Scalable Query Rewriting: A Graph-Based Approach

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

In this paper we consider the problem of answering queries using views, which is important for data integration, query optimization, and data warehouses. We consider its simplest form, conjunctive queries and views, which already is NP-complete. Our context is data integration, so we search for maximally-contained rewritings. By looking at the problem from a graph perspective we are able to gain a better insight and develop an algorithm which compactly represents common patterns in the source descriptions, and (optionally) pushes some computation offline. This together with other optimizations result in an experimental performance about two orders of magnitude faster than current state-of-the-art algorithms, rewriting queries using over 10000 views within seconds.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; H.2.5 [**Database Management**]: Heterogeneous Databases; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Hypergraphs*

General Terms

Algorithms, Performance, Experimentation

1. INTRODUCTION

Given a conjunctive query Q, over a database schema D and a set of view definitions $V_1, ..., V_n$ over the same schema, the problem that we study is to find answers to Q using only $V_1, ..., V_n$.

This is an important problem extensively studied in the contexts of query optimization, data integration and other areas [13, 15]. From the query optimization perspective, views are previous queries who have been already evaluated and their answers have been materialized. In this context, systems rewrite subsequent queries substituting *a part* of the original query with some of the views, so as to come up with an *equivalent* optimized query.

In our context, of data integration, multiple heterogenous sources (as web sites, databases, peer data etc.) are integrated under a global query interface. These sources are usually presented as relational schemas and the system offers a virtual mediated schema to the user for posing queries. Then, the system needs to rewrite user queries as queries over the sources' schemas *only*. Moreover, some sources might be incomplete; hence the system needs to produce query rewritings that instead of equivalent are *maximally-contained*.

Mappings between the sources' schemas and the mediator schema are usually given in the form of logical formulas, which we call *source descriptions*. One approach for describing the sources, known as *global-as-view* (GAV) [11], is to have the mediated relations expressed as views over the schema that the sources' relations constitute. An alternative approach is using *local-as-view* (LAV) mappings [16, 10], where each source relation is expressed as a view over the mediator schema.

In this paper, we address the problem of rewriting a conjunctive query using LAV descriptions, to a maximally-contained union of conjunctive rewritings; this is fundamental to data integration and related areas. To align the problem with the recent frontier of data integration research we should mention current works that build and extend on LAV rewriting algorithms (e.g., [18, 2, 14]) or use their intuitions [20, 6]). There are also recent works that build on LAV rewriting foundations (e.g., in composition of schema mappings [6, 4] or in uncertain data integration [3]) or assume the off-the-self usage of a LAV rewriting algorithm (e.g., [6, 5]). It is critical for all data integration systems to be able to support a large number of sources, so our solution focuses on *scalability* to the number of views.

Generally, the number of rewritings that a single input query can reformulate to, can grow exponentially to the number of views, as the problem (which is NP-complete [16]) involves checking containment mappings[1] from subgoals of the query to candidate rewritings in the cross-product of the views. Previous algorithms (as MiniCon [19] and MCDSAT [7]) have exploited the join conditions of the variables within a query and the views, so as to prune the number of unnecessary mappings to irrelevant views while searching for rewritings.

We pursue this intuition further. The key idea behind our algorithm (called Graph-based Query Rewriting or GQR) is to compactly represent common subexpressions in the views; at the same time we treat each subgoal atomically (as the bucket algorithm [17]) while taking into account (as MiniCon does) the way each of the query variables interacts with the available view patterns, and the way these view patterns interact with each other. Contrary to previous algorithms however, we don't try to a priori map entire "chunks" of the query to (each one of) the views; rather this mapping comes out naturally as we incrementally combine (relevant to the query) atomic view subgoals to larger ones. Consequently, the second

SIGMOD'11, June 12–16, 2011, Athens, Greece.

[1]Not to be confused with source descriptions which are mappings from the sources' schema to that of the mediator.

phase of our algorithm needs to combine much fewer and really relevant view patterns, building a whole batch of rewritings right away. Our specific contributions are the following:

- We present an approach which decomposes the query and the views to simple atomic subgoals and depicts them as graphs; we so abstract from the variable names by having only two types of variable nodes: distinguished and existential.
- That makes it easier to identify the same graph patterns across sources and compactly represent them. This can be done offline, as a view preprocessing phase which is independent of the user query and can be done at any time the sources become available to the system, thereby speeding up system's online performance.
- Subsequently we devise a query reformulation phase where query graph patterns are mapped (through much fewer containment mappings) to our compact representation of view patterns. By bookkeeping some information on our variable nodes (regarding their join conditions) we can combine the different view subgraphs to larger ones, progressively covering larger parts of the query.
- During the above phase each graph "carries" a part of the rewriting, so we incrementally build up the rewriting as we combine graphs; we conclude with a maximally-contained rewriting that uses only view relations, while at the same time we try to minimize its size.
- Our compact form of representing the pieces of the views allows us to also reject an entire batch of irrelevant views (and candidate rewritings), by "failing" to map on a view pattern. This also allows the algorithm to "fail-fast" as soon as a query subgoal cannot map to any of the (few) view patterns.
- These characteristics make our algorithm perform close to two orders of magnitude faster than the current state-of-the-art algorithm, MCDSAT [7]. We exhibit our performance by reformulating queries using up to 10000 views.

In Sect. 2 we present some preliminary notions about queries and containment and introduce the necessary definitions for the problem. Our graph representation of queries and sources is presented in Sect. 3 and our algorithm for scalable query rewriting in Sect. 4. Our experimental results against MCDSAT are given in Sect. 5. In Sect. 6 we compare against related work. We conclude in Sect. 7 where we also state our plans for future work.

2. THE QUERY REWRITING PROBLEM

This section will formally define our problem after going through some necessary preliminary definitions. We use the well-known notions of constants, variables, predicates, terms, and atoms of first-order logic. We use safe conjunctive queries; these are rules of the form $Q(\vec{x}) \leftarrow P_1(\vec{y_1}), ..., P_n(\vec{y_n})$ where Q,P_1, ..., P_n are predicates of some finite arity and $\vec{x}$,$\vec{y_1}$,...,$\vec{y_n}$ are tuples of variables. In the scope of the current paper (and similarly to [7]) we have not considered constant symbols or built-in predicates. We believe however, that our results can be extended to these cases (in the spirit of [19]). We define the body of the query to be $body(Q) = \{P_1(\vec{y_1}), ..., P_n(\vec{y_n})\}$. Any non-empty subset of $body(Q)$ is called a *subgoal* of Q. A singleton subgoal is an *atomic subgoal*. $Q(\vec{x})$ is the *head* of the query.

Predicates appearing in the body stand for relations of a database D, while the head represents the answer relation of the query over D. The query being "safe" means that $\vec{x} \subseteq \bigcup_{i=1}^{n} \vec{y_i}$. All variables in the head are called *head*, *distinguished*, or *returning* variables, while the variables appearing only in the body (i.e., those in $\bigcup_{i=1}^{n} \vec{y_i} \setminus \vec{x}$) are called *existential* variables. We will call *type* of a variable its property of being distinguished or existential. For all sets of atoms S, $vars(S)$ is the set of variables appearing in all the atoms in S (e.g., $vars(Q)$[2] is the set of all query variables). $Q(D)$ refers to the result of evaluating the query Q over the database D.

A *view* V is a named query. The result set of a view is called the *extension* of the view. In the context of data integration a view could be incomplete, in the sense that its extension could only be a subset of the relation $V(D)$. Users pose conjunctive queries over D and the system needs to *rewrite* or *reformulate* these queries into a union of conjunctive queries (UCQ) that only use the views' head predicates so as to obtain all of the tuples in $Q(D)$ that are available in the sources. We will refer to this UCQ as the query *rewriting* while an individual conjunctive query in a rewriting will be called a *conjunctive* rewriting [7]. Query rewriting is closely related to the problem of query containment [16].

We say that Q_2 is *contained* in Q_1, denoted by $Q_2 \subseteq Q_1$, iff for all databases D, $Q_2(D) \subseteq Q_1(D)$. If additionally there exists one database D' such that $Q_2(D') \subset Q_1(D')$, we say that Q_2 is *strictly* contained in Q_1 (denoted $Q_2 \subset Q_1$). If queries Q_1, Q_2 are both contained in each other then they are *equivalent* ($Q_1 \cong Q_2$) and produce the same answers for any database. For all 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 [8]. Given two queries Q_1, Q_2, a *containment mapping* from Q_1 to Q_2 is a homomorphism 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). Deciding query containment for two conjunctive queries is an NP-complete problem [8].

Data integration systems operate with the open-world assumption; they assume that extensions of views might miss some tuples or that the exact information needed by the query cannot be provided by the views. In that case we would like to provide the maximal available subset of the needed information given the views we have. Note that we demand *complete* rewritings in the sense that they contain only view head predicates [15].

DEF. 1. ***Maximally-contained, complete rewriting:*** *For all databases D, for all sets of views over D, $V = \{V_1, ..., V_n\}$, for all queries R, Q: R is a maximally-contained and complete rewriting of Q using V, iff*

- *Q is a conjunctive query over D and R is a UCQ using only head predicates of V, and*
- *for all (possibly incomplete) extensions of the views u_1,...,u_n, where $u_i \subseteq V_i(D)$ it is the case that $R(u_1, ..., u_n) \subseteq Q(D)$, and*
- *there does not exist a query R' such that $R(u_1, ..., u_n) \subset R'(u_1, ..., u_n)$ and $R'(u_1, ..., u_n) \subseteq Q(D)$*

Our algorithm finds a maximally-contained and complete rewriting R, for the problem of answering a conjunctive query Q with a set of conjunctive views. Taking notice at the containment mapping definition we see that in order to check query containment between R and Q, we need them to be over the same schema. This is done by taking the expansions of the conjunctive rewritings in R. Given $r \in R$, we define the *expansion* or r, $exp(r)$, to be the conjunctive query we obtain if we unfold the views, i.e., substitute the view heads in the body of r with their descriptions (i.e, the bodies of those view definitions). Note that, when the views get unfolded,

[2]When clear from the context we'll use Q or V to refer to either the datalog rule, the set of atoms of the rule, or just the head.

the existential variables in the view definitions are renamed so as to get *fresh* variables.

For speeding up the actual query evaluation on the sources, query rewriting algorithms pay attention to finding minimal conjunctive rewritings inside R. More formally we can define a "minimization" ordering $\leq_m$ as follows:

DEF. 2. ***Minimal conjunctive rewriting:*** *For all conjunctive queries R_1, R_2: $R_1 \leq_m R_2$ iff*

- $R_1 \cong R_2$, *and*
- *there exists a set $U \subseteq vars(body(R_2))$ and there exists an isomorphism $i : vars(body(R_1)) \rightarrow U$ (i is extended in the obvious manner to atoms) such that (1) for all atoms $A \in body(R_1)$ it holds that $i(A) \in body(R_2)$, and (2) $i(head(R_1)) = head(R_2)$.*

The problem of minimizing a rewriting is NP-complete [16] and therefore most algorithms produce a number of non-minimal conjunctive rewritings in their solutions. An additional problem related to minimality is that of *redundant* rewritings, when more than one equivalent conjunctions exist in the same UCQ rewriting. Our algorithm produces fewer conjunctive rewritings than the current state-of-the-art algorithm, but we also suffer from redundant and non-minimal ones. A containment mapping from a query Q to a rewriting R is also seen as the *covering* of Q by (the views in) R. Similarly we can define:

DEF. 3. ***Covering:*** *For all queries Q, for all views V, for all subgoals $g_q \in body(Q)$, for all subgoals $g_v \in body(V)$, for all partial homomorphisms $\varphi : vars(Q) \rightarrow vars(V)$, we say that a view subgoal g_v covers a subgoal g_q of Q with φ iff:*

- $\varphi(g_q) = g_v$, *and*
- *for all $x \in vars(g_q)$ if x is distinguished then $\varphi(x) \in vars(g_v)$ is distinguished.*

The intuition behind the second part of the definition is that whenever a part of a query *needs* a value, you can not cover that part with a view that does not explicitly provide this value. On occasion, we might abuse the above definition to say that a greater subgoal, or even V itself, covers q_q with φ (since these coverings involve trivial extensions of φ). For all variables $x \in g_q$ and $y \in g_v$ we say that x *maps* on y (or y covers x) iff for a covering involving φ, $\varphi(x) = y$.

To ground these definitions consider the following example. Assume that we have two sources, S_1 and S_2, that provide information about road traffic and routes (identified by a unique id). S_1 contains ids of routes one should avoid; i.e., routes for which there is at least one alternative route with less traffic. S_2 contains points of intersection between two routes. The contents of these sources are modeled respectively by the two following LAV rules (or views):

$S_1(r_1) \rightarrow$ AltRoutes(r_1,r_2), LessTraffic(r_2, r_1)

$S_2(r_3, r_4, p_1) \rightarrow$ ConnectingRoutes(r_3, r_4, p_1)

Assume the user asks for all avoidable routes and all exit points from these routes:

$q(x, p) \rightarrow$ AltRoutes(x,y), LessTraffic(y,x), ConnectingRoutes(x,z,p)

The rewriting of q is: $q'(x, p) \leftarrow S_1(x), S_2(x, f, p)$

In this example, the selection of relevant views to answer the user query and the reformulation process was quite simple since there were only two views. We just built a conjunctive query q' using the two views and tested that exp(q')$\subseteq q$, where exp(q') is:

$q''(x, p) \rightarrow$ AltRoutes(x,f_1),LessTraffic(f_1, x),ConnectingRoutes(x, f, p)

However, in general, there could be many more conjunctive rewritings in the rewriting than q', and as mentioned checking containment is an expensive procedure. Notice that in order to construct a conjunctive rewriting contained in the query, we select views that have relevant predicates, i.e., that there is a mapping (covering) from a query atom to the view atom. Coverings are essentially components of the final containment mapping from the query to the (expansion of the) combination of views that forms a conjunctive rewriting. Paying close attention to how we construct coverings and select views can help us avoid building conjunctive rewritings which are not contained in the query. The second part of Def. 3 states that coverings should map distinguished query variables to distinguished view ones, as the containment definition demands. In our example, had one of S_1 or S_2 the first attribute (i.e., r_1 or r_3) missing from their head, they would be useless. In effect, we want the variable x of q to map onto a distinguished variable in a relevant view. Additionally to their definition coverings should adhere to one more constraint. Consider q_1 which asks for avoidable routes and all exit points from these routes to *some alternative route with less traffic*:

$q_1(x, p) \rightarrow$ AltRoutes(x,y), LessTraffic(y,x), ConnectingRoutes(x,y,p)

Now the query demands that the second argument of *ConnectingRoutes* is joined with one of x's alternative routes. This is impossible to answer, given S_1 and S_2, as S_1 does not provide x's alternative routes (i.e., r_2 in its definition). The property revealed here is that *whenever an existential variable y 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 y in the query*. This property is referred to as (clause $C2$ in) *Property 1* in MiniCon[19]. This is also the basic idea of the MiniCon algorithm: trying to map all query predicates of q_1 to all possible views, it will notice that the existential query variable y in the query maps on r_2 in S_1; since r_2 is existential it needs to go back to the query and check wether all predicates mentioning y can be covered by S_1. Here *ConnectingRoutes(x,y,p)* 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_1 had multiple occurrences of $AltRoutes$ it would try to use S_1 multiple times and fail (although as the authors of [19] say *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 avoidable routes where also an accident has recently occurred:

$S_3(r_1) \rightarrow$ AltRoutes(r_1,r_2), LessTraffic(r_2, r_1), RoutesWithAccidents(r_1)

S_3 cannot be used for q_1 as it violates MiniCon's Property1, again due to its second variable, r_2, being existential and *ConnectingRoutes* not covered. Our idea is to avoid this redundant work by compactly representing all occurrences of the same view pattern. To this end we use a graph representation of queries and views presented subsequently.

3. QUERIES AND VIEWS AS GRAPHS

Our graph representation of conjunctive queries is inspired by previous graph-based knowledge representation approaches (see conceptual graphs in [9]). Predicates and their arguments correspond 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 need to keep track of the arguments' order inside an atom. Therefore, we equip our edges with integer labels that stand for the variables' positions within the atom's parentheses. In effect,

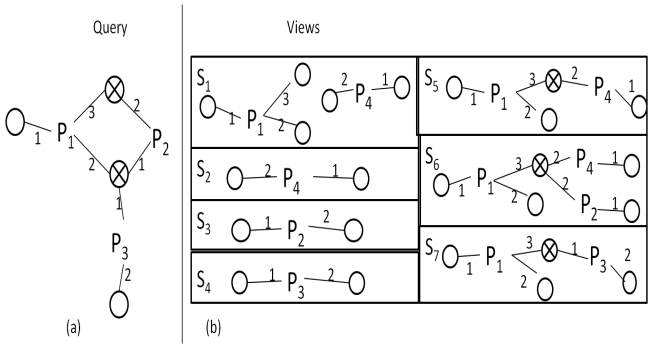

Figure 1: Query Q, and sources S_1-S_7 as a graphs.

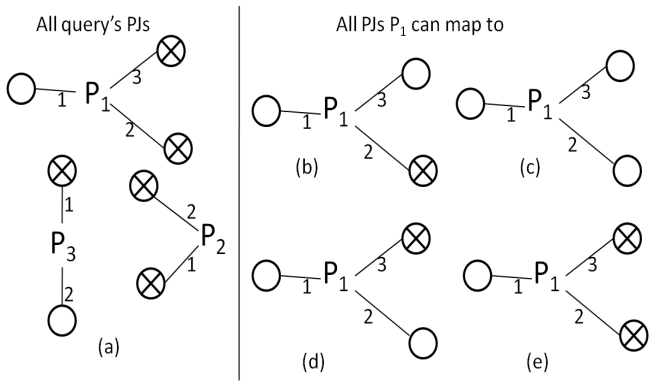

Figure 2: Predicate Join Patterns.

an edge labeled with "1" will be attached to the node representing the leftmost variable within an atom's parentheses, e.t.c. Thus we can discard variables' names; from our perspective we only need to remember a variable's type (i.e., whether the i^{th} variable of a specific atom is distinguished or not within the query or the view). This choice can be justified upon examination of Def. 3; 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:

$Q(x_1, x_2) \leftarrow P_1(x_1, y, z), P_2(y, z), P_3(y, x_2)$

corresponds to the graph seen in Fig. 1(a). Fig. 1(b) shows the graph alterego of the following 7 LAV source descriptions:

$S_1(x, y, z, g, f) \rightarrow P_1(x, y, z), P_4(g, f)$

$S_2(a, b) \rightarrow P_4(b, a)$

$S_3(c, d) \rightarrow P_2(c, d)$

$S_4(e, h) \rightarrow P_3(e, h)$

$S_5(i, k, j) \rightarrow P_1(i, k, x), P_4(j, x)$

$S_6(l, m, n, o) \rightarrow P_1(l, n, x), P_4(m, x), P_2(o, x)$

$S_7(t, w, u) \rightarrow P_1(t, u, x), P_3(x, w)$

3.1 Predicate Join Patterns

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. Fig. 2(a) shows all predicate joins that the query Q contains, (i.e., all the *query PJs*). We will refer to greater subgoals than simple PJs as *compound predicate join patterns* or CPJs (PJs are also CPJs, although atomic ones). We can now restate Def. 3 using our graph terminology: A view CPJ *covers* a query CPJ if there is a graph homomorphism h, from the query graph to the view one, such that (1) h is the identity on predicate nodes and labeled edges and (2) if a query variable node u is distinguished then $h(u)$ is also distinguished. For the query PJ for P_1 in Fig. 2(a), all PJs that can potentially cover it, appear in Fig. 2(b)-(e). Notice that under this perspective:

- Given two specific PJs A an B, we can check whether A covers B in linear time[3].
- Given a specific query PJ A, there is an exponential number of PJs that can cover it (in effect, their number is 2^d with d being the number of existential variables in A).

A critical feature that boosts our algorithm's performance is that the patterns of subgoals as graphs repeat themselves across different source descriptions. Therefore we choose to compactly represent each such different view subgoal with the same CPJ. This has a tremendous advantage (as also discussed in Sect. 6); mappings from a query PJ (or CPJ) to a view are computed just once instead of every time this subgoal is met in a source description (with the exception of repeated predicates which are addressed in Sect. 4.2.4). Nevertheless, the "join conditions", for a particular PJ within each view, are different; more "bookkeeping" is needed to capture this. In Sect. 3.2 we describe a conceptual data structure that takes care of all the "bookkeeping". At this point, we should notice that our graph constructs resemble some relevant and well study concepts from the literature, namely hypergraphs and hyperedges [1] discussed in Sect. 6.

3.2 Information Boxes

Each variable node of a PJ holds within it information about other PJs this variable (directly) connects to within a query or view. To retain this information we use a conceptual data structure called *information box* (or infobox). Each infobox is attached to a variable v. Fig. 3 shows an example infobox for a variable node. A view (or source) PJ represents a specific subgoal pattern found in multiple sources. Therefore we want to document all the joins a variable participates in, for every view. Hence, our infobox contains a list of views that this PJ appears in; for each of these views, we maintain a structure that we call *sourcebox* (also seen in Fig. 3), where we record information about the other PJs, that v is connected to. In effect we need to mark which PJ and on which edge of this PJ, v is attached to in that particular view. We call this information a *join description* of a variable within a sourcebox (inside an infobox, attached to a PJ's variable). For ease of representation we will denote each join description of a variable v in a sourcebox, with the name of the other predicate where we superscript the number of the edge v is attached to, on this other predicate (Sect. 4.2.4 clarifies this in the face of repeated predicates).

Fig. 4 shows for all predicates of Q, all the different PJs that appear in sources $S_1 - S_7$ with their infoboxes. Note that the infoboxes belonging to a query PJ contain only one sourcebox (that of the query), which in turn contains the join descriptions of the variables in the query. We omit to present the infoboxes of Q as its joins can be easily seen from Fig. 1(a).

[3]Since the edges of the two PJs are labeled, we can write them down as strings, hash and compare them (modulo the type of variable nodes).

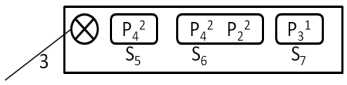

Figure 3: Infobox for a variable node. The node is existential and is attached on its predicate node on edge with label 3 (this variable is the third argument of the corresponding atom). We can find this specific PJ in three views, so there are three sourceboxes in the infobox. The two join descriptions in the sourcebox 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.

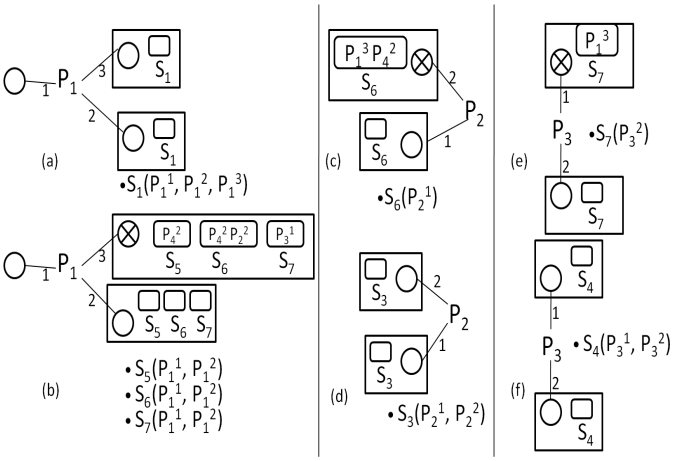

Figure 4: All existing source PJs for the predicates of the query Q. For presentation purposes the infoboxes of the first variable of P_1 are omitted. The partial conjunctive rewritings (view heads) each PJ maintains, are "descriptions" of the current PJ and are candidate parts of the final conjunctive rewritings. During our preprocessing phase, PJs for P_4 (which also exists in the sources) would also be constructed but are omitted from being presented here (as they are dropped by our later phase). Using these constructs we can compactly represent all the 8 occurrences of P_1 P_2 and P_3 in the sources $S1$-$S7$, with the 6 PJs presented here.

3.3 Partial conjunctive rewritings

Our compact representation of patterns allows us another major improvement. Previous algorithms would first finish with a view-query mapping discovery phase and then go on to gather up all "relevant" view subgoals to form a (conjunctive) rewriting. Our approach exploits the insight an algorithm gains during this phase so as to start building the correct rewriting right away. At different steps of our algorithm, each source CPJ covers a certain part of the query and within this source CPJ (and only in source CPJs) we maintain a list of conjunctions of atoms, which are candidate parts of the final conjunctive rewritings that will cover this part of the query. We call these *partial* conjunctive rewritings. These conjunctions maintain information about which view variables out of the views' head are used and which of them are equated, i.e., joined (if at all). For example, the PJ in Fig. 4(a) contains the partial rewriting $S_1(P_1{}^1, P_1{}^2, P_1{}^3)$.

4. 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 (i.e., PJ) to the source PJs and we go on by combining the relevant source PJs to form larger subgraphs (CPJs) that cover larger "underlying" query subgoals. We continue combining source CPJS and during this bottom-up combination, we also combine their partial rewritings by taking their cross-product and either merge some of their atoms or equate some of their variables. Note that we might also need to "drop" some rewritings on the way, if they are no longer feasible (see Sect. 4.2.3). We continue until we cover the entire query graph, whereby we have incrementally built the maximally-contained and complete rewriting.

4.1 Source Preprocessing

Given the set of view descriptions and considering them as graphs, we break them down to atomic PJs, by splitting each graph on the shared variable nodes. On top of the PJ generation we construct infoboxes for every variable node in those PJs. Moreover, for every PJ we generate a list of partial conjunctive rewritings, each containing the head of a different view this PJ can be found in.

Fig. 4 shows all the source PJs as constructed by the preprocessing phase. For presentation purposes, in the partial conjunctive rewritings, the view heads include only the arguments that we are actually using at this phase. Similarly to the naming scheme of the join descriptions, we use a positional scheme for uniquely naming variables in a partial conjunctive rewriting. For instance, $S_7.P_3{}^2$ is the 2nd argument of PJ P_3 in source S_7 (Sect. 4.2.4 explains how we disambiguate occurrences of the same predicate in a given view). This naming policy is beneficial as it allows us to save valuable variable substitution time.

An advantage of our approach is that our preprocessing phase does not need any information from the query, as it was designed to involve only views. Typically, a data integration system has access to all sources and view descriptions a priori (before any query). In such a case, the views can be preprocessed off-line and the PJs can be stored until a query appears. The details of this algorithm are rather obvious and omitted, but it can be easily verified that this preprocessing phase has a polynomial complexity to the number and length of the views.

Nonetheless, one can create more sophisticated indices on the sources. As seen in Fig. 2 there are 2^d potential query PJs that a source PJ can cover, with d being the number of distinguished variable nodes in the source PJ. For our implementation we chose to generate those indices; for every source PJ we construct all the (exponentially many) potential query PJs that the former could cover. Consequently given a query PJ will are able to efficiently retrieve the source PJs that cover it. The payoff for all indexing choices depends on the actual data integration application (e.g., whether the preprocessing is an off-line procedure or how large does d grow per atom etc.). Nevertheless, any offline cost is amortized over the system's live usage. Moreover, the experiments of Sect. 5 show a good performance of our algorithm even when taking the preprocessing time into account.

4.2 Query Reformulation

Our main procedure, called GQR (Graph-based Query Rewriting) and shown in Algorithm 1, retrieves all the alternative source PJs for each query PJ and stores them (as a set of CPJs) in S which is a set of sets of CPJs initially empty (lines 1-6). As we go on (line 7 and below) we remove any two CPJ sets[4] from S, combine all their elements in pairs (as Algorithm 2 shows), construct a set containing larger CPJs (which cover the union of the underlying query PJs) and put it back in S. This procedure goes on until we cover the entire query (combine all sets of CPJs) or until we fail to combine

[4]Our choice is arbitrary; nevertheless a heuristic order of combination of CPJ sets could be imposed for more efficiency.

Algorithm 1 GQR

Input: A query Q
Output: A set of rewritings for the query
1: **for all** predicate join patterns PJ_q in the query **do**
2: $SetP \leftarrow RetrievePJSet(PJ_q)$
3: **if** $SetP$ empty **then**
4: **FAIL**
5: **else**
6: add $SetP$ to S //S is the set of all CPJ sets
7: **repeat**
8: *select and remove* $A, B \in S$
9: $C \leftarrow combineSets(A, B)$
10: **if** C is empty **then**
11: **FAIL**
12: add C to S
13: **until** all elements in S are chosen
14: **return** rewritings in S

Algorithm 2 combineSets

Input: sets of CPJs A, B
Output: a set of CPJs combinations of A, B
for all pairs (a,b) $\in A \times B$ **do**
$c \leftarrow combineCPJs(a, b)$
if c is not empty **then**
add c to C
return C

two sets of CPJs which means there is no combination of views to cover the underlying query subgoals. If none of the pairs in Alg. 2 has a "legitimate" combination as explained subsequently, the sets of CPJs fail to be combined (line 11 of Alg. 1). Fig. 5(c),(g) shows the combination of PJs for P_1 with PJs for P_2, and in turn their combination with the PJ for P_3 (Fig. 5(h),(i)) to cover the whole query.

4.2.1 Join preservation

While we combine graphs we concatenate the partial conjunctive rewritings that they contain. When these rewritings contain the same view, the newly formed rewriting either uses this view twice, or only once; depending on whether the query joins are preserved across the mapping to this source.

DEF. 4. ***Join preservation:*** *For all PJ_A, PJ_B source PJs, for all Q_A, Q_B query PJs where PJ_A covers Q_A and PJ_B covers Q_B, for all views V that contain both PJ_A and PJ_B, we say that V preserves the joins of Q_A and Q_B w.r.t PJ_A and PJ_B iff for all join variables u between Q_A and Q_B:*

- *if a is the variable node u maps onto in PJ_A and b is the variable node u maps onto in PJ_B, then a and b are of the same type, and both a and b's infoboxes contain a sourcebox for V in which a has a join description for b, and b has a join description for a, and*
- *there exists a u such that a and b are existential, or (1) a and b are essentially the same variable of V (in the same position) and (2) for all variables of PJ_A, a', and of PJ_B, b', such that no join variables of the query map on a' or b', either a' and b' are don't care variables (they are distinguished (in V) but no distinguished variable of the query maps on them) or (without loss of generality): a' covers a distinguished query variable and b' is a don't care variable.*

Intuitively when a view preserves the query joins with respect to two PJs, our rewriting can use the view head only once (using both the source PJs) to cover the two underlying query subgoals; this is actually necessary when the PJs cover an existential join variable with an existential view one (as in Property 1 of MiniCon [19]). S_6 for example, preserves the query join between P_1[3],P_2[2] with respect to the PJs of Fig. 4(b) and (c). If on the other hand all view variables that cover join variables are distinguished, Def. 4 states the conditions[5] under which using the view head two times, would be correct but not minimal according to Def. 2. For example consider the following query and view:

$q(x, y, z) \rightarrow p_1(x,y,z), p_2(y,z,w)$

$\mathrm{v}(a, b, c, d) \rightarrow p_1(a,b,c), p_2(b,c,d)$

A naive covering of p_1 and p_2 would use v two times, coming up with the conjunctive rewriting:

$r(x, y, z) \rightarrow \mathrm{v}(x,y,z,f'), \mathrm{v}(f'',y,z,f''')$

If we enforce Def. 4 when combining the view patterns for $p_1(a,b,c)$ and $p_2(b,c,d)$ we'll notice that b and c do fall on the same positions in v, and while the first pattern (p_1) covers more query variables (uses a to cover x) the second one has don't cares (d in p_2 is a don't care as rewritten as f'''). In this case we can merge the two occurrences of v (using their most general unifier) and come up with a more minimal rewriting: $r(x, y, z) \rightarrow \mathrm{v}(x,y,z,f''')$.

This optimization is discussed in Sect. 4 of [16]. It is important to notice the dual function of Def. 4. On one hand, given two source PJs that cover an existential join with existential variables, a view *must* preserve the join in order to be used. If, on the other hand, all source variables that cover query joins are distinguished, the view *can* preserve the join, and be used a minimal number of times.

Algorithm 3 retrievePJSet

Input: a predicate join pattern PJ_q in the query
Output: The set of source PJs that cover PJ_q.
1: **for all** PJ_s, source PJs that covers PJ_q **do**
2: $OkToAdd \leftarrow$ **true**
3: **for all** u variable nodes in PJ_q **do**
4: $v \leftarrow$ variable of PJ_s that u maps on to
5: **if** v is existential **then**
6: **if** u is existential **then**
7: **for all** sourceboxes S in v's infobox **do**
8: **if** joins in $u \nsubseteq$ joins in S **then**
9: drop S from PJ_s
10: **if** some of PJs infoboxes became empty **then**
11: $OkToAdd \leftarrow$ **false**
12: **break**
13: **if** $OkToAdd$ **then**
14: link PJ_s to PJ_q
15: add PJ_s to C
16: Prime members of C returned in the past
17: **return** C

4.2.2 Retrieving source PJs

After the source indexing phase, our first job when a query is given to the system is to construct PJs and information boxes for all its predicates. We then retrieve, with the use of Alg. 3, 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. 3 iterates over all (existing) view PJs that cover the input. Moreover, as already discussed, if both a query variable and its mapped view variable (u and v correspondingly) are existential, we won't be able to use this view PJ if the view cannot preserve the join patterns of the query, under any circumstances. Therefore if view variable v is existential, we have to inspect every sourcebox of v's infobox

[5]These are relaxed constraints; we are investigating more cases under which a rewriting could be minimized.

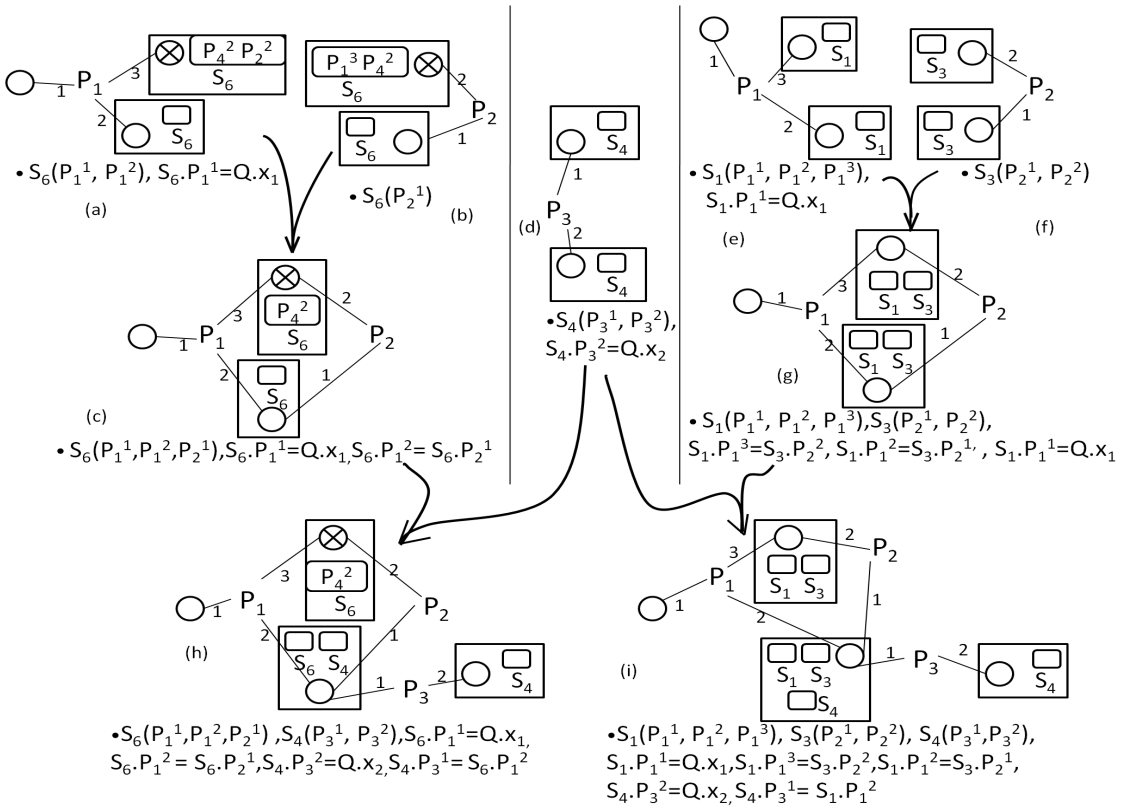

Figure 5: All source PJs that cover P_1 are combined with all PJs that cover P_2. As Alg. 2 suggests, we try all combinations of CPJs that cover the query. The figure does not show the un-combinable pairs (as e.g., the PJ in (a) with the PJ in (f)). Note that combining two nodes we eliminate the join descriptions we just satisfied. In the current example, PJ (d) is combined with both (c) and (g) which alternatively cover the same part of the query. The union of the resulting rewritings of (h) and (i) is our solution.

(lines 7-9) and verify that all join descriptions of u are included in there. If we find a sourcebox that breaks this requirement we drop this sourcebox from every infobox of this view PJ and we delete the partial conjunctive rewriting that mentions the corresponding view as well (line 9). We do all that as if this source PJ pattern never appeared in that view (for the specific subgoal of the query, the specific view subgoal that this source PJ represents is useless).

For example, when the query PJ for P_1, shown in Fig. 2(a), is given to $retrievePJSet$ the latter will consider the preconstructed PJs shown in Fig. 4(a),(b); for the one in Fig. 4(b) line 9 will drop the sourceboxes and the partial conjunctive rewritings related to S_5 and S_7 since only the infobox of S_6 is describing the query joins.

This fail-fast behavior allows us to keep only the necessary view's references in a PJ (which, in the above example is S_6). Moreover, if none of the views can cover a query subgoal, the PJ itself is ignored, leading to a huge time saving 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 example, consider the query PJ for P_3, in Fig. 2(a), which joins existentially with P_1 and P_2 on its 1st argument. In order to use the view PJ of Fig. 4(e) (which also has its first node existential) to cover it, we need to make sure that the PJ of Fig. 4(e) includes at least one sourcebox (attached to its 1st variable) which contains all the query join descriptions.

However the only sourcebox in that infobox is for S_7 and it does not describe the query join with P_2. Therefore sourcebox S_7 is dropped, and as the view PJ remains empty, it is never returned by Alg. 3. 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 our query of Fig. 1, $retrievePJset$ will be called three times (each time it will iterate over one column of Fig. 4): for P_1 it will return the PJs in Fig. 5(a) and (e), for P_2 gives Fig. 5(b) and (f) and for P_3 it returns the PJ shown in Fig. 5(d).

Note that Alg. 3 also marks (line 14) which variables correspond to returning variables of the query[6]. This is done by equating some of the partial conjunctive rewritings' variables to distinguished query variables (as seen in Fig. 5(a), (d) and (e), we include some "equating" predicates in the rewriting). Also, notice that since our example query Q does not contain P_4 none of the PJs for P_4 is retrieved. Line 16 of the algorithm is explained in Sect. 4.2.4. Notice that Alg. 3 returns a set of PJs which alternatively cover the same query PJ. Furthermore the different sets that Alg. 3 returns, cover different (and all) subgoals of the query. That makes it easier to straightforwardly combine these sets to cover larger parts of the query. We treat the element of each set as a CPJ (initially they are atomic CPJs, i.e., PJs), and we pairwise combine all the elements (CPJs) of two sets that cover two query subgoals; we construct a resulting set containing larger combined CPJs. The latter alternatively cover the larger underlying query subgoal. Next section describes how we combine two CPJs to a greater one.

4.2.3 Combination of CPJs

Algorithm 4 takes two CPJs a and b and returns their combination CPJ. If the underlying query subgoals that these CPJs cover do

[6]In an effort to be user-friendly we are maintaining the original names of the query variables.

not join with each other our algorithm just cross products the partial conjunctive rewritings these CPJs contain and returns a greater CPJ containing all PJs in both CPJs. If on the other hand underlying joins exist, procedure $lookupJoins$ in line 1 returns all pairs of variables (v_a,v_b), where v_a in a and v_b in b, that cover the join variables. For example, for the PJs of Fig. 5(a) and (b), $lookupJoins$ will return the two pairs of variables (for the two joins in the query between P_1 and P_2): ($S_6.P_1{}^3$,$S_6.P_2{}^2$) and ($S_6.P_1{}^2$,$S_6.P_2{}^1$). Next we want to enforce Def. 4 and "merge" the view heads in the partial conjunctive rewritings that preserve the join (lines 9 and 17) or equate some of them so as to satisfy the join (line 19). By *merging* two atoms of the same predicate, we keep the predicate (i.e., the view head) only once and merge their arguments. Otherwise we consider their conjunction and *equate* the variables that cover the query join variable.

Algorithm 4 combineCPJs

Input: two CPJS a,b
Output: a CPJ, combination of a, b

```
 1: for all join J in lookupJoins(a, b) do
 2:     v_a ← get from J variable in a
 3:     v_b ← get from J variable in b
 4:     if type of v_a ≠ type of v_b then
 5:         return ∅        //∅ means un-combinable
 6:     else if type of v_a = ⊗ then
 7:         for all sourceboxes S in v_a's infobox do
 8:             if S contains a join description for v_b then
 9:                 markForMerge(S, v_a, v_b)
10:             else
11:                 drop S from a and b
12:                 if v_a infobox = ∅ then
13:                     return ∅
14:     else
15:         for all pairs of sourceboxes (s_a, s_b) ∈ (infobox of v_a) ×
            (infobox of v_b) do
16:             if s_a preserves the joins of the query w.r.t. v_a and v_b
                then
17:                 markForMerge(s_a, v_a, v_b)  //preserves implies
                    that s_a = s_b
18:             else
19:                 markForEquate(s_a, v_a, s_b, v_b)
20: crw ← crossproductRewritings(a,b)
21: enforceMergeAndEquations(crw)
22: c ← mergeGraphsUpdateInfoboxes()
23: return c
```

Notice that we only can enforce joins on view variables of the same type (line 4); we either join existential variables within the same source or join/equate distinguished variables across sources. If the view variables that cover the join are existential "merging" is our only option; if v_a contains sourceboxes for different sources as v_b does, or if their common sourceboxes regard views that don't preserve the join, we drop these sourceboxes (line 11) and the corresponding partial conjunctive rewritings. If by this dropping we "empty" a PJ of source references, this is no longer useful to us and so a and b are *un-combinable* (line 13). This "pruning" is similar to the one in Alg. 3 and can happen often.

On the other hand, if the view variables that cover the query join are distinguished, we either merge the partial conjunctive rewritings on the corresponding view head (in case that the view satisfies Def. 4) or we equate v_a and v_b in the two view heads (line 19). Finally, we consider the cross product of the remaining partial conjunctive rewritings creating larger ones, and we iterate over them to enforce all merging and equating we just noted down. For example, in Fig. 5 when our algorithm examines the existential join between the PJs shown in (a) and (b), it marks S_6 for merging since S_6 preserves the join. At another iteration of line 1, regarding the same PJs but for the distinguished join this time, we need to equate

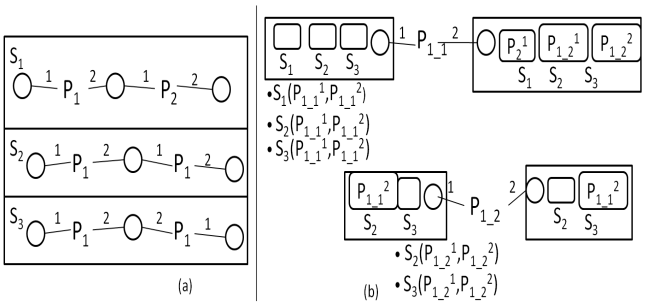

Figure 6: Repeated Predicates. In (b) only the PJs for P_1 are shown; these "capture" all five occurrences of P_1 in the sources S_1,S_2 and S_3 of (a).

two variables of S_6 (namely, $P_1{}^2$ with $P_2{}^1$). At the end both these decisions are enforced as seen in the partial conjunctive rewriting of Fig. 5(c). As we go on, algorithm 1 combines all elements of S and produces C, the set containing the CPJs seen in 5(h) and (i). Their conjunctive rewritings are the conjunctive rewritings of the solution and their union is the maximally-contained and complete rewriting:

$$Q(x_1, x_2) \leftarrow S_6(x_1, \_, y, y), S_4(y, x_2)$$

$$Q(x_1, x_2) \leftarrow S_1(x_1, y, z, \_, \_), S_3(y, z), S_4(y, x_2)$$

4.2.4 Repeated Predicates

In general repeated predicates in the view descriptions should be treated separately; we create multiple PJs per predicate per source and name these PJs differently. We then can create and use infoboxes as described so far. For a query containing the corresponding predicate the algorithm would try to use all these alternative PJs and combine them with PJs that cover other parts of the query so as to capture the multiplicity of the predicate in the sources. An important notice is that we only need to maintain different PJs for predicates within the same source, but not across sources; hence the maximum number of different source PJs for the same predicate is the maximum number a predicate repeats itself within any source. Fig. 6(b) shows how we can use two different PJs to hold the information needed for all the five occurrences of P_1 in Fig. 6(a).

In the face of multiple occurrences of the same predicate in the query, it is suitable to imagine all PJs discussed so far as classes of PJs: we instantiate the set of PJs that cover a specific predicate as many times as the predicate appears in the query. 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 (and as our argument names are positional, "priming" the sources allows us to differentiate among two instances of the same variable in the same source). Line 16 of algorithm 3 does exactly that. Having said the above, Fig. 7 shows all the PJs for the sources of Fig. 6 created for query Q of Fig. 7. Rewritings for this query will come out of the 4 possible combinations of these 4 instantiations of the two PJs (of Fig. 6).

4.2.5 GQR Correctness

Below we give a sketch proof that our algorithm is correct; for soundness we show that any conjunctive rewriting in our output is contained in the query, and for completeness that for any possible conjunctive rewriting of the query, we always produce a conjunctive rewriting which contains it.

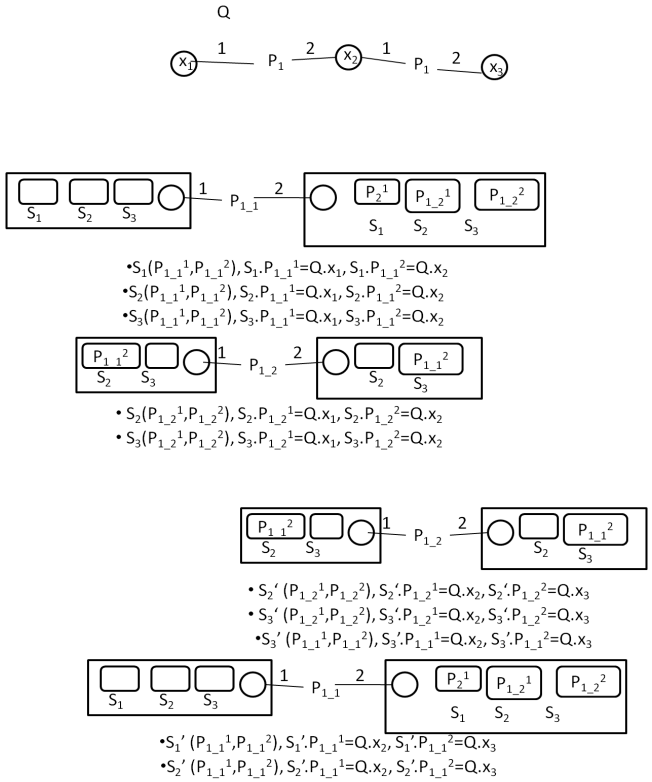

Figure 7: Repeated predicates in the query (returning variable names are shown in Q's graph for convenience). For the sources of Fig. 6(a), the algorithm "instantiates" the matching PJs (Fig. 6(b)) for every occurrence of the same query predicate.

THEOREM 1. *Given a conjunctive query Q and conjunctive views $V_1, ..., V_n$, the GQR algorithm produces a UCQ that is a maximally-contained and complete rewriting of Q using $V_1, ..., V_n$.*

PROOF. *Soundness.* Consider an output conjunctive rewriting r of our algorithm. We need to show that $exp(r) = r' \subseteq Q$. This means that there is a containment mapping from Q to r'. Keep in mind that the atoms in r' however exist in different views (not in the same as the containment mapping definition demands). It is not difficult to see that our approach "constructs" the containment mapping through coverings; each covering is a mapping from a subpart of the query to a part of r'. Considering multiple such coverings will give us our containment mapping from Q to our combination of views (that r contains).

Completeness. Consider a conjunctive rewriting p element of a problem's maximally-contained and complete rewriting. There is a containment mapping from Q to $exp(p)$. This means that depicted as graphs, there is a graph homomorphism h_1 from the query to some of the PJs for the predicates that constitute $exp(p)$. Breaking up the homomorphism per individual target PJ means that there are coverings from parts of the query to these PJs (these coverings do not get affected if we compactly represent these PJs, gathering them up in patterns as in section 4.1). It is not hard to verify that if such coverings exist our algorithm will consider them when looking at different view PJs; hence it will produce a rewriting r for the crossproduct of the views these PJs belong in, for which it will hold that there exists a containment mapping (and a graph homomorphism) $h_2 : vars(Q) \rightarrow vars(exp(r))$. Moreover for all variables q_v of Q which map on distinguished variables d in those view PJs of r (i.e., $h_2(q_v) = d$), it holds also that h_1 maps q_v on the same node d in the PJs of $exp(p)$. Hence whenever r (which is written over view head predicates) has a distinguished variable in some view in its body, p has the same variable on the same position on the same view (modulo renaming). Hence, there is a containment mapping from r to p, which means that $p \subseteq r$. □

5. EXPERIMENTAL EVALUATION

For evaluating our approach we compared with the most efficient (to the best of our knowledge) state-of-the-art algortihm, MCDSAT [7] (which outperformed MiniCon [19]). We show our performance in two kinds of queries/views: star and chain queries. In all cases GQR outperforms MCDSAT even by close to two orders of magnitude. For producing our queries and views we used a random query generator[7].

5.1 Star queries

We generated 100 star queries and a dataset of 140 views for each query. We created a space with 8 predicate names out of which each query or view chooses randomly 5 to populate its body and it can choose the same one up to 5 times (for instantiating repeated predicates). Each atom has 4 randomly generated variables and each query and view have 4 distinguished variables. We measured the performance of each of the queries scaling from 0 to 140 views. We run our experiments on a cluster of 2GHz processors each with 1Gb of memory; each processor was allocated all the 140 runs for one query, and we enforced 24 hours wall time for that job to be finished. Fig. 8(a) shows the average query reformulation time for 99 queries which met the time and memory bounds. On this set of experiments we perform 32 times faster than MCDSAT. Fig. 8(a) also shows how the average number of conjunctive rewritings grows with respect to the number of views.

5.2 Chain queries

For the chain queries we generated again 100 queries and a dataset of 140 views for each query. Our generator could now choose 8 body predicates (on a chain), for any rule, out of a pool of 20 predicates of length 4. Up to 5 predicates in each rule can be the same. Input queries have 10 distinguished variables. With this experiment, we would like to avoid measuring exponential response times simply because the size of the rewriting grows exponentially. Hence, trying to create a "phase transition point", we generated the first 80 views for all our view sets containing 10 distinguished variables and each additional view (up to 140) with only 3 distinguished variables. This causes the number of conjunctive rewritings to grow exponentially up to 80 views, but this rate becomes much slower from there and after. This trend in the number of rewritings can be seen in Fig. 8(b).

As seen in Fig. 8(b), GQR runs 75 times faster than MCDSAT and it is much more scalable, as it "fails" very fast (7 chain queries did not meet our experimental bounds, either for GQR or MCDSAT). This can be seen when there are no rewritings at all, as well as after the point of 80 views; the time of reformulation clearly depends more on the size of the output than that of the problem. As also seen in Fig. 8(b), GQR produces fewer conjunctive rewritings than MCDSAT. An example query where MCDSAT produced a redundant rewriting was the following. Given the query and view:

$q(x_1, x_2, x_3) \rightarrow p_1(x_0, x_1, x_2, x_3), p_2(x_1)$

$\mathbf{v}(y_1, y_2, y_3, y_4, y_5, y_6, y_7) \rightarrow p_1(y_2, y_3, y_9, y_{10}), p_1(y_4, y_5, y_6, y_7), p_2(y_1)$

MCDSAT produces both rewritings below, while GQR produces only the second (which contains the first):

[7]The generator was kindly provided to us by R. Pottinger, and it is the same one that she used for the original experimental evaluation of MiniCon.

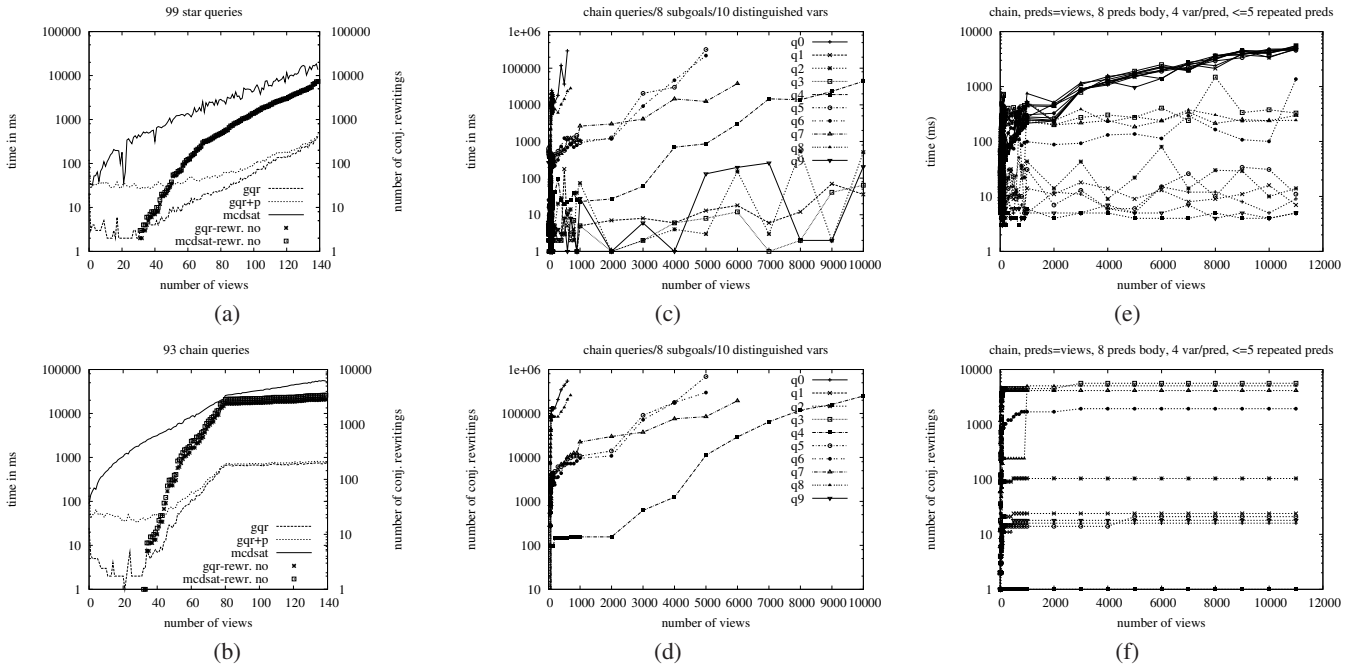

Figure 8: (a) Average time and size of rewritings for star queries. GQR time does not take into account the source preprocessing time while gqr+p does. As we can see the preprocessing phase (for this small set of views) does not add much to our algorithm; the query reformulation phase is dominating the time as the number of views increases. (b) Average time and size of rewritings for chain queries. After the point of 80 views, while the problem size continues to grow linearly, the output (number of rewritings) grows very slowly. (c) Average reformulation time for 10 chain queries. Preprocessing time is not included in the plot. (d) Average size of rewriting for chain queries of (c). Queries $q1$,$q2$,$q3$ and $q9$ don't produce any rewritings. (e) Ten chain queries on views constructed from an increasing predicate space. The upper bunch of straight lines gives the total time for the queries, while the lower dotted part gives only the reformulation time. (f) Number of rewritings for (e).

$q(x_1, x_2, x_3) \rightarrow \text{v}(x_1, f_1, f_2, x_0, x_1, x_2, x_3)$

$q(x_1, x_2, x_3) \rightarrow \text{v}(f_0, f_1, f_2, x_0, x_1, x_2, x_3),$
$\text{v}(x_1, f_8, f_9, f_{10}, f_{11}, f_{12}, f_{13})$

5.3 Chain queries using 10000 views

In Fig. 8(c) we expose the times for 10 chain queries with the very same experimental setting of the previous subsection, using 10000 views. As previously the first 80 views for each query have 10 distinguished variables, and the rest only 3. However, as the figure shows, our "transition phase point" did not work well for the queries that had already produced some rewritings within the first 80 views (namely, $q0$,$q4$,$q5$,$q6$,$q7$ and $q8$). The number of conjunctive rewritings for these queries grows exponentially to the number of views. On the other hand 4 queries did not produce rewritings up to 80 views; and they also did not produce any from that point and on (as the number of distinguished variables for all views after 80 is too constraining to, e.g, cover an atom that has 4 distinguished variables). Nevertheless this figure serves our point. The unfinished queries have crashed on a number of views that caused too many conjunctive rewritings, as Fig. 8(d) shows. In this setting, our algorithm runs in less than a second for queries that don't produce rewritings for 10000 views, while it produces 250.000 conjunctive rewritings for 10000 views in 44 seconds (for $q4$).

Lastly, in an effort to keep the number of conjunctive rewritings low, we decided to set the predicate space (out of which we populate our rules) to equal the number of views at each point. Fig. 8(e), shows how ten chain queries performed with and without the preprocessing times. As seen from the graph the exponential burden of solving the problem lies on the preprocessing phase, while the online reformulation time is less than second for all queries, even the ones producing thousands of rewritings (Fig. 8(f) shows the number of rewritings for these queries).

6. COMPARISON WITH RELATED WORK

Early approaches dealing with query rewriting involve algorithms as the *bucket* [17] and the *inverse rules* [10]. A more efficient approach was proposed in 2001, by the MiniCon [19] algorithm. Similar to our notion of CPJs, MiniCon devises MCDs (MiniCon Descriptions), which are coverings as defined in Def. 3 (they are defined per distinct query and view subgoals) having the additional property that they always consider as a whole the existentially chained pieces of the query whenever one existential query variable is mapped to an existential one on the view (Property 1 as discussed in Sect. 2). Minimal MCDs are MCDs which don't contain anything more than these existential chained pieces. In the following, we briefly describe the two phases of the MiniCon algorithm and give our algorithm's advantages against each one of its steps. It is important to notice that MCDSAT which exhibited a better performance than MiniCon [7], is essentially the MiniCon algorithm casted as a satisfiability problem, and comparing our algorithm against MiniCon, reveals our advantages against the foundations of MCDSAT also.

6.1 MiniCon Phase One

Before considering a covering of an atomic query subgoal g_q with a view V, MiniCon uses a head homomorphism h:$vars(\{head(V)\}) \rightarrow vars(\{head(V)\})$ to possibly equate some variables in the head of V (for all variables x, if x is existential $h(x) = x$ and if it is distinguished, then $h(x)$ is distinguished and $h(h(x)) = h(x)$). It then can look for a homomorphism φ so as to cover g_q with an atomic subgoal $h(g_v) \in h(V)$. Note in that the original MiniCon

algorithm the authors suggest a search over the entire space of all *least-restrictive* head homomorphisms h, and mappings φ so that $\varphi(g_q) = h(g_v) \in h(V)$ ($STEP1$).

Subsequently, for each g_q,g_v and pair of functions h,φ that come out of $STEP1$, the algorithm produces a minimal MCD ($STEP2$). Formally, an MCD M is a tuple $< V, \varphi, h, G_q >$ where (a) $h(V)$ covers $G_q \subseteq body(Q)$ (containing g_q) with φ and (b) for all x, if $\varphi(x)$ is an existential variable, then all atomic subgoals $g_i \in body(Q)$ that mention x are in G_q. An MCD M is minimal if (c) it covers[8] g_v together with the minimum additional subset of $body(Q)$ so as to satisfy (a) and (b). A minimal MCD intuitively covers only one *connected* subgraph of the query graph which adheres to property (b) above (clause $C2$ of Property 1 in [19]).

6.2 MiniCon Phase Two

In its second phase the MiniCon algorithm, needs to choose sets of MCDs which cover *mutually exclusive* parts of the query, and their union covers the entire query ($STEP3$). It then follows a lightweight generation of a conjunctive rewriting for each MCD combination ($STEP4$). For each conjunctive rewriting that it produces, the analogous of our Def. 4 is employed in order to "minimize" some of the rewritings ($STEP5$).

6.3 MCDSAT

In MCDSAT, the authors cast MiniCon's first phase (the MCDs generation problem) into a propositional theory whose models constitute the MCDs. This theory is then compiled into a normal form called d-DNNF that implements model enumeration in polynomial time in the size of the compiled theory. $STEP1$, and properties (a), (b) and (c) of $STEP2$ result in clauses of that theory, which is further extended with more optimization clauses. The result is an efficient algorithm for generating MCDs. Nevertheless, as we discuss below we believe that we have a better encoding of the problem than MiniCon's steps 1 and 2, that MCDSAT also employs.

For the second phase, that of MCD combination and rewriting generation, MCDSAT considers either a traditional implementation of MiniCon's second phase or yet another satisfiability approach: some additional clauses are devised in order to present an extended logical theory whose models are in correspondence with the rewritings. Note, that although the authors present an experimental performance of the compilation times of these extended theories, for our own evaluation we consider (and exhibit much better performance on) the times to get the actual rewritings themselves. Nonetheless, a contribution of these compiled extended theories is that they serve as compact repositories of rewritings; once compiled one can get the rewritings in polynomial time to the size of the compiled theory.

6.4 GQR vs MiniCon and MCDSAT

Our encoding of the problem exhibits several advantages against steps $STEP1$-$STEP5$. Firstly, we don't need to explicitly deal with finding h in $STEP1$, since we abstract from variable names. Returning variables are equated implicitly as we build up our CPJs. Moreover, MiniCon iterates over every input subgoals g_q and g_v. In combination with $STEP2$(b) this implies that some query subgoals will be considered more than once. In effect, while in $STEP2$ the query subgoals g_i are included in an MCD for input g_q (that came out of $STEP1$), subsequent iterations will result them forming the same MCD (among others) when they are themselves considered as the input subgoals of $STEP1$[9]. GQR on the other hand considers each query subgoal exactly once and thus avoids these redundant mappings.

A major advantage of our encoding is that while $STEP2$ is considered for every distinct atomic view subgoal of every view, we consider PJs; being compact representations of view patterns they are dramatically fewer than the distinct view subgoals. Nevertheless this does not mean "more work" for a subsequent phase. On the contrary, during our iteration of this less number of constructs we avoid the heavy computation of $STEP2$. We essentially avoid "tracking" each individual existentially chained piece in a view, for every query subgoal we map onto a part of it (as $STEP2$(b) suggests). Same piece patterns are consider just once for all sources, at the time we combine their individual parts (PJs and CPJs).

This design also benefits our performance in the face of repeated predicates in a view. Consider the view $V_1(x, g, f) \rightarrow P_1(x, y)$, $P_2(y, z)$, $P_3(z, g)$, $P_3(z, f)$ and the query $Q(a, b) \rightarrow P_1(a, y)$, $P_2(y, z)$, $P_3(z, b)$. Here MiniCon will most probably (to the best of our knowledge) create two MCDs for P_3, each time recomputing the mapping of $P_1(a, y)$, $P_2(y, z)$ to the view. On the other hand, we will consider this mapping just once. Moreover as our previous argument exposed, this join pattern of P_1 and P_2 will be at the same time detected in every source it appears in (at the same step we also rule out of future consideration all the non combinable sources that contain these patterns).

We would like to point out that we do consider the whole set of mappings from every atomic query subgoal to all the atomic view subgoals per view, in a way very similar to $STEP1$. We do this however in our preprocessing phase where we additionally compute the PJ infoboxes. As a result we are left with both drastically less constructs to deal with and in a more straightforward and less costly manner than $STEP2$.

Moreover our algorithm exhibits additional advantages in that its second phase (the CPJ combination) already encapsulates the entire second phase of MiniCon and MCDSAT. And it does so even more efficiently; the $STEP3$ of the related algorithms needs to explicitly choose "mutually exclusive" MCDs in order to provide non redundant rewritings; in our case all sets of CPJs (as seen in Sect. 4) are mutually exclusive. Yet, we do need to combine all their elements pairwise, however we can fail-fast in the case one such pair is un-combinable as the reader can also see from Sect. 4. The formulation of the actual rewritings ($STEP4$ for MiniCon) is embodied in our second phase and is done incrementally (through the partial rewritings) during the combination of the CPJs to larger graphs. $STEP5$ is also smoothly incorporated with this incremental building of a rewriting through the implementation of Def. 4. In conclusion, PJs (and CPJs) seem to be better constructs than MCDs in encoding the problem. This is also why we believe our solution performs better than the other algorithms.

6.5 Hypergraphs and Hypertree Decompositions

As already mentioned, our graphs and PJs resemble some relevant and well study concepts from the literature, namely hypergraphs and hyperedges [1] accordingly. Nevertheless we find our own notation more suitable for focusing on the type of the query's variables as well as making their join descriptions explicit. Since hypergraphs were developed for representing single queries, it is less convenient (although entirely symmetrical to our approach) to attach infoboxes on them, or have hyperedges represent information across multiple views. In [12], hypertree decompositions were

[8] Since MCDs are essentially descriptions of coverings, we'll abuse terminology to say that an MCD covers a subgoal.

[9] Although the algorithm suggests such a brute force approach, it is indeed possible that the original MiniCon implementation did prune out some redundant MCDs. The MCDSAT satisfiability perspective most probably also avoids this redundancy.

devised; these were hierarchical join patterns which combined hyperedges bottom-up into ever larger query fragments. Nevertheless the focus of this representation is different; each vertex of these constructs can represent a whole a set of atoms and/or variables and each variable and atom induces a connected subtree. On the other hand, looking at our own incremental build-up of the view CPJs as a tree of combinations, we differentiate; we keep some compact "hyperedges" per vertex and each parent vertex contains all and exactly its children. Again it possible that our CPJ combination could be translated to a variation of hypertree decompositions and we plan to further investigate this perspective in the future.

7. DISCUSSION AND FUTURE WORK

We presented GQR, a scalable query rewriting algorithm that computes the rewritings in an incremental, bottom up fashion. Using a graph perspective of queries and views, it finds and indexes common patterns in the views making rewriting more efficient. Optionally, this view preprocessing/indexing can be done offline thereby speeding up online performance even more. The bottom-up rewriting process also has a fail-fast behavior. In our experiments, GQR is about 2 orders of magnitude faster than the state of the art and scales up to 10000 views.

GQR opens several areas of future work. First, the current algorithm picks an arbitrary order for the bottom up combination of CPJs. However, we plan to investage heuristic orderings that could speed the process even further, by failing faster and/or computing a smaller number of intermediate partial rewritings. Second, we will like to explore the phase transitions in the space of rewritings. Third, we plan to investigate the nature of minimal rewritings and encode additional minimizations. Fourth, we plan to extend the algorithm with constants and interpreted predicates, such as comparisons. Fifth, we will like to use the insights of our graph-based representation to define a novel SAT encoding that may lead to an even faster algortihm, given the significant optimization effort invested in state-of-the-art SAT solvers. Finally, we want to extend our algorithm to richer interschema axioms such as GLAV (source to target rules that have conjunctive antecedents and consequents) and to rewrite in the presence of constraints, particularly from more expressive languages such as Description Logics.

8. ACKNOWLEDGMENTS

We thank Rachel Pottinger and Maria Esther Vidal for graciously sharing their code. 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).

9. REFERENCES

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

[2] F. Afrati and N. Kiourtis. Query answering using views in the presence of dependencies. *New Trends in Information Integration (NTII)*, pages 8—-11, 2008.

[3] P. Agrawal, A. D. Sarma, J. Ullman, and J. Widom. Foundations of Uncertain-Data Integration. *Proceedings of the VLDB Endowment*, 3(1):1–24, 2010.

[4] M. Arenas, J. Pérez, J. Reutter, and C. Riveros. Composition and inversion of schema mappings. *ACM SIGMOD Record*, 38(3):17, Dec. 2010.

[5] M. Arenas, J. Pérez, J. L. Reutter, and C. Riveros. Foundations of schema mapping management. In *PODS '10: Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems of data*, pages 227–238, New York, NY, USA, 2010. ACM.

[6] P. C. Arocena, A. Fuxman, and R. J. Miller. Composing local-as-view mappings: closure and applications. In *Proceedings of the 13th International Conference on Database Theory*, pages 209–218. ACM, 2010.

[7] Y. Arvelo, B. Bonet, and M. E. Vidal. Compilation of query-rewriting problems into tractable fragments of propositional logic. In *AAAI'06: Proceedings of the 21st national conference on Artificial intelligence*, pages 225–230. AAAI Press, 2006.

[8] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM, 1977.

[9] M. Chein and M.-L. Mugnier. *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs*. Springer Publishing Company, Incorporated, 2008.

[10] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 109–116, New York, NY, USA, 1997. ACM.

[11] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Integrating and accessing heterogeneous information sources in tsimmis. In *Proceedings of the AAAI Symposium on Information Gathering, pp. 61-64*, March 1995.

[12] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 21–32. ACM, 1999.

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

[14] H. Kondylakis and D. Plexousakis. Enabling ontology evolution in data integration. In *Proceedings of the 2010 EDBT Workshops*, pages 1–7. ACM, 2010.

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

[16] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 95–104. ACM, 1995.

[17] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. pages 251–262, 1996.

[18] V. Lín, V. Vassalos, and P. Malakasiotis. MiniCount: Efficient Rewriting of COUNT-Queries Using Views. *22nd International Conference on Data Engineering (ICDE'06)*, pages 1–1, 2006.

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

[20] L. Zhou, H. Chen, Y. Zhang, and C. Zhou. A Semantic Mapping System for Bridging the Gap between Relational Database and Semantic Web. *American Association for Artificial InteIligence (www. aaai. org)*, 2008.