Scalable Multi-Query Optimization for SPARQL

Wangchao Le[1] Anastasios Kementsietsidis[2] Songyun Duan[2] Feifei Li[1]
[1]*School of Computing, University of Utah, Salt Lake City, UT, USA*
[2]*IBM T.J. Watson Research Center, Hawthorne, NY, USA*
[1]{lew,lifeifei}@cs.utah.edu, [2]{akement, sduan}@us.ibm.com

***Abstract*—This paper revisits the classical problem of multi-query optimization in the context of RDF/SPARQL. We show that the techniques developed for relational and semi-structured data/query languages are hard, if not impossible, to be extended to account for RDF data model and graph query patterns expressed in SPARQL. In light of the NP-hardness of the multi-query optimization for SPARQL, we propose heuristic algorithms that partition the input batch of queries into groups such that each group of queries can be optimized together. An essential component of the optimization incorporates an efficient algorithm to discover the common sub-structures of multiple SPARQL queries and an effective cost model to compare candidate execution plans. Since our optimization techniques do not make any assumption about the underlying SPARQL query engine, they have the advantage of being portable across different RDF stores. The extensive experimental studies, performed on three popular RDF stores, show that the proposed techniques are effective, efficient and scalable.**

I. Introduction

With the proliferation of RDF data, over the years, a lot of effort has been devoted in building RDF stores that aim to efficiently answer graph pattern queries expressed in SPARQL. There are generally two routes to building RDF stores: (i) migrating the schema-relax RDF data to relational data, *e.g.,* Virtuoso, Jena SDB, Sesame, 3store; and (ii) building generic RDF stores from scratch, *e.g.,* Jena TDB, RDF-3X, 4store, Sesame Native. As RDF data are schema-relax [26] and graph pattern queries in SPARQL typically involve many joins [1], [19], a full spectrum of techniques have been proposed to address the new challenges. For instance, vertical partitioning was proposed for relational backend [1]; side-way information passing technique was applied for scalable join processing [19]; and various compressing and indexing techniques were designed for small memory footprint [3], [18]. With the infrastructure being built, the community is turning to develop more advanced applications, *e.g.,* integrating and harvesting knowledge on the Web [24], rewriting queries for fine-grained access control [17] and inference [13]. In such applications, a SPARQL query over views is often rewritten into an equivalent batch of SPARQL queries for evaluation over the base data. As the semantics of the rewritten queries in the same batch are commonly overlapped [13], [17], there is much room for sharing computation when executing these rewritten queries. This observation motivates us to revisit the classical problem of multi-query optimization (MQO) in the context of RDF and SPARQL.

Not surprisingly, MQO for SPARQL queries is NP-hard, considering that MQO for relational queries is NP-hard [30] and the established equivalence between SPARQL and relational algebra [2], [23]. It is tempting to apply the MQO techniques developed in relational systems to address the MQO problem in SPARQL. For instance, the work by P. Roy *et al.* [27] represented query plans in AND-OR DAGs and used heuristics to partially materialize intermediate results that could result in a promising query throughput. Similar themes can be seen in a variety of contexts, including relational queries [30], [31], XQueries [6], aggregation queries [36], or more recently as full-reducer tree queries [15]. These off-the-shelf solutions, however, are hard to be engineered into RDF query engines in practice. The first source of complexity for using the relational techniques and the like stems from the physical design of RDF data itself. While indexing and storing relational data commonly conform to a carefully calibrated relational schema, many variances existed for RDF data; *e.g.,* the giant triple table adopted in 3store and RDF-3X, the property table in Jena, and more recently the use of vertical partitioning to store RDF data. These, together with the disparate indexing techniques, make the cost estimation for an individual query operator (the corner stone for any MQO technique) highly error-prone and store dependent. Moreover, as observed in previous works [1], [19], SPARQL queries feature more joins than typical SQL queries – a fact that is also evident by comparing TPC benchmarks [34] with the benchmarks for RDF stores [5], [9], [11], [28]. While existing techniques commonly root on looking for the best plan in a greedy fashion, comparing the cost for alternative plans becomes impractical in the context of SPARQL, as the error for selectivity estimation inevitably increases when the number of joins increases [18], [33]. Finally, in W3C's envision [26], RDF is a very general data model, therefore, knowledge and facts can be seamlessly harvested and integrated from various SPARQL endpoints on the Web [38] (powered by different RDF stores). While a specialized MQO solution may serve inside the optimizer of certain RDF stores, it is more appealing to have a generic MQO framework that could smoothly fit into any SPARQL endpoint, which would be coherent with the design principle of RDF data model.

With the above challenges in mind, in this paper, we study MQO of SPARQL queries over RDF data, with the objective to minimize total query evaluation time. Specifically, we employ query rewriting techniques to achieve desirable and consistent performance for MQO across different RDF stores, with the guarantee of *soundness* and *completeness*. While the previous works consider alignments for the common substructures in *acyclic* query plans [15], [27], we set forth to identify common subqueries (cyclic query graphs included) and rewrite them with SPARQL in a meaningful way. Unlike [27], which requires explicitly materializing and indexing the common intermediate results, our approach works on top of any RDF engine and ensures that the underlying RDF stores can automatically cache and reuse such results. In addition, a full range of optimization techniques in different RDF stores and SPARQL query optimizers can seamlessly support our MQO technique. Our contributions can be summarized as follows.

- We present a generic technique for MQO in SPARQL. Unlike the previous works that focus on synthesizing query plans, our technique summarizes similarity in the structure of SPARQL queries and takes into account the unique properties (*e.g.,* cyclic query patterns) of SPARQL.
- Our MQO approach relies on query rewriting, which is built on the algorithms for finding common substructures. In addition, we tailored efficient and effective optimizations for finding common subqueries in a batch of SPARQL queries.
- We proposed a practical cost model. Our choice of the cost model is determined both by the idiosyncrasies of the SPARQL language and by our empirical digest of how SPARQL queries are executed in existing RDF data management systems.
- Extensive experiments with large RDF data (close to 10 million triples) performed on three different RDF stores consistently demonstrate the efficiency and effectiveness of our approach over the baseline methods.

II. Preliminaries

A. SPARQL

SPARQL, a W3C recommendation, is a pattern-matching query language. There are two types of SPARQL queries in which we are going to focus our interest:

Type 1: Q := SELECT RD WHERE GP
Type 2: $\mathsf{Q}_{\mathsf{OPT}}$:= SELECT RD WHERE GP (OPTIONAL $\mathsf{GP}_{\mathsf{OPT}})^{+}$

where, GP is a set of *triple patterns*, *i.e.,* triples involving both variables and constants, and RD is the *result description*. Given an RDF data graph D, the triple pattern GP searches on D for a set of subgraphs of D, each of which *matches* the graph pattern in GP (by binding pattern variables to values in the subgraph). The result description RD for both query types contains a subset of variables in the graph patterns, similar to a projection in SQL. The difference between the two types is clearly in the OPTIONAL clause. Unlike query Q, in the $\mathsf{Q}_{\mathsf{OPT}}$ query a subgraph of D might match not only the pattern in GP but also the pattern (combination) of GP and $\mathsf{GP}_{\mathsf{OPT}}$. While more than one OPTIONAL clauses are allowed, subgraph matching with D independently considers the combination of pattern GP with each of the OPTIONAL clauses. Therefore, with n OPTIONAL clauses in query $\mathsf{Q}_{\mathsf{OPT}}$, the query returns as results the subgraphs that match any of the n (GP + $\mathsf{GP}_{\mathsf{OPT}}$) pattern combinations, plus the results that match just the GP pattern.

Consider the data and SPARQL query in Figure 1(a) and (b). The query looks for triples whose subjects (each corresponding

subj	pred	obj
p1	name	”Alice”
p1	zip	10001
p1	mbox	alice@home
p1	mbox	alice@work
p1	www	http://home/alice
p2	name	”Bob”
p2	zip	”10001”
p3	name	”Ella”
p3	zip	”10001”
p3	www	http://work/ella
p4	name	”Tim”
p4	zip	”11234”

(a) Input data D

```
SELECT ?name, ?mail, ?hpage
WHERE { ?x name ?name, ?x zip 10001,
        OPTIONAL {?x mbox ?mail }
        OPTIONAL {?x www ?hpage }}
```

(b) Example query $\mathsf{Q}_{\mathsf{OPT}}$

name	mail	hpage
”Alice”	alice@home	
”Alice”	alice@work	
”Alice”		http://home/alice
”Bob”		
”Ella”		http://work/ella

(c) Output $\mathsf{Q}_{\mathsf{OPT}}(\mathsf{D})$

Fig. 1. An example

to a person) have the predicates name and zip, with the latter having the value 10001 as object. For these triples, it returns the object of the name predicate. Due to the first OPTIONAL clause, the query also returns the object of predicate mbox, if the predicate exists. Due to the second OPTIONAL clause, the query also independently returns the object of predicate www, if the predicate exists. Evaluating the query over the input data D (can be viewed as a graph) results in output $\mathsf{Q}_{\mathsf{OPT}}(\mathsf{D})$, as shown in Figure 1(c).

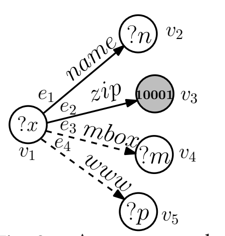
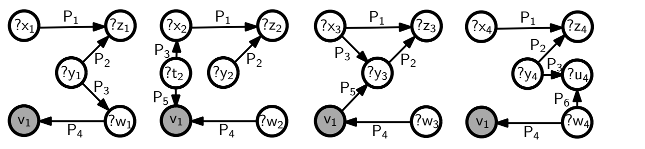


Fig. 2. A query graph

We represent queries graphically, and associate with each query Q ($\mathsf{Q}_{\mathsf{OPT}}$) a *query graph pattern* corresponding to its pattern GP (resp., GP (OPTIONAL $\mathsf{GP}_{\mathsf{OPT}})^{+}$). Formally, a query graph pattern is a 4-tuple (V, E, ν, μ) where V and E stand for vertices and edges, ν and μ are two functions which assign labels (*i.e.,* constants and variables) to vertices and edges of GP respectively. Vertices represent the subjects and objects of a triple; gray vertices represent constants, and white vertices represent variables. Edges represent predicates; dashed edges represent predicates in the optional patterns $\mathsf{GP}_{\mathsf{OPT}}$, and solid edges represent predicates in the required patterns GP. Figure 2 shows a pictorial example for the query in Figure 1(b). Its query graph patterns GP and $\mathsf{GP}_{\mathsf{OPT}}$s are defined separately. GP is defined as (V, E, ν, μ), where $V = \{v_1, v_2, v_3\}$, $E = \{e_1, e_2\}$ and the two naming functions $\nu = \{\nu_1 : v_1 \to ?x, \nu_2 : v_2 \to ?n, \nu_3 : v_3 \to 10001\}$, $\mu = \{\mu_1 : e_1 \to name, \mu_2 : e_2 \to zip\}$. For the two OPTIONALs, they are defined as $\mathsf{GP}_{\mathsf{OPT1}} = (V', E', \nu', \mu')$, where $V' = \{v_1, v_4\}$, $E' = \{e_3\}$, $\nu' = \{\nu'_1 : v_1 \to ?x, \nu'_2 : v_4 \to ?m\}$, $\mu' = \{\mu'_1 : e_3 \to mbox\}$; Likewise, $\mathsf{GP}_{\mathsf{OPT2}} = (V'', E'', \nu'', \mu'')$, where $V'' = \{v_1, v_5\}$, $E'' = \{e_4\}$, $\nu'' = \{\nu''_1 : v_1 \to ?x, \nu''_2 : v_5 \to ?p\}$, $\mu'' = \{\mu''_1 : e_4 \to www\}$.

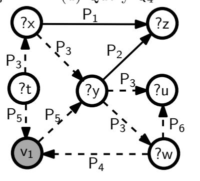
B. Problem statement

Formally, the problem of MQO in SPARQL, from query rewriting perspective, is defined as follows: Given a data graph G, and a set $\mathcal{Q}$ of **Type 1** queries, compute a new set $\mathcal{Q}_{\mathsf{OPT}}$ of **Type 1** and **Type 2** queries, evaluate $\mathcal{Q}_{\mathsf{OPT}}$ over G and distribute the results to the queries in $\mathcal{Q}$. There are two requirements for the rewriting approach to MQO: (i) The query results of $\mathcal{Q}_{\mathsf{OPT}}$ can be used to produce the same results as executing the original queries in $\mathcal{Q}$, which ensures the *soundness* and *completeness* of the rewriting; and (ii) the evaluation time of $\mathcal{Q}_{\mathsf{OPT}}$,



(a) Query Q_1 (b) Query Q_2 (c) Query Q_3 (d) Query Q_4

```
SELECT *
WHERE { $?x$ P$_1$ $?z$, $?y$ P$_2$ $?z$,
    OPTIONAL {$?y$ P$_3$ $?w$, $?w$ P$_4$ $v_1$ }
    OPTIONAL {$?t$ P$_3$ $?x$, $?t$ P$_5$ $v_1$, $?w$ P$_4$ $v_1$ }
    OPTIONAL {$?x$ P$_3$ $?y$, $v_1$ P$_5$ $?y$, $?w$ P$_4$ $v_1$ }
    OPTIONAL {$?y$ P$_3$ $?u$, $?w$ P$_6$ $?u$, $?w$ P$_4$ $v_1$ }
}
```



(e) Example query $\mathsf{Q}_{\mathsf{OPT}}$

```
SELECT *
WHERE { $?w$ P$_4$ $v_1$,
    OPTIONAL {$?x_1$ P$_1$ $?z_1$, $?y_1$ P$_2$ $?z_1$, $?y_1$ P$_3$ $?w$ }
    OPTIONAL {$?x_2$ P$_1$ $?z_2$, $?y_2$ P$_2$ $?z_2$, $?t_2$ P$_3$ $?x_2$, $?t_2$ P$_5$ $v_1$ }
    OPTIONAL {$?x_3$ P$_1$ $?z_3$, $?y_3$ P$_2$ $?z_3$, $?x_3$ P$_3$ $?y_3$, $v_1$ P$_5$ $?y_3$ }
    OPTIONAL {$?x_4$ P$_1$ $?z_4$, $?y_4$ P$_2$ $?z_4$, $?y_4$ P$_3$ $?u_4$, $?w$ P$_6$ $?u_4$ }
}
```

pattern p	α(p)
$?x$ P$_1$ $?z$	15%
$?y$ P$_2$ $?z$	9%
$?y$ P$_3$ $?w$	18%
$?w$ P$_4$ v_1	4%
$?t$ P$_5$ v_1	2%
v_1 P$_5$ $?t$	7%
$?w$ P$_6$ $?u$	13%

(f) Structure and cost-based optimization

Fig. 3. Multi-query optimization example

including query rewriting, execution, and result distribution, should be less than the baseline of executing the queries in $\mathcal{Q}$ sequentially. To ease presentation, we assume that the input queries in $\mathcal{Q}$ are of **Type 1**, while the output (optimized) queries are either of **Type 1** or **Type 2**. Our optimization techniques can easily handle more general scenarios where both query types are given as input (section IV).

We use a simple example to illustrate the MQO envisioned and some challenges for the rewriting approach. Figure 3(a)-(d) show the graph representation of four queries of **Type 1**. Figure 3(e) shows a **Type 2** query $\mathsf{Q}_{\mathsf{OPT}}$ that *rewrites* all four input queries into one. To generate query $\mathsf{Q}_{\mathsf{OPT}}$, we identify the (largest) common subquery in all four queries: the subquery involving triples $?x$ P$_1$ $?z$, $?y$ P$_2$ $?z$ (the second largest common subquery involves only one predicate, P$_3$ or P$_4$). This common subquery constitutes the graph pattern GP of $\mathsf{Q}_{\mathsf{OPT}}$. The remaining subquery of each individual query generates an OPTIONAL clause in $\mathsf{Q}_{\mathsf{OPT}}$. Note that by generating a query like $\mathsf{Q}_{\mathsf{OPT}}$, the triple patterns in GP of $\mathsf{Q}_{\mathsf{OPT}}$ are evaluated only *once*, instead of being evaluated for multiple times when the input queries are executed independently. Intuitively, this is where the savings MQO could bring from. As mentioned earlier, MQO must consider generic directed graphs, possibly with cyclic patterns, which makes it hard to adapt existing techniques for this optimization. Also, the proposed optimization has a unique characteristic that it leverages SPARQL-specific features such as the OPTIONAL clause for query rewriting.

Note that the above rewriting only considers query structures, without considering query selectivity. Suppose we know the selectivity $\alpha(\mathsf{p})$ of each pattern p in the queries, as shown in Figure 3(f). Let us assume a simple cost model that the cost of each query Q or $\mathsf{Q}_{\mathsf{OPT}}$ is equal to the minimum selectivity of the patterns in GP; we ignore for now the cost of OPTIONAL patterns, which is motivated by how *real* SPARQL engines evaluate queries (The actual cost model used in this paper is discussed in Section III-D.). So, the cost for all four queries Q_1 to Q_4 is respectively 4, 2, 4 and 4 (with queries executed on a dataset of size 100). Therefore, executing all queries individually (without optimization) costs $4+2+4+4=14$. In comparison, the cost of the structure-based only optimized query in Figure 3(e) is 9, resulting in a saving of approximately 30%. Now, consider another rewriting in Figure 3(f) that results in from optimization along the second largest common subquery that just contains P$_4$. The cost for this query is only 4, which leads to even more savings, although the rewriting utilizes a smaller common subquery. As this simple example illustrates, it is critical for MQO to construct a cost model that integrates query structure overlap with selectivity estimation.

```
   Input: Set $\mathcal{Q}$ = {Q$_1$, ..., Q$_n$}
   Output: Set $\mathcal{Q}_{\mathsf{OPT}}$ of optimized queries
   // Step 1: Bootstrapping the query optimizer
1  Run $k$-means on $\mathcal{Q}$ to generate a set $\mathcal{M}$ = {M$_1$, ..., M$_k$} of $k$ query
   groups based on query similarity in terms of their predicate sets;
   // Step 2: Refining query clusters
2  for each query group $\mathsf{M} \in \mathcal{M}$ do
3    Initialize a set $\mathcal{C} = \{\mathsf{C}_1, \ldots, \mathsf{C}_{|\mathsf{M}|}\}$ of $|\mathsf{M}|$ clusters;
4    for each query $\mathsf{Q}_i \in \mathsf{M}, 1 \le i \le |\mathsf{M}|$ do $\mathsf{C}_i = \mathsf{Q}_i$;
5    while $\exists$ untested pair $(\mathsf{C}_i, \mathsf{C}_{i'})$ with $\mathcal{J}_{max}(\mathsf{C}_i, \mathsf{C}_{i'})$ do //$\mathcal{J}$:Jaccard
6      Let $\mathcal{Q}^{ii'} = \{\mathsf{Q}_1^{ii'}, \ldots, \mathsf{Q}_m^{ii'}\}$ be the queries of $\mathsf{C}_i \cup \mathsf{C}_{i'}$;
7      Let $\mathcal{S}$ be the top-$s$ most selective triple patterns in $\mathcal{Q}^{ii'}$;
       // Step 2.1: Building compact linegraphs
8      Let $\mu_\cap \leftarrow \mu_1 \cap \mu_2 \ldots \cap \mu_m$ and $\tau = \{\emptyset\}$;
9      for each query $\mathsf{Q}_j^{ii'} \in \mathcal{Q}^{ii'}$ do
10       Build linegraph $\mathcal{L}(\mathsf{Q}_j^{ii'})$ with only the edges in $\mu_\cap$;
11       Keep indegree matrix $m_j^-$, outdegree matrix $m_j^+$ for $\mathcal{L}(\mathsf{Q}_j^{ii'})$;
12     for each vertex $e$ defined in $\mu_\cap$ and $\mu_\cap(e) \neq \emptyset$ do
13       Let $I = m_1^-[e] \cap \ldots \cap m_m^-[e]$ and $O = m_1^+[e] \cap \ldots \cap m_m^+[e]$;
14       if $I = O = \emptyset$ then $\mu_\cap(e) \overset{\mathsf{def}}{=} \emptyset$ and $\tau = \tau \cup \{$triple pattern on $e\}$;
15     for $\mathcal{L}(\mathsf{GP}_j), 1 \le j \le m$ do
16       Prune the $\mathcal{L}(\mathsf{GP}_j)$ vertices not in $\mu_\cap$ and their incident edges;
       // Step 2.2: Building product graphs
17     Build $\mathcal{L}(\mathsf{GP}_p) = \mathcal{L}(\mathsf{GP}_1) \otimes \mathcal{L}(\mathsf{GP}_2) \otimes \ldots \otimes \mathcal{L}(\mathsf{GP}_m)$;
       // Step 2.3: Finding cliques in product graphs
18     $\{K_1, \ldots, K_r\} = AllMaximalClique(\mathcal{L}(\mathsf{GP}_p))$;
19     if $r = 0$ then goto 22;
20     for each $K_i$, $i = 1, 2, \ldots, r$ do
21       find all $K_i' \subseteq K_i$ having the maximal strong covering tree in $K_i$;
22     sort $\mathsf{SubQ} = \{K_1', \ldots, K_t'\} \cup \tau$ in descending order by size;
23     Initialize $K = \emptyset$;
24     for each $q_i \in \mathsf{SubQ}$, $i = 1, 2, \ldots, t + |\tau|$ do
25       if $\mathcal{S} \cap q_i \neq \emptyset$ then Set $K = q_i$ and break
26     if $K \neq \emptyset$ then
27       Let $\mathsf{C}_{tmp} = \mathsf{C}_i \cup \mathsf{C}_{i'}$ and cost($\mathsf{C}_{tmp}$)=cost(sub-query for $K$);
28       if cost($\mathsf{C}_{tmp}$) $\le$ cost($\mathsf{C}_i$) + cost($\mathsf{C}_{i'}$) then
29         Put $K$ with $\mathcal{C}_{tmp}$;
30         remove $\mathsf{C}_i$, $\mathsf{C}_{i'}$ from $\mathcal{C}$ and add $\mathsf{C}_{tmp}$;
   // Step 3: Generating optimized queries
31   for each cluster $\mathsf{C}_i$ in $\mathcal{C}$ do
32     if a clique $K$ is associated with $\mathsf{C}_i$ then
33       Rewrite queries in $\mathsf{C}_i$ using triple patterns in $K$;
34     Output the query into set $\mathcal{Q}_{\mathsf{OPT}}$;
35 return $\mathcal{Q}_{\mathsf{OPT}}$.
```

Fig. 4. Multi-query optimization algorithm

III. The algorithm

Our MQO algorithm, shown in Figure 4, accepts as input a set $\mathcal{Q}$ = {Q$_1$, ..., Q$_n$} of n queries over a graph G. Without loss of generality, assume the sets of variables used in different queries are distinct. The algorithm identifies whether there is a *cost-effective* way to share the evaluation of structurally-overlapping graph patterns among the queries in $\mathcal{Q}$. At a high level, the algorithm works as follows: (1) It partitions the input queries into groups, where queries in the same group are more likely to share common sub-queries that can be optimized through query rewriting; (2) it rewrites a number of **Type 1** queries in each group to their correspondent cost-efficient **Type 2** queries; and (3) it executes the rewritten queries and *distributes* the query results to the original input queries (along with a refinement). Several challenges arise during the above process: (i) There exists an exponential number of ways to partition the input queries. We thus need a heuristic to prune out the space of less optimal partitioning of queries. (ii) We need an efficient algorithm to identify potential common sub-queries for a given query group. And (iii) since different common sub-queries result in different query rewritings, we need a robust cost model to compare candidate rewriting strategies. We describe how we tackle these challenges next.

A. Bootstrapping

Finding structural overlaps for a set of queries amounts to finding the isomorphic subgraphs among the corresponding query graphs. This process is computationally expensive (the problem is NP-hard [4] in general), so ideally we would like to find these overlaps only for groups of queries that will eventually be optimized (rewritten). That is, we want to minimize (or ideally eliminate) the computation spent on identifying common subgraphs for query groups that lead to less optimal MQO solutions. One heuristic we adopt is to quickly prune out subsets of queries that clearly share little in query graphs, without going to the next expensive step of computing their common subqueries; therefore, the group of queries that have few predicates in common will be pruned from further consideration for optimization. We thus define the similarity metric for two queries as the Jaccard similarity of their predicate sets. The rational is that if the Jaccard similarity of two queries is small, their structural overlap in query graphs must also be small; so it is safe to not consider grouping such queries for MQO. We implement this heuristic as a *bootstrap* step in line 1 using k-means clustering (with Jaccard as the similarity metric) for an initial partitioning of the input queries into a set $\mathcal{M}$ of k query groups. Notice that the similarity metric identifies queries with substantial overlaps in their predicate sets, ignoring for now the common sub-structure and the selectivity of these predicates.

B. Refining query clusters

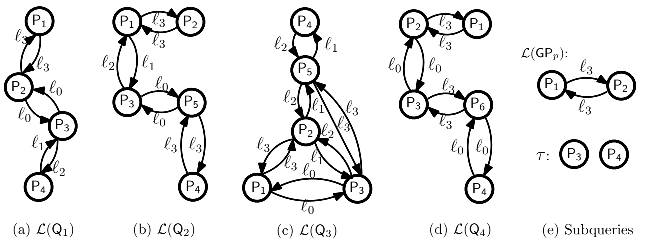
Starting with the k-means generated groups $\mathcal{M}$, we refine the partitioning of queries further based on their structure similarity and the estimated cost. To this end, we consider each query group generated from the k-means clustering $\mathsf{M} \in \mathcal{M}$ in isolation (since queries across groups are guaranteed to be sufficiently different) and perform the following steps: In lines 5–30, we (incrementally) *merge* structurally similar queries within M through *hierarchical clustering* [14], and generate query clusters such that each query cluster is optimized together (*i.e.,* results in one **Type 2** query). Initially, we create one *singleton* cluster C_i for each query Q_i of M (line 4). Given two clusters C_i and $\mathsf{C}_{i'}$, we have to determine whether it is more cost-efficient to merge the two query clusters into a single cluster (*i.e.,* a single **Type 2** query) than to keep the two clusters separate (*i.e.,* executing the corresponding two queries independently). From the previous iteration, we already know the cost of the optimized queries for each of the C_i and $\mathsf{C}_{i'}$ clusters. To determine the cost of the merged cluster, we have to compute the query that merges all the queries in C_i and $\mathsf{C}_{i'}$ through rewriting; which requires us to compute the common substructure of all these queries, and to estimate the cost of the rewritten query generated from the merged clusters. For the cost computation, we do some preliminary work here (line 7) by identifying the most selective triple patterns from the two clusters (selectivity is estimated by [33]). Note that our refinement of M might lead to more than one queries; one for each *cluster* of M, in the form of either **Type 1** or **Type 2**.

Finding common substructures: While finding the maximum common subgraph for two graphs is known to be NP-hard [4], the challenge here is asymptotically harder as it requires finding the largest common substructures for *multiple* graphs. Existing solutions on finding common subgraphs also assume *untyped* edges and nodes in *undirected* graphs. However, in our case the graphs represent queries, and different triple patterns might correspond to different semantics (*i.e.,* typed and directed). Thus, the predicates and the constants associated with nodes must be taken into consideration. This mix of typed, constant and variable nodes/edges is not typical in classical graph algorithms, and therefore existing solutions can not be directly applied for query optimization. We therefore propose an efficient algorithm to address these challenges.

In a nutshell, our algorithm follows the principle of finding the maximal common edge subgraphs (MCES) [25], [37]. Concisely, three major sub-steps are involved (steps 2.1 to 2.3 in Figure 4): (a) transforming the input query graphs into the equivalent linegraph representations; (b) generating a product graph from the linegraphs; and (c) executing a tailored clique detection algorithm to find the maximal cliques in the product graph (a maximal clique corresponds to an MCES). We describe these sub-steps in details next.

Step 2.1: Building compact linegraphs: The linegraph $\mathcal{L}(\mathsf{G})$ of a graph G is a directed graph built as follows. Each node in $\mathcal{L}(\mathsf{G})$ corresponds to an edge in G, and there is an edge between two nodes in $\mathcal{L}(\mathsf{G})$ if the corresponding edges in G share a common node. Although it is straightforward to transform a graph into its linegraph representation, the context of MQO raises new requirements for the linegraph construction. We represent the linegraph of a query graph pattern in a 4-tuple, defined as $\mathcal{L}(\mathsf{G}) = (\mathcal{V}, \mathcal{E}, \pi, \omega)$. During linegraph construction, besides the inversion of nodes and edges for the query graph, our transformation also assigns to each edge in the linegraph one of 4 labels ($\ell_0 \sim \ell_3$). Specifically, for two triple patterns, there are 4 possible joins between their subjects and objects (ℓ_0 = subject-subject, ℓ_1 = subject-object, ℓ_2 = object-subject, ℓ_3 = object-object). The assignment of labels on linegraph edges captures these four join types (useful for pruning and will become clear shortly). Figure 5 (a)-(d) shows the linegraphs for the queries in Figure 3(a)-(d).

The classical solution for finding common substructures of input graphs requires building Cartesian products on their



(a) $\mathcal{L}(\mathsf{Q}_1)$ (b) $\mathcal{L}(\mathsf{Q}_2)$ (c) $\mathcal{L}(\mathsf{Q}_3)$ (d) $\mathcal{L}(\mathsf{Q}_4)$ (e) Subqueries

Fig. 5. (a)–(d) linegraphs, (e) their common substructures

linegraphs. This raises challenges in scalability when finding the maximum common substructure for *multiple* queries in *one* shot. To avoid the foreseeable explosion, we propose fine-grained optimizations (lines 8–16) to keep linegraphs as small as possible so that only the most *promising* substructures would be transformed into linegraphs, with the rest being temporarily *masked* from further processing.

To achieve the above, queries in $\mathcal{Q}^{ii'}$ pass through a two-stage optimization. In the first stage (lines 8–11), we identify (line 8) the common predicates in $\mathcal{Q}^{ii'}$ by building the intersection $\mu_\cap$ for all the labels defined in the μ's (recall that function μ assigns predicate names to graph edges). Predicates that are not common to all queries can be safely pruned, since by definition they are not part of any common substructure, *e.g.,*P_5 and P_6 in Figure 3. While computing the intersection of predicates, the algorithm also checks for *compatibility* between the corresponding subjects and objects, so that same-label predicates with different subjects/objects are not added into $\mu_\cap$. In addition, we maintain two adjacency matrices for a linegraph $\mathcal{L}(\mathsf{GP})$, namely, the indegree matrix m^- storing all incoming, and the outdegree matrix m^+ storing all outgoing edges from $\mathcal{L}(\mathsf{GP})$ vertices. For a vertex v, we use $m^-[v]$ and $m^+[v]$, respectively, to denote the portion of the adjacency matrices storing the incoming and outgoing edges of v. For example, the adjacency matrices for vertex P_3 in linegraph $\mathcal{L}(\mathsf{Q}_1)$ of Figure 5 are $m_1^+[\mathsf{P}_3] = [\emptyset, \ell_0, \emptyset, \ell_2, \emptyset, \emptyset]$, $m_1^-[\mathsf{P}_3] = [\emptyset, \ell_0, \emptyset, \ell_1, \emptyset, \emptyset]$, while for linegraph $\mathcal{L}(\mathsf{Q}_2)$ they are $m_2^+[\mathsf{P}_3] = [\ell_2, \emptyset, \emptyset, \emptyset, \ell_0, \emptyset]$, $m_2^-[\mathsf{P}_3] = [\ell_1, \emptyset, \emptyset, \emptyset, \ell_0, \emptyset]$.

In the second stage (lines 12–16), to further reduce the size of linegraphs, for each linegraph vertex e, we compute the Boolean intersection for the $m^-[e]$'s and $m^+[e]$'s from all linegraphs respectively (line 13). We also prune e from $\mu_\cap$ if both intersections equal $\emptyset$ and set aside the triple pattern associated with e in a set τ (line 14). Intuitively, this optimization acts as a *look-ahead* step in our algorithm, as it quickly detects the cases where the common sub-queries involve only one triple pattern (those in τ). Moreover, it also improves the efficiency of the clique detection (step 2.2 and 2.3) due to the smaller sizes of input linegraphs. Going back to our example, just by looking at the m_1^-, m_1^+, m_2^-, m_2^+, it is easy to see that the intersection $\cap m_i^+[\mathsf{P}_3] = \cap m_i^-[\mathsf{P}_3] = \emptyset$ for all the linegraphs of Figure 5(a)-(d). Therefore, our optimization temporarily masks P_3 (so as P_4) from the expensive clique detection in the following two steps.

Step 2.2: Building product graphs: The product graph $\mathcal{L}(\mathsf{GP}_p) := (\mathcal{V}_p, \mathcal{E}_p, \pi_p, \omega_p)$ of two linegraphs, $\mathcal{L}(\mathsf{GP}_1) := (\mathcal{V}_1, \mathcal{E}_1, \pi_1, \omega_1)$ and $\mathcal{L}(\mathsf{GP}_2) := (\mathcal{V}_2, \mathcal{E}_2, \pi_2, \omega_2)$, is denoted as $\mathcal{L}(\mathsf{GP}_p) := \mathcal{L}(\mathsf{GP}_1) \otimes \mathcal{L}(\mathsf{GP}_2)$. The vertices $\mathcal{V}_p$ in $\mathcal{L}(\mathsf{GP}_p)$ are defined on the Cartesian product of $\mathcal{V}_1$ and $\mathcal{V}_2$. In order to use product graphs in MQO, we optimize the standard definition with the additional requirement that vertices paired together must have the same label (*i.e.,* predicate). That is, $\mathcal{V}_p := \{(v_1, v_2) \mid v_1 \in \mathcal{V}_1 \wedge v_2 \in \mathcal{V}_2 \wedge \pi_1(v_1) = \pi_2(v_2)\}$, with the labeling function defined as $\pi_p := \{\pi_p(v) \mid \pi_p(v) = \pi_1(v_1)$, with $v = (v_1, v_2) \in \mathcal{V}_p\}$. For the product edges, we use the standard definition which creates edges in the product graph between two vertices (v_{1i}, v_{2i}) and (v_{1j}, v_{2j}) in $\mathcal{V}_p$ if either (i) the same edges (v_{1i}, v_{1j}) in $\mathcal{E}_1$, and (v_{2i}, v_{2j}) in $\mathcal{E}_2$ exist; or (ii) no edges connect v_{1i} with v_{1j} in $\mathcal{E}_1$, and v_{2i} with v_{2j} in $\mathcal{E}_2$. The edges due to (i) are termed as *strong* connections, while those for (ii) as *weak* connections [37].

Since the product graph for two linegraphs conforms to the definition of linegraph, we can recursively build the product for multiple linegraphs (line 17). Theoretically, there is an exponential blowup in size when we construct the product for multiple linegraphs. In practice, thanks to our optimizations in Steps 2.1 and 2.2, our algorithm is able to accommodate tens to hundred of queries, and generates the product graph efficiently (which will be verified through Section V). Figure 5(e) shows the product linegraph $\mathcal{L}(\mathsf{GP}_p)$ for the running example.

Step 2.3: Finding Cliques in product graphs: A (maximal) clique with a strong covering tree (a tree only involving strong connections) in the product graph equals to an MCES – a (maximal) common sub-query in essence. In addition, we are interested in finding cost-effective common sub-queries. To verify if the found common sub-query is selective, it is checked with the set $\mathcal{S}$ (from line 7) of selective query patterns.

In the algorithm, we proceed by finding all maximal cliques in the product graph (line 18), a process for which many efficient algorithms exist [16], [21], [35]. For each discovered clique, we identify its sub-cliques with the maximal strong covering trees (line 21). For the $\mathcal{L}(\mathsf{GP}_p)$ in Figure 5(e), it results in one clique (itself): *i.e.,* $K_1' = \{\mathsf{P}_1, \mathsf{P}_2\}$. As the cost of sub-queries is another dimension for query optimization, we look for the substructures that are both large in size (*i.e.,* the number of query graph patterns in overlap) and correspond to selective common sub-queries. Therefore, we first sort SubQ (contributed by K's and τ, line 22) by their sizes in descending order, and then loop through the sorted list from the beginning and stop at the *first* substructure that intersects $\mathcal{S}$ (lines 22–25), *i.e.,* P_4 in our example. We then merge (if it is cost-effective, line 28) the queries whose common sub-query is reflected in K and also merge their corresponding clusters into a new cluster (while remembering the found common sub-query) (lines 26–30). The algorithm repeats lines 5–30 until every possible pair of clusters have been tested and no new cluster can be generated.

C. Generating optimized queries and distributing results

After the clusters are finalized, the algorithm rewrites each cluster of queries into one query and thus generates a set of rewritings $\mathcal{Q}_{\mathsf{OPT}}$ (lines 31–34). The result from evaluating $\mathcal{Q}_{\mathsf{OPT}}$ over the data is a superset of evaluating the input queries $\mathcal{Q}$

(more expositions in section III-E). Therefore, we must filter and distribute the results from the execution of $\mathcal{Q}_{\mathsf{OPT}}$. This necessitates one more step of parsing the result of $\mathcal{Q}_{\mathsf{OPT}}$ (refer to Figure 1(c)), which checks each row of the result against the RD of each query in $\mathcal{Q}$. Note that the result description $\mathsf{RD}_{\mathsf{OPT}}$ is always the union of RD_is from the queries being optimized, and we record the mappings between the variables in the rewritings and the variables in the original input queries. As in Figure 1(c), the result of a **Type 2** query might have *empty* (*null*) columns corresponding to the variables from the OPTIONAL clause. Therefore, a row in the result of $\mathsf{RD}_{\mathsf{OPT}}$ might not conform to the description of every RD_i. The goal of parsing is to identify the valid overlap between each row of the result and the individual RD_i, and return to each query the result it is supposed to get. To achieve this, the parsing algorithm performs a Boolean intersection between each row of result and each RD_i: if the columns of this row corresponding to those columns of RD_i are not null, the algorithm distributes the corresponding part of this row to Q_i as one of its query answers. This step iterates over each row and each Q_i that composed the **Type 2** query. The parsing on the results of $\mathcal{Q}_{\mathsf{OPT}}$ only requires a linear scan on the results to the rewritten query. Therefore, it can be done on-the-fly as the results of $\mathcal{Q}_{\mathsf{OPT}}$ is streamed out from the evaluation.

D. Cost model for SPARQL MQO

The design of our cost module is motivated by the way in which a SPARQL query is evaluated on popular RDF stores. This includes a well-justified principle that the most selective triple pattern is evaluated first [33] and that the $\mathsf{GP}_{\mathsf{OPT}}$ clause is evaluated on the result of GP (for the fact that $\mathsf{GP}_{\mathsf{OPT}}$ is a left-join). This suggests that a good optimization should keep the result cardinality from the common sub-query as small as possible for two reasons: 1) The result cardinality of a **Type 2** SPARQL query is upper bounded by result cardinality of its GP clause since $\mathsf{GP}_{\mathsf{OPT}}$s are simply left-joins; 2) Intermediate result from evaluating the GP clause is not well indexed, which implies that a non-selective GP will result in significantly more efforts in processing the corresponding rewriting $\mathsf{GP}_{\mathsf{OPT}}$s.

In [33], the authors discussed the selectivity estimation for the conjunctive Basic Graph Patterns (BGP). In a nutshell, given a triple pattern $\mathsf{t} = (\mathsf{s}\ \mathsf{p}\ \mathsf{o})$, where each entry could be bound or unbound, its selectivity is estimated by $sel(t) = sel(s) \times sel(p) \times sel(o)$. sel is the selectivity estimation function, whose value falls in the interval of $[0, 1]$. Specifically, for unbound variable, its selectivity equals 1. For bound variables/constants, depending on whether it is a subject, predicate or object, different methods (*e.g.,* [33]) are used to implement sel. Notice that the formula implicitly assumes statistical independence for the subject, predicate and object; thus is an approximation. Pre-computed statistics of the dataset are also required. For a join between two triple patterns, independence assumption is adopted [33]. However, in practice, such estimation is not accurate enough for optimizing complex queries. The culprit comes from the fact that as the number of joins increases, the accuracy of the estimated selectivity drops quickly, resulting in a very loose estimation [19].

With the above limitations in mind, we propose a cost function for conjunctive SPARQL query. It has a simple design and roots on the well justified principle in query optimization that the selective triple patterns have higher priorities in evaluation. Our cost model is an incarnation of this intuition, as in Formula 1:

$$\mathsf{Cost}(\mathsf{Q}) = \begin{cases} \mathsf{Min}(sel(t)) & \mathsf{Q} \text{ is a } \textbf{Type 1} \text{ query, } t \in \mathsf{GP} \\ \mathsf{Min}(sel(t)) + \Delta & \mathsf{Q} \text{ is a } \textbf{Type 2} \text{ query, } t \in \mathsf{GP} \end{cases} \tag{1}$$

For a **Type 1** conjunctive query, Formula 1 simply returns the selectivity for the most selective triple pattern in the query graph GP as the cost of evaluating Q. For a **Type 2** query, the cost is the summation of the cost on evaluating the common graph pattern GP and the cost on the evaluating the OPTIONALs, *i.e.,* the cost denoted by Δ. We extrapolate (backed by a comprehensive empirical study on three different RDF query engines) that Δ is a hidden function of (i) the cost of GP; (ii) the number of OPTIONALs; and (iii) the cost of the query pattern of each $\mathsf{GP}_{\mathsf{OPT}}$. However, we observed empirically that when the cost of GP is small (being selective), Δ would be a trivial value and Cost(Q) is mostly credited to the evaluation of GP. Hence, we can approximate Cost(Q) with the cost of GP in such cases. We show (experimentally) that using our cost model to choose a good common substructure can consistently improve the performance of query evaluation over the pure structure-based optimization (*i.e.,* without considering the evaluation cost of common sub-queries) on different RDF stores.

Finally, notice that the proposed cost function requires using the pre-computed statistics of the RDF dataset to estimate the selectivity of triple patterns. Therefore, it requires to collect some statistics from the dataset. This mainly includes (i) building the histogram for distinct predicates in the dataset and (ii) that for each disparate predicate, we build histograms for the subjects and objects attached to this predicate in the dataset. In practice, for some RDF stores, like Jena, part of such statistics (*e.g.,* the histogram of predicates) is provided by the SPARQL query optimizer and is accessible for free; for the others, *e.g.,* Virtuoso and Sesame, the statistics of the dataset need to be collected in a preprocessing step.

E. Completeness and soundness of our MQO algorithm

Completeness: Suppose a **Type 2** rewritten query $\mathsf{Q}_{\mathsf{OPT}}$ optimizes a set of n **Type 1** queries, *i.e.,* $\mathcal{Q} = \{\mathsf{Q}_1, \mathsf{Q}_2, \ldots, \mathsf{Q}_n\}$. Without loss of generality, denote the common relation (*i.e.,* the common sub-query) used in $\mathsf{Q}_{\mathsf{OPT}}$ as GP and its outer join relations (*i.e.,* the OPTIONALs) as GP_i $(i = 1, 2, \ldots, n)$. As we only consider conjunctive queries as input, hence by construction $\mathcal{Q} = \cup_{i=1}^{n} \mathsf{GP} \bowtie \mathsf{GP}_i$ and $\mathsf{Q}_{\mathsf{OPT}} = \cup_{i=1}^{n} \mathsf{GP} \ltimes \mathsf{GP}_i$. By the definition of left outer join $\ltimes$, $\mathsf{GP} \bowtie \mathsf{GP}_i \subseteq \mathsf{GP} \ltimes \mathsf{GP}_i$ for any i. It follows $\mathcal{Q} \subseteq \mathsf{Q}_{\mathsf{OPT}}$ in terms of query results.

Soundness: Soundness requires $\mathcal{Q} \supseteq \mathsf{Q}_{\mathsf{OPT}}$. This is achieved by evaluating the results from $\mathsf{Q}_{\mathsf{OPT}}$ and distributing the matched results to correspondent queries in $\mathcal{Q}$ (section III-C). As such, false positives are discarded and the remainings are valid bindings for one or more graph patterns in $\mathcal{Q}$. Therefore, $\mathcal{Q} \supseteq \mathsf{Q_{OPT}}$ in terms of results after the refining step.

Completeness and soundness together guarantee that the final answers resulted by our MQO techniques are equivalent to the results from evaluating queries in $\mathcal{Q}$ independently.

IV. Extensions

For the ease of presentation, the input queries discussed so far are **Type 1** queries using constants as their predicates. It is interesting to note that with some minimal modifications to the algorithm and little preprocessing of the input, the algorithm in Figure 4 can optimize more general SPARQL queries. Here, we introduce two simple yet useful extensions: (i) optimizing input queries with variables as the predicates; and (ii) optimizing input queries of **Type 2** (*i.e.,* with OPTIONALs).

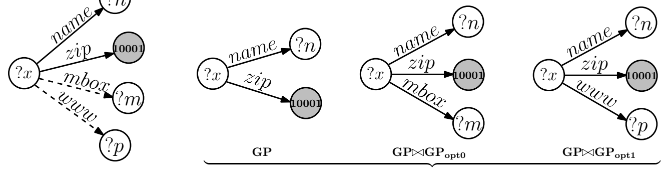
A. Queries with variable predicates

We treat variable predicates slightly differently from the constant predicates when identifying the structural overlap of input queries. Basically, a variable predicate from one query can be matched with any variable predicate in another query. In addition, each variable predicate of a query will correspond to one variable vertex in the linegraph representation, but the main flow of the MQO algorithm remains the same.

B. TYPE 2 queries

Our MQO algorithm takes a batch of **Type 1** SPARQL queries as input and rewrites them to another batch of **Type 1** and **Type 2** queries. It can be extended to optimize a batch of input queries with both **Type 1** and **Type 2** queries.

To this end, it requires a preprocessing step on the input queries. Specifically, by the definition of left-join, a **Type 2** input query will be rewritten into its equivalent **Type 1** form, since our MQO algorithm only works on **Type 1** input queries. The equivalent **Type 1** form of a **Type 2** query GP (OPTIONAL $\mathsf{GP_{OPT}})^+$) consists two sets of queries: (i) a **Type 1** query solely using the GP as its query graph pattern; and (ii) the queries by replacing the left join(s) with inner join(s) between GP and each of the $\mathsf{GP_{OPT}}$ from the OPTIONAL, *i.e.,* $\cup \mathsf{GP} \bowtie \mathsf{GP_{OPT}}$. For example, to strip off the OPTIONALs in the **Type 2** query in Figure 6(a), applying the above preprocessing will result in a group of three **Type 1** rewritings as in Figure 6(b).



(a) A **Type** 2 query (b) Equivalent **Type** 1 rewritten queries

Fig. 6. A **Type 2** query to its equivalent **Type 1** form

By applying the above transformation to all **Type 2** queries in the input and then passing the batch of queries to algorithm in Figure 4 for optimization, we can handle **Type 2** queries seamlessly. Finally, the result to the original **Type 2** query can be generated through the union of the results, from the transformed **Type 1** queries after MQO.

V. Experimental Evaluation

We implemented all algorithms in C++ and performed an extensive experimental evaluation using a 64-bit Linux machine with a 2GHz Intel Xeon(R) CPU and 4GB of memory.

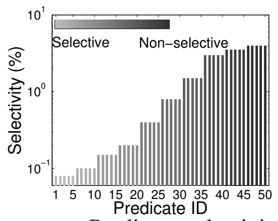


Fig. 7. Predicate selectivity

Datasets: Our evaluation is based on LUBM benchmark. The popular benchmark models universities with students, departments, *etc.*, using only 18 predicates [11]. This limits the complexity of queries we can evaluate (similar limitations in [5], [28]), and results in queries with considerable overlaps (which favors MQO but is not very realistic). Thus, we extended the LUBM data generator, and added a random subset from 50 new predicates to each person in the dataset, where predicate selectivity follows the distribution in Figure 7. Therefore, given the number of triples N in a dataset D, the number of times that a predicate appears in D (dubbed its frequency) is precisely its selectivity multiplied by N.

RDF Stores: We experimented with three popular RDF stores: Jena TDB 0.85, OpenLink Virtuoso 6.01, and Sesame Native 2.0. Due to space constraints, we analyze mainly the experiments with Jena TDB. Results for the other two stores are highly consistent with the results from Jena TDB. For all stores, we created full indexes using the technique in [39]. For Virtuoso, we also built bitmap indexes as reported in [3].

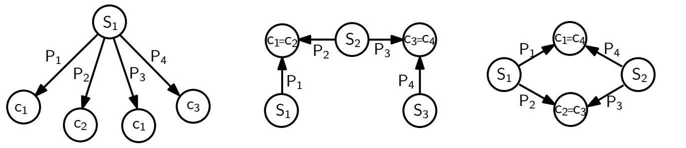
Metrics: For all experiments, we measure the number of optimized queries and their end-to-end evaluation time, including query rewriting, execution and result distribution. We compare our MQO algorithm with the evaluation without any optimization (*i.e.,* No-MQO), and the approach with structure-only optimization (*i.e.,* MQO-S). To realize the latter strategy as a baseline solution, we need to turn off all the cost-based comparisons in Figure 4. Specifically, in line 24 of Figure 4, instead of walking through the set of SubQ (which correspond to different common substructures), structure-based optimization (*i.e.,* MQO-S) simply returns the the largest clique (*i.e.* the largest common subquery) for optimization.

Comparing MQO with MQO-S illustrates the benefits of blending structured-based with the cost-based optimization versus a purely structural approach. In the algorithms, we use the suffix -C to denote the cost by rewriting queries (*e.g.,* MQO-C) and the suffix -P to denote the cost by parsing and distributing the query results (*e.g.,* MQO-P). For finding cliques, we customized the Cliquer library [21], which is an efficient implementations for clique detection. For selectivity estimation, we implemented the technique in [33]. All experiments are performed using cold caches, and the bootstrapping parameter k in the k-means algorithm is set to $k = \lceil |\mathcal{Q}|/40 \rceil$. Table I provides the summary along with ranges and the default values used for various parameters in our experiments.

Queries: LUBM has only 14 SPARQL queries, which have limited variance in both structure and evaluation cost. Therefore,

Parameter	Symbol	Default	Range
Dataset size	D	4M	3M to 9M
Number of queries	$\lvert\mathcal{Q}\rvert$	100	60 to 160
Size of query (num of triple patterns)	$\lvert\mathsf{Q}\rvert$	6	5 to 9
Number of seed queries	κ	6	5 to 10
Size of seed queries	$\lvert q_{\text{cmn}}\rvert$	$\sim \lvert\mathsf{Q}\rvert/2$	1 to 5
Max selectivity of patterns in Q	$\alpha_{max}(\mathsf{Q})$	random	0.1% to 4%
Min selectivity of patterns in Q	$\alpha_{min}(\mathsf{Q})$	1%	0.1% to 4%

TABLE I
PARAMETER TABLE.



(a) Star (b) Chain (c) Circle
Fig. 8. Three basic query patterns

we created a module to generate query sets $\mathcal{Q}$ with varying sizes $|\mathcal{Q}|$, where we generated queries that combine *star*, *chain*, and *circle* pattern structures. In addition, we attached to each person (as a subject) in the LUBM data a (random) subset of 50 new predicates $\mathsf{P}_1 \sim \mathsf{P}_{50}$. In particular, we customized the data generator of LUBM in such a way that whenever a triple ($\mathsf{s}\ \mathsf{P}_i\ \mathsf{c}_i$) is added to the data, c_i is an integer value serving as the object of this triple and it is set to the number of predicate P_i *existed in the dataset so far*. Therefore, triples with different predicates could join on their subjects or objects, so as the triple patterns in the query, which we will detail next.

Our query generator utilizes the aforementioned patterns in the customized data to compose queries. Specifically, we ensure that the queries have reasonably high randomness in structure (such that they are not replicas of limited query templates) and reasonable variances in selectivity (such that any predicate could be part of a query regardless of the structure). To this end, we first show how to compose three basic query patterns: star, chain and circle with a set of four basic triple patterns; see Figure 8 (a) – (c). The star and the chain can be built with any number of triple patterns while the circle can only be built with an even number of triple patterns.

To blend the three basic patterns into one query Q with $|\mathsf{Q}|$ triple patterns, the generator first randomly distributes the set of triple patterns into k groups of subqueries (k is a random integer, $k \in (0, |\mathsf{Q}|)$), with each subquery randomly composing one of the three basic patterns, *i.e.,* star, chain and if possible, circle. To ensure Q to be conjunctive, the generator then makes equal the (randomly) chosen pairs of subjects and/or objects from the k subqueries by unifying their variable names or binding them to the same constant. This concludes composing the structure of Q. Finally, to ensure that Q conforms to the selectivity requirement posed by a specific experiment (refer to Table I), the generator fills in the structure of Q with the predicates that would make Q a legitimate query.

In the experiments, a group of queries in $\mathcal{Q}$ were rendered to share a common seed sub-query q_{cmn}. The generator first constructs q_{cmn} and the remaining portion of the queries independently. Then, by equaling the subjects and/or objects of these two sub-queries, the generator propagates q_{cmn} over the group such that q_{cmn} joins with each of the sub-queries in the group. In addition, individual query sizes $|\mathsf{Q}|$ can be varied where the probability of a predicate being part of a query conforms to its frequency in the dataset. We ensure that 90% of the queries in $\mathcal{Q}$ are amenable to optimization, while 10% are not. We use a parameter κ to determine seed queries that will be used to generate the queries in this 90%. For a given κ, κ seed-groups are generated, each corresponding to $\lceil(90/\kappa)\rceil$% of queries in $\mathcal{Q}$. The seed in each seed-group is what our algorithm will (hopefully) discover.

In short, we generated datasets and queries with various size, complexity, and statistics to evaluate the proposed MQO algorithm in a comprehensive way.

A. Experimental Results

The objective of our experiments is to evaluate: (i) how much each step of MQO (from bootstrapping step to cost estimation) contributes to the optimization, *i.e.,* drop in performance due to omission of each step; (ii) whether the combination of structure and cost-based optimization *consistently* outperforms purely structure-based optimizations; (iii) how well Algorithm MQO optimizes its alternatives, including the comparison with the baseline approach without any optimization, in *every* experimental setting; and (iv) whether Algorithm MQO consistently works across RDF stores.

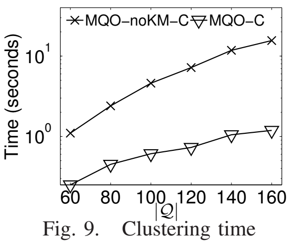


Fig. 9. Clustering time

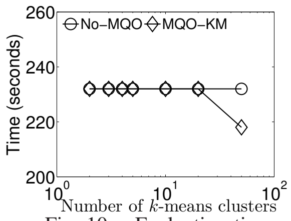


Fig. 10. Evaluation time

Impact of each MQO step: We start with an experiment to illustrate the benefit of bootstrapping MQO using k-means. Figure 9 shows the cost of hierarchical clustering in Step 2 of MQO with (MQO-C) and without (MQO-noKM-C) bootstrapping. The figure shows an order of magnitude difference between the MQO-C and MQO-noKM-C, since without bootstrapping Step 2 of MQO requires $\mathcal{O}((|\mathsf{Q}| \times |\mathcal{Q}|)^2)$ pairwise checks between all the queries in the input set $\mathcal{Q}$. The next experiment, in Figure 10, illustrates algorithm MQO-KM which after Step 1 of MQO, it finds the common substructures for the *coarse-grained* groups that result in from k-means and then performs Step 3 (*i.e.,* MQO-KM does not perform hierarchical clustering in Step 2). The figure shows that the resulting optimization has limited (less than 10%) to no benefits in evaluation time, when compared with the case of having no optimizations (No-MQO). This is because k-means ignores query structures and relies solely on the predicate names to determine groups. Therefore, the fine-grained groups that result in from hierarchical clustering (in Step 2) are *necessary* for the considerable savings (as illustrated in the following experiments) in terms of evaluation times.

Varying $|\mathcal{Q}|$: We study scalability w.r.t. the cardinality $|\mathcal{Q}|$ of the query set $\mathcal{Q}$, for which we vary from 60 to 160 queries, by an increment of 20. As Figure 11 shows, both MQO and MQO-S are successful in identifying common substructures,

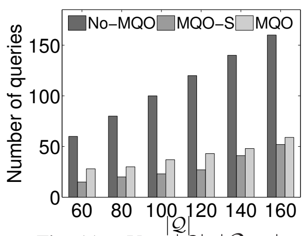


Fig. 11. Vary $|\mathcal{Q}|$: $|\mathcal{Q}_{\text{OPT}}|$

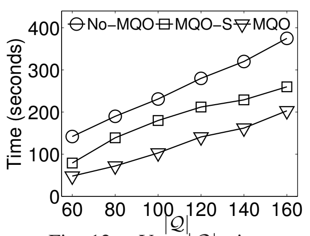


Fig. 12. Vary $|\mathcal{Q}|$: time

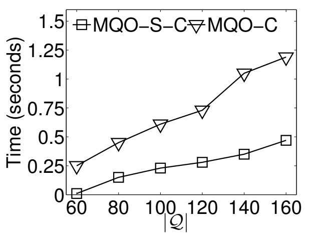


Fig. 13. Clustering cost

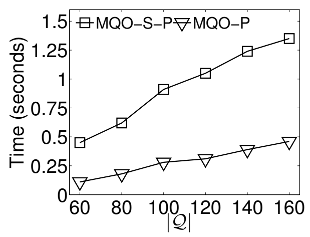


Fig. 14. Parsing cost

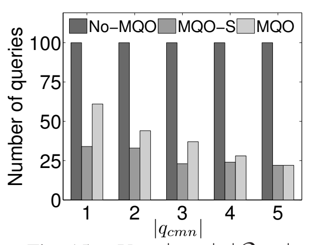


Fig. 15. Vary $|q_{\text{cmn}}|$: $|\mathcal{Q}_{\text{OPT}}|$

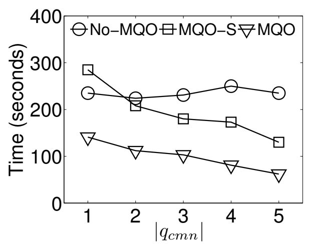


Fig. 16. Vary $|q_{\text{cmn}}|$: time

the former resulting in up to 60% savings and the latter having up to 80% savings in terms of the number of queries, compared to No-MQO. However, in terms of evaluation times (see Figure 12), MQO-S results in *less* savings than MQO, with the former achieving up to 45%, and the latter up to 60% savings in evaluation times, when compared to No-MQO. So MQO is more efficient, despite generating a larger number of optimized queries than MQO-S. The following example, along with the example in Figure 3, illustrates this situation. Consider a set of queries $\mathcal{Q}$, such that (i) predicate p_{cmn} is common to all the queries in $\mathcal{Q}$; (ii) predicate p_1 is common to the subset $\mathcal{Q}_1 \subset \mathcal{Q}$; and (iii) predicate p_2 is common to the subset of queries in $\mathcal{Q}_2 \subset \mathcal{Q}$, with $\mathcal{Q}_1 \cap \mathcal{Q}_2 = \emptyset$. MQO-S looks only at the structure and thus it may opt to generate a single optimized query for $\mathcal{Q}$ with $q_{\text{cmn}} = p_{\text{cmn}}$. If predicate p_{cmn} is not selective, while predicates p_1 and p_2 are highly selective, then MQO will generate two different optimized queries, one for set $\mathcal{Q}_1$ and involving q_1, and one for set $\mathcal{Q}_2$ and involving p_2. As this simple example illustrates, MQO-S can generate fewer but cost-wise less optimized queries when compared with MQO; which is exactly the pattern in Figure 12.

Next, we further analyze the evaluating cost spent on clustering/rewriting the queries, and distributing the final results. In Figure 13, we report the clustering time, which includes both the bootstrapping k-means clustering and the hierarchical clustering that relies on finding common substructures. Notice that MQO requires more time than MQO-S. This is because (i) MQO involves an additional check on the selectivity; and (ii) queries with non-selective common subqueries are recycled into the pool of clusters by MQO, leading to more rounds of comparisons and thus a slower convergence. Contrarily, since the common subqueries rewritten by MQO-S are on average less selective, parsing and distributing these results inevitably requires more effort, as in Figure 14. Nevertheless, clustering and parsing times are a small fraction of the total evaluating cost (less than 2% in the worst case). In the remaining experiments, we only report the end-to-end evaluating cost.

Varying $|q_{\text{cmn}}|$: Here, we study the impact on optimization of the size $|q_{\text{cmn}}|$ of the common subquery, i.e., the size of seed queries. At iteration i we make sure that for the queries in the same group of $\mathcal{Q}$, we have $|q_{\text{cmn}}| = i$. Figure 15 shows the number of optimized queries generated by MQO-S and MQO. Notice that the number of optimized queries is reduced (optimization improves) as $|q_{\text{cmn}}|$ increases . This is because, as the maximum size of each query is kept constant, the more $|q_{\text{cmn}}|$ increases the more the generated queries become similar (less randomness in query generation). Therefore, more queries are clustered and optimized together. Like before, MQO-S is more aggressive and results in less queries compared to MQO. But, like before, Figure 16 shows that MQO is *always* better and results in optimized queries whose evaluation time is half less than MQO-S and up to 75% less than No-MQO. Notice in the figure that for small values of $|q_{\text{cmn}}|$, MQO-S performs worse than No-MQO. Intuitively, the more selective GP is in a **Type 2** optimized query, the less *work* a SPARQL query engine needs to do to evaluate the GP_{OPT} terms in the OPTIONAL of the query. MQO-S relies only on the structural similarity, while ignoring predicate selectivity, negatively influences the overall evaluation time for the optimized query to the point that any benefits from the optimization are alleviated by the extra cost of evaluating the OPTIONAL terms.

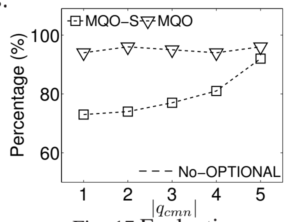


Fig. 17.Evaluating q_{cmn}

MQO combines structured and cost optimization and does not suffer from these limitations. This is evident in Figure 17, which plots the percentage of the evaluation time of the optimized query that is spent evaluating q_{cmn}. By carefully selecting the common subquery q_{cmn}, MQO results in optimized queries whose evaluation time goes mostly (more than 90%) into evaluating q_{cmn} (while less than 10% goes to evaluating OPTIONAL terms). In contrast, MQO-S results in queries whose large extent of evaluation time goes into evaluating OPTIONAL terms (when $|q_{\text{cmn}}| = 1$ this is almost 30%). Things improve for MQO-S as the size of q_{cmn} increases, but still MQO retains the advantage of selecting substructures not just based on their size, but also on their selectivity, and therefore overall evaluation times are still much better.

Varying κ: In Figures 18 and 19, we analyze the impact of the number κ of seed queries on the optimization, by varying κ from 5 to 10. Figure 18 shows that as κ increases, less queries can be optimized by both MQO-S and MQO, which resulted in more rewritten queries. Not surprisingly, a larger κ increases query diversity and reduces the potential for optimization. This affects evaluation times, but MQO is still the best of the three.

Varying $|\text{Q}|$: In Figures 20 and 21, we study the impact of

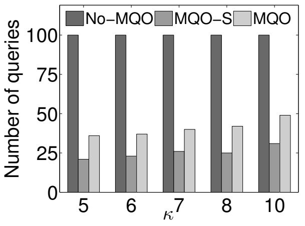


Fig. 18. Vary κ: $|\mathcal{Q}_{\mathsf{OPT}}|$

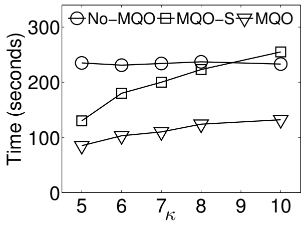


Fig. 19. Vary κ: time

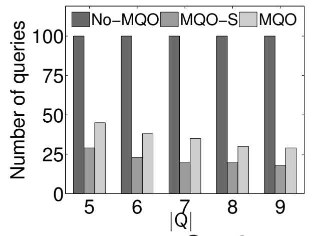


Fig. 20. Vary $|\mathsf{Q}|$: $|\mathcal{Q}_{\mathsf{OPT}}|$

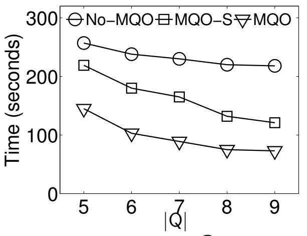


Fig. 21. Vary $|\mathsf{Q}|$: time

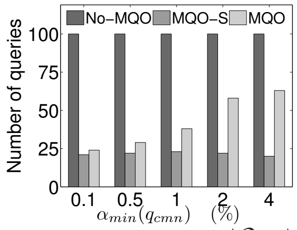


Fig. 22. Vary α_{min}: $|\mathcal{Q}_{\mathsf{OPT}}|$

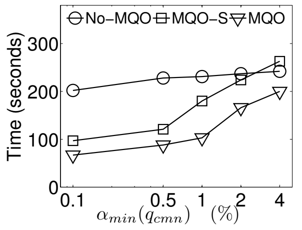


Fig. 23. Vary α_{min}: time

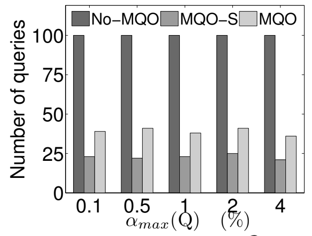


Fig. 24. Vary α_{max}: $|\mathcal{Q}_{\mathsf{OPT}}|$

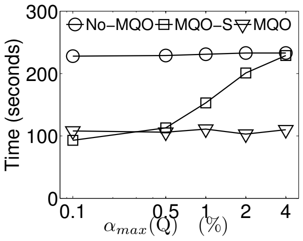


Fig. 25. Vary α_{max}: time

query size, which we increase from 5 to 9 predicates in GP of a query Q. For this experiment we *keep the* $|q_{cmn}|/|\mathsf{Q}|$ *a rough constant and equal to* 0.5. So, the increase in query size does not result in a significant change in query overlap (or potential for optimization). Since the size of a query increases, there is higher chance for the query generator to assign it a selective predicate, which in turn affects the evaluation times. As a result, Figure 21 shows that the evaluation time of No-MQO decreases with the query size. Clearly, MQO still provides savings in evaluation time, ranging from 40% to 70%.

Varying $\alpha_{min}(q_{\mathbf{cmn}})$: We study the impact of the minimum predicate selectivity in q_{cmn} (seed query), by varying $\alpha_{min}(q_{\text{cmn}})$ from 0.1% to 4%. As Figure 22 shows, selectivity has minimal impact for MQO-S which ignores evaluation costs, but noticeable impact in MQO. As selectivity is reduced, the number of optimized queries increases (less optimization) since MQO increasingly rejects optimizations that lead to more expensive (non-selective) common subqueries. While reduced selectivity increases the evaluation time of queries for all algorithms (Figure 23), MQO still achieves between 10% and 50% savings in evaluation times.

Varying $\alpha_{max}(\mathsf{Q})$: While changing minimum selectivity has an impact on deciding the sub-structure that forms q_{cmn}, maximum selectivity mostly affects the cost of evaluating the (non-seed) OPTIONAL terms. Here, we vary the maximum selectivity for predicates in a query, $\alpha_{max}(\mathsf{Q})$, from 0.1% to 4%. Like before, Figure 24 shows that the number of optimized queries is almost unaffected for MQO-S. Unlike the previous experiment, this number is also unaffected for algorithm MQO since the change in selectivity concerns OPTIONAL predicates and thus has less of an effect in the generation of optimized queries. Figure 25 shows that both MQO-S and MQO outperform No-MQO, with MQO achieving a *minimum* of 50% savings. Again, notice that when MQO-S chooses non-selective predicates for optimization, evaluation times quickly degrade to No-MQO as when $\alpha_{max}(\mathsf{Q}) > 1\%$.

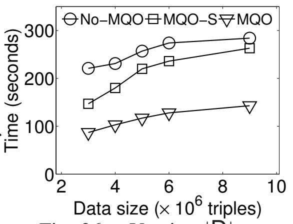
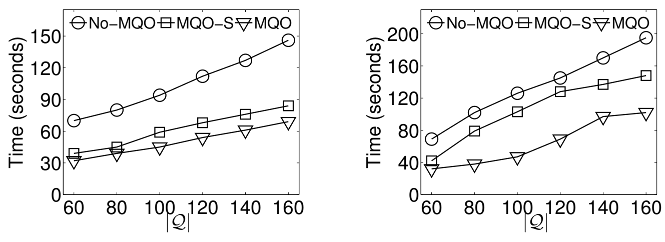


Fig. 26. Varying $|\mathsf{D}|$

Varying $|\mathsf{D}|$: We investigate the impact of dataset size $|\mathsf{D}|$ on the optimization results, by varying $|\mathsf{D}|$ from 3M to 9M triples. While this does not affect the number of rewritings of $\mathcal{Q}$ it clearly affects evaluation times, as shown in Figures 26. Notice that MQO consistently has a *minimum* of 50% (achieving up to 65%) savings.

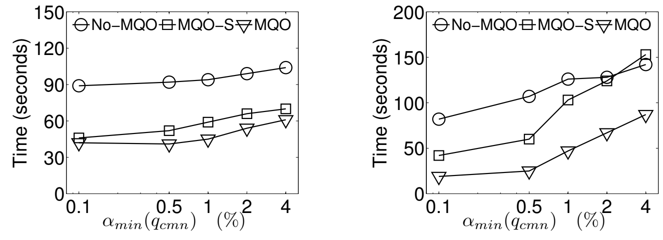
Effect of our cost model: In Section III, we extrapolate that the evaluation cost of a **Type 2** query is inversely correlated with the estimated cost of GP, *i.e.,* the minimum selectivity of its triple patterns. This is indeed a reasonable approximation in practice. As shown in Figure 23, reduced minimum selectivity in the common subquery GP would incur higher evaluation cost for **Type 2** queries. Similarly, both the number of OPTIONALs and the cost of the query pattern of each $\mathsf{GP}_{\mathsf{OPT}}$ are indispensable factors in determining the value of Δ, as shown respectively in Figure 19 and Figure 16. However, we observed that when the cost of GP is small (being selective), Δ would be a trivial value and Cost(Q) is mostly credited to the evaluation of GP. This is clearly shown in Figure 17 that when GP is selective, the dominant cost is contributed by evaluating GP (more than 90%) with the rest factors being almost irrelevant. This suggests that when dealing with a selective GP, a possibly good approximation of Cost(Q) can set $\Delta \simeq 0$. This observation also motivates us to choose a selective GP in rewriting. In practice, this simple cost model and its approximation give excellent cost estimation in MQO.

Results from other stores: Up to now, all results reported were performed with Jena TDB. Using the same queries and parameters, we also ran the experiments on Virtuoso and Sesame native, to evaluate the desired property of store independence. In general, the results from Virtuoso and Sesame are consistent with what we observed in Jena TDB, see Figures 27∼32 when we used the same setup as that in the experiments for Jena TDB, and varied values of one parameter while using default values for all other parameters. The proposed optimization algorithm, MQO, significantly reduces the evaluation time of multiple SPARQL queries on both stores.



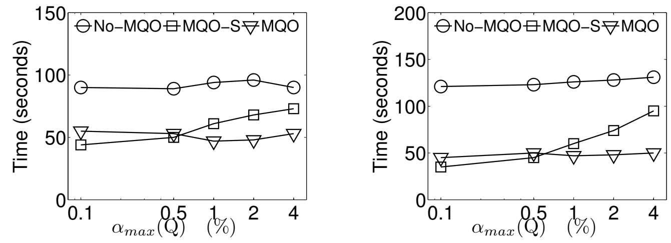
(a) Virtuoso (b) Sesame

Fig. 27. Vary $|\mathcal{Q}|$: evaluation time



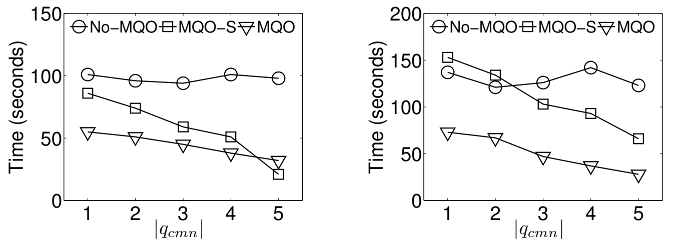
(a) Virtuoso (b) Sesame

Fig. 28. Vary $\alpha_{min}(q_{\text{cmn}})$: evaluation time



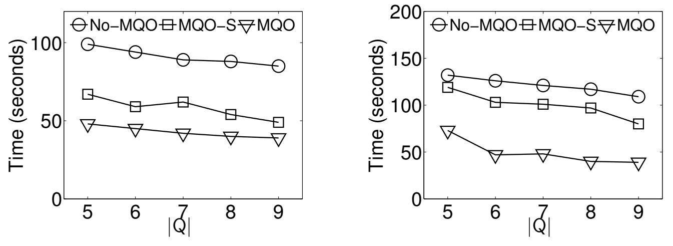
(a) Virtuoso (b) Sesame

Fig. 29. Vary $\alpha_{max}(\mathsf{Q})$: evaluation time



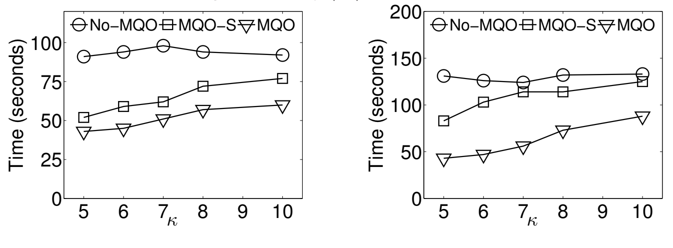
(a) Virtuoso (b) Sesame

Fig. 30. Vary $|q_{\text{cmn}}|$: evaluation time



(a) Virtuoso (b) Sesame

Fig. 31. Vary $|\mathsf{Q}|$: evaluation time



(a) Virtuoso (b) Sesame

Fig. 32. Vary κ: evaluation time

In particular, we consistently observed that the cost-based optimization can remarkably improve the performance in almost all experiments, leading to a 40%–75% speedup compared to No-MQO on both Virtuoso and Sesame. For example, using the same setting and optimized queries as Figure 12 where we vary the number of queries in a batch $\mathcal{Q}$, Figures 27(a) and 27(b) report the results from Virtuoso and Sesame. It is clear that MQO consistently outperforms MQO-S and No-MQO, leading to savings of 50%–60% across engines. Similarly, in the experiment that studies the impact of minimum selectivity in q_{cmn}, *i.e.,* Figure 28(a) and Figure 28(b), reducing the minimum selectivity of q_{cmn} results in increasing evaluation times for all algorithms. While MQO-S is sensitive to such variance since it does not proactively take cost into account, MQO still achieves 40%–75% savings in evaluation times.

VI. Related Work

The problem of multi-query optimization has been well studied in relational databases [22], [27], [31], [32], [42]. The main idea is to identify the common sub-expressions in a batch of queries. Global optimized query plans are constructed by reordering the join sequences and sharing the intermediate results within the same group of queries, therefore minimizing the cost for evaluating the common sub-expressions. The same principle was also applied in [27], which proposed a set of heuristics based on dynamic programming to deal with nested sub-expressions. There has also been studies on identifying common expressions [10], [40] with complexity analysis of MQO; the general MQO problem for relational databases is NP-hard. Even with heuristics, the search space for individual candidate plans and their combinatorial hybrid (*i.e.,* the global plan) is often astronomical [27]. In light of the hardness, [27] proposed some heuristics which were shown to work well in practice; however, those heuristics were proposed to work inside query optimizers (*i.e.,* engine dependent), and are only applicable when the query plans are expressible as AND-OR DAGs. Dalvi et al. [7] considered pipelining intermediate results to avoid unnecessary materialization. In addition to pipelining, Diwan et al. [8] studied the issue of scheduling and caching in MQO. A cache-aware heuristics was proposed in [20] to make maximal use of the buffer pool.

All of the above work focus on MQO in the relational case, MQO has also been studied on semi-structured data. Hong et al. [12] considered concurrent XQuery join optimization in publish/subscribe systems. Join queries were mapped to a pre-computed tree structure, called query template, for evaluation. Due to the limitation of the pre-computed templates, only basic join structures were supported. Another work by Bruno et al. [6] in XML studied navigation and index based path MQO. Unlike the MQO problem in relational and SPARQL cases, path queries can be encoded into a prefix tree where common prefixes share the same branch from the root. This nature provides an important advantage in optimizing concurrent path queries. Nevertheless, the problem of multi-query join optimization was not addressed. The work of Kementsietsidis et al. [15] considered a *level-wise merging* of query trees based on the tree depth of edges in a distributed setting, with the main objective to minimize the communication cost in evaluating tree-based queries in a distributed setting.

In summary, existing MQO techniques proposed in relational and XML cases cannot be trivially extended to work for SPARQL queries over RDF data (which can be viewed as SPJ queries over generic graphs), since relational techniques need to reside in relational query optimizers, which cannot be assumed in the management of RDF data, and notions like prefix-tree and tree depth do not apply to generic graphs. Also there have been work on query optimization for *single* SPARQL query [18], [29], [33], as well as *single* graph query optimization for general graph databases [41]. However, to the best of our knowledge, our work is the first to address MQO for SPARQL queries over RDF data.

VII. Conclusion

We studied the problem of multi-query optimization in the context of RDF and SPARQL. Our optimization framework, which integrates a novel algorithm to efficiently identify common subqueries with a fine-tuned cost model, partitions input queries into groups and rewrites each group of queries into equivalent queries that are more efficient to evaluate. We showed that our rewriting approach to multi-query optimization is both sound and complete. Furthermore, our techniques are store-independent and therefore can be deployed on top of any RDF store without modifying the query optimizer. Useful extensions on handling more general SPARQL queries are also discussed. Extensive experiments on different RDF stores show that the proposed optimizations are effective, efficient and scalable. An interesting future work is to extend our study to generic graph queries over general graph databases.

VIII. Acknowledgment

Wangchao Le and Feifei Li were partially supported by NSF Grant CNS-0831278.

References

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[2] R. Angles and C. Gutierrez. The expressive power of SPARQL. In *ISWC*, 2008.

[3] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: A scalable lightweight join query processor for RDF data. In *WWW*, 2010.

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

[5] C. Bizer and A. Schultz. The berlin SPARQL benchmark. *International Journal On Semantic Web and Information Systems*, 2009.

[6] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In *ICDE*, 2003.

[7] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *PODS*, 2001.

[8] A. A. Diwan, S. Sudarshan, and D. Thomas. Scheduling and caching in multi-query optimization. In *COMAD*, 2006.

[9] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *SIGMOD*, 2011.

[10] S. Finkelstein. Common expression analysis in database applications. In *SIGMOD*, 1982.

[11] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 2005.

[12] M. Hong, A. J. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, 2007.

[13] G. Ianni, T. Krennwallner, R. Martello, and A. Polleres. Dynamic querying of mass-storage RDF data with rule-based entailment regimes. In *ISWC*, 2009.

[14] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 1999.

[15] A. Kementsietsidis, F. Neven, D. V. de Craen, and S. Vansummeren. Scalable multi-query optimization for exploratory queries over federated scientific databases. *PVLDB*, 2008.

[16] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 2001.

[17] W. Le, S. Duan, A. Kementsieditis, F. Li, and M. Wang. Rewriting queries on SPARQL views. In *WWW*, 2011.

[18] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. In *PVLDB*, 2008.

[19] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.

[20] K. O'Gorman, D. Agrawal, and A. E. Abbadi. Multiple query optimization by cache-aware middleware using query teamwork. In *ICDE*, 2002.

[21] P. R. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, pages 195–205, 2002.

[22] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In *ICDE*, 1988.

[23] A. Polleres. From SPARQL to rules (and back). In *WWW*, 2007.

[24] N. Preda, F. M. Suchanek, G. Kasneci, T. Neumann, W. Yuan, and G. Weikum. Active knowledge : Dynamically enriching RDF knowledge bases by web services. In *SIGMOD*, 2010.

[25] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16:521–533, 2002.

[26] Resource Description Framework. http://www.w3.org/RDF/.

[27] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.

[28] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^{2}Bench: A SPARQL performance benchmark. *ICDE*, 2009.

[29] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *ICDT*, 2010.

[30] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Trans. Knowl. Data Eng.*, 2(2):262–266, 1990.

[31] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[32] K. Shim, T. K. Sellis, and D. Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data and Knowledge Engineering*, 12(2):197–222, 1994.

[33] M. Stocker, A. Seaborne, and A. Bernstein. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.

[34] The TPC Benchmarks. http://www.tpc.org/.

[35] E. Tomita and T. Seki. An efficient branch-and-bound algorithm for finding a maximum clique. *Discrete Mathematics and Theoretical Computer Science, LNCS*, 2003.

[36] N. Trigoni, Y. Yao, A. J. Demers, J. Gehrke, and R. Rajaraman. Multi-query optimization for sensor networks. In *Distributed Computing in Sensor Systems (DCOSS)*, LNCS.

[37] P. Vismara and B. Valery. Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. *Modelling, Computation and Optimization in Information Systems and Management Sciences*, pages 358–368, 2008.

[38] W3C Wiki for Currently Alive SPARQL Endpoints. http://www.w3.org/wiki/SparqlEndpoints.

[39] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 2008.

[40] H. Z. Yang and P. Larson. Query transformation for PSJ-queries. *PVLDB*, 1987.

[41] P. Zhao and J. Han. On graph query optimization in large networks. In *PVLDB*, 2010.

[42] Y. Zhao, P. Deshpande, J. F. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *SIGMOD*, 1998.