Optimizing Query Rewriting for Multiple Queries

George Konstantinidis
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292
konstant@usc.edu

José Luis Ambite
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292
ambite@isi.edu

ABSTRACT

We present an scalable algorithm for answering *multiple* conjunctive queries using views. This is an important problem in query optimization, data integration and ontology-based data access. Since rewriting one conjunctive query using views is an NP-hard problem, we develop an approach where answering n queries takes less than n times the cost of answering one query, by compactly representing and indexing common patterns in the input queries and the views. Our initial experimental results show a promising speed up.

1. INTRODUCTION

In this paper, we consider the problem of query answering using views, an important problem in query optimization and data integration [5, 7]. In query optimization, the system rewrites queries by replacing *a part* of the original query with view predicates to obtain an *equivalent*, but more efficient, query. In data integration systems, users pose queries over a global virtual schema. The system, using schema mappings, rewrites the user query to a query using *only* the schemas of the data sources. In this paper we consider *Local-as-View* (LAV) mappings [5, 7], where each source relation is expressed as a logical formula (a view) over the global schema. We also call these formulas *source descriptions*. We focus on conjunctive queries and views, which correspond to select-project-join queries and are the core of every query language. In data integration, sources are often incomplete and hence the system needs to produce query rewritings that instead of equivalent are *maximally-contained*.

Although multi-query processing has been studied in traditional relational database systems [12], relevant algorithms and systems in data integration have focused on rewriting a single user query using the views. We advocate the need for (and present) an approach optimized to rewrite multiple input queries using a set views, by taking advantage of overlaps across queries. Our algorithm is useful in several integration contexts: (1) systems that serve multiple users each issuing different queries simultaneously, (2) systems where users issue unions of conjunctive queries (UCQ), and (3) systems enhanced with integrity constraints, as in *ontology-based data integration* (OBDI) [3, 9, 8, 11]. For example, in [9] the user writes a query over a global schema/ontology. Then, the system rewrites the original query by compiling the inferences embodied in the ontology into an expanded query, which is a UCQ, still over the global ontology terms, whose size may be exponential on the size of the ontology in the worst case. Then, the integration system will rewrite this UCQ into another UCQ query that uses only the sources' schemas. In all these cases, it is important to efficient process multiple queries over large numbers of sources.

In order to address the multi-query rewriting problem we leverage insights revealed by our recent work on GQR [6] a scalable algorithm for query rewriting that compactly represents and indexes common subexpressions in the views. In GQR we used a graph representation of views, which we also adopt in this paper for our input queries.

Our main contribution is an algorithm, MGQR, for scalable rewriting of *multiple* queries in the presence of large numbers of views. Our algorithm extends the GQR approach in two significant ways. First, MGQR finds common graph patterns in *both* the queries and in the views, compactly representing and indexing these patterns, but carefully keeping track of which patterns are relevant for which queries. Second, the graph patterns are combined incrementally in a way that multiple views are used to cover multiple queries simultaneously. Our initial experimental results show a promising speedup of rewriting the user queries in batch versus rewriting the user queries one by one.

2. THE QUERY REWRITING PROBLEM

To define the problem formally we introduce the concepts of query containment [4, 1] and query rewritings [5].

Definition 1 (Query Containment): A query Q_1 is contained in a query Q_2, $Q_1 \subseteq Q_2$, iff for all databases D, the result of evaluating Q_1 on D, denoted $Q_1(D)$, is contained in the result of evaluating Q_2, that is, $Q_1(D) \subseteq Q_2(D)$.

Definition 2 (Query Equivalence): Q_1 is equivalent to Q_2, denoted $Q_1 = Q_2$ iff $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$.

Definition 3 (Maximally-Contained Rewriting): A query Q' is a maximally-contained rewriting of Q using views $\mathcal{V}$ if: (1) Q' is using only source predicates, $\mathcal{V}$, (2) $Q' \subseteq Q$, and (3) there is no rewriting Q'' of Q using $\mathcal{V}$, such that $Q' \subseteq Q'' \subseteq Q$ and $Q'' \neq Q$.

To ground these definitions, consider the following LAV rules describing medical records sources. S_1 contains doctors that treat patients with a chronic disease. S_2 contains

IIWeb '12 , May 20 2012, Scottsdale, AZ, USA

doctors, patients and clinics where the doctor is responsible for discharging the patient from the clinic.

S_1: V1(doctor, disease) → TreatsPatient(doctor, patient), HasChronicDisease(patient,disease)

S_2: V2(doctor, patient, clinic) → DischargesPatientFromClinic(doctor, patient, clinic)

We use a logical notation for queries and views [1]. For historical reasons and similarity to datalog, in LAV rules the (single source predicate) antecedent is called the *head* and the consequent the *body* of the rule. In our example, S_1 contains only references to the doctors and the diseases they treat, but not to the patients that have these diseases; this information conceptually exists in the body but is not provided by the source (perhaps due to privacy concerns). The variables in the head of a query/view are called *distinguished*, e.g., "doctor" and "disease" in S_1. Otherwise, they are called *existential*, e.g., "patient" in S_1.

Logical implication (→) in the mapping (view) indicates that the source (antecedent, head) contains tuples that satisfy the logical formula in the consequent (body), but it does not contain *all* such tuples (i.e., if follows an open-world assumption). Usually in data integration (e.g., web sources, peer data, or medical records), we can describe the constraints that a source satisfies, but rarely be assured that the source is complete. For example, we do not expect S_2 to provide all possible tuples of doctors, patients and clinics, but rather a subset specific to the source, e.g., for a region or a hospital. In this paper, we focus on open-world and maximally-contained rewritings.

Assume the user asks for doctors treating chronic diseases and the clinics that they work (discharge patients from):

q_2(d, c) ← TreatsPatient(d,x), HasChronicDisease(x,y), DischargesPatientFromClinic(d,z,c)

A rewriting of q is: q'(d, c) ← V1(d, y), V2(d, z, c).

Intuitively, we can get the doctors treating chronic diseases from V1 and join it on "doctor" with V2 to find the clinics. In this example, the selection of relevant views and the reformulation process was quite simple. We just built the conjunctive query q' using the two views and tested that $q' \subseteq q$. Since V1 and V2 are the only sources available, q' is the best we can do and it is a maximally-contained rewriting of q using V1 and V2. In general, there may be a large number of sources and the number of possible rewritings that would need to be considered and tested for containment grows exponentially.

Conjunctive query containment is NP-complete and can be computed through containment mappings [4].

Definition 4 (Containment Mapping): For two conjunctive queries over the same schema, a *containment mapping* from Q_1 to Q_2 is a homomorphism from the variables of Q_1 to those of Q_2, h:$vars(Q_1) \rightarrow vars(Q_2)$ (h is extended over atoms and queries in the obvious manner), such that: (1) for all atoms $A \in body(Q_1)$, it holds that $h(A) \in body(Q_2)$, and (2) $h(head(Q_1)) = head(Q_2)$.

For all conjunctive queries Q_1, Q_2 over the same schema, $Q_2 \subseteq Q_1$ iff there is a containment mapping from Q_1 to Q_2. In order to check whether a rewriting q' (over the source schemas, $\mathcal{V}$) is contained in a user query q (over the global schema), we unfold the atoms of q' with their definitions. The new query $unfold(q')$ is defined over the global schema, and we can check whether $unfold(q') \subseteq q$ through containment mappings.

3. OUR APPROACH: MGQR

Approaches to scalable LAV query rewriting [10, 2] have focused on pruning views that would result in a non-contained rewriting. In [6], we pushed this intuition further and developed a much more efficient approach. In this paper we use the same intuitions, but present a novel solution for the case of multiple query reformulation.

Coverings are restrictions of containment mappings that map a sub-part of the query body to a sub-part of a view. A rewriting essentially consists of multiple view sub-parts, so we can "combine" these partial mappings (coverings) to establish the containment mapping between the query and the rewriting. Query rewriting algorithms look for legitimate coverings to select which views participate in a rewriting.

Definition 5 (Covering): For all queries Q, views V, predicates $g_q \in body(Q)$, predicates $g_v \in body(V)$, and partial homomorphisms $\varphi : vars(Q) \rightarrow vars(V)$, we say that a view predicate g_v covers a predicate g_q of Q with φ iff: (1) $\varphi(g_q) = g_v$, and (2) for all $x \in vars(g_q)$ if x is distinguished then $\varphi(x) \in vars(g_v)$ is distinguished.

Condition 2 in Def. 5 is exactly the same as condition 2 in the containment mapping definition. The intuition behind this is that whenever a part of a query needs a value, you cannot cover that part with a view that does not explicitly provide that value. Abusing definition we say that a set of predicates of V, or even V itself, covers q_q with φ (since these coverings involve trivial extensions of φ).

Coverings should adhere to one more constraint. Consider the example sources S_1 and S_2 of Sect. 2 and q_3 below which asks for doctors that treat patients with chronic diseases and the clinics where they discharge those *same* patients from:

q_3(d, c) ← TreatsPatient(d,x), HasChronicDisease(x,y), DischargesPatientFromClinic(d,x,c)

In contrast to q_2, q_3 requires that the second argument of *DischargesPatientFromClinic* is joined with the patients that are treated for chronic diseases. This is impossible to answer, given S_1 and S_2, as S_1 does not provide the patients. The property revealed here is that *whenever an existential variable x in the query maps on an existential variable in a view, this view can be used for a rewriting only if it covers all predicates that mention x in the query* (clause $C2$ in *Property 1* in [10]). This is the basic idea of the MiniCon algorithm: trying to map all query predicates of q_3 to all possible views, it will notice that the existential query variable x in the query maps to *patient* in S_1; since *patient* is existential it needs to go back to the query and check whether all predicates mentioning x can be covered by S_1. Here *DischargesPatientFromClinic(d,x,c)* cannot.

We notice that MiniCon does some redundant work in this process. First, it would try to do this mapping and "backtracking" for *every possible view*, even for those that contain the same pattern of S_1, like S_3 below, which provides surgeons and the (chronic) diseases of the patients they treat:

S_3: V3(doctor, disease) → TreatsPatient(doctor, patient), HasChronicDisease(patient,disease), Surgeon(doctor)

S_3 cannot be used for q_3 as it violates MiniCon's Property1, again due to *patient* being existential and *DischargesPatientFromClinic* not covered. Second, any "one-by-one" algorithm would check this property for *every possible query*, regardless of the overlap with previous queries, as in q_4:

q_4(d, c) ← TreatsPatient(d,x),DischargesPatientFromClinic(d,x,c)

S_1 and S_3 cannot be used for q_4 exactly for the same reason as q_3: *patient* is existential and *DischargesPatient-FromClinic* is not covered. Despite multiple occurrences of $TreatsPatient$ across different input queries, any "one-by-one" query rewriting algorithm would try to use S_1 and S_3 multiple times (and fail for all queries that join $TreatsPatient$ with $DischargesPatientFromClinic$ in the way shown in q_3 and q_4). Note that these redundancies hold even for successful rewritings. If S_1 and S_3 did cover *DischargesPatient-FromClinic*, in order to use them, a "one-by-one" algorithm would make the same steps for both q_3 and q_4.

Our idea is to avoid this redundant work by compactly representing all occurrences of the same query or view pattern, extending [6] to handle multiple queries. Our solution is divided in an offline phase which preprocess all the views, and an online phase which produces rewritings in the face of a set of input queries. Our online phase has three stages. First, we represent the queries as graphs and find common patterns. Second, for every query graph pattern (which now represents pieces of multiple queries), we retrieve the pre-constructed view patterns that cover it (which in turn represent pieces of multiple views). Third, we incrementally combine the view patterns to larger ones, progressively covering the underlying queries. Consequently, we naturally come up with a "batch" of contained rewritings using multiple views to cover multiple queries at the same time.

We will use the following queries and views to illustrate our algorithm:

$q_3(x,z) \leftarrow P_1(x,y), P_2(y,z), P_3(y)$	$S_4(x,y) \rightarrow P_1(x,y)$
$q_4(x,z) \leftarrow P_1(x,y), P_2(y,z)$	$S_5(x,y) \rightarrow P_2(x,y)$
$q_5(w) \leftarrow P_2(y,w), P_3(w)$	$S_6(z) \rightarrow P_3(z), P_1(z,x), P_2(z,y)$
	$S_7(x,y) \rightarrow P_1(x,z), P_2(z,y)$

3.1 Graph Modeling

Our graph representation translates predicates and their arguments to graph nodes. Predicate nodes are labeled with the name of the predicate, and they are connected through edges to their arguments. Edges are labeled with the argument position. Shared variables between atoms result in shared variable nodes. We discard variable names, because the only knowledge we require for deciding on a covering is the type of the variables. Distinguished variables are depicted with a circle, and existential ones with the symbol $\otimes$. Queries q_3,q_4,q_5 correspond to the graphs on Fig. 1(a). The graphs for source descriptions S_4, S_6 and S_7 appear in Fig. 1(b). Our algorithm consists of mapping subgraphs of the queries to subgraphs of the sources, and to this end the smallest subgraphs we consider represent one atom's "pattern": they consist of one central predicate node and its (existential or distinguished) variable nodes. These primitive graphs we call *predicate join patterns* (for short, PJs). Fig. 1(c) shows all predicate join patterns that the query q_3 contains, (i.e., the *query PJs* for q_3).

A critical feature that boosts our algorithm's performance is that the graph patterns of predicates repeat themselves in multiple queries (sources). Therefore we compactly represent each occurrence of the same predicate across different queries (sources) with the same PJ. This has a tremendous advantage; mappings from a query PJ to a view one are computed just once instead of every time this predicate (or set of predicates) is met in a query (resp. source description). For the query PJ for P_1 seen in Fig. 1(c), all source PJs that could potentially cover it (cf. Def. 5) appear in Fig. 1(d). Unless our sources contain one of these two patterns any query that contains this variation of P_1 will fail (immediately) to be rewritten.

Nevertheless, the "join conditions" for a particular PJ within each query or view are different and some "bookkeeping" is needed to capture these joins. To retain this information, we attach to each variable a data structure that we call *infobox*. A variable's infobox contains a list of queries/views in which this PJ appears, and for each such rule the variable's *join descriptions*, which record which other PJs this variable (directly) joins to within the specific query/view. Fig. 1(e) shows two example infoboxes for one query and one view variable node. The upper level of Fig. 2 also shows all the different PJs that appear in queries $q_1 - q_3$ with their infoboxes (aggregating information from all queries where they appear). All the different source PJs, relevant to the query ones, that appear in sources $S_4 - S_7$ with their infoboxes are shown on the middle level of the same figure. Additionally, at different steps of our algorithm, each source graph consisting of PJs, covers a certain part of the queries and within this graph we maintain a list of "candidate" parts of the final conjunctive rewritings per query (that will eventually be "responsible" for covering this part of the query); we call these, *partial* conjunctive rewritings.

3.2 Multi-query rewriting

The three stages of MGQR online phase correspond to the three horizontal levels of Fig. 2. First, MGQR constructs unique PJs and infoboxes for all common patterns that appear across the queries. This procedure can be implemented in time polynomial in the number and the length of the queries (we omit its description for space).

Second, having constructed and indexed all source PJs offline, MGQR can efficiently retrieve the source PJs that cover our query PJs at runtime. All source PJs that cover a query PJ form a set (each "bubble" in the second level of Fig. 2), whose elements contain alternative partial rewritings for the pieces of the *queries* represented by the query PJ. At the third level of Fig. 2, MGQR combines these sets into larger ones, combining their partial rewritings and progressively covering larger subgoals of the underlying queries.

3.2.1 Retrieving source patterns

MGQR uses Algorithm 1 to retrieve all the relevant source PJs that cover each query PJ (i.e., the output of the algorithm is a set of PJs). For an input query PJ, line 1 of Alg. 1 iterates over all (existing) view PJs that cover the input. Moreover, due to MiniCon's Property 1 discussed in Sect. 2, if both a query variable and its mapped view variable (u_q and v_s correspondingly in our pseudocode) are existential, we won't be able to use this view PJ for a particular query if the view cannot "preserve" the join patterns of the query. Depending on this, all the sources in a source PJ's infobox are associated with some or all the queries in the underlying query PJ.

When considering a source variable node and its covered query variable we examine all pairs of sources and queries that the corresponding infoboxes contain; if the variables are existential (line 7) we need to make sure that each source in the source PJ (in our algorithm PJ_s) will be used to cover underlying queries for which it describes their joins.

Therefore for every source of v_s's infobox, and query in u_q's infobox we have to verify that all join descriptions of u_q

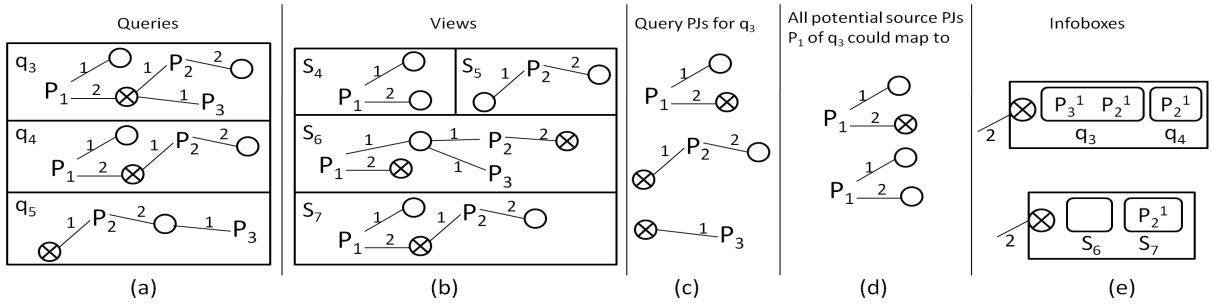

Figure 1: (a) Queries q_3,q_4,q_5, and (b) sources S_4-S_7 as a graphs. Query (c) and Source (d) Predicate Join Patterns. (e) Infoboxes for a query and a view variable nodes. The variable in the upper box is existential and it appears in two queries q_3, q_4. The join description associated with q_3 states that the variable joins, in q_3, with the first argument of P_3 and the first argument of P_2.

are included in the sources. If we find a source S that breaks this requirement we mark that q_i and S are uncombinable in PJ_s (line 9). On the other hand, if S can cover q_i on this variable's joins (line 13) or it doesn't need to because u_q and v_s are distinguished (line 16), we associate S with q_i. Note that further checks on a different variable node might break a previous association of a query with a source on PJ_s (we do this in line 9 as well). Association, apart from the creation of a relative pointer, means that we create a partial conjunctive rewriting that uses a source for a specific query. In the set of source PJs that is retrieved for query PJ P_1 in Fig. 2, source S_7 does not cover the existential joins of query q_3 and hence it is only associated to q_4.

If a source cannot be associated to any query for a certain query PJ (e.g. source S_6 in Fig. 2 for P_1), we drop this source from every infobox of this source PJ and we delete the partial conjunctive rewriting that mentions the corresponding view as well (line 18). We do all that as if this source PJ pattern never appeared in that view (for the specific subgoal of this query PJ's queries, the specific view subgoal that this source PJ represents is useless). Moreover if some query (of the query PJ) does not get associated to any view for PJ_s, this means the specific query subgoal (that PJ_q represents) and consequently the entire query cannot be rewritten. Hence, we destroy all the information for this query in the system, as if this query never existed in our input (line 23).

This fail-fast behavior allows us to keep only the necessary query/view references in a PJ. Moreover, if none of the views can cover a query PJ, the source PJ itself is ignored and never returned, leading to a faster reformulation performance as (1) a dropped view PJ means that a significant number of source pieces/partial rewritings are ignored and (2) if we ignore all source PJs that could cover a specific query PJ, the algorithm fails instantly for the queries of this query PJ. On the other hand if some PJs go through this procedure and get returned, these are really relevant and have a high possibility of generating rewritings for their associated queries.

The algorithm also addresses repeated predicates, i.e., self-joins, in the input queries (line 29). In the face of multiple occurrences of the same predicate in a query, it is convenient to consider all source PJs discussed so far as classes of PJs: we instantiate the set of source PJs that cover a specific predicate as many times as the predicate appears in the algorithm's input. Each time we instantiate the same PJ we "prime" the sources appearing in the partial rewritings so as to know that we are calling the same source but a second, different time. This modeling is a natural extension of our approach for repeated predicates in the views (described in [6]). We omit further discussion of repeated predicates in the same query or view due to space limitations.

Alg. 1 returns a set of PJs which alternatively cover the same query PJ. Also, different queries of the query PJ could be associated with different source PJs in the returned set. Furthermore the different sets that Alg. 1 returns, cover different (and all) subgoals of the queries. Next we want to

Algorithm 1 Retrieve Source PJ Sets for Input Queries

Input: Predicate join patterns PJ_q in the queries
Output: Set of source PJs that "alternatively" cover each PJ_q.

```
1: for all PJ_s, source PJ that covers PJ_q do
2:   OkToAdd ← true
3:   for all u_q variable nodes in PJ_q do
4:     v_s ← variable of PJ_s that u_q maps on to
5:     for all sources S in v_s's infobox do
6:       for all queries q_i in PJ_q do
7:         if v_s is existential and u_q is existential then
8:           if joins in u_q for q_i ⊈ joins in S then
9:             mark q_i and S as uncombinable in PJ_s
10:          else
11:            if q_i and S are not marked as uncombinable in PJ_s then
12:              if joins in u_q for q_i ⊆ joins in S then
13:                Associate S with q_i
14:        else
15:          if q_i and S are not marked as uncombinable in PJs then
16:            Associate S with q_i
17:    if There is some source marked as uncombinable with all queries of PJ_q then
18:      drop S from PJ_s
19:    if some of PJs infoboxes became empty then
20:      OkToAdd ← false
21:      break
22:    if some queries are not associated with any source then
23:      drop all those queries from all query PJs and all return cover-sets of source PJs
24:      OkToAdd ← false
25:      break
26:  if OkToAdd then
27:    add PJ_s to C
28: if we have seen the input query PJ_q before, i.e., it is a repeated pattern (selfjoin) then
29:   Rename (i.e. prime) the elements of C (which have also been returned in the past)
30: return C
```

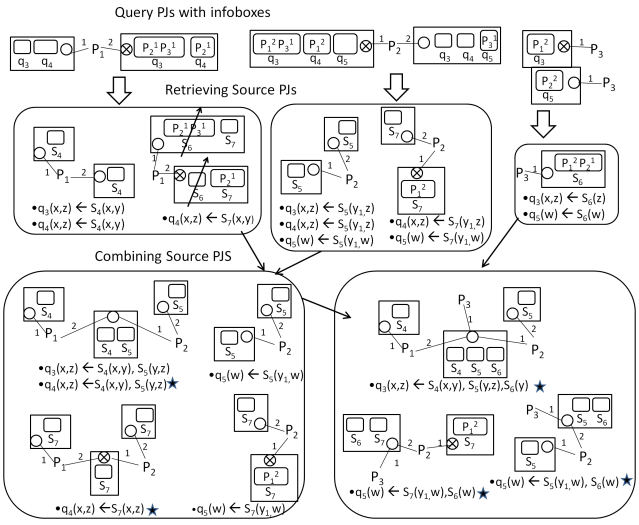

Figure 2: The PJs covering P_1 are first combined with the PJs covering P_2, and then with the PJs covering P_3. The union of the complete rewritings (marked with $\star$) is the solution.

combine these sets to cover larger parts of the queries.

3.2.2 Combination of Source Graph Patterns

To combine two sets of source graphs that cover two different query PJs and consequently two different sets of pieces of queries, MGQR uses Algorithm 2. We want to combine elements of these sets that cover the same queries (no need to try to combine a PJ that covers a part of q_3 with a PJ that covers a part of q_4 but not q_3). In fact, if one of the two sets covers queries that are not covered by the other set, we copy the corresponding source PJs directly in the resulting set (lines 7-11). In line 10, in case these PJs are also covering queries common between the two sets, we keep a copy of them in the original set so we can go on and combine it; this copy is now associated only with the common queries (in essence we "project out" of the source PJs the non-common queries before we combine them). Thus, we only combine elements of these sets (i.e., source PJs) if they cover common queries (lines 12-17). The third level of Fig. 2 shows this procedure; the all-distinguished-variable source PJs for P_1 and P_2 are combined on their common queries (i.e., q_3 and q_4), while a copy of the source PJs for P_2 associated with q_5 is directly passed onto the resulting set.

Lines 1-6 of Alg.2 check whether some queries have completed so far, in which case we output their rewritings and delete their PJs (if the PJs are not associated with any other "active" rewriting). We omit the procedure that combines two specific graphs patterns (line 15), but we should state that it does so based on the underlying query joins, combining/merging the source PJ's partial conjunctive rewritings per query, into larger ones, eventually producing the maximally-contained rewritings of each query. This combination could also fail (due to existential-distinguished patterns or due to violation of the corresponding underlying query joins), in which case nothing will be added in the resulting set in line 17. If combining two sets returns an empty set, the two query PJs and all their associated queries instantly fail. The order in which we combine the retrieved

Algorithm 2 Combine Source PJ Sets

Input: Two sets of source PJs: A, B
Output: 1) Set C combining A and B, with partial rewritings. 2) Complete rewritings R found so far.

```
1: for all queries q_i covered by set A (or set B) do
2:   if q_i is completely covered then
3:     Add rewritings to R
4:     delete this query from set A (resp. set B) and any
       associated PJs
5:     if some sources PJs are "empty" of queries then
6:       delete them from the set
7: for all queries q covered by exactly one set out of A or
   B do
8:   for all PJs PJ_i (elements of A or B) associated with
     q do
9:     create a copy of PJ_i, associate it only with q and
       put it in set C
10:    if PJ_i is also associated with some queries that do
       exist in both A and B then
11:      the copy of PJ_i remaining in A or B should be
         associated only with queries common between A
         and B (drop information about other queries)
12: for all queries q common in sets A, B do
13:   for PJ_a elements of A associated to q do
14:     for PJ_b elements of B associated to q do
15:       combine (PJ_a, PJ_b)
16:       if combination successful then
17:         put result in C
18: return C, R
```

sets of PJs is currently driven by a simple heuristic: prefer to combine the sets that share the biggest number of queries.

4. EXPERIMENTAL RESULTS

To evaluate our multiple-query rewriting algorithm, we generated 100 experiments, each one testing a multi-query input on a set of views. Each input had 10 chain queries and each view set had up to 1000 chain views. Each query/view had 8 predicates out of which up to 4 could be repeated. Each atom had 4 randomly generated variables and each query had 10 distinguished variables.[1] We ran our experiments on a cluster of 2GHz processors each with 2Gb of memory, and gave each processor one (out of the one hundred) experiment: one multi-query and one view set. Each processor runs the experiments between 0 and 1000 views (in increments of 50 views at a time) in two settings: (1) using our MGQR multi-query "batch" rewriting algorithm, and (2) using GQR to rewrite the queries one-by-one.

Since we compute all rewritings, we want to avoid problems that produce an exponential number of rewritings or produce no rewritings. In the first case, the times would be dominated by the exponential output, and in the second case, our algorithms prove unsatisfiability extremely fast. So, none of those settings would be interesting. Hence, we try to find the "phase transition" of the query rewriting problem, where there are a number of rewritings produced that are hard to find. To simulate this condition, we generated the first 180 views for all our view sets containing 10 distinguished variables and each additional view (up to 1000) with only 3 distinguished variables. We created the bodies of our views by randomly choosing 8 predicates out of an increasing predicate space. These two choices led to view

[1]The initial version of the query/view generator was kindly provided to us by Rachel Pottinger and it was the same one she used for the experiments of MiniCon.

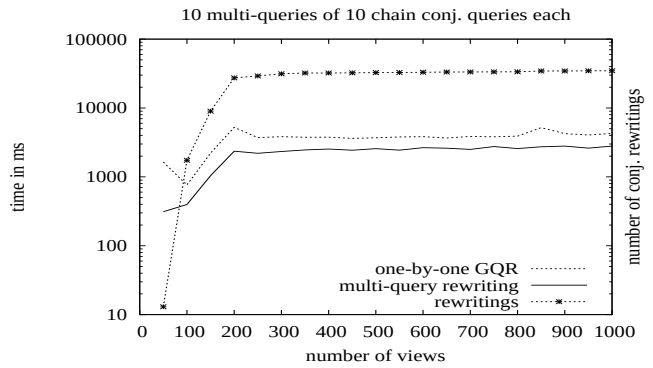

Figure 3: Average online time and number of rewritings for 10 chain queries over up to 1000 views. The MGQR multiquery rewriting algorithm outperforms GQR rewriting queries one by one. Offline times are the same for both algorithms (not shown).

sets that initially generate an exponential number of rewritings, but subsequently the number of rewritings grows very slowly as the number of views grows (see Fig. 3).

Initially, we generated the user queries randomly, as we did for the views. To produce overlap in the user queries, we used a space of 20 predicate names out of which each query chooses randomly 8 to populate its body. We hoped to create enough overlap in the queries to demonstrate the benefits of our algorithm. However, it turned out that this was not adequate. In each multi-query experiment only one or two, out of the ten, queries would generate rewritings. As both GQR and MGQR fail very quickly when there are no rewritings, both algorithms had similar performance. In fact we were only measuring the rewriting time for one or two queries, making GQR and MGQR indistinguishable.

Thus, we decided to generate the user queries based on combinations of the queries that did produce rewritings in our early experiments. Specifically, we chose ten multi-query sets that produced a substantial amount of rewritings and kept only the two queries per set that produced rewritings. We replicated these queries in order to grow them back to ten queries per set. For each set, we also deleted two predicates out of each query body (different predicates each time). This way each set of queries, now of length 6, were all different, but overlapping (and at the same time rewritable).

Fig. 3 shows the average online times that these 10 multi-query sets, each having 10 conjunctive queries, took to reformulate over different number of sources, up to 1000. Batch MGQR outperforms the sum of the one-by-one rewritings by GQR by a factor of approximately 1.5 (for the 1000 view problem, GQR takes 4265 ms and MGQR takes 2798 ms). Fig. 3 also shows that the number of rewritings grows up to 34771 for the 1000 view problem up from 27370 rewritings for the 200 view problems. Note that in the region where the number of rewritings grows slowly, both GQR and MGQR times are proportional to the number or rewritings, instead of depending on the number of sources available. In summary both GQR and MGQR can rewrite multiple queries over large numbers of views, producing tens of thousands of rewritings, under a few seconds. MGQR outperforms GQR in the multiple query problems when there is meaningful overlap between the queries.

5. DISCUSSION

We have presented MGQR, an scalable algorithm for *multiple* query rewriting, that exploits common patterns across queries and source descriptions. Our initial experiments are promising, and demonstrate that MGQR can take advantage of the overlap in the user queries.

In future work, we plan to investigate heuristics for the order of combination of our algorithm that may improve its performance. We plan to perform additional experiments with star and random queries, and to scale to larger numbers of views and queries. We also plan to support richer schema mappings, namely GLAV (aka st-tgds), as well as more expressive description languages, such as DL-Lite, so that we can use the MGQR approach in OBDI applications. We plan to explore whether our approach to compact representation of common patterns can be extended to perform efficient query rewriting under ontological constraints.

6. ACKNOWLEDGMENTS

This work was supported in part by the NIH through the NCRR grant: the Biomedical Informatics Research Network (1 U24 RR025736-01), and in part through the NIMH grant: Collaborative Center for Genetic Studies of Mental Disorders (2U24MH068457-06).

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Y. Arvelo, B. Bonet, and M. E. Vidal. Compilation of query-rewriting problems into tractable fragments of propositional logic. In *Proc. of AAAI'06*.

[3] D. Calvanese, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *Journal of automated reasoning*, 39(3):385–429, 2007.

[4] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. 9^{th} ACM Symposium on Theory of Computing, 1977*.

[5] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[6] G. Konstantinidis and J. L. Ambite. Scalable query rewriting: A graph-based approach. In *ACM SIGMOD Conference*, Athens, Greece, June 2011.

[7] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[8] H. Pérez-Urbina, I. Horrocks, and B. Motik. Tractable query answering and rewriting under description logic constraints. *Journal of Applied Logic*, 8(2):186–209, 2010.

[9] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *J. on Data Semantics*, X:133–173, 2008.

[10] R. Pottinger and A. Halevy. MiniCon: a scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2–3):182–198, 2001.

[11] R. Rosati and A. Almatelli. Improving query answering over dl-lite ontologies. pages 290–300, Toronto, Canada, 2010.

[12] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.