Optimizing Regular Path Expressions Using Graph Schemas (full, revised version)

Mary Fernández
mff@research.att.com
AT&T Labs
Florham Park, NJ 07932

Dan Suciu
suciu@research.att.com
AT&T Labs
Florham Park, NJ 07932

Abstract

Query languages for data with irregular structure use *regular path expressions* for navigation. This feature is useful for querying data where parts of the structure is either unknown, unavailable to the user, or changes frequently. Naive execution of regular path expressions is inefficient, however, because it ignores any structure in the data. We describe two optimization techniques for queries with regular path expressions. Both rely on *graph schemas* for specifying partial knowledge about the data's structure. *Query pruning* uses this structure to restrict navigation to only a fragment of the data; we give an efficient algorithm for rewriting any regular path expression query into a pruned one. *Query rewriting using state extents* can eliminate or reduce navigation altogether; it is reminiscent of optimizing relational queries using indices. There may be several ways to optimize a query using state extents; we give a polynomial-space algorithm that finds *all* such optimizations. For restricted forms of regular path expressions, the algorithm is provably efficient. We also give an efficient approximation algorithm that works on all regular path expressions.

1 Introduction

Recent research has focused on data with irregular, unknown, or frequently changing structure; such data is called *semistructured data* [Abi97]. The data is modeled as a labeled graph, in which the nodes correspond to the objects and the edges to their attributes. Most query languages proposed for semistructured data can navigate the data using *regular path expressions*, thus traversing arbitrary long paths in the graph.

Applications of semistructured data are diverse. TSIMMIS [PGMW95, PGMU95, PAGM96], is a system for data integration, which allows the integration of data residing in a variety of data formats. Representing data uniformly in a graph model simplifies coping with different data formats, varying types, and missing data. Lore [QRS+95] is a general purpose data repository for semistructured data; its query language, Lorel, supports regular path expressions. Biological databases are another application of semistructured data; their structure is often irregular, deeply-nested, and rapidly changing, which makes

it difficult to store in a relational form. ACeDB [TMD92] uses a tree data model to store biological data; the query language UnQL [BDHS96] was designed in part for such tree-like data. Web sites can also be viewed as semistructured data [AV97a]. WebSQL [MMM96], W3QS [KS95], and WebLog [LSS96] were designed to query the Web's graph structure. STRUDEL [FFK+97, FFLS97] applies semistructured data to the problem of managing Web sites.

Regular path expressions are common in semistructured data systems. They allow the user to navigate through arbitrary long paths in the data. Naive evaluation of regular path expressions is costly, however, because it ignores any structure that might be present in the data. We show that even partial knowledge about such structure can be exploited to improve evaluation. To do this, we first describe the data's partial structure using recently introduced *graph schema* [BDFS97]. Graph schemas are flexible; at one extreme, they can leave the structure unspecified, or at the other extreme, they can strictly enforce the names and types of the attributes of each object. Then, we describe two novel optimization techniques, *query pruning* and *query rewriting using state extents*, which exploit the structure specified by a graph schema. Query pruning rewrites a given regular path expression into another query that searches only a portion of the data. The second technique rewrites the query into one that traverses data by starting from entry points deeper in the graph, instead of from the root, and that can sometimes avoid all navigation. This technique relies on a solution to an instance of the recursive query rewriting problem [LMSS95]. While the general case for recursive queries is undecidable, our work shows that this problem is decidable for a particular case.

1.1 The Problem

We illustrate the problem on a hypothetical Web site of a research organization, shown in Fig. 1. We view a Web site as an instance of semistructured data and model it as an edge-labeled graph. Nodes denote HTML pages and edges denote hyperlinks. Suppose we want to retrieve project pages that are accessible from some department page and exclude all other project pages. We express this as:

Query 1.1 select p where $*.Dept.*.Project.p$ in DB

The where clause contains a *regular path expression*; the first "$*$" denotes any path from the root of the database DB up to a *Dept* link, and the second "$*$" denotes any path up to a *Project* link. *Dept*(l) and *Project*(l) are user-defined predicates that determine whether a link l points to a department or a project page. For the Web site fragment in Fig. 1, the query returns the two pages r_1, r_2; the page under the "Safe Language Project" link is excluded, because it is not reachable from a department.

We want to identify efficient evaluation techniques for queries with regular path expressions. We emphasize that a regular path expression refers to a graph's *logical structure* (i.e., link structure) and not to its *physical structure* (URL names). Current techniques suggest two plans for this query:

Plan 1: Exhaustive search. Start at the graph's root, search exhaustively for *Dept* links, and from there search exhaustively for a *Project* link. This strategy requires traversal of the entire database.

Plan 2: Use predicate extents. Assume that the extent of the predicate *Dept*, i.e., the set of nodes that satisfy *Dept*, is known. For each page x in *Dept*'s extent, exhaustively

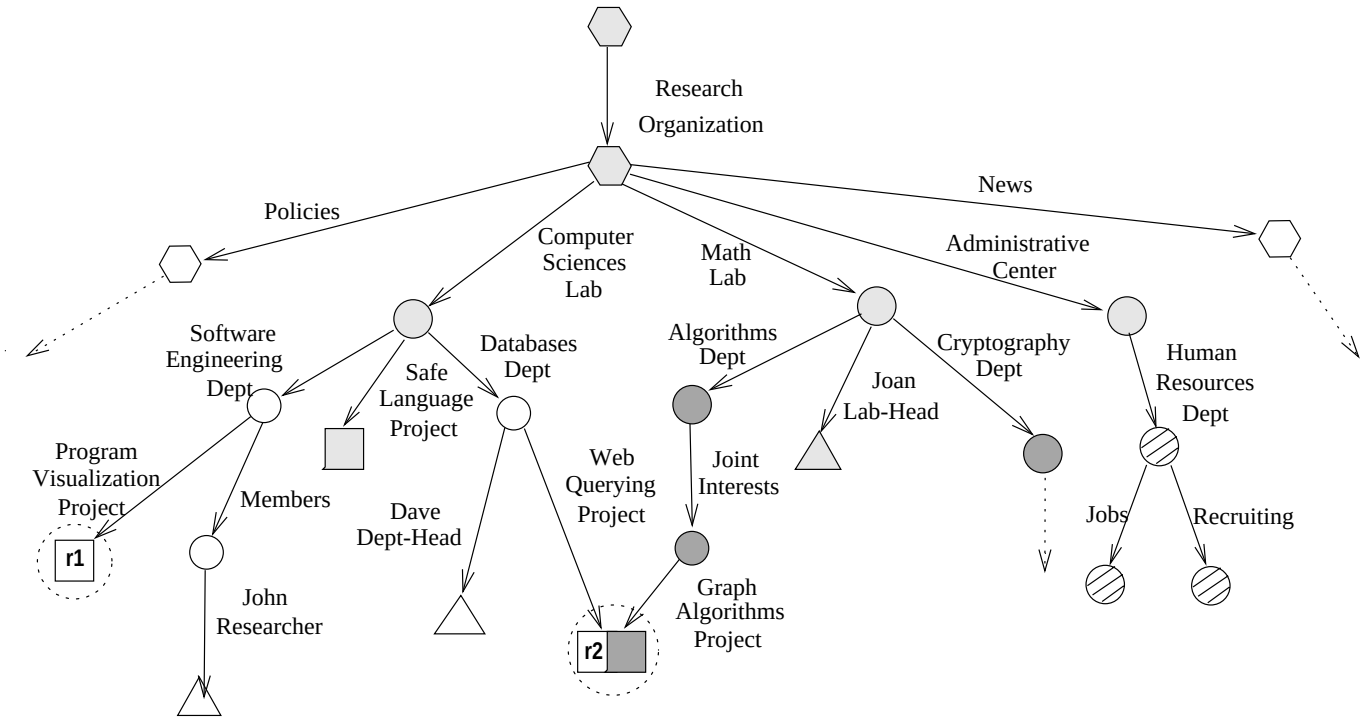

Figure 1: Example Web site of research laboratory.

search the pages accessible from x for a *Project* link. This plan requires that *Dept*'s extent be precomputed and maintained. Although better than Plan 1, this plan is still inefficient, because it cannot exploit multiple extents in more complex regular path expressions. For example, the plan cannot use *Project*'s extent, because for a given project p, it cannot determine whether p is reachable from a *Dept* edge. We need more powerful techniques to further restrict the search.

1.2 Our Contribution

Our optimization techniques rely on graph schemas [BDFS97], which describe partial knowledge of a graph's structure. Fig. 2 (a) contains a graph schema for our Web site. Edges are labeled with basic predicates or boolean combinations thereof. The $Person(l)$, $Project(l)$, and $Lab\text{-}or\text{-}Center(l)$ predicates are self-explanatory. $Other(l)$ denotes the negated disjunction of all the other predicates occurring in the schema, i.e. $\neg(Person(l) \vee Project(l) \vee \ldots)$. Intuitively, this schema specifies that the database contains some unclassified edges (the *Other* loop at the root) followed by either a *General-Info* edge or a *Lab-or-Center* edge, or both. After a *Lab-or-Center* edge, the database may contain more unclassified edges followed by either a *Software-Dept*, *Person*, or *Project* edge, etc. Graph schemas allow any edges to be missing in the source database. They only restrict what kind of edges may occur and where, e.g., the example schema guarantees that no *Software-Dept* edge follows *General-Info*. The nodes in graph schemas, called *states*, implicitly classify the pages of a database, e.g., in Figures 1 and 2, the geometric shapes show the correspondence between nodes in the source graph and the schema states. Now we can describe two improved plans.

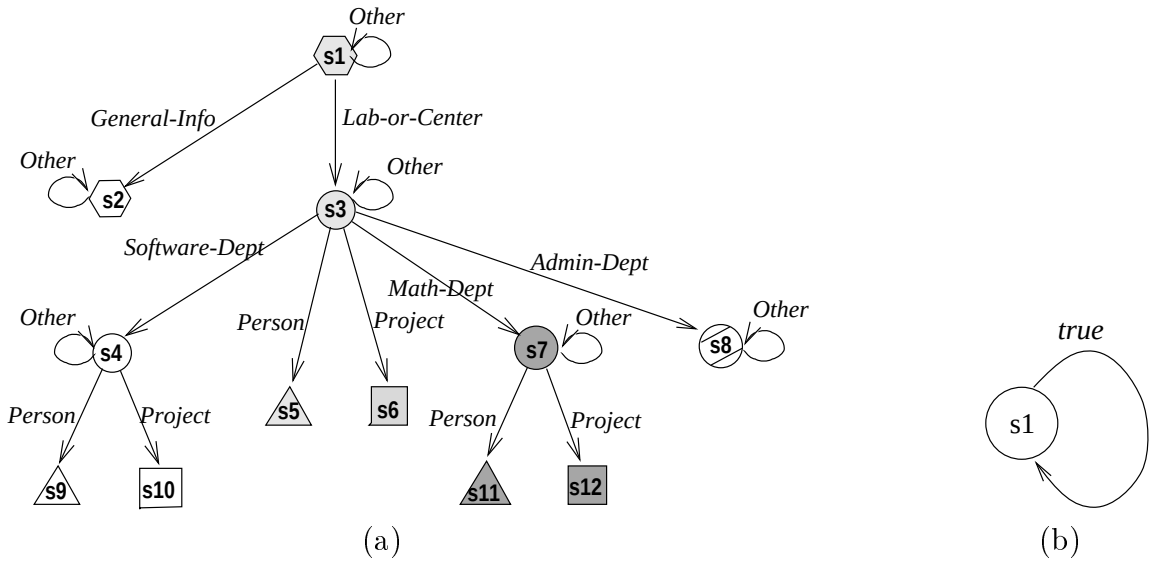

Figure 2: (a) Graph schema for research lab. Here $Other = \neg(Person \vee Project \vee \ldots)$. (b) Universal graph schema.

Plan 3: Prune the query. This plan refines Plan 1 by pruning the search at certain sub-graphs. Recall that the first stage of the search looks for a *Dept* link. We assume that $Dept(l) = Software\text{-}Dept(l) \vee Math\text{-}Dept(l) \vee Admin\text{-}Dept(l)$. Therefore, when a label satisfies $Other(l)$, it cannot satisfy $Dept(l)$, hence, using the schema, we can ignore every sub-graph under an edge satisfying *General-Info*, because they contain no *Dept* links. Referring to Fig. 1, the pruned query will not traverse the sub-graphs under *Policies* and *News*. The second stage searches for *Project* links, and similarly, we can ignore any sub-graph under an *Admin-Dept* edge, because there are no projects there. Query pruning is a flexible technique. We can also apply it to Plan 2: start the search from pages in *Dept*'s extent *and* prune the search.

Plan 4: Answer the query directly. Sometimes we can avoid all graph traversal by rewriting the query using *state extents*; we call the graph schema's nodes *states*. A *state extent* is the set of all database nodes that "belong" to that state. State extents refine predicate extents, e.g., the extent of the *Project* predicate in Fig. 2(a) is $ext(s_6) \cup ext(s_{10}) \cup ext(s_{12})$. By inspecting both the schema and the query, we see that the result of the query is the set $ext(s_{10}) \cup ext(s_{12})$; pages in $ext(s_6)$ are excluded, because they are not reachable from a *Dept* edge. This plan avoids all graph search and answers the query directly. In general, we may not be able to avoid all graph search, but can reduce it substantially by starting the search deeper in the graph.

2 Background

Semistructured data is best modeled by edge-labeled graphs [Abi97]. We define a *graph database*, DB, to be a *rooted, edge-labeled graph*, with labels from an infinite universe *Label*. We assume that each node is accessible from the root. Fig. 1 depicts a graph database. In general, DB may contain shared nodes and cycles.

Query Language We describe our optimization techniques in terms of a query language with regular path expressions and joins. A *regular query* Q is described by the grammar:

$$Q ::= (\textsf{select } Q \textsf{ where } C, \ldots, C) \mid var$$
$$C ::= R.var \textsf{ in } var$$
$$R ::= Pred \mid \epsilon \mid (R.R) \mid (R|R) \mid R*$$

All variables denote nodes in the data graph; DB is the variable denoting the root. A *condition* (or *conjunct*) C of the form "$R.x$ in y" means that there exists a path from node y to node x whose labels match the regular path expression R. Note that regular path expressions permit predicates on edges. The predicate *true* denotes any label, and *true* $*$ any sequence of labels; we abbreviate the latter with $*$. We call queries with a single condition (select y where $R.y$ in x) *single regular queries*. Multiple conditions can introduce joins. For example,

$$\textsf{select } p \textsf{ where } *.Project.p \textsf{ in } DB, *.Dept.*.p \textsf{ in } DB$$

searches for all pages pages p that are independently accessible through a path containing a *Project* link and one containing a *Dept* link. Finally, we may have *subqueries*, e.g., select (select y where $R.y$ in x) where $R'.x$ in DB. Here, the subquery is $Q(x) =$ select y where $R.y$ in x. Although any query is equivalent to one without subqueries[1], we isolate subqueries so they can be optimized separately.

The meaning of a regular query $Q =$ select Q' where ... applied to a database DB is a set of nodes in DB obtained as follows. Consider all substitutions of Q's variables with nodes from DB that satisfy the where conditions, evaluate Q' to obtain a set of nodes, and take the union of all such sets.

Graph Schemas Our optimization techniques exploit partial knowledge about the structure specified by a *graph schema* [BDFS97]. A graph schema is a graph S with n nodes $s_1, \ldots, s_n$ called *states*, a designated *root* state s_1, and edges labeled with unary predicates over *Label*. Fig. 2(a) represents a schema with 12 states. A schema is called *deterministic* if for every state s and any two distinct outgoing edges labeled P and P', P, P' are disjoint, i.e., not $(\exists l.P(l) \wedge P'(l))$. Unless otherwise specified, we will assume throughout the paper that all schemas are *deterministic*.

Formally, a database DB *conforms* to a schema S if there exists a *simulation* from DB to S. The latter is a binary relation $\sim$ between nodes in DB and states in S s.t. (1) $r \sim s_1$, where r, s_1 are the roots in DB, S, and (2) whenever $u \sim s_i$ and $u \xrightarrow{l} v$ is an edge in DB, then there exists an edge $s_i \xrightarrow{P} s_j$ in S s.t. $P(l)$ is true and $v \sim s_j$. If DB conforms to S and S is deterministic, there always exists a minimal simulation [BDFS97]; in this paper, we always refer to this minimal simulation. For any state s in S, we define its *extent*, $ext(s)$, to be the set of all nodes u in DB s.t. $u \sim s$. For example, the database DB in Fig. 1 conforms to the schema S in Fig. 2(a). The simulation relation is suggested graphically; a node in DB belongs a state's extent iff they have the same shape. Thus, *s10*'s extent is $\{r1, r2\}$. If DB is not a tree, some nodes may belong to more than one extent, e.g., $r2 \in ext(s10) \cap ext(s12)$. Graph schemas do not fully describe a graph's structure, e.g., Fig. 2(a) does not specify the

[1] This one is equivalent to select y where $R'.x$ in $DB, R.y$ in x, which is also equivalent to select y where $R'.R.y$ in DB.

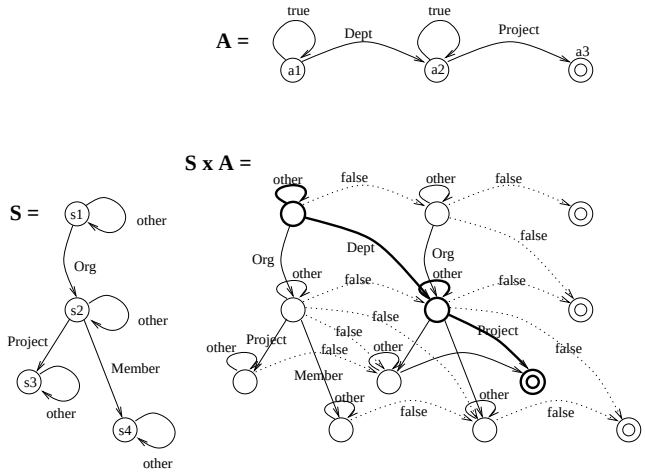

Figure 3: Schema S, automaton A, and product automaton $S \times A$. The pruned automaton $S \sqcap A$ is in bold. Here $other \stackrel{\text{def}}{=} \mathsf{not}\ (Org \vee Project \vee Member)$

graph structure under a *General-Info* edge. At one extreme, the graph schema S in Fig. 2(b) does not impose any constraints on the database, i.e., *any* database DB conforms to S.

3 Query Pruning

Query pruning takes an input query Q and a schema S and rewrites the query into a new query, Q^S. The naive evaluation of Q traverses a large fragment of the database graph, possibly the entire graph, starting at the root. The naive evaluation of Q^S starts at the root and traverses a smaller fragment of the graph by skipping parts of the database which, based on S, are guaranteed not to contribute to the result. In general, Q and Q^S are not equivalent; they are equivalent, however, on databases that conform to the schema S.

Query pruning applies to arbitrary regular queries. We describe the technique for single regular queries first, then show how to generalize it.

Consider the query $Q = \mathsf{select}\ y\ \mathsf{where}\ R.y\ \mathsf{in}\ DB$. We start by constructing a nondeterministic automaton A, equivalent to R, i.e., we do not insist on determinizing A, hence its size is proportional to R's size. A has p states, $a_1, \ldots, a_p$, and its transitions are labeled with predicates. Given a schema S with states $s_1, \ldots, s_n$, we construct the product automaton $S \times A$, with $n \times p$ states, $(s_1, a_1), (s_1, a_2), \ldots, (s_n, a_p)$. Its transitions are $(s, a) \xrightarrow{P \wedge Q} (s', a')$ for any edge $s \xrightarrow{P} s'$ in S and any transition $a \xrightarrow{Q} a'$ in A. Here $P \wedge Q$ is the logical conjunction of the predicates P and Q. When $P \wedge Q \equiv \mathit{false}$, then that transition is vacuous, and we can discard that edge. The initial states in $S \times A$ are all states of the form (s, a) with s the root in S and a an initial state in A; similarly, the terminal states are of the form (s, a) with a a terminal state. Finally we "prune" the automaton $S \times A$ i.e., remove all states and transitions except those that are on a path from the initial state to some terminal state. In doing so, we ignore the edges labeled false, which is equivalent to discarding them immediately. We denote the pruned automaton by $S \sqcap A$. Finally, the rewritten, pruned query Q^S is $\mathsf{select}\ y\ \mathsf{where}\ R^S.y\ \mathsf{in}\ DB$, where R^S is a regular expression equivalent to $S \sqcap A$.

To illustrate, recall Query 1.1, with automaton A appearing in Fig. 3. A naive evaluation of this query will start at the root, search for a *Dept* edge, and from there search for a *Project* edge. Suppose that DB conforms to schema S in Fig. 3. The schema specifies that (1) there is at most one *Org*(anization) on any path from the root, and (2) on any path from the root there is either a *Project* edge or a *Member* edge, but not both. We assume that each *Dept* is an *Org*(anization), i.e. $Dept \subseteq Org$. Using this information, we can improve the query's evaluation by (1) ending the search when we encounter some other *Org* edge which is not a *Dept* edge, and (2) ending the search when we encounter some *Member* edge. Figure 3 illustrates the construction of $S \times A$ (with 12 states), then of $S \sqcap A$ (the 3 bold states). The dotted transitions are labeled *false*, e.g., the topmost, left one is $other \wedge Dept \equiv false$. Hence, the pruned automaton consists of only three states (in bold), and the pruned query Q^S is:

select p where $(other*).Dept.(other*).Project.p$ in DB

Now we describe how to prune an arbitrary regular query, which has k conjuncts, $R_1.y_1$ in $x_1, \ldots, R_k.y_k$ in x_k. First, we construct the k product automata $S \times A_1, \ldots, S \times A_k$; to distinguish their states, we denote the state (s_i, a) in automaton $S \times A_k$ by (s_i, a, A_k). Next, we construct an AND/OR graph (also called *alternating graph*), G. We give the definition of an AND/OR graph below, with a minor deviation from the standard one [GHR95, Imm87]:

Definition 3.1 *An* AND/OR graph $G = (V, E)$ *is a directed graph whose nodes are partitioned into two sets:* $V = V_1 \cup V_2$ *called OR nodes and AND nodes. An* accessibility property *on* G *is a set of nodes,* $A \subseteq V$*, s.t.:*

1. *If* v *is an OR node and* $v \in A$*, then there exists some predecessor in* A $(\exists u.(u, v) \in E \wedge u \in A)$*.*
2. *If* v *is an AND node and* $v \in A$*, then all its predecessors are in* A $(\forall u.(u, v) \in E \Longrightarrow u \in A)$*.*

A node u *in* G *is* accessible *if there exists an accessiblity property* A *s.t.* $u \in A$*.*

An OR node without predecessor s is not accessible. An AND node without predecessors is accessible. Consider a rooted graph G: we can transform it into an AND/OR graph by declaring its root to be an AND node and all other nodes to be OR nodes. Then a node u in G is accessible according to Definition 3.1 iff there exists a path from the root to u: hence the definition is a natural extension of the traditional accessibility notion. In general, there can be many accessiblity properties A: there always exists a maximal one, and a node u is accessible iff it belongs to the maximal accessibility property. Here our definition deviates slightly from the standard one [Imm87], where a minimal accessibility relation is considered[2] Given an AND/OR graph G, one can easily compute the accessible nodes in PTIME as follows:

1. Start by marking all nodes as being accessible. Then repeat the following steps until no more changes.
2. If there exists some OR node u which does not have at least one predecessor marked accessible, then mark u "unacessible".
3. If there exists some AND node u which has at least one predecessor which is not accessible, then mark u "unaccessible".

[2]Our definition is the dual of [Imm87], obtained by switching AND and OR nodes, and "accessible" with "unaccessible".

We return now to our query Q with k conjunts, and describe the construction of the AND/OR graph G. The intuition behind G is the following. Consider some condition $R_l.z$ in x_l in isolation: z may be bound to some node in $ext(s_i)$ iff (s_i, a) is in $S \sqcap A_l$ for at least *some* terminal state a in A_l. This is an OR condition. Now consider all conjuncts on z, $R_{l_1}.z$ in $x_{l_1}, \ldots R_{l_m}.z$ in x_{l_m},: z may be bound to some node in $ext(s_i)$ iff it can be bound in *each* of these conjuncts. This is an AND condition. Formally the graph is obtained as follows: (1) take the disjoint union of $S \times A_1, \ldots, S \times A_k$, where all nodes are OR-nodes, (2) for every conjunct $R_l.z$ in x_l, and every state s_i in S, create an OR-node (s_i, z, A_l) and add edges $(s_i, a, A_l) \rightarrow (s_i, z, A_l)$, for every terminal state a in A_l, and (3) for every variable z and state s_i in S, create an AND-node (s_i, z) and add edges $(s_i, z, A_l) \rightarrow (s_i, z)$, for every conjunct $R_l.z$ in x_l, and edges $(s_i, z) \rightarrow (s_i, a, A_l)$, for every conjunct $R_l.y_l$ in z, with a the initial state in A_l. Note that the node (s_1, DB), where s_1 is the root state in S, will be accessible, since the query has no condition of the form $R.DB$ in ... We drop all other nodes $(s_2, DB), (s_3, DB), \ldots$ Finally, we compute the accessible nodes of the AND/OR graph. We define $S \sqcap A_l$ to consist of those states and transitions in $S \times A_l$ that are on a path between an accessible initial state and an accessible terminal state, and we define R_l^S to be the equivalent regular expression. The pruned query has all conjuncts replaced by $R_1^S.y_1$ in $x_1, \ldots, R_k^S.y_k$ in x_k. Summarizing:

Proposition 3.2 *Let Q be a regular query and S a schema. One can compute the* pruned *query Q^S in* PTIME *in the size of Q and S.*

Example 3.3 Consider a query with four conjuncts:

$$Q(DB) = \text{select } z$$
$$\text{where } R_1.x \text{ in } DB, R_2.y \text{ in } x, R_3.x \text{ in } y, R_4.z \text{ in } x$$

and assume that the schema S has three states, s_1, s_2, s_3, with s_1 the root. The AND/OR graph G is depicted in Figure 4. The product automata $S \times A_1, \ldots, S \times A_4$ are not shown, but suggested by boxes. Each A_i will have three input states (suggested as being at the top of the box), of the form $(s_1, a), (s_2, a), (s_3, a)$, where a is the input state in A_i. They may have arbitrarily many output states, represented at the bottom. To avoid clutter, we split the nodes $(s_1, x), (s_2, x), \ldots, (s_2, z), (s_3, z)$ and represented them both at the top and at the bottom of the figure. The important part of this figure are the three AND nodes $(s_1, x), (s_2, x), (s_3, x)$ at the bottom. Their meaning is "(s_i, x) is accessible if x may be in $ext(s_i)$". Since there are two conditions on x, ($R_1.x$ in DB and $R_3.x$ in y), each is an AND node with two predecessors.

Example 3.4 Continuing the previous example, we consider particular instances for R_1, R_2, R_3, R_4 and a particular schema S. Namely take the query to be:

$$Q = \text{select } z$$
$$\text{where } (a*|a*.b.a*).x \text{ in } DB,$$
$$c.y \text{ in } x,$$
$$c.x \text{ in } y,$$
$$a.z \text{ in } x$$

(That is $R_1 = (a*|a*.b.a*), R_2 = c, R_3 = c, R_4 = a$.) Let S be given by Figure 5. Then, the AND/OR graph basically discovers that $x \in ext(s_1), y \in ext(s_3), z \in ext(s_1)$. We explain next

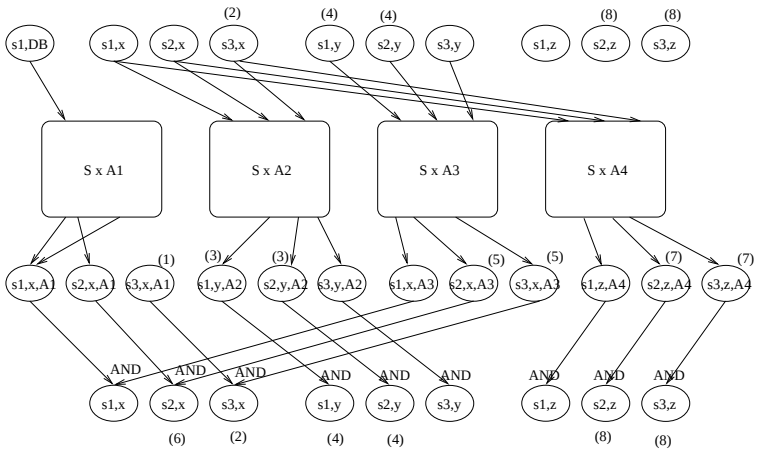

Figure 4: AND/OR graph for the query in Example 3.3. The nodes at the bottom are the same as those on the top: they are separated only to avoid clutter. The numbers in parentheses refer to the steps in Example 3.4 when that node is discovered to be unaccessible.

how this is derived, by computing the accessible nodes. We do not expand the automata, in order to avoid clutter. We start by marking all nodes in Figure 4 "accessible". Then we perform the steps below, which are chosen in random order, and which mark certain nodes as "unaccessible": the step number of discovering each "unaccessible" node is marked in Figure 4.

1. Considering the condition $(a*|a*.b.a*).x$ in DB, we derive that (s_3, x, A_1) is unaccessible (formally, this is done by observing that the terminal states of the form $(s_3, \_)$ are not reachable in A_1 from its initial state of the form $(s_1, \_)$).
2. Hence (s_3, x) is "unaccessible".
3. Moving to A_2, we discover that $(s_1, y, A_2), (s_2, y, A_2)$ are "unaccessible".
4. Hence $(s_1, y), (s_2, y)$ are "unaccessible".
5. Moving to A_3, we discover now that (s_2, x, A_3) and (s_3, x, A_3) are unacessible.
6. Now (s_2, x) becomes "unaccessible", because it is an AND node.
7. In A_4 we discover that (s_2, z, A_4) and (s_3, z, A_4) are "unaccessible".
8. Finally, (s_2, z), (s_3, z) are "unaccessible".
9. No other "unaccessible" nodes are found.

Hence, the pruned query is:

$$
\begin{aligned}
Q^S = \ & \mathsf{select}\ z \\
& \quad \mathsf{where}\ a*.x\ \mathsf{in}\ DB, \\
& \qquad c.y\ \mathsf{in}\ x, \\
& \qquad c.x\ \mathsf{in}\ y, \\
& \qquad a.z\ \mathsf{in}\ x
\end{aligned}
$$

Note that the set A of nodes which remain accessible in Fig. 5 is a maximal accessibility property. A minimal one would not work (even after changing slightly the definition: in our definition the minimal one is $\emptyset$), because the accessible nodes (s_1, x) and (s_3, y) depend on each other in a cyclic way.

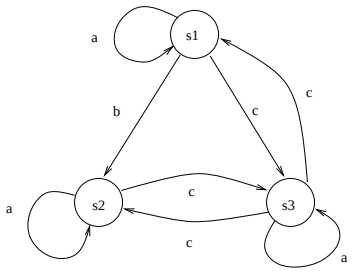

Figure 5: Schema for Example 3.4.

We prove now that pruning is sound. Given a schema S, two regular queries Q and Q' are *equivalent* over S, iff $Q(DB) = Q'(DB)$ for any database DB conforming to S. They are *equivalent* iff $Q(DB) = Q'(DB)$ for any DB. We have to prove that Q and Q^S are equivalent over S, for every regular query Q and schema S. We prove a more general statement for the case of single regular queries, which we need in Section 4. We need to develop some technical tools for that.

First, notice that every database DB can be viewed as a nondeterministic schema, by considering each label constant a in DB as the predicate $P_a(l) \stackrel{\text{def}}{=} (l = a)$. Conversly, each schema S where every predicate is of the form P_a for some $a \in Label$ can be viewed as a database, by replacing P_a with a. Next, for any two schemas S, S', we define $S \sqcap S'$ to be obtained by (1) constructing the product schema, with nodes of the form (u, u'), for u in S and u' in S', and edges $(u, u') \stackrel{P \wedge P'}{\rightarrow} (v, v')$, for any edges $u \stackrel{P}{\rightarrow} v$ in S and $u' \stackrel{P'}{\rightarrow} v'$ in S', then (2) "pruning" $S \times S'$, i.e. eliminating all edges labeled *false*, and all nodes which are unaccessible from the root (s_1, s'_1). In particular, if DB is any database, then it makes sense to talk about $DB \sqcap S$, and one can verify that $DB \sqcap S$ is a database. Moreover, it is easy to check that $DB \sqcap S$ conforms to S: the simulation is $\sim$ given by $(x, u) \sim u'$ iff $u = u'$. Note that we have $S \sqcap S = S$, since all our schemas are deterministic, but, in general, $DB \sqcap DB \neq DB$, because databases are not deterministic. We have:

Lemma 3.5 *If DB conforms to S, then for any path from the root of DB to some node x labeled $a_1 a_2 \ldots a_n$, there exists some path from the root of $DB \sqcap S$ to some node (x, u) labeled $a_1 a_2 \ldots a_n$*

Next we take another look at the way we evaluate a single regular query $Q(DB) =$ select x where $R.x$ in DB. Let A be some automaton equivalent to R, and let $A \sqcap DB$ be as above (just view A as a schema). Define:

$$output(A \sqcap DB) \stackrel{\text{def}}{=} \{x \mid \exists u.(u, x) \in A \sqcap DB, \text{ and } u \text{ is a terminal state}\}$$

Then it is easy to check that $output(A \sqcap DB)$ is precisely the result of evaluating Q on DB. We can prove now:

Lemma 3.6 *Let Q_1, Q_2 be two single regular queries and S a schema. Then Q_1, Q_2 are equivalent over S iff Q_1^S is equivalent to Q_2^S.*

Proof: Let A_1, A_2 be nondeterministic automata for the queries Q_1, Q_2. We have to prove that $output(DB \sqcap A_1) = output(DB \sqcap A_2)$ for all databases DB conforming to S iff $output(DB \sqcap (S \sqcap A_1)) = output(DB \sqcap (S \sqcap A_2))$ for all databases DB. We prove the implication $\Longrightarrow$ first. First notice that $DB \sqcap (S \sqcap A_1) = (DB \sqcap S) \sqcap A_1$ and $output(DB \sqcap (S \sqcap A_1)) =$

$\pi_1(output((DB \sqcap S) \sqcap A_1))$. Similarly for A_2. Then the claim follows by:

$$\begin{aligned} output(DB \sqcap (S \sqcap A_1)) &= \pi_1(output((DB \sqcap S) \sqcap A_1)) \\ &= \pi_1(output((DB \sqcap S) \sqcap A_2)) \text{ since } DB \sqcap S \text{ conforms to } S \\ &= output(DB \sqcap (S \sqcap A_2)) \end{aligned}$$

For the implication $\Longleftarrow$ we use Lemma 3.5. Let DB be some database which conforms to S. Using Lemma 3.5, we have that $output(DB \sqcap A_1) = \pi_1(output((DB \sqcap S) \sqcap A_1)) = output(DB \sqcap (S \sqcap A_1))$, and similarly for A_2. Now we use our assumption that $output(DB \sqcap (S \sqcap A_1)) = output(DB \sqcap (S \sqcap A_2))$ for all databases DB. □

We prove now soundness for the pruning of single regular queries:

Proposition 3.7 *For any single regular query Q and schema S, Q is equivalent to Q^S over S.*

Proof: Using Lemma 3.6 we have that Q is equivalent to Q^S over S iff Q^S is equivalent to $(Q^S)^S$. The latter has automaton $S \sqcap (S \sqcap A) = (S \sqcap S) \sqcap A = S \sqcap A$, because S is deterministic. □

Now we can prove soundness for the general case:

Proposition 3.8 *For any regular query Q and schema S, Q is equivalent to Q^S over S.*

Proof: Let Q have k conjuncts, $R_1.y_1$ in $x_1, \ldots, R_k.y_k$ in x_k, and let G be the AND/OR graph constructed for Q. We show that, for any database DB conforming to S, and for any variable z occurring in Q, if there exists a substitution θ of Q's variables satisfying all conjuncts s.t. $\theta(z) \in ext(s_i)$, for some state s_i in S, then the node (s_i, z) in G is accessible. For that we construct a certain set A of nodes of G, based on θ, then show that A is an accessibility property (see Definition 3.1). Namely A consists of (1) all nodes of the form (s_i, z) for which $\theta(z) \in ext(s_i)$, (2) all nodes of the form (s_i, z, A_l) for which $\theta(z) \in ext(s_i)$, (3) all states (s_i, a) of automaton A_l for which, refering to the condition $R_l.y_l$ in x_l, there exists a path in DB from $\theta(x_l)$ to $\theta(y_l)$ satisfying R_l s.t. one of the intermediate nodes u is in $ext(s_i)$ and corresponds to state a of the automaton A_l. Based on the fact that θ is a substitution which satisfies all k conjunts, one can verify that A is indeed an accessibility property. Hence, if $\theta(z) \in ext(s_i)$ for some variable z and state s_i, then (s_i, z) is certainly accessible, which proves our claim. Finally, we observe that states dropped from the automata $S \sqcap A_l$, for $l = 1, k$, do not contribute for any substitution θ, since they are not on a path from an accessible input state to an accessible output state. □

Next, we show that in the case of single regular queries, the pruned query is optimal, if we assume a naive query-evaluation strategy that relies only on graph traversal. Given a single regular query $Q(DB) =$ select x where $R.x$ in DB and a database DB, define the *cost* of computing $Q(DB)$, $cost(Q, DB)$, to be the number of edges in DB that are accessible from the root with a path whose labels are a prefix of R. This is precisely the number of edges traversed by a naive algorithm, which searches for paths in the graph that match the regular expression R.

Proposition 3.9 *Let S be a schema and Q a single regular query. Then for any single regular query Q' equivalent to Q over S and any database DB conforming to S, $cost(Q^S, DB) \leq cost(Q', DB)$.*

Proof: Assume the contrary, and let DB be some database conforming to S and $u \xrightarrow{a} v$ an edge which is traversed by the query Q^S but not by Q'. We will construct an extended database DB_1 which still conforms to Q but on which Q' gives the wrong answer. Let $S \sqcap A$ and A' be the automata for Q^S and Q' respectively. By definition, there exists some path in DB from the root to u, labeled $a_1 a_2 \ldots a_n$, s.t. $a_1 a_2 \ldots a_n a$ is a prefix of the language accepted by $S \sqcap A$. We can extend that word to a word $a_1 a_2 \ldots a_n a b_1 \ldots b_m$ accepted by $S \sqcap A$. We extend DB with $m+1$ new nodes and edges: $u \xrightarrow{a} v_0 \xrightarrow{b_1} v_1 \xrightarrow{b_2} v_2 \ldots \xrightarrow{b_m} v_m$, and call the resulting database DB_1. It still conforms to S: for that, it suffices to extend any simulation from DB to S with pairs of the form (v_i, s_j), whenever v_i corresponds to some state of the form $(s_j, \_)$ in $S \sqcap A$. Hence v_m is in the answer $Q^S(DB_1)$, hence in $Q(DB_1)$ too. But it cannot be in the answer of $Q'(DB_1)$, because Q' does not traverse the edge $u \xrightarrow{a} v_0$ (since it did not traverse $u \xrightarrow{a} v$ either). Note that we had to introduce the fresh node v_0, because Q' may still visit v on some other path, which avoids the edge $u \xrightarrow{a} v$. □

Finally, we discuss the cost of the pruned query for the case of arbitrary regular queries. We face two problems here which do not allow us to generalize Proposition 3.9 easily. The first one is that there doesn't seem to exists a unique canonical naive cost model, consisting of all edges "touched" by a certain query, as was the case for single regular queries. To see that, consider $Q(DB) =$ select z where $R_1.x$ in $DB, R_2.y$ in $x, R_3.x$ in $y, R_4.z$ in y. One can evaluate Q naively in the following ways: (1) Evaluate each of the for conditions separatedly, yielding four binary relations, then compute their join. Here the cost is that of computing each of the four regular expressions (in which only the first one starts at the root, the other three start at arbitrary nodes), plus the cost of computing the joins. (2) Impose some order on the conjuncts, say R_1, R_2, R_3, R_4, then proceed as follows: first *find* all nodes x reachable from DB by R_1, then *find* all nodes y reachable from x by R_2, then *check* that x is reachable from y by R_3, finally *find* all nodes z reachable from y by R_4. Here the cost is the cost of all edges touched during the four traversals

A second problem is a fine point in our method of pruning arbitrary regular queries. Namely, the essence of our method consists in discovering, for each variable z, in which states it may be in, i.e. which nodes (s_i, z) in the AND/OR graph are accessible, giving us which subsitutions $\theta(z) \in ext(s_i)$ make sense. One variable z may be in several states s_i, and, for some conjunct $R_l.y_l$ in z, our pruning method proceeds conservatively, by assuming that the search in R_l may start in any of these states. However not all combinations of variables/states are valid. That is, referring to the conjuct $R_l.y_l$ in z, it may be the case that z may be in both s_5 and s_7, while y_l may be in s_2 and s_9, but when z is in s_5 then y_l is in s_2 only, and when z is in s_7 then y_l is in s_9 only. By missing this correspondence, our pruned query may follow some useless links, that is when starting with z in s_5 it may follow links which made sense only for searches starting in s_7, and vice versa. There are two ways to improve this picture. The first is to improve the naive evaluation algorithm as follows. To evaluate a pruned query Q^S, consider some condition $R_l.y_l$ in x_l, and suppose we have a binding of the variable x_l to some node $\theta(x_l)$ in the database DB. Consider all states s_i s.t. $\theta(x_l) \in ext(s_i)$. Then, instead of the pruned regular expression R_l^S consider the tighter one obtained from the automaton $S \times A_l$ by taking as input state only (s_i, a), with s_i one of the states above, and a the input state in A_l. This is "tighter" than R_l^S. This algorithm is probably optimal when compared to the naive evaluation algorithm of unpruned equivalent queries, but here we are comparing apples with oranges, since the improved algorithm makes sense only for pruned queries. A second way to go is to improve the way we prune a query, to

obtain a tighter bound on the regular expressions. Define a *binding*, β, to be a function from the variables in the query to states in S s.t. $(\beta(z), z)$ is accessible in the AND/OR graph, for every variable z. For each binding β we can produce a specialized query $Q^{S,\beta}$, in which the regular expressions are specifially pruned for that binding (i.e. for the condition $R_l.y_l$ in x_l, we take only the state $(\beta(x_l), a)$ to be the initial state in $S \times A_l$, where a is the initial state in A_l, and with $(\beta(y_l), a')$ to be the terminal states, where a' is a terminal state in A_l). Some bindings make no sense, in that they yield empty regular expressions $R_l^{S,\beta}$: we discard those. Finally, we express $Q(DB)$ as a big union $\bigcup_\beta Q^{S,\beta}(DB)$. Each $Q^{S,\beta}$ is "optimal". The problem with this approach is that the number of bindings may be exponentially large in the size of the query. To illustrate, consider the following:

Example 3.10 Consider the following query:

$$\begin{aligned} Q(DB) = \mathsf{select}\ & z \\ \mathsf{where}\ & \mathit{true}\ .x_1 \ \mathsf{in}\ DB, \ \ldots, \ \mathit{true}\ .x_n \ \mathsf{in}\ DB, \\ & \mathit{true}\ .y_1 \ \mathsf{in}\ x_1, \quad \ldots, \ \mathit{true}\ .y_n \ \mathsf{in}\ x_n, \\ & *.c_1.z \ \mathsf{in}\ y_1, \qquad \ldots, *.c_n.z \ \mathsf{in}\ y_n \end{aligned}$$

and schema S in Figure 6. During pruning we discover that each x_i is in either s_1 or s_2, each y_i is in s_1' or s_2', and z is in s. But we lose the information that, when x_i is in s_1, then y_i is in s_1', and cannot be in s_2'. Hence, the pruned query is:

$$\begin{aligned} Q^S(DB) = \mathsf{select}\ & z \\ \mathsf{where}\ & (a_1|a_2).x_1 \ \mathsf{in}\ DB, \ \ldots, (a_1|a_2).x_n \ \mathsf{in}\ DB, \\ & (b_1|b_2).y_1 \ \mathsf{in}\ x_1, \qquad \ldots, (b_1|b_2).y_n \ \mathsf{in}\ x_n, \\ & (d_1|d_2)*.c_1.z \ \mathsf{in}\ y_1, \ \ldots, (d_1|d_2)*.c_n.z \ \mathsf{in}\ y_n \end{aligned}$$

This query however is not optimal. Indeed, when x_1 is bound to some node in $\mathit{ext}(s_1)$, then y_1 is by necessity in $\mathit{ext}(s_1')$, and then the last regular expression on that chain should be: $d_1*.c_1.z$ in y_1: in other words, we should stop searching for c_1 when we encounter a d_2, something which Q^S doesn't do (because a c_1 may still occur under a d_2 in the other case, when both x_1 is in $\mathit{ext}(s_2)$ and y_2 in $\mathit{ext}(s_2')$). The optimally pruned query however can be expressed as a union of 2^n queries, as follows. Consider all 2^n sequences $\bar{\beta} = (\beta_1, \ldots, \beta_n)$, where $\forall i, \beta_i \in \{1, 2\}$. For each $\bar{\beta}$, define the query $Q^{S,\bar{\beta}}$ to be:

$$\begin{aligned} Q^{S,\bar{\beta}}(DB) = \mathsf{select}\ & z \\ \mathsf{where}\ & a_{\beta_1}.x_1 \ \mathsf{in}\ DB, \ \ldots, a_{\beta_n}.x_n \ \mathsf{in}\ DB, \\ & b_{\beta_1}.y_1 \ \mathsf{in}\ x_1, \qquad \ldots, b_{\beta_n}.y_n \ \mathsf{in}\ x_n, \\ & d_{\beta_1}*.c_1.z \ \mathsf{in}\ y_1, \ \ldots, d_{\beta_n}*.c_n.z \ \mathsf{in}\ y_n \end{aligned}$$

Obviously $Q(DB)$ is equivalent to $\bigcup_{\bar{\beta}} Q^{S,\bar{\beta}}$, because each $\bar{\beta}$ covers one possibile combination of assigning $x_1, \ldots, x_n$ to $\mathit{ext}(s_1)$ or $\mathit{ext}(s_2)$ (the assignments for the y_i's follow), and each $Q^{S,\beta}$ is obtimal, for its own particular assignment.

4 Query Rewriting Using State Extents

Our second optimization technique, called *query rewriting using state extents*, applies only to single regular queries, $Q(DB) =$ select x where $R.x$ in DB. Given schema S with

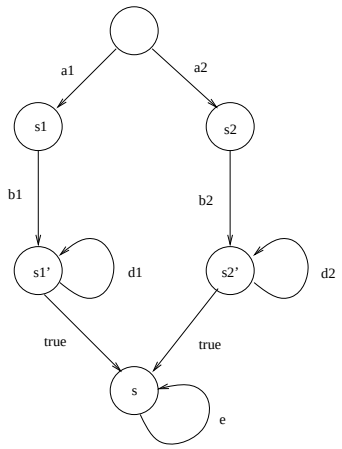

Figure 6: Schema for Example 3.10.

states $s_1, \ldots, s_n$, we assume that we have computed all states' extents. Our optimization re-expresses Q as:

$$Q(DB) \quad = \quad Q_1(ext(s_{j_1})) \cup \ldots \cup Q_p(ext(s_{j_p})) \tag{1}$$

where $Q_1, \ldots, Q_p$ are arbitrary single regular queries, and $s_{j_1}, \ldots, s_{j_p}$ are distinct states in S. The meaning of $Q_i(ext(s_{j_i}))$ is the following: for each node $u \in ext(s_{j_i})$, evaluate Q_i on the database with u designated as root, then take the union of all results. Each solution to Equation (1) corresponds to a logical plan, which consists of several graph traversals starting at nodes in $ext(s_{j_1}), \ldots, ext(s_{j_p})$. Compared with the naive evaluation of $Q(DB)$, which requires a single traversal from DB's root, this plan could be much more efficient if each of $Q_1, \ldots, Q_p$ performs only short traversals or none at all (i.e., returns the nodes u directly).

For example, consider Query 1.1, and schema S in Fig. 2(a). One solution to Equation (1) is:

$$\begin{aligned} Q(DB) = &(\mathsf{select}\ p\ \mathsf{where}\ p\ \mathsf{in}\ ext(s_{10})) \cup \\ &(\mathsf{select}\ p\ \mathsf{where}\ p\ \mathsf{in}\ ext(s_{12})) \end{aligned}$$

which is just $ext(s_{10}) \cup ext(s_{12})$, and corresponds to Plan 4 in Section 1.2. Here, neither Q_1 nor Q_2 traverses the graph, hence the cost of evaluating the right hand side of Equation (1) is much lower than the evaluation of Q. Another solution is:

$$\begin{aligned} Q(DB) = &(\mathsf{select}\ p\ \ \mathsf{where}\ *.Project.p\ \mathsf{in}\ ext(s_4)) \cup \\ &(\mathsf{select}\ p\ \ \mathsf{where}\ *.Project.p\ \mathsf{in}\ ext(s_7)) \end{aligned}$$

Here Q_1 and Q_2 do traverse the graph, searching for *Project* starting at nodes in $ext(s_4)$ and $ext(s_7)$. The evaluation cost is higher than in the previous solution, but may still be lower than the evaluation of $Q(DB)$, which starts at the root.

For each solution $E \stackrel{\text{def}}{=} Q_1(ext(s_{j_1})) \cup \ldots \cup Q_p(ext(s_{j_p}))$ of Equation (1), define its cost, $cost(E)$, to be the total number of edges traversed in a naive evaluation of E. That is, using the notations in Section 3, $cost(E) \stackrel{\text{def}}{=} cost(Q_1, ext(s_{j_1})) + \ldots + cost(Q_p, ext(s_{j_p}))$.

In general, we want to find all solutions to Equation (1) and give them to a cost-based optimizer to choose the best plan. The difficulty in solving Equation (1) lies in the fact that it mixes three independent notions: (1) some equality between queries based on regular expressions, (2) the *extents*, which are defined in terms of simulations, and (3) the fact that we have the additional assumption that DB conforms to S.

Our first insight is that this problem is an instance of the *query rewriting problem* [LMSS95] for recursive queries. To see this, define n views $V_1, \ldots, V_n$ as follows. Consider each state s_j in S, and let T_j be the regular expression equivalent to the automaton obtained from S by defining s_1 as initial state and s_j as terminal state. Define $V_j(DB) \stackrel{\text{def}}{=} \mathsf{select}\ x\ \mathsf{where}\ T_j.x\ \mathsf{in}\ DB$. For example, refering to the schema S in Figure 3, the regular expression associated to s_2 is $T_2 = \mathit{other}*.\mathit{Org}.\mathit{other}*$. We show:

Proposition 4.1 *Let S be a graph schema with states $s_1, \ldots, s_n$, and DB a database conforming to S Then, for any state s_j, $ext(s_j) = V_j(DB)$.*

Proof: Throughout the proof, let $\sim$ be the minimal simulation from DB to S, i.e. $u \sim s_j$ iff $u \in ext(s_j)$. First we show $ext(s_j) \subseteq V_j(DB)$. Let $u \in DB$ be some node in $ext(s_j)$. Since $\sim$ is the minimal simulation, there exists a path from DB's root to u labeled $a_1.a_2 \ldots a_k$, and there exists a path in S from s_1 to s_j labeled $P_1.P_2 \ldots P_k$ s.t. $P_i(a_i)$ is true, for every $i = 1, k$. This proves that $u \in V_j(DB)$. Next, we show that $V_j(DB) \subseteq ext(s_j)$. Let $u \in V_j(DB)$, i.e. there exists a path in DB leading to u, labeled $w \stackrel{\text{def}}{=} a_1.a_2 \ldots a_k$, s.t. w leads in S from s_1 to s_j. Using the simulation $\sim$, and following the path w in DB from the root, we discover that each intermediate node in DB is similar to some state in S: since S is deterministic, that state is precisely the corresponding state on the path for w in S. Hence, $u \sim s_j$. □

Equation (1) expresses the query $Q(DB)$ in terms of the views $V_1, \ldots, V_n$. That is, our problem can be restated as: find all sets of pairs $(Q_1, s_{j_1}), \ldots, (Q_p, s_{j_p})$ s.t. $Q_1, \ldots, Q_p$ are single regular queries and $s_{j_1}, \ldots, s_{j_p}$ distinct states in S, s.t., for all databases DB conforming to S:

$$Q(DB) \quad = \quad Q_1(V_{j_1}(DB)) \cup \ldots \cup Q_p(V_{j_p}(DB))$$

This is an instance of the view rewriting problem for *recursive* queries (since both the query and the views are recursive), with the additional complication that the databases are constraint to conform to S.

The particular query rewriting problem we have to solve can be expressed in terms of equivalence between two regular expressions. If $Q_i = \mathsf{select}\ y\ \mathsf{where}\ R_i.y\ \mathsf{in}\ x$, then $Q_i(V_{j_i}(DB)) = \mathsf{select}\ y\ \mathsf{where}\ T_{j_i}.R_i.y\ \mathsf{in}\ DB$, hence Equation (1) can be rewritten as:

$$Q(DB) \quad = \quad \mathsf{select}\ y\ \mathsf{where}\ R'.y\ \mathsf{in}\ DB \tag{2}$$

where $R' = (T_{j_1}.R_1 \mid \ldots \mid T_{j_p}.R_p)$. However, $R \equiv R'$ is only a sufficent, and not a necessary condition, because Equation 2 has to hold only for databases conforming to S.

Checking containment and equivalence of regular expressions is decidable and PSPACE complete [SM73]. In our setting however, regulare expressions contain *predicates*, and the universe of constants, *Label*, is infinite. If the predicates contain all constant-equality predicates $P_a(l) \stackrel{\text{def}}{=} (l = a)$, for all $a \in \mathit{Label}$, then surely containment and equivalence of regular expressions remains PSPACE hard. We show next that it is still in PSPACE. For that we

assume an encoding of the predicates and of the labels in *Label* such that the following conditions hold: (1) for any predicate P and constant a, one can compute $P(a)$ (which is *true* or *false*) in polynomial space in the size of the encodings of P and a, and (2) let $E(l)$ be a boolean expression on predicates, of the form $E(l) \stackrel{\text{def}}{=} P_0(l) \wedge \neg P_1(l) \wedge \neg P_2(l) \wedge \ldots \wedge P_m(l)$; then if E is satisfiable ($\exists l.E(l)$), then there exists some $a \in Label$ which admits and encoding whose size is bounded by a polynomial in that of E, s.t. $E(a)$ is true. Then we show:

Proposition 4.2 *Under the assumptions above, containment and equivalence of regular expressions with predicates remains PSPACE complete.*

Proof: We only have to show how to check $R \subseteq R'$ in PSPACE. Let A, A' be nondeterministic automaton for R, R'. We check $A \not\subseteq A'$ using a non-deterministic, polynomial space Turing Machine, then use Sawitch's theorem [Pap94, pp.149-150]. For that, we generate nondeterministically a sequence of labels $a_1, a_2, \ldots a_n \in Label$ which is accepted by A but not by A'. As we generate the sequence, we do not keep its history, but only the current label a_i, as well as the states in A and A'. To check at the end that the sequence is accepted by A, it suffices to keep the current state in A, and pick nondeterministically the next one, at each transition (and, at the end, check that we are in a terminal state). To show at the end that the sequence is *not* accepted by A', we keep the *set* of all states in A' where the sequence lead us so far: in the end we check that no terminal state is in the current set of states. So far this does not differ from the classical case. The novel part is that here the alphabet *Label* is infinite. We restrict the search to sequences in which every a_i has an encoding whose size is polynomial in R and R': then the total spaced used is polynomial, and at each transition we can compute the predicates involved in polynomial space. We show that it suffices to search for sequences with this restriction. Indeed, take any sequence $b_1 b_2 \ldots b_n \in Label$ which is accepted by A but not by A'. We will show how to construct a sequence of the same length, $a_1 \ldots a_n$, in which the size of every a_i is bounded by a polynomial. Considering b_i, let P_0 be the predicate in A which corresponds to the transition of b_i on the successful path in A: that is, $P_0(b_i)$ is true. Looking at A', b_i generates a transition from some set of states S_i to another set of states S_{i+1}. Let $P_1, \ldots, P_m$ be all the predicates on all transitions in A' which do not hold on b_i. Define $E(l) \stackrel{\text{def}}{=} P_0(l) \wedge \neg P_1(l) \wedge \ldots \neg P_m(l)$: then $E(b_i)$ is true, and there exists some a_i, of polynomial size, s.t. $E(a_i)$ is true too. If we replace b_i with a_i in the sequence, it will still traverse the successful path in A, while in A' it will generate transitions from the set of states S_i to a possible smaller set of states $S'_{i+1} \subseteq S_{i+1}$. This proves our claim. □

Still, our query rewriting problem does not reduce immediatley to equivalence of regular expressions. We face two obstacles. First, we have to *find* the regular expressions $R_1, \ldots, R_p$, before we check $R \equiv R'$, and, in general, there are infinitely many. Second, Equation (2) must hold only on databases conforming to S, hence it does not translate immediately into $R \equiv R'$.

To illustrate the first obstacle, we give an example for which the set of solutions is infinite. Let S have a single edge[3] $s_1 \stackrel{a}{\to} s_1$, and $Q =$ select x where $*.x$ in DB. Define $Q_n(x) \stackrel{\text{def}}{=}$ select y where $(a|\underbrace{b.b\ldots b}_{n})*.y$ in x: then $Q(DB) = Q_n(ext(s_1))$ for every $n \geq 0$, and Equation (1) has infinitely many solutions. We show how to reduce the search space to a finite subset of solutions. Fix some nondeterministic automaton A equivalent to R,

[3] S specifies that all edges in the database are labeled a.

with states $a_1, \ldots, a_p$. For each state a_i let R_i be the regular expressions equivalent to the automaton obtained from A by re-designating a_i as initial state. We call the query select y where $R_i.y$ in x an *atomic canonical query*. A *canonical query*, C, is a union of atomic canonical queries. Also, for each state s_j in S, let S_j denote the schema obtained from S by designating s_j to be the root. Recall that C^{S_j} is the query C pruned against S_j. Then:

Theorem 4.3 *Consider a solution* $E \stackrel{\text{def}}{=} \bigcup_{i=1,p} Q_i(ext(s_{j_i}))$ *to Equation (1). Then there exists* p *canonical queries* $C_1, \ldots, C_p$ *s.t.* $E' \stackrel{\text{def}}{=} \bigcup_{i=1,p} C_i^{S_{j_i}}(ext(s_{j_i}))$ *is still a solution and* $cost(E') \leq cost(E)$.

Proof: For each $i = 1, p$, let C_i be the following canonical query. Consider the automaton $S \cap A$, and all states a in A s.t. (s_{j_i}, a) is in $S \cap A$ (i.e. (s_{j_i}, a) is on some path from the initial state to a terminal state in $S \times A$). Each such a defines an atomic canonical query. Let us denote with $W_1(s_{j_i}, a)$ the regular language of words causing in $S \times A$ a transition from the initial state to (s_{j_i}, a), and let $W_2(s_{j_i}, a)$ denote the regular language of words causing in $S \times A$ a transition from (s_{j_i}, a) to some terminal state. Call (s_{j_i}, a) *contentious* if $\exists w_2 \in W_2(s_{j_i}, a), \exists a', \exists w_1 \in W_1(s_{j_i}, a)$ s.t. $w_1.w_2$ is not accepted by A (or, equivalently, by $S \sqcap A$). For example, in Figure 7, (s_2, a_3) is contentious, because $d \in W_2(s_2, a_3)$ and $a \in W_1(s_2, a_3)$ and $a.d$ is not in the language. We take C_i to be the union of those atomic canonical queries corresponding to states a s.t. $(s_{j_i}, a) \in S \sqcap A$ and (s_{j_i}, a) is not contentious. One can check that the pruned query, $C_i^{S_{j_i}}$ is precisely that corresponding to the automaton $S \sqcap A$ in which all non-contentious states (s_{j_i}, a) are initial states (i.e. its regular language is the union of all $W_2(s_{j_i}, a)$, for (s_{j_i}, a) non-contentious).

We show first that $E' \stackrel{\text{def}}{=} \bigcup_{i=1,p} C_i^{S_{j_i}}(ext(s_{j_i}))$ is still a solution. First, we show that $E' \subseteq Q(DB)$, by showing that $C_i^{S_{j_i}}(ext(s_{j_i})) \subseteq Q(DB)$, for every $i = 1, p$. Let $u \in C_i^{S_{j_i}}(ext(s_{j_i}))$. That is, there exists some node $v \in ext(s_{j_i})$ s.t. $u \in C_i^{S_{j_i}}(v)$; let w_2 be the word labeling some path from v to u s.t. in $S \sqcap A$, there exists a path labeled w_2 from some state of the form (s_{j_i}, a) to a terminal state, and the state (s_{j_i}, a) is not contentious. That is, $w_2 \in W_2(s_{j_i}, a)$. On the other hand, since $v \in ext(s_{j_i})$, there exists some path from DB's root to v, labeled w_1, which in S causes a transition from the root to s_{j_i}. We have to show that $w_1.w_2$ is accepted by $S \sqcap A$ (or, equivalently, by A). First we show that there exists a transition in $S \sqcap A$ corresponding to w_1, starting at the root (i.e. w_1 does not get stuck). For that we recall that the query Q_i is nonemtpy, i.e. there exists some word w accepted by its regular expression. Consider the database DB' consisting of a single chain labeled $w_1.w$, and let v' be the intermediate node, i.e. there exists a path labeled w_1 from the root to v' and one labeled w from v' to a leaf u'. Obviously DB' still conforms to S and v' is in the extent of s_{j_i}. Hence $u' \in Q_i(ext(s_{j_i}))$, hence $u' \in Q(DB')$, hence $w_1.w$ causes some transition in $S \sqcap A$. Let (s_{j_i}, a') be the state in $S \sqcap A$ reached after w_1: obviously it is the same s_{j_i} (because S is deterministic), but we may have $a' \neq a$. We have now $w_1 \in W_1(s_{j_i}, a')$. Since (s_{j_i}, a) is not contentious, it follows that $w_1.w_2$ is accepted by A.

Next we show $Q(DB) \subseteq E'$. Let $u \in Q(DB)$ and let w be some word on a path from DB's root to u which is accepted by $S \sqcap A$. Considering the intermediate states (s, a) in $S \sqcap A$ traversed by the successful path for w, let (s_{j_i}, a) be the first one for which s is one of our states in consideration, $s_{j_1}, \ldots, s_{j_p}$ (there has to be at least one such state: otherwise, we construct another database DB' consisting of a single chain labeled w ending in a node u', and argue that $u' \in Q(DB)$ but $u' \notin E$, contradicting $Q(DB) = E$). This splits w into

$w_1.w_2$. We will show that (s_{j_i}, a) is not contentious: then, obviously, $u \in C_i^{S_{j_i}}(ext(s_{j_i}))$, which proves our claim. So assume the contrary, that (s_{j_i}, a) is contentious. That is there exists some other state (s_{j_i}, a') and $w_1' \in W_1(s_{j_i}, a')$, $w_2' \in W_2(s_{j_i}, a)$ s.t. $w_1'.w_2' \notin S \sqcap A$. We have that $w_1.w_2' \in S \sqcap A$, because w_1 leaves $S \sqcap A$ in the same state (s_{j_i}, a) where w_2' picks up to lead to a terminal state. Now consider again a database DB' consisting of the single chain $w_1.w_2'$ ending in some leaf node u': obviously DB' still conforms to S, and $u' \in Q(DB)$, hence $u' \in Q_k(ext(s_{j_k}))$ for some $k = 1, p$. The path $w_1.w_2'$ cannot reach a state s_{j_k} in its w_1 fragment — recall that we took s_{j_i} to be the first such state encountered by w — hence it is w_2' which splits int $w_2' = w_3.w_4$ (with w_3 possibly empty) s.t. $w_1.w_3$ leads in DB' to a node in state s_{j_k}, and w_4 is accepted by Q_k. So $w = w_1.w_3.w_4$. At this point we replace w_1 with w_1', and construct a new database DB'' consisting of exactly the word $w_1'.w_3.w_4 = w_1'.w_2'$. It still conforms to S, and $w_1'.w_3$ still leads to a node in state s_{j_k}, from where Q_k accepts w_4: in short the whole word $w_1'.w_2'$ must be accepted by Q, which is a contradiction.

Finally, we argue that $cost(E') \leq cost(E)$. Since our naive cost model is simply the total number of edges ispected by E or E' respectively, we can use the same argument as in Proposition 3.9.

□

Example 4.4 We illustrate with an example a difficulty in the proof of Theorem 4.3. Consider $Q(DB) =$ select x where $(a.c|b.c|b.d).x$ in DB, and schema S in Figure 7. Here A (not shown) and $S \sqcap A$ are isomorphic, and both happen to be deterministic. One solution to Equation (1) is:

$$Q(DB) = Q_1(ext(s_1)) \cup Q_2(ext(s_2)) \quad (3)$$

$$Q_1(x) = \text{select } y \text{ where } b.d.y \text{ in } x \quad (4)$$

$$Q_2(x) = \text{select } y \text{ where } c.y \text{ in } x \quad (5)$$

For the particular automaton A chosen, there are four atomic canonical queries corresponding to the four states. This example illustrates why we *cannot* choose C_i to be the union of all atomic canonical queries for states a s.t. $(s_{j_i}, a) \in S \sqcap A$. In our example, consider state s_2. Only the automaton states a_2 and a_3 appear with s_2 in $S \sqcap A$, so we may be tempted to define C_2 to be their union, that is $C_2(x) \stackrel{\text{def}}{=}$ select y where $(c|d).y$ in x; for s_1, there is a single atomic canonical query, hence $C_1(x) \stackrel{\text{def}}{=}$ select y where $(a.c|b.c|b.d).y$ in x. But:

$$Q(DB) \neq C_1(ext(s_1)) \cup C_2(ext(s_2))$$

because $C_2(ext(s_2))$ would also return nodes reachabel by a path labeled $a.d$. Instead, in the proof of Theorem 4.3 we defined C_2 to be the union of only those atomic canonical queries corresponding to non-contentious states. In our example (s_2, a_3) is contentious: we have $W_2(s_2, a_3) = \{c, d\}$ and $W_1(s_2, a_2) = \{a\}$, hence we can pick $w_1 = a$ and $w_2 = d$ and we have $w_1.w_2 \notin S \sqcap A$. Only (s_2, a_2) is noncontentious, which gives us $C_2(x) \stackrel{\text{def}}{=}$ select y where $c.y$ in x.

Finally we comment on the cost of the resulting canonical solution. Obivously, in our example $C_2(ext(s_2))$ is superfluous, since we already have $Q(DB) = C_1(ext(s_1))$. However, according to our naive cost model, $C_1(ext(s_1)) \cup C_2(ext(s_2))$ has the same cost as $C_1(ext(s_1))$, simply because they touch the same set of edges, in any database conforming to S.

In short, Theorem 4.3 states that it suffices to search for solutions to Equation 1 that consist only of canonical queries, which are finitely many, and prune those queries against the schemas S_j.

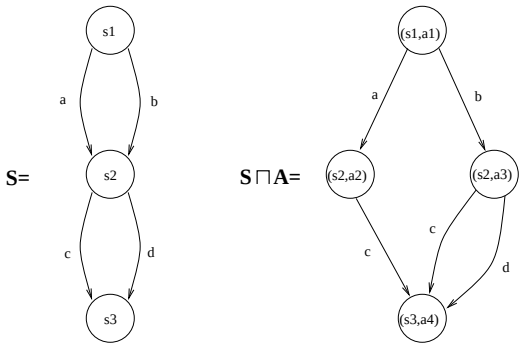

Figure 7: Schema and automaton for Example 4.4.

To illustrate the second obstacle, consider the schema $s_1 \xrightarrow{a} s_2 \xrightarrow{b \vee c} s_2$; databases conforming to S have a-edges starting at the root, leading to arbitrary subgraphs containing only b and c edges. The regular expression defining $ext(s_2)$ is $T_2 = a.(b|c)*$. Let $Q(DB) =$ select x where $R.x$ in DB, with $R = a.((b|d)*)$. One canonical query is $C_1(x) =$ select y where $R_1.y$ in x, with $R_1 = (b|d)*$. Then, $Q(DB) = C_1(ext(s_2))$ is a solution to Equation (1) (and (2)). However, $R \not\equiv T_2.R_1$. The emphasis is that Equation (2) must hold *only* for databases that conform to S.

We solve this problem by pruning the queries $Q_1, \ldots, Q_p$ in Equation (1). In particular,

Proposition 4.5 *$Q_1, \ldots, Q_p$ are solutions to Equation (1) iff the following equation holds for* all *databases:*

$$Q^S(DB) = Q_1^{S_{j_1}}(V_{j_1}(DB)) \cup \ldots \cup Q_p^{S_{j_p}}(V_{j_p}(DB))$$

Proof: We have to show that:

$$Q(DB) = Q_1(V_{j_1}(DB)) \cup \ldots \cup Q_p(V_{j_p}(DB))$$

for all databases DB conforming to S iff

$$Q^S(DB) = Q_1^{S_{j_1}}(V_{j_1}(DB)) \cup \ldots \cup Q_p^{S_{j_p}}(V_{j_p}(DB))$$

for all databases. We use Lemma 3.6. For that, it suffices to observe that the pruned version of the query $P_i(DB) \stackrel{\text{def}}{=} Q_i(V_{j_i}(DB))$ is $P_i^S(DB) = Q_i^{S_{j_i}}(V_{j_i}(DB))$. This follows from the fact that $V_{j_i}^S(DB) = V_{j_i}(DB)$, and that all nodes in the answer of $V_{j_i}(DB)$ are "in" state s_{j_i}. □

Putting everything together, we obtain the following algorithm *State-Rewrite* for query rewriting using state extents.

Algorithm *State-Rewrite*

Step 1 Iterate through all sets of pairs $(C_1, s_{j_1}), \ldots, (C_p, s_{j_p})$, where $C_1, \ldots, C_p$ are canonical queries and $s_{j_1}, \ldots, s_{j_p}$ are distinct states in S. Let $R_1, \ldots, R_p$ be the regular path expressions of the canonical queries, and $R_1^{S_{j_1}}, \ldots, R_p^{S_{j_p}}$ be their pruned versions.

Step 2 Check whether $R \equiv T_{j_1}.R_1^{S_{j_1}} \mid \ldots \mid T_{j_p}.R^{S_{j_p}}$. If yes, then mark $\bigcup_{i=1,p} C_i(ext(s_{j_i}))$ as a solution.

Step 3 Using some cost model, select the solution with the lowest cost.

Theorem 4.6 *Algorithm State-Rewrite is sound and complete, and runs in PSPACE.*

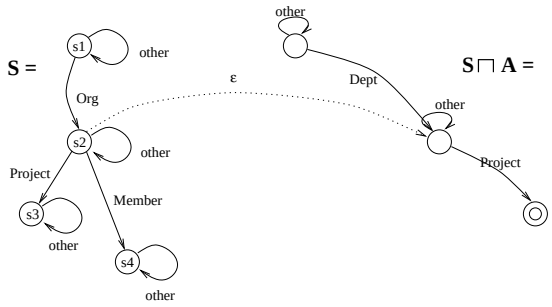

Figure 8: $(S \sqcap A)(\{(s_2, a_2)\})$, for S, A defined in Fig. 3.

4.1 Rephrasing Algorithm *State-Rewrite*

We take now a closer look at algorithm *State-Rewrite*. Recall that A is some automaton equivalent to R, and let $S \sqcap A$ be the pruned automaton. Then Step 1 can be replaced with a single iteration through subsets Σ of states in $S \sqcap A$. This is because a canonical query C_i is a union of atomic canonical queries, hence it is specified by a set Σ_i of states in A; a set of pairs $(C_1, s_{j_1}), \ldots, (C_p, s_{j_p})$ corresponds to the set $\Sigma \stackrel{\text{def}}{=} \bigcup_i \{(s_{j_i}, a) \mid a \in \Sigma_i\}$. Next, we observe that the regular expression $R' \stackrel{\text{def}}{=} T_{j_1}.R_1^{S_{j_1}} \mid \ldots \mid T_{j_p}.R_p^{S_{j_p}}$ is equivalent to an automaton obtained as follows: (1) take the disjoint union of S and $S \sqcap A$, (2) for each $(s_{j_i}, a) \in \Sigma$, add a ε transitions from the state s_{j_i} in S to the state (s_{j_i}, a) in $S \sqcap A$, and (3) the initial state is s_1 (the root of S), and the terminal states are of the form (s_j, a), with s_j arbitrary and a a terminal state. We denote this automaton with $(S \sqcap A)(\Sigma)$. See Fig. 8 for an illustration. The equivalence $R \equiv T_{j_1}.R_1^{S_{j_1}} \mid \ldots \mid T_{j_p}.R^{S_{j_p}}$ translates into an equivalence between two automata, $A \equiv (S \sqcap A)(\Sigma)$. Hence, algorithm *State-Rewrite* can be re-expressed as follows:

Algorithm *State-Rewrite* (rewised)

Step 1: Compute all pairs (s, a) for which $(S \sqcap A)(\{(s, a)\}) \subseteq A$; we call such pairs *candidates*.

Step 2: Iterate through all sets Σ of candidates, and test whether $A \subseteq (S \sqcap A)(\Sigma)$; each Σ determines a solution.

Step 3: Select the minimum cost solution.

Example 4.7 Consider Query 1.1 and schema S in Fig. 3. The automata A and $S \sqcap A$ are also illustrated in Fig. 3. The pair (s_2, a_2) is not a candidate. To see that, consider $(S \sqcap A)(\{(s_2, a_2)\})$: it is illustrated in Fig. 8. We have $(S \sqcap A)(\{(s_2, a_2)\}) \not\subseteq A$. Indeed, the language accepted by $(S \sqcap A)(\{(s_2, a_2)\})$ is $other*.Org.other*.Project$, while that accepted by A is $*.Dept.*.Project$: if d is a label of some organization which is not a department (i.e. $Org(d) \wedge \neg Dept(d)$), and p a project, then $d.p$ is in the frist language but not in the second. In fact, one can check that $\{(s_1, a_1)\}$ is the only candidate for Query 1.1 and schema S in Fig. 3, yielding only a trivial solution to Equation (1).

Example 4.8 We illustrate with a more complex example. Consider the query:

$$Q(DB) = \text{select } x \quad \text{where } (true\ .b.*.e|a.(b|c).*.e|a.c).x \text{ in } DB$$

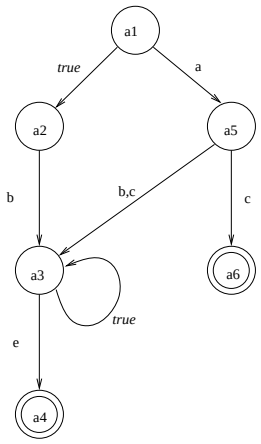

Figure 9: Automaton for the query in Example 4.8.

One possible automaton for this query is depicted in Figure 9. The particular choice of A may seem peculiar (it is chosen with the intent to illustrated a difficulty further in the text), but recall that all our methods so far work for any choice of automaton A. Consider the schema S in Fig. 10: the pruned automaton $S \sqcap A$ is also shown. The predicate not (c) means "all labels except c", i.e. $P(l) \stackrel{\text{def}}{=} (l \neq c)$. Here there are exactly four candidates: $\{(s_1, a_1), (s_2, a_2), (s_5, a_6), (s_2, a_5)\}$. Fig. 10 illustrates $(S \sqcap A)(\Sigma)$, for $\Sigma = \{(s_2, a_2), (s_5, a_6)\}$. The transition $(s_2, a_5) \xrightarrow{c} (s_5, a_6)$ has been dropped because there exists no path from (s_5, a_6) to any terminal state (recall that $S \sqcap A$ contains only those states (s, a) which are on some path from an initial state to a terminal state). We explain next for some of these pairs why they are candidates. Consider (s_2, a_2): $(S \sqcap A)(s_2, a_2)$ accepts the language $a.b.d*.e$, which is included in that accepted by A. Consider (s_5, a_6). The language $(S \sqcap A)(s_5, a_6)$ is $a.c$, which, again, is included in that accepted by A. We explain briefly why some of the others are not candidates, for example (s_3, a_3): $(S \sqcap A)(s_3, a_3)$ accepts the language $a.$ not $(c).d*.e$, which is not included in that accepted by A (because of the not (c) piece: e.g. $a.a.d.d.d.e$ is in $a.$ not $(c).d*.e$, but not in A). Finally, we argue why Σ is indeed a solution. Refering to Fig. 10, $S \sqcap A$ accepts the language $(a.b.d*.e|a.b.d*.e|a.c) = (a.b.d*.e|a.c)$, and is included in (in fact, equal to) that accepted by $(S \sqcap A)(\Sigma)$, which is $a.(b.d*.e|c)$. Other solutions are $\{(s_2, a_2), (s_2, a_5)\}$, $\{(s_1, a_1)\}$, and supersets thereof. Here, the "lower" a solution is in Fig. 10, the lower its cost. Hence, the optimal solution is that corresponding to Σ:

$$Q(DB) \quad = \quad Q_1(ext(s_2)) \cup Q_2(ext(s_5))$$

where Q_1 is the pruned canonical query corresponding to a_2, and Q_2 the pruned atomic canonical query corresponding to a_6:

$$\begin{aligned} Q_1(x) &= \text{select } y \text{ where } b.d*.e.y \text{ in } x \\ Q_2(x) &= \text{select } y \text{ where } y \text{ in } x \end{aligned}$$

Note that Q_2 does not do any traversal.

4.2 Polynomial-Time Approximations for Algorithm *State-Rewrite*

There are two sources for the high complexity of Algorithm *State-Rewrite*: (1) enumerating all canonical queries and all subsets of states (Step 1), and (2) testing regular query con-

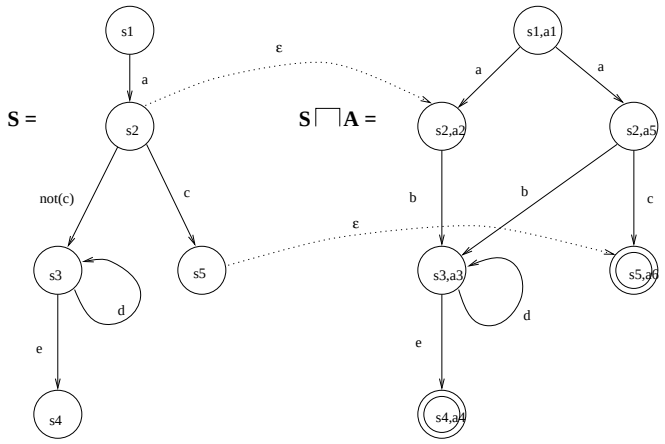

Figure 10: Schema S and pruned automaton $S \sqcap A$ for Example 4.8.

tainment (Step 2). We show here how to reduce their cost. First, we describe a PTIME algorithm which finds the optimal solution to Equation 1, or reports that the equation does not have any solutions, for a restricted class of single regular queries. Second, we describe a PTIME variant of the algorithm *State-Rewrite*, which works on all single regular queries, but which may fail to report solutions, even when one exists.

We need the notion of a "cut" in a graph [Koz91]. We use a non-standard definition, which is reducible to the standard one. For an automaton A, a *cut* is a set of states such that any path from the initial state to some terminal state goes through a state in the cut. Assuming we have a cost $c(a)$ associated to each state a in A, a minimum-cost cut can be found efficiently (in PTIME, with low degrees [GT89]). We show:

Proposition 4.9 *Let A be some automaton, S a graph schema, and Σ a cut in $S \sqcap A$. Then $(S \sqcap A) \subseteq (S \sqcap A)(\Sigma)$.*

Proof: Let w be some word accepted by $S \sqcap A$. That is, there exists a successful path in $S \sqcap A$, from the initial state to a terminal state labeled w. Let (s, a) be some state on that path which belongs to Σ: $(s, a) \in \Sigma$. It splits w into two halves, $w = w_1.w_2$. Then the following is a successful path in $(S \sqcap A)(\Sigma)$: start at the root s_1 of S, and follow the unique path in S for w_1. It will end precisely in state s. From there follow the ε edge to (s, a), and finally follow the path in $S \sqcap A$ from (s, a) to some terminal state, labeled w_2. □

The converse does not hold in general: Figure 10 illustrates such an example, where Σ is not a cut, but $(S \sqcap A) \subseteq (S \sqcap A)(\Sigma)$. However, the converse holds when A is deterministic. We have to take certain precautions however, as illustrated by Figure 11. It shows a deterministic automaton A accepting $(a.c|b.c)$ and schema S, where, for $\Sigma = \{(s_2, a_2)\}$ we have $S \sqcap A \subseteq (S \sqcap A)(\Sigma)$ although Σ is not a cut. But note that the atomic canonical query generated by (s_2, a_2) is the same as that generated by (s_2, a_3), so we may as well take $\bar{\Sigma} = \{(s_2, a_2), (s_2, a_3)\}$ which corresponds to exactly the same solution, and which *is* a cut. Formally, define $\bar{\Sigma} \stackrel{\text{def}}{=} \{(s, a) \mid \exists (s', a') \in \Sigma, W_2(s, a) = W_2(s', a')\}$ (with the notations from the proof of Theorem 4.3, where $W_2(s, a)$ denoted the language accepted by A in which (s, a) has be redesignated as initial state). Obviously $\Sigma \subseteq \bar{\Sigma}$, and Σ and $\bar{\Sigma}$ generate precisely the same sets of canonical queries, hence the same solutions. Refering to Fig. 10, we have $\bar{\Sigma} = \{(s_2, a_2), (s_5, a_6), (s_4, a_4)\}$ is a cut. However, we can still show that, for a nondeterministic automaton, $\bar{\Sigma}$ need not be a

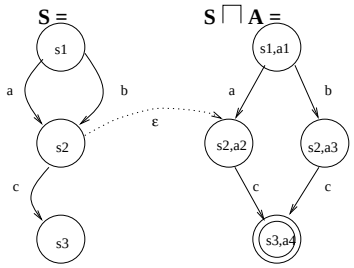

Figure 11: A schema S and a deterministic automaton A with a solution which is not a cut.

cut: for that, just add a new edge $s_5 \xrightarrow{e} s_6$ to S, and a new transition $(s_5, a_6) \xrightarrow{e} (s_6, a_7)$ in $S \sqcap A$. We still have $S \sqcap A = (S \sqcap A)(\Sigma)$, but now $\bar{\Sigma} = \Sigma$ is not a cut any more.

For deterministic automata, $\bar{\Sigma}$ is always a cut:

Theorem 4.10 *Let A be a deterministic automaton, S a schema, and Σ a set of candidates s.t. $(S \sqcap A) \subseteq (S \sqcap A)(\Sigma)$ (equivalently: $(S \sqcap A) = (S \sqcap A)(\Sigma)$). Then $\bar{\Sigma}$ is a cut in $S \sqcap A$.*

Proof: Assume the contrary, and let w be a word accepted by $S \sqcap A$ whose unique acceptance path does not go through any state in $\bar{\Sigma}$. (Recall that $S \sqcap A$ is also a deterministic automaton.) Since w is accepted by $(S \sqcap A)(\Sigma)$, there exits a split $w = w_1.w_2$ s.t. w_1 determines a path in S from the root to some state s, from there there exists an ε link to some state $(s, a_1) \in \Sigma$, and further a path labeled w_2 to some terminal state in $S \sqcap A$ (i.e. $w_2 \in W_2(s, a_1)$). We illustrate throughout the proof with Fig. 12. If there are several choices for splitting w into $w_1.w_2$, we pick that with w_1 having minium length. On the other hand, following the sucessful path of $w_1.w_2$ in $S \sqcap A$ alone, the middle state will be some (s, a_2): it must be the same s (because S is deterministic), but $a_1 \neq a_2$, and by assumption $(s, a_2) \notin \bar{\Sigma}$. Therefore $W_2(s, a_2) \not\subseteq W_2(s, a_1)$, and let v denote a word in $W_2(s, a_2)$ which is not in $W_1(s, a_1)$. Since all states in $S \sqcap A$ are accessible, there exist some $u \in W_1(s, a_1)$. Clearly $u.v \notin S \sqcap A$, because if it were, the first part of the successful path, corresponding to u, would end in (s, a_1) (because $S \sqcap A$ is deterministic), so it would follow $v \in W_2(s, a_1)$. So far we have:

$$\begin{aligned} w_1.w_2 &\in S \sqcap A \\ u.w_2 &\in S \sqcap A \\ w_1.v &\in S \sqcap A \\ u.v &\notin S \sqcap A \end{aligned}$$

Now we look at $w_1.v$. It must also be in $(S \sqcap A)(\Sigma)$, hence we must find a successful path which starts in S, at some point traverses an ε edge into $S \sqcap A$, then leads to a terminal state. If the ε traversal occurs after w_1, then we may replace w_1 with u (it leads to the same state s), hence we get $u.v \in (S \sqcap A)(\Sigma)$, contradiction. So the traversal occurs somewhere strictly in w_1, i.e. $w_1 = w_1'.w_1''$, with $w_1'' \neq \varepsilon$. We have $w = w_1'.w_1''.v$, where w_1' goes through S, then there is the ε traversal, then $w_1''.v$ lead to a terminal state in $S \sqcap A$. Consider the state after w_1'': it must be of the form (s, a_3), with the same s (because S is deterministic). Let $u' \in W_1(s, a_3)$. We show that $u'.w_1''.w_2 \in (S \sqcap A)(\Sigma)$. Indeed u' leads in S to the same state as w_1', hence $u'.w_1''$ leads to the same state s as w_1. From here we follow the ε-transition to (s, a_1), and from here we follow w_2. Hence it is also in $S \sqcap A$. Since the latter is deterministic, the successful path leads through (s, a_3), in short we have shown $w_2 \in W_2(s, a_3)$. But this shows that we could have split $w = w_1.w_2$ with a shorter w_1: namely as $w = w_1'.(w_1''.w_2)$,

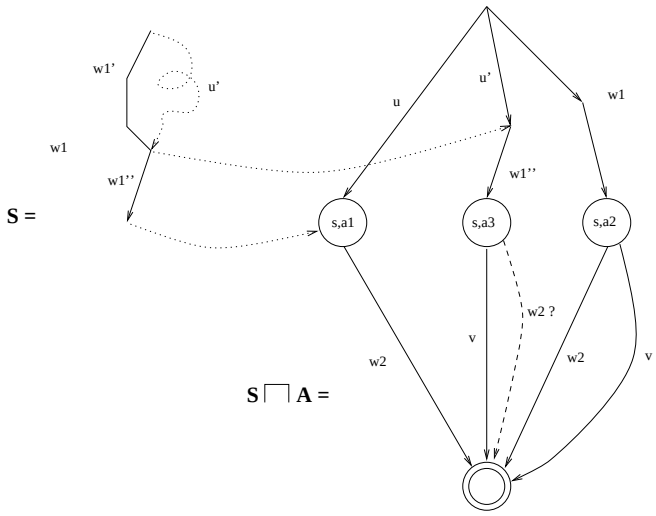

Figure 12: Illustration for the proof of Theorem 4.10.

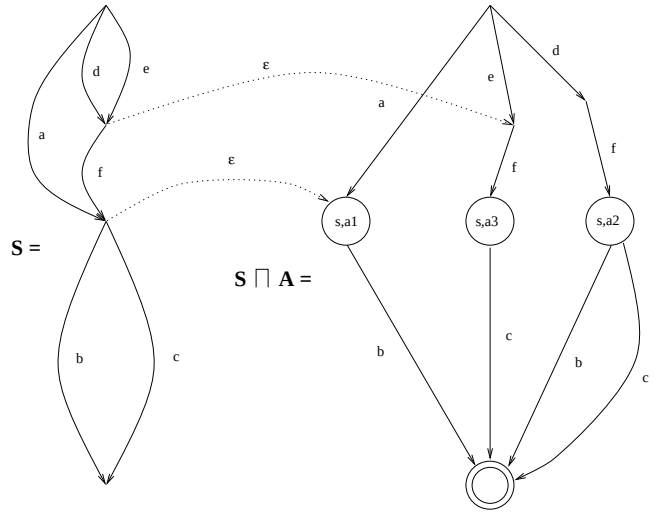

Figure 13: A counterexample for why equality rather than containment is necessary in Theorem 4.10.

with w_1' leading in S to some state, then follow an ε-transition, then follow $w_1''.w_2$. This contradicts our choice of $w_1.w_2$. □

As a comment, we notice that Σ may avoid containing a complete cut only by dropping some states (s, a) for which $W_2(s, a)$ is contained in that of some other state in Σ. It cannot avoid taking (s, a) if $W_2(s, a)$ is contained in the *union* of two or more W_2's applied to states in Σ. This is a subtle consequence of the fact that we don't require only $A \sqcap S \subseteq (S \sqcap A)(\Sigma)$, but to have equality. But without this requirement Σ can be totally unrlated to a cut, as Fig. 13 shows. Here Σ consists of two states (those with ε transitions), and we have $S \sqcap A \subset (S \sqcap A)(\Sigma)$, with $e.f.b \in (S \sqcap A)(\Sigma) - (S \sqcap A)$.

Our restricted PTIME algorithm works on any class of queries whose regular expression admits a determinstic automaton with polynomial size (and, which can be constructed in PTIME). One example of such queries are *restricted* regular queries. Define a *restricted regular expression* to be a regular expression R of the form $P_1.P_2 \ldots P_m$, where $m \geq 0$ and

each P_i is either a predicate or $*$, and in which any two predicates P_i, P_j, $i \neq j$ are either the same, or disjoint (not $(\exists l.P_i(l) \wedge P_j(l))$). Using Morris and Pratt's algorithm [Per90], we can construct for any restricted regular expression an equivalent deterministic automaton whose size is polynomial in the size of the regular expression. To see that, consider first restricted regular expressions which have a single $*$ at the beginning. That is $R = *.P_1.P_2 \ldots P_m$, where $P_1, \ldots, P_m$ are all predicates. Replace each predicate with a distinct letter (i.e. several occurrences of the same predicate will be replaced with the same letter), construct the deterministic Morris and Pratt automaton A, then replace in A the lettters back with their predicates. The resulting automaton is still deterministic, due to our requirement that any two predicates are either disjoint or equal. Such an automaton consists of (1) a chain of states $s_0 \stackrel{P_1}{\rightarrow} s_1 \stackrel{P_2}{\rightarrow} \ldots \stackrel{P_m}{\rightarrow} s_m$, and some back transitions $s_i \rightarrow s_j$ with $j \leq i$ labeled either P_k, for some $k = 1, m$, or $\neg(P_1 \vee \ldots \vee P_m)$ (see [Per90]), where s_1 is the initial state and s_m the unique terminal state. Consider now an arbitrary restricted regular expression R. Since $*.* = *$, R can be rewritten as an alternation of $*$-free blocks spearated by $*$, i.e. $R = R_0.*.R_1.*.R_2.* \ldots *.R_k$ where $R_0, R_1, \ldots, R_k$ are $*$-free, and all are nonempty, possible with the exception of R_0 and R_k. The deterministic automaton for R is obtained as follows. We assume $R_0 = \epsilon$: the other case is treated similarly. First construct the nondeterministic automata $A_1, \ldots, A_k$ for each of $R_1, \ldots, R_k$. For each A_i, $i = 1, k-1$, delete all transitions from the terminal state. Then, construct A by concatenating $A_1, \ldots, A_k$, i.e. collapsing the terminal state of A_{i-1} with the initial state of A_i. The idea is that A will search eagerly for each block R_i (except the last one): as soon as it found one, it will immediately proceed with the next block. This is correct, since any two blocks are separated with $*$.

Define a *restricted regular query* to be a single regular query Q whose regular expression is restricted. We describe now Algorithm *State-Rewrite-Cut*, which solves the query rewriting problem for restricted regular queries. Let A be a deterministic automaton equivalent to R.

Algorithm *State-Rewrite-Cut*

Step 1: Compute all candidate pairs (s, a) for which $(S \sqcap A)(\{(s, a)\}) \subseteq A$. Containment can be tested in PTIME since A is deterministic.

Step 2: For each pair (s, a) define $c(s, a) \stackrel{\text{def}}{=} cost(C, ext(s))$ if (s, a) is a candidate (where C is the pruned atomic canonical query corresponding to (s, a)), and $c(s, a) \stackrel{\text{def}}{=} \infty$ otherwise. Compute a minimum cost cut Σ in $S \sqcap A$ using the algorithm in [GT89].

By Theorem 4.10, *State-Rewrite-Cut* finds the minimum cost canonical solution to Equation (1), whenever one exists, therefore, it is *sound* and *complete*. Obviously, it runs in PTIME.

We don't need to restrict the class of queries in order to run in PTIME. The following algorithm *State-Rewrite-Approx* works on arbitrary single regular queries, as follows.

Algorithm *State-Rewrite-Approx*

Step 1: Compute all candidates (s, a). For the test $(S \sqcap A)(\{s, a\}) \subseteq A$, we use the following PTIME approximation: check whether there exists a simulation from $(S \sqcap A)(\{s, a\})$ to A. This is a sufficient, but not necessary condition.

Step 2: Compute a minimum-cost cut Σ as before. Again, this is only a sufficient, but not necessary condition.

Summarizing, we have:

Theorem 4.11 *Algorithm State-Rewrite-Cut is sound and complete for restricted regular queries, and runs in PTIME. Algorithm State-Rewrite-Approx is sound, but not complete, for arbitrary single regular queries, and runs in PTIME.*

Example 4.12 We illustrate Algorithm *State-Rewrite-Cut* by applying it to Query 1.1 and the schema S in Fig. 2 and show how it derives all plans. The pruned version of Query 1.1 is

$$\begin{aligned}&\mathsf{select}\ p\ \mathsf{where}\ (Other)*.Lab\text{-}or\text{-}Center.(Other)*.\ ((Software\text{-}Dept.(Other)*.Project)\\ &\qquad\qquad |(Math\text{-}Dept.(Other)*.Project)).p\ \mathsf{in}\ DB\end{aligned}$$

and its automaton, $S \sqcap A$, is in Fig. 14. It is deterministic, since we started with a restricted regular expression.

Step 1. We search for candidates, i.e. states (s, a) in $S \sqcap A$ s.t. $(S \sqcap A) \subseteq A$. The figure illustrates this step for $(s10, a4)$. We have $(S \sqcap A)(s10, a4) \subseteq A$ because:

$$\begin{aligned}(S \sqcap A)(\{(s10, a4)\}) &= (Other*).Lab\text{-}or\text{-}Center.(Other*).Software\text{-}Dept.(Other*).Project\\ A &= *.Dept.*.Project\end{aligned}$$

and we assumed that *Software-Dept* $\subseteq$ *Dept*. In the algorithm we would prove the inclusion by finding a simulation from $(S \sqcap A)(\{(s10, a4)\})$ to A: details omitted. In this example all states in $S \sqcap A$ are candidates.

Step 2. Instead of considering all subsets Σ of candidates, we search for a cut in $S \sqcap A$. All cuts are solutions and, according to Theorem 4.10 any solution can be described by a cut. The cuts are (we only consider those which are minimal under set inclusion):

$\{(s1, a1)\}$
$\{(s3, a2)\}$
$\{(s4, a3), (s7, a3)\}$
$\{(s4, a3), (s12, a4)\}$
$\{(s10, a4), (s7, a3)\}$
$\{(s10, a4), (s12, a4)\}$

Of these, the last one has the least cost, hence we obtain the following optimal solution to the rewriting problem:

$$Q(DB) = Q_1(ext(s1)) \cup Q_2(ext(s12))$$

where both $Q_1(x)$ and $Q_2(x)$ are $\mathsf{select}\ y\ \mathsf{where}\ y\ \mathsf{in}\ x$, i.e. do not perform any graph traversal.

5 Related Work

Several query languages [BDHS96, KS95, LSS96, MMM96, PGMU95, QRS+95] support complex path traversals; types of expressions include wildcards (W3QS, WebSQL), regular expressions (Lorel, UnQL), and DATALOG queries (MSL, WebLog).

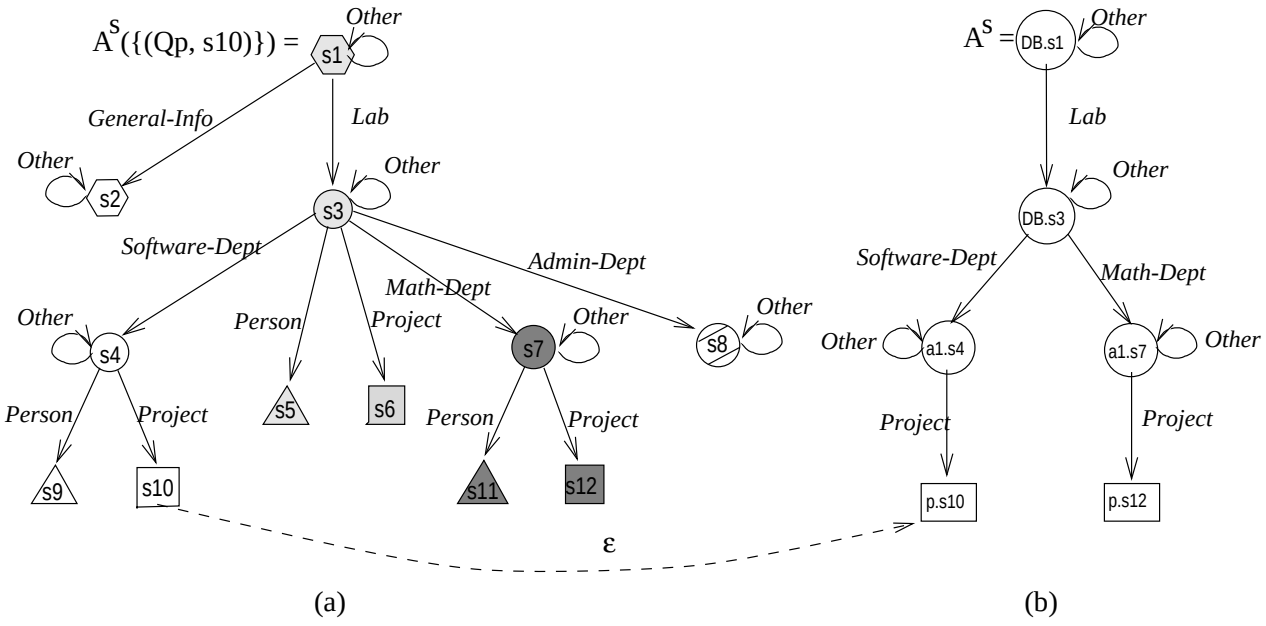

Figure 14: One of the graphs ($A^S(\{(p.s_{10}, s_{10})\})$) constructed in Step 1.

Optimization techniques for *generalized path expressions* (GPE) [CCM96] exist for an object-oriented database [CACS94]. GPEs are restricted to acyclic paths in the schema; this avoids optimization of recursive queries and allows exhaustive instantiation of each GPE. These techniques do not apply to semistructured data or Web sites, because our path expressions may be matched by infinitely many paths in the database, for example, when the database is cyclic.

Abiteboul and Vianu recently proved the decidability of equivalence of regular path expressions with constraints [AV97b]. This is different from our work, because we describe the structure in terms of graph schemas, rather than path constraints.

Regular path expressions can be translated into DATALOG queries over a ternary relation that encodes the labeled graph. Several optimization techniques exist for general DATALOG [BR86, BR87, AHV95] Our techniques are less general (they apply only to regular path expressions), but they are more powerful. For example, as Plan 4 suggests, we can avoid inspecting the facts in the *sufficient set of minimal facts* [BR86], which any optimized DATALOG program must inspect.

Mendelzon and Wood consider the problem of matching regular path expressions in a graph, by applying them to cycle-free paths [MW95]. The data complexity can become NP-complete; the authors describe constraints on graph structure under which the data complexity remains in PTIME. We avoid this problem, because our path expressions are interpreted as arbitrary paths, and therefore, our data complexity is always PTIME.

6 Discussion

Regular path expressions are a useful and necessary feature of query languages for semistructured data, because they support querying of graph structure and content. Naive evaluation of regular queries, however, ignores any structure and may result in exhaustive search, which

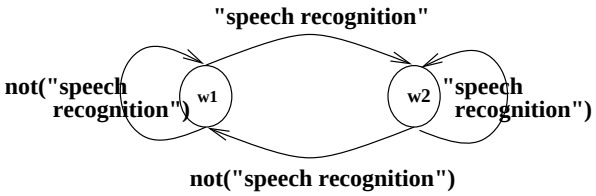

Figure 15: A word-index schema W.

is prohibitively expensive. Our optimization techniques make regular queries feasible, because they use existing structure to substantially reduce and, in some cases, completely eliminate graph traversal. Our techniques can be implemented efficiently for a large class of regular queries. Query pruning applies to any regular query and runs in polynomial time. Query rewriting using state extents is PSPACE in general, but runs in polynomial time for a restricted class of regular queries. Moreover, a polynomial-time, approximation algorithm exists for all single regular queries. Most importantly, both techniques are algebraic and compositional, which permits us to apply them to nested queries.

Both optimizations rely on graph schemas, which specify a graph database's general structure. Our graph schemas are more flexible than traditional schemas, because they do not restrict nor fully describe a graph's structure. Multiple graph schemas can be defined for the same data, and there always exists a canonical, maximal fragment of a database that conforms to a schema [BDFS97]. We believe that specifying graph schemas is natural for semistructured databases. The page and graph structure of Web sites, for example, is often generated automatically by scripts (or, more recently, by queries [FFK+97, FFLS97]) that format pages stored in an underlying database and that allow a site manager to control the "look and feel" and organization of the site's pages. The Web site of AT&T Research is managed in this way. We expect that site-management tools could generate automatically the graph schema for a site.

Our techniques can be combined with a keyword-based search engine to answer queries that consist of both regular path expressions and keyword searches. For example, this query produces all pages reachable from *Dept* edges that contain the phrase "speech recognition":

select p where $*.Dept.*.p$ in DB,
p matches "speech recognition"

To optimize this query, we want to use any word indexes for DB, which can efficiently produce pages that contain a keyword, as well as DB's graph schema S. We observe that a word index can be modeled by a graph schema W; see Fig. 15. State w_2 contains all pages containing the string "speech recognition"; w_1 contains all other pages. We construct the product of DB's schema S and the word index W, $S \sqcap W$. We can rewrite the query using state extents by solving Equation 1 over the combined schema $S \sqcap W$. The resulting plan uses the word index W to retrieve pages directly whenever possible.

Lastly, we report preliminary results of applying our optimizations to regular queries. We have implemented a system that evaluates UnQL [BDHS96] queries on Web site graphs and that implements query pruning. Using our UnQL interpreter, we applied both optimization techniques to a large fragment of the AT&T Research Web site, which contains approximately 9000 nodes and 43,000 edges. We ran some preliminary experiments on three queries (see Fig. 16); each searches for a regular path and then matches one or more keywords in the

Query select p where	# Matching pages		# Edges traversed		
	Keyword only	Keyword & reg. path	Naive eval.	Query pruning	Query rewriting
$*.Dept.*.p$ in DB, p matches $ProjectNames$	140	9	43,000	9947	3848
$*.SysAdminDept.*.p$ in DB, p matches `"security"`	113	6	43,000	617	-
$*.(Project\|People).p$ in DB, p matches `"security"`	113	29	43,000	16584	10552

Figure 16: Results for three simple examples.

target page. The queries were hand-optimized, and the schema, which is similar to that in Fig. 2(a), was hand-computed. We compared the number of pages returned by a keyword-only search with that returned by the complete regular query; this comparison measures the usefulness of regular queries over keyword-only searches. Using keywords only, the first query produces 140 pages that contain one of several project names, and the second and third produce 113 pages that contain the word "security". Restricting the search to a regular path reduces the number of relevant pages significantly. To measure the benefits of our techniques, we compared the number of edges traversed by a naive evaluation of each query and by optimized versions. Naive evaluation requires exhaustive graph search, but both query pruning and query rewriting substantially reduce the number of edges traversed.

Although substantial work is needed to apply these techniques in practice, these preliminary results suggest that regular queries combined with word indices can produce fewer and more precise results than word indices alone, and that both query pruning and query rewriting are effective techniques for reducing the cost of evaluating regular queries.

Acknowledgments We thank H.V. Jagadish for his comments on an earlier version of this paper.

References

[Abi97] Serge Abiteboul. Querying semi-structured data. In *ICDT*, 1997.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley Publishing Co, 1995.

[AV97a] Serge Abiteboul and Victor Vianu. Queries and computation on the web. In *ICDT*, pages 262–275, Deplhi, Greece, 1997. Springer Verlag.

[AV97b] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *Proceedings of ACM Symposium on Principles of Database Systems*, 1997.

[BDFS97] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. Adding structure to unstructured data. In *ICDT*, pages 336–350, Deplhi, Greece, 1997. Springer Verlag.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, 1996.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of ACM SIGMOD Conference on Management of Data*, May 1986.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of 6th ACM Symposium on Principles of Database Systems*, pages 269–283, 1987.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In Richard Snodgrass and Marianne Winslett, editors, *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, May 1994.

[CCM96] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996.

[FFK+97] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. STRUDEL - a web-site management system. In *SIGMOD*, Tucson, Arizona, May 1997.

[FFLS97] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, September 1997.

[GHR95] Raymond Greenlaw, H.James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation. P-Completeness Theory.* Oxford University Press, New York, Oxford, 1995.

[GT89] Harold Gabow and Robert Tarjan. Faster scaling algorithms for network problems. *SIAM Journal of Computing*, 18(5):1013–1036, 1989.

[Imm87] Neil Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[Koz91] Dexter C. Kozen. *The Design and Analysis of Algorithms.* Springer-Verlag, New York, 1991.

[KS95] David Konopnicki and Oded Shmueli. Draft of W3QS: a query system for the World-Wide Web. In *Proc. of VLDB*, 1995.

[LMSS95] Alon Levy, Alberto Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of the 14th Symposium on Principles of Database Systems*, San Jose, CA, June 1995.

[LSS96] Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. A declarative language for querying and restructuring the web. In *Post-ICDE IEEE Workshop on Research Issues in Data Engineering (RIDE-NDS'96)*, New Orleans, February 1996.

[MMM96] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. In *Proceedings of the Fourth Conference on Parallel and Distributed Information Systems*, Miami, Florida, December 1996.

[MW95] A. Mendelzon and P. Wood. Finding regular simple paths in graph databases. *SIAM Journal of Computing*, 24(6):1235–1258, December 1995.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of VLDB*, September 1996.

[Pap94] Christos Papadimitriou. *Computational Complexity.* Addison Wesley Publishing Company, 1994.

[Per90] D. Perrin. Finite automata. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 1, pages 1–57. Elsevier, Amsterdam, 1990.

[PGMU95] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. In *IEEE International Conference on Data Engineering*, pages 132–141, March 1995.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, March 1995.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructure heterogeneous information. In *International Conference on Deductive and Object Oriented Databases*, 1995.

[SM73] L. J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time. In *5th STOC*, pages 1–9. ACM, 1973.

[TMD92] J. Thierry-Mieg and R. Durbin. Syntactic Definitions for the ACEDB Data Base Manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge,CB2 2QH, UK, 1992.