Towards Scalable Data Integration under Constraints

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

ABSTRACT

In this paper we consider the problem of answering queries using views, with or without ontological constraints, which is important for data integration, query optimization, and data warehouses. Our context is data integration, so we search for maximally-contained rewritings. We have produced a very scalable and efficient solution for its simplest form, conjunctive queries and views, and we are working towards the full relational case. When considering constraints, the problem is usually divided in two phases: (1) query expansion, which rewrites queries w.r.t. the intentional knowledge and (2) expanded query reformulation using the views. Relevant algorithms have given little attention to the second phase and have studied a limited form of view definition languages overall (namely, only GAV). By looking at the problem from a graph perspective we are able to gain a better insight and develop designs which compactly represent common patterns in the source descriptions, and (optionally) push some computation offline. This allows us to contribute significantly in both aforemention phases individually, tailor one to each other, and moreover address them in a unified way. We intend to provide a solution that supports a variety of ontology languages, and all prevalent view definition languages (G/LAV). Towards such a general and scalable system our preliminary results for the relational case, show an experimental performance about two orders of magnitude faster than current state-of-the-art algorithms, rewriting queries using over 10000 views within seconds.

1. INTRODUCTION

In information integration, a virtual mediator integrates information from multiple heterogeneous sources by defining a global schema and then describing the contents of the sources in terms of this schema. The user poses queries to the system using the global schema as if it were a single centralized repository. The actual data however are stored at the sources and are organized according to independently developed source schemas. Therefore, the sources must be queried accordingly: the mediator reformulates (rewrites) the user query into another query that only uses terms from the source schemas. In the problem's most prevalent form (widely known as *answering queries using views* and extensively studied in query optimization, data integration and other areas [10, 12]) the sources and the mediator are modeled by relational schemas.

Nevertheless, a significant part of industrial and academic interest has recently focused on imposing various forms of constraints on the mediator schema, that allow for intensional knowledge [5, 12, 14, 16]. In this problem, usually referred to as *ontology-based data integration* (OBDI), a set of constraints (written e.g., in DL-lite [5]) form an ontology that lies on top of the mediator. The user's query addresses the ontology schema (could be written, e.g, in SPARQL) and needs to be rewritten not only in terms of the sources but taking the ontological constraints into account as well. We are focusing on the query rewriting problem in data integration both in the relational case and under richer ontological constraints.

Mappings between the sources' schemas and the mediator schema are usually given in the form of logical formulas, which we call *source descriptions*, or views[1]. In the Global-as-View (GAV) [9] approach, each mediator relation (or predicate) is defined by a view involving source predicates. Conversely, in the Local-as-View(LAV) [13, 7] approach each source predicate is defined by a view over mediator predicates. GLAV [8] is a generalization of GAV and LAV. We intend using all different kinds of mapping and towards this goal we choose to start from the most interesting case of LAV.

In [11] we looked at the relational query rewriting problem using LAV mappings, from a graph perspective and we were able to gain better insights and design a solution which compactly represents common patterns in the mappings, and (optionally) pushes some computation offline. This together with other optimizations resulted in an experimental performance that rewrites queries using over 10000 views within seconds, and is about two orders of magnitude faster than current state-of-the-art algorithms. We are currently extending this work to cover the full relational case.

At the same time we point out that relevant OBDI approaches, have paid little attention to the integration aspect of the problem. The problem is usually divided into two phases, and relevant algorithms focus mostly on the first one, known as *query expansion* [5], which rewrites the original (ontological) query by taking into account the ontology inferences; it is, in essence, "compiling" the ontology in the query and expanding the latter by adding a (possibly exponential) number of queries that account for the intentional knowledge. This technique, rather than integration, is tailored for *Ontology-Based Query Answering* of data stored in a single database. The actual integration of sources happens in a second phase, where the aforementioned approaches call upon existing algorithms to reformulate the expanded queries, typically using GAV views. It is worth noticing that LAV query reformulation might produce an exponential number of rewritings, for each one of the expo-

[1]In the context of query optimization views are materialized answers of previously evaluated queries.

nentially many queries that the query expansion outputs. Hence, it is very difficult for practical systems to emerge from this approach. No relevant work, to the best of our knowledge, has addressed the two phases in a unified way, and moreover no relevant approach has focused in addressing the interesting case of LAV (or GLAV) mappings. Our contributions are foreseen as follows:

- We present a scalable algorithm for the problem of relational query answering using LAV views. We are working to optimize it further and to extend it to cover constants, GLAV rules, interpreted predicates, minimization of output and more expressive queries as our input such as unions of conjunctive queries (UCQs) and non-recursive (nr) datalog programs.
- Having a scalable algorithm for query rewriting will allow "porting" a query expansion phase on top, to support practical OBDI. Moreover efficient UCQ rewriting using views will allow rewriting entire outputs of the query expansion phase as a batch, rather than each one in isolation, optimizing our ontology-based data integration solution even more.
- In parallel, the two major contributions of our relational algorithm,i.e., compact representation of common patterns and offline preprocessing can apply also for the case of query expansion. These insights will allow for a faster query expansion phase that yields a compact output which additionally is tailored for our query reformulation algorithm.
- Coupling the two phases together will (1) yield a more efficient algorithm, (2) avoid redundant work by not rewriting similar or redundant ontology-expanded queries using the views multiple times, and (3) allow us to deal with the challenging case of LAV (and GLAV) mappings, reusing our insights and leveraging the benefits of our relational approach.

2. THE DATA INTEGRATION PROBLEM

Answering Relational Queries Using Views. To define the problem formally we introduce the concepts of query containment [6, 1] and query rewritings [13]. Initially, we focus on conjunctive queries; the core of every query language. Their body is a conjunction of atoms and they correspond to select-project-join queries.

Definition 1 (Query Containment): For queries Q_1,Q_2 defined over the same schema, Q_1 is contained in 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, $Q_1(D) \subseteq Q_2(D)$.

Chandra and Merlin [6], showed that conjunctive query containment is NP-complete, and can be done through *containment mappings*. For two same schema conjunctive queries, 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, sets of 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)$h(head(Q_1))$ = $head(Q_2)$ (modulo the names of Q_1, 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 [6].

Q_1 is *equivalent* to Q_2, denoted $Q_1 = Q_2$ iff $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. Given a query Q and a set of view definitions $\mathcal{V} = \{V_1, ..., V_n\}$ both over the same schema R, a query Q' is a *rewriting* of Q using $\mathcal{V}$ if Q' uses *only* predicates from $\mathcal{V}$. In our context we are looking for all maximally-contained query rewritings:

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

In order to check whether a rewriting Q' (defined over $\mathcal{V}$) is contained in a query Q (defined over schema $\mathcal{R}$), we need to get the rewriting's *unfolding* [13], i.e., unfold the atoms of $\mathcal{V}$ in the body of Q' with their definitions (which are over $\mathcal{R}$). The new query $unfold(Q')$ is defined over R, and we can check wether $unfold(Q') \subseteq Q$. To ground these definitions consider the following example. Assume that we have two LAV sources, S_1 and S_2, that provide information about healthcare medical records. S_1 contains doctors that treat patients with a chronic disease. S_2 contains doctors, patients and clinics where the doctor is responsible for discharging the patient from the specific clinic. The contents of these sources are modeled respectively by the two views:

S_1: V1(doctor, disease) $\rightarrow$ TreatsPatient(doctor, patient), HasChronicDisease(patient,disease)

S_2: V2(doctor, patient, clinic) $\rightarrow$ DischargesPatientFromClinic(doctor, patient, clinic)

We use a logical notation for queries and views similar to datalog [1]. Logical implication is denoted by $\rightarrow$ or $\leftarrow$, and conjunction between atoms by ','. Same variable names used in two predicates denotes equality of the corresponding arguments of the predicates, within this source (variables across sources are considered different). On the left side of a rule, lies the *head*; the relation the view or the query contain or ask for respectively. On the right side we write the *body* of the view/query, which is its description. It is important to stretch out that the bodies of such rules consist of mediator predicates which stand for virtual relations. The actual data are provided by the head of a view. 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 (it could be missing or considered private). Assume the user asks for doctors treating chronic diseases and the clinics that they work (discharge patients from):

q(d, c) $\leftarrow$ TreatsPatient(d,x), HasChronicDisease(x,y), DischargesPatientFromClinic(d,z,c)

A rewriting of q is:

q'(d, c) $\leftarrow$ V1(d, y), V2(d, z, c)

Intuitively, we can get the chronic diseases' doctors information from V1 and join it on "doctor" with the patient discharging information from V2. In this example, the selection of views relevant to answer the user query and the reformulation process was quite simple since there were only two views. We just built q' using the two views and tested that $unfold(q') \subseteq q$. Here, $unfold(q')$ $\leftarrow$ TreatsPatient(d,p), HasChronicDisease(p,y), DischargesPatientFromClinic(d,z,c). 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. However, in general, there may be a large number of sources and the number of possible rewritings that would need to be tested for containment grows exponentially to this number of sources. Algorithms in this area, as discussed in Sect. 3, try to reduce the number of candidate rewritings. Notice that, in our example, we had to select views that contained a predicate needed by the query: the definition of V1 involves TreatsPatient and HasChronicDisease, and V2 involves DichargesPatientFromClinic. This is, however, not a sufficient condition for selecting a view. Had the attribute "doctor" been missing from V1 or V2, they would be useless (since the query asks for the "doctor" attribute). In effect, we want "doctor" to be a *distinguished* (returning) variable in the relevant views. If a variable is not in the head of the query (i.e., returning or distinguished), we call it *existential*.

Ontology-Based Data Integration. To introduce the OBDI problem, we employ the description logic syntax [3] which we will here use as the mediator language. In this syntax, consider the following rules as a part of the mediator ontology exposed to the user: (1) Dentist $\sqsubseteq$ Doctor, (2) Doctor $\sqsubseteq$ $\exists$ TreatsPatient, and (3)

$\exists$HasDoctor$^-$ $\sqsubseteq$ Doctor. These axioms state that dentists are doctors, that doctors participate at least once in the TreatsPatient relation, and they are the range of the HasDoctor relation. Given a user query q on the ontology (written over the ontology's predicates) and a set of (relational) views, the problem is again to obtain all maximally-contained query rewritings of q using only the views. Towards this, related algorithms consider the problem in two separate phases [5, 14, 16]. The first phase, namely query expansion, is to rewrite the user query taking the ontology inferences into account. Consider for example the query on the mediator being: q_1(x) $\leftarrow$ TreatsPatient(x,y). Upon examination the reader can verify that the complete query in this example (that is, closed with respect to ontology inferences) is the union of the original query plus the following additional "entailed" ones:

q_1(x) $\leftarrow$ Doctor(x), q_1(x) $\leftarrow$ Dentist(x), q_1(x) $\leftarrow$ HasDoctor(y,x).

Intuitively if asking for all individuals that treat some patient, and given the incompleteness of the sources in the data integration context, we should also query the sources for all doctors (since all doctors treat patients by axiom 2), and hence all dentists and all the ranges of the HasDoctor relation (axioms 1 and 3 resp.).

Note that all these ontology-expanded queries use mediator predicates, and so we need a second phase in order to reformulate them with respect to the views. Most approaches assume the trivial case of GAV mappings as the source descriptions available after the ontology is "compiled" in the query. GAV reformulation using the views is straightforward; one needs simply to "unfold" (i.e., substitute) the ontology concepts and relations that appear in the expanded queries with the source queries that define them on the bodies of the corresponding mappings. Relevant algorithms usually focus on optimizing the "ontology compilation" phase. The union of ontology-compiled queries is exponential to the size of the ontology, and a lot of the queries in this union are possibly redundant. Hence the biggest focus of recent algorithms has been to minimize the output of the query expansion.

3. OUR PREVIOUS WORK: GQR

Recent approaches [15, 2] in query rewriting using LAV views have focused on pruning the selection of views that will potentially form the rewriting, so as not to result in a non-contained rewriting to the query. In [11], we pushed this intuition even further and managed to develop a much more efficient and scalable solution. This section summarizes our results, and defines the foundations for our proposed approach. The reader should refer to [11], for the full details of our approach and algorithms.

We will start with some preliminaries. We introduce *coverings*, which are restrictions of containment mappings. Coverings map a sub-part of the query body to a sub-part of a view. Recall that a query 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. As we explain later, all relevant algorithms, look for legitimate coverings in order to select a view for participation in a rewriting.

Definition 3 (Covering): For all queries Q, for all views V, for all predicates $g_q \in body(Q)$, for all predicates $g_v \in body(V)$, for all 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.

The second part of Def. 3 is exactly condition (2) in the containment mapping definition, and the intuition behind it, is that whenever a part of a query needs a value, you cannot cover that part with a view that does not explicitly provide this 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 sources defined in the example of Sect. 2 and q_2 below which asks for doctors that treat patients with chronic diseases and the clinics where they discharge those *same* patients from:

q_2(d, c) $\leftarrow$ TreatsPatient(d,x), HasChronicDisease(x,y), DischargesPatientFromClinic(d,x,c)

In contrast to q of Sect. 2, this query demands 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 (i.e., $patient$ in its definition). 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*. This property is referred to as (clause $C2$ in) *Property 1* in MiniCon[15]. This is also the basic idea of the MiniCon algorithm: trying to map all query predicates of q_2 to all possible views, it will notice that the existential query variable x in the query maps on $patient$ 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 *Discharges-PatientFromClinic(d,x,c)* cannot. We notice that there is duplicate work being done in this process. First, MiniCon does this procedure *for every query predicate*, this means that if q_2 had multiple occurrences of $TreatsPatient$ it would try to use S_1 multiple times and fail (although as [15] states *certain* repeated predicates can be ruled out of consideration). Second, MiniCon would try to do this for *every possible view*, even for those that contain the same pattern of S_1, as S_3 below which offers doctors and the diseases they treat on some patient, where the doctors are also dentists:

S_3: V3(doctor, disease) $\rightarrow$ TreatsPatient(doctor, patient), HasChronicDisease(patient,disease), Dentists(doctor)

S_3 cannot be used for q_2 as it violates MiniCon's Property1, again due to its second variable, $patient$, being existential and *DischargesPatientFromClinic* not covered. Our idea is to avoid this redundant work by compactly representing all occurrences of the same view pattern. Our algorithm (called Graph-based Query Rewriting or GQR [11]) compactly represents common subexpressions in the views. Instead of considering every view subgoal, we only consider the distinct patterns that all views contain. We start by finding coverings for small atomic view patterns, that repeat themselves accross views and hence compactly represent pieces of multiple views (which alternatively cover the same query part). We then incrementally combine these patterns to larger ones, progressively covering the underlying query. Consequently, we naturally come up with a "batch" of contained rewritings, right away. In our solution, we use a graph representation of queries and views presented subsequently.

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. Shared variables between atoms result in shared variable nodes, directly connected to predicate nodes. We equip our edges with integer labels that stand for the variables' positions within the atom's parentheses, and we discard variables' names; the latter is because the only knowledge we require for deciding on a covering is the types of the variables involved. Distinguished variable nodes are depicted with a circle, while for existential ones we use the symbol $\otimes$. Using these constructs the query q of Sect. 2 (with abbreviated predicate names for brevity, i.e., $q(d, c) \leftarrow TP(d, x)$, $HCD(x, y)$, $DPFC(d, z, c)$) corresponds to the graph seen on

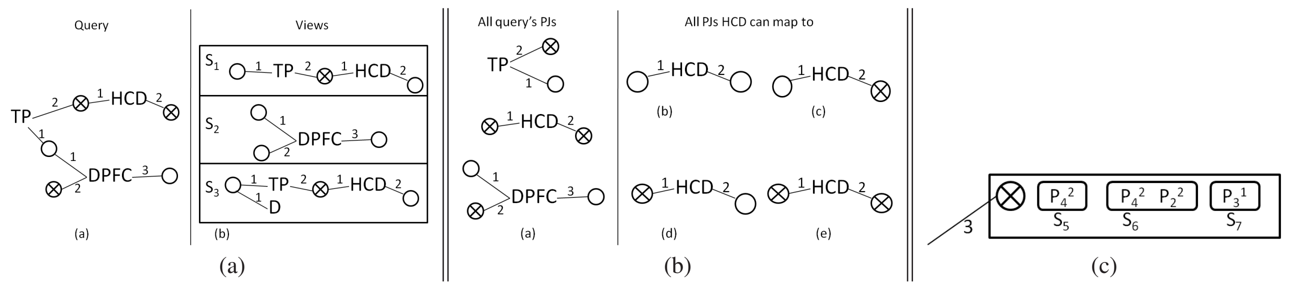

Figure 1: (a) Query Q, and sources S_1-S_3 as a graphs. (b) Predicate Join Patterns. (c) Infobox for a variable node. The node is existential and is attached on its predicate node on edge with label 3 (this variable is the third argument of the corresponding atom). We can find this specific PJ in three views, so there is information about the join descriptions of each of these views in the infobox. The two join descriptions associated to S_6 tell us that this variable, in view S_6, joins with the second argument of P_4 and the second argument of P_2.

the left part of Fig. 1(a). The right part of Fig. 1(a) shows the graph representation (again with abbreviated predicate names) of our running example's LAV source descriptions S_1, S_2, and S_3. Our algorithm consists of mapping subgraphs of the query 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 are called *predicate join patterns* (or PJs) for the predicate they contain. Left part of Fig. 1(b) shows all predicate join patterns that the query Q contains, (i.e., all *query PJs*).

A critical feature that boosts our algorithm's performance is that the patterns of predicates as graphs repeat themselves in multiple sources. Therefore we choose to compactly represent each such occurrence of the same predicate across different 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 source description. For the query PJ for HCD seen in the left part of Fig. 1(b), all source PJs that could potentially cover it, per Def. 3[2], can be seen in the right part of the same figure. Unless our sources contain one of these four patterns the query fails (right away) to be rewritten.

Nevertheless, the "join conditions" for a particular PJ within each view are different and some "bookkeeping" is needed to capture these joins. To retain this information we use a conceptual data structure called *information box* (or infobox). Each infobox is attached to a variable v. Fig. 1(c) shows an example infobox for a variable node. A variable's infobox contains a list of views that this PJ appears in and for each such view the variable's *join descriptions* which record which other PJs this variable (directly) joins to within the specific view. Fig. 2 shows for all predicates of Q, all the different relevant PJs that appear in sources $S_1 - S_3$ with their infoboxes (aggregating information from all sources where they appear). Additionally, at different steps of our algorithm, each source graph consisting of PJs, covers a certain part of the query and within this graph we maintain a list of "candidate" parts of the final conjunctive rewritings (that will eventually be "responsible" for covering this part of the query). We call these *partial* conjunctive rewritings.

3.2 Graph-based query rewriting

Our solution is divided in two phases. Initially, we process all view descriptions and construct all source PJs. In our second phase, we start by matching each atomic query subgoal to the source PJs and we go on by combining the relevant source PJs to form larger subgraphs that cover larger "underlying" query subgoals.

[2]Def. 3 translates using our graph terminology: A view PJ *covers* a query PJ if there is a graph homomorphism from the query graph to the view one (preserving predicates and edges), s.t. it maps distinguished query variables to distinguished view ones

One of the core ideas of our approach is that our preprocessing phase does not need any information from the query, as we designed it to involve only views. This preprocessing constructs (1) unique PJs for all common patterns that appear across the views, (2) their infoboxes and (3) their initial partial rewritings. Although this phase has a polynomial complexity to the number and length of the views one can create more sophisticated indices on the source PJs, at the expense of space and additional offline time. For our prototype implementation, we are creating an exponential index on the source PJs so as to be able to retrieve the relevant to the query ones very efficiently on runtime (in essence we create all different potential query PJs that could map on the sourcePJs at hand). As discussed in Sect. 4 we plan to elaborate on different offline indexing approaches and research on offline vs. runtime trade-offs.

After the source indexing phase, the first thing we do when a query is given to the system is to retrieve all the relevant source PJs that cover each query PJ[3]. During this retrieval we perform some pruning on the PJs returned, based on the distinguished-existential allowed mappings, as well as the join descriptions in their infoboxes. We might, for example, prune some of the views out of a PJs infobox, in case that a pattern appears in a specific source, but the specific join descriptions in this source do not satisfy the underlying query's join descriptions (we do this to satisfy Minicon's Property 1 discussed in the beginning of this section). This leads to a fail-fast behavior of our algorithm.

Subsequently, we start exhaustively combining PJs forming larger ones which progressively cover a larger part of the underlying query. The order of source patterns combination (or alternatively the order of query predicates we choose to cover) is currently random. We plan to do further research on good heuristics on this order which would improve our algorithm's performance even more. During the combination procedure pruning is done again. Moreover, even if two patterns are combinable this does not mean that all the views in their infoboxes are combinable to each other, as they need to satisfy the corresponding underlying query join descriptions, so we may again have to prune some of the infoboxes information (as well as some of the partial rewritings). As we combine source graphs (progressively covering larger parts of the query) we also combine the remaining partial conjunctive rewritings they contain into larger partial rewritings, eventually producing the maximally-contained rewritings of the query. Due to space limitations we omit the relevant algorithms (see [11]). For our running example (query q, with sources S_1, S_2, S_3) Fig. 2 gives a schematic intuition behind this procedure. As seen from the figure the resulting rewritings are: q(d,c) $\leftarrow$ V1(d,y),V2(d, z, c) and q(d,c) $\leftarrow$ V3(d,y),V2(d, z, c).

[3]More details of our approach and the exact algorithms can be found in [11].

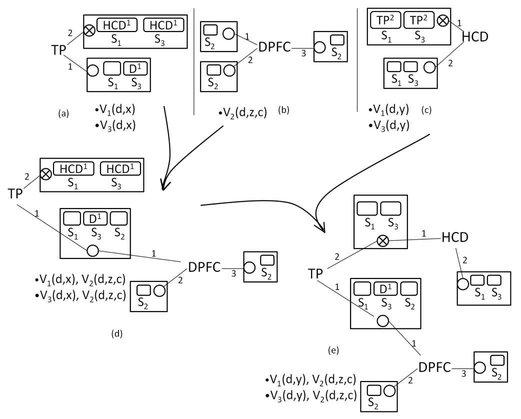

Figure 2: The PJs that cover TP are combined with the PJs that cover $DPFC$, and the PJs that cover HCD. Here there is only one source PJ covering each query predicate but, in general, we try all combinations of PJs that cover the query. The figure does not show source PJs for predicate D (predicate Dentist in source S_3; this will not be retrieved as the query does not mention dentists). In the current example, PJ (a) is combined with (b), resulting in (d), and subsequently (d) combined with (c) cover the entire query. The union of the rewritings of the resulting graph (e) is our solution.

3.3 Initial Results

For evaluating our prototype approach we compared with the most efficient (to the best of our knowledge) state-of-the-art algortihm, MCDSAT [2]. We should note that MCDSAT is a satisfiability translation of MiniCon. Hence the inside advantages that we discussed against MiniCon apply to MCDSAT as well. We show our performance in Fig. 3 for two kinds of randomly generated queries/views: star and chain queries. In all cases GQR outperforms MCDSAT even by close to two orders of magnitude. Moreover, GQR runs in seconds for thousands of views, producing thousands of rewritings. Our approach is very scalable, as it "fails" very fast (when output rewritings are not produced, the algorithm realizes that quickly). As seen from the figures the time of reformulation at runtime clearly depends more on the size of the output than that of the problem. Moreover, we save a big overhead by choosing to have an offline source preprocessing phase; Fig. 3 shows that when the number of views grows substantially, the exponential time of the preprocessing phase dominates. See [11] for details, more experiments, discussion and the experimental setting.

As discussed throughout this section our design for an incremental covering of the query is ideal for early pruning of irrelevant view patterns. Moreover, this is a "batch" pruning: due to our compact representation entire sets of irrelevant views are pruned out simultaneously. On top of that, the same compact representation of view patterns leads to a compact and very efficient view combination phase (for the really relevant views). Lastly, this design together with the off-line preprocessing makes our algorithm able to scale to tens of thousands of views, where no algorithm scaled before.

4. TOWARDS SCALABLE AND GENERAL MEDIATORS

Query rewriting under relational constraints. We have some impressive results on query rewriting using LAV sources and we are extending to solve the full relational case, as follows. A first step is to increase the expressiveness of the language of conjunctive queries that we use to describe our queries and views: we plan to add constant symbols and interpreted predicates, which were absent from our approach. Constants would be represented by a third different type of node in our graphs, and constants in the query should be covered either by the same constants in the views, or by distinguished variables (so we get a hold on them and "manually" set the constant value). Interpreted predicates ($\leq$,$=$,$\neq$, etc.) are more tricky. Our feeling is that we can use them as constraints and early detect irrelevant rewritings in our incremental building. We also plan to investigate different heuristics to come up with an efficient order of combination of source patterns, or equivalently an efficient order of selecting how to cover the query predicates.

We intend to leverage our incremental covering of the query to do on-the-fly optimization on our output; we plan to recognize and delete redundant predicates or partial rewritings, early on in our PJ combination phase. Incrementally checking for containment is inherent in our algorithm, and so we hope to attack the optimization problem at little additional cost.

Currently, we have used a naive exponential indexing of the source PJs, that allows for a very efficient runtime retrieval of the relevant ones. Next, we plan to design more clever indices; we can offline index the available source PJs on a lattice capturing the generality of their variables' distinguished/existential patterns, where a source PJ higher in the hierarchy will have more distinguished variables. To retrieve the source PJs matching a query PJ, the system will traverse the generalization lattice down from the root, until the frontier of nodes that cannot cover the query PJ. Another approach is to leverage existing relational technology for our search of relevant PJs. When faced with a query PJ we essentially ask (as seen in Fig. 1(b)) for source PJs with the same or a more general pattern (one that has distinguished variables where our query has existential ones). Let 1 stand for distinguished variables and 0 for the existential ones. We can arrange our PJs in a relational table, with each row corresponding to each existential/distinguished (0/1) pattern of each source. Then, it suffices to query for the source PJs that provide distinguished variables in the same position as the query PJ. Both approaches will avoid the exponential indexing and we will compare and use the one has the most efficient runtime PJ retrieval.

As already mentioned, we plan to extend our approach to GLAV rules. Also known as tuple-generating-dependencies (or tgds) [1], these rules look like LAV but have more than one view predicates in the head. We plan to investigate two alternatives here. The first, is to rewrite the GLAV rules using some temporary intermediate predicates, into a combination of GAV and LAV rules; then we can rewrite using the LAV mappings and unfold using the temporary GAV ones. Since the same view predicate will generally participate in multiple heads of the GLAV rules, it will possibly redundantly end up in multiple places in a single rewriting, so we again have to employ our envisioned incremental optimization technique. A parallel idea, is to use directly the entire conjunction of source predicates of the rules' heads in our partial rewritings. This is similar as using intermediate predicates, but will allow for more fine-grained optimization. Our plan is to evaluate both aforementioned approaches and keep the best one. Note that our system will able to work with all kinds of G/LAV mappings simultaneously.

After the above extensions, and towards richer mediator languages, we want to address unions of conjunctive queries (UCQs) as the input user query. To the best of our knowledge, no system or specific algorithm addresses UCQs in a unified way. This will allows us to also support nr-datalog programs. Our graph-encoding of PJs, is perfectly fitted for capturing overlapping parts of multiple rules. Hence, we plan to exploit this design to compactly represent UCQs as inputs of our algorithm. This is especially beneficial when

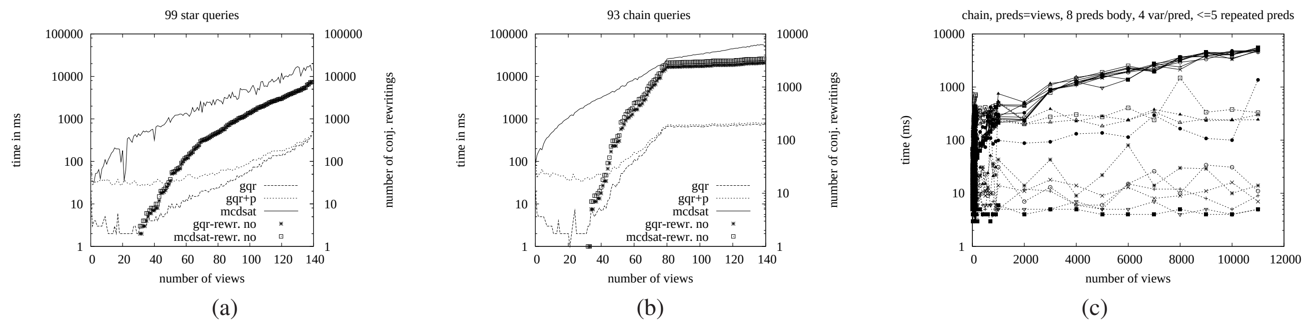

Figure 3: (a) Average time and size of rewritings for star queries. GQR time does not take into account the source preprocessing time while gqr+p does. As we can see the preprocessing phase (for this small set of views) does not add much to our algorithm; the query reformulation phase is dominating the time as the number of views increases. (b) Average time and size of rewritings for chain queries. The data were generated in a way s.t after the point of 80 views, while the problem size continues to grow linearly, the output (number of rewritings) grows very slowly. (c) Ten chain queries on views constructed so as the rewritings don't grow exponentially. The upper bunch of straight lines gives the total time for the queries, while the lower dotted part gives only the reformulation time. The number of rewritings is proportional to the dotted lines and ranges from several hundreds to several thousands (see in [11]).

considering ontological constraints, since UCQs produced by the query expansion phase are usually highly redundant [16].

Query rewriting under ontological constraints. We will consider different "interesting" ontology languages, such as DL-lite [5], various OWL2 profiles[4] and fragments of Datalog+/- [4], all of which are sweet-spots between high expressivity, and being able to expand into first-order queries (equivalent to SQL).

We plan to use our common pattern representation idea to the query expansion phase as well. Having a compact representation of the ontology axioms will allow us to design a (1) faster expansion algorithm, that will (2) avoid redundancy in the output, and so save time for subsequent reformulation phases (both GAV and LAV). It will also (3) help us integrate the output specifically with GQR and hence take advantage of its optimizations and performance, and (4) possibly lead us to design an algorithm that does query expansion at the same time as query reformulation using the views. A first idea towards building graph patterns that "capture" the ontology axioms is rewriting the latter into logical clauses in the spirit of [14].

The aforementioned approach couples the two OBDI phases in a top-down manner, by optimizing query expansion and tailoring it to relational query reformulation using views. However, we plan to explore deeper forms of integration. We are particularly interested in being able to check for partial containment of ontological queries (in the spirit of coverings, see Def. 3), which might be easier than full containment, which requires the expansion of one of the queries. Being able to check for partial containment without expanding any queries, would mean that GQR, would be almost directly applicable in the ontological context: our coverings would be partial ontological containments from source PJs to the query PJs. Combining source PJs in a legitimate way as to maintain the containment would end up in maximally contained (w.r.t. the constraints) rewritings using the LAV sources. This approach enjoys all the benefits discussed, ranging from the use of GLAV rules, to offline preprocessing of the ontological views and from incremental optimization of our output to support for UCQs as inputs.

With our proposed approach queries would be rewritten much faster (in correspondence with our initial experiments for conjunctive queries) and systems will be much more scalable. We hope to build a mediator capable of offering languages of adjustable expressivity, ranging form relational to rich ontological constraints, and all with efficient runtime cost.

[4]http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/

5. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Citeseer, 1995.

[2] Yolifé Arvelo, Blai Bonet, and María Esther Vidal. Compilation of query-rewriting problems into tractable fragments of propositional logic. In *Proc. of AAAI'06*.

[3] F. Baader. *The description logic handbook: theory, implementation, and applications*. Cambridge Univ Pr, 2003.

[4] A. Calì, G. Gottlob, T. Lukasiewicz, and A. Pieris. A logical toolbox for ontological reasoning. *SIGMOD Record, 2011*.

[5] D. Calvanese, G. De Giacomo, 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.

[6] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc of the 9th ACM Symposium on Theory of Computing,1977*.

[7] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proc. of PODS '97*.

[8] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data exchange: Semantics and query answering. *In Proc. of ICDT 2003*.

[9] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Integrating and accessing heterogeneous information sources in tsimmis. In *Proc. AAAI Symposium on Information Gathering, 1995*.

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

[11] G. Konstantinidis and J.L. Ambite. Scalable query rewriting: a graph-based approach. In *Proc. of SIGMOD 2011*.

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

[13] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views (extended abstract). In *Proc. of PODS 1995*.

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

[15] R. Pottinger and A. Halevy. MiniCon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 2001.

[16] R. Rosati and A. Almatelli. Improving query answering over dl-lite ontologies. *Proc. of KR*, 2010, 2010.