Algebraic Rewritings for Optimizing Regular Path Queries

Gösta Grahne and Alex Thomo
Concordia University
{grahne, thomo}@cs.concordia.ca

Abstract

Rewriting queries using views is a powerful technique that has applications in query optimization, data integration, data warehousing etc. Query rewriting in relational databases is by now rather well investigated. However, in the framework of semistructured data the problem of rewriting has received much less attention. In this paper we focus on extracting as much information as possible from algebraic rewritings for the purpose of optimizing regular path queries. The cases when we can find a complete exact rewriting of a query using a set a views are very "ideal." However, there is always information available in the views, even if this information is only partial. We introduce "lower" and "possibility" partial rewritings and provide algorithms for computing them. These rewritings are algebraic in their nature, i.e. we use only the algebraic view definitions for computing the rewritings. This fact makes them a main memory product which can be used for reducing secondary memory and remote access. We give two algorithms for utilizing the partial lower and partial possibility rewritings in the context of query optimization.

1 Introduction

Semistructured data is a self-describing collection, whose structure can naturally model irregularities that cannot be captured by relational or object-oriented data models [ABS99]. This kind of data is usually best formalized in terms of labelled graphs, where the graphs represent data found in many useful applications such web information systems, XML data repositories, digital libraries, communication networks, and so on. Almost all the query languages for semi-structured data provide the possibility for the user to query the database through regular expressions. The design of query languages using regular path expressions is based on the observation that many of the recursive queries that arise in practice amount to graph traversals. These queries are in essence graph patterns and the answers to the query are subgraphs of the database that match the given pattern [MW95, FLS98, CGLV99, CGLV2000].

For example, for answering a query containing in it the regular expression $(\_^* \cdot \mathit{article}) \cdot (\_^* \cdot \mathit{ref} \cdot \_^* \cdot (\mathit{ullman} + \mathit{widom}))$ one should find all the paths having at some point an edge labelled *article*, followed by any number of other edges then by an edge *ref* and finally by an edge labelled with *ullman* or *widom*.

Based on practical observations, the most expensive part of answering queries on semistructured data is finding these graph patterns described by regular expressions. This is, because a regular expression can describe arbitrary long paths in the database which means in turn an arbitrary number of physical accesses. Hence it is clear that, having a good optimizer for answering regular path (sub)queries is very important. This optimizer can be used for the broader class of full fledged query languages for semistructured data.

In semistructured data, as well as in other data models such as relational and object oriented, the importance of utilizing views is well recognized. [LMSS95, CGLV99, Lev99]. Simply stated, the problem is: Given a query Q and a set of views $\{V_1, \ldots, V_n\}$, find a representation of Q by means of the views and then answer the query on the basis of this representation. Several papers investigate this problem for the case of conjunctive queries [LMSS95, Ull97, CSS99, PV99]. Their methods are based on the query containment and the fact that the number of literals in the minimal rewriting is bounded from above by the number of literals in the query.

It is obvious that a method for rewriting of regular path queries requires a technique for rewriting of regular expressions, i.e. given a regular expression E and a set of regular expressions $E_1, E_2, ..., E_n$ one wants to compute a function $f(E_1, E_2, ..., E_n)$ which approximates E. As far as the authors know, there are two methods for computing such a function f which approximates E from below. The first one of Conway [Con71] is based on the derivatives of regular expressions introduced by Brzozowski [Brzo64], which provide the ground for the development of an algebraic theory of factorization in the regular algebra [BL80] which in turn gives the tools for computing the approximating function. The second method by Calvanese et al [CGLV99] is automata based. Both methods are equivalent in the sense that they compute the same rewriting of a query. However, these methods model –using views– only full paths of the database, i.e. paths whose labels spell a word belonging to the regular language of the query. But in practice, the cases in which we can infer from the views full paths for the query are very "ideal".The views can cover partial paths which can be satisfactory long for using them in optimization but if they are not complete paths, they are ignored by the above mentioned methods. So, it would probably be better to give a partial rewriting in order to encapture all the information provided by the views. The information provided by the views is always useful, even if it is partial and not complete. The problem of a partial rewriting is touched upon briefly in [CGLV99]. However, there this problem is considered only as an extension of the complete rewriting, enriching the set of the views with new elementary one-symbol views, and materializing them before query evaluation. The choice of the new elementary views to be materialized is done in a brute force way, using some cost criteria depending on the application.

In this paper we use a very different approach. For each word in the regular language of the query we do the best possible using views. If the word contains a sub-path that a view has traversed before, we use that view for evaluation. We present generalized query answering algorithms that access the database only when necessary. For the "been there" subpaths our algorithms use the views. Note that we do not materialize any new views, we only consult the database "on the fly," as needed.

The outline of the paper is as follows. In Section 2 we formalize the problem of query rewriting using views in the realistic framework of cached views and available database. Then we discuss the utility of algebraic rewritings. We show through a realistic example that the complete rewritings can be empty for a particular query, while the partial information provided by the views is no less than 99% of the complete "missing" information. In Section 3 we introduce and formally define a new algebraic, formal-language operator, the *exhaustive replacement*. Simply described, given two languages L_1 and L_2, the result of the exhaustive replacement operation is the replacement, by a special symbol, of all the words of L_2 that occur as sub-words of words in L_1. Moreover, between any two occurrences of words of L_2 as sub-words in a word from L_1, no word from L_2 appears as a subword. In Section 4 we give a theorem showing that the result of the exhaustive replacement can be represented as an intersection of a rational transduction and a regular language. The proof of the theorem is constructive and provides an algorithm for computing the exhaustive replacement operator. Then in Section 5 we present the partial possibility rewriting that is a generalization of the previously introduced exhaustive replacement operator. In Section 6 we define a partial lower rewriting. It is the largest subset of words in the partial possibility rewriting such that their expansions to the the database alphabet are contained in the query language. In Section 7 we review a typical query answering algorithm for regular path queries and show how two modify it into two other "lazy" algorithms for utilizing the partial lower and possibility rewritings respectively. The computational complexity is studied in Section 8. We show that the algorithms proposed for computing the partial possibility and partial lower rewritings are essentially optimal.

2 Background

Rewriting regular queries. Let Δ be a finite alphabet, called the *database alphabet*. Elements of Δ will be denoted $R, S, T, R', S', \ldots, R_1, S_1, \ldots$, etc. Let $\mathbf{V} = \{V_1, \ldots, V_n\}$ be a set of *view definitions*, with each V_i being a finite or infinite regular language over Δ. We call the set $\Omega = \{v_1, \ldots, v_n\}$ the *outer alphabet*, or *view alphabet*. For each $v_i \in \Omega$, we set $def(v_i) = V_i$. The substitution def associates with each "view name" v_i in the view alphabet the language V_i. The substitution def is applied to words, languages, and regular expressions in the usual way (see e. g. [HU79]).

A *database query* Q is a finite or infinite regular language over Δ. Sometimes we need to refer to a regular expression representing a language Q. We then write $re(Q)$ to denote this expression.

A *maximal lower rewriting* (l-rewriting) of a user query Q using $\mathbf{V}$ is a language Q' over Ω, that includes *all* the words $v_{i_1} \ldots v_{i_k} \in \Omega$, such that

$$def(v_{i_1} \ldots v_{i_k}) \subseteq Q.$$

A *maximal possibility rewriting* (p-rewriting) of a user query Q using $\mathbf{V}$ is a language Q'' over Ω, that includes *all* the words $v_{i_1} \ldots v_{i_k} \in \Omega$, such that

$$def(v_{i_1} \ldots v_{i_k}) \cap Q \neq \emptyset.$$

For instance, if $re(Q)$ is $(RS)^*$, and we have the views V_1, V_2, V_3 and V_4 available, with $re(V_1) = R + SS$, $re(V_2) = S$, $re(V_3) = SR$ and $re(V_4) = (RS)^2$ respectively, the l-rewriting is v_4^* and the p-rewriting is $(v_4 + v_1 v_3^* v_2)^*$.

Semistructured databases. We consider a database to be an edge labeled graph. This graph model is typical in semistructured data, where the nodes of the database graph represent the objects and the edges represent the attributes of the objects, or relationships between the objects.

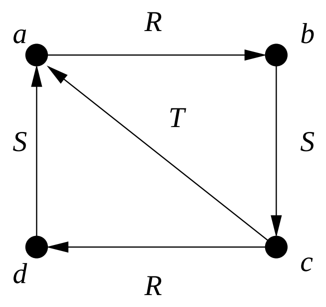

Figure 1: An example of a graph database

Formally, we assume that we have a universe of objects D. Objects will be denoted $a, b, c, a', b', \ldots, a_1, b_2, \ldots$, and so on. A *database* DB over (D, Δ) is a pair (N, E), where $N \subseteq D$ is a set of nodes and $E \subseteq N \times \Delta \times N$ is a set of directed edges labeled with symbols from Δ. Figure 1 contains an example of a graph database.

If there is a path labeled $R_1, R_2, \ldots, R_k$ from a node a to a node b we write $a \stackrel{R_1.R_2\ldots R_k}{\longrightarrow} b$. Let Q be a query and $DB = (N, E)$ a database. Then the *answer to* Q on DB is defined as

$$ans(Q, DB) = \{(a, b) \in N^2 \ : \ a \stackrel{W}{\longrightarrow} b \text{ for some } W \in Q\}.$$

For instance, if DB is the graph in Figure 1, and $Q = \{SR, T\}$, then $ans(Q, DB) = \{(b, d), (d, b), (c, a)\}$

What are rewritings good for? In a scenario with a database and materialized views there are various assumptions, such as the exactness/soundness/completeness of the views, and whether the database relations are available, and if so, at what cost compared to the cost of accessing the views. Depending on the application (information integration, cache-based optimization, etc) different assumptions are valid. The use of rewritings in answering user queries using views have been thoroughly investigated in the case of relational databases (see e.g. the survey [Lev99]). For the case of semi-structured databases much less is currently known. Notably, Calvanese et al [CGLV99] show how to obtain l-rewritings, and the same authors, in [CGLV2000] discuss the possible use of l-rewritings in information integration applications. The present authors show in [GT2000] how p-rewritings are obtained and how they are profitable in information integration applications, where the database graph is unavailable. The paper [GT2000] shows that running an l-rewriting on the view graph is guaranteed to produce a subset of the desired answer, while running the p-rewriting is guaranteed to produce a superset.

In particular, the l-rewriting can be empty, even if the desired answer is not. Suppose for example that query Q is $re(Q) = R_1 \ldots R_{100}$ and we have available two views V_1 and V_2, where $re(V_1) = R_1 \ldots R_{49}$ and $re(V_2) = R_{51} \ldots R_{100}$. It is easy to see that the l-rewriting is empty. However, depending on the application, a "partial rewriting" such as $v_1 R_{50} v_2$ could be useful. In the next section we develop a formal algebraic framework for the partial rewritings. This framework is flexible enough and can be easily tailored to the specific needs of the various applications. In Section 4 we demonstrate the usability of the partial rewritings in query optimization.

3 Replacement – A New Algebraic Operator

In this section we introduce and study a new algebraic operation, the *Exhaustive Replacement* in words and languages.

Let W be a word, and M a λ-free language over some alphabet, and let $\dagger$ be a symbol outside that alphabet. Then we define

$$\rho_M(W) = \begin{cases} \{W_1 \dagger W_3 : \text{ there is a } W_2 \in M \text{ such that } W = W_1 W_2 W_3\} & \text{if non-empty} \\ \{W\} & \text{otherwise.} \end{cases}$$

Furthermore, let L be a set of words over the same alphabet as M. Then define $\rho_M(L) = \bigcup_{W \in L} \rho_M(W)$. We can now define the *powers of* ρ_M as follows:

$$\rho_M^1(\{W\}) = \rho_M(W), \quad \rho_M^{i+1}(\{W\}) = \rho_M(\rho_M^i(\{W\})).$$

Let k be the smallest integer such that $\rho_M^{k+1}(\{W\}) = \rho_M^k(\{W\})$. We then set

$$\rho_M^*(W) = \rho_M^k(\{W\}).$$

(It is clear that k is at most the number of symbols in W.)

The Exhaustive Replacement (ER) of a λ-free language M in a language L, using a special symbol $\dagger$ not in the alphabet, can be simply defined as

$$L \rhd M = \bigcup_{W \in L} \rho_M^*(W).$$

Intuitively, the exhaustive replacement $L \rhd M$ replaces in every word $W \in L$ the non-overlapping occurrences of words from M with the special symbol $\dagger$. Moreover, between two occurrences of words of M to be replaced, no nonempty word from M appears as a subword.

Example 1 Let $L = \{RSRSRSR, RRSRSR, RSRRSRRSR\}$, $M = \{RSR\}$. Then

$$L \rhd M = \{\dagger S \dagger, RS\dagger SR, R\dagger SR, RRS\dagger, \dagger\dagger\dagger\},$$

being the union of the sets:

$$\begin{aligned} \rho_{\{RSR\}}^*(\{RSRSRSR\}) &= \{\dagger S \dagger, RS\dagger SR\}, \\ \rho_{\{RSR\}}^*(\{RRSRSR\}) &= \{R\dagger SR, RRS\dagger\}, \\ \rho_{\{RSR\}}^*(\{RSRRSRRSR\}) &= \{\dagger\dagger\dagger\}. \end{aligned}$$

4 Computing the Replacement Operation

In this section we will present an algorithm for computing the partial rewriting of a database query. To this end, we will give first a characterization of the ER-operator. The construction in the proof of our characterization provides the basic algorithm for computing the result of the ER-operator on given languages. The construction is based on finite transducers.

A *finite transducer* $T = (S, I, O, \delta, s, F)$ consists of a finite set of states S, an input alphabet I, and output alphabet O, a starting state s, a set of final states F, and a transition-output function δ from finite subsets of $S \times I^*$ to finite subsets of $S \times O^*$. Intuitively, for instance $(q_1, W) \in \delta(q_0, U)$ means that if the transducer is in state q_0 and reads word U, it can go to state q_1 and emit the word W. For a given word $U \in I^*$, we say that a word $W \in O^*$ is an *output of T for U* if there exists a sequence $(q_1, W_1) \in \delta(s, U_1)$, $(q_2, W_2) \in \delta(q_1, U_2)$, $\ldots$, $(q_n, W_n) \in \delta(q_{n-1}, U_n)$ of state transitions of T, such that $q_n \in F$, $U = U_1 \ldots U_n \in I^*$, and $W = W_1 \ldots W_n \in O^*$. We write $W \in T(U)$, where $T(U)$ denotes the set of all outputs of T for the input word U. For a language $L \subseteq I^*$, we define $T(L) = \bigcup_{U \in L} T(U)$.

We are now in a position to state our characterization theorem.

Theorem 1 *Let L and M be regular languages over an alphabet Δ. There exists a finite transducer T and a regular language M' such that:*

$$L \rhd M = T(L) \cap M'.$$

Proof sketch. Let $A = (S, \Delta, \delta, s_0, F)$ be a nondeterministic finite automaton that accepts the language M. Let us consider the finite transducer:

$$T = (S \cup \{s_0'\}, \Delta, \Gamma, \delta', s_0', \{s_0'\}),$$

where $\Gamma = \Delta \cup \{\dagger\}$, and, written as a relation,

$$\begin{aligned} \delta' \quad = \quad & \{(s, R, s', \lambda) : (s, R, s') \in \delta\} \cup \\ & \{(s_0', R, s_0', R) : R \in \Delta\} \cup \\ & \{(s_0', R, s, \lambda) : (s_0, R, s) \in \delta\} \cup \\ & \{(s_0', R, s_0', \dagger) : (s_0, R, s) \in \delta \text{ and } s \in F\} \cup \\ & \{(s, R, s_0', \dagger) : (s, R, s') \in \delta \text{ and } s' \in F\}. \end{aligned}$$

Intuitively, transitions in the first set of δ' are the transitions of the "old" automaton modified so as to produce λ as output. Transitions in the second set mean that "if we like, we can leave everything unchanged," i.e. each symbol gives itself as output. Transitions in the third set are for non-deterministically jumping from the new initial state s_0' to the states of the old automaton A, that are reachable in one step from the old initial state s_0. These transitions give λ as output. Transitions in the fourth set are for handling special cases, when from the old initial state q_0, an old final state can be reached in one step. In these cases we can replace the one symbol words accepted by A with the special symbol $\dagger$. Finally, the transitions of the fifth set are the most significant. Their meaning is: in a state, where the old automaton has a transition by a symbol, say R, to an old final state, there will in the transducer be an additional transition $R/\dagger$ to s_0', which is also the (only) final state of T. Observe, that if the transducer T decides to leave the state s_0' while a suffix U of the input string is unscanned, and enter the old automaton A, then it can return back only if there is a prefix U' of U, such $U' \in L(A)$. In this case the trasducer replaces U', which is a subword of the input string, by the special symbol $\dagger$.

Given a word of $W \in L$ as input, the finite transducer T replaces arbitrary many occurences of words of M in W with the special symbol symbol $\dagger$.

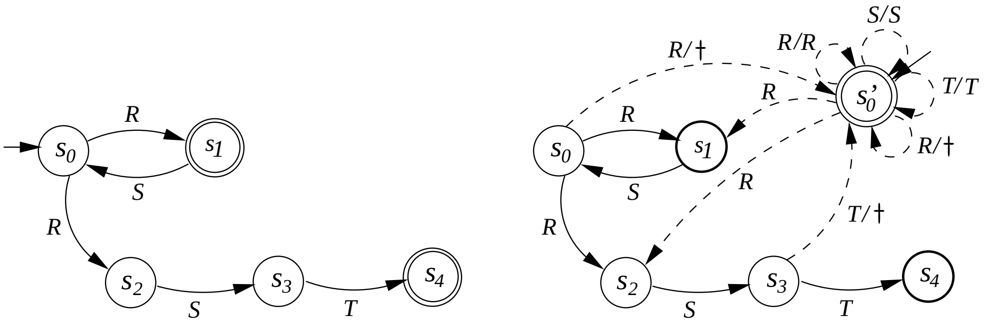

Figure 2: An example of a replacement transducer

For an example, suppose M is $R(SR)^* + RST$. Then an automaton that accepts this language is given in Figure 2 drawn with solid arrows. The corresponding rational transducer is shown in the same figure in the right. It consists of the automaton A, whose transitions now produce as output λ, plus the state s_0' and the additional transitions drawn with dashed arrows. For simplicity we have not drawn the λ outputs of some transitions. It can now be shown that

$$T(L) = L \cup \{U_1 \dagger U_2 \dagger \ldots \dagger U_k : \text{ for some } U \text{ in } L \text{ and words } W_i \text{ in } M,\ U = U_1 W_1 U_2 W_2 \ldots W_{k-1} U_k\}.$$

¿From the transduction $T(L)$ we get all the words of L having replaced in them an arbitrary number of words from M. What we like is not an arbitrary but an exhaustive replacement of words from M. To achive this goal we will intersect the language $T(L)$ with a regular language M' which will serve as a "mask" for the words of $L \triangleright M$. We set

$$M' = (\Gamma^* M \Gamma^*)^c .$$

Now M' guarantees that no other candidate for replacing occurs inside the words of the final result. ■

5 Partial P-Rewritings

We can give a natural generalization of the definition of the replacement operator for the case when we like to exhaustively replace subwords not from one language only, but from a finite set of languages (such as a finite set of view definitions). For this purpose, let W be a word and $\mathbf{M} = \{M_1, \ldots, M_n\}$ be a set of languages over some aplhabet, and let $\{\dagger_, \ldots, \dagger_n\}$ be a set of symbols outside that alphabet. Now we define

$$\rho_{\mathbf{M}}(W) = \begin{cases} \{W_1 \dagger_i W_3 : \text{ there is a } W_2 \in M_i \text{ such that } W = W_1 W_2 W_3\} & \text{if non-empty} \\ \{W\} & \text{otherwise.} \end{cases}$$

The *generalized exhaustive replacement* of $\mathbf{M} = \{M_1, \ldots, M_n\}$ in a language L, by the corresponding special symbols $\dagger_1, \ldots, \dagger_n$, is:

$$L \triangleright \mathbf{M} = \bigcup_{W \in L} \rho^*_{\mathbf{M}}(W).$$

In the following we will define the notion of the *partial p-rewriting* of a database query Q using a set $\mathbf{V} = \{V_1, \ldots, V_n\}$ of view definitions.

Definition 1 The *partial p-rewriting* of a query Q over Δ using a set $\mathbf{V} = \{V_1, \ldots, V_n\}$ of view definitions over Δ is:

$$Q \rhd \mathbf{V},$$

with $\Omega = \{v_1, \ldots, v_n\}$ as the corresponding set of special symbols.

As a generalization of Theorem 1 we can give the following result about the partial p-rewriting of a query Q over Δ using a set $\mathbf{V} = \{V_1, \ldots, V_n\}$ of view definitions over Δ.

Theorem 2 *The partial p-rewriting $Q \rhd \mathbf{V}$ can be effectively computed.*

Proof sketch. Let $A_i = (S_i, \Delta, \delta_i, s_{0i}, F_i)$, for $i \in [1, n]$ be n nondeterministic finite automata that accept the corresponding V_i languages. Let us consider the finite transducer:

$$T = (S_1 \cup \ldots \cup S_n \cup \{s'_0\}, \Delta, \Delta \cup \Omega, \delta', s'_0, \{s'_0\}),$$

where

$$\begin{aligned}
\delta' \quad = \quad & \{(s, R, s', \lambda) : (s, R, s') \in \delta_i,\ i \in [1, n]\} \cup \\
& \{(s'_0, R, s'_0, R) : R \in \Delta\} \cup \\
& \{(s'_0, R, s, \lambda) : (s_{0i}, R, s) \in \delta_i,\ i \in [1, n]\} \cup \\
& \{(s'_0, R, s'_0, \dagger_i) : (s_{0i}, R, s) \in \delta_i \text{ and } s \in F_i,\ i \in [1, n]\} \cup \\
& \{(s, R, s'_0, \dagger_i) : (s, R, s') \in \delta_i \text{ and } s' \in F_i,\ i \in [1, n]\}.
\end{aligned}$$

The transducer T performs the following task: given a word of Q as input, it replaces nondeterministically some words of $V_1, \ldots, V_n$ from the input with the corresponding special symbols. The proof of this claim is similar as in the previous theorem.

¿From the tranduction $T(Q)$ we get all the words of Q having replaced in them an arbitrary number of words from $V_1 \cup \ldots \cup V_n$. But what we like is the exhaustive replacement $Q \rhd \mathbf{V}$. For this we intersect the language $T(Q)$ with the regular language

$$((\Delta \cup \Omega)^* (V_1 \cup \ldots \cup V_n)(\Delta \cup \Omega)^*)^c,$$

which will serve as a mask for extracting the words in the exhaustive replacement. ■

We note here that the partial p-rewriting of a query is a generalization of the p-rewriting. Indeed, consider the the substitution from $\Omega \cup \Delta$ that maps each $v_i \in \Omega$ to the corresponding regular view language V_i and each database symbol $R \in \Delta$ to itself. This substitution is the extension of the *def* substitution to the Δ alphabet and we call it *def*$'$. Then the partial p-rewriting is the set of *all* the words on $\Omega \cup \Delta$, with no subwords in any $V_1, \ldots, V_n$, such that the result on them of the *def*$'$ substitution, has a non empty intersection with Q. The conceptual similarity of the partial p-rewriting with p-rewriting can also be observed in another way; change the above mask to Ω^* and the result will be the p-rewriting, as opposed to the partial p-rewriting.

6 Partial L-Rewritings

We defined the l-rewriting of a query Q given a set of view definitions $\mathbf{V} = \{V_1, \ldots, V_n\}$ as the set of all the words on the view alphabet Ω such that their substitution by *def* is contained in the query language Q. In the same spirit we will define the partial l-rewriting. It will be the set of all the "mixed" words on the alphabet $\Omega \cup \Delta$, with no subword in $(V_1 \cup \cdots \cup V_n)$, such that their substitution by the extended def' is contained in the query Q. The condition that there is no subword in $(V_1 \cup \cdots \cup V_n)$, says that in fact the partial l-rewriting is a subset of the partial p-rewriting.

Definition 2 The *partial l-rewriting* of a query Q on Δ is the language Q' on $\Omega \cup \Delta$ given by

$$Q' = \{U \in Q \rhd \mathbf{V} : def'(U) \subseteq Q\}.$$

We now give a method for computing the partial l-rewriting.

Algorithm 1

1. *Compute the complement Q^c of the query.*
2. *Construct the transducer T used for the partial p-rewriting. Then compute the transduction $T(Q^c)$ of the complement of the query.*
3. *Compute the complement $(T(Q^c))^c$ of the previous transduction.*
4. *Intersect the complement $(T(Q^c))^c$ with the mask*

$$M = ((\Delta \cup \Omega)^* (V_1 \cup \ldots \cup V_n)(\Delta \cup \Omega)^*)^c$$

Denote with Q' the result. ■

Theorem 3 *The mixed $\Omega \cup \Delta$ language Q' gives exactly the partial l-rewriting of Q.*

Proof. "$\subseteq$". $T(Q^c)$ is the set of all words U on $\Omega \cup \Delta$ such that $def'(U) \cap Q^c \neq \emptyset$. Hence, $(T(Q^c))^c$, being the complement of this set, will contain only $\Omega \cup \Delta$ words such that *all* the Δ-words in their substitution by def' will be contained in Q. This is the first condition for a word on $\Omega \cup \Delta$ to be in the partial l-rewriting of Q. Furthermore, intersecting with the above mask we keep in $(T(Q^c))^c$ only the $\Omega \cup \Delta$ words that do not contain Δ subwords in $(V_1 \cup \cdots \cup V_n)$. This is the second condition for a word on $\Omega \cup \Delta$ to be in the partial l-rewriting of Q.
"$\supseteq$". We will prove this direction by a contradiction. First observe that both the partial l-rewriting and the set Q' are subsets of the partial p-rewriting, that is, all their words "pass" the above mask. In other words their words do not have subwords in $(V_1 \cup \cdots \cup V_n)$. Suppose now, that the mixed $\Omega \cup \Delta$-word $U = U_1 \ldots U_n$ is in the partial l-rewriting but not in Q'. That is $def'(U) \subseteq Q$. On the other hand, since $U \notin Q'$ it follows that $U \in Q'^c$ which means that $U \in T(Q^c) \cup M^c$. But as we mentioned before, the word U, which belongs in the partial l-rewriting, "passes" the mask M and this implies that it cannot "pass" the complement of the mask. Therefore, $U \in T(Q^c)$. Thus $def'(U) \cap Q^c \neq \emptyset$ that is, $def'(U) \not\subseteq Q$ i.e. U cannot be in the partial l-rewriting, a contradiction. ■

7 Query Optimization Using Partial Rewritings and Views

In this section we show how to utilize partial rewritings in query optimization in a scenario where we have available a set of precomputed views, as well as the database itself. The views could be materialized views in a warehouse, or locally cached results from previous queries in a client/server environment. In this scenario the views are assumed to be excact, and we are interested in answering the query by consulting the views as far as possible, and by accessing the database only when necessary.

Formally, let $\Omega = \{v_1, \ldots v_n\}$ be the view alphabet and let $\mathbf{V} = \{V_1, \ldots, V_n\}$ be a set of view definitions as before. Given a database DB, which is a graph, where the edges are labelled with database symbols from Δ, we define the *view graph* $\mathcal{V}$ over $(\mathbf{V}, \Omega)$ to be a database over (D, Ω) induced by the set

$$\bigcup_{i \in \{1,\ldots,n\}} \{(a, v_i, b) : (a, b) \in ans(V_i, DB)\}.$$

of Ω-labelled edges.

It is now straightforward to show, that if the l-rewriting Q' is exact (meaning $def(Q') = Q$), then $ans(Q, DB) = ans(Q', \mathcal{V})$ (see Calvanese et al. [CGLV2000]).

However, the cases when we are able to obtain an exact rewriting of the query using the views would be rare in practice, in the general we have in the views only part of the information needed to answer the query. So, should we ignore this partial information only beacuse it is not complete? In the previous sections we showed how this partial information can be captured algebraically by the partial rewritings. In the following, we use the partial rewritings not to avoid *completely* accesing database, but to *minimize* such access as much as possible.

However, in order to be able to utilize the partial l-rewriting Q', it should be exact, i.e. we require that $def'(Q') = Q$. We can use for testing the exactness the optimal algorithm of [CGLV99].

Given an exact partial l-rewriting, we can use it to evaluate the query on the view-graph, and accessing the database in a "lazy" fashion, only when necessary. Before describing the lazy algorithm, let us review how query answering on semistructured databases typically works [ABS99].

Algorithm 2 We are given a regular expression Q and a database graph DB. First construct an automaton A_Q for Q. Let N be the set of nodes in the database graph, and let $\{s_0, s_1, s_2, \ldots s_m\}$ be the set of states in A_Q, with s_0 being the initial state. For each node $a \in N$ compute a set $Reach_a$ as follows.

1. Initialize $Reach_a$ to $\{(a, s_0)\}$.

2. Repeat 3 until $Reach_a$ no longer changes.

3. Choose a pair $(x, s) \in Reach_a$. If there is a database symbol R, such that a transition $s \xrightarrow{R} s'$ is in A_Q and an edge $a \xrightarrow{R} a'$ is in the database DB, then add the pair (x', a') to $Reach_a$.

Finally, set $ans(Q, DB) = \{(a, b) : a \in N, (b, s) \in Reach_a,$ and s is a final state in $A_Q\}$. ■

In the following we modify this algorithm into a lazy algorithm for answering a query Q using its partial l-rewriting with respect to a set of cached exact views.

Algorithm 3 We are given an automaton $A_{Q'}$, corresponding to an exact partial l-rewriting Q' and the view graph $\mathcal{V}$. Let N be the set nodes in $\mathcal{V}$, and let $\{s_0, s_1, s_2, \ldots s_m\}$ be the states in $A_{Q'}$. For each node $a \in N$ then compute a set $Reach_a$.

1. Initialize $Reach_a$ to $\{(a, s_0)\}$, and $Expanded_a$ to **false**.
2. For each database symbol R, if there is in $A_{Q'}$ a transition $s_0 \xrightarrow{R} s$ from the initial state s_0, then access the database and add to $\mathcal{V}$ the subgraph of DB induced by the R-edges.
3. Repeat 4 until $Reach_a$ no longer changes.
4. Choose a pair $(x, s) \in Reach_a$. If there is a view or database symbol R, such that a transition $s \xrightarrow{R} s'$ is in A_Q, go to 5.
5. If there is an edge $a \xrightarrow{R} a'$ in the viewgraph, add the pair (x', a') to $Reach_a$. Otherwise, if $Expanded_a$ = **false**, set $Expanded_a$ = **true**, access the database and add to $\mathcal{V}$ the subgraph of DB induced by all edges originating from a.

Set $eval(Q', \mathcal{V}, DB) = \{(a, b) : a \in N, (b, s) \in Reach_a,$ and s is a final state in $A_{Q'}\}$. ■

Theorem 4 *Given a query Q and a set of exact views, if the partial l-rewriting Q' of Q is exact, then* $\text{eval}(Q', \mathcal{V}, DB) = \text{ans}(Q, DB)$.

Next, let us discuss how to utilize the partial p-rewriting Q'' of a query Q for computing the answer set $ans(Q, DB)$. If we use the same algorithm as in the case of the partial l-rewriting we might get a proper superset of the answer. Note however that, contrary to Algorithm 3, in any case the partial p-rewriting does not need to be exact.

Theorem 5 *Given a query Q and a set $\mathcal{V}$ of exact views, if Q'' is the partial p-rewriting of Q using $\mathcal{V}$, then* $\text{ans}(Q, DB) \subseteq eval(Q'', \mathcal{V}, DB)$.

In other words, we are not sure if all the pairs are valid. To be able to discard false hits, suppose that the views are materialized using Algorithm 2. We can then associate each pair (a, b) in the view graph with their derivation. That is, for each pair (a, b) connected with an edge, say v_i, in the view graph, we have an automaton, say A_{ab}, with start state a and final states $\{b\}$. What is this automaton? For each pair (a, b), we can consider the database graph as a non-deterministic automaton DB_{ab} with initial state a and final states $\{b\}$. It is now easy to see that

$$A_{ab} = DB_{ab} \cap A_{V_i}$$

where A_{V_i} is an automaton for the view V_i. We are now ready to formulate the algorithm for using the partial p-rewriting in query answering.

Algorithm 4

1. Compute *eval*$(Q'', \mathcal{V}, DB)$ using Algorithm 3. During the execution of Algorithm 3 the view graph $\mathcal{V}$ is extended with new edges and nodes as described. Call the extended view graph it $\mathcal{V}'$.

2. Replace in $\mathcal{V}$' each edge labeled with a view symbol, say v_i, between two objects a and b with the automaton A_{ab} of the derivation. Call the new graph $\mathcal{V}$".

3. Set *verified*$(Q'', \mathcal{V}, DB) =$ *eval*$(Q'', \mathcal{V}, DB) \cap \{(a, b) : Q \cap \mathcal{V}''_{ab} \neq \emptyset\}$. ■

Theorem 6 *Given a query Q and a set $\mathcal{V}$ of exact views, if Q'' is the partial p-rewriting of Q using $\mathcal{V}$, then $verified(Q'', \mathcal{V}, DB) = \mathrm{ans}(Q, DB)$.*

8 Complexity Analysis

The following theorem establishes an upper bound for the problem of generating the exhaustive replacement $L \triangleright M$, where L and M are regular languages.

Theorem 7 *Generating the exhaustive replacement of a regular language M from another language L can be done in exponential time.*

Proof. Let us refer to the cost of the steps in the constructive proof of the Theorem 1. To construct a non-deterministic automaton for the language M and using it to construct the transducer g is polynomial. To compute the transduction of the regular language L, $g(L)$, is again polynomial. But at the end, in order to compute the subset of the words in $g(L)$, to which no more replacement can be applied, is exponential. This is because we intersect with a mask that is a language described by an extended regular language containing complementation. ■

Theorem 8 *Let Γ be an alphabet and A, B be regular languages over Γ. Then the problem of deciding the emptiness of $A \cap (\Gamma^* B \Gamma^*)^c$ is PSPACE complete.*

The proof of this theorem is given in the Appendix. We are now in a position to prove the following result.

Theorem 9 *There exist regular languages L and M, such that the exhaustive replacement $L \triangleright M$ cannot be computed in polynomial time, unless PTIME=PSPACE.*

Proof. Suppose that given two regular expressions A and B on alphabet Γ we like to test the emptiness of $A \cap (\Gamma^* B \Gamma^*)^c$. Without loss of generality let us assume that there exists one symbol in A that does not not appear in B. To see why even with this restriction the above problem of emptiness is still PSPACE complete, imagine that we can simply have a tape symbol which does not appear at all in the definition of the transition function of the Turing machine. Then this symbol will appear in the above set A but not in B (see Appendix). Let us denote this special symbol with †. We substitute the † symbol in A with the regular expression B. The result will be another regular expression A' which has polynomial size. Clearly, $A \cap (\Gamma^* B \Gamma^*)^c = A \cap (A' \triangleright B)$.

As a conclusion, if we had a polynomial time algorithm producing a polynomial size representation for $A' \triangleright B$, we could polynomially construct an NFA for $A \cap (A' \triangleright B)$. Then we could check in NLOGSPACE the emptiness of this NFA. This means that, the emptiness of $A \cap (\Gamma^* B \Gamma^*)^c$ could be checked in PTIME, which is a contradiction, unless PTIME=PSPACE. ∎

Corollary 1 *The algorithm in the proof of Theorem 2 for computing the partial p-rewriting of a query Q using a set $\mathbf{V} = \{V_1, \ldots, V_n\}$ of view definitions, is essentially optimal.*

Theorem 10 *Given a query Q and a set $\mathbf{V} = \{V_1, \ldots, V_n\}$ of view definitions, to compute the partial l-rewriting is in 2EXPTIME.*

Proof. Let us refer to the constructive proof of the Theorem 3. To compute the complement Q^c of the query is exponential. To transduce it to $T(Q^c)$ is polynomial. To complement again is exponential. So, in total we have 2EXPTIME. To compute the mask is EXPTIME and to intersect is polynomial. Finally, 2EXPTIME + EXPTIME = 2EXPTIME. ∎

For the partial lower rewriting we have the following.

Theorem 11 *The presented Algorithm 1 for computing the partial l-rewriting of a query Q using a set $\mathbf{V} = \{V_1, \ldots, V_n\}$ of view definitions, is essentially optimal.*

Proof. Polynomially intersect the partial l-rewriting with Ω^* and get the l-rewriting of [CGLV99]. But, the l-rewriting is optimally computed in doubly exponential time in [CGLV99], so our algorithm is essentially optimal. ∎

References

[Abi97] S. Abiteboul. Querying Semistructured Data. *Proc. of ICDT* 1997 pp. 1-18.

[ABS99] S. Abiteboul, P. Buneman and D. Suciu. *Data on the Web : From Relations to Semistructured Data and Xml.* Morgan Kaufmann, 1999.

[AHV95] S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Int. J. on Digital Libraries* 1997 1(1) pp. 68-88.

[Bun97] P. Buneman. Semistructured Data. *Proc. of PODS* 1997, pp. 117-121.

[BDFS97] P. Buneman, S. B. Davidson, M. F. Fernandez and D. Suciu. Adding Structure to Unstructured Data. *Proc. of ICDT* 1997, pp. 336-350.

[Brzo64] J. A. Brzozowski. Derivatives of Regular Expressions. *JACM 11(4)* 1964, pp. 481-494

[BL80] J. A. Brzozowski and E. L. Leiss. On Equations for Regular Languages, Finite Automata, and Sequential Networks. *TCS 10* 1980, pp. 19-35

[CGLV99] D. Calvanese, G. Giacomo, M. Lenzerini and M. Y. Vardi. Rewriting of Regular Expressions and Regular Path Queries. *Proc. of PODS* 1999, pp. 194-204.

[CGLV2000] D. Calvanese, G. Giacomo, M. Lenzerini and M. Y. Vardi. Answering Regular Path Queries Using Views. *Proc. of ICDE* 2000, pp. 389-398.

[CGLV2000] D. Calvanese, G. Giacomo, M. Lenzerini and M. Y. Vardi. View-Based Query Processing for Regular Path Queries with Inverse. *Proc. of PODS* 2000, pp. 58-66.

[CSS99] S. Cohen, W. Nutt, A. Serebrenik. Rewriting Aggregate Queries Using Views. Proc. of PODS 1999, pp. 155-166

[Con71] J. H. Conway. *Regular Algebra and Finite Machines.* Chapman and Hall 1971.

[DFF+99] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, D. Suciu. A Query Language for XML. *WWW8 / Computer Networks 31(11-16)* 1999, pp. 1155-116.

[DG97] O. Duschka and M. R. Genesereth. Answering Recursive Queries Using Views. *Proc. of PODS* 1997, pp. 109-116.

[FS98] M. F. Fernadez and D. Suciu. Optimizing Regular path Expressions Using Graph Schemas *Proc. of ICDE* 1998, pp. 14-23.

[FLS98] D. Florescu, A. Y. Levy, D. Suciu Query Containment for Conjunctive Queries with Regular Expressions *Proc. of PODS* 1998, pp. 139-148.

[GM99] G. Grahne and A. O. Mendelzon. Tableau Techniques for Querying Information Sources through Global Schemas. *Proc. of ICDT* 1999 pp. 332-347.

[GT2000] G. Grahne and A. Thomo. An Optimization Technique for Answering Regular Path Queries. *Proc. of WebDB* 2000.

[HU79] J. E. Hopcroft and J. D. Ullman *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley 1979.

[HRS76] H. B. Hunt and D. J. Rosenkrantz, and T. G. Szymanski, On the Equivalence, Containment, and Covering Problems for the Regular and Context-Free Languages. *Journal of Computing and System Sciences* 12(2) 1976, pp. 222-268

[Kari91] L. Kari. *On Insertion and Deletion in Formal Languages.* Ph.D. Thesis, 1991, Department of Mathematics, University of Turku, Finland.

[Lev99] A. Y. Levy. *Answering queries using views: a survey.* Submitted for publication 1999.

[LMSS95] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, D. Srivastava. Answering Queries Using Views. *Proc. of PODS* 1995, pp. 95-104.

[MW95] A. O. Mendelzon and P. T. Wood, Finding Regular Simple Paths in Graph Databases. *SIAM J. Comp. 24:6,* (December 1995).

[MMM97] A. O. Mendelzon, G. A. Mihaila and T. Milo. Querying the World Wide Web. *Int. J. on Digital Libraries 1(1)*, 1997 pp. 54-67.

[MS99] T. Milo and D. Suciu. Index Structures for Path Expressions. *Proc. of ICDT*, 1999, pp. 277-295.

[PV99] Y. Papakonstantinou, V. Vassalos. Query Rewriting for Semistructured Data. proc. of SIGMOD 1999, pp. 455-466

[Ull97] J. D. Ullman. Information Integration Using Logical Views. *Proc. of ICDT* 1997, pp. 19-40.

[Var88] M. Y. Vardi. The universal-relation model for logical independence. *IEEE Software.*

[Yu97] S. Yu. Reqular Languages. In: *Handbook of Formal Languages.* G. Rozenberg and A. Salomaa (Eds.). Springer Verlag 1997, pp. 41-110

9 Appendix

Theorem 8 *Let Γ be an alphabet and A, B be regular languages over Γ. Then the problem of deciding the emptiness of $A \cap (\Gamma^* B \Gamma^*)^c$ is PSPACE complete.*

Proof. First, observe that

$$[A \cap (\Gamma^* B \Gamma^*)^c = \emptyset] \Leftrightarrow [A \subseteq \Gamma^* B \Gamma^*]$$

But, this problem is a sub-case of the problem of testing regular expression containment, which is known to be PSPACE complete [HRS76]. So, there exists an algorithm running in polynomial space that decides the above problem.

Next, we show that the problem is PSPACE-hard. Let $\mathcal{L}$ be a language that is decided by a Turing machine M running in polynomial space n^k for some constant k. The reduction maps an input w into a pair of regular expressions explained in the following.

Let's denote with Γ the alphabet consisting of all symbols that may appear in a computation history. If Σ and Q are the M's tape alphabet and states, then $\Gamma = \Sigma \cup Q \cup \{\#\}$. We assume that all configurations have length n^k and are padded on the right by blank symbols if they otherwise would be too short. Let's suppose for a moment that we have organized some configurations in a tableau where each row of the tableau contains a configuration and we mark the begining and the end of each one by the marker $\#$. Now, in this organization we consider all the 2×3 windows. A window is legal if that window does not violate the actions specified by the M's transition function. In other words, a window is legal if it might appear when each configuration correctly follows another. By a proved claim in the the proof of the

Cook-Levin Theorem we know that, if the top row of the table is the start configuration and every window in the table is legal, each row of the table is a configuration that legally follows the preceding one. We encode a set of configurations $C_1 \ldots C_l$ as a single string, with the configurations separated from each other by the # symbol as shown in the following figure.

$$\#\underbrace{\quad}_{C_1}\#\underbrace{\quad}_{C_2}\#\cdots\#\underbrace{\quad}_{C_l}\#$$

Now, we can describe the set of words in Γ^* with at least one illegal window with the following regular expression.

$$\Gamma^* B \Gamma^*$$

where

$$B = \bigcup_{bad(abc,def)} abc\Gamma^{(n^k-2)}def.$$

Clearly, the set of configuration sequences with no illegal windows is described by

$$(\Gamma^* B \Gamma^*)^c.$$

What we need now, is be able to extract from the set of sequences of this form, an accepting computation history for the input w. We already have assured that there is not any illegal window. After that, we need two more things: the start configuration C_1 must be

$$\#q_0 w_1 \ldots w_n \underbrace{\sqcup \ldots \sqcup}_{n^k-n} \#,$$

where $w = w_1 \ldots w_n$, and there must appear a symbol q_{accept}. We encapture the condition about C_1 by the regular expression

$$A_1 = \#q_0 w_1 \ldots w_n \sqcup^{n^k-n} \#\Gamma^*,$$

and the condition that there should be an accepting configuration by the regular expresion

$$A_2 = \Gamma^* q_{accept} \Gamma^*.$$

Putting A_1 and A_2 together we have the following regular expression

$$A = A_1 \cap A_2 = \#q_0 w_1 \ldots w_n \sqcup^{n^k-n} \#\Gamma^* q_{accept} \Gamma^*.$$

Summarising, there is an accepting computation of M on input w if and only if

$$A \cap (\Gamma^* B \Gamma^*)^c \neq \emptyset.$$

We finish the proof by emphasizing that the size of the above expression is polynomial. ∎