INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

IMPLEMENTING QUERY PROCESSING USING VIEWS IN SEMISTRUCTURED DATABASES

Ze Hui Liu

A Major Report
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

September 2001

© Ze Hui Liu, 2001

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-64086-8

Canada

Abstract

Implementing Query Processing Using Views in Semistructured Databases

Ze Hui Liu

Since XML's introduction, this new eXtensible Markup Language, has quickly emerged as the universal format for publishing and exchanging data in the World Wide Web. As a result, data sources, including object-relational databases, are now faced with a new class of users: clients and customers who would like to deal directly with XML data rather than being forced to deal with the data source particular schema and query languages. XML is also rapidly becoming popular for representing web data as it brings a finely granulated structure to the web information and exposes the semantics of the web content. In all these web applications including electronic commerce and intelligent agents, view mechanisms are recognized as critical and are being widely employed to represent users' specific interests. Rewriting the user queries using views is a powerful technique in the above described applications, which can be categorized as data integration, data warehousing and query optimization. The current implementation experiments the feasibility of using rewritings for query processing in semistructred data. Experiments showed that computing rewriting is very time efficent even for many views as for example 500 views. The implematation is using automata.

To my family

Acknowledgments

I would like to take this chance to thank my supervisor Dr. Gösta Grahne for his help and kindness. His expertise of the field has give me a great chance to learn. And also I would like to thank Alex Thomo for his help and patience. And also I would like to take this chance to thank Dr. shiri Nematollaah for his suggestions and comments.

Contents

List of Figures viii

1 Introduction 1

1.1 Preamble 1

1.2 Semistructured Databases and Regular Path Queries 5

1.3 Outline of the report 7

2 Rewriting of Regular Path Queries 8

2.1 Introduction 8

2.2 L-Rewriting of Regular Path Queries. 9

2.3 Complexity of Computing the L-Rewriting 11

3 Query Processing in Information Integration Systems 13

3.1 Introduction 13

3.2 Warm-Up 16

3.3 Formal Background 17

3.4 P-Rewriting 20

3.5 Computing the P-Rewriting 22

3.6 Design for Implementation of the P-Rewriting 28

3.7 Source code for Implementation of the P-Rewriting 29

3.8 Experiment for Implementation of the P-Rewriting 44

4 Conclusion and Future work **55**

Bibliography **59**

List of Figures

1 XML as a graph. 4

2 An example of a graph database 6

3 The DFA for the query and the corresponding "view" automaton. . . 10

4 The resulting l-rewriting given by DFA $\overline{B}$. 10

5 An example of a view graph 15

6 View graph S 16

7 The smallest possible databases 17

8 A query and a source collection. 18

9 Visualisation of the proof for Theorem 7. 21

10 A finite transducer T 22

11 Decomposing a "macro" transducer I. 23

12 Decomposing a "macro" transducer II. 23

13 Query automaton and macro transducer. 25

14 Transducers for the substitution and inverse substitution. 25

15 Automaton of the p-rewriting. 25

16 Source collection for Example 4 27

17 architecture diagram 30

18 Performance Analysis Chart 54

Chapter 1

Introduction

1.1 Preamble

The electronic data which until a few years ago was limited to a few scientific and technical areas is now becoming universal. Most People see such data as web document. but these documents. rather than being manually composed. are now increasingly generated automatically from databases. It is possible to publish enormous volumes of data in this way. and we are now starting to see the development of software that extracts structured data from Web pages that were generated to be readable by humans. The emergence of XML (Extended Markup Language) as a standard for data representation on the Web is expected greatly to facilitate the publication of electronic data by providing a simple syntax for data that is both human and machine-readable.

Since its introduction. XML, the eXtended Markup Language, has quickly emerged as the universal format for publishing and exchanging data in the World Wide Web. As a result. data sources, including object-relational databases. are now faced with a new class of users: clients and customers who would like to deal directly with XML data rather than being forced to deal with the data source particular schema and query languages. XML is also rapidly becoming popular for representing web data as it brings a finely granulated structure to the web information and exposes the semantics of the web content.

XML versus HTML. Consider the following example of a common situation in the data exchange of Web. An organization publishes data about books, articles and softwares. The source for this data is a relational database, and the Web pages are generated on demand by invoking an SQL query and formatting its output into HTML. A second organisation wants to obtain some product analyses of this data but has only access to the HTML page(s).One first solution to this problem is to write software to parse the HTML and convert it into a structure suitable for the analysis software. This solution has a serious defect: It is brittle, since a minor (text) formatting change in the source could break the parsing program.

In XML, the schema information is stored with the data. Structured values are called elements, attributes and tags. For instance

```
<person><name>Ullman</name><address>Stanford</address></person>
```

is a well-formed statement in XML. Thus, XML data is self-describing and can naturally model irregularities that cannot be modeled by relational or object-oriented data. For example, data items may have missing elements or multiple occurrences of the same element; Also elements may have atomic values in some data items and structured values in others, and collections of elements can have heterogeneous structure.

XML as a graph. We can consider an XML database to be an edge labelled graph. In this graph the nodes represent the objects in the database and the edges represent the attributes of the objects, or relationships between the objects.

For an example suppose we are given the following XML file.

```
<BOOK bookId="ds">
	<TITLE>Distributed Systems</TITLE>
	<AUTHOR authorId="smith">
		<NAME>Dan Smith</NAME>
		<EMAIL>ds@cs.mcgill.ca</EMAIL>
	</AUTHOR>
	<AUTHOR authorId="bouret">
		<NAME>Emil Bouret</NAME>
		<EMAIL>bouret@alpha.net</EMAIL>
	</AUTHOR>
</BOOK>

<ARTICLE articleId="xml">
	<JOURNAL>ACM J. 2000-12</JOURNAL>
	<TITLE>XML Programming</TITLE>
	<AUTHOR authorId="smith"/>
	<REF refId="ds"/>
</ARTICLE>

<ARTICLE articleId="vb">
	<JOURNAL>PCWorld 2000-12</JOURNAL>
	<TITLE>VBScript Integration</TITLE>
	<AUTHOR authorId="Bouret"/>
	<REF refId="xml"/>
</ARTICLE>

<SOFTWARE>
	<COMPANY>Microsoft</COMPANY>
	<PRODUCT>MsOffice</PRODUCT>
	<SOFTWARE>
		<PRODUCT>MS Access 97</PRODUCT>
	</SOFTWARE>
	<SOFTWARE>
		<PRODUCT>MS PowerPoint 97</PRODUCT>
	</SOFTWARE>
	<SOFTWARE>
		<PRODUCT>Word</PRODUCT>
		<SOFTWARE>
			<PRODUCT>MS Equations</PRODUCT>
			<CATEGORY>Mathematics</CATEGORY>
		</SOFTWARE>
	</SOFTWARE>
</SOFTWARE>

<SOFTWARE>
	<COMPANY>Microsoft</COMPANY>
	<PRODUCT>MsPaint</PRODUCT>
	<CATEGORY>Images</CATEGORY>
</SOFTWARE>

<SOFTWARE>
	<COMPANY>Borland</COMPANY>
	<PRODUCT>Borland C++</PRODUCT>
	<CATEGORY>Programming</CATEGORY>
</SOFTWARE>
```

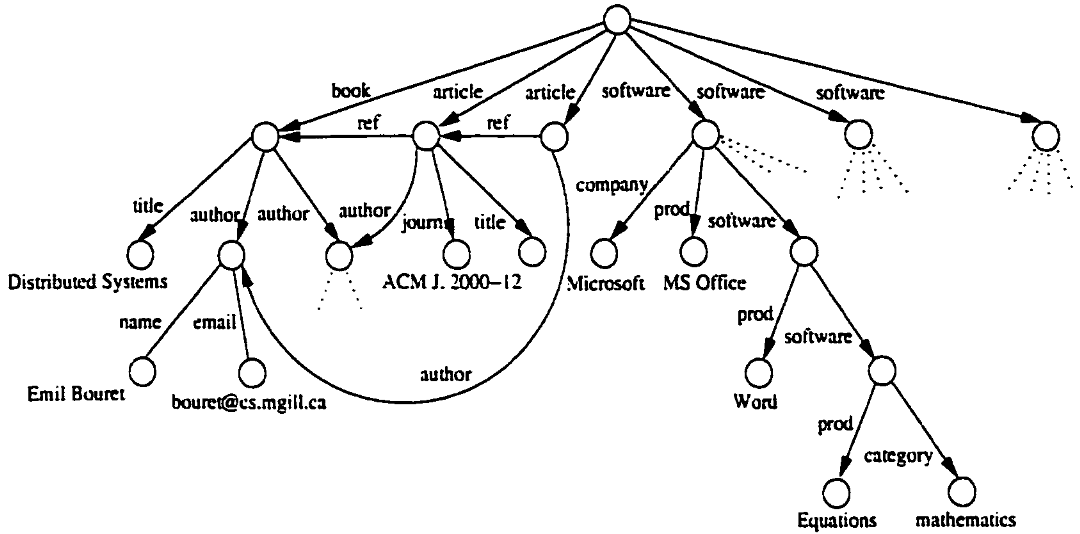


Figure 1: XML as a graph.

Intuitively, this XML database can be represented as the graph of Figure 1. Considering the graph abstraction of XML, instead of XML text, makes it more convenient to study the theoretical database relevant properties and to establish powerful query languages.

1.2 Semistructured Databases and Regular Path Queries

As mentioned before, we abstract the XML data by a data-graph which is the visual representation of the so called *semistructured data model*. Semistructured data is a self-describing collection, whose structure can naturally model irregularities that cannot be captured by standard relational or object-oriented data models [ABS99]. Semi-structured data is usually best formalized in terms of labelled graphs, where the graphs represent data found in many useful applications such as web information systems, XML data repositories, digital libraries, communication networks, and so on.

Formally, let Δ be a finite set, called the *database alphabet*. Elements of Δ will be denoted $R, S, T, R', S', \ldots, R_1, S_1, \ldots$, etc.

Now, assume that we have a universe of objects D. We will denote the objects by $a, b, c, a', b', \ldots, a_1, b_2, \ldots$, and so on. A graph *database* DB over (D, Δ) is a pair (N, E), where $N \subseteq D$ is a set of nodes and $E \subseteq N \times \Delta \times N$ is a set of directed edges labelled with symbols from Δ. Figure 2 illustrates an example of a graph database.

In order to traverse arbitrarily long paths in graph databases, almost all the query languages for semistructured data provide a facility to the user to query through regular path queries, which are queries represented by regular expressions. The design of regular path queries is based on the observation that many of the recursive queries that arise in practice amount to graph traversals. These queries are in essence graph patterns and the answers to the query are subgraphs of the database that match the given pattern [MW95, FLS98, CGLV99, CGLV2000]. For example, the regular path query $(_^* \cdot article) \cdot (_^* \cdot ref \cdot _^* \cdot (ullman + widom))$ specifies all the paths having at some point an edge labelled *article*, followed by any number of other edges then by an edge labelled *ref* and finally by an edge labelled with *ullman* or *widom*.

Formally, we consider a *(user) query* Q to be a finite or infinite regular language over Δ. We denote by $re(Q)$ a regular expression describing the regular language Q.

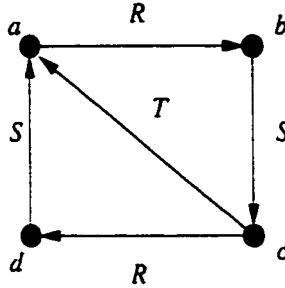


Figure 2: An example of a graph database

If there is a path labelled $R_1, R_2, \ldots, R_k$ from a node a to a node b we write

$$a \stackrel{R_1.R_2 \ldots R_k}{\longrightarrow} b.$$

Let $DB = (N, E)$ be a database and Q be a query. Then the *answer to* Q on DB is defined as

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

Example 1 *For instance, if DB is the graph in Figure 2, and $Q = \{SR.T\}$, then $ans(Q, DB) = \{(b, d), (d, b), (c, a)\}$.*

Query rewriting using views is a well known problem in semistructured data, data integration, data warehousing and query optimization. [LMSS95, Ull97, CGLV99, Lev99]. Simply stated, the problem is: Given a query Q and a set of views $\{V_1, \ldots, V_n\}$, find a representation of Q by means of the views and then answer the query on the basis of this representation.

Query rewriting in relational databases is by now rather well investigated. Several papers investigate this problem for the case of conjunctive queries [LMSS95, Ull97, CSS99, PV99]. These methods are based on query containment and the fact that the number of subgoals in the minimal rewriting is bounded from above by the number of subgoals in the query.

However, in the framework of semistructured data the problem of rewriting queries using views has received much less attention until recently. In this report we identify

some difficulties with currently known methods for using rewritings in semistructured databases and deal with the problem in one realistic scenario.

This senario is related to information integration systems such as the Information Manifold, in which the data sources are modelled as sound views over a global schema. We give in this setting a new rewriting, which we call the *possibility rewriting*, that can be used in pruning the search space when answering queries using views. The possibility rewriting can be computed in time polynomial in the size of the original query and the view definitions. Finally, we show by means of a realistic example that our method can reduce the search space by an order of magnitude.

1.3 Outline of the report

This report will include the theoretical part , a design and implementation of some algorithm as well as the experiment results. The report includes concluding remarks and some future directions.

In theoretical part, it introduce the basic concept of XML as graph database, rewriting the Regular path Queries. P-Rewriting for information intergration systems.

In the experiment part.the design and the source code of the implementation are included. How the experiments were conducted and the performance analysis are also supplied.

We also give the conclusion and some suggestions for the future work at the end of the report.

Chapter 2

Rewriting of Regular Path Queries

2.1 Introduction

It is obvious that a method for rewriting of regular path queries requires a technique for the rewriting of regular expressions, i.e. given a regular expression E and a set of regular expressions $E_1, E_2, ..., E_n$ one wants to compute a function $f(E_1, E_2, ..., E_n)$ which approximates E.

Example 2 *[GT2000] Let* $E = (R+S)^*$ *and* $E_1 = RS$, $E_2 = SR$, $E_3 = R$. *Then the best approximation of* E *using* E_1, E_2 *and* E_3 *is*

$$E' = (E_1 + E_2 + E_3)^*.$$

As far as the author knows, there are two methods for computing such a function f which best approximates E from below. The first one of Conway [Con71] is based on the derivatives of regular expressions introduced by Brzozowski [Brzo64], which provide the ground for the development of an algebraic theory of factorization in the regular algebra [BL80] which in turn gives the methods for computing the approximating function. The second method by Calvanese *et. al.* [CGLV99] is based on automata theory. Both methods compute the same rewriting, which is the largest subset of the query, that can be represented by the views.

In the next section we formalize the problem of query rewriting using views and shortly present the query rewriting proposed by Calvanese *et. al.* [CGLV99], which is called l-rewriting (lower-rewriting) in the sequel. The complexity of computing the l-rewriting is double exponential in the worst case and is discussed at the end of the section.

2.2 L-Rewriting of Regular Path Queries.

Let $\mathbf{V} = \{V_1, \ldots, V_n\}$ be a set of *view definitions*, with each V_i being a finite or infinite regular language over Δ. Associated with each view definition V_i is a view name v_i. We call the set $\Omega = \{v_1, \ldots, v_n\}$ the *outer alphabet*, or *view alphabet*. For each $v_i \in \Omega$, we set $def(v_i) = V_i$. The substitution *def* associates with each view name v_i in the alphabet Ω the language V_i. The substitution *def* is applied to words, languages, and regular expressions in the usual way (see e. g. [HU79]).

A *lower-rewriting* (l-rewriting) of a user query Q using $\mathbf{V}$ is a language Q' over Ω, such that

$$def(Q') \subseteq Q.$$

If for any l-rewriting Q'' of Q using $\mathbf{V}$, it holds that $def(Q'') \subseteq def(Q')$ we say that Q' is *maximal*. If $def(Q') = Q$ we say that the rewriting Q' is *exact*.

Calvanese *et. al.* [CGLV99] have given a method for constructing an l-rewriting Q' from Q and $\mathbf{V}$. Their method is guaranteed to always find the maximal l-rewriting, and it turns out that the maximal l-rewriting is always regular. An exact rewriting might not exist, while a maximal rewriting always exists, although there is no guarantee on the lower bound. For an extreme example, if $\mathbf{V} = \emptyset$, then the maximal rewriting of any query is $\emptyset$.

Their algorithm is as follows:

1. Construct a DFA $A_Q(\Delta, S, s_0, \rho, F)$ such that $Q = L(A_Q)$.

2. Construct an automaton $B = (\Omega, S, s_0, \rho', S - F)$, where $s_j \in \rho'(s_i, v)$ iff $\exists W \in V$ such that $s_j \in \rho^*(s_i, W)$.

3. The maximal l-rewriting is the Ω language accepted by complementing the B automaton.

Step 2. says: *Consider each pair of states. If they are connected in A_Q by a* **walk** *labeled with a word in V_i, put an* **edge** *v_i between them in B.*

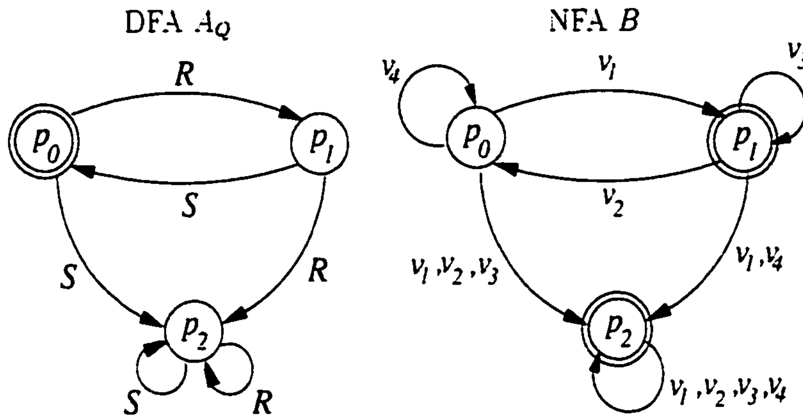
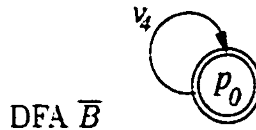


Figure 3: The DFA for the query and the corresponding "view" automaton.

Example 3 *Let $Q = (RS)^*$ be the regular path query and $V_1 = R + S^2$, $V_2 = S$, $V_3 = SR$, $V_4 = (RS)^2$ the regular path views. Then the minimal*[1] *DFA A_Q for the query Q is given in Figure 3, left and the corresponding B automaton with view symbols is given in in Figure 4, right. The resulting complement automaton $\overline{B}$ is given in Figure 4. Note that the "garbage" and unreachable states have been removed for clearness.*



DFA $\overline{B}$

Figure 4: The resulting l-rewriting given by DFA $\overline{B}$.

[1]The constructed DFA for the query does not need to be minimal.

Observe that, if B accepts an Ω-word $v_1 \cdots v_n$, then there exist n Δ-words $W_1, \ldots, W_n$ such that $W_i \in V_i$ for $i = 1, \ldots, n$ and such that the Δ-word $W_1 \ldots W_n$ is rejected by A_Q. On the other hand if there exists a Δ-word $W_1 \ldots W_n$ that is rejected by A_Q such that $W_i \in V_i$ for $i = 1, \ldots, n$, then the Ω-word $v_1 \cdots v_n$ is accepted by B. That is B accepts an Ω-word $v_1 \cdots v_n$ if and only if there is a Δ-word in $def(\{v_1 \cdots v_n\})$ that is rejected by A_Q. Hence, $\overline{B}$ being the complement of B accepts an Ω-word if and only if all Δ-words $W = W_1 \ldots W_n$ such that $W_i \in V_i$ for $i = 1, \ldots, n$, are accepted by A_Q.

2.3 Complexity of Computing the L-Rewriting

Unfortunately, the complexity of computing the l-rewriting of a regular path query Q given a set $\mathbf{V} = \{V_1, \ldots, V_n\}$ of view definitions is very high as the following results in [CGLV99] show.

Theorem 1 *The problem of generating the Ω-maximal rewriting of a regular path query with respect to a set $\mathbf{V} = \{V_1, \ldots, V_n\}$ of regular path view definitions is in EXPTIME.*

In order to use the l-rewriting Q' of a query Q alone for query answering, the rewriting should be algebraically exact (see [CGLV99]), i.e. $def(Q') = Q$. So before talking how the rewritings can be utilized for answering the query (Chapter 3), let us shortly show an optimal algorithm for exactness testing, which is presented in [CGLV99].

Their algorithm for testing if an Ω rewriting of query Q is exact is as follows.

1. Construct an automaton $B = (\Delta, S_B, s_{B0}, \rho_B, F_B)$ that accepts $def(Q')$, by replacing each edge labeled by v_i in the automaton for Q', say $A_{Q'}$, by an automaton A_i such that $L(A_i) = def(v_i)$ for $i = 1, \ldots, n$. Each edge labeled by v_i is replaced by a fresh copy of A_i. We assume without loss of generality,

that A_i has unique start state and accepting state, which are identified with the source and target of the edge respectively. Observe that, since Q' is an l-rewriting of Q, $L(B) \subseteq Q = L(A_Q)$.

2. Check whether $L(A_Q) \subseteq L(B)$, that is, check whether $L(A_Q \cap \overline{B}) = \emptyset$.

Theorem 2 *The l-rewriting Q' is an exact rewriting of the query Q with respect to a set* **V** *of regular view definitions, if and only if $L(A_Q \cap \overline{B}) = \emptyset$.*

Corollary 1 *An exact rewriting of Q with respect to* **V** *exists if and only if $L(A_Q \cap \overline{B}) = \emptyset$.*

Theorem 3 *The problem of verifying the existence of an exact rewriting of a regular path query Q with respect to a set* **V** *of regular view definitions, is in 2EXPSPACE.*

Observe that, if we construct $L(A_Q \cap \overline{B}) = \emptyset$., we get a cost of 3EXPTIME, since $\overline{B}$ is of triply exponential size with respect to the size of the input. However, we can construct $\overline{B}$ "on the fly"; whenever the non-emptiness algorithm wants to move from a state s_1 of the intersection of A_Q and $\overline{B}$ to a state s_2, the algorithm guesses s_2 and checks that it is directly connected to s_1. Once this has been verified, the algorithm can discard s_1. Thus, at each step the algorithm needs to keep in memory at most two states and there is no need to generate all of $\overline{B}$ at any single step of the algorithm.

In [CGLV99] it is shown that the complexity bounds established in the previous theorems are essentially optimal.

Theorem 4 *The problem of checking whether there is a non-empty rewriting of a regular path query Q with respect to a set* **V** *of regular view definitions, is EXPSPACE-complete.*

Note that Theorem 4 implies that the upper bound established in Theorem 1 is essentially optimal. If we can generate maximal rewritings in, say, EXPTIME, then we could test emptiness in PSPACE, which is impossible by Theorem 4.

Chapter 3

Query Processing in Information Integration Systems

3.1 Introduction

Much of the work on answering queries using views has been spurred because of its application to data integration systems. A data integration system provides a uniform query interface to a multitude of autonomous hetereogeneuos data sources. Prime examples of data integration systems include enterprise integration, querying multiple sources in the World Wide Web, and integration of data from distributed scientific experiments. The sources in such an application may be traditional databases, legacy systems, or even structured files. The goal of data integration is to free the user from having to find the data sources relevant to the query, interact with each source in isolation, and manually combine data from the different sources.

To provide a uniform interface, a data integration system exposes to the user a *mediated schema*. A mediated schema is set of virtual relations, in the sense that they are not stored anywhere. The mediated schema is designed manually for a particular data integration application. To be able to answer the queries the system must also contain a set of *source descriptions* that specify the contents of the data sources.

One of the various approaches that has been adopted in several systems, is to describe the contents of a source as a *view* over the mediated schema. In order to answer a query, a data integration system needs to translate a query formulated on the mediated schema into one that refers directly to the schemas in data sources. Since the contents of the data sources are described as views, the translation problem amounts to finding a way to answer a query using a set of views.

We illustrate the problem with the following example, where the mediated schema exposed to the user is our bookstore database graph of Figure 1 with the binary relations specified by the edge labels in the graph.

Suppose, we have the following three data sources. The first provides us a listing of each pair (x, y) of objects such that x is an article that refers to the article or book y. This source can be described by the following view definition.

$$\text{ArticleRefArticle: } ref.$$

The second source is supposed to contain all the pairs (x, y) of objects such that x is a book and y is either a book, article or software referred to, directly or indirectly, in the book. This source can be described by the following view definition.

$$\text{BookRefs: } ref^* + ref^*.software.$$

And the third source contains all the pairs (x, y) of objects such that y is a sub-software of x. This source can be described by the following view definition.

$$\text{SoftwareAndSubs: } software^*$$

If we were to ask now, "which objects are somehow related to some other objects", and we have only the contents of the above data sources available, then we would be able to answer this query using the following regular expression.

$$\text{Q: } ArticleRefArticle^* + BookRefs.SoftwareAndSubs + SoftwareAndSubs.$$

It is important here to note that this formulation of the query is to be answered against the so called "view graph" which consists of nodes representing the objects

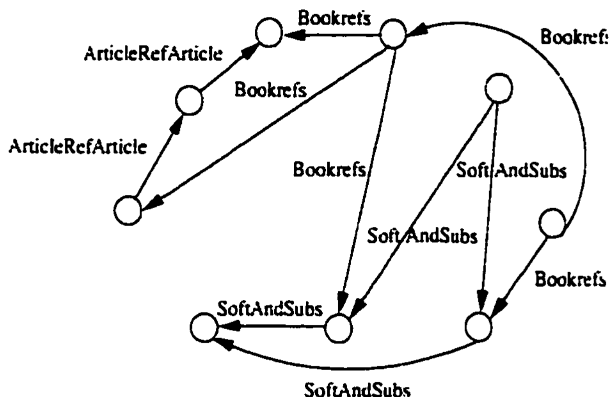


Figure 5: An example of a view graph

found in the data sources and edges labeled with view names. An edge labeled for example with *BookRef* between the objects x and y shows that the pair (x, y) is in the data source described by the second view definition. An example of a view graph is given in Figure 5.

In the previous chapter we showed what is a lower or contained rewriting of a regular path query with respect to a set of regular view definitions, and how to compute a maximal l-rewriting. Posed in the framework of [GM99], we show that the answer to the query that we get if we use the (maximal) l-rewriting only, is a (sometimes strict) subset of the *certain rewriting*. If we want to be able to produce the complete certain answer, the only alternative left is then to apply an extremely intractable decision procedure of Calvanese *et. al.* [CGLV2000] for *all* pairs of objects (nodes) found in the views. One of the contributions of this thesis is an algorithm for computing a regular rewriting that will produce a superset of the certain answer. The use of this rewriting in query optimization is that it restricts the space of possible pairs needed to be fed to the decision procedure of Calvanese *et. al.* We show by means of a realistic example that our algorithm can reduce the number of candidate pairs by an order of magnitude.

The outline of the chapter is as follows. In Section 3.3 we formalize the problem of query rewriting using views in a commonly occurring setting, proposed in information integration systems, in which the data sources are modeled as sound views over a global schema. We give some results about the applicability of previous work in this setting, and discuss further possibilities of optimization. At the end of Section 3.3 we

give an algorithm for utilizing simultaneously the "subset" and a "superset" rewriting in query answering using views. In Section 3.4 we present our main results. First we give an algebraic characterization of a rewriting that we call the *possibility rewriting* and then we prove that the answer computed using this rewriting is a superset of the certain answer of the query, even when algebraically the rewriting does not contain the query. Section 3.5 is devoted to the computation of the possibility rewriting. It amounts to finding the transduction of a regular language and we give the appropriate automata-theoretic constructions for these computations.

3.2 Warm-Up

Let the user query be $Q = R_3R_4 + R_3R_5 + R_1R_4 + R_2R_5$ and suppose that we have the following data sources available.

$$\begin{array}{rclcrcl} V_1 & = & R_1 & & ext(V_1) & = & \{(a,b)\} \\ V_2 & = & R_2 + R_3 & \text{with} & ext(V_2) & = & \{(a,b)\} \\ V_3 & = & R_4 + R_5 & & ext(V_3) & = & \{(b,c)\} \end{array}$$

These data sources can be graphically represented as the view-graph[1] S in Figure 6. It is easy to see that, the only *contained rewriting* of Q using the views is $Q' = \emptyset$,

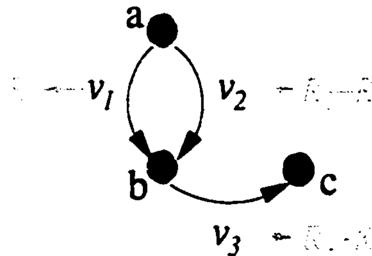


Figure 6: View graph S

and therefore $ans(Q', S) = \emptyset$.

[1]We will use interchangebly the terms "view graph" and "source collection" throughout this thesis.

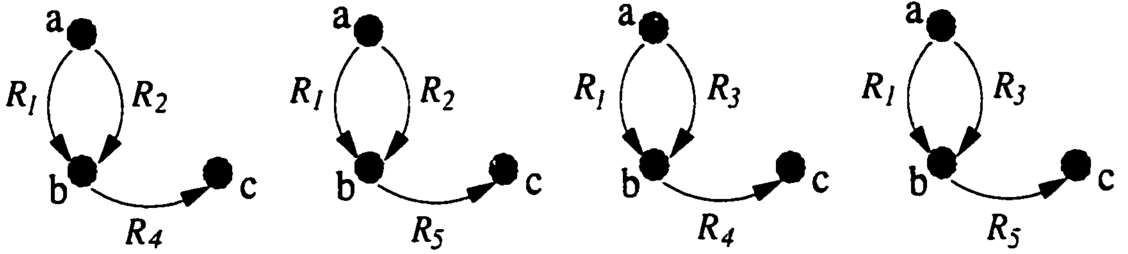


Figure 7: The smallest possible databases

However, what about the rewriting

$$\begin{aligned} Q'' &= V_1V_3 + V_2V_3, \text{ with} \\ ans(Q'', S) &= \{(a, c)\}. \end{aligned}$$

If we examine all the *possible databases* we can see that the pair (a, c) always belongs to the answer of the query (see Figure 7). We say that the pair (a, c) is a *certain answer* to the query, and observe that it is contained in $ans(Q'', S)$.

3.3 Formal Background

Views and answering queries using views. Let $\Omega = \{v_1, \dots v_n\}$ be the view alphabet and let $\mathbf{V} = \{V_1, \dots, V_n\}$ be a set of view definitions as before. Then a *source collection* S over $(\mathbf{V}, \Omega)$ is a database over (D, Ω). A source collection S defines a set $poss(S)$ of databases over (D, Δ) as follows (cf. [GM99]):

$$poss(S) = \{DB : S \subseteq \bigcup_{i \in \{1, \dots, n\}} \{(a, v_i, b) : (a, b) \in ans(V_i, DB)\}\}.$$

Suppose now the user gives a query Q in the database alphabet Δ, but we only have a source collection S available. This situation is the basic scenario in information integration (see e.g. [Ull97, LMSS95, GM99]). The best we can do is to approximate Q by

$$\bigcap_{DB \in poss(S)} ans(Q, DB).$$

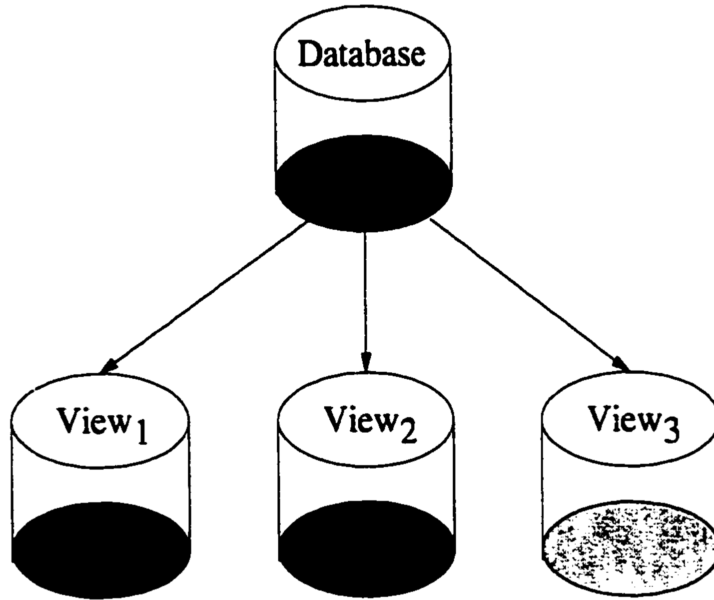


Figure 8: A query and a source collection.

This approximation is called the *certain answer for Q using S*. Calvanese *et. al.* [CGLV2000], in a follow-up paper to [CGLV99] describe an algorithm $\mathcal{A}_{Q,S}(a,b)$ that returns "yes" or "no" depending on whether given pair (a,b) is in the certain answer for Q or not. This problem is coNP-complete in the number of objects in S (data complexity), and if we are to compute the certain answer, we need to run the algorithm for *every* pair of objects in the source collection. A brute force implementation of the algorithm runs in time exponential in the number of objects in S. From a practical point of view it is thus important to invoke algorithm $\mathcal{A}_{Q,S}$ for as few pairs as possible.

Restricting the number of input pairs is not considered by Calvanese *et. al.* Instead they briefly discuss the possibility of using rewritings of regular queries in answering queries using views. Since rewritings have proved to be highly successful in attacking the corresponding problem for relational databases [Lev99], one might hope that the same technique could be used for semistructured databases. Indeed, when the exact rewriting of a query Q using V exists, Calvanese *et. al.* show that, under the "exact view assumption" the rewriting can be used to answer Q using S. Unfortunately, under the more realistic "sound view assumption[2] " adopted in this

[2] If all views are relational projections, the exact view assumption corresponds to the pure universal relation assumption, and the sound view assumption corresponds to the weak instance assumption. For an explanation of the relational assumptions, see [Var88].

chapter we are only guaranteed to get a subset of the certain answer. The following propositions hold:

Theorem 5 *Let Q' be an l-rewriting of Q using* $\mathbf{V}$. *Then for any source collection S over* $\mathbf{V}$,

$$ans(Q', S) \subseteq \bigcap_{DB \in poss(S)} ans(Q, DB).$$

Proof. Let $(a, b) \in Q'(S)$ and let DB be an arbitrary database in $poss(S)$. Since $(a, b) \in Q'(S)$ there exist objects $c_{i_1} \ldots c_{i_k}$ and a path $a\, v_{i_1} c_{i_1} \ldots c_{i_k} v_{i_{k+1}} b$ in S such that $v_{i_1} \ldots v_{i_{k-1}} \in Q'$. Since $DB \in poss(S)$, there must be a path $a\, W_{i_1} c_{i_1} \ldots c_{i_k} W_{i_{k+1}} b$ in DB, where $W_{i_j} \in def(v_{i_j})$, for $j \in \{1, \ldots, k+1\}$. Furthermore we have that $W_{i_1} \ldots W_{i_k} \in def(Q') \subseteq Q$. In other words, $(a, b) \in ans(Q, DB)$. ■

Theorem 6 *There is a query Q and a set of view definitions* $\mathbf{V}$, *such that there is an exact rewriting Q' of Q using* $\mathbf{V}$, *but for some source collections S, the set $ans(Q', S)$ is a proper subset of $\bigcap_{DB \in poss(S)} ans(Q, DB)$.*

The data-complexity for using the rewriting is NLOGSPACE, which is a considerable improvement from coNP. There is an EXSPACE price to pay though. At the compilation time finding the rewriting requires exponential amount of space measured in the size of the regular expressions used to represent the query and the view definitions (expression complexity). Nevertheless, it usually pays to sacrifice expression complexity for data complexity. The problem however is that the l-rewriting is guaranteed only to produce a subset of the certain answer. We would like to avoid running the testing algorithm $\mathcal{A}_{Q,S}$ for all other pairs of objects in S.

In the next section we describe a "possibility" rewriting (p-rewriting) Q'' of Q using $\mathbf{V}$, such that for all source collections S:

$$ans(Q'', S) \supseteq \bigcap_{DB \in poss(S)} ans(Q, DB).$$

The p-rewriting Q'' can be used in optimizing computation of the certain answer as follows:

1. Compute Q' and Q'' from Q using $\mathbf{V}$.

2. Compute $ans(Q', S)$ and $ans(Q'', S)$. Output $ans(Q', S)$

3. Compute $\mathcal{A}_{Q,S}(a, b)$, for each $(a, b) \in ans(Q'', S) \setminus ans(Q', S)$. Output those pairs (a, b) for which $\mathcal{A}_{Q,S}(a, b)$ answers "yes."

3.4 P-Rewriting

As discussed in the previous section, the rewriting Q' of a query Q is only guaranteed to be a contained rewriting. From Propositions 5 and 6 it follows that if we use Q' to evaluate the query, we are only guaranteed to get a subset of the certain answer (recall that the certain answer itself is an approximation from below). In this section we will give an algorithm for computing a rewriting Q'' that satisfies the relation $ans(Q'', S) \supseteq \bigcap_{DB \in poss(S)} ans(Q, DB)$. Our rewriting is related to the inverse substitution of regular languages and as consequence it will be a regular language.

Definition 1 Let L be a language over Ω^*. Then L is a *p-rewriting* of a query Q, using $\mathbf{V}$, if for all $v_{i_1} \ldots v_{i_m} \in L$, there exists $W_{i_1} \ldots W_{i_m} \in Q$ such that $W_{i_j} \in def(v_{i_j})$, for $j \in \{1, \ldots, m\}$, and there are no other words in Ω^* with this property.

The intuition behind this definition is that we include in the p-rewriting all the words in the view alphabet Ω, such that their substitution by *def* contains a word in Q. The p-rewriting has the following desirable property[GT2000]:

Theorem 7 *Let Q'' be a p-rewriting of Q using $\mathbf{V}$. Then*

$$ans(Q'', S) \supseteq \bigcap_{DB \in poss(S)} ans(Q, DB),$$

for any source collection S.

Proof. Assume that there exists a source collection S and a pair

$$(a, b) \in \bigcap_{DB \in poss(S)} ans(Q, DB),$$

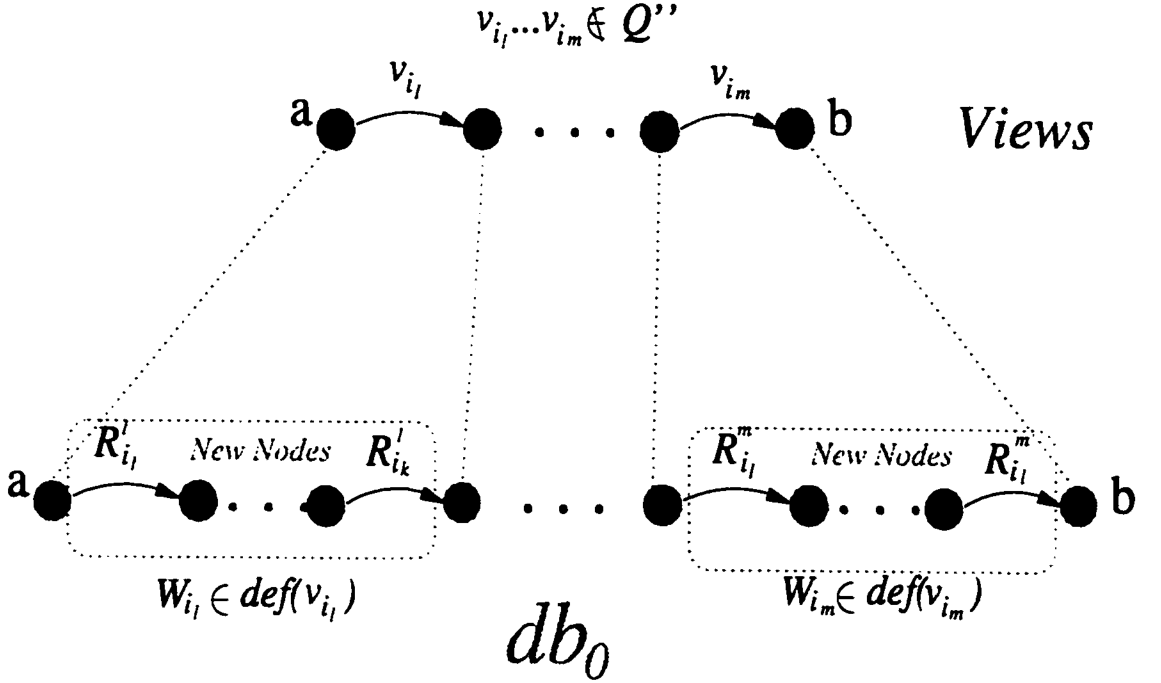


Figure 9: Visualisation of the proof for Theorem 7.

such that $(a, b) \notin ans(Q'', S)$. Since the pair (a, b) is in the certain answer of the query Q, it follows that for each database $DB \in poss(S)$ there is a path $a \xrightarrow{W} b$, where $W \in Q$. Now, we will construct from S a database DB_S such that $ans(Q, DB_S) \not\ni (a, b)$. For each edge labelled v_i from one object x to another object y in S we choose an arbitrary word $W_i \in def(v_i)$ and put in DB_S the "new" objects $c_1, \ldots, c_{k-1}$, where k is the length of W_i, and a path $x, c_1, \ldots, c_{k-1}, y$ labelled with the word W_i. Each time we introduce "new" objects, so all the constructed paths are disjoint. Obviously, $DB_S \in poss(S)$. It is easy to see that $ans(Q, DB_S) \not\ni (a, b)$ because otherwise there would be a path $v_{i_1} \ldots v_{i_m}$ in S from a to b such that $def(v_{i_1} \ldots v_{i_m}) \cap Q \neq \emptyset$, that is $v_{i_1} \ldots v_{i_m} \in Q''$ and $(a, b) \in ans(Q'', S)$. From the fact that $ans(Q, DB_S) \not\ni (a, b)$ it then follows that $\bigcap_{DB \in poss(S)} ans(Q, DB) \not\ni (a, b)$; a contradiction. For a visulalisation of the proof see Figure 9. ■

It is worth noting here that the Theorem 7 shows that $ans(Q'', S)$ contains the certain answer to the query Q even when algebraically $def(Q'') \not\supseteq Q$.

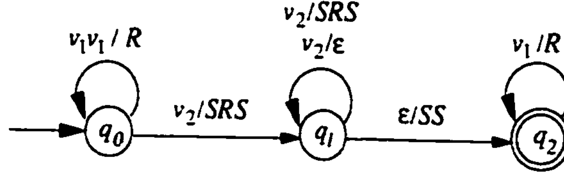


Figure 10: A finite transducer T

3.5 Computing the P-Rewriting

Recall that the definition of a view name $v_i \in \Omega$ is a regular language $def(V_i)$ over Δ. Thus def is in effect a *substitution* from Ω to 2^{Δ^*}. The *inverse* of this substitution is defined by, for each $W \in \Delta^*$,

$$def^{-1}(W) = \{U \in \Omega^* : W \in def(U)\}.$$

It is now easy to see that a p-rewriting Q'' of Q using **V** equals $def^{-1}(Q)$. This suggests that Q'' can be computed using finite transducers.

A *finite transducer* (see e.g. [Yu97]) $T = (S, I, O, \delta, s, F)$ consists of a finite set of states S, an input alphabet I, and output alphabet O, a starting state s, a set of final states F, and a transition-output relation $\delta \subseteq S \times I^* \times S \times O^*$. An example of a finite transducer $(\{q_0, q_1, q_2\}, \{v_1, v_2\}, \{R, S\}, \delta, \{q_2\})$ is shown in Figure 10.

Intuitively, for instance $(q_0, v_2, q_1, SRS) \in \delta$ means that if the transducer is in state q_0 and reads word v_2, it can go to state q_1 and emit the word SRS. For a given word $U \in I^*$, we say that a word $W \in O^*$ is an *output of T for U* if there exists a sequence $(s, U_1, q_1, W_1) \in \delta$, $(q_1, U_2, q_2, W_2) \in \delta$, ..., $(q_{n-1}, U_n, q_n, W_n) \in \delta$ of state transitions of T, such that $q_n \in F$, $U = U_1 \ldots U_n \in I^*$, and $W = W_1 \ldots W_n \in O^*$. We write $W \in T(U)$, where $T(U)$ denotes the set of all outputs of T for the input word U. For a language $L \subseteq I^*$, we define $T(L) = \bigcup_{U \in L} T(U)$.

A finite transducer $T = (S, I, O, \delta, s, F)$ is said to be in the *standard form*, if δ is a relation in $S \times (I \cup \{\epsilon\}) \times S \times (O \cup \{\epsilon\})$. Intuitively, the standard form restricts the input and output of each transition to be only a single letter or ϵ. It is known that any finite transducer is equivalent to a finite transducer in the standard form (see [Yu97]).

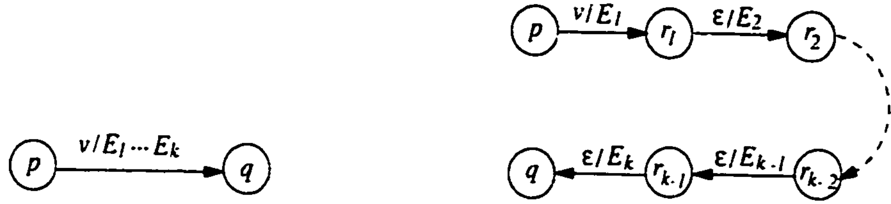


Figure 11: Decomposing a "macro" transducer I.

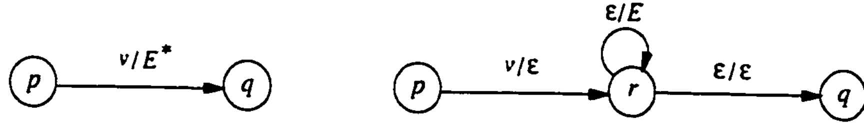


Figure 12: Decomposing a "macro" transducer II.

From the above definitions, it is easy to see that a substitution can be characterized by a finite transducer. Start with one node representing both the starting state and the final state. Then build a "macro-transducer" by putting a self-loop corresponding to each $v_i \in \Omega$ on the sole state. In each such self-loop we first have the view symbol v_i as input and a regular expression representing the substitution of v_i as output. After that, we transform the "macro-transducer" into an ordinary one in standard form. The transformation is done by applying recursively the following three steps. First, replace each edge $v/(E_1 + \ldots + E_n)$, $n \geq 1$, by the n edges $v/E_1, \ldots, v/E_n$. Second, for each edge of the form $v/E_1 \ldots E_k$ from a node p to a node q (Figure 11, left), we introduce $k-1$ new nodes $r_1, \ldots r_{k-1}$ and replace the edge $v/E_1 \ldots E_k$ by the edges v/E_1 from p to r_1, ϵ/E_2 from r_1 to $r_2, \ldots, \epsilon/E_k$ from r_{k-1} to q (Figure 11, right). Third, we get rid of "macro-transitions" of the form v/E^*. Suppose we have an edge labelled v/E^* from p to q in the "macro-transducer." (See Figure 12, left). We introduce a new node r and replace the edge v/E^* by the edges v/ϵ from p to r, ϵ/E from r to r, and ϵ/ϵ from r to q, as shown in Figure 12, right.

By interchanging the input and output of the finite transducer, we see that the inverse of a substitution can also be characterized by a finite transducer.

We now describe an algorithm that given a regular language L and finite transducer T constructs a finite state automaton that accepts the language $T(L)$. Let $A = (P, I, \delta_A, s_0, F_A)$ be an ϵ-free NFA that accepts L, and let $T = (S, I, O, \delta_T, p_0, F_T)$

be a transducer in standard form. We construct an NFA:

$$\mathcal{A} = (P \times S, O, \delta, (p_0, q_0), F_A \times F_T),$$

where δ is defined by,

$$\begin{aligned}
\delta &= \{((p,q), v, (p',q')) : (p,R,p') \in \delta_A \text{ and } (q,R,q',v) \in \delta_T\} \\
&\cup \{((p,q), v, (p,q')) : (q,\epsilon,q',v) \in \delta_T\}
\end{aligned}$$

Theorem 8 *The automaton $\mathcal{A}$ accepts exactly the language $T(L)$.*

Collecting the results together, we now have the following methodology.

Corollary 2 *Let $\mathbf{V} = \{V_1, \ldots, V_n\}$ be a set of view definitions, such the $def(v_i) = V_i$, for all $v_i \in \Omega$, and let Q be a query over Δ. Then there is an effectively characterizable regular language Q'' over Ω that is the p-rewriting of Q using $\mathbf{V}$.* ∎

Example 4 Let the query be $Q = \{(RS)^n : n \geq 0\}$ and the views be v_1, v_2, v_3,and v_4, where $def(v_1) = \{R, SS\}$, $def(v_2) = \{S\}$, $def(v_3) = \{SR\}$, and $def(v_4) = \{RSRS\}$. The DFA[3] $\mathcal{A}$ accepting the query Q is given in Figure 13 (left), and the transducer characterizing the substitution *def* is given in Figure 13 (right). We transform the transducer into standard form (Figure 14, left), and then interchange the input with output to get the transducer characterizing the inverse substitution (Figure 14, right). The constructed automaton $\mathcal{A}$ is shown in Figure 15, where $r_0 = (p_0, q_0)$, $r_1 = (p_1, q_0)$, $r_2 = (p_0, q_2)$ and the inaccessible and garbage states have been removed.

Our algorithm computes the p-rewriting Q'' represented by $(v_4 + v_1 v_3^* v_2)^*$, and the algorithm of Calvanese *et. al.* [CGLV99] computes the l-rewriting Q' represented by v_4^*. Suppose that the the source collection $\mathcal{S}_n$ is induced by the following set of labelled edges:

$$\{(i, v_1, a_i) : 1 \leq i \leq n-1\} \cup$$

[3] An ϵ-free NFA would do as well.

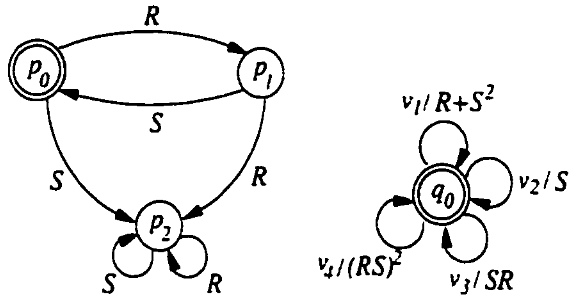


Figure 13: Query automaton and macro transducer.

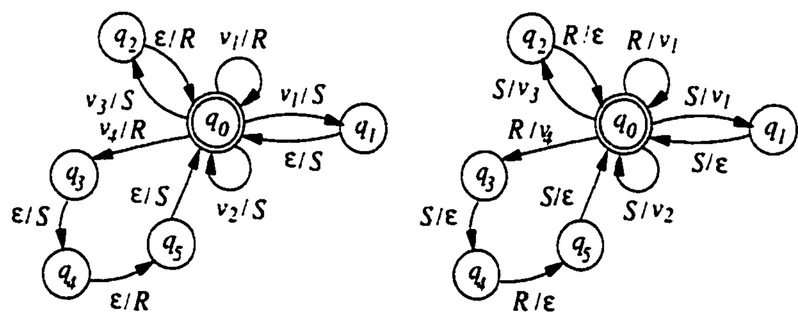


Figure 14: Transducers for the substitution and inverse substitution.

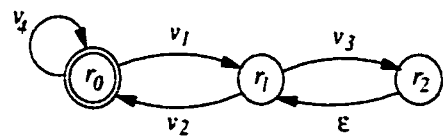


Figure 15: Automaton of the p-rewriting.

$$\{(a_i, v_2, i+1) : 1 \leq i \leq n-1\} \cup$$
$$\{(a_i, v_3, a_{i+1}) : 1 \leq i \leq n-1\} \cup$$
$$\{(i, v_4, i+2) : 1 \leq i \leq n-2\}.$$

The above source collection is visualised in Figure 16.

We can now compute

$$ans(Q'', S_n) = \{(i, j) : 1 \leq i \leq n-1, i \leq j \leq n\},$$

and

$$ans(Q', S_n) = \{(i, 2k) : 1 \leq i \leq n-1, 0 \leq k \leq n/2\}.$$

Then we have that the cardinality of

$$\begin{aligned} \| ans(Q'', S_n) \| &= n + \ldots + 2 \\ &= \sum_{i=1}^{n-1} (i+1) \\ &= \frac{n(n-1)}{2} - 1 \\ &\approx \frac{n^2}{2} \end{aligned}$$

and the cardinality of

$$\begin{aligned} \| ans(Q', S_n) \| &= \sum_{i=1}^{n-1} \left(\left\lfloor \frac{n-i}{2} \right\rfloor + 1 \right) \\ &\approx \frac{n(n-1) - n}{4} + n \\ &\approx \frac{n^2}{4}. \end{aligned}$$

Thus the cardinality of $ans(Q'', S_n) \setminus ans(Q', S_n)$ is approximately $n^2/2 - n^2/4 = n^2/4$, that is 16 times better that $(2n)^2$ which the number of all the possible pairs. ■

Now let us calculate the cost of our algorithm for computing the "possibility" regular rewriting.

Theorem 9 *The automaton characterizing Q'' can be built in time polynomial in the size of the regular expression representing Q and the size of the regular expressions representing* $\mathbf{V}$. ■

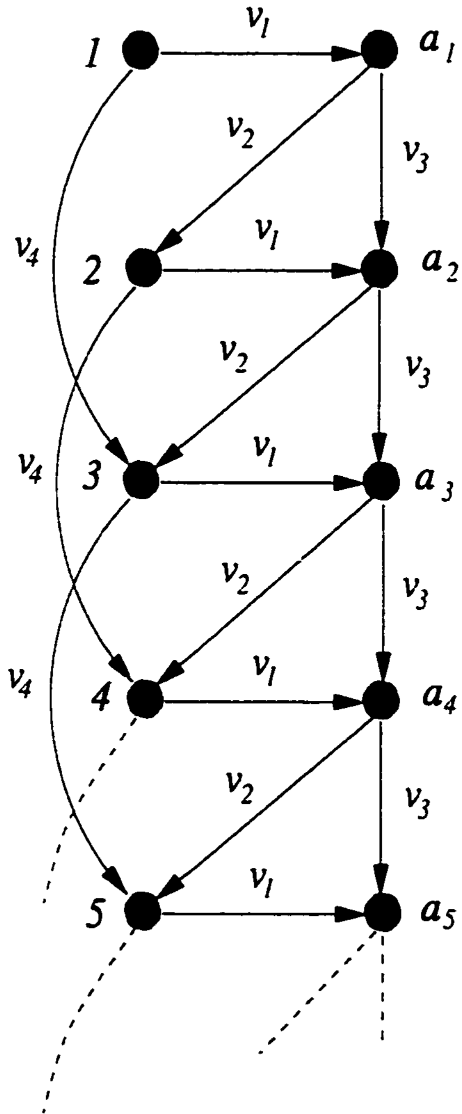


Figure 16: Source collection for Example 4

We note that the above complexity analysis is wrt expression and not data complexity. Since the decision procedure of [CGLV2000] is coNP-complete wrt data complexity, and this data is very large in DB system reducing the set of candidate pairs is very desirable.

3.6 Design for Implementation of the P-Rewriting

This section contains the design of the Implementation of P-Rewriting. The intent is to provide a clear and understandable view of Implementation architecture.

The implementation targets on the feasibility experiment of P-Rewriting algorit hm. We need to run some hundred views using this implementation. So, the performance constaint will be our highest priority. We decide to choose C as our implementation language for its efficiency.The implementation will use Unix on a convension architecture.

The implementation will use input files which contains the data for the query and views. and store the parsed data in the inner data structure , which in our case is the STL vector. In addition. this implementation can optionally output the result in a file.

For the user interface, we will not have any Graphical User Interface. We will use command line user interface which will have the command line argument format at the prompt as:
% execuable query view1 view2 ....

For the file we are using as our input and output will have the format as follows:

```
for example:
2 * 2
B {1,3} {-1}
N {-1} {0}
```

The first line indicate the number of rows and columns of the table. And each row represents a begining state and column represents the input and data inside curtly bracelets represent the ending state.

B represents the this row is both the start state and end state.
N represents this is a normal state.
S represents this is a start state.
F represents this is a final state.
-1 inside curtly bracelets represents there is no ending state from the start state with corresponding input.

For the main data structure, we will use STL vector to modelling the transition talbe and views, which in our case will be using three dimension to modelling the automata gragh.

In Figure 17, shows the inner relationship between the components. The input file of query will be parsed and stored in the inner data structure as NFA by calling the function $Create_NFA()$, and input view files will be parsed and stored also in the inner data structure as transducer by calling the function $CreateTransducer()$. The output of the above two functions will be the input of the function $Cartesian_product()$ to do the cartesian product. Then this output will be the input to the function $cartesian2NFA()$ and returns the result. For the algorithm , please refer to [GT2000].

3.7 Source code for Implementation of the P-Rewriting

We are using the tables to implement the automata and transducer to represent the data structure of the views. And we also do the cartesian product of the NFA and Transducer to come up with a new NFA.

This implementation has been experimented with 500 views and has completed the computation rather fast which shows that the theroy of the query rewriting enjoys

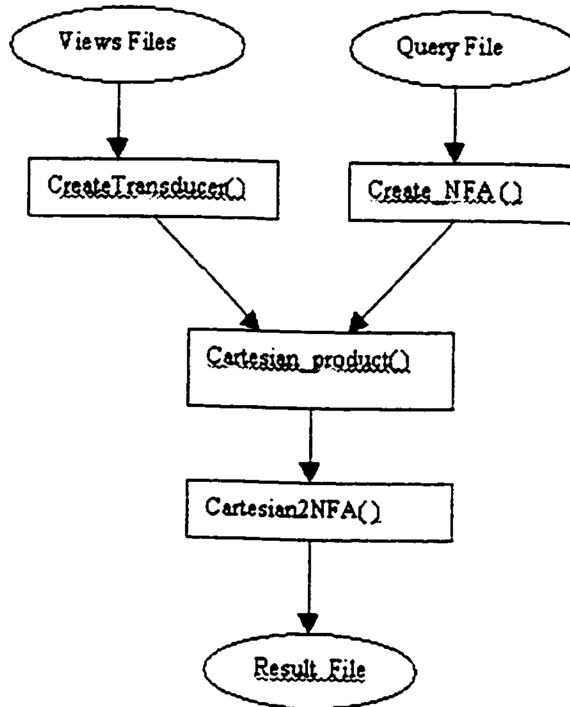


Figure 17: architecture diagram

efficient and reasonable preformance.

The main function will take the input as command line argument and output will be in the output file.

We have provide comments in the code so that it will be self explanatory.

The following is the code:

```
#include <iostream.h>
#include <strings.h>
#include <vector.h>
#include <fstream.h>
#include <stdio.h>

using namespace std;
```

```
struct state_and_output {
state_and_output(int state, int output) {
this->state = state;
this->output = output;
}
int state;
int output;
};


struct cart_state_and_output {
cart_state_and_output(int NFA_state, int TRANSDUCER_state, int output) {
this->NFA_state = NFA_state;
this->TRANSDUCER_state = TRANSDUCER_state;
this->output = output;
}
int NFA_state;
        int TRANSDUCER_state;
int output;
};


// The function will create the NFA
{\bf void create_NFA}(vector<int>** &NFA,
vector<int> &start_state,
vector<int> &final_state,
vector<char> &indicator,
char* filename,
int &n,
int &m)  {

// n is row number and m is colum number
```

```
char s[8000];
char* token1;
char* token2;

int i;

//open the datafile
        ifstream fin(filename);

//get row number and colum the vector NFA
fin.getline(s,8000);
token1 = strtok(s," *");
n=atoi(token1);             // the row number
token1 = strtok(NULL," *");
m=atoi(token1);     // the colum number

//cout<<"row is "<<n <<" and colum is "<<m<<endl;

//allocating memory
NFA = new (vector<int> *) [n];
for( i=0; i<n; i++)
NFA[i] = new (vector<int>) [m];

// read from the file to get the state and symbol

for( i=0; i<n; i++){
fin.getline(s,8000);

   token1 = strtok(s," "); //get the status S or F or N
   //cout<<token1<<endl;


//create start state and final state vector for later
```

```
//use
if ( (char)*token1 == 'S') start_state.push_back(i);
else if ((char)*token1 == 'F') final_state.push_back(i);

//insert the status of the state in the indicator vector
indicator.push_back((char)*token1);

vector<char*> temp;
for(int j=0; j<m; j++) {

token1 = strtok(NULL," ");
if (token1 != NULL) temp.push_back(token1);

}

//Start processing the vector temp.
for (int j=0; j<m; j++) {
token2= strtok(temp[j], "{,}\b");

while (token2 != NULL){
NFA[i][j].push_back(atoi(token2));
                token2 = strtok(NULL,"{,}\b");
}
  }

}
}

// the algorithm 4 steps which will be repeated.
// notation -100 means lumta, -i from 1 to 99 means views
void create_T_Algorithm(vector<state_and_output>** &TRANSDUCER,
vector<int> **NFA,
vector<int> start_state,
```

```
vector<int> final_state,
int start,             // start postion in the transducer
int n, int m,          // the number of rows and colums in this view
int last,              // the last row of the transducer
int viewnumber
){

int i,j,k,l;

//first step add lumta as outpyut to fm
int state;

for( i=0; i<n; i++){
for ( j=0; j<m; j++) {
for (k=0; k<NFA[i][j].size(); k++)
{ if (NFA[i][j][0]!=-1)
TRANSDUCER[start+i][j].push_back(
state_and_output(NFA[i][j][k]+start, -100));
else TRANSDUCER[start+i][j].push_back(
state_and_output(-1, -100));

}
  }

}

// third step

int startState;

        for (i=0; i<start_state.size(); i++){

startState=start_state[i];
```

```
for (j=0; j<m; j++) {
for (k=0; k<NFA[startState][j].size(); k++) {
if (NFA[startState][j][0]!=-1){
TRANSDUCER[last][j].push_back(
state_and_output(NFA[i][j][k]+start, -100));

//fourth step

  for (l=0; l<final_state.size();l++)
   if (NFA[i][j][k]==final_state[l])
TRANSDUCER[last][j].push_back(
   state_and_output(last, 0-viewnumber));
 }



}
}
}



//fifth step

for( i=0; i<n; i++){
for ( j=0; j<m; j++) {
for (k=0; k<NFA[i][j].size(); k++)
{ if (NFA[i][j][0]!=-1){

for (l=0; l<final_state.size();l++)
   if (NFA[i][j][k]==final_state[l])
TRANSDUCER[start+i][j].push_back(
   state_and_output(last, 0-viewnumber));
 }
}
```

```
  }
}
}



// create the transducer with row n and colum m
{\bf void create_TRANSDUCER}( vector<state_and_output>** &TRANSDUCER,
int argc, char** argv, int &n_final, int &m
){
int i,j,k,l;

vector<int> **NFA;
vector<int> start_state;
vector<int> final_state;
vector<char> indicator;
char s[8000];
char* token1;

vector<int> temp;

int n=0;
for (i=2; i<argc; i++){
 ifstream fin(argv[i]);

//get row number and colum the vector NFA
fin.getline(s,8000);
token1 = strtok(s," *");
//put the row number of each file in the temp vector
temp.push_back(atoi(token1));
n+=atoi(token1);              // the row number
token1 = strtok(NULL," *");
```

```
m=atoi(token1);    // the colum number

}

//cout << "n = "<<  n << endl;
n_final = n+1; //one more for s'0 state

//allocating memory
// we allocate n+1 row , because we need to add a new node
TRANSDUCER = new (vector<state_and_output> *) [n_final];
for( i=0; i<n_final; i++)
TRANSDUCER[i] = new (vector<state_and_output>) [m];



//second step add S0 with all the input to itself
/*
for (j=0; j<m; j++){
TRANSDUCER[n][j].push_back(
state_and_output(n, j));
}
*/


// create the NFA and using algorith step 2-5
int start=0;
int n1, m1; //n1 is NFA row number and m1 is NFA colum number
for (i=0; i<temp.size();i++) {
vector<int> start_state;
vector<int> final_state;
create_NFA(NFA,start_state, final_state,indicator,argv[i+2], n1, m1);
create_T_Algorithm(TRANSDUCER, NFA, start_state,
  final_state, start, n1, m1 ,n,i+1);
```

```
start+=temp[i];

}

}




// create cartisen _product
{\bf void cartisen_product}(vector<cart_state_and_output>** &CARTISAN, vector<i
vector<state_and_output> **TRANSDUCER, int n1,int n2, int m) {

        int l= n1*n2;

        //allocating memory
CARTISAN = new (vector<cart_state_and_output> *) [l];
for(int i=0; i<l; i++)
CARTISAN[i] = new (vector<cart_state_and_output>) [m];




 for(int h=0; h<n1; h++)
for (int i=0; i<n2; i++)
for (int j=0; j<m; j++){
for (int k=0; k<NFA[h][j].size(); k++)
for (int l=0; l<TRANSDUCER[i][j].size(); l++){
CARTISAN[h*n2+i][j].push_back(
cart_state_and_output(
NFA[h][j][k],
TRANSDUCER[i][j][l].state ,
TRANSDUCER[i][j][l].output ));
```

```
}
}

}

// transfer the cartisen product format  to our NFA format
{\bf void cartesian2NFA}(vector<cart_state_and_output>** &CARTISAN,
   vector<int>** &CAR_NFA,
   int n1, int n2, int m, int no_of_views){

int i,j,k, output;

//allocating memory
int l= n1*n2;  //lines in the cartesian product
CAR_NFA = new (vector<int> *) [l];
for( i=0; i<l; i++)
CAR_NFA[i] = new (vector<int>) [no_of_views + 1];
// +1 for the lambda


//We scan the cartesian product
for (i=0; i<l; i++)
for(j=0; j<m; j++)
//We scan inside the elements that are vectors
for(k=0; k<CARTISAN[i][j].size(); k++){
output=CARTISAN[i][j][k].output; //the third elment of the triple i.e. the
//output symbol
if (output==-100){ //lambda, so we put it in the last column
//If one of the two elemnets of the triple is -1 it means empty
//so we put emty in the new NFA.
if((CARTISAN[i][j][k].NFA_state==-1)||
  (CARTISAN[i][j][k].TRANSDUCER_state==-1)) {
```

```
if(CAR_NFA[i][no_of_views].size() == 0)
CAR_NFA[i][no_of_views].push_back(-1);
}
else {
   if(CAR_NFA[i][no_of_views].size() && CAR_NFA[i][no_of_views][0] == -1) {
//If we already have pushed in -1 this means that
//we union with empty set. Since, it is no need to do so
//we pop out -1.
CAR_NFA[i][no_of_views].pop_back();
   }
   CAR_NFA[i][no_of_views].push_back(
CARTISAN[i][j][k].NFA_state*n2
+CARTISAN[i][j][k].TRANSDUCER_state );
}
}

else {  //The third element is not lambda, it is view special symbol
//If one of the two elements of the triple is -1 it means empty
//so we put emty in the new NFA.
        if((CARTISAN[i][j][k].NFA_state==-1)||
  (CARTISAN[i][j][k].TRANSDUCER_state==-1)) {
if(CAR_NFA[i][-1-output].size() == 0)
CAR_NFA[i][-1-output].push_back(-1); //we put empty
}
      else {
   if(CAR_NFA[i][-1-output].size() && CAR_NFA[i][-1-output][0] == -1) {
//If we already have pushed in -1 this means that
//we union with empty set. Since, it is no need to do so
//we pop out -1.
CAR_NFA[i][-1-output].pop_back();
   }
   CAR_NFA[i][-1-output].push_back(
    CARTISAN[i][j][k].NFA_state*n2
```

```
+CARTISAN[i][j][k].TRANSDUCER_state );

}
}
}


for(i=0; i<l; i++)
for(j=0; j<no_of_views + 1; j++){
if(CAR_NFA[i][j].size() == 0)
CAR_NFA[i][j].push_back(-1);
}
}


// printing a  NFA
void print_NFA(vector<int> **NFA, vector<char> indicator,
int n, int m, char* filename){

ofstream fout(filename);
fout<<n<<" * "<<m <<endl;

for(int i=0; i<n; i++) {

fout <<indicator[i]<<"  ";

for(int j=0; j<m; j++) {

fout << "{";
for(int k=0; k<NFA[i][j].size(); k++){
fout << NFA[i][j][k] << ",";
}
```

```
fout << "\b} ";
}
fout << endl;
}
}

//Printing a TRANSDUCER
void print_TRANSDUCER(vector<state_and_output> **TRANSDUCER, int n, int m){
for(int i=0; i<n; i++) {
for(int j=0; j<m; j++) {

cout << "{"; int k;
for(k=0; k<TRANSDUCER[i][j].size(); k++)
cout << "(" <<
TRANSDUCER[i][j][k].state << "," <<
TRANSDUCER[i][j][k].output << "),";
cout << "\b} ";
}
cout << endl;
}
}


//Print the result of cartisen product of NFA and TRANSDUCER
void print_cartisen_product (vector<cart_state_and_output>** CARTISAN,
                              int n1,int n2, int m) {

for(int i=0; i<n1*n2; i++) {
for(int j=0; j<m; j++) {

cout << "{"; int k;
for(k=0; k<CARTISAN[i][j].size(); k++)
cout << "(" <<
```

```
CARTISAN[i][j][k].NFA_state << "," <<
CARTISAN[i][j][k].TRANSDUCER_state << "," <<
CARTISAN[i][j][k].output << "),";
cout << "\b} ";
}
cout << endl;
}

}

// the main function of Implementation
void main (int argc, char* argv []) {

int n1, n2, m;  // n is number of row, and m is the number of colume

vector<int> **NFA;
vector<int> start_state;
vector<int> final_state;
vector<char> indicator;

vector<state_and_output> **TRANSDUCER;
create_TRANSDUCER(TRANSDUCER, argc, argv, n2, m);
cout<<"***********************************************"<<endl;
print_TRANSDUCER(TRANSDUCER, n2, m);
cout<<"***********************************************"<<endl;

create_NFA(NFA,start_state, final_state,indicator,argv[1],n1,m);
print_NFA(NFA, indicator, n1, m,"output");
        cout<<"***********************************************"<<endl;


//cartisan product
```

```
vector<cart_state_and_output>** CARTISAN;
cartisen_product(CARTISAN, NFA, TRANSDUCER, n1, n2, m);
print_cartisen_product (CARTISAN, n1, n2, m);


//transfer cartisan product to NFA


int no_of_views = argc - 2;

vector<int> **CAR_NFA;
cartesian2NFA (CARTISAN, CAR_NFA, n1,n2, m, no_of_views);
printf("Hello!\n");
print_NFA(CAR_NFA, indicator, n1*n2, no_of_views+1,"cartisen.txt");

}
```

3.8 Experiment for Implementation of the P-Rewriting

The above code will be able to compile and run in Unix machine. We choose the Orchid machine to do the experiment.

Experiment 1 For 100 views:
time table200 views/query views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view

views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
output:
Cartesian product done!
Cartesian2NFA done!
Cartesian indicator done!
real 0.7
user 0.3
sys 0.0

Experiment 2 For 200 views:
time table200 views/query views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view

views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view
output:
Cartesian product done!
Cartesian2NFA done!
Cartesian indicator done!
real 1.5
user 1.2
sys 0.0

Experiment 3 For 300 views:
time table200 views/query views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view

views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view

views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view

output:
Cartesian product done!
Cartesian2NFA done!
Cartesian indicator done!
real 2.7
user 2.5
sys 0.0

Experiment 4 For 400 views:

time table200 views/query views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view

views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view

views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view

output:
Cartesian product done!
Cartesian2NFA done!
Cartesian indicator done!
real 4.9

user 4.5
sys 0.0

Experiment 5 For 500 views:
time table200 views/query views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view

views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view

views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view views/001.view views/002.view
views/003.view views/004.view views/001.view views/002.view views/003.view views/004.view
views/001.view views/002.view views/003.view views/004.view

output:
Cartesian product done!
Cartesian2NFA done!
Cartesian indicator done!
real 7.6
user 7.0
sys 0.0

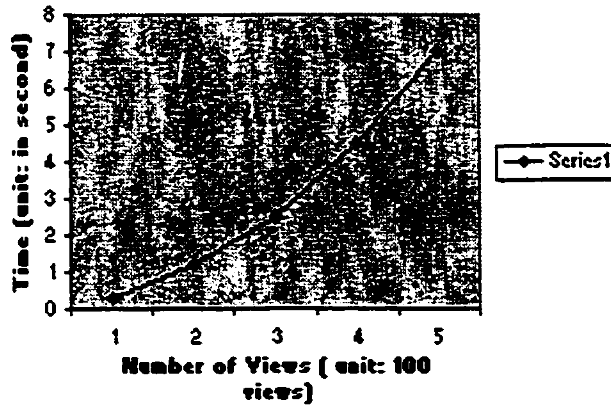


Figure 18: Performance Analysis Chart

For experiment purpose we only create 4 different views. As they repeat themselves, the system will treat them as new views when doing the cartesian product. We comment off the output part of the application since we do not need to output our result in a file in actual use. And file IO takes too much time which will not give us the real picture of the performance. We follow the same idea for all our experiments.

From the above experiments, we draw the performance anlaysis chart in Figure 18. As we can see from the figure, the x-axis represent the number of views we are using to do the experiment and the y-axis represent the time consuming of the experiment. For 500 views , it only takes 7 second to finish. Similar observation indicate that the algorithm is scalable and feasible.

The above data may be different from each try based on the host we use and the CPU usage rate. Even the result shows it will take more time than what we record, it still can be considered as "good" and feasible performance.

Chapter 4

Conclusion and Future work

This implementation has been experimented with 500 views and has completed the computation rather fast which has shown that the theroy of the query rewriting is feasible and practical.

Although the data we have collected during the experiment may be different based on the machine we choose to do the experiment and the CPU speed, we are convinced that this algorithm is scalable, although we have not provided a formal argument to support this. It can be used for some hundred views as experiment shows. Moreover, we observed that the implementation has a somewhat linear behavior scales according to the number of views.

To our knowledge , there is no other implementation on algebraic rewriting using which we could find a super set of the certain answer of semi-structured query having a set of views. So, we can not do any comparison at this moment.

In the current implementation we represent the automata and the transducer as transition tables based on STL vector classes. As future extensions to our work we would propose implementing the automata and the view transducer using other data-structures. Since, the non-deterministic automata and transducers can be viewed as labeled directed graphs, we could as well use the graph data-structures which would best fit to our specific experimenting purpose. For example, implementing

the automata and the transducer by using an adjacent matrix instead, would greatly simplify the procedure for garbage state removal, which in turn would enhance the performance of computing the cartesian product. By using this way of modelling the automata and the transducer. we could also produce a smaller cartesian product which would have as a side effect a faster query answering computation. However, as it is well known the adjancy matrix representation. is not efficient when the graphs are sparse since there would be a lot of wasted memory. Hence, we also propose as a future work. to conduct a study of memory-speed trade off for data-structure models.

Bibliography

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

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

[AD98] S. Abiteboul. O. M. Duschka. Complexity of Answering Queries Using Materialized Views. *Proc. of PODS* 1998 pp. 254-263

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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