Towards Practically Feasible Answering of Regular Path Queries in LAV Data Integration

Manuel Tamashiro
University of Victoria
British Columbia, Canada
manuelt@cs.uvic.ca

Alex Thomo
University of Victoria
British Columbia, Canada
thomo@cs.uvic.ca

Srinivasan Venkatesh
University of Victoria
British Columbia, Canada
venkat@cs.uvic.ca

ABSTRACT

Regular path queries (RPQ's) are given by means of regular expressions and ask for matching patterns on labeled graphs. RPQ's have received great attention in the context of semistructured data, which are data whose structure is irregular, partially known, or subject to frequent changes. One of the most important problems in databases today is the integration of semistructured data from multiple sources modeled as views. The well-know paradigm of computing first a view-based rewriting of the query, and then evaluating the rewriting on the view extensions is indeed possible for RPQ's. However, computing the rewriting is computationally hard as it can only be done (in the worst case) in not less than 2EXPTIME. In this paper, we provide practical evidence that computing the rewriting is hard on the average as well. On the positive side, we propose automata-theoretic techniques, which efficiently compute and utilize instead the complement of the rewriting. Notably using the latter, it is possible to answer a query, and this makes the view-based answering of RPQ's fairly feasible in practice.

1. INTRODUCTION

Regular path queries (RPQ's) are in essence regular expressions over a fixed database alphabet. They have received a great deal of attention in the recent years due to the well-known semistructured data model. Semistructured data is data whose structure is irregular, partially known, or subject to frequent changes [1]. They are commonly found in a multitude of applications in areas such as communication and traffic networks, web information systems, digital libraries, biological data management, etc.

Semistructured data are formalized as edge labeled graphs, and there is an inherent need to navigate these graphs by means of a recursive query language. As pointed out by seminal works in the field (cf. [7, 15, 4, 3, 6]), regular path queries (RPQ's) are the "winner" when it comes expressing navigational recursion over graph data. These queries are in essence regular expressions over the database edge symbols, and in general, one is interested in finding query-matching database paths, which spell words in the (regular) query language. For example, the RPQ

$$Q = AirCanada^* \cdot (Lufthansa + \epsilon)$$

asks for all the pairs of cities connected by (possibly multihop) Air Canada routes, followed by a last optional segment serviced by the partner company Lufthansa. We can observe that evaluating RPQ's on semistructured databases amounts to [regular expression] pattern matching on graphs as opposed to strings.

Now, suppose that we do not have a database available. Rather, what we have is a set of views on the possible data. These views represent partial information about the database and are expressed by regular expressions as well. For example, we could be given two views with definitions $V_1 = AirCanada \cdot AirCanada$ and $V_2 = Lufthansa$. Notably, the view definitions are nothing else but regular path queries. Additionally, for each view, we are given a set of pairs that represent the answer to these views (considering them as RPQ's).

This is the classical scenario in LAV ("local-as-view") data integration (cf. [8, 4, 3, 13, 6, 2, 11]). The basic problem in this setting is to be able to answer a given query using only the available view information. This is a very important problem which emerges in a variety of situations both commercial (when two similar companies provide partial access to their data) and scientific (combining research results from different bioinformatics repositories). Data integration appears with increasing frequency as the volume and the need to share existing data explodes.

Answering queries using views is typically achieved by reformulating the query in terms of the view definitions and then evaluating it on the provided view data. For example, the above query Q can be reformulated (or rewritten) as $Q' = V_1^* \cdot (V_2 + \epsilon)$. Then, if there are pairs (a, b) and (b, c) associated with views V_1 and V_2 respectively, we produce (a, c) as an answer to Q. Of course, had we a database in the classical sense, we would be able to also produce pairs of nodes connected by paths with an *odd* number of Air Canada segments followed by an optional Lufthansa segment. However, for the given views (recall $V_1 = AirCanada \cdot AirCanada$) this is not possible.

There are two lines of research for answering RPQ's in LAV data integration. Works in the first line of research study the computation of rewritings, which are regular path queries over the view names (cf. [3, 9]). Having such rewritings is desirable because they enable query answering in polynomial time with respect to the size of the data. On the other hand,

works in the second line of research study the perfect (or certain) answering of RPQ's, which asks for all the answer tuples obtainable on every possible database consistent with the views (cf. [4, 5, 11]). However, obtaining all the certain answer is much more computationally expensive. As it has been shown in [4], to decide whether a tuple belongs in the certain answer is coNP-complete with respect to the size of data. Notably, the answer obtained by using a rewriting is a subset (sometimes strict) of the certain answer, and thus, by using a rewriting we get a lower approximation, which might be an acceptable compromise given the data complexity of computing the full certain answer (see [6] for a discussion).

The most important cornerstone in the rewriting of RPQ's using views is the work by Calvanese, De Giacomo, Lenzerini, and Vardi [3], which shows that the rewriting is indeed possible by giving an algorithm for computing it. The complexity of computing the (maximal) view-based rewriting of a regular path query Q is shown to be in 2EXPTIME (in the size of the query) and this bound is also shown to be tight ([3]). Also, in [3] it is shown that the size of the automaton for the rewriting can be doubly exponential in the size of the query Q as measured by the size of a simple NFA for Q.

It should be clear what the inherent problem complexity of 2^{2^n} (tight) faces us with in practice. If n, the query size, is just 6 for example, then only printing a doubly exponential rewriting would need about $2^{2^6} \approx 18 \cdot 10^{18}$ instructions that is $18 \cdot 10^{18}/(30,000 \cdot 10^6 \cdot 60 \cdot 60 \cdot 24 \cdot 365) \approx 19$ years for a modern Intel processor working at about $30,000$ millions of instructions per second.

This illustrates that obtaining a view-based rewriting is computationally hard except for very small query instances. However, it is possible to argue that the analysis in [3] is worst-case and hence it might take only reasonable amount of time to compute rewritings on the average. Unfortunately, our experimental results indicate that this is not the case (see Section 6). Experimentally, we were unable to compute[1] the view-based rewriting, in reasonable time and space, for about one third of the time while working on "randomly generated" instances. This gives us evidence that computing rewritings is indeed hard on the average as well. We believe that this observation is an important contribution of our paper given the importance of the database problem being studied.

In order to make feasible the answering of RPQ's using views, we examine each step in the method of [3]. Then, we show that we can in fact avoid the most expensive step in the method by evaluating instead the complement of the rewriting on the view data. The complement is in the form of an NFA as opposed to a DFA for the rewriting (if the latter is fully computed). This might suggest that the evaluation on the view data would be slower compared to the evaluation of the DFA for the rewriting. Of course, this is relevant only for the cases when the rewriting can be computed in reasonable time and space. Nevertheless, we show that even in such cases, by using a bitvector implementation of NFA's, we can achieve similar performance and sometimes even better. This is attributed to hardware parallelism and better hardware cache utilization.

We also found that a seemingly inexpensive polynomial step in the method of [3] was a serious performance bottleneck. In order to overcome it, we show a simple optimization which gives more than six fold speedup.

In short, we show that by partially employing the method of [3], and using our techniques, the hard problem of answering regular path queries using views becomes practically (fairly) feasible.

This paper is the first to shed light on the practical feasibility of the very basic and important problem of answering RPQ's in LAV data integration systems and to provide positive results in this direction.

The rest of the paper is organized as follows. In Section 2, we formally define semistructured databases, regular path queries, and their semantics. In Section 3, we discuss the query answering in LAV information integration systems. Next, in Section 4, we examine the algorithm of [3] for obtaining maximal view-based rewritings. Then, in Section 5 we present our optimization techniques. We show our experimental evaluations in Section 6. Finally, Section 7 concludes the paper.

2. SEMISTRUCTURED DATABASES AND REGULAR PATH QUERIES

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.

Example 1. We show in Figure 1 a database with information about an online store, which sells books and software products. A book has an author and covers some software product(s). A software product has a company and possibly other software subproducts. A company might recommend some books for its products. The database is semistructured because the schemas of its objects are not rigid. For example, a company can only optionally recommend books, or we might be missing information about what products a book might cover. □

Formally, let Δ be a finite alphabet. We shall call Δ the *database alphabet.* Elements of Δ will be denoted $R, S, \ldots$. As usual, Δ^* denotes the set of all finite words over Δ. Words will be denoted by $u, w, \ldots$. We also assume that we have a universe of objects, and objects will be denoted $a, b, c, \ldots$. A *database DB* over Δ is a subset of $N \times \Delta \times N$, where N is a set of objects, that we usually will call nodes. We view a database as a directed labeled graph, and interpret a triple (a, R, b) as a directed edge from object a to object b, labeled with R. If there is a path labeled $R_1, R_2, \ldots, R_k$ from a to b, we write $a \xrightarrow{R_1 R_2 \ldots R_k} b$.

A *(user) query* Q is a regular language over Δ. For the ease of notation, we will blur the distinction between regular

[1] Using the method of [3]

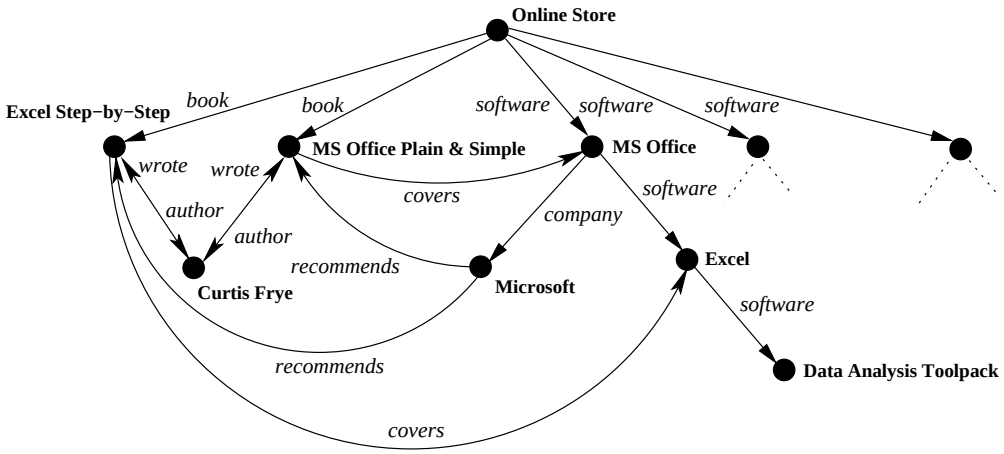


Figure 1: A graph database.

languages and regular expressions that represent them. Let Q be a query and DB a database. Then, the *answer* to Q on DB is defined as

$$ans(Q, DB) \quad = \quad \{(a, b) : a \xrightarrow{w} b \text{ in } DB \text{ for some } w \in Q\}.$$

Example 2. Consider again the database in Figure 1. Suppose that the user would like to know for each software product, all the books that might have some useful information about the product. For this, the user can give the regular path query $Q = covers \cdot software^*$. This query, on the database DB in Figure 1, will have as answer

$ans(Q, DB)$ =
{(MS Office Plain & Simple, MS Office),
(MS Office Plain & Simple, Excel),
(MS Office Plain & Simple, Data Analysis Toolpack),
(Excel Step-by-Step, Excel),
(Excel Step-by-Step, Data Analysis Toolpack)}

□

The well-known method for answering RPQ's on a given database (cf. [1]) is as follows. In essence, we create state-object pairs from the query automaton and the database. For this, let $\mathcal{A}$ be an NFA that accepts an RPQ Q. Starting from an object a of a database DB, we first create the pair (p_0, a), where p_0 is the initial state in $\mathcal{A}$. Then, we create all the pairs (p, b) such that there exist a transition from p_0 to p in $\mathcal{A}$, and an edge from a to b in DB, and furthermore the labels of the transition and the edge match. In the same way, we continue to create new pairs from existing ones, until we are not anymore able to do so. In essence, what is happening is a lazy construction of a Cartesian product graph of the query automaton with the database graph. Of course, only a small (hopefully) part of the Cartesian product is really constructed depending on the selectivity of the query.

After obtaining the above Cartesian product graph, producing query answers becomes a question of computing reachability of nodes (p, b), where p is a final state, from (p_0, a), where p_0 is the initial state. Namely, if (p, b) is reachable from (p_0, a), then (a, b) is a tuple in the query answer.

3. VIEWS IN INFORMATION INTEGRATION SYSTEMS

Let $V_1, \ldots, V_n$ be languages (queries) on alphabet Δ. We will call them *views* and associate with each V_i a view name v_i.

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 Ω alphabet the language V_i. The substitution *def* is applied to words, languages, and regular expressions in the usual way (see e.g. [19]).

A *view graph* is database $\mathcal{V}$ over Ω. In other words, a view graph is a database where the edges are labeled with symbols from Ω. View graphs can also be queried by regular path queries over Ω. However, as explained below these are not queries given by the user, but rather rewritings computed by the system.

In a LAV ("local-as-view") information integration system [13], we have the "global schema" Δ, the "source schema" Ω, and the "assertion" $def : \Omega \to 2^{\Delta^*}$. The only extensional data available is a view graph $\mathcal{V}$ over Ω (see also [4, 5, 6, 11]).

The user queries are expressed on the global schema Δ, and the system has to answer based solely on the information provided by the views. In order to do this, the system has to reason with respect to the set of *possible databases* over Δ that $\mathcal{V}$ could represent. Under the *sound view* assumption, a view graph $\mathcal{V}$ defines a set $poss(\mathcal{V})$ of databases as follows:

$$poss(\mathcal{V}) \quad = \quad \{DB : \mathcal{V} \subseteq \bigcup_{i \in \{1,\ldots,n\}} \{(a, v_i, b) : (a, b) \in ans(V_i, DB)\}\}.$$

(Recall that $V_i = def(v_i)$.) The above definition reflects the intuition about the connection between an edge (a, v_i, b) in $\mathcal{V}$ with some path from a to b in the possible DB's, labeled by some word in V_i.

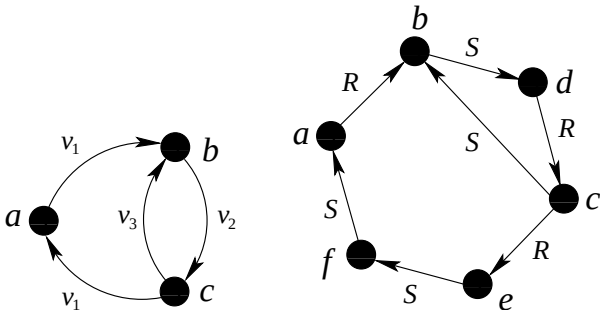


Figure 2: A view graph and a possible database.

Example 3. Consider the view graph in Figure 2 [left], and view definitions $V_1 = def(v_1) = RS^*$, $V_2 = def(v_2) = S^*R$, and $V_3 = def(v_3) = S^+$. Then, a possible database is shown in the same figure [right]. Observe that the views are sound only. They are not required to be complete. For example, we do not have a v_2-edge from f to b in the view graph. In fact, we do not even have an f object in the view graph. We remark that view soundness is usually the only "luxury" that we have in information integration systems, where the information is often incomplete. □

The meaning of querying a view graph through the global schema Δ is defined as follows. Let Q be a query over Δ. Then

$$\text{ANS}(Q, \mathcal{V}) = \bigcap_{DB \in poss(\mathcal{V})} ans(Q, DB).$$

There are two approaches for computing $\text{ANS}(Q, \mathcal{V})$. The first one is to use an *exponential* procedure in the size of the data (i.e. $\mathcal{V}$) in order to completely compute $\text{ANS}(Q, \mathcal{V})$ (see [4]). There is little that one can better hope for, since in the same paper it has been proven that to decide whether a tuple belongs to $\text{ANS}(Q, \mathcal{V})$ is co-NP complete with respect to the size of data.

The second approach is to compute first a view-based rewriting Q' for Q, as in [3]. Such rewritings are regular path queries on Ω. Then, we can approximate $\text{ANS}(Q, \mathcal{V})$ by $ans(Q', \mathcal{V})$, which can be computed in polynomial time with respect to the size of data ($\mathcal{V}$). In general, for a view-based rewriting Q' computed by the algorithm of [3], we have that

$$ans(Q', \mathcal{V}) \subseteq \text{ANS}(Q, \mathcal{V}),$$

with equality when the rewriting is exact ([4]). In the rest of the paper, we will assume that the data-integration system follows the second approach.

4. MAXIMAL VIEW-BASED REWRITINGS

Our proposed techniques enhance the computation and use of maximal view-based rewritings given in [3]. Thus, we first examine these maximal view-based rewritings and the method of [3] for their computation.

Formally, for a given query Q, the maximal view-based rewriting Q', is the set of *all* words on Ω such that their substitution through *def* is contained in the query language Q,

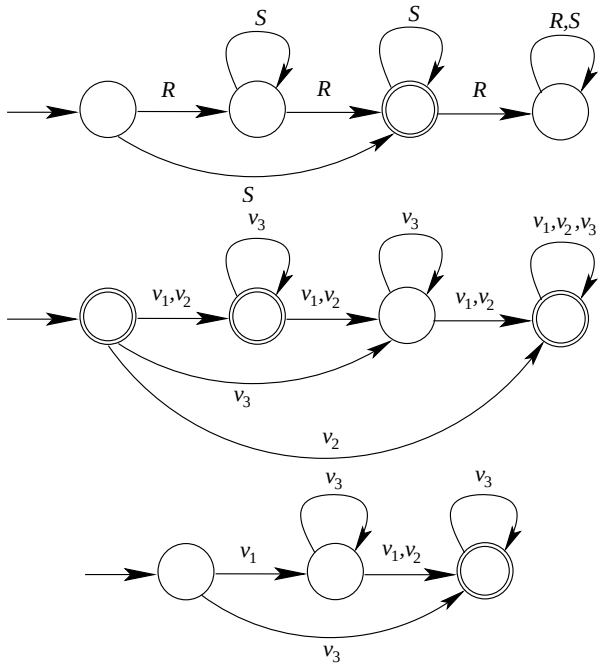


Figure 3: Automata $\mathcal{A}$ [top], $\mathcal{B}$ [middle], and $\mathcal{C}$ [bottom].

i.e.

$$Q' = \{w : w \in \Omega^* \text{ and } def(w) \subseteq Q\}.$$

Interestingly, as shown in [3], the above set is a regular language on Ω and the algorithm of [3] for computing an automaton for this language is as follows.

ALGORITHM 1.

1. *Construct a DFA $\mathcal{A} = (\Delta, S, s_0, \tau_{\mathcal{A}}, F)$ such that $Q = L(\mathcal{A})$.*
2. *Construct automaton $\mathcal{B} = (\Omega, S, s_0, \tau_{\mathcal{B}}, S - F)$, where $(s_i, v_x, s_j) \in \tau_{\mathcal{B}}$ iff there exists $w \in V_x$ such that $(s_i, w, s_j) \in \tau^*_{\mathcal{A}}$.*
3. *The rewriting Q' is the Ω language accepted by an automaton $\mathcal{C}$ obtained by complementing automaton $\mathcal{B}$.*

□

Step 2 can also be expressed equivalently as: Consider each pair of states (s_i, s_j). If in $\mathcal{A}$ there is a path from s_i to s_j, which spells a word in some view language V_x, then insert a corresponding v_x-transition from s_i to s_j in $\mathcal{B}$.

Example 4. Let query be $Q = (RS^*)^2 + S^+$ and the view definitions be as in Example 3, i.e. $V_1 = def(v_1) = RS^*$, $V_2 = def(v_2) = S^*R$, and $V_3 = def(v_3) = S^+$. The DFA $\mathcal{A}$ for the query Q is shown in Figure 3 [top] and the corresponding automaton $\mathcal{B}$ is shown in in Figure 3 [middle].

The resulting complement automaton $\mathcal{C}$ is shown in Figure 3 [bottom]. Note that the "trap" and unreachable states have been removed for clarity. □

Observe that, if $\mathcal{B}$ accepts an Ω-word $v_1 \cdots v_m$, then there exist m Δ-words w_1, ..., w_m such that $w_i \in V_i$ for $i = 1, \ldots, m$ and such that the Δ-word $w_1 \ldots w_m$ is rejected by $\mathcal{A}$. On the other hand, if there exists a Δ-word $w_1 \ldots w_m$ that is rejected by $\mathcal{A}$ such that $w_i \in V_i$ for $i = 1, \ldots, m$, then the Ω-word $v_1 \cdots v_m$ is accepted by $\mathcal{B}$. That is, $\mathcal{B}$ accepts an Ω-word $v_1 \cdots v_m$ if and only if there is a Δ-word in $def(v_1 \cdots v_m)$ that is rejected by $\mathcal{A}$. Hence, $\mathcal{C}$ being the complement of $\mathcal{B}$ accepts an Ω-word if and only if all Δ-words $w = w_1 \ldots w_m$ such that $w_i \in V_i$ for $i = 1, \ldots, m$, are accepted by $\mathcal{A}$.

As mentioned in the previous section, the view-based rewriting Q' represented by automaton $\mathcal{C}$ is evaluated on a view graph $\mathcal{V}$ obtaining $ans(Q', \mathcal{V})$ which is an approximation of $\text{ANS}(Q, \mathcal{V})$.

Example 5. Consider the rewriting Q' represented by the automaton $\mathcal{C}$ in Figure 3 [bottom], and the view graph $\mathcal{V}$ in Figure 2 [left]. It is easy to see that $ans(Q', \mathcal{V}) = \{(a, b), (a, c), (c, b)\}$. □

Assuming that the user query is given by means of a regular expression, [3] showed, using the algorithm above, that the complexity of computing the maximal view-based rewriting is in 2EXPTIME. Moreover, this bound was shown to be tight by constructing a query instance Q, whose rewriting has a doubly exponential size compared to the size of a simple NFA for Q.

5. OUR OPTIMIZATION TECHNIQUES

The above 2EXPTIME bound tells us that to obtain a view-based rewriting is computationally hard except for very small query instances. While the first determinization [for obtaining automaton $\mathcal{A}$] is in practice quite tolerable for typical user queries, the second determinization [for obtaining automaton $\mathcal{C}$ by complementing $\mathcal{B}$] is often prohibitively expensive. However, it is possible to argue that the analysis in [3] is worst-case and hence Algorithm 1 might take only reasonable amount of time on "typical" instances (or on the average). Our experimental results indicate that this is not the case (see Section 6). Experimentally, we were unable to compute automaton $\mathcal{C}$, in reasonable time and space, for about one third of the time while working on "randomly generated" instances.

In this section, we first describe how to optimize the construction of automaton $\mathcal{B}$ in step 2, which is a significant bottleneck when the number of views is considerable. Then, we deal with step 3 of the algorithm, and propose a technique which essentially eliminates this step.

5.1 Computing Automaton $\mathcal{B}$ Efficiently

We present an optimization technique for the step 2 of the above algorithm for computing automaton $\mathcal{B}$. In our experiments we observed that this step, although a polynomial one, is very time consuming if implemented in the straightforward manner.

Taking a closer look at step 2, let s_i and s_j be two arbitrary states in automaton $\mathcal{A}$. Now consider automaton $\mathcal{A}_{ij}$, which is obtained by keeping all the states and transitions in $\mathcal{A}$, but making state s_i and s_j initial and final respectively. All the other states in $\mathcal{A}_{ij}$ are neither initial nor final.

In step 2 of the algorithm, we want to determine whether there should be transition v_x between states s_i and s_j in $\mathcal{B}$. It is easy to see that this is in fact achieved by testing for the emptiness of the intersection $L(\mathcal{A}_{ij}) \cap V_x$. Namely, we insert a transition (s_i, v_x, s_j) in $\mathcal{B}$ iff $L(\mathcal{A}_{ij}) \cap V_x \neq \emptyset$. Recall that the intersection $L(\mathcal{A}_{ij}) \cap V_x$ is obtained by constructing the Cartesian product $\mathcal{A}_{ij} \times \mathcal{A}_{V_x}$, where $\mathcal{A}_{V_x}$ is an automaton for V_x.

However, the automata $\mathcal{A}_{ij}$ for different i's and j's have the same states and transitions [namely those of automaton $\mathcal{A}$]. Only their initial and final states are different. Thus, we construct only one Cartesian product $\mathcal{A} \times \mathcal{A}_{V_x}$ for a given view V_x. Then, we test emptiness on this Cartesian product automaton for $|\mathcal{A}|^2$ different combinations of [one] initial and [one] final states. Although asymptotically there is no gain in doing this, experimentally, we found that for typical queries and views, the speedup achieved by this optimization is often more than 6-fold. This is explained by a better utilization of the CPU cache because there is only one Cartesian product automaton to be constructed and examined.

5.2 Answer Computation Using Automaton $\mathcal{B}$

In this subsection, we describe how to essentially eliminate step 3 of the algorithm of [3].

Recall that the "riskier penalty" in the algorithm of [3] is the computation of automaton $\mathcal{C}$ in step 3 by complementing the automaton $\mathcal{B}$ obtained in step 2. $\mathcal{C}$ might be doubly exponential in the size of the query. Once $\mathcal{C}$ is computed, the final step is to compute $ans(Q', \mathcal{V})$ by constructing the Cartesian product of the automaton $\mathcal{C}$ and a viewgraph $\mathcal{V}$. We ask if it still possible to compute $ans(Q', \mathcal{V})$ directly without first computing the DFA for $L(\overline{\mathcal{B}})$? We achieve this by merging the underlying determinization procedure of step 3 and the subsequent computation of the Cartesian product graph into a single step. We illustrate this using an example.

Example 6. Consider the NFA $\mathcal{B}$ and the viewgraph $\mathcal{V}$ shown in Figure 4 [top–left] and [top–right] respectively. To compute the set of all objects reachable, say from a in $\mathcal{V}$ and following paths spelling words rejected by $\mathcal{B}$, we will build a lazy Cartesian product graph, whose nodes are object–bitvector pairs and edges are labeled with Ω symbols.

We start with the pair $(a, 100)$, where 100 is an abbreviation for $(1, 0, 0)$. This bitvector says that automaton $\mathcal{B}$ is now in state s_0. Next, we construct the pair $(b, 110)$ and put a v_1–edge from $(a, 100)$ to $(b, 110)$. This is because when reading symbol v_1, we hop to object b in $\mathcal{V}$, and in states s_0 and s_1 in $\mathcal{B}$. Continuing in this way, we obtain the Cartesian product graph shown in Figure 4 [bottom].

Building of the above bitvectors is reminiscent of the classical subset construction for converting an NFA into a DFA.

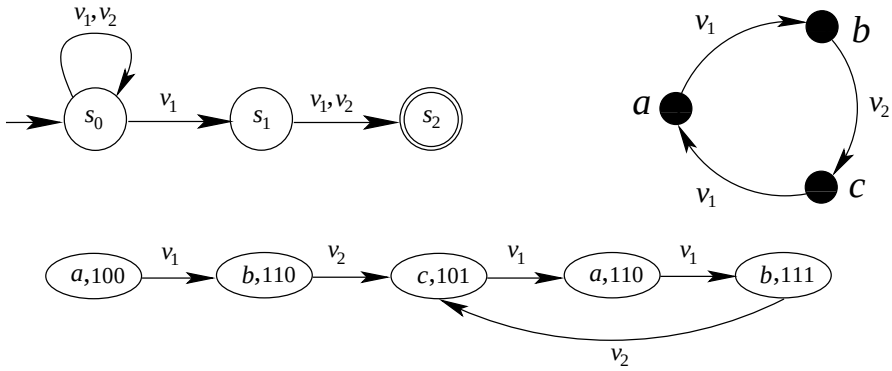


Figure 4: Automaton $\mathcal{B}$ [top–left], viewgraph $\mathcal{V}$ [top–right], and Cartesian product graph [bottom].

In fact, each bitvector corresponds to a state in a DFA for $\mathcal{B}$. However, we only build those bitvectors, which are asked for by the input viewgraph. Thus, observe that in this example only 4 bitvectors are needed, namely 100, 110, 101, and 111. On the other hand, the minimum size DFA corresponding to $\mathcal{B}$ has $2^3 = 8$ states (cf. [12] for this family of automata).

Now, once the Cartesian product graph is constructed, it is easily seen that b is reachable from a using a string not in $L(\mathcal{B})$ but c is not. □

In general, for a $\mathcal{B}$ automaton with set S of states, we use bitvectors of size $|S|$ to keep track of the states that $\mathcal{B}$ can be when reaching some object of the viewgraph. As illustrated by the above example, the nodes of the (lazy) Cartesian product graph are of the form (a, u) where a is an object in the viewgraph and u is a bitvector of size $|S|$. Since the input is a graph as opposed to a string, there can be different bitvectors associated with the same given object (for instance with objects a and b in the example).

We want to stress that we build the Cartesian product graph starting from all the viewgraph objects. In the above example, for clarity we showed the Cartesian product constructed starting from one object only. However, these Cartesian products overlap, and thus, in order to not generate the same object–bitvector pair twice, we maintain a hashtable of the pairs generated so far. In fact, even for a single Cartesian product, the same pair might be needed more than once, and the hashtable is necessary for this case as well in order for the method to terminate.

The edge labels in the Cartesian product graph are of no importance when it comes to generating the query answers. The only thing that matters in this graph is pure reachability. Namely, we produce a pair (a, b) as an answer, if there exists a path [in the Cartesian product graph] from (a, u_0) to (b, w), where u_0 is the initial bitvector $10\ldots0$, and w is a bitvector having no bit set to 1 for any final state in $\mathcal{B}$.

Formally, our algorithm is as follows.

ALGORITHM 2.

Input: Automaton $\mathcal{B}$ and a viewgraph $\mathcal{V}$.

Output: $ans(Q', \mathcal{V})$, where $Q' = L(\overline{\mathcal{B}})$.

Method:

1. Denote by u_0 the bitvector $10\ldots0$ corresponding to the initial state s_0 in $\mathcal{B}$.
2. Initialize
   (a) A processing queue $P = \{(a, u_0) : a \text{ object in } \mathcal{V}\}$.
   (b) A hashtable $H = \emptyset$.
   (c) A Cartesian product graph $\mathcal{G} = \emptyset$.
3. Repeat (a), (b), and (c) until queue P becomes empty.
   (a) Dequeue a pair (a, u) from P.
   Lookup (a, u) in H.
   If (a, u) is not yet in H, then insert it in H and $\mathcal{G}$.
   Otherwise, discard (a, u) as we have already dealt with it.
   (b) For each outgoing edge from a to b in $\mathcal{V}$, labeled by some symbol, say v_{ab}, compute the "next" bitvector w by procedure
   $$w = Next(u, v_{ab}).$$
   [We discuss this procedure soon.]
   (c) If w is different from the all zero's vector, then insert (b, w) in P.
   Also, insert edge $((a, u), v_{ab}, (b, w))$ in $\mathcal{G}$
4. Finally, set
   $ans(Q', \mathcal{V})$ =
   $\{(a, b) :$ there exists a path in $\mathcal{G}$
   from (a, u_0) to (b, w) such that
   w has no bit set to 1 for any final state in $\mathcal{B}\}$.

□

Implementation of $Next(u, v)$.

We optimize the amount of time taken to compute adjacent bitvectors as follows.

Normally, each entry in the transition table of $\mathcal{B}$ is just a list of next possible states of the NFA given the current state and input symbol. Instead of storing this list, we store a bitvector α of $|S|$ bits, that is the characteristic vector of this list of states. Using the various values of α in the transition table, given an object-vector pair of the form (a, u) and an input symbol v, we can compute $Next(u, v)$ in only $O(n)$ time using a sequence of bitwise-OR operations [compared to the naive method of updating vectors that takes $O(n^2)$ in the worst case]. In particular, without loss of generality, suppose the set of indices in u which have a 1 is exactly $\{i_1, i_2 \ldots, i_k\}$. Then it is easy to see that

$$Next(u, v) = \alpha_{i_1,v} \vee \alpha_{i_2,v} \vee \ldots \vee \alpha_{i_k,v}$$

where $\alpha_{i_j,v}$ is the bitvector α in the transition table corresponding to the state q_{i_j} and the input symbol v.

In the next section, we show that our ideas give substantial improvement in running time making it possible to solve the problem of view-based answering on much larger instances compared to the naive implementation.

6. EXPERIMENTAL RESULTS

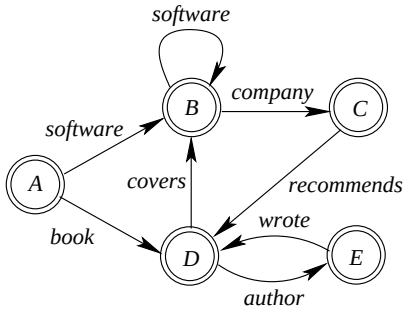
We conducted experiments in order to assess the improvements offered by answering the query using automaton $\mathcal{B}$ over answering using automaton $\mathcal{C}$.

First, we give some details on how we generated queries, views, and viewgraphs. For this we used a simple DataGuide (cf. [1]). DataGuides are essentially finite state automata capturing all the words spelled out by the database paths. In general, DataGuides are compact representations of graph databases. They are small automata presented to the user in order to guide her in writing queries. Each word in a DataGuide could possibly represent many paths that spell that word in a database. For example a DataGuide, capturing databases such as the one shown in Figure 1, contains a word *software·company·recommends*. Certainly, there are many such paths in databases about online stores.

In our experiments, we used the DataGuide given in Figure 5 [top], where all the states are both initial and final.

For generating view language definitions, we obtained a right linear grammar from the DataGuide of Figure 5 [top]. The rules of this grammar are given in the same figure [bottom], where A, B, C, D, and E are the non terminal symbols corresponding to the DataGuide states.

Then, we randomly generated partial derivations using the above grammar. Such a partial derivation is for example $B \rightarrow company \cdot recommends \cdot D$. By randomly selecting such partial derivations, we created new right linear grammars. We kept only those grammars generating non-empty languages. Clearly, the grammars generated in this way capture sublanguages of the DataGuide. By this random procedure, we created 50 test sets of 40 view definitions each.



$$\begin{array}{lcl} A & \rightarrow & software \cdot B \mid book \cdot D \mid \epsilon \\ B & \rightarrow & software \cdot B \mid company \cdot C \mid \epsilon \\ C & \rightarrow & recommends \cdot D \mid \epsilon \\ D & \rightarrow & covers \cdot B \mid author \cdot E \mid \epsilon \\ E & \rightarrow & wrote \cdot D \mid \epsilon \end{array}$$

Figure 5: [Top] DataGuide corresponding to the database in Figure 1. [Bottom] Grammar for the given DataGuide.

For each set of views, we created random queries as follows. Let $\mathbf{V} = \{V_1, \ldots, V_{40}\}$ be a view set. Then, the outer-alphabet is $\Omega = \{v_1, \ldots, v_{40}\}$. First, we randomly created a regular expression on Ω of length not more than 10. For instance such a regular expression could be $re = v_1 \cdot v_{13}^* + v_{40}$. Next, we set $Q = def(re)$, which is a language on Δ, and computed its view-based rewriting using set $\mathbf{V}$ of views.

We could certainly generate queries in a similar fashion as for generating view languages i.e. directly from the DataGuide. However, doing so generates many cases when the rewriting is empty, and the experiments would be uninteresting. On the other hand, generating queries as above guarantees that the rewritings will not be empty.

Regarding the generation of view graphs, we first randomly generated databases from the Data-Guide, and then evaluated on these databases each of the generated views. In this way, we obtained an "answer" for each view. For instance, we could have $\{(a, b), (b, c), \ldots\}$ as the answer for V_1 in some randomly generated database. Then, we inserted edges (a, v_1, b), (b, v_1, c), ... in the the corresponding viewgraph. For each of the 50 sets of views, we randomly generated as above a viewgraph of more than 10,000 nodes.

Then, we computed automaton $\mathcal{B}$ for each set of views, and evaluated it [as described in Section 5] on the corresponding viewgraph. Also, we tried to compute automaton $\mathcal{C}$ accepting $L(\overline{\mathcal{B}})$. We used GRAIL+ (see [17]), which is a well-engineered automata package written in C++. As already mentioned, computing $\mathcal{C}$ was not always possible. Out of our 50 cases, computing $\mathcal{C}$ timed out in 15 of them. We used a big timeout of 4 hours. Whenever we were able to obtain a DFA $\mathcal{C}$, we evaluated it on the corresponding viewgraph. For these cases, we compared the times of evaluating $\mathcal{B}$ versus evaluating $\mathcal{C}$ on the viewgraphs.

In all the test cases, we computed automaton $\mathcal{B}$ using the

ID	B-NFA Size	C-DFA-size	C-DFA-time	C-DFA-V-time	C-DFA-V TTime	B-BitNFA-V-time	B-BitNFA-V-size	Ratio
1	11	35	2	317	319	348	27887	1.1
2	9	16	1	396	397	390	23967	1
3	12	67	7	330	337	396	31875	1.2
4	11	34	3	410	413	417	29003	1
5	9	40	1	407	409	419	26172	1
6	12	74	5	489	494	530	31766	1.1
7	13	57	7	573	580	651	32501	1.1
8	15	83	11	630	641	678	35332	1.1
9	23	462	393	454	847	773	40899	0.9
10	12	69	8	805	813	887	37850	1.1
11	14	114	8	703	711	901	39252	1.3
12	17	166	29	540	569	911	44261	1.6
13	13	72	9	905	914	1037	38095	1.1
14	13	221	19	642	661	1159	50413	1.8
15	16	513	87	609	696	1180	48698	1.7
16	20	319	153	743	896	1247	47582	1.4
17	12	82	8	1067	1074	1457	47051	1.4
18	35	1442	2148	824	2972	1505	52686	0.5
19	33	3316	4126	592	4718	1593	61860	0.3
20	16	266	61	1058	1119	1867	56361	1.7
21	21	552	296	859	1154	1879	58879	1.6
22	35	723	710	1106	1816	2074	55939	1.1
23	31	831	461	913	1374	2113	61329	1.5
24	21	1316	526	867	1392	2121	66651	1.5
25	20	1098	379	1046	1425	2206	65004	1.5
26	18	238	60	1372	1432	2846	63561	2
27	18	523	104	1056	1160	3061	74273	2.6
28	20	550	177	1083	1260	3403	83515	2.7
29	26	2001	855	1245	2099	3512	80085	1.7
30	33	3578	2197	1106	3303	3599	83338	1.1
31	38	3492	2937	1628	4565	3666	76546	0.8
32	35	1720	1210	959	2169	3674	84318	1.7
33	28	2894	2515	1330	3845	4477	102625	1.2
								1.3
34	49	N/P	N/P	N/A	N/A	697	103251	
35	42	N/P	N/P	N/A	N/A	820	101291	
36	44	N/P	N/P	N/A	N/A	892	92852	
37	53	N/P	N/P	N/A	N/A	1224	50903	
38	41	N/P	N/P	N/A	N/A	1554	56048	
39	52	N/P	N/P	N/A	N/A	1754	44805	
40	48	N/P	N/P	N/A	N/A	2033	66406	
41	53	N/P	N/P	N/A	N/A	2239	66052	
42	43	N/P	N/P	N/A	N/A	2270	76941	
43	42	N/P	N/P	N/A	N/A	2549	85026	
44	48	N/P	N/P	N/A	N/A	3358	80330	
45	44	N/P	N/P	N/A	N/A	3468	83515	
46	30	N/P	N/P	N/A	N/A	3542	86632	
47	42	N/P	N/P	N/A	N/A	3816	81133	
48	40	N/P	N/P	N/A	N/A	3890	84563	
49	45	N/P	N/P	N/A	N/A	4872	103183	
50	47	N/P	N/P	N/A	N/A	5985	123831	

Figure 6: Table of results.

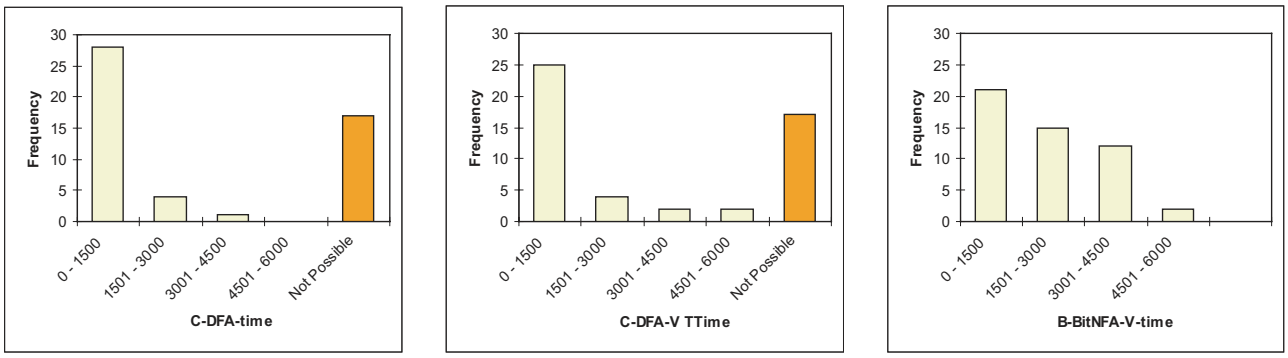


Figure 7: Number of instances for different time ranges of C-DFA-V-time, C-DFA-V-TTime, and B-BitNFA-V-time. The darker bars for C-DFA-V-time and C-DFA-V-TTime show the fraction of instances for which the computation of automaton $\mathcal{C}$ was not possible.

technique described in Subsection 5.1. It was this technique that made possible the computation of $\mathcal{B}$ in a reasonable amount of time for each test case (of 40 views each). As mentioned in Subsection 5.1, using our technique we were able to achieve a speedup of more that six-fold in computing $\mathcal{B}$. Our times for computing the $\mathcal{B}$ automata range between 10 to 15 minutes.

We have tabulated our time and size results in Figure 6. The results were obtained using a modern Sun-Blade-1000 machine with 1GB of RAM. In the following, we describe the column headers of our result table.

ID: ID of test set.

B-NFA-size: Size of automaton (NFA) $\mathcal{B}$.

C-DFA-size: Size of automaton (DFA) $\mathcal{C}$.

C-DFA-time: Time (in secs) to compute automaton (DFA) $\mathcal{C}$.

C-DFA-V-time: Time (in secs) to evaluate automaton (DFA) $\mathcal{C}$ on the corresponding viewgraph.

C-DFA-V-TTime: Total time (in secs) to compute and then evaluate automaton (DFA) $\mathcal{C}$ on the corresponding viewgraph. [This is the sum of the above two times.]

B-BitNFA-V-time: Time (in secs) to bitwise evaluate automaton (NFA) $\mathcal{B}$ on the corresponding viewgraph.

B-BitNFA-V-size: Size of the input-aware Cartesian product of automaton (NFA) $\mathcal{B}$ with the corresponding viewgraph.

Ratio: Ratio of the time to obtain the answers using bitwise evaluation of automaton (NFA) $\mathcal{B}$ to the time to obtain the answers using automaton (DFA) $\mathcal{C}$ whenever possible. The last number of 1.3 in this column is the average of the column.

We have sorted the results in ascending order of the **B-BitNFA-V-time**. The first part of the table contains the results for the cases when the computation of automaton $\mathcal{C}$ succeeded. The second part of the table contains the results for the cases when the computation of automaton $\mathcal{C}$ failed. As such, the second part of the table has results which relate to the use of automaton $\mathcal{B}$ only. The shaded area of this part of the table is marked by N/P (Not Possible) or (N/A) (Not Applicable) as appropriate.

Also, we have graphed columns **C-DFA-V-time**, **C-DFA-V-TTime**, and **B-BitNFA-V-time** in Figure 7 in order to more clearly show the fractions of instances corresponding to different time ranges, as well as the fraction of instances for which the computation of rewritings is impossible in reasonable time and space.

Based on the table of results, we are able to draw the following natural conclusions.

1. Computing in full the view-based rewriting represented by automaton $\mathcal{C}$ is hard and fails in a considerable number of cases (30% of them). Hence, one should not pursue this route for producing view-based query answers.
2. Even when constructing $\mathcal{C}$ is possible, the performance advantage offered by the determinism of $\mathcal{C}$ over automaton $\mathcal{B}$ is small. In average, bitwise evaluation of $\mathcal{B}$ is only 1.3 times slower on the average, while in some cases it can be even faster. This is due to the smaller footprint of $\mathcal{B}$ having so a better hardware cache utilization.
3. For all the test cases, the size of the input-aware bitwise Cartesian product of automaton $\mathcal{B}$ with the corresponding viewgraph $\mathcal{V}$ is very far from the worst case of $2^{|\mathcal{B}|} \cdot |\mathcal{V}|$.

From all the above, one can see that by employing our techniques, the view-based answering of RPQ's becomes (fairly) feasible in practice.

7. CONCLUSIONS

In this paper, we examined the well-known problem of answering regular path queries (RPQ) using views in LAV information integration. This problem is particularly important because RPQ's are part of virtually all the languages

for semistructured data, which are very prevalent in information integration.

This paper makes two useful contributions towards a better understanding of this important problem. Firstly, it shows experimental evidence that the problem, known to have a worst-case lower bound of 2EXPTIME, also takes lot of time to be solved on the average. Secondly, it proposes automata-theoretic techniques, which make it fairly feasible to obtain the answer for large query instances. In particular, we would like to emphasize the usefulness of the "answering through rewriting complement" that we have used in this paper. We hope that this paper will lead to further study of this very important problem.

8. REFERENCES

[1] Abiteboul S., P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers. San Francisco, CA., 1999.

[2] Bravo L., and L. Bertossi. Disjunctive Deductive Databases for Computing Certain and Consistent Answers to Queries from Mediated Data Integration Systems. *Journal of Applied Logic* 3 (1): 329–367, 2005.

[3] Calvanese D., G. Giacomo, M. Lenzerini and M. Y. Vardi. Rewriting of Regular Expressions and Regular Path Queries. *J. Comput. Syst. Sci.* 64 (3): 443–465, 2002.

[4] Calvanese D., G. Giacomo, M. Lenzerini and M. Y. Vardi. Answering Regular Path Queries Using Views. *Proc. of ICDE '00.*

[5] Calvanese D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. View-Based Query Processing and Constraint Satisfaction. *Proc. of LICS '00.*

[6] Calvanese D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. View-based query processing: On the relationship between rewriting, answering and losslessness. *Theor. Comput. Sci.*, 371 (3): 169–182, 2007.

[7] Consens M. P, A. O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. *Proc of PODS'90.*

[8] Grahne G., and A. O. Mendelzon Tableau Techniques for Querying Information Sources through Global Schemas. *Proc. ICDT '99.*

[9] Grahne G., and A. Thomo. An Optimization Technique for Answering Regular Path Queries. *Proc. of WebDB '00.*

[10] Grahne G., and A. Thomo. Algebraic Rewritings for Optimizing Regular Path Queries. *Proc. ICDT '01.*

[11] Grahne G., A. Thomo, and W. Wadge. Preferentially Annotated Regular Path Queries. *Proc. of ICDT'07.*

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

[13] Lenzerini M. Data Integration: A Theoretical Perspective. *Proc. of PODS'02.*

[14] Levy A. Y., Mendelzon A. O., Sagiv Y., Srivastava D. Answering Queries Using Views. *Proc. PODS '95.*

[15] Mendelzon A. O., and P. T. Wood, Finding Regular Simple Paths in Graph Databases. *SIAM J. Comp.* 24 (6): 1235–1258, 1995.

[16] Mendelzon A. O. G. A. Mihaila and T. Milo. Querying the World Wide Web. *Int. J. Dig. Lib.* 1 (1): 57–67, 1997.

[17] Raymond R. D., and D. Wood. Grail: A C++ Library for Automata and Expressions. *J. Symb. Comput.* 17 (4): 341–350, 1994.

[18] Ullman J. D. Information Integration Using Logical Views. *Proc. ICDT '97.*

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