# Optimizing Regular Path Expressions Using Graph Schemas

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* [1]. 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 [25, 24, 23], 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 [27] 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 [29] uses a tree data model to store biological data; the query language UnQL [8] was designed in part for such tree-like data. Web sites can also be viewed as semistructured data [3]. WebSQL [21], W3QS [17], and WebLog [19] were designed to query the Web's graph structure. STRUDEL [11, 12] 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* [7]. 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 [20]. 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 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 [7], 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.

**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 [1]. 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

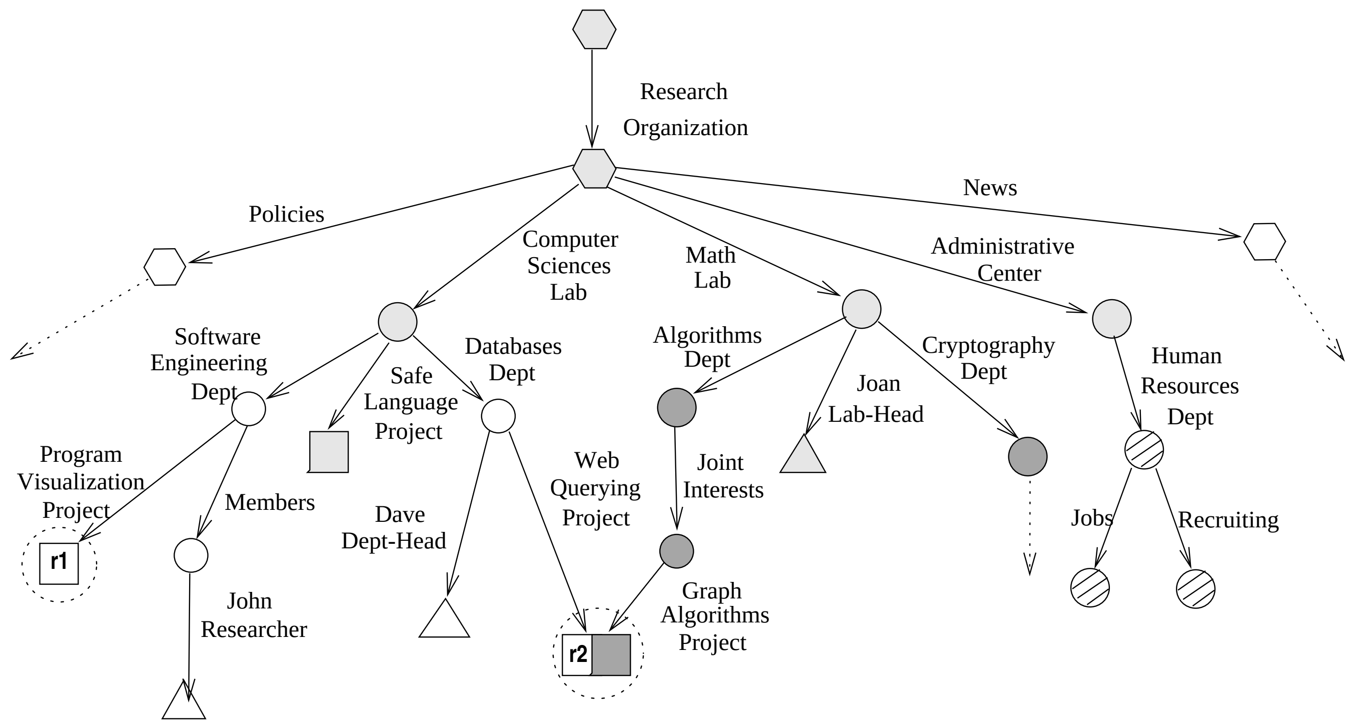



Figure 1: Example Web site of research laboratory.

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:

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

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,

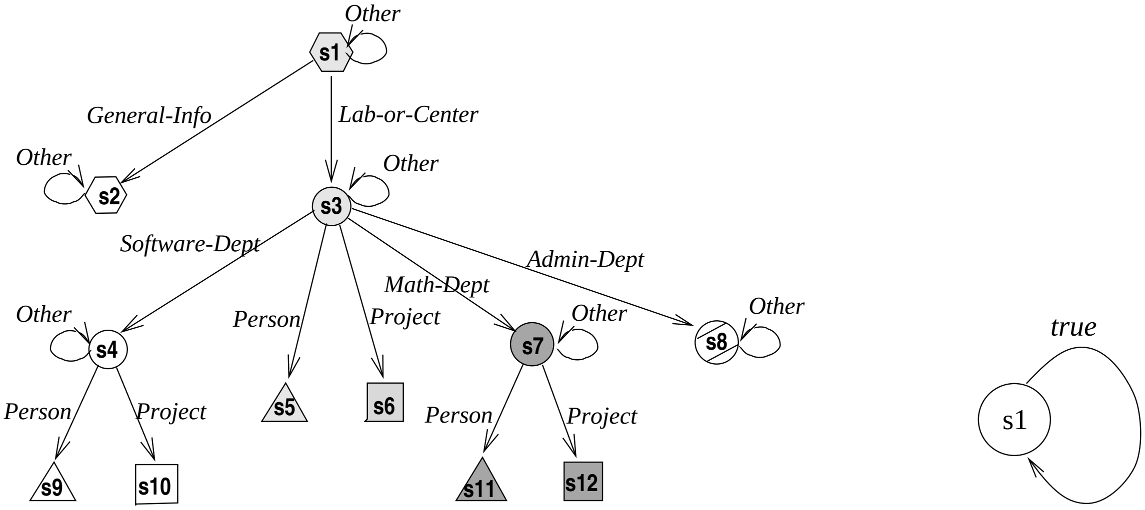
$$\mathsf{select}\ p\ \mathsf{where}\ *.Project.p\ \mathsf{in}\ DB, *.Dept.*.p\ \mathsf{in}\ DB$$

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* [7]. 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. We only consider *deterministic* schemas, which for every state $s$ and any two distinct outgoing edges labeled $P$ and $P'$, $P, P'$ are disjoint, i.e., not $(\exists l.P(l) \wedge P'(l))$.

[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 $DB$.




(a) (b)

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

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 [7]; 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 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 $Q =$ select $y$ where $R.y$ 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

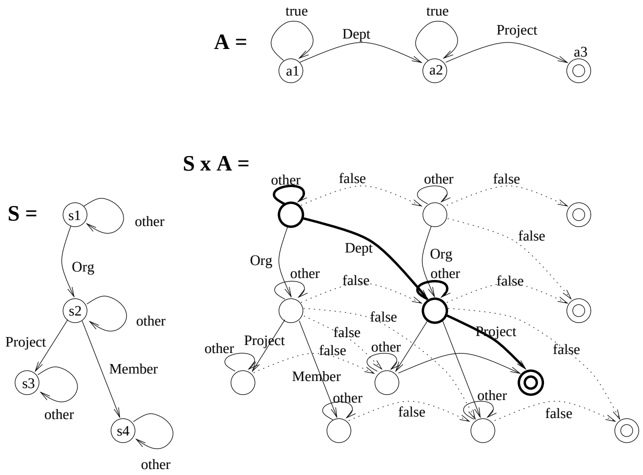



Figure 3: Schema $S$, automaton $A$, and product automaton $S \times A$. The pruned automaton $S \sqcap A$ is in bold. Here $other \stackrel{\text{def}}{=} \text{not } (Org \vee Project \vee Member)$

query $Q^S$ is select $y$ where $R^S.y$ 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 briefly 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 [15, 16]. The intuition for this graph 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 $(s_i, a, A_l) \to (s_i, z, A_l)$, 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 $(s_i, z, A_l) \to (s_i, z)$, for every conjunct $R_l.z$ in $x_l$, and edges $(s_i, z) \to (s_i, a, A_l)$, for every conjunct $R_l.y_l$ in $z$, with $a$ the initial state in $A_l$. We mark the node $(s_1, DB)$ accessible, where $s_1$ is the root state in $S$. Finally, we compute the accessible nodes of the AND/OR graph[2]. 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 3.1** *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 $S$.*

We can show in the case of single regular queries that the pruned query is optimal if we assume a naive query-evaluation strategy that relies only on graph traversal. 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 $S$. 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 3.2** *Let $S$ be a schema and $Q$ a single regular query. Then (1) $Q^S$ is equivalent to $Q$ over $S$, and (2) for any single regular query $Q'$ equivalent to $Q$ over $S$ and any database $DB$ conforming to $S$, $cost(Q^S, DB) \leq cost(Q', DB)$.*

[2] An OR node is accessible if at least some of its predecessors is accessible, while an AND node is accessible if all its predecessors are accessible.

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

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

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:

$$Q(DB) = (\text{select } p \text{ where } p \text{ in } ext(s_{10})) \cup$$
$$(\text{select } p \text{ where } p \text{ in } ext(s_{12}))$$

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:

$$Q(DB) = (\text{select } p \text{ where } *.Project.p \text{ in } ext(s_4)) \cup$$
$$(\text{select } p \text{ where } *.Project.p \text{ in } ext(s_7))$$

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.

In general, we want to find all solutions to Equation (1) and give them to a cost-based optimizer to choose the best plan. Our first insight is that this problem is an instance of the *query rewriting problem* [20] 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 $V_j(DB) \stackrel{\text{def}}{=}$ select $x$ where $T_j.x$ in $DB$. Then, one can show that, for any database $DB$ conforming to $S$, $ext(s_j) = V_j(DB)$. Equation (1) expresses the query $Q(DB)$ in terms of the views $V_1, \ldots, V_n$; since both the views and the query are recursive, this is an instance of the view rewriting problem for recursive queries, on databases conforming to $S$.

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

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

where $R' = (T_{j_1}.R_1 \mid \ldots \mid T_{j_p}.R_p)$. Apparently, we have to check equivalence of two regular expressions: $R \equiv R'$. Checking containment and equivalence of regular expressions is decidable and PSPACE complete [28]. This result remains true in our setting, where regular path expressions contain predicates, provided that certain simple assumptions on the predicates hold [13]. We face two obstacles, however. 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 $R$, 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:

**Proposition 4.1** *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.* $E' \stackrel{\text{def}}{=} \bigcup_{i=1,p} C_i(ext(s_{j_i}))$ *is still a solution and* $cost(E') \leq cost(E)$.

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

The proof is in [13]. In short, this proposition 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 $S_j$.

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

$$Q(DB) = Q_1^{S_{j_1}}(V_1(DB)) \cup \ldots \cup Q_p^{S_{j_p}}(V_p(DB))$$

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

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

### 4.1 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 containment (Step 2). We show here how to reduce their cost in two ways. First, 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 each 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 [26], we can construct for any restricted regular expressions an equivalent deterministic automaton whose size is polynomial in the size of the regular expression. A *restricted regular query* is a single regular query $Q$ whose regular expression is restricted. We will show here that algorithm *State-Rewrite* can be adapted to a sound and complete PTIME algorithm for restricted regular queries, and to a PTIME approximation algorithm for arbitrary single regular queries.

We start by taking 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 $\Sigma$ 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 $\Sigma_i$ 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 $\varepsilon$ 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. 4 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. 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 $\Sigma$ of candidates, and test whether $A \subseteq (S \sqcap A)(\Sigma)$. Step 3: select the minimum cost solution. For example, the pair $(s_2, a_2)$ in Fig. 4 is not a candidate, because $(S \sqcap A)(\{(s_2, a_2)\}) \not\subseteq A$[4]. We describe how to do the first two steps more efficiently.

We need the notion of a "cut" in a graph [18]. 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 de-

[4] In fact $\{(s_1, a_1)\}$ is the only candidate for Query 1.1 and schema $S$ in Fig. 3.

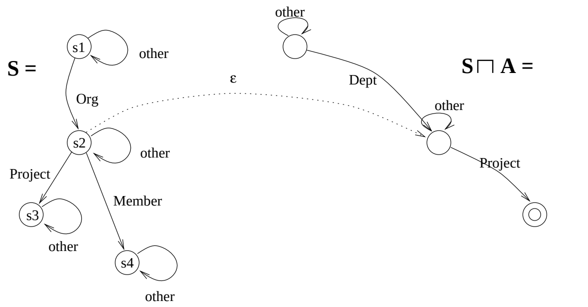



Figure 4: $(S \sqcap A)(\{(s_2, a_2)\})$, for $S$, $A$ defined in Fig. 3.

grees [14]). We can show that if $\Sigma$ is a cut in $S \sqcap A$, then $A \subseteq (S \sqcap A)(\Sigma)$. The converse does not hold in general, but does hold when $A$ is deterministic. We describe next Algorithm *State-Rewrite-Cut*, which solves the query rewriting problem for restricted regular queries. Let $A$ be a deterministic automaton equivalent to $R$. Step 1: compute all candidate pairs $(s, a)$ for which $(S \sqcap A)(\{(s, a)\}) \subseteq A$. This step can be computed in PTIME since $A$ is deterministic. Step 2: for each pair $(s, a)$ define $c(s, a) \stackrel{\text{def}}{=} \mathit{cost}(C, \mathit{ext}(s))$ 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 $\Sigma$ in $S \sqcap A$ using the algorithm in [14]. Obviously, *State-Rewrite-Cut* runs in PTIME and finds the minimum cost canonical solution to Equation (1), whenever one exists, therefore, it is *sound* and *complete*.

Finally, we derive an approximation algorithm, *State-Rewrite-Approx*, which works on arbitrary single regular queries, as follows. 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 $\Sigma$ as before. Again, this is only a sufficient, but not necessary condition. Summarizing, we have:

**Theorem 4.4** *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.*

## 5 Related Work

Several query languages [8, 17, 19, 21, 24, 27] support complex path traversals; types of expressions include wildcards (W3QS, WebSQL), regular expressions (Lorel, UnQL), and DATALOG queries (MSL, WebLog).

Optimization techniques for *generalized path expressions* (GPE) [10] exist for an object-oriented database [9]. 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 [4]. 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 [5, 6, 2] 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* [5], 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 [22]. 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 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.

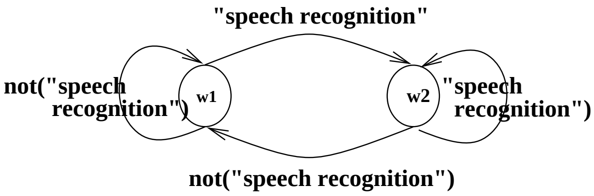



Figure 5: A word-index schema $W$.

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 [7]. 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 [11, 12]) 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. 5. 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 [8] 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. 6); each searches for a regular path and then matches one or more keywords in the 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

[1] S. Abiteboul. Querying semi-structured data. In *ICDT*, 1997.

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

[3] S. Abiteboul and V. Vianu. Queries and computation on the web. In *ICDT*, pp. 262–275, Deplhi, Greece, 1997. Springer Verlag.

[4] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proc. of ACM Symposium on Principles of Database Systems*, 1997.

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

[6] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. of 6th ACM Symposium on Principles of Database Systems*, pp. 269–283, 1987.

| Query<br><br>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 6: Results for three simple examples.


[7] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *ICDT*, pp. 336–350, Deplhi, Greece, 1997. Springer Verlag.

[8] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, 1996.

[9] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In R. Snodgrass and M. Winslett, editors, *Proc. of 1994 ACM SIGMOD Int'l Conf. on Management of Data*, May 1994.

[10] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *SIGMOD*, June 1996.

[11] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. STRUDEL - a Web-site management system. In *SIGMOD*, May 1997.

[12] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, Sept. 1997.

[13] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas (full version), 1997. Manuscript available from `http://www.research.att.com/~{mff,suciu}`.

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

[15] R. Greenlaw, H. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation. P-Completeness Theory.* Oxford University Press, 1995.

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

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

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

[19] L. V. S. Lakshmanan, F. Sadri, and I. 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)*, Feb. 1996.

[20] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. of the 14th Symposium on Principles of Database Systems*, June 1995.

[21] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. In *Proc. of the Fourth Conf. on Parallel and Distributed Information Systems*, Dec. 1996.

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

[23] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proc. of VLDB*, Sept. 1996.

[24] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. In *IEEE Int'l Conf. on Data Engineering*, pp. 132–141, March 1995.

[25] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE Int'l Conf. on Data Engineering*, March 1995.

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

[27] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructure heterogeneous information. In *Int'l Conf. on Deductive and Object Oriented Databases*, 1995.

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

[29] 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.