# Multi-Query Optimization in Linked Data

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 Linked Data. Since RDF sources in Linked Data are autonomous and integrated as a federated database systems, existing distributed multi-query optimization techniques that rely on query plan synthesizing or data movement cannot be utilized. Thus, 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 of Linked Data 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.**

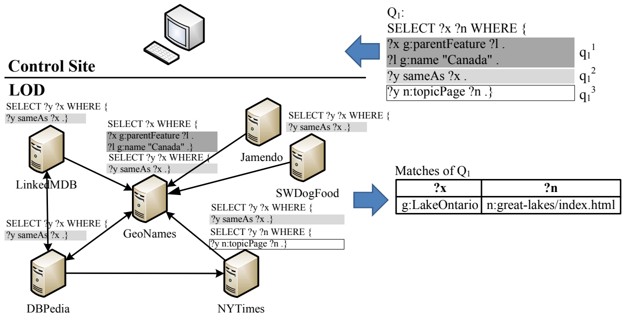
## I. Introduction

To connect distributed data sets across the Web, Linked Data principles have been formulated to publish, share and interlink structured data using open standards such as RDF and SPARQL [5]. RDF is 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.

Linked Open Data (LOD), which is the focus of this paper, uses RDF to link web objects through their Uniform Resource Identifiers. LOD forms a federated (or virtually integrated) database with the RDF triple files stored at *autonomous* sites some of which have *SPARQL endpoints* to enable applications to query RDF knowledge bases via the SPARQL language. An autonomous site with a SPARQL endpoint is called an *RDF source*.

In the context of LOD, there is an enormous need for querying over multiple distributed autonomous RDF sources[22], [26], [1], [9], [24], [23], since a SPARQL query cannot be answered using triples at a single source (See Example 1 below). In this case, 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.

*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 LOD.



```
Q1:
SELECT ?x ?n WHERE {
?x g:parentFeature ?l .
?l g:name "Canada" .
?y sameAs ?x .
?y n:topicPage ?n .}
```

```
SELECT ?y ?x WHERE {
?y sameAs ?x .}
```

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

```
SELECT ?y ?x WHERE {
?y sameAs ?x .}
SELECT ?y ?n WHERE {
?y n:topicPage ?n .}
```

Matches of $Q_1$

| ?x | ?n |
|---|---|
| g:LakeOntario | n:great-lakes/index.html |

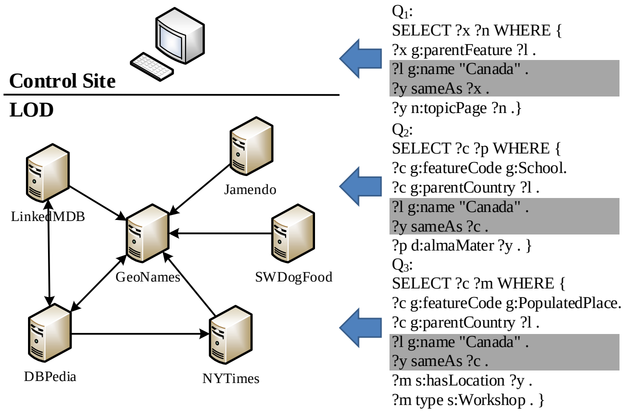
Fig. 1. A Federated SPARQL Query

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 IV. These local query results are sent back to the query originating site and joined together to form complete results. □

As shown in the above example, the spirit of LOD is interconnecting multiple sources and answering federated queries that cannot be evaluated at a single source. Existing federated SPARQL query engines, such as classical FedX [26], DARQ [22], and recent SPLENDID [9], HiBISCuS [23], only consider query evaluation for a single SPARQL and miss the opportunity for multiple federated query optimization.

*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. □

Consider a batch of queries (e.g., $Q_1$, $Q_2$ and $Q_3$) that are posed simultaneously over LOD. Example 2 introduces two additional queries over the same interconnected RDF sources. 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 LOD. To the best of our knowledge, no prior work studies optimizations for multiple SPARQL queries over LOD. This is the motivation of our work.



```
Q1:
SELECT ?x ?n WHERE {
?x g:parentFeature ?l .
?l g:name "Canada" .
?y sameAs ?x .
?y n:topicPage ?n .}
Q2:
SELECT ?c ?p WHERE {
?c g:featureCode g:School.
?c g:parentCountry ?l .
?l g:name "Canada" .
?y sameAs ?c .
?p d:almaMater ?y . }
Q3:
SELECT ?c ?m WHERE {
?c g:featureCode g:PopulatedPlace.
?c g:parentCountry ?l .
?l g:name "Canada" .
?y sameAs ?c .
?m s:hasLocation ?y .
?m type s:Workshop . }
```

The shaded triple patterns correspond to the common subgraph.

Fig. 2. Multiple Federated SPARQL Queries

### A. Challenges & Our Solutions

Multi-query optimization have been studied in distributed relational databases [17], but these techniques are difficult to apply in the LOD context. First, 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 [15] are not easily applicable in LOD context. For example, we cannot require one source to send intermediate results directly to another source [12]. Moreover, moving data from one source to another one for join processing [17] is also infeasible.

To the best of our knowledge, only one proposal about multiple SPARQL query optimization exists in the literature [16], but *only in the centralized environment*, where all RDF datasets are collected in one physical database. Given a batch of SPARQL queries, [16] proposes to identify all maximal common edge subgraphs (MCES) among query graphs. Each MCES leads to a possible MCES-based rewritten query. Then, the rewritten query with the cheapest cost is selected. As we know, identifying all MCES for a group of query graphs is a classical NP-hard problem [6]. The proposed algorithm in [16] employs the exponential time complexity, which means that the rewriting approach in [16] is not scalable with respect to the sizes and the number of query graphs.

Different from only considering "OPTIONAL" in rewriting in [16], our method considers the full spectrum of SPARQL clauses, such as "OPTIONAL", "FILTER" and the hybrid one. Furthermore, we utilize the cost model to explore the search space and select the "optimal" rewriting plan. Unfortunately, the cost model-driven optimal rewriting is a NP-hard problem (see Theorem 1). Thus, we propose a greedy rewriting algorithm with the bounded approximation ratio. Experiments on both real and synthetic datasets confirm that our cost-driven full spectrum rewriting technique is faster than using the MCES-based approach in [16] by 3-5 times in query response times.

Besides the cost-model driven re-writing strategy for multiple query optimization, we also make two technical contributions in the context of Linked Data.

First, we optimize the query-decomposition and relevant source selection in the context of LOD. Given a SPARQL query $Q$ over LOD, existing approaches only consider the properties in each source. These techniques often overestimate the set of relevant sources. Instead, we consider the whole topology structure between sources for query decomposition and relevant source selection. Experiments confirm that our approach can reduce 30% remote requests compared with the existing data localization techniques.

Second, due to the distributed nature of LOD, we study how to optimize partial match joining during multiple query processing in LOD, which does not arise in the centralized counterpart.

To the best of our knowledge, this is the first study of multiple SPARQL query optimization over LOD, with the objective to reduce the query response time and the number of remote requests. In a nutshell, we make the following contributions in this paper.

- We propose to employ the topology relation between sources in LOD to filter out irrelevant sources.
- We take into account the unique properties of S-PARQL and LOD 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 LOD, 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.

## II. Background

### A. RDF and Linked Data

*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 corresponding property.

In the context of LOD, RDF graph $G$ is distributed over different source sites. We stay close to similar definitions as found in [11] to define LOD.

*Definition 2:* **(Linked Open Data)** *Linked Open Data* 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

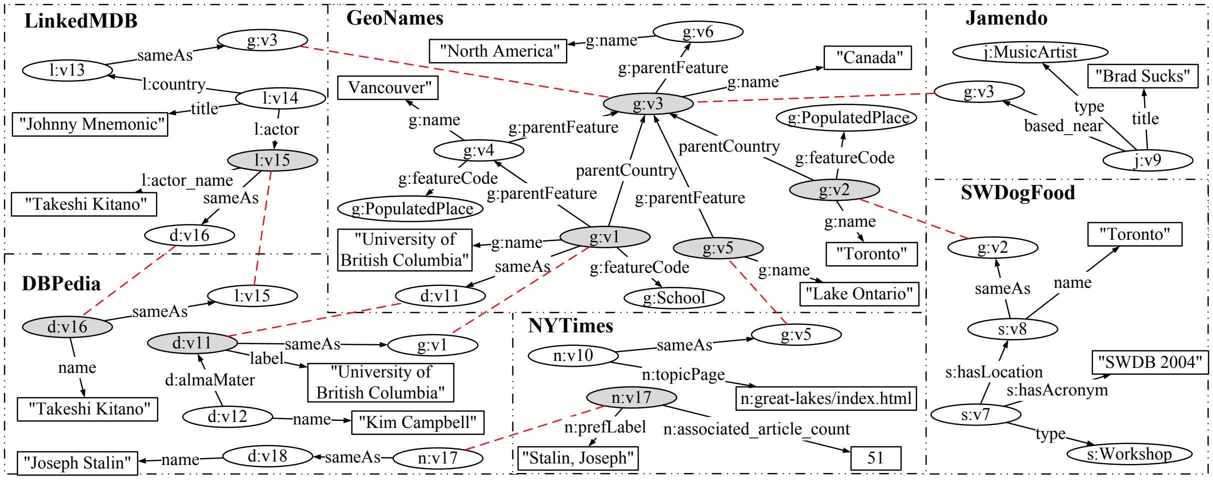


Fig. 3. LOD example

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.

## *B. 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 LOD, 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 $\mu$ from vertices in $V(Q)$ to vertices in $\bigcup_{s \in S'} g(s)$.

*Definition 4:* **(BGP Match over LOD)** Consider an RDF graph $G$, LOD $W = (S, g, d)$ 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* $\mu$ 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 $S'$, $\mu_1$ and $\mu_2$ define two matching functions $V(Q_1) \rightarrow V$ and $V(Q_2) \rightarrow V$, respectively. $\mu_1$ and $\mu_2$ 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 [20].

*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 $S'$ is defined as follows.

*Definition 7:* **(SPARQL Result over LOD)** Given LOD $W = (S, g, d)$, 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 linked data $W$, we call $[\![Q]\!]_S$ the results of $Q$ over LOD $W$.

In real applications, multiple queries over LOD at the same time are commonly overlapped. This motivates us to revisit the classical problem of multi-query optimization in the context of LOD, which is defined as follows.

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

# III. 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 our technical report [19]. A federal SPARQL query system has a control site that is amenable to receive query $Q$

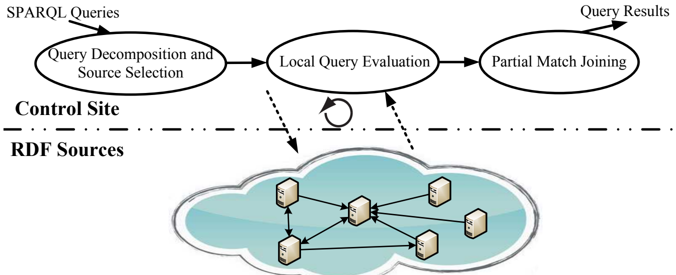


Fig. 4. Scheme for Distributed SPARQL Query Processing

and decompose $Q$ into several subqueries[1] that are sent to and evaluated at relevant sources. Local query results are returned to the control site and joined together to form complete results that are presented to users. The whole 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.

**Query Decomposition and Source Selection.** Given a query $Q$, we first decompose $Q$ into subqueries expressed on the 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 of LOD to further filter out irrelevant sources. The details of the baseline and our optimized solution will be discussed in Section IV.

We consider multiple SPARQL query over LOD. Specifically, given a batch of SPARQL queries $\{Q_1, ..., Q_n\}$, we obtain local queries $Q = \{q_1^1@S(q_1^1), ..., q_1^{m_1}@S(q_1^{m_1}); q_2^1@S(q_2^1), ..., q_2^{m_2}@S(q_2^{m_2}); ...; q_n^1@S(q_n^1), ..., q_n^{m_n}@S(S(q_n^{m_n}))\}$, where $\{q_i^1@S(q_1^1), ..., q_i^{m_i}@S(q_i^{m_i})\}$ comes 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 the response time and the number of remote accesses.

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

**Partial Match Joining.** For each local query $q_i^j$ in $Q$, collecting the matches at each relevant source in $S(q_i^j)$, we obtain all its matches. Assume that an original query $Q_i$ $(i = 1, ..., n)$ 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 LOD, we propose an optimized solution to avoid duplicate computation in join processing (in Section VI).

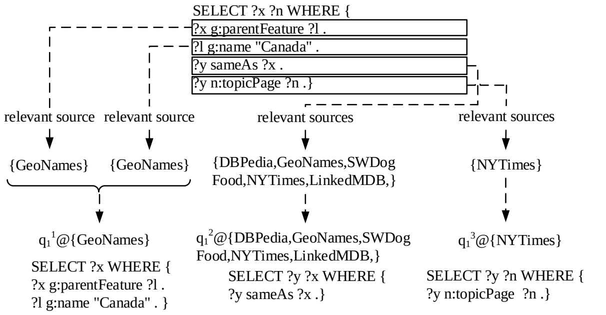
## IV. Query Decomposition and Source Selection

### A. Basic Solution

We first discuss the existing method for query decomposition and source selection [26]. Given SPARQL $Q_1$ in Fig. 1, each triple pattern corresponds to a subquery, as shown in Fig. 5. Each triple pattern maps to a set of relevant sources based on the values of its subject, property and object. 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 subqueries of one triple pattern 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 together to form a larger subquery. For example, both "?x g:parentFeature ?l " and "?l g:name "Canada" " of $Q_1$ correspond to a single source, "GeoNames". Thus, these two triple patterns can be combined together to form a larger subquery $q_1^1$. 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 subqueries $q_1^1$, $q_1^2$ 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".


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

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

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

### B. Source Topology Graph-Based Solution

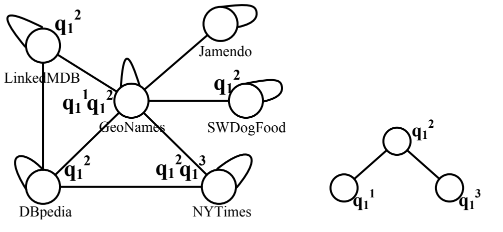
However, the baseline solution possibly overestimates the set of relevant sources. For example, for the triple pattern "?y sameAs ?x", there are five relevant sources in the baseline solution; but there is no result during join evaluation with actual mappings substituted in sources LinkedMDB and SWDogFood.

In this paper, we employ the topology structure between sources to filter out more irrelevant sources. A Web crawler for LOD [13] 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.

[1] Each subquery will be sent to a source, we also call it "local" query. Note that, for ease of presentation, we use the terms "subquery" and "local quer" interchangeably in this paper.

*Definition 8:* (**Source Topology Graph**) Given LOD $W = (S, g, d)$, 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 $\overrightarrow{u_i u_j} \in g(s_i)$ (or $\overrightarrow{u_j u_i} \in g(s_i)$ or $\overrightarrow{u_i u_j} \in g(s_j)$ or $\overrightarrow{u_j u_i} \in g(s_j)$ ), where $d(u_i) = s_i$ and $d(u_j) = s_j$. Note that $g(s_i)$ denotes the RDF graph in source $s_i$ and $d(u_i) = s_i$ means that $s_i$ is the host source of $u_i$.

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 subqueries. Specifically, according to the baseline solution, query $Q$ is decomposed into several subqueries and each of them is associated with a set of relevant sources. For example, "NYtimes" is a relevant source to $q_1^3$, we annotate "NYTimes" in STG with $q_1^3$. Fig. 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 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. 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 $Q_1$

Fig. 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 subquery $q$ does not map to a source $s$ in any homomorphism match, $s$ is not the relevant source of subquery $q$. 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 are pruned from the relevant sources of $q_1^2$. Therefore, in the optimized solution, $q_1^2$ has three relevant sources, as shown in Figure 8(a).

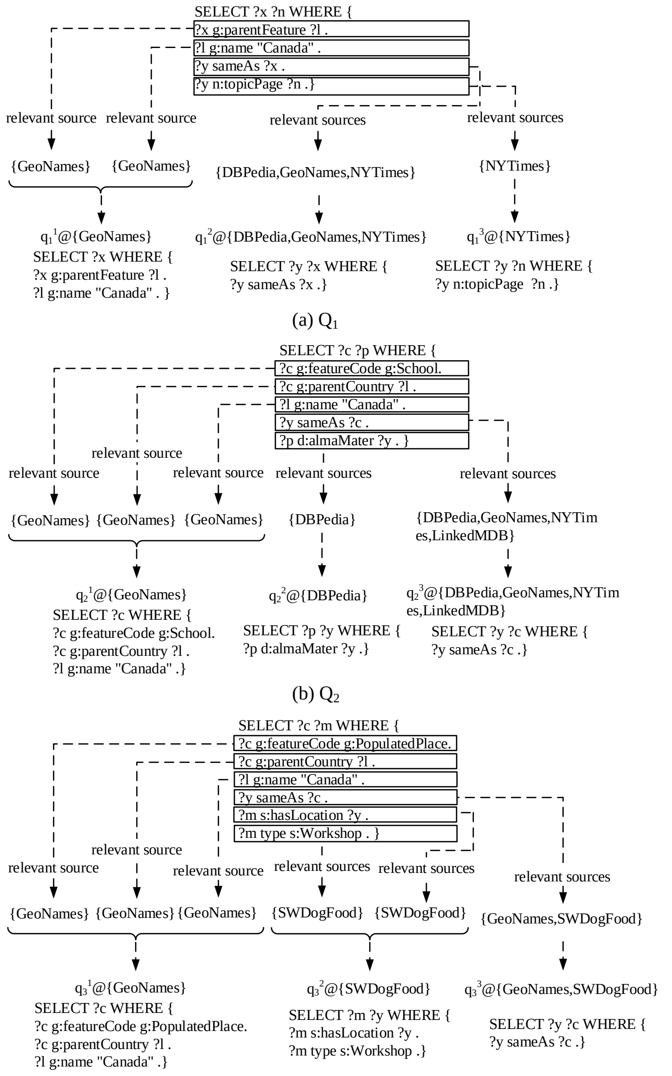
| **Match 1:** | | **Match 2:** | | **Match 3:** | |
|---|---|---|---|---|---|
| $q_1^1 \longrightarrow$ | GeoNames | $q_1^1 \longrightarrow$ | GeoNames | $q_1^1 \longrightarrow$ | GeoNames |
| $q_1^2 \longrightarrow$ | DBpedia | $q_1^2 \longrightarrow$ | GeoNames | $q_1^2 \longrightarrow$ | NYTimes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

Fig. 7. Homomorphism Matches for $JG(Q_1)$ over the Annotated Source Topology Graph

Analogously, we can also decompose $Q_2$ and $Q_3$ into subqueries 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.

## V. Local Query Evaluation

In real applications, queries at the same time are often overlapped. Thus, there is much room for sharing computation when executing these queries. 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-aware query rewriting scheme.



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

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

```
SELECT ?y ?x WHERE {
?y sameAs ?x .}
```

```
SELECT ?y ?n WHERE {
?y n:topicPage  ?n .}
```

(a) $Q_1$

```
SELECT ?c ?p WHERE {
?c g:featureCode g:School.
?c g:parentCountry ?l .
?l g:name "Canada" .
?y sameAs ?c .
?p d:almaMater ?y . }
```

```
SELECT ?c WHERE {
?c g:featureCode g:School.
?c g:parentCountry ?l .
?l g:name "Canada" .}
```

```
SELECT ?p ?y WHERE {
?p d:almaMater ?y .}
```

```
SELECT ?y ?c WHERE {
?y sameAs ?c .}
```

(b) $Q_2$

```
SELECT ?c ?m WHERE {
?c g:featureCode g:PopulatedPlace.
?c g:parentCountry ?l .
?l g:name "Canada".
?y sameAs ?c .
?m s:hasLocation ?y .
?m type s:Workshop . }
```

```
SELECT ?c WHERE {
?c g:featureCode g:PopulatedPlace.
?c g:parentCountry ?l .
?l g:name "Canada" .}
```

```
SELECT ?m ?y WHERE {
?m s:hasLocation ?y .
?m type s:Workshop .}
```

```
SELECT ?y ?c WHERE {
?y sameAs ?c .}
```

(c) $Q_3$

Fig. 8. Optimized Query Decomposition and Source Selection Results for $Q_1$, $Q_1$ and $Q_3$

### A. 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.

*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.

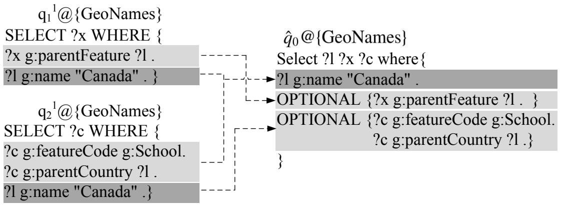
Formally, given a set of local queries $\{q_1@\{s\}, q_2@\{s\}, ..., q_n@\{s\}\}$ over the same source $s$, if

$p$ is their common subgraph among $q_1$,...,$q_n$, we rewrite these local queries to a query with OPTIONAL operator as follows.

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

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 by using left-joins, the result cardinality of a SPARQL query with OPTIONAL operator is upper bounded by result cardinality of its main graph pattern [16]. Thus, the query response time of the rewritten query does not increase much.



```
q1^1@{GeoNames}
SELECT ?x WHERE {
?x g:parentFeature ?l .
?l g:name "Canada" . }

q2^1@{GeoNames}
SELECT ?c WHERE {
?c g:featureCode g:School.
?c g:parentCountry ?l .
?l g:name "Canada" .}
```

```
q̂0@{GeoNames}
Select ?l ?x ?c where{
?l g:name "Canada" .
OPTIONAL {?x g:parentFeature ?l .  }
OPTIONAL {?c g:featureCode g:School.
                ?c g:parentCountry ?l .}
}
```

Fig. 9. Rewriting Local Queries using OPTIONAL Operators

*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.

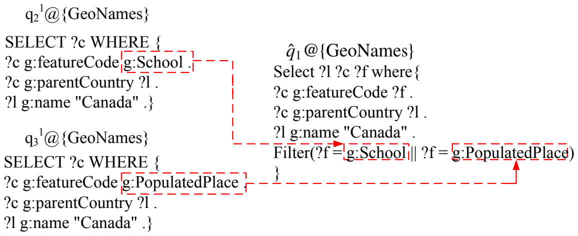
$$\hat{q}@\{s\} = p\ FILTER(\bigvee_{1 \leq i \leq n} (\bigwedge_{v \in V(q_i)} f_i(v) = v))@\{s\} \tag{2}$$

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.

*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.

Given a set of local queries $\{q_1@\{s\}, q_2@\{s\}, ..., q_n@\{s\}\}$, we rewrite them into a SPARQL query $\hat{q}$. Then, we distribute the results $[\![\hat{q}]\!]_{\{s\}}$ to different local queries. We will discuss post-processing in Section V-D.



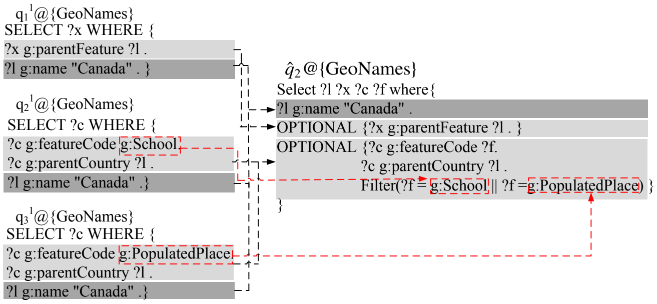
```
q2^1@{GeoNames}
SELECT ?c WHERE {
?c g:featureCode g:School .
?c g:parentCountry ?l .
?l g:name "Canada" .}

q3^1@{GeoNames}
SELECT ?c WHERE {
?c g:featureCode g:PopulatedPlace .
?c g:parentCountry ?l .
?l g:name "Canada" .}
```

```
q̂1@{GeoNames}
Select ?l ?c ?f where{
?c g:featureCode ?f .
?c g:parentCountry ?l .
?l g:name "Canada" .
Filter(?f = g:School || ?f = g:PopulatedPlace)
}
```

Fig. 10. Rewriting Local Queries using FILTER Operator

Obviously, 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-aware query rewriting strategy. We first propose our cost model in Section V-B and then discuss our cost-based query rewriting algorithm in Section V-C.



```
q1^1@{GeoNames}
SELECT ?x WHERE {
?x g:parentFeature ?l .
?l g:name "Canada" . }

q2^1@{GeoNames}
SELECT ?c WHERE {
?c g:featureCode g:School.
?c g:parentCountry ?l .
?l g:name "Canada" .}

q3^1@{GeoNames}
SELECT ?c WHERE {
?c g:featureCode g:PopulatedPlace
?c g:parentCountry ?l .
?l g:name "Canada" .}
```

```
q̂2@{GeoNames}
Select ?l ?x ?c ?f where{
?l g:name "Canada" .
OPTIONAL {?x g:parentFeature ?l . }
OPTIONAL {?c g:featureCode ?f.
               ?c g:parentCountry ?l .
               Filter(?f = g:School || ?f =g:PopulatedPlace) }
}
```

Fig. 11. Rewriting Local Queries using OPTIONAL and FILTER Operators

## B. Cost Model

*1) Cost Model for BGPs:* As mentioned in [16], selective triple patterns in BGP have higher priorities in evaluation. We verify the principle in both real and synthetic RDF repositories, such as DBpedia[2] and WatDiv [2], and experiment with a popular RDF store, Sesame 2.7[3]. For DBpedia, we download real SPARQL workload that records more than 8 millions SPARQL queries issued in 14 days of 2012 [4] 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.

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

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

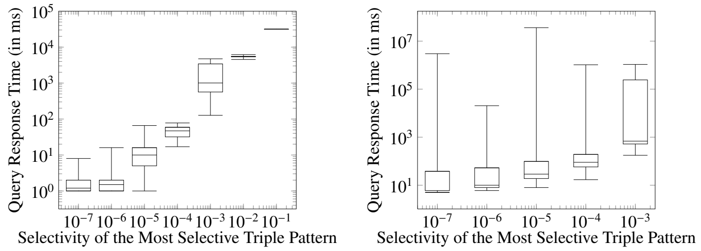
where $sel(e)$ is the selectivity of triple pattern $e$ in $Q$.

In real applications, for estimating the selectivity of triple pattern, we employ the heuristics introduced by [27] that can estimate the selectivity without pre-computed statistics about

[2] http://wiki.dbpedia.org/

[3] http://rdf4j.org/

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



(a) DBPedia

(b) WatDiv 10M

Fig. 12. Experiment Results of the Relationship Between a Query and Its Most Selective Triple Pattern

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 [27].

Furthermore, many RDF sources also provide VOID (Vocabulary of Interlinked Datasets)[5] that expresses metadata about RDF datasets. Although VOID was not initially designed for being used for query optimization, the statistics in VOID can also be utilized to implement the heuristic with precomputed statistics in [27] to improve the estimation accuracy.

*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) [16]. 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 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 $\Delta_1$ and $\Delta_2$ are empirically trivial values [16].

*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, $\Delta_1$ and $\Delta_2$ are trivial values and $cost(\hat{q})$ is mostly credited to the evaluation of $p$. Hence, we ignore the trivial variables $\Delta_1$ and $\Delta_2$ 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 $Q = \{q_1@s, q_2@s, ..., q_n@s\}$ 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 the rewriting is the cost of the rewritten query $\hat{q}$ with main basic graph pattern $p$ as shown in the following formula:

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

### C. 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 $\{\hat{q_1}, ..., \hat{q_m}\}$ ($m \le n$) 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 V-A, 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.

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

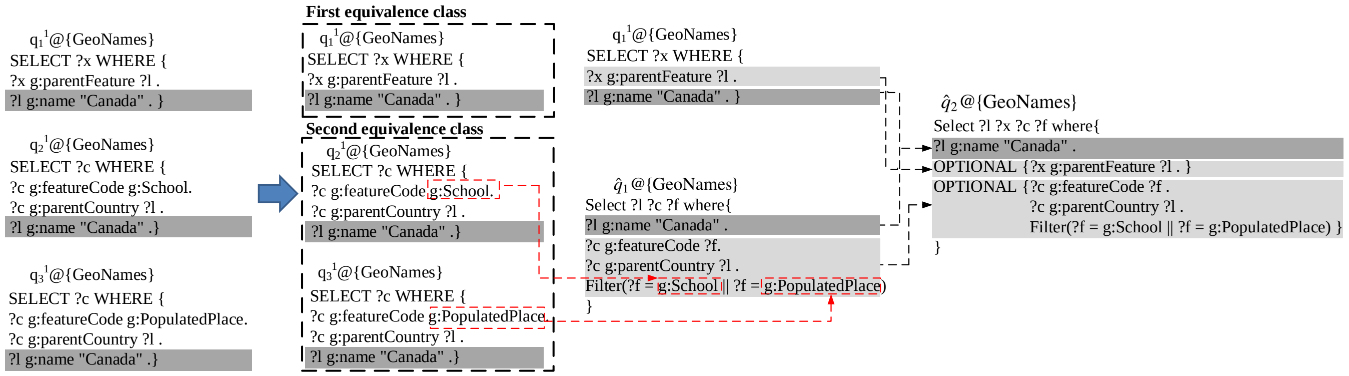
Given a set of original queries $Q$, Equation 7 is a set-function with respect to set $P$, i.e., a set of main patterns. Unfortunately, finding the optimal rewriting is a NP-complete problem as discussed in the following theorem.

*Theorem 1:* 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. [6]

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

[5] http://www.w3.org/TR/void/

[6] All proofs for theorems are given in our technical report [19]



```
q1^1@{GeoNames}
SELECT ?x WHERE {
?x g:parentFeature ?l .
?l g:name "Canada" . }

q2^1@{GeoNames}
SELECT ?c WHERE {
?c g:featureCode g:School.
?c g:parentCountry ?l .
?l g:name "Canada" .}

q3^1@{GeoNames}
SELECT ?c WHERE {
?c g:featureCode g:PopulatedPlace.
?c g:parentCountry ?l .
?l g:name "Canada" .}
```

```
q̂1@{GeoNames}
Select ?l ?c ?f where{
?l g:name "Canada" .
?c g:featureCode ?f.
?c g:parentCountry ?l .
Filter(?f = g:School || ?f = g:PopulatedPlace)
}
```

```
q̂2@{GeoNames}
Select ?l ?x ?c ?f where{
?l g:name "Canada" .
OPTIONAL {?x g:parentFeature ?l . }
OPTIONAL {?c g:featureCode ?f .
          ?c g:parentCountry ?l .
          Filter(?f = g:School || ?f = g:PopulatedPlace) }
}
```

Fig. 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 $Q_{OPT}$.
while $Q \neq \emptyset$ do
    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 $e_{max}$;
    Extract the largest common pattern $p$ of queries in $Q'$;
    Build a rewritten query $\hat{q}$, where $p$ is its main pattern;
    Divide $Q'$ into a collection of equivalence classes $C$, where each class contains local queries isomorphic to each other;
    for each class $C \in C$ do
        Generalize a pattern $p'$ isomorphic all patterns in $C$, where $p'$ does not contain any constants;
        Build a query pattern with $p'$;
        Add FILTER operators by mapping $p'$ to patterns in $C$;
        Add the pattern into $\hat{q}$ as an OPTIONAL pattern;
    Add $\hat{q}$ into $Q_{OPT}$;
    $Q = Q - Q'$;
Return $Q_{OPT}$;
```

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 pattern with FILTER operators as discussed in Section V-A. 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 2:* 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.

### *D. Postprocessing*

Given a set of local queries $Q$ that will be sent to source $s$, we rewrite them into $\hat{Q}$. Then, we evaluate each rewritten query $\hat{q}$ in $\hat{Q}$ at source $s$. Let $[\![\hat{q}]\!]_{\{s\}}$ denote the result set of $\hat{q}$ 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 being optimized, and we record the mappings between the variables in the rewritten query and the variables in the original input queries. The result of a rewritten query might have empty (null) columns corresponding to the variables from the OPTIONAL clause. Therefore, a result in $[\![\hat{q}]\!]_{\{s\}}$ may not conform to the description of every local query in $Q$. We should identify the valid overlap between each result in $[\![\hat{q}]\!]_{\{s\}}$ and the results of each local query in $Q$, and check whether the source of 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.

To achieve the above objective, we performs a Boolean intersection between each result in $[\![\hat{q}]\!]_{\{s\}}$ and each local query: if the columns of this result corresponding to those columns of a local query $q@s \in Q$ are not null and the source of this result belongs to $s$, the algorithm distributes the corresponding part of this result to $q@\{s\}$ as one of its query results. This step iterates over each row and each local query in $Q$. The checking on $[\![\hat{q}]\!]_{\{s\}}$ only requires a linear scan on $[\![\hat{q}]\!]_{\{s\}}$. Therefore, it can be done on-the-fly as the results of $\hat{q}@\{s\}$ is streamed out from the evaluation.

## VI. Joining Partial Matches

Unlike the centralized environment in [16], multiple query evaluation over Linked Data requires joining local partial matches. In this section, we discuss how to join partial matches efficiently.

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 $[\![q_i^{m_i}]\!]$.

The straightforward method to obtain results of all original queries is that we join local query matches for each original SPARQL query independently. However, considering multiple queries, there may exist some common computation in joining partial matches. Taking advantage of these common 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 $\llbracket q_i^{i_1} \rrbracket \bowtie \llbracket q_i^{i_2} \rrbracket$ and $\llbracket q_j^{j_1} \rrbracket \bowtie \llbracket q_j^{j_2} \rrbracket$ into $(\llbracket q_i^{i_1} \rrbracket \cup \llbracket q_j^{j_1} \rrbracket) \bowtie (\llbracket q_i^{i_2} \rrbracket \cup \llbracket q_j^{j_2} \rrbracket)$.

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: $\llbracket q_i^{i_1} \rrbracket \bowtie \llbracket q_i^{i_2} \rrbracket$ and $\llbracket q_j^{j_1} \rrbracket \bowtie \llbracket q_j^{j_2} \rrbracket$ versus $(\llbracket q_i^{i_1} \rrbracket \cup \llbracket q_j^{j_1} \rrbracket) \bowtie (\llbracket q_i^{i_2} \rrbracket \cup \llbracket q_j^{j_2} \rrbracket)$.

The cost of executing $\llbracket q_i^{i_1} \rrbracket \bowtie \llbracket q_i^{i_2} \rrbracket$ and $\llbracket q_j^{j_1} \rrbracket \bowtie \llbracket q_j^{j_2} \rrbracket$ separately is the sum of the costs of two joins. Thus,

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

On the other hand, the cost of executing $(\llbracket q_i^{i_1} \rrbracket \cup \llbracket q_j^{j_1} \rrbracket) \bowtie (\llbracket q_i^{i_2} \rrbracket \cup \llbracket q_j^{j_2} \rrbracket)$ is as follows.

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

The our optimization technique is better if it acts as a sufficient reducer, that is, if $\llbracket q_i^{i_1} \rrbracket$ and $\llbracket q_j^{j_1} \rrbracket$ overlap a lot and $\llbracket q_i^{i_2} \rrbracket$ and $\llbracket q_j^{j_2} \rrbracket$ 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 [28]. 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.

## VII. Experimental Evaluation

In this section, we evaluate our method over both real (FedBench) and synthetic datasets (WatDiv). We compare our system with two classical federated SPARQL query engines, such as FedX [26] and SPLENDID [9]. We also conduct two additional comparative experiments. One is to compare our STG-based source selection with state-of-the-art approaches [10], [21] (see Exp. 1). The other one is to evaluate our cost model-driven full spectrum rewriting approach, compared with [16] (denoted as [Le et al.], see Exp. 2).

**Datasets.** (1) 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 workloads by instantiating some templates with actual RDF terms from the dataset, and we generate different workloads to test our approaches.

(2) FedBench [25] 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. FedBench is generated by collecting 9 RDF datasets, we assume each dataset is distributed over a source site. To distribute 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 [14] 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.

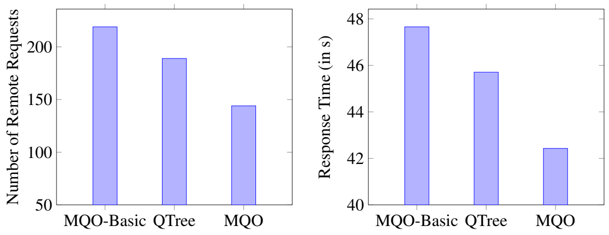
### A. Evaluation of Proposed Techniques

In this section, we use WatDiv 10M to evaluate each proposed technique in this paper. By default, we use 150 queries in the following experiments, meaning 150 queries are posed simultaneously.

**Exp.1: Evaluating STG-based Source Selection.** In Fig. 14, we compare our STB-based solution with the baseline (denoted as MQO-Basic). Furthermore, we also compare the source selection method proposed in [10], [21] (denoted as QTree). It only employs the neighborhood information between adjacent sources without considering the whole topology structure between sources.

Obviously, MQO-Basic is not as good as the other two approaches. QTree only uses local neighborhood information and does not consider the whole topology of relevant sources. Hence, the effectiveness of its pruning power in source selection is limited, compared with our STG-based solution. Figure 14 shows that our method can reduce both the number of remote accesses and query response times.

**Exp 2. Effect of the Rewriting Strategies.** In this experiment, we evaluate two aspects of our cost model-driven full spectrum rewriting. First, in order to evaluate the effectiveness



(a) Number of Remote Requests

(b) Response Time

Fig. 14. Evaluating Source Topology-based Source Selection Technique

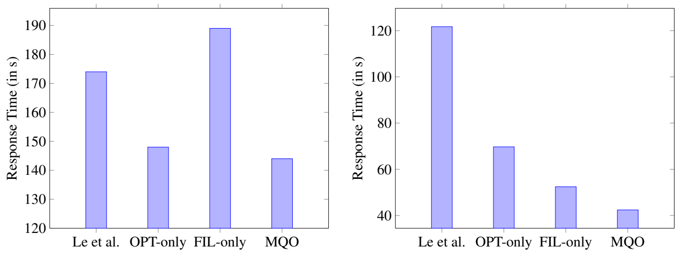
of our full spectrum rewriting, we compare our method with only using OPTIONAL operators (denoted as OPT-only) and only using FILTER operators for rewriting (denoted as FIL-only). However, in both OPT-only and FIL-only, we also implement the cost-model driven approach. Second, In order to evaluate the effectiveness of our cost model-driven approach, we also re-implement the rewriting strategies proposed in [16] (denoted as Le et al.) to rewrite local queries.

**Evaluating Full Spectrum Rewriting.** From Figure 15, we find that FIL-only leads to the largest number of rewritten queries, meaning it results in more remote requests than other rewriting strategies. FIL-only rewrites queries that share the exact same query structures. Obviously, the chance of rewriting in FIL-only is much smaller than others. Even though FIL-only leads to the largest number of remote accesses, but it is still faster than OPT-only in the query response time, since the latter often leads to an expensive left-out join. Our full spectrum rewriting combines both and are good at both the number of remote accesses and query response times.

**Evaluating Cost-model Driven Approach.** Since Le et al. [16] only consider OPTIONAL-based rewriting, it is fair to compare [Le et al.] with OPT-only. From Figure 15, we find that cost model-driven OPT-only has fewer remote accesses and faster query response times. The reason is that [Le et al.] only rewrite queries when their query graph structures are quite similar, since [Le et al.] needs to divide a batch of queries into different clusters according to their graph structure similarity. They only rewrite queries in the same cluster. However, we adopt cost-model driven approach. Even though some queries are quite different in structures, they can also be rewritten together if they share a single selective triple pattern. It is cost-model driven rewriting instead of the structure-driven. In other words, OPT-only can lead to less rewritten queries, meaning less remote accesses. Fewer remote accesses leads to faster query response times.

In a word, both our proposed optimized techniques are effective in reducing the number of remote accesses and query response times.

**Exp 3. Effect of Optimization Techniques for Joins.** We evaluate our optimized join strategy proposed in Section VI. We design a baseline (denoted as MQO-QR) that implements query rewriting strategies but without optimized join strategy. Obviously, MQO-QR has the same performance of local query evaluation stage, but they are different in join processing. Generally speaking, it reduces 10% join processing time, as shown in Fig. 16, since our method takes advantage of sharing computing.



(a) Number of Remote Requests

(b) Response Time

Fig. 15. Evaluating Different Rewriting Strategies

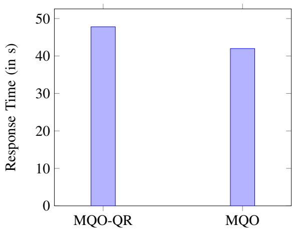
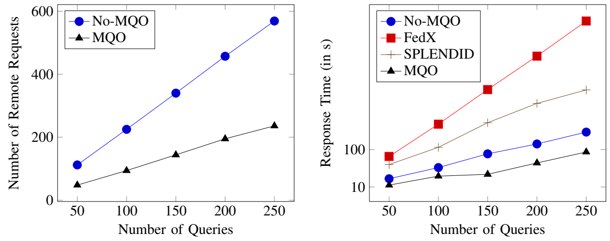


Fig. 16. Effect of Optimization Techniques for Joins

## B. *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-MQO). This baseline uses STG-based source selection but it does not employ any optimizations for multiple queries. We also compare our method with two classical federated SPARQL query system over LOD, such as FedX and SPLENDID. Since both of them are designed for single query evaluation, to enable comparison, we runs multiple federated queries sequentially in 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.

**Exp. 4: Varying Number of Queries.** We study the impact on the number of queries, for which we vary from 100 to 250 queries, by an increment of 50. Fig. 17 shows the experimental results.



(a) Number of Remote Requests

(b) Response Time

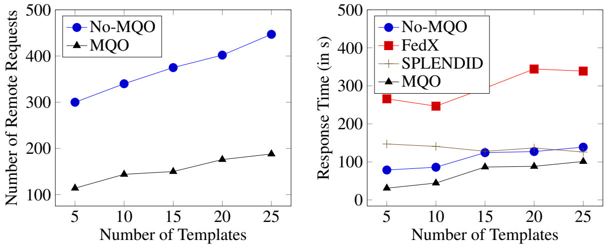
Fig. 17. Varying Number of Queries

Due to query rewriting, our method (MQO) can merge many local queries into fewer rewritten queries, which result in smaller number of remote requests, as shown in Fig. 17(a). Generally speaking, MQO can save the number of remote access by $\frac{1}{2}$-$\frac{2}{3}$, compared with No-MQO (evaluating multiple queries sequentially). Note that, since FedX and SPLENDID system does not report their numbers of remote requests, we do not compare MQO with FedX or SPLENDID in Fig. 17(a).

In terms of evaluation times (see Fig. 17(b)), fewer remote requests result in less evaluation time, and the baseline method without any optimization (i.e., No-MQO) takes a third more times than MQO approach in running time. Furthermore, the optimized solution for multiple queries in MQO also speeds up query response times. The join processing approach in FedX and SPLENDID is very expensive according to our experiments, leads to much slower query response time than both No-MQO and MQO.

**Exp 5: 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 Fig. 18. Similarly, we do not compare MQO with them in Fig. 18(a).

Fig. 18(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 remote requests lead to worse performance as shown in Fig. 18(b). However, the response time of MQO is still 50% less than No-MQO and SPLENDID, and two thirds less than FedX.



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

Fig. 18. Varying Number of Query Templates

**Exp 6: 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. 19 shows the results.

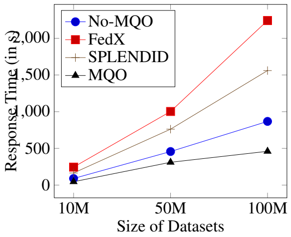
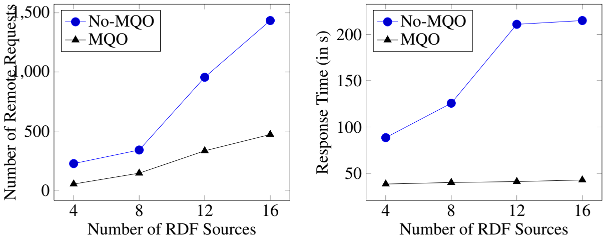


Fig. 19. Varying Size of Datasets

While this does not affect the number of remote request, it clearly affects evaluation times, as shown in Fig. 19. As the size of RDF datasets gets larger, the response time of all three methods increases. However, the rate of rise for MQO is smaller than other competitors. The response time of MQO decreases from 60% of No-MQO to 50% of No-MQO, while the response time of MQO always is 30% of SPLENDID to 20% of FedX.

**Exp. 7: Varying Number of Sources.** In this experiment, we vary the number of sources from 4 to 16. Fig. 20 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 remote request are needed to evaluate a query. However, MQO grows much slower than No-MQO in both the number of remote accesses and query response time. It confirms that MQO has better scalability with the number of sources.

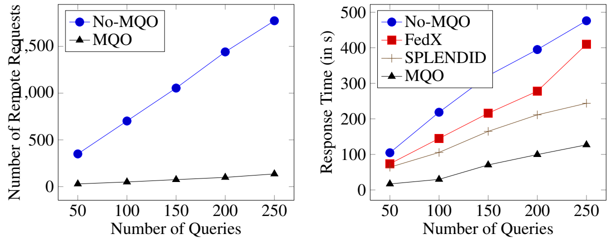


(a) Number of Remote Requests (b) Response Time

Fig. 20. Varying Number of RDF Sources

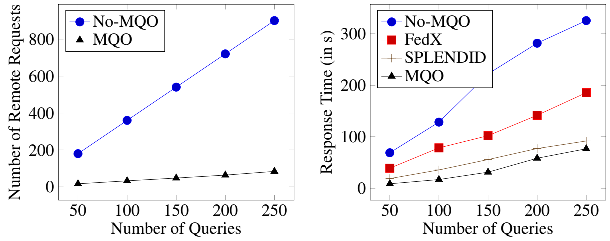
## *C. Performance over Real Dataset*

In this experiment, we test our MQO 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 Fig. 21 and Fig. 22. Experiments confirm that MQO lead to fewest remote requests for FedBench and result in best query performance.



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

Fig. 21. FedBench (Cross Domain)



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

Fig. 22. FedBench (Life Science)

# VIII. Related Work

There are two threads of related work: SPARQL query processing in Linked Data and multi-query Optimization.

**Federated Query Processing in Linked Data.** Recently, many approaches [22], [26], [9], [10], [21], [24], [23] have been proposed for federated SPARQL query processing in Linked Data. Since RDF sources in Linked Data are autonomous, they cannot be interrupted during query execution. Therefore, the major challenges for processing SPARQL queries in Linked Data are data localization and global query optimization. The challenges are 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 [22] 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], [21] 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 [23] 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.

Besides the metadata-assisted methods, there are still a few papers that do not require the metadata. In FedX [26], 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.

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 [22] and FedX [26] discuss how to use a semijoin algorithm to join intermediate results.

**Multiple SPARQL Queries Optimization.** Le et al. [16] 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 [18], [3] on Hadoop. However, LOD employs an autonomous distributed environment, which is quite different from MapReduce systems. Therefore, some existing solutions in [18], [3] are infeasible in the context of LOD.

## IX. Conclusion

We studied the problem of multi-query optimization over Linked Data. 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.

## 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] N. Biggs, E. Lloyd, and R. Wilson. Graph Theory. *Oxford University Press*, 1986.

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

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

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

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

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

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

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

[18] 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.

[19] P. Peng, L. Zou, T. M. Ozsu, and D. Zhao. (technical report), github.com/bnu05pp/distributedfederatedmqo/blob/master/full_paper.pdf.

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

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

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

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

[24] M. Saleem, A. N. Ngomo, J. X. Parreira, H. F. Deus, and M. Hauswirth. DAW: Duplicate-AWare Federated Query Processing over the Web of Data. In *ISWC*, pages 574–590, 2013.

[25] 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.

[26] 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.

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

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

## Appendix

So far, we only consider BGP (basic graph patterns) evaluation over LOD. 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. 23(a) to a group of two BGPs as in Fig. 23(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

Fig. 23. 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. 24(a), it is decomposed to two local queries as in Fig. 24(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

Fig. 24. 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. 24(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)}
```

Fig. 25. Rewritten Query with FILTER Operators

*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. ■

*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 [8], the problem of selecting the optimal set of patterns is also NP-complete. ■

*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 [7] shows that the approximation ratio of the greedy algorithm to the optimal solution of the weighted set-cover problem is $(1 + ln| \cup_{q \in Q} E(q)|)$. ■