Q_1 , it is decomposed into three subqueries q_1^1 , q_1^2 and q_1^3 , which are sent to relevant sources as discussed in Section 3. These subquery results are sent back to the query originating site and joined together to form complete results. □

However, existing federated RDF systems only consider query evaluation for a single query and miss the opportunity for multiple query optimization. Real SPARQL query workloads reveal that many SPARQL queries are often posed simultaneously, so this is a real issue. Let us consider a real SPARQL query workload over

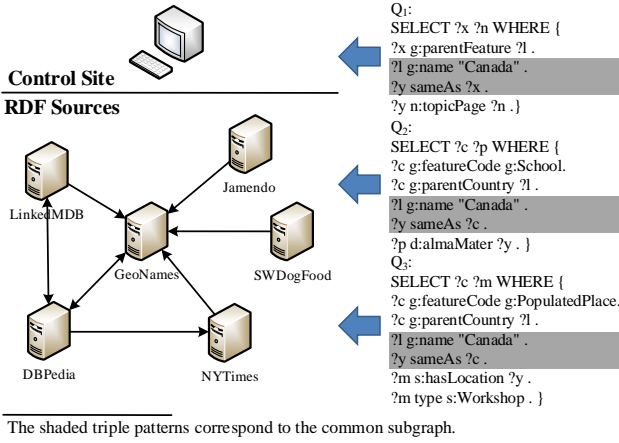


Figure 2: Multiple Federated SPARQL Queries

DBPedia² that includes 8 million SPARQL queries issued over 14 days. On average, there are more than six SPARQL queries per second. Furthermore, if we set the minimum support threshold as 0.1% of the total number of queries, we can mine 163 frequent patterns in the DBPedia workload. The most surprising is that 97% query graphs are isomorphic to one of the 163 frequent patterns. There is room for sharing computation when executing these queries. Take another example. According to the workload of SPARQL queries over Linked Geo Data³, more than 600 queries are issued per second at peak time. If we also set the minimum support threshold as 0.1% of the total number of queries, we mine 90 frequent subgraph patterns. Almost 80% queries in LinkedGeoData are isomorphic to one of the 90 frequent subgraph patterns. Therefore, it is desirable to design a multiple SPARQL query optimization strategy.

Consider a batch of queries (e.g., Q_1 , Q_2 and Q_3 in Fig. 2) that are posed simultaneously over the federated RDF system in Fig. 1. It is easy to identify some common substructures over these three queries, which suggests the possibility of some sharing computation. This motivates us to revisit the classical problem of multiple query optimization in the context of federated RDF systems. Specifically, Q_1 is introduced in Example 1 and 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. All the three queries will be used as running examples throughout this paper.

1.1 Challenges & Our Solution

Although multiple query optimization have been well studied in distributed relational databases [16], some techniques commonly referred to as data movement and data/query shipping [14] are not easily applicable in federated RDF systems. 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 [16] is also infeasible.

To the best of our knowledge, only one proposal about multiple SPARQL query optimization exists in literature [15], but *only in the centralized environment*, where all RDF datasets are collected in one physical database. Given a batch of SPARQL queries, it identifies all maximal common edge subgraphs (MCES) among query graphs. Each MCES leads to a possible MCES-based rewritten query. A cost model is proposed to select one MCES-based

rewriting with the minimum cost. Since MCES identification algorithm has exponential time complexity, the approach is not scalable with respect to the size and the number of query graphs. Furthermore, there is no data shipment in the centralized environment. Rewriting multiple queries into some rewritten queries as [15] can generate many intermediate results in federated RDF systems and result in large cost for data shipment.

Our method has two novel characteristics:

1. As we rewrite multiple queries with commonalities, we consider both “OPTIONAL” and “FILTER” clauses of SPARQL. This allows us significant rewrite opportunities.
2. We propose a cost model-driven greedy solution for multiple query rewriting; our cost model considers the cost for both query evaluation and data shipment, while our solution considers both the common substructures and the selectivity of queries. Also, the linear time complexity of our greedy algorithm guarantees the scalability of our rewriting strategy with regard to the sizes and numbers of queries that are posed simultaneously.

In addition, we also study relevant source selection and partial match joins in federated RDF systems, which do not arise in the centralized counterpart.

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. 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 30% compared with the existing data localization techniques. Second, we study how to optimize partial match joins during multiple query processing in federated RDF systems. Experiments show that the optimized join approach can improve query performance by 10% 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, with the objective to reduce the query response time and the number of remote requests.

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, G is a combination of many RDF graphs located at different source sites.

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

²<http://aksw.org/Projects/DBPSB.html>

³<http://aksw.github.io/LSQ/>

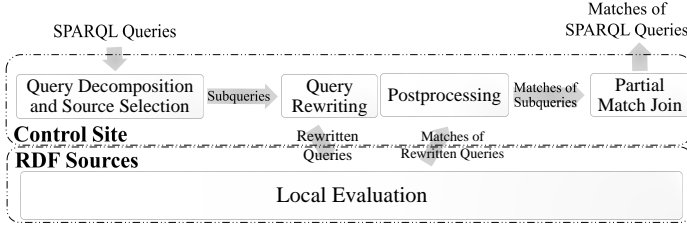


Figure 4: Scheme for Federated SPARQL Query Processing

3. FRAMEWORK

The federated RDF system consists of a control site as well as some RDF sources. The control site is amenable to receive a SPARQL query Q and decompose it into several subqueries that are sent to relevant sources. Assume that several subqueries that are sent to the same source share common substructures. In order to improve the query performance, the control site will rewrite them as a single SPARQL that is sent to the relevant source. Results of decomposed subqueries are returned to the control site and joined together to form complete results that are presented to users.

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

Query Decomposition and Source Selection. We first decompose a query Q into a set of subqueries expressed over relevant sources. Most existing solutions [21, 9, 10, 20, 22] select relevant sources based on properties of triple patterns. Thus, they often overestimate the set of relevant sources. In this paper, we propose to utilize the interconnection structure between different RDF sources to further filter out irrelevant sources. The query decomposition and source selection are not the focus of our paper. We utilize existing query decomposition and source selection techniques in this work.

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

Query Rewriting. After the first step, we have multiple subqueries. 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 FILTER and OPTIONAL operators to rewrite these subqueries as a single query that will be sent to the relevant source. Our query rewriting technique is based on a cost model that reduces both the response time and the number of remote accesses while considering the time for both local evaluation and data shipment.

Assume that source s is assigned a set of subqueries $Q_s = \{q_s^1, \dots, q_s^n\}$. After query rewriting, we obtain a set of rewritten queries \hat{Q}_s ($|\hat{Q}_s| \leq |Q_s|$) that will be sent to source s . Each rewritten query comes from a subset of Q_s . We will discuss this in Section 5.

Local Evaluation. After query rewriting, we rewrite subqueries into a set of rewritten queries \hat{Q}_s . We send the set of rewritten queries to their relevant sources and evaluate them there. Local evaluation results will be return back to the control site.

Postprocessing. The local evaluation results from evaluating \hat{Q}_s over the RDF sources are a superset of evaluating the original sub-

queries Q_s . Therefore, the control site necessitates a postprocessing step to check each local evaluation result against each query in Q_s . In this paper, we propose a postprocessing method which only requires a linear scan on the local evaluation results of the rewritten queries. We will discuss the details in Section 6.

Partial Match Join. For each subquery q_i^j in 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 ($i = 1, \dots, n$) is decomposed into a set of subqueries $\{q_i^1 @ S(q_i^1), \dots, q_i^{m_i} @ S(q_i^{m_i})\}$, we obtain query results $\llbracket Q_i \rrbracket_S$ by joining $\llbracket q_i^1 \rrbracket_{S(q_i^1)}, \dots, \llbracket q_i^{m_i} \rrbracket_{S(q_i^{m_i})}$. Considering the context of multiple SPARQL queries over a federated RDF system, we propose an optimized solution to avoid duplicate computation in join processing (in Section 7).

4. QUERY DECOMPOSITION AND SOURCE SELECTION

Given a SPARQL query, each triple pattern corresponds to a subquery. Each triple pattern maps to a set of relevant sources based on the values of its subject, property and object [25]. Consider triple pattern “?y sameAs ?x” in Q_1 of Figure 1. Since RDF sources “GeoNames”, “NYTimes”, “DBPedia”, “SWDogFood” 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”.

Since one-triple patterns can be combined together to form a larger subquery if a source is exclusively selected for a set of connected triple patterns, they can be combined to form a larger subquery. 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 to form a larger subquery q_1^1 . Note that, if a group of triple patterns shares exactly the same set of multiple RDF sources, they cannot be combined together. This is because that combining them together may miss some matches crossing different RDF sources.

For query Q_1 , the baseline solution decomposes it into three subqueries q_1^1 , q_1^2 and q_1^3 , as shown in Figure 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”.

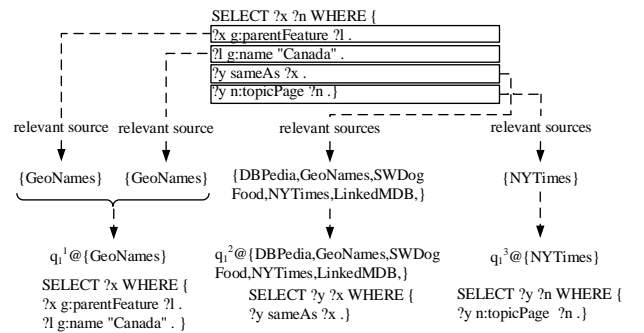


Figure 5: Basic Query Decomposition and Source Selection Result for Q_1

It is important to properly estimate the set of relevant sources, since overestimating will cause unnecessary joins that do not produce results. In this paper, we employ the interconnection structures among 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* (maintained in the control site) as follows:

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, where (1) each vertex in $V(T)$ corresponds to a source $s_i \in S$; (2) there is an edge between vertices s_i and s_j in T , if and only if there is at least one edge $\vec{u_i u_j} \in g(s_i)$ (or $\vec{u_j u_i} \in g(s_i)$ or $\vec{u_i u_j} \in g(s_j)$ or $\vec{u_j u_i} \in g(s_j)$), 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 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 . Figure 6(a) shows the annotated STG T^* for query Q_1 . Then, each query results in an annotated source topology graph. Note that, the number of RDF sources in a real federated RDF system is limited, so the total cost of the annotated source topology graphs for all queries is also limited.

Meanwhile, for a BGP Q , we build a *join graph* (denoted as $JG(Q)$) as follows. In a join graph, one vertex indicates a subquery of Q . We introduce an edge between two vertices in the join graph if and only if the corresponding subqueries are connected in the original SPARQL query. Figure 6(b) shows the join graph $JG(Q_1)$.

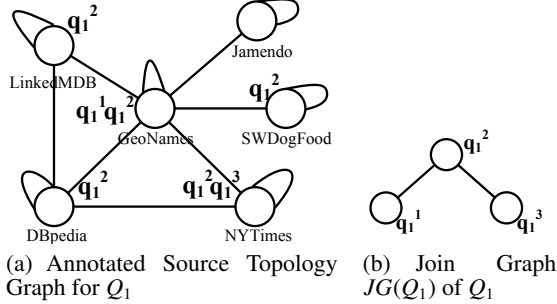


Figure 6: Example Join Graph and Annotated Source Topology Graph for Q_1

Given a join graph $JG(Q)$ and the annotated source topology graph T^* , we find all homomorphism matches of $JG(Q)$ over T^* (SPARQL semantic is based on homomorphism). If a subquery q does not map to a source s in any match, s is not a relevant source of subquery q . We formalize this observation in Theorem 1. For example, we can find three homomorphism matches of $JG(Q_1)$ over T^* in Figure 7. However, source “LinkedMDB” or “SWDogFood” do not match q_1^2 , so both of them can be pruned from the relevant sources of q_1^2 .

Match 1:	Match 2:	Match 3:
$q_1^1 \rightarrow \text{GeoNames}$	$q_1^1 \rightarrow \text{GeoNames}$	$q_1^1 \rightarrow \text{GeoNames}$
$q_1^2 \rightarrow \text{DBpedia}$	$q_1^2 \rightarrow \text{GeoNames}$	$q_1^2 \rightarrow \text{NYTimes}$
$q_1^3 \rightarrow \text{NYTimes}$	$q_1^3 \rightarrow \text{NYTimes}$	$q_1^3 \rightarrow \text{NYTimes}$

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

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

PROOF. Given a source s that is pruned by the theorem for subquery q , it means that there do not exist any homomorphism matches of q over T^* that contains s . Then, there exists another subquery q' of which relevant sources do not 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. \square

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

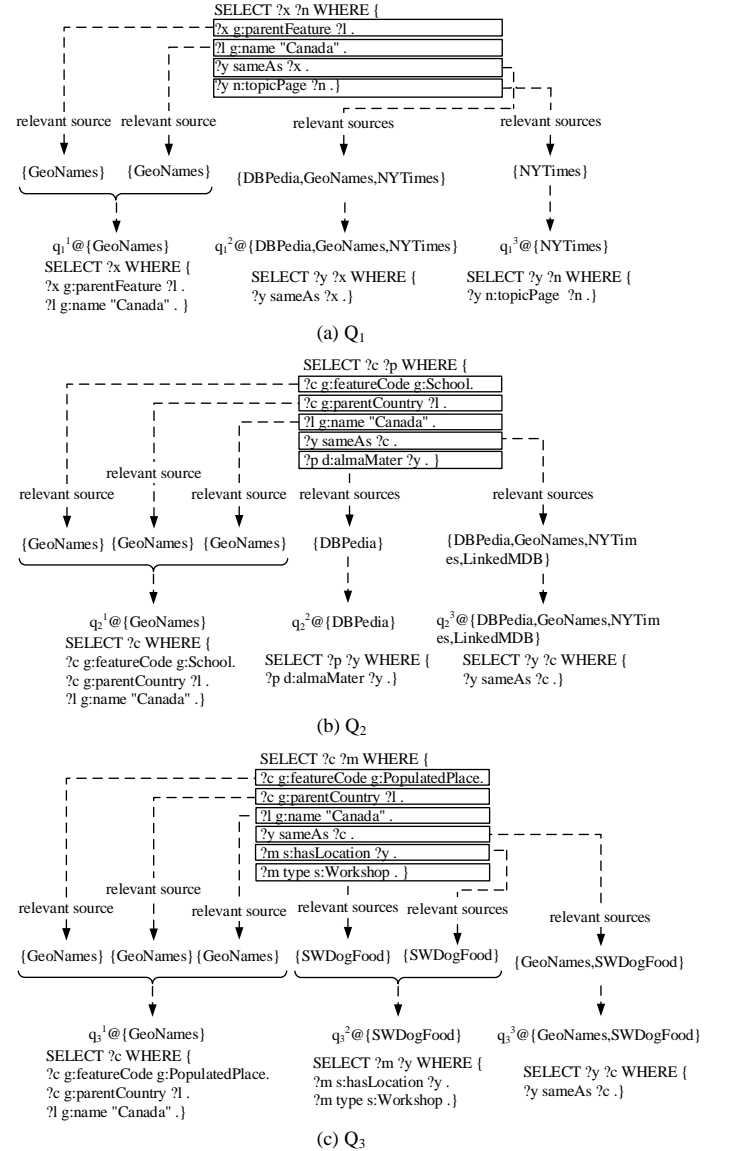


Figure 8: Optimized Query Decomposition and Source Selection Results for Q_1 , Q_1 and Q_3

5. 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 sent to the same source share some common substructures. A possible optimization is to rewrite them (at the control site) into a single SPARQL query and then send it to relevant sources, which can save both the number of remote accesses and query response time. Obviously, different query rewritings may lead to different performance; thus, this section proposes a cost-driven query rewriting scheme. To simplify presentation, we first assume that the SPARQL query originally issued at the control site is a BGP. Our solution is easily extended to handle general SPARQL queries, which is discussed in Section 7.

5.1 Intuition

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

5.1.1 OPTIONAL-based Rewriting

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

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

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

Let us consider two subqueries over GeoNames, $q_1^1@{\text{GeoNames}}$ and $q_2^1@{\text{GeoNames}}$, as shown in Fig. 9. The two subqueries 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. Thus, the cardinality of the rewritten query does not increase much. We also introduce a cost model to measure the benefit of the rewriting rule in Section 5.2.

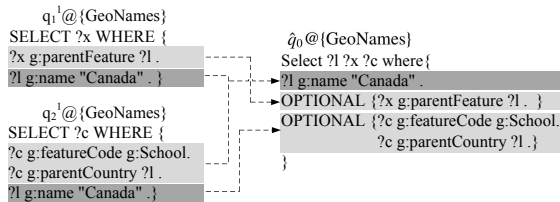


Figure 9: Rewriting Queries using OPTIONAL Operators

Although the intermediate results increase little, using OPTIONAL operators adds at least one extra left-joins for each optional clause. This results in larger local evaluation cost. Thus, this need a trade off between many queries being rewritten together and many

optional clauses. We will consider the number of OPTIONALs into the cost model as discussed in Section 5.2.

5.1.2 FILTER-based Rewriting

Let us consider two subqueries $q_2^1@{\text{GeoNames}}$ and $q_3^1@{\text{GeoNames}}$ in Fig. 10, which are decomposed from Q_2 and Q_3 . Although they distinguish from each other at the first triple pattern, the only difference is constant bounded to objects in the first triple pattern. We can rewrite the two queries using FILTER, as shown in Fig. 10. In other words, if some subqueries issued at the same source have the common structure except the constants on some vertices (subject or object positions), they can be rewritten as a single query with FILTER.

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

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

where f_i is a bijective 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.

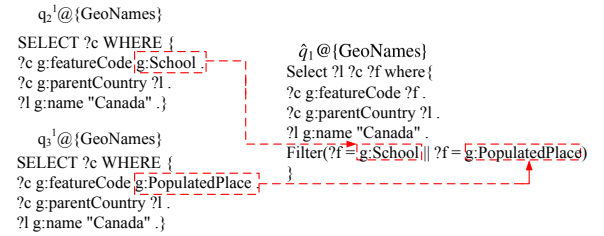


Figure 10: Rewriting Queries using FILTER Operator

In addition, unlike OPTIONAL operators adding extra columns, FILTER operators do not introduce extra joins and only generate a little more partial matches. Hence, the cost of data shipment will increase little.

5.1.3 Hybrid Rewriting

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

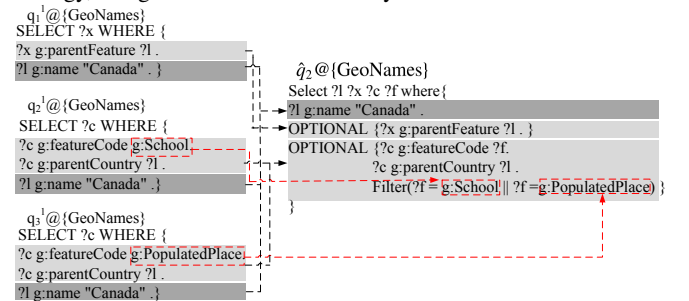


Figure 11: Rewriting Queries using OPTIONAL and FILTER Operators

5.2 Cost Model

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

5.2.1 Cost Model for BGPs

As mentioned in [15], selective triple patterns in BGP have higher priorities in evaluation. We verify the principle in two real and synthetic RDF repositories, DBpedia and WatDiv. For DBpedia, we download real SPARQL workload that records more than 8 millions SPARQL queries issued in 14 days of 2012⁴ and randomly sample 10000 queries to verify the principle; for WatDiv, to verify the principle, we generate 12500 queries from 125 templates provided in [3]. Given a triple pattern e , its selectivity is defined as $sel(e) = \frac{||e||}{|E(G)|}$, where $||e||$ denotes the number of matches of e and $|E(G)|$ denotes the number of edges in RDF graph G . The experimental results are shown in Fig. 12. As the experimental results show, the cardinality of a query is positively associated with the selectivity of the most selective triple pattern for both real and synthetic datasets.

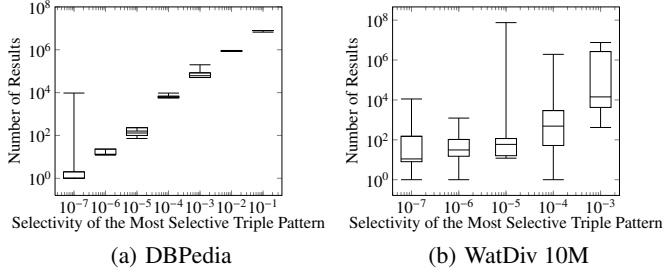


Figure 12: Experiment Results of the Relationship Between the Cardinality and the Most Selective Triple Pattern

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

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

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

As mentioned before, all results of queries are sent to the control site to join for the final results after local evaluation. Hence, the time for data shipment can be defined based on the number of results. Given a BGP Q , its cost of data shipment is defined 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.

Meanwhile, the time for evaluating a query is proportional to the size of the results. Hence, given a BGP Q , its cost of local evaluation 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 result.

In real applications, for estimating the selectivity of triple pattern, we can employ the heuristics introduced by [26] that can estimate the selectivity without pre-computed statistics about the RDF source. Simply speaking, the selectivity of a triple pattern is computed according to the type and number of unbound components

⁴<http://aksw.org/Projects/DBPSB.html>

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 [26]. Furthermore, many RDF sources also provide VOID (Vocabulary of Interlinked Datasets)⁵ that expresses metadata about RDF datasets and can also be utilized to implement the heuristic with pre-computed statistics in [26] to improve the estimation accuracy.

The relative coefficients, T_{CPU} and T_{MSG} , in the cost model are greatly influenced by the resources of each RDF sources and the network topology of the federated RDF system. They can be estimated offline as the metadata.

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 patterns in OPTIONAL clauses and the expressions in FILTER operators are evaluated on the results of the main pattern (for the fact that the graph pattern in the OPTIONAL clause is a left-join and the FILTER operator is selection) [15]. 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 result from evaluating the main graph pattern is not well indexed, which means that more efforts are needed for processing the rewriting graph patterns in the OPTIONAL clause and FILTER expressions if a main graph pattern is not selective.

According to the above reasons, we assume that the OPTIONAL clauses and the FILTER expressions are evaluated on the results of the main pattern. Hence, given a SPARQL Q , its cardinality $card(Q)$ is defined as follows.

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

where Δ_1 and Δ_2 are empirically trivial values [15].

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

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

Here, because Δ_1 and Δ_2 are trivial values and $card(\hat{q})$ is mostly credited to the cardinality of p , we ignore the trivial variables Δ_1 and Δ_2 while assuming that the term $cost(p)$ is much larger. Hence, we have the following cost function of data shipment.

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

On the other hand, since the OPTIONAL and FILTER operators are evaluated on the result of main pattern on popular RDF stores, the time for local evaluation is also based on the cardinality of the main pattern.

Furthermore, since each OPTIONAL operator adds simply a left-join with the results of the main pattern on popular RDF stores, the time for local evaluation of multiple OPTIONAL operators is approximately equal to the time of multiple execution of main pattern.

⁵<http://www.w3.org/TR/void/>

We verify the above principle by using WatDiv 10M on four popular RDF stores, Jena 2.13.0⁶, Sesame 2.7⁷, Virtuoso 7.2⁸ and g-Store 0.4.0⁹. We generate some queries with multiple OPTIONAL operators by using the query rewriting algorithm proposed in Section 5.3. The results in Fig. 13 show that the query response time of a query with multiple OPTIONALs is proportional to the number of OPTIONALs on all RDF stores.

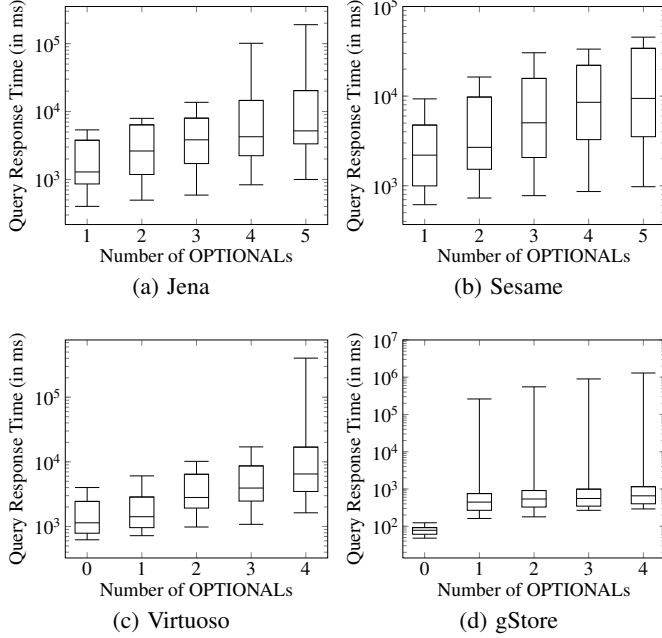


Figure 13: Experiment Results of the Relationship Between the Local Evaluation Cost and the Number of OPTIONALs

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

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

where $|OPT_{Q_2}|$ is the number of OPTIONALs in Q_2 , and Δ_3 and Δ_4 are also empirically trivial values. Here, the definition of the local evaluation cost of a query containing OPTIONAL operators is recursive, which means that the local evaluation cost of a query is proportional to the number of OPTIONAL operators.

Thus, given a rewritten query \hat{q} . The local evaluation cost of evaluating \hat{q} is defined as follows.

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

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

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

5.2.3 Total Rewriting Cost

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

$$\begin{aligned} cost(\hat{q}) &= cost_{LE}(\hat{q}) + cost_{DS}(\hat{q}) \\ &= \min_{e \in p} \{sel(e)\} \times (|OPT_{\hat{q}}| \times T_{CPU} + T_{MSG}) \end{aligned} \quad (3)$$

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

Obviously, given a set of subqueries 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 subqueries $Q = \{q_1@s, q_2@s, \dots, q_n@s\}$ on a source s using OPTIONAL and FILTER, if p is their common subgraph among q_1, \dots, 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:

$$cost_{LE}(Q, \hat{q}) = cost_{LE}(\hat{q}) = \min_{e \in p} \{sel(e)\} \times (|OPT_{\hat{q}}| \times T_{CPU} + T_{MSG}) \quad (4)$$

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

5.3 Query Rewriting Algorithm

The problem of query rewriting is that given a set Q of subqueries $\{q_1, \dots, q_n\}$, we compute a set \hat{Q} of rewritten queries $\{\hat{q}_1, \dots, \hat{q}_m\}$ ($m \leq n$) with the smallest cost (The cost function is defined in Equation 4). Note that each rewritten query \hat{q}_i ($i = 1, \dots, m$) comes from rewriting a set of original subqueries in Q , where these subqueries share the same main pattern p_i .

The cost of the whole rewriting is formatted as follows:

$$cost(Q, \hat{Q}) = \sum_{\hat{q} \in \hat{Q}} cost(\hat{q}) \quad (5)$$

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

$$cost(Q, \hat{Q}) = \sum_{p \in P} \min_{e \in p} \{sel(e)\} \times (|OPT_{\hat{q}}| \times T_{CPU} + T_{MSG}) \quad (6)$$

where $|OPT_{\hat{q}}|$ is the number of OPTIONAL operators that the rewritten query constructed by p contains.

Given a set of original queries Q , Equation 6 is a set-function with respect to set P , i.e., a set of main patterns. Unfortunately, finding the optimal rewriting is a NP-complete problem as discussed in the following theorem.

THEOREM 2. Given a set of subqueries Q , finding an optimal rewriting \hat{Q} to minimize the cost function in Equation 6 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 $U = \{1, 2, \dots, m\}$ (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

⁶<http://jena.apache.org/>

⁷<http://rdf4j.org/>

⁸<https://virtuoso.openlinksw.com/rdf-quad-store/>

⁹<https://github.com/Caesar11/gStore>

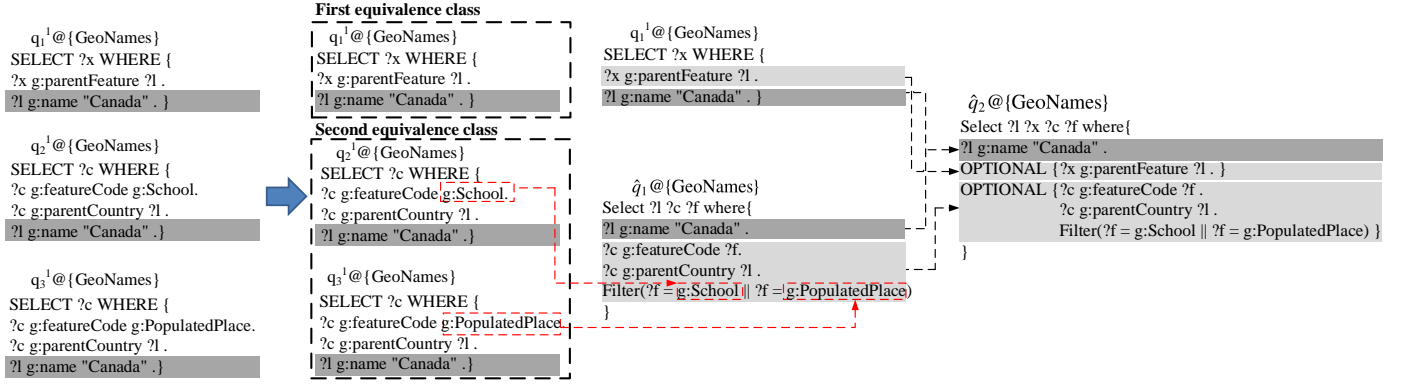


Figure 14: Example of Rewriting Subqueries

and each element in the universe U to a subquery. A set s in S containing an element e in U maps to the pattern corresponding to s hitting the subquery corresponding to e . The weight of a set s in S is the cost of its corresponding pattern.

Hence, finding the smallest weight collection of sets from S whose union covers all elements in U is equivalent to the problem of selecting the optimal set of patterns. Since the weighted set cover problem is NP-complete [8], 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 in Algorithm 1. Algorithm 1 is a greedy algorithm. Let Q denote all original subqueries. At each iteration, we select a triple pattern e_{max} with the largest value $\frac{|Q'|}{sel(e_{max}) \times (OPT_q \times T_{CPU} + T_{MSG})}$, where Q' denote all subqueries hit by e_{max} , i.e., these queries containing e_{max} . We propose a hybrid strategy to rewrite all queries in Q' . We divide Q' into several equivalence classes, where each class contains subqueries with the same structure except for some constants on subject or object positions. Subqueries in the same equivalence class can be rewritten to a query pattern with FILTER operators as discussed in 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 subqueries $q_1^1 @ \{GeoNames\}$, $q_2^1 @ \{GeoNames\}$ and $q_3^1 @ \{GeoNames\}$ in Fig. 14, 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 $\{q_1^1\}$, $\{q_2^1, q_3^1\}$ 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 subqueries in Q and $cost_{opt}$ denotes the smallest cost of patterns that hit all subqueries.*

PROOF. According to Equation 6, selecting patterns to hit subqueries is equivalent to selecting triple patterns to hit subqueries. Thus, although we only select the most beneficial triple pattern in Algorithm 1 (Line 2), it is equivalent to selecting the most beneficial pattern graph to hit subqueries. A result in [7] shows that the approximation ratio of the greedy algorithm to the optimal solution of the weighted set-cover problem is $(1 + \ln |\cup_{q \in Q} E(q)|)$. □

Algorithm 1: Query Rewriting Algorithm

Input: A set of subqueries 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
    $\frac{|Q'|}{sel(e_{max}) \times (OPT_q \times T_{CPU} + T_{MSG})}$ , where  $Q'$  is the set of subqueries
   hit by  $e_{max}$ ;
3   Extract the largest common pattern  $p$  of queries in  $Q'$ ;
4   Initialize 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 subqueries isomorphic to each other;
6   for each class  $C \in C$  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  $Q_{OPT}$ ;

```

6. LOCAL EVALUATION & POSTPROCESSING

A set of subqueries Q that will be sent to source s are rewritten as queries \hat{Q} and evaluated at source s . Let $\llbracket \hat{Q} \rrbracket_{\{s\}}$ denote the result set of \hat{Q} ($\in \hat{Q}$) at source s ; \hat{q} is obtained by rewriting a set of original subqueries in Q ; Thus, $\llbracket \hat{Q} \rrbracket_{\{s\}}$ is always the union of the results of the subqueries that are rewritten, and we track the mappings between the variables in the rewritten query and the variables in the original subqueries. The result of a rewritten query might have empty (null) columns corresponding to the variables from the OPTIONAL operators. Therefore, a result in $\llbracket \hat{Q} \rrbracket_{\{s\}}$ may not conform the description of every subquery in Q . We should identify the valid overlap between each result in $\llbracket \hat{Q} \rrbracket_{\{s\}}$ and each subquery in Q , and check whether a result in $\llbracket \hat{Q} \rrbracket_{\{s\}}$ belongs to the relevant sources of a subquery. We return to each query the result it is supposed to get.

To achieve the above objective, we perform an intersection between each result in $\llbracket \hat{Q} \rrbracket_{\{s\}}$ and each subquery. The algorithm distributes the corresponding part of this result to $q @ \{s\}$ as one of its query results, if the result meets the following two conditions: 1) the columns of this result corresponding to those columns of a sub-

query $q@s \in Q$ are not null; and 2) the columns of the result meet the constraints in the FILTER operators rewritten from q . This step iterates over each row and each subquery in Q . The checking on $\llbracket \hat{q} \rrbracket_{\{s\}}$ only requires a linear scan on $\llbracket \hat{q} \rrbracket_{\{s\}}$. Therefore, it can be done on-the-fly as the results of $\hat{q}@s$ are streamed out from the evaluation.

Fig. 15 illustrates how we distribute $\llbracket \hat{q}_1 \rrbracket_{\{GeoNames\}}$ 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 subquery $q_1^1@GeoNames$, so we distribute the row to $\llbracket q_1^1 \rrbracket_{\{GeoNames\}}$. The second row in $\llbracket \hat{q}_1 \rrbracket_{\{GeoNames\}}$ corresponds to the columns of both subqueries $q_2^1@GeoNames$ 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 $\llbracket q_2^1 \rrbracket_{\{GeoNames\}}$. We iterate this process until that all results are distributed.

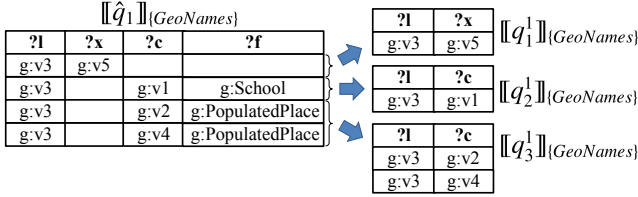


Figure 15: Example for Distributing Results

7. JOINING PARTIAL MATCHES

Query evaluation over federated RDF systems requires joining partial matches, which we discuss in this section.

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

However, considering multiple queries, there may exist some common computation in joining partial matches. For example, let us consider the subqueries in Fig. 5. Fig. 16 shows their join graphs. We can observe that q_2^1 is isomorphic to q_3^1 , and q_2^2 is isomorphic to q_3^2 . Meanwhile, the join variable between q_2^1 and q_2^2 is the same to the join variable between q_3^1 and q_3^2 . Hence, we can merge $\llbracket q_2^1 \rrbracket \bowtie \llbracket q_2^2 \rrbracket$ and $\llbracket q_3^1 \rrbracket \bowtie \llbracket q_3^2 \rrbracket$ into $(\llbracket q_2^1 \rrbracket \cup \llbracket q_3^1 \rrbracket) \bowtie (\llbracket q_2^2 \rrbracket \cup \llbracket q_3^2 \rrbracket)$.

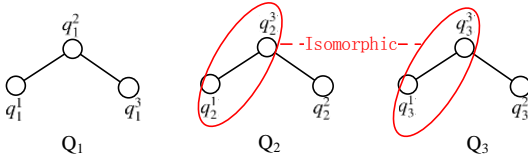


Figure 16: Example Join Graphs

Taking advantage of these common join structures, we can speed up the query response time for multiple queries. Formally, given subqueries q_i^1 and q_i^2 for query Q_i and subqueries q_j^1 and q_j^2 for query Q_j , if q_i^1 has the same structure to q_j^1 , q_i^2 has the same structure to q_j^2 and the join variables between q_i^1 and q_i^2 are the same to the join variables between q_j^1 and q_j^2 , then we can merge $\llbracket q_i^1 \rrbracket \bowtie \llbracket q_i^2 \rrbracket$ and $\llbracket q_j^1 \rrbracket \bowtie \llbracket q_j^2 \rrbracket$ into $(\llbracket q_i^1 \rrbracket \cup \llbracket q_j^1 \rrbracket) \bowtie (\llbracket q_i^2 \rrbracket \cup \llbracket q_j^2 \rrbracket)$.

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: $\llbracket q_i^1 \rrbracket \bowtie \llbracket q_j^1 \rrbracket$ and $\llbracket q_i^2 \rrbracket \bowtie \llbracket q_j^2 \rrbracket$ versus $(\llbracket q_i^1 \rrbracket \cup \llbracket q_j^1 \rrbracket) \bowtie (\llbracket q_i^2 \rrbracket \cup \llbracket q_j^2 \rrbracket)$.

The cost of executing $\llbracket q_i^1 \rrbracket \bowtie \llbracket q_j^1 \rrbracket$ and $\llbracket q_i^2 \rrbracket \bowtie \llbracket q_j^2 \rrbracket$ separately is the sum of the costs of two joins. Thus,

$$\begin{aligned} & \text{cost}(\llbracket q_i^1 \rrbracket \bowtie \llbracket q_j^1 \rrbracket) + \text{cost}(\llbracket q_i^2 \rrbracket \bowtie \llbracket q_j^2 \rrbracket) \\ &= \min\{\text{card}(\llbracket q_i^1 \rrbracket), \text{card}(\llbracket q_j^1 \rrbracket)\} + \min\{\text{card}(\llbracket q_i^2 \rrbracket), \text{card}(\llbracket q_j^2 \rrbracket)\} \end{aligned} \quad (7)$$

On the other hand, the cost of executing $(\llbracket q_i^1 \rrbracket \cup \llbracket q_j^1 \rrbracket) \bowtie (\llbracket q_i^2 \rrbracket \cup \llbracket q_j^2 \rrbracket)$ is as follows.

$$\begin{aligned} & \text{cost}((\llbracket q_i^1 \rrbracket \cup \llbracket q_j^1 \rrbracket) \bowtie (\llbracket q_i^2 \rrbracket \cup \llbracket q_j^2 \rrbracket)) \\ &= \min\{\text{card}(\llbracket q_i^1 \rrbracket \cup \llbracket q_j^1 \rrbracket), \text{card}(\llbracket q_i^2 \rrbracket \cup \llbracket q_j^2 \rrbracket)\} \end{aligned} \quad (8)$$

The our optimization technique is better if it acts as a sufficient reducer, that is, if $\llbracket q_i^1 \rrbracket$ and $\llbracket q_j^1 \rrbracket$ overlap a lot and $\llbracket q_i^2 \rrbracket$ and $\llbracket q_j^2 \rrbracket$ 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 [27]. Specifically, we can first find a common substructure among all join graphs, where vertices (i.e., the subqueries) 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.

8. 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 $Q_1 \text{ UNION } Q_2$ 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 $Q_1 \text{ OPTIONAL } Q_2$ is rewritten into two BGPs, since our MQO algorithm only works on BGPs. The equivalent BGPs of a query $Q_1 \text{ OPTIONAL } Q_2$ are several BGPs: Q_1 and $Q_1 \text{ AND } Q_2$. For example, after applying the above preprocessing, we can transform the query with OPTIONAL operators in Fig. 17(a) to a group of two BGPs as in Fig. 17(b).

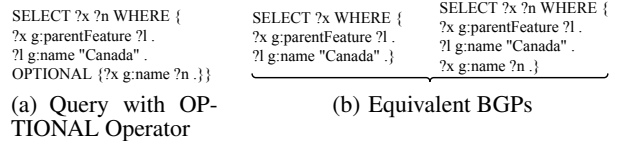


Figure 17: Query with OPTIONAL Operator to Its Equivalent BGPs

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. 18(a), it is decomposed to two local queries as in Fig. 18(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"))}
```

(b) Local Queries

Figure 18: 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. 18(b) can be rewritten to the query in Fig. 8.

```
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 19: Rewritten Query with FILTER Operators

9. EXPERIMENTAL EVALUATION

In this section, we evaluate our federated multiple query optimization method (FMQO) over both real (FedBench) and synthetic datasets (WatDiv). Table 1 shows the statistics of datasets. In addition, we compare our system with two state-of-the-art federated SPARQL query engines: FedX [25] and SPLENDID [9].

9.1 Setting

WatDiv. WatDiv [3] 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 10 million to 100 millions triples. WatDiv provides its own workload generator which generates templates and instantiates them with actual RDF terms from the dataset. We directly use WatDiv’s own workload generator of WatDiv to generate different workloads for testing.

FedBench. FedBench [24] 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 with 7 federated queries for each RDF dataset. To enable multiple query evaluation, we use these 14 queries as seeds and generate different kinds of workloads in our experiments. 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 instantiate these templates from benchmark queries with actual RDF terms from the dataset. By default, we generate 150 queries each for cross-domain and life science domain RDF datasets.

We conduct all experiments on a cluster of 10 machines running Linux, each of which has one CPU with four cores of 3.06GHz.

Collection	Dataset	Number of Triples	Domain
WatDiv	WatDiv10M	10,916,457	Synthetic
	WatDiv 50M	54,963,869	
	WatDiv100M	108,997,714	
Cross-Domain	DBPedia subset	42,855,253	Generic
	NY Times	337,563	News
	LinkedMDB	6,147,997	Movies
	Jamendo	1,049,647	Music
	GeoNames	107,950,085	Geography
	SW Dog Food	103,595	SW Conferences and Publications
Life Science	DBPedia subset	42,855,253	Generic
	KEGG	1,090,830	Chemicals
	Drugbank	766,920	Drugs
	ChEBI	7,325,744	Compounds

Table 1: Basic Statistics of Datasets

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 FedBench, we assume that each dataset is resident at a source site. For WatDiv, we partition them using METIS [12] into m parts located at different source sites, where parameter m varies from 4 to 16 in our experiment.

9.2 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 Fig. 20, 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 [10, 20], which is denoted as QTree. It only uses the neighborhood information in the source topology to prune some irrelevant sources for each triple patterns.

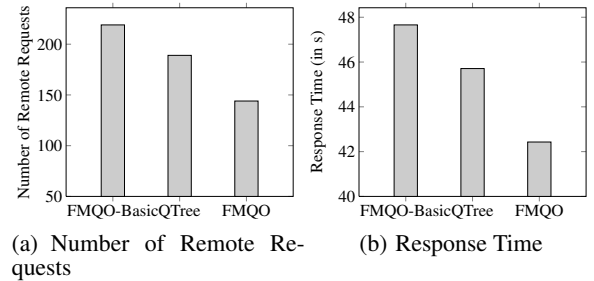


Figure 20: Evaluating Source Topology-based Source Selection Technique

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, 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 it considers the whole topology information of the federated RDF system. Thus, our method leads to the smallest numbers of remote requests (as shown in Fig. 20(a)) and the least query response time (as shown in Fig. 20(b)).

Effect of the Rewriting Strategies. In this experiment, we compare our SPARQL query rewriting techniques using only OPTIONAL operators (denoted as OPT-only) and only FILTER operators (denoted as FIL-only). We also re-implement the rewriting strategies proposed in [15] (denoted as Le et al.) to rewrite subqueries. Our query rewriting technique is denoted as FMQO. Fig. 21 shows the query response time and the number of remote requests for a workload by using the four rewriting strategies.

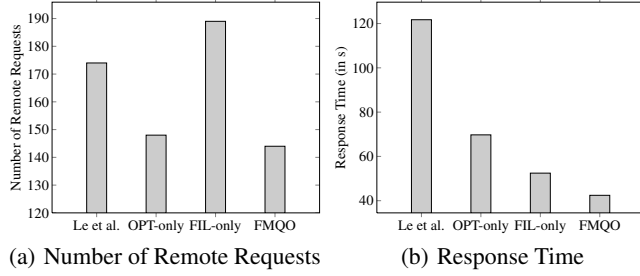


Figure 21: Evaluating Different Rewriting Strategies

Given a workload, since the number of subqueries sharing common substructures is often more than the number of subqueries 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 subqueries into some groups, and then find the maximal common edge subgraphs of the group of subqueries 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 subqueries. 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 of our selected triple patterns. Hence, Le et al. generate more rewritten queries than OPT-only and FMQO, which means more remote requests. Finally, FMQO obtains the smallest number of rewritten queries.

Since OPT-only generates smaller number of rewritten queries and share more computation than Le et al., OPT-only can result in faster query response time. 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 latter 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 Fig. 22(b) and two thirds of OPT-only. Furthermore, as shown in Fig. 22(b), our rewriting technique using both OPTIONAL and FILTER operators has the best performance. This is because it takes advantages of both the rewriting strategy only using OPTIONAL and the rewriting strategy only using FILTER.

Evaluation 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 Fig. 22, 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 subqueries.

Compared to the baseline, we find out that cost-based selection causes fewer remote requests, as shown in Fig. 22(a). 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. 22(b). Generally speaking, the cost model-based approach can provide speed up of twice.

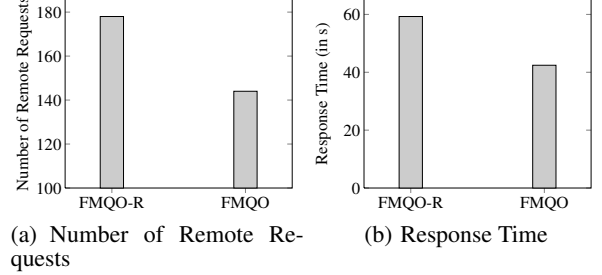


Figure 22: Evaluating Cost Model

Effect of Optimization Techniques for Joins. We evaluate our optimized join strategy proposed in Section 7. We design a baseline that runs multiple federated queries with only rewriting strategies but not our optimization techniques for joins (denoted as FMQO-QR). Although this technique does not affect the number of remote requests, it reduces the join cost by making use of common join structures. In general, it reduces join processing time by 10%, as shown in Fig. 23.

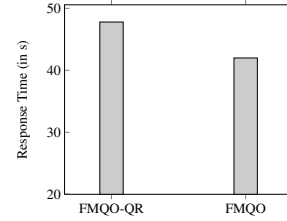


Figure 23: Effect of Optimization Techniques for Joins

9.3 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 dataset 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 of the size of the query set, which we vary from 100 to 250 queries, in increments of 50. Fig. 24 shows the experimental results.

Due to query rewriting, FMQO can merge many subqueries into fewer rewritten queries, which results in smaller number of remote requests, as shown in Fig. 24(a). FMQO can reduce the number of remote accesses by 1/2-2/3, compared with No-FMQO. Since

FedX and SPLENDID do not provide their numbers of remote requests, we do not compare FMQO with FedX and SPLENDID in Fig. 24(a).

In terms of evaluation times (Fig. 24(b)), since the baseline method without any optimization, No-FMQO, does not share any replicate computation, it takes a third more time than FMQO approach. In addition, FedX and SPLENDID always employ a semi-join algorithm to join intermediate results. Since almost all partial matches of subqueries participate in the join in WatDiv, the semi-join algorithm is not always efficient for this dataset. Hence, No-FMQO and FMQO are also twice faster than FedX and SPLENDID.

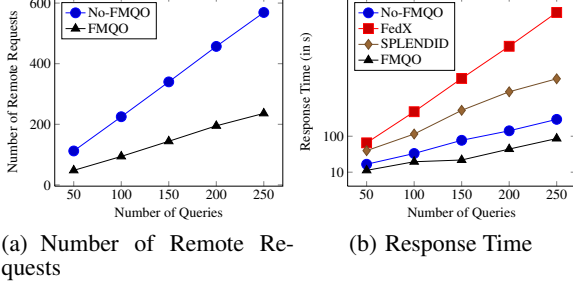


Figure 24: Varying Number of Queries

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. 25. As before, we do not compare FMQO with FedX and SPLENDID in Fig. 25(a).

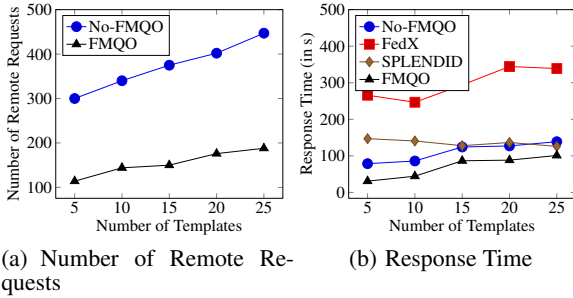


Figure 25: Varying Number of Query Templates

Fig. 25(a) shows that the number of remote requests increases with the number of templates. This is because, as the number of queries is kept constant, more templates mean that fewer queries have the common substructures. Since our rewriting techniques use the common substructures to rewrite queries, fewer queries containing the common substructures result in more number of rewritten queries. Therefore, the number of the remote requests increases by 50% (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 Fig. 25(b). However, the response time of FMQO increases slowly, so it 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. Fig. 26 shows the results.

While this does not affect the number of remote requests, it clearly affects evaluation times, as shown in Fig. 26. As the size of RDF datasets gets larger, the response time of all three methods increases. However, the rate of increase 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.

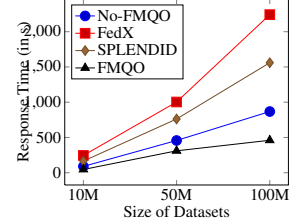


Figure 26: Varying Size of Datasets

Varying Number of Sources. In this experiment, we vary the number of sources from 4 to 16. Fig. 27 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 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.

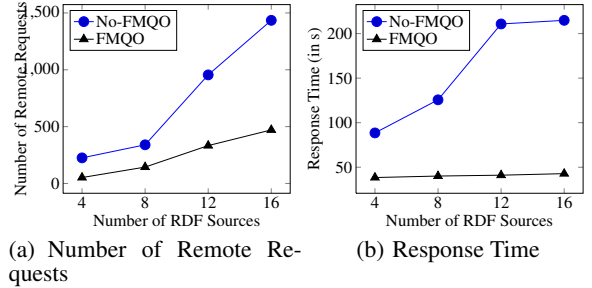


Figure 27: Varying Number of RDF Sources

9.4 Performance over Real Dataset

In this experiment, we test FMQO using the real RDF dataset, FedBench. 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 Figs 28 and 29. FMQO Experiments confirm that FMQO lead to fewest remote requests for FedBench and best query performance, since FMQO often rewrite multiple queries into a single SPARQL, which results in fewer remote requests. Furthermore, the cost-driven rewriting strategy in our method guarantees that rewritten queries are always faster than evaluating them sequentially.

10. RELATED WORK

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

Federated Query Processing. Many approaches [21, 25, 9, 10, 20, 22] have been proposed for federated SPARQL query processing. Since RDF sources in federated RDF systems are autonomous,

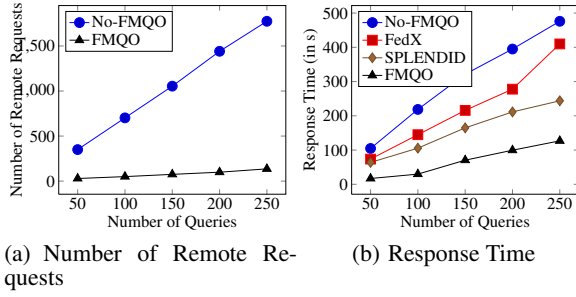


Figure 28: FedBench (Cross Domain)

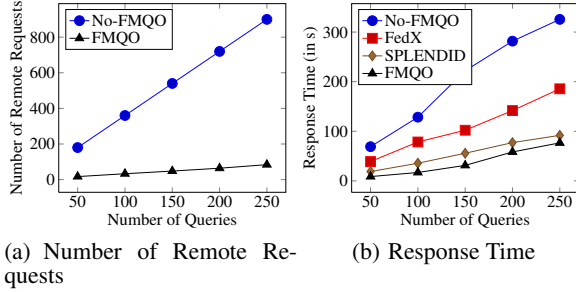


Figure 29: FedBench (Life Science)

they cannot be interrupted during query execution. Therefore, the major challenge is query decomposition and source selection. The challenge is the main differences among existing approaches.

For data localization, 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 [21] is named service descriptions, which describes the data available from a data source in form of capabilities. SPLENDID [9] uses Vocabulary of Interlinked Datasets (VOID) as the metadata. QTree [10, 20] is another kind of metadata. It is a variant of RTree, and its leaf stores a set of source identifiers, including one for each source of a triple approximated by the node. HiBISCuS [22] 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 [23] 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 [18] extends HiBISCuS by adding the indices named unique predicate types about all distinct predicates at each RDF source. A unique predicate is a property that is not involved in some special kinds of joins. SemaGrow [6] proposes a query optimizer which can generate the nearly optimal query plans in the absence of metadata with little overhead. In addition, SemaGrow also exploits non-blocking and asynchronous stream processing technologies to improve the query execution efficiency. SHEPHERD [1] reduces the communication cost and generate query plans where costly operators are placed at the RDF sources. ANAPSID [2] is a federated query engine which can dynamically generate the query plan while considering data availability and run-time conditions. Moreover, ANAPSID implement a memory-based caching mechanism to maintain previously computed matches.

Besides the metadata-assisted methods, there are still a few papers that do not require the metadata. In FedX [25], the source selection is performed by using ASK queries. FedX sends ASK

queries for each triple pattern to the RDF sources. Based on the results, It annotates each pattern in the query with its relevant sources.

Multiple SPARQL Queries Optimization. Le et al. [15] 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. Konstantinidis et al. [13] discuss how to optimize multiple SPARQL queries evaluation over multiple views. They first find out some atomic join operations among multiple queries. Then, they map each atomic join operation to a view one and compute it just once to avoid redundant work.

There also have been a few papers on multi-query processing and optimization [17, 4] on Hadoop. HadoopSPARQL [17] discuss how to translate a set of join operators into one Hadoop job to share the computation of multiple SPARQL queries. Anyanwu et al. [4] extend the “multi starjoin” processing in multiple SPARQL queries to “multi-OPTIONAL” processing, which reduces the number of MapReduce cycles.

11. CONCLUSION

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 equivalent queries that are more efficient to evaluate. We also discuss how to efficiently join intermediate results. Extensive experiments show that our optimizations are effective.

12. REFERENCES

- [1] M. Acosta, M. Vidal, F. Flöck, S. Castillo, C. B. Aranda, and A. Harth. SHEPHERD: A Shipping-Based Query Processor to Enhance SPARQL Endpoint Performance. In *ISWC*, pages 453–456, 2014.
- [2] 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.
- [3] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified Stress Testing of RDF Data Management Systems. In *ISWC*, pages 197–212, 2014.
- [4] K. Anyanwu. A Vision for SPARQL Multi-Query Optimization on MapReduce. In *Workshops of ICDE*, pages 25–26, 2013.
- [5] T. Berners-Lee. Linked Data? Design Issues. *W3C*, 2010.
- [6] A. Charalambidis, A. Troumpoukis, and S. Konstantopoulos. SemaGrow: Optimizing Federated SPARQL Queries. In *SEMANTICS*, pages 121–128, 2015.
- [7] V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of operations research*, 4(3):233–235, 1979.
- [8] 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.
- [9] O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *COLD*, 2011.
- [10] 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.

- [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] G. Konstantinidis and J. L. Ambite. Optimizing query rewriting for multiple queries. In *IIWeb*, pages 7:1–7:6, 2012.
- [14] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.
- [15] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable Multi-query Optimization for SPARQL. In *ICDE*, pages 666–677, 2012.
- [16] J. Li, A. Deshpande, and S. Khuller. Minimizing Communication Cost in Distributed Multi-query Processing. In *ICDE*, pages 772–783, 2009.
- [17] 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.
- [18] 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.
- [19] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.
- [20] F. Prasser, A. Kemper, and K. A. Kuhn. Efficient Distributed Query Processing for Autonomous RDF Databases. In *EDBT*, pages 372–383, 2012.
- [21] B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *ESWC*, pages 524–538, 2008.
- [22] M. Saleem and A. N. Ngomo. HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation. In *ESWC*, pages 176–191, 2014.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *ICDM*, pages 721–724, 2002.