Automata Methods and Techniques for
Graph-Structured Data

by

Maryam Shoaran
B.Eng., University of Shahid Beheshti, 2005
M.Sc., University of Victoria, 2007

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

© Maryam Shoaran, 2011
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by
photocopying or other means, without the permission of the author.
Automata Methods and Techniques for
Graph-Structured Data

by

Maryam Shoaran
B.Eng., University of Shahid Beheshti, 2005
M.Sc., University of Victoria, 2007

Supervisory Committee

Dr. Alex Thomo, Supervisor
(Department of Computer Science)

Dr. Bruce Kapron, Departmental Member
(Department of Computer Science)

Dr. Lin Cai, Outside Member
(Department of Electrical and Computer Engineering)
**Supervisory Committee**

Dr. Alex Thomo, Supervisor  
(Department of Computer Science)

Dr. Bruce Kapron, Departmental Member  
(Department of Computer Science)

Dr. Lin Cai, Outside Member  
(Department of Electrical and Computer Engineering)

**ABSTRACT**

Graph-structured data (GSD) is a popular model to represent complex information in a wide variety of applications such as social networks, biological data management, digital libraries, and traffic networks. The flexibility of this model allows the information to evolve and easily integrate with heterogeneous data from many sources.

In this dissertation we study three important problems on GSD. A consistent theme of our work is the use of automata methods and techniques to process and reason about GSD.
First, we address the problem of answering queries on GSD in a distributed environment. We focus on regular path queries (RPQs) – given by regular expressions matching paths in graph-data. RPQs are the building blocks of almost any mechanism for querying GSD. We present a fault-tolerant, message-efficient, and truly distributed algorithm for answering RPQs. Our algorithm works for the larger class of weighted RPQs on weighted GSDs.

Second, we consider the problem of answering RPQs on incomplete GSD, where different data sources are represented by materialized database views. We explore the connection between “certain answers” (CAs) and answers obtained from “view-based rewritings” (VBRs) for RPQs. CAs are answers that can be obtained on each database consistent with the views. Computing all of CAs for RPQs is NP-hard, and one has to resort to an exponential algorithm in the size of the data–view materializations. On the other hand, VBRs are query reformulations in terms of the view definitions. They can be used to obtain query answers in polynomial time in the size of the data. These answers are CAs, but unfortunately for RPQs, not all of the CAs can be obtained in this way. In this work, we show the surprising result that for RPQs under local semantics, using VBRs to answer RPQs gives all the CAs. The importance of this result is that under such semantics, the CAs can be obtained in polynomial time in the size of the data.

Third, we focus on XML – an important special case of GSD. The scenario we consider is streaming XML between exchanging parties. The problem we study is flexible validation of streaming XML under the realistic assumption that the schemas of the
exchanging parties evolve, and thus diverge from one another. We represent schemas by using Visibly Pushdown Automata (VPAs), which recognize Visibly Pushdown Languages (VPLs). We model evolution for XML by defining formal language operators on VPLs. We show that VPLs are closed under the defined language operators and this enables us to expand the schemas (for XML) in order to account for flexible or constrained evolution.
Contents

Supervisory Committee ii

Abstract iii

Table of Contents vi

List of Figures ix

Acknowledgements x

Dedication xi

1 Introduction 1

1.1 Graph-Structured Data ........................................... 1

1.2 Regular Path Queries ............................................. 5

1.3 Distributed Evaluation of RPQs .................................. 5

1.4 View-Based Query Answering ..................................... 6

1.5 XML (Extensible Markup Language) .............................. 7

1.5.1 Schemas for XML .............................................. 8

1.6 Contributions and Outline ....................................... 10
2 Fault-Tolerant Distributed Regular Path Queries

2.1 Introduction .................................................. 11

2.1.1 Distributed evaluation of generalized RPQs ............... 13

2.2 Databases and Weighted RPQ's ............................. 16

2.3 Distributed Algorithm ...................................... 20

2.4 Termination .................................................. 37

2.5 Complexity .................................................. 40

2.6 Soundness and Completeness ............................... 43

2.7 Fault Tolerance .............................................. 47

2.7.1 Intermittent Process Failures ........................... 59

2.8 Conclusions .................................................. 61

3 Certain Answers and Rewritings for Local Regular Path Queries on Graph-Structured Data

3.1 Introduction .................................................. 62

3.2 Graph-Structured Data and Local Regular Path Queries ....... 67

3.3 Certain Answer .............................................. 69

3.4 View-Based Rewritings ..................................... 70

3.5 Computing View-Based Rewritings .......................... 73

3.6 Related Work ............................................... 76

3.7 Conclusions .................................................. 77

4 Evolving Schemas for Streaming XML .......................... 79
4.1 Introduction ................................................. 79
4.2 Related Work .............................................. 87
4.3 Visibly Pushdown Automata and Transducers ................. 89
  4.3.1 Visibly Pushdown Automata .............................. 89
  4.3.2 Visibly Pushdown Transducers ......................... 94
4.4 Language Deletion and Insertion ............................ 95
  4.4.1 Language Deletion ...................................... 95
  4.4.2 Language Insertion ...................................... 101
  4.4.3 Transforming a VPL with Language Operations ........ 106
4.5 Constrained Deletions and Insertions ......................... 108
  4.5.1 Coloring VPTs for Deletion Rules ...................... 110
  4.5.2 Coloring VPTs for Insertion Rules ...................... 111
  4.5.3 Color-Tolerant VPTs ..................................... 112
  4.5.4 Transforming a VPL with Constrained Operations .... 113
4.6 Conclusions ............................................... 116

5 Conclusions .................................................. 118

Bibliography ................................................... 120
List of Figures

Figure 1.1 A relational database .................................................. 3
Figure 1.2 A graph-structured database ......................................... 4
Figure 1.3 Different representations of an XML patient record: (a) linear,
            (b) tree ................................................................. 8
Figure 2.1 A database $DB$ and a query automaton $A$ ...................... 18
Figure 2.2 Task Status Diagram. ................................................... 24
Figure 2.3 A database and a query automaton .................................. 30
Figure 2.4 A possible execution of Algorithm 1. Due to space constraints,
            we have abbreviated $prov$ by $p$, and $opt$ by $o$. We show in $\textbf{bold}$
            the tuples under expansion. ....................................... 31
Figure 2.5 Tuple Status Diagram. ................................................ 49
Figure 3.1 A database $D$. ......................................................... 65
Figure 3.2 Incomplete view extension $D_v$ for $V = 70s(70s + 80s)$. ...... 65
Figure 3.3 (a) $D_v$ ($O_v = \{a_1\})$. (b) Two possible databases for view ($o, r+s$). 71
Figure 3.4 A query automaton $A$. ............................................. 76
Figure 4.1 Example of a VPA. ...................................................... 92
ACKNOWLEDGEMENTS

“In the name of God, Most Gracious, Most Merciful.

All praise be to God, the Lord of the worlds.”

Holy Quran, Chapter 1, Verses 1–2

First and foremost, thanks be to God for His endless grace through which impossibles can truly be accomplished.

I would like to express my deep gratitude to my supervisor, Dr. Alex Thomo, for his mentorship, guidance, and patience throughout my graduate studies at UVic. I appreciate his expertise in the area, which provided insights that guided and challenged my thinking, also his kind support and assistance in writing this thesis.

I would also like to acknowledge Dr. B. Kapron as well as Dr. L. Cai for taking time out from their busy schedules to serve as my committee members.

My special and sincere thanks go towards my family for constant support they have provided me through my entire life. Without their love, patience, and encouragement I would never have been able to overcome the challenges in my life.
DEDICATION

To my beloved mother, and late father.
Chapter 1

Introduction

1.1 Graph-Structured Data

Nowadays data is generated at an exponential rate from scientific experiments, measurement devices, business information systems, social networks, applications extracting facts from unstructured text, and so on. The need is to make sense of these data, to explore and link them, to extract intelligence from them, and to integrate them with data from other sources. The common difficulty today in handling data is the existence of many different representations among applications, which need to share, exchange, and link data.

As pointed out by [1], the most basic problem with attempting to connect data sets is the fact that most data is stored in inflexible structures. Typically the data is stored in relational databases, most of which have predefined tables to fit the data that was initially believed to be important. This is fine for well-defined data because relational
databases have excellent performance when well configured, but presents problems when the application requires new kinds of data, new fields, or new relationships to be added frequently.

To build up the intuition, we present an example in Fig. 1.1, which shows a simple relational table organization for people and student supervision\(^1\). The people in our database are academicians, and they have an affiliation, when it is known. Their name and affiliation is stored in table \textit{People}. The other table, \textit{Supervision}, stores supervisor-student relationships in the natural way, as pairs.

However, we might not be happy with this state of affairs. For one, we might not know the affiliation for a considerable number of people, and thus have a lot of \textit{Nulls} in the database. Or, we might want to specify when a supervision took place, e.g. during the 70s, 80s, etc. We might also like to indicate what research areas each person has, e.g. databases, algorithms, and so on. For all these, the current database needs to change. Specifically, new fields, new tables, and new relationships should be added. This process of re-design can be costly and disruptive.

Another, new, and efficient approach is to have a very simple organization in the form of a graph of entities (nodes) and labeled–connections (edges) between them. Remarkably, such an organization can support any form of data. In Fig. 1.2 we show an example of such a graph capturing the data of Fig. 1.1, and where we have additionally recorded the time periods of supervisions as labels of the edges connecting entities corresponding to people. Observe that the \textit{Nulls} are “irrelevant”\(^1\)

\(^1\)The information is from http://infolab.stanford.edu/pub/voy/museum/ullmantreeBU.txt
in this model. If we do not know the affiliation of a person, there will just not be an edge to an affiliation node. Another nice thing about the graph organization is that costly operations in the relational database world, such as adding new columns or tables, translate only into adding new nodes and edges in the graph. For instance, if we want to add areas of research for some people, we would need to create two extra tables, one containing research areas, and the other connecting people with research areas. This is an overkill, especially if we only want to record the areas for few people. On the other hand, if the data is organized as a graph, then we only need to add few nodes corresponding to research areas and add edges connecting people with those nodes.

Data organized in this way are called “graph-structured data” (GSD). As illustrated, GSD provides a great flexibility as the data can now vary over time and within
a single database. The flexible nature of graph-structured data has made it the foundation for a multitude of applications in many important areas such as information integration, Web and communication networks, and biological data management, to name a few. Spatial, web, or social networks can also be modeled as graph databases. Furthermore, there are ongoing, enormous efforts as well for capturing the immense knowledge available on the Web and representing it in the form of “subject-predicate-object” triples, which are in essence “node-label-node” graph edges. The graphs they form are the well-known RDF graphs (often called triplestores) and they are abundant today (cf. [2, 3]).

Although the structure of GSD might look like the older Network and Hierarchical database models, databases built using those models were queried by programs. Namely, users needed significant programming skills and also the compilation of the query programs had to be done by stopping and starting the database.
1.2 Regular Path Queries

There is an inherent need to navigate GSD by means of a recursive query language. As pointed out by seminal works in the field (cf. [4, 5, 6, 7, 8]), regular path queries (RPQ’s) are the “winner” when it comes to expressing navigational recursion over GSD. These queries are in essence regular expressions over the database edge labels, and in general, one is interested in finding database paths which spell words in the (regular) query language.

As an example over the database given in Fig. 1.2, consider the regular expression

\[(70s + 80s)^* \cdot affiliation\]

over the edge alphabet \(\{70s, 80s, 90s, 00s, affiliation\}\). In plain language, this regular expression, considered as an RPQ, asks for all pairs \((a, b)\) of objects (or entities) where \(a\) is a supervisor, and \(b\) is an affiliation of some “academic descendant” of \(a\) supervised during 70s or 80s.

1.3 Distributed Evaluation of RPQs

The first problem we study is distributed evaluation of RPQs. Today, data, including GSDs, are generated in massive volumes and stored in various locations usually remote from each other. Therefore, an intelligent way to process such distributed data is to create fully distributed applications to be able to make use of the information and
processing capabilities of remote sources.

In Chapter 2 of this dissertation, we focus on answering RPQs in a fully distributed setting. We assume that each node of the data graph is processed by a separate process. In this way we make use of whatever parallelism is available. We can make use of as many processors as there are nodes in the database. Of course, when there are fewer processors, more than one process run on each processor, and thus, they can be started as “soft threads” in order to not consume the operating system. Our proposed algorithm is truly distributed in the sense that each process is completely autonomous and there is no central authority.

We also extend our algorithm to handle process failures in a transparent way. In plain words, this means that the query evaluation does not need to be restarted in face of process failures, but instead it adapts to preserve the computation done so far and output at least the best answer were the evaluation to restart using the remaining live processes.

1.4 View-Based Query Answering

The second problem we study in this dissertation is answering RPQs when only database view information is available.

One of the important problems in a database context is reasoning about query answering when the available data is incomplete. An important variant of the problem, known as “view-based query answering”, is extensively studied in different contexts,
including relational databases, and graph-structured data (cf. [6, 9]).

We explore the connection between “certain answers” (CAs) and answers obtained from “view-based rewritings” (VBRs) for RPQs. CAs are answers that can be obtained on each database consistent with the views. Computing all of CAs for RPQs is NP-hard, and one has to resort to an exponential algorithm in the size of the data–view materializations. On the other hand, VBRs are query reformulations in terms of the view definitions. They can be used to obtain query answers in polynomial time in the size of the data. These answers are CAs, but unfortunately for RPQs, not all of the CAs can be obtained in this way.

In this work, we show the surprising result that for RPQs under “local” semantics, using VBRs to answer RPQs gives all the CAs. Local regular path queries (LRPQs) are RPQs that match paths starting from a given node. This is an important case in practice, where the user is interested in retrieving results only starting from some object they prefer, e.g. Ullman in our previous example. The importance of our result is that for LRPQs, the CAs can be obtained in polynomial time in the size of the data.

1.5 XML (Extensible Markup Language)

The third problem addressed in this dissertation is about XML data—an important special case of GSD, where the data graph is a tree.

XML was introduced as an augmentation of HTML allowing to annotate data
with information about its meaning rather than just its presentation [10]. Now XML is a ubiquitous standard for representing information on the Web.

XML data\textsuperscript{2} can be represented as node-labeled, ordered trees. An example is given in Fig. 1.3. On the left we show a linear representation of XML data\textsuperscript{3}, whereas on the right we show its tree representation, called the XML tree. The tags in the linear representation correspond to the labels of the nodes in the XML tree. While XML trees are node-labeled, they can be easily transformed into edge-labeled ones by giving the node labels to the incoming edges.

![XML tree representation](image)

Figure 1.3: Different representations of an XML patient record: (a) linear, (b) tree

1.5.1 Schemas for XML

Lacking a strong schema gives flexibility to GSD. However, the advantage of a pre-defined schema, which provides a description of the data, is that the users are better

\footnote{XML data are also called XML documents in the literature. Hence, we will use these terms interchangeably.}

\footnote{This is a linear representation as a long string, even though, for better readability, we have added end of lines and tabs.}
positioned to formulate meaningful queries and the query optimization and storage are more efficient. This has motivated attempts to introduce flexible tree schema formalisms for XML.

DTD (Document Type Definition) was the first schema formalism for XML. A DTD schema is a grammar like set of rules. Among other, stronger, schema languages proposed for XML, we mention the XML Schema ([11]) and Relax NG ([12]) which can be captured by Extended Document Type Definitions (EDTDs) (cf. [13]).

The specific problem we study is flexible validation of XML. The validation is about checking whether XML data conforms to a schema or not. The scenario we consider is streaming XML between exchanging parties. We study the problem under the realistic assumption that the schemas of the exchanging parties evolve, and thus diverge from one another. We represent schemas by Visibly Pushdown Automata (VPAs), which recognize Visibly Pushdown Languages (VPLs). We retain full generality because it is shown that VPAs have the same power as EDTDs. However, the advantage of using VPAs is that we can process XML in a streaming fashion. Namely, we work on the linear representation of XML and we perform only a single pass on data.

We model evolution for XML schemas by defining formal language operators on VPLs. We show that VPLs are closed under the defined language operators and this enables us to expand the schemas and account for flexible or constrained evolution.
1.6 Contributions and Outline

Specifically we make the following contributions in this dissertation.

- In Chapter 2 we present a fully distributed algorithm to answer RPQs on GSD. We give detailed proofs for the soundness and completeness of our algorithm and show that the message complexity is optimal. We also present a resilient version of our algorithm, which can tolerate any number of process failures without imposing extra message load on the system [14, 15].

- In Chapter 3 we show that for local RPQs computing the set of certain answers can be done in polynomial time. This is due to the fact, also proved in Chapter 3, that for such queries, the set of certain answers coincides with the set of answers obtained by a view-based rewriting, which in turn can be evaluated (answered) in polynomial time in the size of the data. Next, we present an algorithm that computes maximal view-based rewriting for local RPQs. The rewriting is in the form of a multientry DFA which facilitates evaluating it in parallel on the data graphs [16].

- After proving the closure of Visibly Pushdown Automata (VPA) under language operations of insertion and deletion, in Chapter 4 we first introduce $k$-bounded, as well as, constrained language operations on VPLs. Then, using VPAs to characterize XML schemas, we devise algorithms based on automata techniques, to represent XML schema evolution in a streaming setting of data-exchange applications [17, 18].
Chapter 2

Fault-Tolerant Distributed Regular Path Queries

2.1 Introduction

In this chapter we evaluate regular path queries on graph-structured data (GSD) in a truly distributed setting. Our proposed algorithm can be used not only to answer simple RPQs, but also a more general form of them given by preferentially weighted regular expressions or automata. Moreover, the database graph can have weighted edges as well, depending on the application. The best example to illustrate weights in queries and/or databases comes from spatial networks (SN).

Let us consider an SN (such as [19]), which is essentially a graph with edges labeled by highway, road, street, and so on. Suppose now the user wants to find paths consisting mainly of highway segments and tolerating up to $k$ provincial roads or city
streets. Clearly, such paths can easily be captured by the RPQ

\[ Q = \text{highway}^* \| (\text{road} + \text{street} + \epsilon)^k, \]

where \( \| \) is the shuffle operator (see e.g. [20]).

However, the user might want to express preferences beyond simple edge labels. For instance suppose the user wants to quantify her preferences to indicate how much more she prefers highways compared to roads or city streets. For this we consider generalized RPQs with weights as in [21, 22, 23, 24, 25]. For example, the user can write

\[ Q = (\text{highway} : 1)^* \| (\text{road} : 2 + \text{street} : 3 + \epsilon)^k, \]

to express that she ideally prefers highways, then roads, which she prefers less, and finally she can tolerate streets, but with an even lower preference.

Moreover, inherent database edge weights (or importance) can be naturally incorporated to scale up or down query preferences. Thus, in our spatial example, the edge importance could simply be the edge-length, and so, traversing a 100 kms highway would be less preferable than traversing a 49 kms provincial road, even though in general provincial roads are less preferable than highways.

This assumes the existence of a meaningful “multiplication” operator which combines query preferences with edge weights. Of course, it is completely the choice of the user to specify query preferences, or to indicate her interest in considering the importance of traversed database edges.
Computing the answer to such an RPQ amounts to computing the “all-pairs shortest paths” in the subgraph of database paths spelling words in the query language. However, for each user query, there would be a new subgraph on which to compute all-pairs shortest paths, and such a subgraph cannot be known in advance, but rather only after the query evaluation finishes. This is “too late” for applying algorithms, which need global knowledge of the whole graph. With such algorithms, the user cannot see partial answers while waiting for the query to finish, and there is extra computation and communication overhead incurring after the subgraph [relevant to the query] is determined. Thus, the well-known Floyd-Warshall algorithm and its distributed variants are not appropriate to our database setting.

2.1.1 Distributed evaluation of generalized RPQs

Regarding work on distributed shortest path computation, we remark here Haldar’s algorithm in [26], which computes all-pairs shortest paths with the best known number of messages. In this chapter, inspired by [26], we devise an algorithm to compute answers to RPQs and to work in an environment where the relevant part of the database graph is not known beforehand, but rather incrementally computed on the fly.

Our algorithm works under the assumption that the nodes of the relevant graph are computed on demand and they have local [neighbor] knowledge only. The central idea of our algorithm is to overlap computations starting from different database objects. We achieve this overlap in a careful way in order to guarantee the expansion of the best path first, in a similar spirit with the Dijkstra’s methodology. However,
at the same time we allow multiple expansions at different processes, which is what makes the algorithm truly distributed.

Next, we extend our algorithm to account for process failures. Having a fault-tolerant algorithm is very important especially in today’s new paradigm of grid computing. Notably, in a grid setting the power comes from the synergy of many participating machines, whose main purpose might be completely different from the “grid-community service” performed during their low intensity periods. As such, grid machines are quite “unreliable” because they can withdraw at any time from a grid computation in order to perform their main “duties” they primarily are intended for.

Our fault-tolerant algorithm can smoothly adapt and be resilient to any number of process failures. Furthermore, it guarantees finding at least all the query answers obtainable if the computation were to be started from the scratch on the remaining live processes. Furthermore, we remark that, since some of the computation used supersets of these remaining processes, in general, we get more results than those strictly available if we were to restart the computation on the remaining processes only.

Finally, we note that our fault-tolerant algorithm does not require additional messages apart from the “ping”-like messages of the infrastructure for detecting process failures. We require for the processes to monitor the health of their neighbors only.

Notably, all the above are important and desirable properties for distributed fault-tolerant algorithms.
**Related Work.** To the best of our knowledge, only very few works present a distributed evaluation of regular path queries. In [23], a distributed algorithm is presented, which works based on local knowledge only. However, it has a message complexity which is quadratically worse than the complexity in this work.

Besides [23], other works that have dealt with distributed RPQ’s are [27, 28, 29, 30]. All four consider the local variant of RPQ’s.

Finally, two recent works, [31] and [32], have presented distributed methods for the XPath query evaluation over XML trees using partial evaluation techniques. Their methods are not applicable to our case due to the following reasons. First, the methods of [31] and [32] work on a tree structure of XML documents, whereas databases in our context are general graphs and there are no “leaf” designated nodes. Second, they consider *unweighted* tree databases, and thus, the problem they deal with is in fact about reachability rather than shortest paths, which in turn is the case for our algorithm.

**Organization.** The rest of the chapter is organized as follows. In Section 2.2, we give the definitions we are based on. In Section 2.3, we present our distributed algorithm. Next, in Section 2.4 and 2.5, we discuss its termination and complexity, respectively. In Section 2.6, we show the soundness and completeness of our algorithm. In Section 2.7, we extend our algorithm to be resilient against process failures. Finally, Section 2.8 concludes the chapter.
2.2 Databases and Weighted RPQ’s

We consider a database to be an edge-labeled graph with positive real values assigned to its edges. Intuitively, the nodes of the database graph represent objects and the edges represent relationships (and their importance) between the objects.

Formally, let $\Delta$ be an alphabet. Elements of $\Delta$ will be denoted $R,S,\ldots$. As usual, $\Delta^*$ denotes the set of all finite words over $\Delta$. We also assume that we have a universe of objects, and objects will be denoted $a,b,c,\ldots$. A database $DB$ is then a weighted graph $(V,E)$, where $V$ is a finite set of objects and $E \subseteq V \times \Delta \times \mathbb{R}^+ \times V$ is a set of directed edges labeled with symbols from $\Delta$ and weighted with numbers from $\mathbb{R}^+$.

Before talking about weighted preference path queries, it will help to first review the classical path queries.

A regular path query (RPQ) is a regular language over $\Delta$. Computationally, an RPQ is a finite state automaton (FSA) $A = (P,\Delta,\tau,p_0,F)$, where $P$ is the set of states, $\Delta$ is the alphabet, $\tau \subseteq P \times \Delta \times P$ is the transition relation, $p_0$ is the initial state, and $F$ is the set of final states. For the ease of notation, we will blur the distinction between RPQ’s and FSA’s that represent them.

Let $A$ be a query FSA and $DB = (V,E)$ a database. Then, the answer to $A$ on $DB$ is defined as

$$\text{Ans}(A, DB) = \{(a,b) \in V \times V : a \xrightarrow{w} b \text{ in } DB \text{ and } w \text{ is accepted by } A\},$$

where $a \xrightarrow{w} b$ denotes a path from $a$ to $b$ spelling $w$ in the database.
Now, let $\mathbb{N} = \{1, 2, \ldots \}$. A *weighted finite state automaton* (WFSA) $A$ is a quintuple $(P, \Delta, \tau, p_0, F)$, where $P$, $p_0$, and $F$ are similarly defined as for a classical FSA, while the transition relation $\tau$ is now a subset of $P \times \Delta \times \mathbb{N} \times P$. Query WFSA’s are given by means of weighted regular expressions (WRE’s). The reader is referred to [33] for efficient algorithms translating WRE’s into WFSA’s.

Given a weighted database $DB = (V, E)$, and a query WFSA $A = (P, \Delta, \tau, p_0, F)$, the preferentially *scaled weighted answer* (SWAns) of $A$ on $DB$ is

$$\text{SWAns}(A, DB) = \{(a, b, r) \in V \times V \times \mathbb{R}^+ : r = \inf \left\{ \sum_{i=1}^{n} r_i k_i : n \in \mathbb{N}, (c_{i-1}, R_i, r_i, c_i) \in E, (p_{i-1}, R_i, k_i, p_i) \in \tau \right\}$$

$$c_0 = a, c_n = b, \text{ and } p_n \in F \} \}.$$

Observe that, according to this definition, if $(a, b, r) \in \text{SWAns}(A, DB)$, then there exists a path (possibly a set of paths) from $a$ to $b$ in $DB$ spelling some word(s) in the query language. Furthermore $r$ is the weight of the cheapest sequence of edge-transition matches corresponding to such paths. Number $n \in \mathbb{N}$ denotes the length of a path and is (possibly) different for different paths.

As an example, consider the database $DB$ and query automaton $A$ in Fig. 2.1. There are three paths going from object $a$ to object $c$. The shortest path consisting of a single edge $T$ of weight 1, is not the cheapest path according to the query. Rather, the cheapest path is the one spelling $RS$. The other path, spelling $RT$, does not match any query automaton path, so it is not considered at all. Hence, we have that
\((a, c, 3)\) is the answer with respect to \(a\) and \(c\).

Similarly, we find the other query answers and finally have \(SWAns(A, DB) = \{(a, b, 1), (a, c, 3), (a, d, 6), (a, a, 7), (b, c, 5), (b, d, 8), (b, a, 9)\}\).

![Figure 2.1: A database DB and a query automaton A](image)

In order to help understanding of our distributed algorithm, we will first review the well-known method for the evaluation of classical RPQ’s (cf. [34]). The evaluation proceeds by creating object-state pairs from the query automaton and the database. For this, let \(A\) be a query FSA. Starting from an object \(a\) of a database \(DB\), we first create the pair \((a, p_0)\), where \(p_0\) is the initial state in \(A\). Then, we create all the pairs \((b, p)\) such that there exists an edge from \(a\) to \(b\) in \(DB\) and a transition from \(p_0\) to \(p\) in \(A\), and furthermore the labels of the edge and the transition match. In the same way, we continue to create new pairs from existing ones, until we are no longer able to do so. In essence, what is happening is a lazy construction of a Cartesian product graph of the database with the query automaton. Of course, only a small (hopefully) part of the Cartesian product is really constructed. This ultimately depends on the selectivity of the query.

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

Now, when having instead a weighted query automaton and database, one can build a weighted Cartesian product graph. We show that in order to compute weighted answers, we have to find, in the Cartesian product graph, the cheapest paths from all \((a, p_0)\) to all \((b, p)\), where \(p\) is a final state in the query automaton \(A\).

As we mentioned in the Introduction, in general there is a different Cartesian product graph for each query. Thus, a useful distributed algorithm must not rely on having global knowledge about this graph, since it will only be known after the completion of the query evaluation.

We formally define the Cartesian product \(C\) of a database \(DB = (V, E)\) and a query automaton \(A = (P, \Delta, \tau, p_0, F)\) as the graph with

- nodes \((b, p)\), where \(b\) is an object in \(V\) and \(p\) is a state in \(P\), and
- edges \(((b, p), R, rk, (c, q))\), such that there exists an edge \((b, R, r, c)\) in \(E\) and a transition \((p, R, k, q)\) in \(\tau\).

Based on this definition, we have that

**Theorem 1.** \((a, b, r) \in SWAns(A, DB)\) if and only if there exists some path from \((a, p_0)\) to \((b, p_y)\) in \(C\), with \(p_y\) being a final state in \(A\) and \(r\) the weight of a cheapest of such paths.

**Proof.** By the construction of \(C\), we have that:
1. For every path $\pi_1$ in $DB$ matching some weighted transition path $\pi_2$ in $A$, there exists some path $\pi$ in $C$ spelling the same word as $\pi_1$ (and $\pi_2$) and annotated by the product of the weights of the edges and transitions in $\pi_1$ and $\pi_2$, respectively.

2. For every path $\pi$ in $C$ there exist paths $\pi_1$ in $DB$ and $\pi_2$ in $A$, which match and spell the same word as $\pi$, and furthermore, the corresponding edges and transitions of $\pi_1$ and $\pi_2$, respectively, have weights whose products give the weights of the edges in $\pi$.

Now, our claim is a direct consequence of the above, and the definition of $SWAns(A, DB)$.

\[\square\]

### 2.3 Distributed Algorithm

The key feature of our algorithm is the overlapping of computations starting from different database objects. We assume that each database object has only local knowledge about the database graph, that is, it only knows the identities of its neighbors and the labels and weights of its outgoing edges. Further, we assume that each object $a$, is being serviced by a dedicated process for that object $P_a$. Our algorithm can be easily modified for the case when subgraphs of the database (as opposed to single objects) are being serviced by the processes. In such a case, many of the basic computation messages are sent and received locally by the processes from and to themselves.

First, the query automaton is sent to each process. Such a service is commonly
achieved by distributively creating a minimum spanning tree (MST) of the processes before any query starts to be evaluated (cf. [35] for a message optimal MST algorithm).

We can note here that such an MST can be used by the processes to transmit their id’s and get so to know each other. However, we do not require this coordination step. Even if such a step is undertaken, the real challenge [which remains] is that the relevant subgraph of the [query–database] Cartesian product cannot be known in advance for a new query. In other words, a shortest path algorithm has to work with a target graph not known beforehand.

Continuing the description of our algorithm, a process, say \( P_a \) (which serves object \( a \)), starts by creating an initial task for itself. The tasks are “keyed” (uniquely identified) by the automaton states, with the initial tasks being keyed by the initial state \( p_0 \). Each task has three components:

1. an automaton state,

2. a status flag that can switch between active, passive, and completed values, and

3. a table (or set) of tuples representing knowledge about “objects reached so far” along with additional information (to be precisely described soon).

A typical task will be written as \( \langle p_x, status, \{ \ldots \} \rangle \). We will refer to the table \( \{ \ldots \} \) as \( P_a.p_x.T \) or \( p_x.T \) when \( P_a \) is clear from the context. The tuples in this table have four components, and will be written as \([ (c, p_z), (b, p_y), weight, status ] \), where
1. \((c, p_z)\) states that the algorithm, starting from object \(a\) and state \(p_x\), has reached (possibly through multiple hops) object \(c\) and state \(p_z\),

2. \((b, p_y)\) states that the best path (known so far) to reach \((c, p_z)\) is by passing via object \(b\) and state \(p_y\), where \(b\) and \(p_y\) are neighbors of \(a\) and \(p_x\) in the database and query automaton, respectively,

3. weight is the weight of this best path (determined as in Section 2.2), and

4. status is a flag switching from prov to opt values telling whether weight is provisional and would possibly be improved or optimal and permanently stay as is.

Initially, when a \(p_x\)-task is created, process \(P_a\) tries to find all the outgoing edges from \(a\), which match (w.r.t. the symbol label) outgoing transitions from \(p_x\). Let \((a, R, r, b)\) be such an edge which matches transition \((p_x, R, k, p_y)\). Then, \(P_a\) inserts tuple \(\left[(b, p_y), (b, p_y), k \cdot r, prov\right]\) in table \(P_a.p_x.T\). If there are multiple \((a, \ldots, b)\) - \((p_x, \ldots, p_y)\) edge-transition matches, then only the match with the cheapest weight product is considered.

Each process \(P_a\) starts by creating and initializing a passive \(p_0\)-task, which is possibly selected next for processing. We say “possibly” because a process might receive new tasks from neighboring processes.

When a task is selected for processing, its provisional-status tuples (or provisional tuples in short) will be “expanded” in a best-first order with respect to their weights. If there are no more provisional tuples in the table of the \(p_0\)-task, then the task attains
a *completed* status, and the process reports its *local termination*.

All (working) processes run in parallel exactly the same algorithm, which consists of four concurrent threads. These threads are as follows:

**Expansion:** A process $P_a$ selects a *passive* task, say $p_x$–task, which still has provisional tuples in its table.

Then, $P_a$ makes the $p_x$–task *active*, and selects for expansion the cheapest *provisional* tuple in its table $P_a.p_x.T$.

The *active* status for the $p_x$–task prevents the expansion of other *provisional* tuples in $P_a.p_x.T$.

Next, $P_a$ sends a request message to its neighbor $P_b$ asking it to: (1) create a task $p_y$, and (2) send its “knowledge” regarding the $[(c, p_z), \ldots, \ldots]$ tuple.

**Task Creation:** When a process $P_b$ receives a request message from $P_a$ (w.r.t $p_x$) for the creation of a task, say $p_y$, it creates a $p_y$-keyed task (if such does not exist) and properly initializes it. Next, $P_b$ establishes a virtual communication channel between its $p_y$-task and the $p_x$-task of $P_a$. This communication channel is specialized for the relevant tuple (keyed by $(c, p_z)$), whose expansion caused the request message. The weight of the channel will be equal to the cost of going from $(a, p_x)$ to $(b, p_y)$, which is in fact the weight of the $(b, p_y)$–keyed tuple in $P_a.p_x.T$.

Notably, overlapping of computations happens when process $P_b$ receives another request message for the same task from a different neighboring process. In such
a case, the receiving process $P_b$ only establishes a communication channel with the sending process.

**Reply:** After creating the communication channel, process $P_b$ will send table $P_b.p_y.T$ backward to task $P_a.p_x$. This backward message will be sent only when the $(c,p_z)$-keyed tuple in $P_b.p_y.T$ attains an *optimal* status. The weight of the communication channel is added to the weights of the tuples as they are bundled together to be sent. We refer to this modified (message) table as $P_b.p_y.T^*$. 

**Update:** When a process $P_a$ receives from some process $P_b$ a backward reply message, which is related to a tuple $[(c,p_z), \ldots, prov]$ of task $P_a.p_x$, and contains the table $P_b.p_y.T^*$, it will: (1) update (relax) the *provisional* tuples in $P_a.p_x.T$ as appropriate (if there are tuples with the same keys in $P_b.p_y.T^*$), (2) add to table $P_a.p_x.T$ all tuples of $P_b.p_y.T^*$, which do not have any “peer” (tuple with the same key) in $P_a.p_x.T$, and (3) change the status of the $p_x$-task to *passive*.

![Task Status Diagram](image)

Figure 2.2: Task Status Diagram.

Figure 2.2 illustrates the different possible statuses of a task during the execution of the algorithm. As described above, at the moment of creation, each task has *passive* status. If a *passive*-status task does not have any *provisional* tuple in
its table, the status is changed to completed. Otherwise, the process can start the expansion of provisional tuples in the task table. Starting the expansion of a tuple, the task status is changed to active which, as mentioned in the Expansion thread, prevents the expansion of other provisional tuples until receiving the reply to the last request message. When an active-status task receives a reply message for the recent expansion, it starts the Update thread, at the end of which the task status is changed to passive making the task ready for another expansion. So, the passive and active statuses can interleave several times during the execution of the algorithm, but the completed status does not change once it has been reached.

Formally our algorithm is as follows.

**Algorithm 1.**

**Input:**

1. A database $DB$. For simplicity we assume that each database object, say $a$, is being serviced by a dedicated process for that object $P_a$.

2. A query WFSA $A = (P, \Delta, \tau, p_0, F)$.

**Output:** $\text{SWAns}(A, DB)$.

**Method:**

1. **Initialization:** Each process $P_a$ creates a task $\langle p_0, \text{passive}, \{\ldots\} \rangle$ for itself. The table $\{\ldots\}$ (referred to as $P_a.p_0.T$) is initialized as follows:

   (a) insert tuple $[(a, p_0), (a, p_0), 0, \text{opt}]$, and
(b) For each edge-transition match, 

\((a, R, r, b)\) in \(DB\) and  

\((p_0, R, k, p)\) in \(A\), 

insert tuple \([(b, p), (b, p), k \cdot r, prov]\)  

(if there are multiple \((a, \_, \_, b) - (p_0, \_, \_, p)\) edge-transition matches, 
then the cheapest weight product is considered.) 

If at point (b) there is no edge-transition match, then make the status of the \(p_0\)-task completed.

2. Concurrently execute all the four following threads at each process in parallel until termination is detected. [For clarity, we describe the threads at two processes, \(P_a\) and \(P_b\).] 

3. **Expansion:** [At process \(P_a\)]

   (a) Select a passive \(p_x\)-task for processing. Make the status of the task active.

   (b) Select the cheapest provisional-status tuple, say \([(c, p_z), (b, p_y), w, prov]\) from table \(P_a.p_x.T\).

   (c) Request \(P_b\), with respect to state \(p_y\), to provide information about \((c, p_z)\). 
   For this, send a message \(\langle p_y, [p_x, (c, p_z), w_{ab}]\rangle\) to \(P_b\), where \(w_{ab}\) is the cost of going from \((a, p_x)\) to \((b, p_y)\), which is equal to the weight of the \((b, p_y)\)-keyed tuple in \(P_a.p_x.T\).
(d) Sleep, with regard to \( p_x \)-task, until the reply message for \((c, p_z)\) comes from \(P_b\).

4. **Task Creation**: [At process \(P_b\)]

Upon receiving a message \(\langle p_y, [p_x, (c, p_z), w_{ab}]\rangle\) from \(P_a\):

- **if** there is not yet a \(p_y\)-task

  then create a task \(\langle p_y, \text{passive}, \{\ldots\} \rangle\) and initialize its table similarly as in the first phase.

  That is,

  (a) insert tuple \([(b, p_y), (b, p_y), 0, \text{opt}]\), and

  (b) For each edge-transition match,

  \[(b, R, r, d) \text{ in } DB \text{ and} \]

  \[(p_y, R, k, p_u) \text{ in } A, \]

  insert tuple \([(d, p_u), (d, p_u), k \cdot r, \text{prov}]\)

  (if there are multiple \((b, \ldots, d) - (p_y, \ldots, p_u)\) edge-transition matches, then the cheapest weight product is considered.)

Also, establish a virtual communication channel with \(P_a\). This channel relates the \(p_y\)-task of \(P_b\) with the \(p_x\)-task of \(P_a\). Further, it is indexed by \((c, p_z)\) and is weighted by \(w_{ab}\) (the weight included in the received message).

- **else** [\(P_b\) has already a \(p_y\)-task.] Do not create a new task, but only establish a communication channel with \(P_a\) as described above.
5. **Reply:** [At process $P_b$]

When in the $p_y$-task, the tuple $[(c, p_z), (\_\_\_), \_\_]$ is or becomes optimally weighted, *reply back* to all the neighbor processes, which had sent a task requesting message $\langle p_y, [\_\_, (c, p_z), \_\_\_] \rangle$ to $P_b$.

For example, $P_b$ sends to such a neighbor, say $P_a$, through the corresponding communication channel, the message $\langle P_b.p_y.T^* \rangle$, which is table $P_b.p_y.T$ after adding the channel weight to the weight of each tuple.

6. **Update:** [At process $P_a$]

Upon receiving a reply message $\langle P_b.p_y.T^* \rangle$ from a neighbor $P_b$ w.r.t. the expansion of a $(c, p_z)$-keyed tuple in table $P_a.p_x.T$ do:

(a) Change the status of $(c, p_z)$-keyed tuple to the status of the same keyed tuple in $P_b.p_y.T^*$.\(^1\)

(b) For each tuple $[(d, p_u), (\_\_), v, prov]$ in $P_b.p_y.T^*$, which has a smaller weight $(v)$ than a same-key tuple $[(d, p_u), (\_\_), prov]$ in $P_a.p_x.T$, replace the latter by $[(d, p_u), (b, p_y), v, prov]$.

(c) Add to $P_a.p_x.T$ all the rest of the $P_b.p_y.T^*$ tuples, i.e., those which do not have corresponding same-key tuples in $P_a.p_x.T$.

Also, change the via component of these tuples to be $(b, p_y)$.

(d) **if** the $p_x$-task does not have anymore *provisional* tuples,

\(^1\)This status is *optimal*. 
then make its status *completed*.

If \( p_x = p_0 \), then report that all query answers from \( P_a \) have been computed.

else make the status of the \( p_x \)-task *passive*.

Finally upon termination, which happens when all the tasks in every process have attained *completed* status, set

\[
eval(A, DB) = \{(a, b, r) : [(b, p_y), (\_\_, r, \text{opt})] \in P_a.p_0.T \text{ and } p_y \in F\}\.
\]

In the next section, we show the soundness and completeness of our algorithm. Based on them, the following theorem can be stated.

**Theorem 2.** Upon termination of the above algorithm, we have that

\[
eval(A, DB) = \text{SWAns}(A, DB).
\]

The algorithm can report answers as soon as their corresponding tuples become *optimal*.

We define a *partial answer set* to be a subset of \( \text{SWAns}(A, DB) \).

Now, instead of creating \( \text{eval}(A, DB) \) upon termination of the algorithm, we can incrementally grow it each time that a tuple becomes optimal. Because the weight of an optimal tuple does not change any further, any snapshot of \( \text{eval}(A, DB) \) at any time during the execution of the above algorithm is a partial answer set. Upon
termination, all the answers would have been reported. While the user waits for the query evaluation to finish, new answers will eventually arrive. However, the ones already reported preserve their weights, which are optimal. This is in contrast to [23] in which the user might see the already reported answers to possibly get their weights lowered.

Now, we illustrate Algorithm 1 by the following example. Consider the database and query automaton in Fig. 2.3, left and right respectively.

A possible sequence of actions for Algorithm 1 is given in Table 2.4. In the first column labeled “T” we number the hypothetical time points in which we observe the system. An explanation of the actions at each time point follows.

1. All processes create a task \( \langle p_0, \text{passive}, \{\ldots\} \rangle \) for themselves and initialize their tables.

2. (a) \( P_a \) and \( P_d \) do have provisional tuples in the tables of their \( p_0 \)-tasks, and thus, make their \( p_0 \)-tasks active and expand their cheapest provisional tuples.

   For this, they send a request message to \( P_b \) for the creation of a \( p_1 \)-task.
Figure 2.4: A possible execution of Algorithm 1. Due to space constraints, we have abbreviated prov by $p$, and opt by $o$. We show in **bold** the tuples under expansion.
On the other hand, processes $P_b$ and $P_c$ do not have provisional tuples in their $p_0$-tasks. Hence, they make their $p_0$-tasks completed. That is, there are no $(b, _, _)\text{ and } (c, _, _)\text{ query answers to be expected.}$

(b) $P_b$ receives the request messages from $P_a$ and $P_d$, and creates the $p_1$-task. Also, $P_b$ initializes this task as described in the algorithm. Of course, $P_b$ creates only one such task to serve both $P_a$ and $P_d$, and thus, we see here an effective computation overlap.

Then, $P_b$ establishes the appropriate communication channels between its $p_1$-task and the $p_0$-tasks in $P_a$ and $P_d$.

$P_b$ is not only asked to create the $p_1$-task, but also to provide information about the $(b, p_1)$-keyed tuple. Since the status of this tuple in the $p_1$-task of $P_b$ is optimal, $P_b$ sends its $p_1.T$ knowledge to $P_a.p_0$ and $P_d.p_0$ adding along the way the weights of the related channels.

3. Upon receiving the reply message from $P_b$, processes $P_a$ and $P_d$ update the tables of their $p_0$-tasks. Note that the statuses of the $(b, p_1)$-keyed tuples in $P_a.p_0.T$ and $P_d.p_0.T$ become optimal.

$P_a$ relaxes the $(c, p_1)$-keyed tuple in $p_0.T$ and changes its via to $(b, p_1)$. $P_d$ adds to $P_d.p_0.T$ the rest of the $P_b.p_1.T^*$ tuples setting their via component to $(b, p_1)$.

Then, $P_a$ and $P_d$ change the status of their $p_0$-tasks to passive becoming thus ready for the next expansion.
4. (a) $P_a$ and $P_d$ make the status of their $p_0$-tasks *active*, and expand the tuples $[(c, p_1), (b, p_1), 2, \text{prov}]$ and $[(c, p_1), (b, p_1), 3, \text{prov}]$ respectively by sending request messages to process $P_b$.

(b) $P_b$ has already a $p_1$-task, and thus, it just establishes communication channels with $P_a$ and $P_d$ specialized for $(c, p_1)$.

As the status of the $(c, p_1)$-keyed tuple in $P_b.p_1.T$ is *provisional*, $P_b$ cannot yet reply back to $P_a$ or $P_d$.

Instead, $P_b$ makes the status of task $p_1$ *active* and starts its processing. That is, $P_b$ selects the cheapest *provisional* tuple, i.e., the tuple $[(c, p_1), (c, p_1), 1, \text{prov}]$, and sends a request message to $P_c$ to create task $p_1$.

(c) Upon receiving the request message from $P_b$, process $P_c$ creates and initializes a $p_1$-task. Also, $P_c$ establishes a communication channel with $P_b$, which is specialized for $(c, p_1)$. Since the status of the $(c, p_1)$-keyed tuple is *optimal*, $P_c$ replies back to $P_b$ with the message $\langle P_c.p_1.T^* \rangle$.

[The rest of the steps will be described more briefly.]

5. (a) Upon receiving the reply message from $P_c$, $P_b$ updates its $p_1.T$ table as appropriate.

(b) Now, $P_b$ has an *optimal* status for the $(c, p_1)$-keyed tuple in $p_1.T$, and thus, replies back to $P_a$ and $P_d$ with the message $\langle P_b.p_1.T^* \rangle$. 
(c) Upon receiving the reply message from \( P_b \), \( P_a \) and \( P_d \) update their \( p_0.T \) tables as appropriate.

6. (a) \( P_a \) and \( P_d \) expand the tuples \([(d, p_1), (b, p_1), 3, prov]\) and \([(d, p_1), (b, p_1), 4, prov]\) respectively.

(b) In effect, \( P_b \) expands \([(d, p_1), (c, p_1), 2, prov]\), and then \( P_c \) expands \([(d, p_1), (d, p_1), 1, prov]\).

\( P_c \) requests from \( P_d \) to create a \( p_1 \)-task and provide information about \((d, p_1)\).

(c) \( P_d \) creates and initializes the \( p_1 \)-task and replies back to \( P_c \) with the message \( \langle P_d.p_1.T^* \rangle \).

7. (a) Upon receiving the reply message from \( P_d \), \( P_c \) updates its \( p_1.T \) table as appropriate.

Then, \( P_c \) replies back to \( P_b \) with the message \( \langle P_c.p_1.T^* \rangle \).

(b) Upon receiving the reply message from \( P_c \), \( P_b \) updates its \( p_1.T \) table as appropriate.

Then, \( P_b \) replies back to \( P_a \) and \( P_d \).

(c) Upon receiving the reply message from \( P_b \), \( P_a \) and \( P_d \) update their \( p_0.T \) tables as appropriate.

8. Finally, as there are no more provisional tuples in any of the tasks, they attain a *completed* status.
Observe that we can terminate as soon as the $p_0$-tasks become completed in all the processes. There is no need to continue with the completion of the rest of the tasks. Their completion would not bring any new query answers, thus we can safely abort them.

Note that, we can incrementally report the query answers as soon as their corresponding tuple appears in the table of a $p_0$-task in some process and obtains optimal status. For example, $(a, b, 1)$ and $(d, b, 2)$ can be reported at time point 3, $(a, c, 2)$ and $(d, c, 3)$ can be reported at time point 5, and so on.

At time point 4, when $P_a$ and $P_d$ expand the $(c, p_1)$-keyed tuples requesting $P_b$ to provide information about such a tuple in $P_b.p_1.T$, it happens that this tuple is the cheapest provisional tuple in $P_b.p_1.T$. Another instance of such a situation is at time point 6, in which again, the requested information is about a tuple that is the cheapest provisional tuple in $P_b.p_1.T$. These are not coincidental and by the following theorem, we show that this is indeed a property of the algorithm which guarantees the soundness (in Section 2.6). Of course, the request might be for an optimal tuple, and there is no need for further expansion in order to reply back. Note, that the following theorem is about the case when the request is for a provisional tuple.

**Theorem 3.** If a process, through a task request message, is asked to provide information about a provisional tuple, then this tuple is the cheapest one among such tuples in the requested task.

**Proof.** Suppose process $P_a$ asks process $P_b$ for a tuple in its $p_y$-task. Let the
expanded tuple in \( P_a \) be \([(c, p_z), (b, p_y), w_{ac}, prov] \). This expansion will ask from \( P_b \) to provide information about the \((c, p_z)\)-keyed tuple in its \( p_y \)-task. Let this tuple be \([(c, p_z), (\_\_, \_), w_{bc}, prov] \). We want to show that this tuple is the cheapest among the provision tuples in \( P_b.p_y.T \).

Since \((b, p_y)\) is the via component of the \((c, p_z)\)-keyed tuple in \( P_a.p_x.T \), we conclude that this tuple has got its weight, during an update phase, from the tuple \([(c, p_z), (\_\_, \_), w_{bc}, prov] \) in \( P_b.p_y.T \) after adding the weight of the corresponding communication channel.

Along with the \([(c, p_z), (\_\_, \_), w_{bc}, prov] \) tuple, \( P_a \) got from \( P_b \) all the other tuples in \( P_b.p_y.T \), on whose weights the same channel weight \( w_{ab} \) was added. Now, since \([(c, p_z), (b, p_y), w_{ac}, prov] \) is the cheapest provision tuple in \( P_a.p_x.T \), and its weight \( w_{ac} \) is in fact equal to \( w_{ab} + w_{bc} \), we have that \([(c, p_z), (\_\_, \_), w_{bc}, prov] \) is the cheapest tuple in \( P_b.p_y.T \).

Based on the above, we show now the following theorem which is needed in the proofs for the soundness and completeness of our algorithm (Section 2.6).

**Theorem 4.** Let \([(c, p_z), (b, p_y), w, prov] \) be a tuple in \( P_a.p_x.T \) selected for expansion, and

\([(c, p_z), (b, p_y), w', opt] \) be this tuple with optimal status after the expansion. Then, \( w = w' \).

**Proof.** When \([(c, p_z), (b, p_y), w, prov] \) gets expanded, a request message asking information about \((c, p_z)\) is propagated through a path \( \pi \) with nodes \((a, p_x), (b, p_y), \ldots \),
(c, p_z) until reaching a process with an optimal (c, p_z)-keyed tuple (P_c.p_z.T, at least, will have such an optimal tuple). Let π' be the subpath of π (starting from (a, p_x)) that is in fact traversed. Of course, π' might be the whole π when the only optimal (c, p_z)-keyed tuple along π is the one in P_c.p_z.T (which is surely optimal due to the initialization).

According to Theorem 3, in the task tables of the processes along π' there is no provisional tuple with a weight less than the weight of the (c, p_z)-keyed tuple. Thus, all the processes along π' expand in turn their (c, p_z)-keyed tuples. Since there is no other expansion during the processing of the (c, p_z)-keyed tuples along π', there is no change in the weight of these (c, p_z)-keyed tuples including the weight of tuple [(c, p_z), (b, p_y), w, prov] in P_a.p_x.T. Thus, we have w = w'.

2.4 Termination

In the following theorem we show that the algorithm terminates and it does not enter an infinite loop. That is, eventually there will be no more provisional tuples in the tables of the p_0 tasks, which is the condition for termination of the algorithm at each process.

Theorem 5. Algorithm 1 (positively) terminates.

Proof. Suppose there is a deadlock. Without loss of generality and for better clarity, assume there are only three processes involved in a deadlock.

Such deadlock can assumably be created in the following scenario.
1. Process $P_a$ expands tuple $[(d, p_u), (b, p_y), w_{ad}, prov]$ in its $p_x$-task. Thus, it sends a corresponding message to $P_b$ requesting a $p_y$-task and asking information about the $(d, p_u)$-keyed tuple in this $p_y$-task.

2. Process $P_b$ already has a $p_y$-task, but cannot reply back at the moment since there is some tuple $[(e, p_v), (c, p_z), w_{be}, prov]$ in the $p_y$-task, whose $w_{be}$ weight is smaller than the weight of the $(d, p_u)$-keyed tuple. Thus, $P_b$ sends a message to $P_c$ requesting a $p_z$-task and asking information about the $(e, p_v)$-keyed tuple in this $p_z$-task.

3. Process $P_c$ already has a $p_z$-task, but cannot reply back at the moment since there is some tuple $[(f, p_w), (a, p_x), w_{cf}, prov]$ in the $p_z$-task, whose $w_{cf}$ weight is smaller than the weight of the $(e, p_v)$-keyed tuple. Thus, $P_c$ sends a message to $P_a$ requesting a $p_x$-task and asking information about the $(f, p_w)$-keyed tuple in this $p_x$-task.

4. Process $P_a$ has an $(f, p_w)$-keyed tuple in the table of its $p_x$-task, and this tuple has a provisional status. Note that an $(f, p_w)$-keyed tuple certainly exists in the $p_x$-task of $P_a$. This is because otherwise, the via object-state pair of the $(f, p_w)$-keyed tuple in $P_c.p_x$ would not be $(a, p_x)$.

On the other hand, process $P_a$ has the $p_x$-task in active status waiting for a reply to the expansion of the $(d, p_u)$-keyed tuple. This prevents $P_a$ to expand any other tuple including the $(f, p_w)$-keyed tuple. Hence, it cannot reply back to $P_c$ and the deadlock assumedly occurs.
Now, we show that such a situation cannot happen during the execution of our algorithm.

Since $P_a$ expands tuple $[(d, p_u), (b, p_y), w_{ad}, prov]$ (in the table of the $p_x$-task), we have that $w_{ad}$ is the smallest weight among the provisional tuples of the $p_x$-task. In particular, $w_{af} \geq w_{ad}$, where $w_{af}$ is the weight of the $(f, p_w)$-keyed tuple in $P_a.p_x.T$.

Process $P_a$ has to get information about the $(d, p_u)$-keyed tuple through its neighbor process $P_b$, which is the via process for that tuple.

By the Update thread, we have that

$$w_{ad} = w[(a, p_x), (b, p_y)] + w_{bd},$$

where $w[(a, p_x), (b, p_y)]$ is the cheapest weight product of a matching automaton transition from $p_x$ to $p_y$ with a database edge from $a$ to $b$, and $w_{bd}$ is the weight of the $(d, p_u)$-keyed tuple in $P_b.p_y.T$.

Hence, $w_{af} \geq w_{ad} = w[(a, p_x), (b, p_y)] + w_{bd}$. As $P_b$ selects the tuple keyed by $(e, p_v)$ to expand, we have $w_{bd} \geq w_{be}$. Therefore, it can be concluded that $w_{af} \geq w[(a, p_x), (b, p_y)] + w[(b, p_y), (c, p_z)] + w_{ce}$.

According to the deadlock scenario outlined in the beginning of this proof, $P_c$ tries to expand tuple $[(f, p_w), (a, p_x), w_{cf}, prov]$ of the $p_z$-task when it is asked for information on the $(e, p_v)$-keyed tuple. So, $w_{ce} \geq w_{cf}$, and hence,

$$w_{af} \geq w[(a, p_x), (b, p_y)] + w[(b, p_y), (c, p_z)] + w_{cf}$$

$$= w[(a, p_x), (b, p_y)] + w[(b, p_y), (c, p_z)]$$

$$+ w[(c, p_z), (a, p_x)] + w_{af}.$$
However, recall from Section 2.2 that the edge weights are positive numbers, and thus the above cannot happen, reaching so a contradiction.

As mentioned earlier, the algorithm should terminate when each process has a completed \( p_0 \)-task. However, there is the question of how to detect the global termination of our algorithm. This can be done using an algorithm for distributed termination detection. There are many of such algorithms (see [36] for a thorough review) and they can be superimposed into any other distributed algorithm.

### 2.5 Complexity

**Theorem 6.** The number of messages required for a query evaluation is \( 2|E| \), where \( E \) is the set of edges in the lazy database-query Cartesian product graph.

**Proof.** We base our claim on the following facts:

1. Each (traversed) edge in the Cartesian product graph indicates a communication channel between two tasks of two processes which also is indexed by an object-state pair.

2. Only one forward message is needed to cause the creation of a communication channel.

3. Each communication channel is traversed only once, which happens when the tuple keyed by the object-state pair of the channel becomes optimally weighted.
The real number of messages ultimately depends on the query selectivity, and in practice one hopes that the lazy Cartesian product size is much smaller than the size of the database (cf. [34]).

Note that if a set of database objects is serviced by a process as opposed to having only one object serviced by a process, then the message complexity will be $2|E'|$, where $E' (E' \subset E)$ is the set of inter-process edges of the lazy Cartesian product.

We note that the above upper bound coincides with the message lower bound of Ramarao and Venkatesan in [37] for the distributed computation of single-source shortest paths. However, the messages in [37] have a constant size, while our messages have an $O(|V|)$ size, where $V$ is the set of object-state pairs in the lazy Cartesian product graph. Thus, in terms of $O(1)$ size messages, our algorithm can be considered as having $O(|E| \cdot |V|)$ such messages. On the other hand, our problem is more difficult than the classical single-source shortest paths problem of [37].

**Remark 1.** The result of a preliminary experiment\(^2\) shows that despite the fact that in theory the (worst case) message complexity of the algorithm is quadratic in the number of nodes of Cartesian product graph, in practice this complexity is so close to be linear.

**Remark 2.** One might be tempted to apply instead of our fully distributed algorithm the following semi-distributed approach.

First, collect the whole database in one process only. Then, apply a centralized shortest path algorithm on the Cartesian product of the database and query automa-

\(^2\)performed by Chris Pearson
This semi-distributed approach has several shortcomings. First, depending on the selectivity of the queries, large parts of the transmitted database might not be used at all during evaluation, thus resulting in unnecessary communication traffic.

Second, this solution asks from a single process to perform a huge computation which needs also to store the complete database. In other words, the memory requirement for the process performing the computation is at least $|E_{DB}|$, where $E_{DB}$ is the set of edges in database $DB$.

On the other hand, the memory requirement for each process in our fully distributed algorithm is only $O(|V|)$, where $V$ is the set of object-state pairs in the lazy Cartesian product graph.

**Remark 3.** One might be willing to use distributed shortest path algorithm directly on the implicit Cartesian product graph. In this case, in each distributed process all possible tasks should initially be created and fill their corresponding tables of reachable nodes. This will disturb the whole network if the database is large, which is usually the case, and consume the resources of each process. On the other hand, in database systems the queries are usually selective enough to not ask for the whole database, but rather for a relatively small portion of it. That is, most of the initial computations will be wasted if we follow this direct approach.

As an example, suppose the data graph is the graph of people in a social network with one million subscribers, and the query automaton contains $n$ states and $m$ different transition labels. Using direct approach, $O(10^6 \times n)$ tasks and $O((10^6 \times n)^2 \times m)$
tuples has to be initially created. While, using our algorithm presented in this thesis, each task is created on demand and we do not need to create most of these tasks and tuples based on the selectivity of the query.

2.6 Soundness and Completeness

In this section, we show the soundness and completeness of Algorithm 1. For the former, we show that each reported query answer is optimally weighted. For the latter, we show that all the query answers are indeed reported. In the following, we present two lemmas and then the main theorem of the section.

**Lemma 1.** If there exists a path from $(a, p_0)$ to $(c, p_z)$ in $C$, then there will be some $(c, p_z)$-keyed tuple that will be eventually inserted into $P_a.p_0.T$.

*Proof.* Suppose that there exists a path $\pi$ from $(a, p_0)$ to $(c, p_z)$ in $C$, but the algorithm, during its execution, never inserts some $(c, p_z)$-keyed tuple into $P_a.p_0.T$. Let $\pi$ be the sequence $(c_0, p_0), (c_1, p_1), \ldots, (c_n, p_n)$, where $n \geq 1$, $c_0 = a$, $c_n = c$, and $p_n = p_z$. Clearly, $[(c_0, p_0), (c_0, p_0), 0, opt]$ will be inserted into $P_a.p_0.T$ by the Initialization thread.

Let $k \in [1, n - 1]$ be the number for which we have that for all $h \in [0, k - 1]$ there is some $(c_h, p_h)$-keyed tuple inserted at some point into $P_a.p_0.T$, but there is never a $(c_k, p_k)$-keyed tuple inserted into $P_a.p_0.T$.

Clearly, there will be some expansion (in fact only one) of tuple $[(c_{k-1}, p_{k-1}), (\_, \_), (\_, prov)]$. Now, as $(c_{k-1}, p_{k-1})$ and $(c_k, p_k)$ are consecutive nodes in $\pi$, there exists at
least one edge connecting them; (at least) the edge in $\pi$.

The expansion of $[(c_{k-1}, p_{k-1}), (-, -), prov]$ will trigger a series of request messages all the way to process $P_{c_{k-1}}$ for task $p_{k-1}$. Process $P_{c_{k-1}}$ will in turn create (if it has not already done so) a $p_{k-1}$-task and insert an optimal $(c_{k-1}, p_{k-1})$-keyed tuple into $P_{c_{k-1}}.p_{k-1}.T$. Also, by the task creation, since $(c_{k-1}, p_{k-1})$ and $(c_k, p_k)$ are connected in $C$, we have that a $[(c_k, p_k), (-, -), prov]$ tuple is as well inserted into $P_{c_{k-1}}.p_{k-1}.T$ (see step 4.b in Algorithm 1). Now, through the back-reply messages, tuple $[(c_k, p_k), (-, -), prov]$ will travel and reach process $P_a$ where it is inserted into $P_a.p_0.T$. But this, contradicts our initial supposition.

Thus, for all the nodes $(c_i, p_i)$ in $\pi$, where $i \in [0, n]$, we have that some $(c_i, p_i)$-keyed tuple will be certainly inserted (at some point) in $P_a.p_0.T$. This applies to $(c_n, p_n) = (c, p_z)$ as well, and so, some tuple keyed by $(c, p_z)$ will be eventually inserted into $P_a.p_0.T$. \qed

From the above lemma and the specification of the Expansion and Update threads, we have that

**Corollary 1.** If there exists a path from $(a, p_0)$ to $(c, p_z)$ in $C$, then there will be eventually a tuple $[(c, p_z), (-, -), opt]$ in $P_a.p_0.T$.

Clearly, there is only one such tuple in $P_a.p_0.T$. Now we show that

**Lemma 2.** Let $[(c, p_z), (-, -), w, opt]$ be a tuple in $P_a.p_0.T$. Then, $w$ is the weight of a cheapest path going from $(a, p_0)$ to $(c, p_z)$ in $C$.

**Proof.** Let $[(c, p_z), (-, -), w, prov]$ be the $(c, p_z)$-keyed tuple in $P_a.p_x.T$ that gets
expanded. By the specification of the Update thread, after receiving the back-reply message corresponding to the expansion, the \((c, p_z)\)-keyed tuple gets an optimal status and by Theorem 4 its weight is \(w\).

Now, let \(\pi\), with a weight \(z\), be a cheapest path from \((a, p_0)\) to \((c, p_z)\) in \(C\). Then, we claim that \([ (c, p_z), (\_\_), z, prov] \) will exist at some point in \(P_{a-p_0}.T\), eventually expanded, and finally attain an optimal status. From this, our claim will follow as there can be only one \((c, p_z)\)-keyed tuple in \(P_{a-p_0}\), i.e. \(w\) will have to be equal to \(z\).

Suppose \(\pi\) has the following nodes: \((c_0, p_0), (c_1, p_1), \ldots, (c_n, p_n)\), where \(n \geq 1\), \(c_0 = a, c_n = c, p_n = p_z\). Let \(w_h\), where \(h \in [1, n]\), be the weight of the subpath of \(\pi\) from \((c_0, p_0)\) to \((c_h, p_h)\).

Clearly, \([ (c_0, p_0), (c_0, p_0), 0, opt] \) will be inserted into \(P_{a-p_0}.T\) by the Initialization thread. Suppose now that \([ (c_{h-1}, p_{h-1}), (\_\_), w_{h-1}, opt] \), for some \(h \in [1, n]\), is in \(P_{a-p_0}.T\). By the specification of the Expansion and Update threads, and Theorem 4, we have that (the provisional variant) \([ (c_{h-1}, p_{h-1}), (\_\_), w_{h-1}, prov] \) has been in \(P_{a-p_0}.T\) and at some point has been expanded.

Since \((c_{h-1}, p_{h-1})\) and \((c_h, p_h)\) are neighbors, reasoning similarly as in the proof of Lemma 1, we have that the expansion of the \((c_{h-1}, p_{h-1})\)-keyed tuple causes, through the corresponding back-reply message, the arrival (in \(P_{a-p_0}\)) of tuple \([ (c_h, p_h), (\_\_), w_h, prov] \).

If there is no \((c_h, p_h)\)-keyed tuple in \(P_{a-p_0}.T\), then \([ (c_h, p_h), (\_\_), w_h, prov] \) will be inserted in this table, and preserve weight \(w_h\) till the end (becoming eventually an optimal tuple with weight \(w_h\)). This is because \(\pi\) and its subpaths are cheapest paths,
and thus, there does not exist a cheaper way going from \((c_0, p_0)\) to \((c_h, p_h)\) in \(C\).

On the other hand, if there is already a \((c_h, p_h)\)-keyed tuple in \(P_{a.p_0.T}\), then its weight cannot be less than \(w_h\) because, otherwise, we could go from \((c_0, p_0)\) to \((c_h, p_h)\) through a cheaper way than the subpath of \(\pi\) between these two nodes, and this would imply that \(\pi\) is not a cheapest path. Thus, in this case, the arriving tuple \([\langle c_h, p_h \rangle, \langle \_\_ \_ \_ \rangle, w_h, prov \rangle\) will lower the weight of the \((c_h, p_h)\)-keyed tuple in \(P_{a.p_0.T}\) to \(w_h\) (if it is not already so).

Concluding, in both cases, \(P_{a.p_0.T}\) will have at some point a \([\langle c_h, p_h \rangle, \langle \_\_ \_ \_ \rangle, w_h, prov \rangle\) tuple, whose weight cannot be lowered any further. Since the algorithm continues until there is no provisional tuple in \(P_{a.p_0.T}\), there will be a moment when the \((c_h, p_h)\)-keyed tuple will get expanded and then attain an optimal status while preserving weight \(w_h\) (by Theorem 4).

Inductively, \(P_{a.p_0.T}\) will have at some point a \([\langle c_n, p_n \rangle, \langle \_\_ \_ \_ \rangle, w_n, prov \rangle = \langle c, p_z \rangle, \langle \_\_ \_ \_ \rangle, z, prov \rangle\) tuple, whose weight cannot be lowered any further. Upon expansion, based on Theorem 4, we will have \([\langle c, p_z \rangle, \langle \_\_ \_ \_ \rangle, z, opt \rangle\) in \(P_{a.p_0.T}\), and this completes our proof.

Based on all the above, we have that

**Theorem 7.** Algorithm 1 is sound and complete.

**Proof.**

“soundness.” From the definition of \(\text{eval}(A, DB)\) we have that the output pro-
duced by the algorithm is

\[ \{(a, b, r) : [(b, p_y), (\_), r, opt] \in P_{a,p_0}.T \text{ and } p_y \in F\}. \]

Now, let \([(b, p_y), (\_), r, opt]\) be a tuple in \(P_{a,p_0}.T\) and \((a, b, r)\) be the corresponding produced answer to the given query. From Lemma 2, we have that \(r\) is the weight of a cheapest path in \(C\) connecting \((a, p_0)\) to \((b, p_y)\). From Theorem 1, \((a, b, r)\) is an answer to the given query.

“completeness.” Let \((a, b, r)\) be an answer to the given query. From Theorem 1, there exists some path from \((a, p_0)\) to \((b, p_y)\) in \(C\), and \(r\) is the weight of a cheapest of such paths. From Corollary 1, the existence of some path from \((a, p_0)\) to \((b, p_y)\) in \(C\) means that a tuple \([(b, p_y), (\_), (\_), opt]\) will be eventually inserted into \(P_{a,p_0}.T\). From Lemma 2, the exact weight of this tuple will be equal to the weight of a cheapest path from \((a, p_0)\) to \((b, p_y)\) in \(C\), i.e. \(r\). Thus, \((a, b, r)\) will be produced as an answer by the algorithm.

\[\Box\]

### 2.7 Fault Tolerance

Having a fault-tolerant algorithm is very important in distributed settings that are prone to process failures. Although defunct hardware is rare, fault-tolerance is very prevalent today due to the popular geographically distributed grid systems (see PlanetLab [38]). In such systems, extreme power comes from the participation of numerous machines, whose service in a grid is usually offered during their low intensity peri-
ods. As such, grid machines are quite “unreliable” because they can withdraw at any
time from a grid computation in order to perform other “duties” they are primarily
intended for.

In this section, we show how to extend Algorithm 1 in order to be resilient against
process failures. We assume that even if a process fails, the corresponding database
object still exists. This assumption is the norm in database applications, where the
data lives longer than the processes that access it.

Let $DB'$ be the subset of database $DB$ serviced by the remaining alive processes
at the end of the query evaluation. After each failure, we will have a smaller database
being serviced by the live processes. Since the query evaluation is not started from
the scratch on $DB'$, but is continually evaluated on a series of databases which are
supersets of $DB'$, we can obtain more and better-weighted answers than what we
would get on $DB'$ only.

To make formal the description of the query answers returned by our fault-tolerant
algorithm, we first present the following definition.

Let $A$ and $B$ be sets of object-object-weight triples, i.e. $A, B \subseteq V \times V \times \mathbb{R}^+$. Then, we say that $A$ is superior to $B$, denoted by $A \supseteq B$, if $(a, b, r) \in B$ implies that $(a, b, r') \in A$, and $r' \leq r$.

Now, our fault-tolerant algorithm will compute a set $eval(A, DB)$ of triples. After
the description of the algorithm, we will show that

$$SWAns(A, DB) \supseteq eval(A, DB) \supseteq SWAns(A, DB').$$
Thus, our algorithm produces better answers than restarting the computation from the scratch on $DB'$, while saving time by not wasting the computation done so far.

Furthermore, we are able to clearly identify the answers which happen to be optimal with respect to $DB$, i.e. belong to $SWAns(A, DB)$.

In the following we provide a description of our fault-tolerant algorithm.

We assume that the network infrastructure provides a fault-detection service, in which any process can subscribe in order to be informed of the failure of the processes of interest. Such fault-detection service might be as simple as a ping command, and its existence is the common assumption in constructing fault-tolerant algorithms (cf. [39]). We make each process subscribe to the fault-detection service and be informed of the health of its neighbors only.

Now, we are ready to present our fault-tolerant algorithm. First, we introduce an additional status value for the tuples. This new value is $gone$, and is given to a tuple when the process of its key or via component has failed. Thus, the algorithm deals now with tuples whose status can be $optimal$, $provisional$, or $gone$.

Figure 2.5: Tuple Status Diagram.
Figure 2.5 illustrates these three different status values and the transitions among them.

A tuple might start with one of the three possible status values. If a tuple is (or becomes) *optimal* it preserves this status until the end of the algorithm. On the other hand, a tuple with a *provisional* status may later have a status change to *optimal* or *gone*. Similarly, a tuple with a *gone* status may later have a status change to *optimal* or *provisional*. In the end, only tuples with an *optimal* or *gone* status will be in the tables of the $p_0$-tasks across processes.

Each process keeps track of its failed neighbors in a list. Suppose that a process $P_a$ detects a failed neighbor, say $P_b$. Thus, $P_a$ first adds a failure record for process $P_b$ in its failed-neighbor list. Then, $P_a$ changes the status of all *provisional* tuples having $b$ in their key or via component to *gone* in all of its tasks. In our fault tolerant algorithm, we assign these jobs to a new thread called **Failure Detection**.

Regarding the other threads, they change as follows.

In the **Initialization** thread, we set to a *gone* status all the tuples having their key component refer to a failed process. Since in this thread, the process of the key is a neighbor process, we can easily determine its health by consulting the list of failed neighbors. The same is also done in the **Task Creation** thread when the table of a new task is being initialized.

The **Expansion** thread remains unchanged and continues to expand only *provisional* tuples.

In the **Reply** thread, we make the process send replies also in the case when it is
asked to provide information about tuples with a \textit{gone} status.

In the \textbf{Update} thread, the tuple under expansion might get an \textit{optimal} or \textit{gone} status. Also in this thread, a \textit{provisional} or \textit{gone} tuple carried in the reply message can relax a \textit{provisional} or \textit{gone} tuple with the same key in the table of the receiver task. We note that, the incoming \textit{provisional} tuples can relax \textit{provisional} or \textit{gone} tuples. Similarly, the incoming \textit{gone} tuples can relax \textit{provisional} or \textit{gone} tuples. Thus, as shown in Figure 2.5, we have transitions from a \textit{provisional} status to a \textit{gone} status and vice-versa.

In the modified \textbf{Reply} and \textbf{Update} threads, the \textit{gone}-status tuples are treated as being \textit{optimal} ones. These tuples are backpropagated in a similar way in reply messages causing along the way, through the \textbf{Update} threads, other tuples (in other processes) to attain a \textit{gone} status. For example, suppose as above that \( P_a \) detects the failure of its neighbor \( P_b \). The neighbor processes of \( P_a \), having a \((b, \_)-\)keyed \textit{provisional} tuple with an \((a, \_)-\) as their via component, will eventually assign a \textit{gone} status to this tuple. Specifically, this will happen when such tuples are expanded and \( P_a \) is asked for its \((b, \_)-\)keyed tuple.

We emphasize that a \textit{gone} status prevents tuples from being expanded by the process. Nevertheless, the weight and via of a \textit{gone} status tuple might be updated to some lower values as an effect of the expansion of some \textit{provisional} tuple in the same task. Such an update might also change the status of the tuple, as we explained above, from \textit{gone} to \textit{provisional}, thus making the expansion of the tuple possible again. Also, through such updates, a \textit{gone} status tuple can even attain an \textit{optimal}
status.

Finally, we note that the message complexity of our extended algorithm is the same as that of Algorithm 1.\(^3\)

Formally, our fault tolerant algorithm is given in the following, where we emphasize only the changes and extensions to Algorithm 1. The parts that remain unchanged are shown in gray.

**Algorithm 2.**

**Input:**

1. A database \(DB\). For simplicity we assume that each database object, say \(a\), is being serviced by a dedicated process for that object \(P_a\).

2. A query WFSA \(A = (P, \Delta, \tau, p_0, F)\).

**Output:** Set \(\text{eval}(A, DB)\) which will be characterized in Theorem 8.

**Method:**

1. Each process subscribes to the fault-detection service.

2. Each process \(P_a\) creates a list, called \(\text{FailList}_a\), and initializes it to \(\emptyset\).

3. **Initialization:** Each process \(P_a\) creates a task \((p_0, \text{passive}, \{\ldots\})\) for itself. The table \(\{\ldots\}\) (referred to as \(P_a.p_0.T\)) is initialized as follows:

\[(a)\] insert tuple \([((a, p_0), (a, p_0), 0, opt],\) and

\[^3\text{We do not consider the elementary messages of the fault-detection infrastructure.}\]
(b) For each edge-transition match, 

\((a, R, r, b)\) in \(DB\) and 

\((p_0, R, k, p)\) in \(A\),

insert tuple \([((b, p), (b, p), k \cdot r, prov)]\)

(if there are multiple \((a, \_\_\_, b)\) – \((p_0, \_\_\_, p)\) edge-transition matches, 
then the cheapest weight product is considered.)

(c) For each tuple in the task,

if the process of its key component is found in \(FailList_a\),

then change the status of the tuple to \(gone\).

If at point (b) there is no edge-transition match, then make the status of 
the \(p_0\)-task \(completed\).

4. Concurrently execute all the five following threads at each process in parallel until termination is detected. [For clarity, we describe the threads at two processes, \(P_a\) and \(P_b\).]

5. Expansion: [At process \(P_a\)]

(a) Select a \(passive\) \(p_x\)-task for processing. Make the status of the task \(active\).

(b) Select the cheapest \(provisional\)-status tuple, say \([((c, p_z), (b, p_y), w, prov)]\) 
from table \(P_a.p_x.T\).

(c) Request \(P_b\), with respect to state \(p_y\), to provide information about 
\((c, p_z)\).
For this, send a message \((p_y, [p_x, (c, p_z), w_{ab}])\) to \(P_b\), where \(w_{ab}\) is the cost of going from \((a, p_x)\) to \((b, p_y)\), which is equal to the weight of the \((b, p_y)\)-keyed tuple in \(P_a.p_x.T\).

(d) Sleep, with regard to \(p_x\)-task, until the reply message for \((c, p_z)\) comes from \(P_b\).

6. Task Creation: [At process \(P_b\)]

Upon receiving a message \((p_y, [p_x, (c, p_z), w_{ab}])\) from \(P_a\):

if there is not yet a \(p_y\)-task,

then create a task \((p_y, \text{passive}, \{\ldots\})\) and initialize its table similarly as in the first phase.

That is,

(a) insert tuple \([(b, p_y), (b, p_y), 0, \text{opt}]\), and

(b) For each edge-transition match,

\((b, R, r, d)\) in \(DB\) and

\((p_y, R, k, p_u)\) in \(A\),

insert tuple \([(d, p_u), (d, p_u), k \cdot r, \text{prov}]\)

(if there are multiple \((b, \ldots, d)-(p_y, \ldots, p_u)\) edge-transition matches, then the cheapest weight product is considered.)

(c) For each tuple in the task,

if the process of the key component is in \(\text{FailList}_b\),

then change the status of the tuple to \textit{gone}.
Also, establish a virtual communication channel with \( P_a \). This channel relates the \( p_y \)-task of \( P_b \) with the \( p_x \)-task of \( P_a \). Further, it is indexed by \((c, p_z)\) and is weighted by \( w_{ab} \) (the weight included in the received message).

**else** [\( P_b \) has already a \( p_y \)-task.] Do not create a new task, but only establish a communication channel with \( P_a \) as described above.

7. **Reply:** [At process \( P_b \)]

When in the \( p_y \)-task, the tuple \([(c, p_z), (\_\_\_), \_\_\_\_\_\_\_\_\_\_\_\_\_\_)\) is or becomes optimally weighted, or if it has **gone** status, reply back to all the neighbor processes, which have sent a task requesting message \( \langle p_y, [\_\_\_, (c, p_z), \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] \rangle \) to \( P_b \).

For example, \( P_b \) sends to such a neighbor, say \( P_a \), through the corresponding communication channel, the message \( \langle P_b.p_y.T^* \rangle \), which is table \( P_b.p_y.T \) after adding the channel weight to the weight of each tuple.

8. **Update:** [At process \( P_a \)] Upon receiving a reply message \( \langle P_b.p_y.T^* \rangle \) from a neighbor \( P_b \) w.r.t. the expansion of a \((c, p_z)\)-keyed tuple in table \( P_a.p_x.T \) do:

(a) Change the status of \((c, p_z)\)-keyed tuple to the status of the same keyed tuple in \( P_b.p_y.T^* \). \(^4\)

(b) For each tuple \([(d, p_u), (\_\_\_), v, s]^{5} \) in \( P_b.p_y.T^* \), which has a smaller

\(^4\)This status is either **optimal** or **gone**.

\(^5\)s can be **prov** or **gone**.
weight \((v)\) than a same keyed tuple \([(d, p_a), (\_ , \_), \_, s')\] in \(P_{a, p_x, T}\), replace the latter by \([(d, p_a), (b, p_y), v, s]\).

(c) Add to \(P_{a, p_x, T}\) all the rest of the \(P_{b, p_y, T^*}\) tuples, i.e., those which do not have corresponding same-key tuples in \(P_{a, p_x, T}\). Also, change the via component of these tuples to be \((b, p_y)\).

(d) if the \(p_x\)-task does no longer have *provisional* tuples,

\[\text{then }\text{ make its status }\text{completed}.\]

If \(p_x = p_0\), then report that all query answers from \(P_a\) have been computed.

\[\text{else }\text{ make the status of the }p_x\text{-task }\text{passive}.\]

9. **Failure Detection:** [At process \(P_a\) upon detecting failure of a neighbor process \(P_b\)]

(a) Add an entry in \(\text{FailList}_a\) for process \(P_b\).

(b) For each *provisional* tuple in each task of \(P_a\),

\[\text{if the key or via component is } (b, \_),\]

\[\text{then change the status of the tuple to } \text{gone}.\]

\(\square\)

As for Algorithm 1, the termination happens when each process has a *completed* \(p_0\)-task. Detecting this can be done by using an algorithm for fault-tolerant distributed

\(^6\)\(s'\) can be *prov* or *gone*.\)
termination detection (cf. [39]). Finally upon termination, we set
\[ \text{eval}(A, DB) = \{(a, b, r) : [(b, p_y), (_, _), r, s] \in P_a.p_0.T, p_y \in F \text{ and } s \in \{\text{opt, gone}\}\}. \]

Let \( C \) and \( C' \) be the Cartesian products of databases \( DB \) and \( DB' \) with query automaton \( A \). We show the following two lemmas.

**Lemma 3.** If there exists \((a, b, r) \in \text{eval}(A, DB)\) then there exists a path, in \( C \), from \((a, p_0)\) to \((b, p_y)\), where \( p_y \) is a final state of \( A \).

**Proof.** By the definition of \( \text{eval}(A, DB) \), if \((a, b, r) \in \text{eval}(A, DB)\), then there will exist a tuple 
\[ [(b, p_y), (_, _), r, s], \text{ where } s \in \{\text{opt, gone}\}, \text{ in } P_a.p_0.T. \]

Now, by the specification of the Initialization, Expansion and Update threads, it is clear that if there is no path from \((a, p_0)\) to \((b, p_y)\) in \( C \), then a \([(b, p_y), (_, _), r, s] \) tuple would never be in \( P_a.p_0.T \), and this would be a contradiction. \( \Box \)

**Lemma 4.** Let \((a, p_0)\) and \((b, p_y)\) be connected in \( C' \), and let \( r \) be the weight of a cheapest path between these two nodes (in \( C' \)). Then, there exists a triple \((a, b, r')\) in \( \text{eval}(A, DB) \), and \( r' \leq r \).

**Proof.** Since \((a, p_0)\) and \((b, p_y)\) are connected in \( C' \), they were never disconnected during the execution of the algorithm, and thus, Lemma 1 holds (with respect to Algorithm 2). Similar to Corollary 1, we have that eventually there will exist tuple 
\[ [(b, p_y), (_, _), r', s] \text{ in } P_a.p_0.T, \text{ where } s \in \{\text{opt, gone}\}. \] Value \( r' \) is the weight of the
cheapest paths that Algorithm 2 could explore, and clearly this set of paths is a superset of paths from $(a, p_0)$ to $(b, p_y)$ in $C'$. Hence, $r' \leq r$.  

Now, we show that

**Theorem 8.** $SWAns(A, DB) \supseteq eval(A, DB) \supseteq SWAns(A, DB')$.

**Proof.**

"$SWAns(A, DB) \supseteq eval(A, DB)$." Let $(a, b, r) \in eval(A, DB)$. By Lemma 3, this means that there exists a path $\pi$ (in $C$) from $(a, p_0)$ to $(b, p_y)$, where $p_y$ is a final state in $A$. Since there are process failures, path $\pi$ might not be a cheapest one in $C$ going from $(a, p_0)$ to $(b, p_y)$. Let $\pi'$ be a cheapest path in $C$ with a weight of $r'$. Clearly, $r' \leq r$, and by the definition of $SWAns(A, DB)$, $(a, b, r') \in SWAns(A, DB)$.

"$eval(A, DB) \supseteq SWAns(A, DB')$." Let $(a, b, r) \in SWAns(A, DB')$. By Theorem 1, $(a, p_0)$ is connected to $(b, p_y)$ in $C'$, and the weight of the cheapest paths between these two nodes is $r$.

By Lemma 4, there exists a tuple $(a, b, r')$ in $eval(A, DB)$, and $r' \leq r$, and this concludes our proof.  

Now, we further characterize the produced query answers. Suppose that upon termination, in the table of $P_a.p_0$, we have some tuples with a $gone$ status. Let $[\langle c, p_z \rangle, \langle \_, \_ \rangle, w_a, gone]$ be the cheapest of those tuples.

We classify the tuples in $P_a.p_0.T$ as

1. tuples with smaller (or equal) weight than $w_a$, and
2. tuples with greater weight than \( w_a \).

We can show that the tuples in the first set have weights which are optimal with respect to the original database \( DB \), i.e. they belong to \( SWAns(A, DB) \).

For this, observe that at the end of the algorithm, since tuple \([(c, p_z), (\_ , \_), w_a, gone] \) is the cheapest of the tuples with a \( gone \) status, the tuples with weight less than \( w_a \) are all \emph{optimal}. They have obtained this status by the expansion of \emph{provisional} tuples, which at the time of expansion have been the cheapest ones among \emph{provisional} and \emph{gone} tuples. Since there is no \( gone \) status tuple with a weight smaller than the weight of these tuples, all the cheaper paths possibly reaching the nodes corresponding to these tuples have been already explored. Thus, reasoning similarly as in Section 2.6, these tuples attain the cheapest weight obtainable in the original \( DB \).

We can also observe that weight \( w_a \) of the cheapest \emph{gone} status tuple in \( P_{a,p_0,T} \) is optimal considering the original database \( DB \). This is because a \( gone \) status tuple has been a \emph{provisional} one earlier, and thus, the cheapest \emph{gone} tuple would have been the next tuple to be expanded if there had been no failure in the corresponding path. By Theorem 4, the weight of this tuple is optimal with respect to the original database.

Clearly, the \( w_a \) values, for \( a \in V \), can be produced as additional output in order to characterize the query answers as the above discussion suggests.

### 2.7.1 Intermittent Process Failures

Here, we discuss how Algorithm 2 can be extended to handle a dynamic scenario, where the failed processes can come back to the computation.
Let us assume that when a failed process, say $P_b$, comes back to the computation it has no information from the past. Therefore, it creates task $p_0$, initializes its failed-neighbor list, and starts expanding tuples in its $p_0$-task.

Each neighbor of process $P_b$, say $P_a$, realizing that $P_b$ is back, continues processing as follows:

1. $P_a$ deletes the $P_b$-entry in $FailList_a$ and then changes, in all its tasks, the \textit{gone} status of the tuples having $b$ in their key or via component to \textit{provisional}.

2. $P_a$ will possibly cancel the current expansion should some smaller weighted \textit{provisional} tuple become available due to a status change from \textit{gone} to \textit{provisional}. The eventual back-reply message corresponding to the canceled expansion is ignored.

3. Upon receival of some back-reply message, due to expansion of a tuple, say $[(c, p), (\cdot, \cdot), w_{ac}, prov]$, $P_a$ will not only update \textit{provisional} tuples as before, but it will also update (if applicable) the \textit{optimal} tuples having weights greater than $w_{ac}$. This is because these \textit{optimal} tuples have an optimal weight in a subset of original $DB$. By having $P_b$ back to the computation, we can (possibly) lower the weight of such \textit{optimal} tuples.

4. $P_a$ propagates the news about $P_b$ becoming alive again through neighboring relationships. In turn, all processes receiving the news about $P_b$ behave exactly as $P_a$. 
Finally, we remark that the behavior of the $P_b$’s neighbors remains the same as described above even if $P_b$ does have information from the past. The only difference is that, in this case, $P_b$ will continue processing using its stored information.

### 2.8 Conclusions

In this chapter we have presented a fully distributed algorithm for answering generalized regular path queries on graph-structured data. We have discussed in detail the complexity of our algorithm and shown that the number of messages is proportional to the number of inter-process edges in the lazy Cartesian product graph of the database and query.

Then, we presented a resilient algorithm against process failures. This algorithm can tolerate any number of process losses and possesses two desirable properties:

1. It produces answers which are at least as good as those obtainable on the remaining live processes.

2. It does not need additional, algorithm-specific, messages to achieve resilience against process losses.

Given that RPQs are an important building block of virtually all the query languages for GSD, our work is an important step towards effective and efficient solutions for distributed and resilient computation of queries on GSD.
Chapter 3

Certain Answers and Rewritings

for Local Regular Path Queries on

Graph-Structured Data

3.1 Introduction

One of the main reasons for the rapid growth of graph-structured data is the easiness of integrating disparate data sources.\(^1\) The result is colossal and heterogeneous graphs, and the imperative need is to find effective mechanisms to query them. Due to the heterogeneous nature of these graphs, their (path) structure might not be fully known to the user. The natural way of specifying patterns for exploring graphs of partially known structure is to use regular path queries. Here we revisit our people database

\(^1\)This amounts to triplestore merging after settling for a common referencing scheme (cf. Freebase.com).
example from Chapter 1 in Fig. 3.1. Even without knowing the exact structure (possible structure of the paths) one can easily specify an RPQ such as \( \ast \cdot \) affiliation, which will bring all the pairs of nodes connected by a path matching this query.

The above RPQ is “global” in the sense that it needs to be matched starting from any node in the database (cf. [6, 9, 40, 41]). However, we are sometimes interested in matching an RPQ starting from a designated node (or entry point), for instance, we might want to match the above RPQ starting from node “Ullman” only. This is the “local” variant of RPQs (cf. [27, 42, 43, 30]). It is this variant of RPQs that we consider here.

Specifically, in this chapter we focus on answering local queries (LRPQs) using views. A view provides only incomplete information about a database and the problem is to use this information to infer query answers which would “certainly” be obtained on each database consistent with the view. For example suppose that we have a view given by RPQ \( 70s \cdot (70s + 80s) \) and its “extension,” given in Fig. 3.2, which is an incomplete but sound “piece” of the original database reachable by paths starting from Ullman and matching the view RPQ definition. This is a typical example of a “data island” that an organization might make available for querying. The rest of the database might be too sensitive to be exposed, not sufficiently cleansed to become public, or not yet available at all.

Now consider the query \( (70s+80s)\ast \cdot \) affiliation which needs to be matched starting from Ullman, and assume that only the view RPQ and extension are available. In this situation we produce as answers to the query the set \( \{Purdue, Princeton\} \) of nodes.
Evidently, these answers are “certain” in the sense that we would have obtained them on the original database. On the other hand, if the query were instead \((70s)^* \cdot affiliation\), then we should produce only \(Purdue\) as an answer, but unfortunately this cannot be inferred by the given view information. This is because we do not know whether node \(Comer\) has been reached in the original database by a 70s only path. Therefore, if we want to be certain that we do not bring “bogus” answers to the query, then we should return the empty answer in this case.

The notion of certain answer is the well adopted approach for defining query answering on incomplete databases (cf. [44]). Thus, we start by defining the notion of certain answers for local RPQs over possible databases, incompletely specified by a view. Then we turn to finding an algorithm for computing the certain answer. For this we define and compute view-based rewritings which are purely algebraic artifacts based on the RPQ definitions of the query and view. A rewriting can be matched on the view extension producing a query answer set. The question is whether what is computed by a rewriting is the full certain answer or not. We prove a characterization theorem saying that this is indeed the case. This is a surprising result as the counterpart for global RPQs is not true (see [6]).

Let us underline the importance of this result. There are two complexities associated with view-based query answering, namely data complexity and expression complexity (cf. [6]). The data complexity is clearly more important because data is huge compared to the query and view definitions (RPQs). As shown in [6], computing the certain answer to global RPQs is coNP hard with respect to data complexity,
which means that there is an inherent data intractability for global RPQs. On the other hand, computing the answer set using a rewriting is polynomial, but this is only a subset of the certain answer. It is also polynomial to compute the answer set using a rewriting in our case of local RPQs. Hence, by showing that for local RPQs, what is obtained by a rewriting coincides with the certain answer, we have that computing the certain answer can be done (using a rewriting) in polynomial time in the size of the data.

![Figure 3.1: A database $D$.](image1)

![Figure 3.2: Incomplete view extension $D_\mathcal{V}$ for $\mathcal{V} = 70s(70s + 80s)$.](image2)

Our algorithm for computing view-based rewritings is an automata-theoretic approach. Its complexity is exponentially lower than what we would get by a straight-
forward adaptation of the rewriting algorithm of [9] for global RPQs. Again, we would like to emphasize the importance of this result. [9] shows that the size of their rewriting can be doubly exponential in the size of the query, and this bound is tight. While this is “expression complexity” (not data complexity), still a double exponential bound makes computing the rewriting prohibitive even for very small query sizes (in the worst case). For example, if the query size (as measured by the length of its regular expression) is, say 6, then a modern processor executing 150,000 million instructions per second would need about 3.5 computation years only to print a rewriting of size $2^{2^6}$. On the other hand, for local RPQs, we are able to compute the rewriting in single exponential time. The representation we get is a multientry deterministic finite automaton of size $2^n$, where $n$ is the query size. Not only is the size of such a rewriting quite manageable in practice (negligible, even for $n$ as large as 20), but the fact that the automaton is deterministic greatly facilitates matching of paths in the view extension.

The rest of the chapter is organized as follows. In Section 3.2 we define graph-structured databases, local RPQs, and views. In Section 3.3 we define the certain answer of the query. In Section 3.4 we define view-based rewritings for local RPQs and give our characterization theorem connecting the certain answer with rewritings. In Section 3.5 we present our algorithm for computing rewritings. In Section 3.6 we discuss related works. Finally, Section 3.7 concludes the chapter.
3.2 Graph-Structured Data and Local Regular Path Queries

We consider a database to be an edge labeled graph. The nodes of the database graph represent objects and the edges represent relationships between the objects.

Formally, let $\Delta$ be a finite alphabet. We shall call $\Delta$ the database alphabet. Elements of $\Delta$ will be denoted $r, s, \ldots$. As usual, $\Delta^*$ denotes the set of all finite words over $\Delta$. Words will be denoted by $u, w, \ldots$. We also assume that we have a universe of objects, and objects will be denoted $a, b, c, \ldots, o, \ldots$. A database $D$ over $\Delta$ is a subset of $N \times \Delta \times N$, where $N$ is a set of objects, that we usually will call nodes. We view a database as a directed labeled graph, and interpret a triple $(a, r, b)$ as a directed edge from object $a$ to object $b$, labeled with $r$. In essence a database is a set of such triples, and it is commonly called in practice a “triplestore” (cf. [3]).

If there is a path labeled $r_1, r_2, \ldots, r_k$ from $a$ to $b$, we write $a \xrightarrow{r_1 r_2 \cdots r_k} b$.

A local regular path query (LRPQ) is a pair $(o, Q)$, where $o$ is an object in $N$, and $Q$ is a regular language over $\Delta$. For the ease of notation, we will blur the distinction between regular languages and regular expressions that represent them. Let $(o, Q)$ be an LPRQ and $D$ a database. Then, the answer to $(o, Q)$ on $D$ is defined as

$$\text{ans}(o, Q, D) = \{ a : o \xrightarrow{w} a \text{ in } D \text{ for some } w \in Q \}.$$
Given an LRPQ \((o, Q)\), we represent \(Q\) by a finite state automaton (FSA)

\[
A = (P, \Delta, \tau, p_0, F)
\]

where \(P\) is the set of states, \(\Delta\) is the alphabet, \(\tau\) is the transition relation, \(p_0\) is the initial state, and \(F\) is the set of final states.

The well-known method for answering a regular path query on a given database \(D\) starting from an object \(o\) is as follows (cf. [34]). In essence, we create state-object pairs from the query automaton and the database. We first create the pair \((o, p_0)\), where \(p_0\) is the initial state in \(A\). Then, we create all the pairs \((a, p)\) such that there exist an edge from \(o\) to \(a\) in \(D\), and a transition from \(p_0\) to \(p\) in \(A\), and furthermore the labels of the transition and the edge match. In the same way, we continue to create new pairs from existing ones, until we are not anymore able to do so. In essence, what is happening is a lazy construction of a Cartesian product graph of the query automaton with the database graph. Typically, only a small part of the Cartesian product is really constructed depending on the selectivity of the query. After obtaining the above Cartesian product graph, producing query answers becomes a question of computing reachability of nodes \((p, a)\), where \(p\) is a final state, from \((p_0, o)\), where \(p_0\) is the initial state. Namely, if \((p, a)\) is reachable from \((p_0, o)\), then \(a\) is in the query answer.

Let \((o, V)\) be an LRPQ. We call this a view and associate with it an extension, which is a pair \((O_V, D_V)\), where \(D_V\) is a database and \(O_V\) is a set of objects in \(D_V\).
Intuitively, set $O_v$ represents a (possibly incomplete) answer of $(o,V)$ on some database. $D_v$ is only a “piece” of such a database, available to be further explored by LRPQs starting from objects in $O_v$. As common in data integration, $D_v$ might be “incomplete,” but it is “sound” in the sense that it does not contain “bogus” facts (object-label-object edges). These notions will be illustrated now and formalized in the next section.

**Example 1.** In Fig.3.1 we show a database $D$ representing J. D. Ullman’s academic descendants. The alphabet of edge labels comprises “Affiliation” and different decades, such as “70’s”, “80’s”, etc. Let us consider $V = 70s(70s + 80s)$. The result of evaluating $(Ullman, V)$ on $D$ is $\{Comer, Appel, Chan\}$.

An extension for this view is $\{(Comer, Appel), D_v\}$, where $D_v$ is given in Figure 3.2. Evidently, this extension is incomplete, information about Chan is not included. However, $D_v$ is sound, as it does not contain information not present in $D$.

### 3.3 Certain Answer

The notion of “certain answer” is a semantic definition of answering queries using views.

First recall that a database $D$, as defined in Section 3.2, is nothing else but a set of object-label-object triples, a triplestore.

Now, let $(O_v, D_v)$ be an extension for a view $(o, V)$.
Definition 1. A database $D$ is a possible database if

1. $O_V \subseteq \text{ans}(o, V, D)$

2. $D_V \subseteq D$.

In such a case we say that the view is sound. We remark that it is soundness that is typically required for views in data-integration scenarios (cf. [45]).

We denote the set of possible databases by

$$\text{poss}((o, V), (O_V, D_V)).$$

Example 2. In Fig. 3.3 (a) we show an extension for view $(o, r + s)$ with $O_V = \{a_1\}$. In (b) we show two possible databases.

Let $(o, Q)$ be an LRPQ.

Definition 2. The certain answer to $(o, Q)$, given view $(o, V)$ and its extension $(O_V, D_V)$, is defined as

$$\text{ANS}((o, Q), (o, V), (O_V, D_V)) = \bigcap_{D \in \text{poss}((o, V), (O_V, D_V)) \text{ans}(o, Q, D)}.$$

3.4 View-Based Rewritings

The certain answer is a “semantic definition.” The question now is how to compute the certain answer. For this we will introduce view-based rewritings which can be
Figure 3.3: (a) $D_V (O_V = \{a_1\})$. (b) Two possible databases for view $(o, r + s)$.

evaluated on the view extension. In general, for global regular path queries, the answers we get from rewritings do not constitute the whole certain answer, but rather only a subset of it (see [6]). Calvanese, De Giacomo, Lenzerini, and Vardi showed in [6] that computing the certain answers of regular path queries is co-NP-hard with respect to data complexity. On the other hand, computing view-based query answers by evaluating view-based rewritings on the view extension is polynomial.

Interestingly, as we show, for the case of LRPQs, the answer set obtained using rewritings coincides with the certain answer.

First, let us define view-based rewritings. They are purely algebraic artifacts, thus we only use $Q$ and $V$ in their definition.
Definition 3. A language $R$ is a contained rewriting of $Q$ using $V$ if $V \cdot R \subseteq Q$.

Definition 4. A rewriting $R$ is maximal if $R' \subseteq R$ for any rewriting $R'$.

We denote the maximal rewriting of $Q$ using $V$ by $Q_V$. This rewriting can be evaluated (answered) on a view extension $(O_V, D_V)$ giving the output

$$ans(O_V, Q_V, D_V) = \bigcup_{o \in O_V} ans(o, Q, D).$$

Now, we have the following theorem which shows that evaluating $Q_V$ on the view extension amounts to computing the certain answer of $Q$ for the given view and its extension.

Theorem 1. $ans(O_V, Q_V, D_V) = \text{ANS}((o, Q), (o, V), (O_V, D_V))$.

Proof.

“$\subseteq$”. The proof proceeds by contradiction.

Let $a \in ans(O_V, Q_V, D_V)$, but $a \notin \text{ANS}((o, Q), (o, V), (O_V, D_V))$. By the definition of certain answers, there is at least one possible database $D$, with respect to $(o, V)$ and $(O_V, D_V)$, such that $a \notin ans(o, Q, D)$. Since $Q_V$ is a contained rewriting, i.e. $V \cdot Q_V \subseteq Q$, we have $a \notin ans(o, V \cdot Q_V, D)$. Based on this and the fact that $D_V \subseteq D$, we get that $a \notin ans(O_V, Q_V, D_V)$, which is a contradiction.

“$\supseteq$”.

Let $a \in \text{ANS}((o, Q), (o, V), (O_V, D_V))$. This means that $a \in ans(o, Q, D)$ for each possible database $D$. Recall that, since the view is sound, $D_V \subseteq D$. Let $o'$ be an
object in $O_V$, and $w$ be a word in $V$. Now we build a database $D^w$ rooted at $o$, containing $D_V$ and having paths spelling $w$ which connect $o$ to each $o'$ in $O_V$ (recall $o'$ is an object of $D_V$). Clearly, $D^w$ is a possible database, and $\text{ans}(o, V, D^w) \supseteq O_V$. As $D^w$ is a possible database, $a \in \text{ans}(o, Q, D^w)$. Since there is only one path going from $o$ to each $o' \in O_V$ (which spells a word in $V$), and since $Q_V$ is a maximal rewriting, we have that

$$a \in \bigcup_{o' \in O_V} \text{ans}(o', Q_V, D^w).$$

On the other hand, since the path from $o$ to $o'$ labeled by $w$ is of no use in obtaining any additional answer starting from $o'$, we have that $a \in \text{ans}(O_V, Q_V, D_V)$. \qed

### 3.5 Computing View-Based Rewritings

Now we present an algorithm to compute the maximal view-based rewriting.

Let $Q$ and $V$ be two regular languages over $\Delta$, and let $A = (S, \Sigma, \Delta, s_0, F)$ be a deterministic finite state automaton (DFA) for $Q$.

**Construction.**

1. For every state $s$ in $S$ define the automaton $A_{\leftarrow s} = (S, \Sigma, \Delta, s_0, \{s\})$, which is the same as $A$, but final state being only $s$. Note that $A_{\leftarrow s}$ is a DFA for each $s \in S$. 
2. Define the “multi-entry” DFA $A' = (S, \Sigma, \Delta, S_0, F)$ where

$$S_0 = \{ s : L(A \leftarrow s) \cap V \neq \emptyset \}.$$ 

Let, $A \rightarrow s = (S, \Sigma, \Delta, s, F)$ such that $s \in S$. We have that

**Theorem 2.** $Q_v = \bigcap_{s \in S_0} L(A \rightarrow s)$ is a maximal rewriting of $Q$ using $V$.

**Proof.** Since $A$ is a complete DFA, it never gets stuck. As such, we have that $V \subseteq \bigcup_{s \in S} L(A \leftarrow s)$. That is, for each word in $V$ there is a (transition) path spelling this word and going from $s_0$ to some state $s$ in $A$ (s could be a “dead state”). Based on this fact and the construction, $V \cdot Q_v \subseteq Q$, that is, $Q_v$ is a contained rewriting of $Q$ using $V$. Observe that if some word of $V$ takes DFA $A$ to a dead state, then $V \cdot Q_v$ is empty.

We prove the maximality of $Q_v$ by contradiction. Let $R$ be another rewriting of $Q$ such that $R \notin Q_v$. Let $w$ be a word such that $w \in R$ but $w \notin Q_v$. Since $R$ is a rewriting, $V \cdot R \subseteq Q$, and thus, there exists a state $s_i$ in $A$ such that $w \in L(A \rightarrow s_i)$. Now, two cases can be considered.

1. $s_i \notin S_0$. As $R$ is a contained rewriting, we should still have that $L(A \leftarrow s_i) \cap V \neq \emptyset$, and this is in contradiction to our construction.

2. $s_i \in S_0$ but $w \notin \bigcap_{s \in S_0} L(A \rightarrow s)$. In this case, there is at least one state $s_j \in S_0$ such that $w \notin L(A \rightarrow s_j)$. Suppose $\{v_i, v_j\} \subseteq V$ such that $v_i \in L(A \leftarrow s_i)$ and $v_j \in L(A \leftarrow s_j)$. Therefore, $V \cdot R \ni v_i, w \notin Q$ which contradicts the fact of $R$
being a rewriting of $Q$.

Example 3. Fig. 3.4 shows a deterministic query automaton $A$. Suppose that we want to rewrite this query using $V = r + s$.\footnote{Recall that for ease of notation we blur the distinction between regular languages and the regular expressions representing them.} By applying the above algorithm we have that $S_0 = \{q_1, q_2\}$, $L(A \rightarrow q_1)$ is $r^*s(r + sr^*)^r$, and $L(A \rightarrow q_2)$ is $(r + s)r^* = r^* + sr^*$. Now, the rewriting is $Q_V = L(A \rightarrow q_1) \cap L(A \rightarrow q_2)$ which is $sr^*$.

Regarding the complexity of our rewriting algorithm, it is polynomial in the size of a DFA for the query. If the query is given by a regular expression or NFA, then constructing a DFA can be exponential, and thus our algorithm is exponential for such query representation.

As a lower bound we have the following.

Theorem 3. Computing $Q_V$, given the query and view NFAs, is PSPACE-hard.

Proof. We present here a reduction from the problem of regular language (NFA) containment which is PSPACE-complete. Let $E$ and $F$ be two regular languages (given by NFAs or regular expressions). In order to decide whether $E \subseteq F$, take $V = S_ES$ and $Q = S_F$, where $S$ is a symbol not in the alphabet of $E$ and $F$. Now, we have that $E \subseteq F$ if and only if $\epsilon \in Q_V$. The latter can be decided in polynomial time in the size of $Q_V$. This proves our claim that computing $Q_V$ is PSPACE-hard.\footnote{Recall that for ease of notation we blur the distinction between regular languages and the regular expressions representing them.}

Finally, matching a multi-entry DFA on a view extension is polynomial in the size of the data. For this, it is sufficient to match the automaton starting from each
initial state and then intersect the answer sets thus obtained. Clearly the automata
matchings (for each initial state) can also be performed in parallel.

![Query Automaton](image)

Figure 3.4: A query automaton $A$.

### 3.6 Related Work

View-based query answering and rewriting for *global* RPQs has been mainly explored
in [6, 46, 9, 47] and [40, 41, 48, 49]. In [6] and [46] Calvanese, DeGiacomo, Lenzerini,
and Vardi show that certain answering of global RPQs is co-NP hard with respect
to data complexity. In [9] the same authors present a doubly-exponential algorithm
for computing view-based rewriting for global RPQs. They prove that this algo-
rithm is optimal by showing that the size of the rewriting can be doubly-exponential
in the length of the query and this bound is tight. [49] shows experimentally that
rewritings are too big also in the average case and proposes techniques to deal with
the complexity in practice. [47] presents connections between view-based answering
and rewritings for global RPQs. [40, 41] introduce partial rewritings and present
algorithms for computing them. [48] studies RPQ containment and rewritings un-
der constraints. We remark again here that straightforward attempts to adapt the rewriting algorithms of [9, 40, 41, 48] to our case would result in a double exponential complexity.

View-based rewriting for XPath has been addressed in [50, 51, 52]. XPath is related to RPQ in which only concatenation, union, and \( \_\* \) is allowed. However, XPath is intended for node-labeled trees and contains additional mechanisms such as branching. Thus, XPath queries and RPQs have different strengths and weaknesses, the former have branching, but not full Kleene star, whereas, RPQs have full Kleene star, but not branching.

3.7 Conclusions

In this chapter we have presented a characterization result showing that, for local regular path queries, the certain answer coincides with the answer set computed by a view-based rewriting. The importance of this is that the certain answer can be computed by using a view-based rewriting in polynomial time in the size of the data. This is in contrast to the result of [6] which says that computing the certain answer to a global RPQs is intractable with respect to data complexity.

We have also presented an algorithm for computing view-based rewritings which produces a multientry DFA for the rewriting. The complexity and the output of our algorithm is an exponential order of magnitude better than the counterpart for global RPQs. Computing the query answer using a multi-entry DFA is polynomial and can
be done in parallel starting from each initial state of the automaton.
Chapter 4

Evolving Schemas for Streaming XML

4.1 Introduction

The ubiquitous theme in the modern theory of software systems is that evolution is unavoidable in real-world systems. The force of this fact is increasingly prominent today when software systems have numerous online interconnections with other systems and are more than ever under the user pressure for new changes and enhancements. It is often noted that no system can survive without being agile and open to change.

In this chapter we propose ways to evolve schemas for XML in an online, streaming setting. As XML is by now the omnipresent standard for representing data and documents on the Web, there is a pressing need for having the ability to smoothly adapt schemas for XML to deal with changes to business requirements, and exchange
One important use of schemas for XML is the validation of documents, which is checking whether or not a document conforms to a given schema. Notably, the validation is the basis of any application involving data-exchange between two or more parties.

Due to various changing business requirements, the schemas of the sending parties could diverge from the schemas of the receiving parties. In such a case, we need to “expand” the receivers’ schemas by making them more “tolerant” against incoming XML documents.

To address the problem, in this chapter we make the receiving systems adapt by evolving their schemas using language operations of insertion and deletion. We present an example here to illustrate our framework.

**Example 1.** Suppose a user collects information about day-specials from different restaurants. She has a basic XML schema which is used to validate arriving XML documents about day-specials. The schema is simple. The root is a day-specials element with eight children: three soup elements, three main-dish elements, and two dessert elements. We abbreviate the above by $ds$, $sp$, $md$, and $dt$, respectively. Furthermore, each of $sp$, $md$, and $dt$ has a name and a price. We abbreviate these last two elements by $n$ and $p$, respectively. Now an expression capturing our simple schema in terms of opening and closing tags is

$$ds (sp n p sp p sp)^3 (md n p md p md)^3 (dt n p dt dt)^2 ds.$$
Of course, in general we need a more powerful mechanism than regular expressions to represent schemas. Nevertheless, the example still serves the point of illustrating our framework.

Now imagine that some restaurants deviate from this schema. For example some restaurants might provide less, whereas some others might provide more information than what the schema requires. In order to tolerate some variation from the basic schema, the user decides to evolve the schema by applying some language operations. Let us itemize some possible scenarios the user can consider.

1. The user is willing to tolerate documents deviating from the schema, but they should not miss more than two of sp, md, and dt elements. The particular combination of missing elements is not important. Whether it is two of a kind, or two of different kinds, it really does not matter for this user. The only thing of interest to him at the moment is that there is at least one soup and at least one main dish in what he receives. Apparently, the user is hungry, and he does not want to risk filtering out restaurants in his vicinity by being too picky.

Solution. We apply one or two deletions on the schema language. We specify the language of (sub)words that can be deleted by the following expression

$$sp(\text{'_'}^*)sp + md(\text{'_'}^*)md + dt(\text{'_'}^*)dt,$$

where "_" is a “don’t-care” symbol.\(^1\)

\(^1\)In the formal development of the chapter, we supply instead the set of all nested words.
2. The user is willing to tolerate documents missing the price of soups. Apparently, it is cold winter, and the user wants to have a soup at any price!

**Solution.** We apply up to three deletions of the (sub)word $p\overline{p}$, but those deletions have to be constrained to apply only inside $sp$ elements. We can specify the scope of the deletions by an XPath expression such as

$$/ds/sp.$$ 

3. The user is willing to tolerate documents containing more dessert items than what the schema allows. Namely, he would not mind seeing even 10 more desserts in the day-specials! Probably, the user is a child who loves desserts. Naturally, he does not want to be bothered with extra soups or main dishes.

**Solution.** We apply up to 10 insertions into the schema language. We specify the language of words that can be inserted by

$$dt(^*)\overline{dt}.$$ 

4. Finally, the user, being misled several times in the past by fancy main-dish names, is willing to tolerate, and even encourage, documents containing more complete main dish information, for example additionally to name and price, information about their ingredients (abbreviated by $ig$).
Solution. We apply up to three insertions into the schema language. We specify the language of words that can be inserted by

\[ ig(\_\_\_)_i g. \]

The insertions have to be constrained to apply only inside md elements. We can specify the scope of the insertions by an XPath expression such as

\[ /ds/md. \]

Deletions and Insertions. Interestingly, language deletions and insertions have been studied as operators for regular languages in representing biological computations (cf. [53, 54]). In this chapter, we investigate the deletion and insertion operators as means for evolving languages of nested words capturing the common schema formalism for XML.

As illustrated in the above example, we also consider constrained variants of these language operators. Specifically, we provide means for specifying that we want to allow an operation to apply only at certain elements of the XML documents. For instance, in our example, we specified that the price deletions and ingredient insertions were allowed to occur only inside soup and main-dish elements, respectively, and not anywhere else.

Schemas for XML. As mentioned in the Introduction, when it comes to XML
schema specifications, the most popular ones are Document Type Definition (DTD), XML Schema ([11]) and Relax NG ([12]). Notably, all these schema formalisms can be captured by Extended Document Type Definitions (EDTDs) (cf. [55, 56, 13, 57]). It is well known that the tree languages specified by EDTDs coincide with (unranked) regular tree languages (cf. [57]).

In this chapter, we will represent XML schemas by Visibly Pushdown Automata (VPAs) introduced in [58]. VPAs are in essence pushdown automata, whose push or pop mode can be determined by looking at the input only (hence their name). VPAs recognize Visibly Pushdown Languages (VPLs), which form a well-behaved and robust family of context-free languages. VPLs enjoy useful closure properties and several important problems for them are decidable. Furthermore, VPLs have been shown to coincide with the class of (word-encoded) regular tree languages, i.e. VPAs are equivalent in power with EDTDs. Recent work [59] has also shown that EDTDs can be directly compiled into VPAs in polynomial time.

Now, the validation problem reduces to the problem of accepting or rejecting the XML document (string) using a VPA built for the given schema. Notably, a VPA accepts or rejects an XML document without building a tree representation for it, and this is a clear advantage in a streaming setting, where transforming and storing the XML into a tree representation is a luxury we do not have.

Another reason for preferring VPAs over tree automata for XML is that VPAs are often more natural and exponentially more succinct than tree automata when it comes to “semi-formally” specify documents using pattern-based conditions on the
global linear order of XML (cf. [60, 61]).

Also, considering the schemas for XML as word languages opens the way for a natural extension of deletion and insertion operations, thus making the schemas evolve in a similar spirit to biological computing artifacts.

We show that the deletion and insertion operations can be efficiently computed for VPLs, and furthermore they can be combined with useful constraints determining the scope of their applications.

**From language operations to schema evolution.** Let us make more precise the scenario which we are dealing with in this chapter. We investigate the problem from the receivers’ point of view. Each receiver has already an original schema represented as a VPA and is willing to allow certain “evolutions,” which she specifies using allowed (language) operations. We apply these operations on the receiver’s schema modeled as a VPA, and do not modify the arriving documents.

**Contributions.** More specifically, our contributions in this chapter are as follows.

1. We show that VPLs are closed under the language operations of deletion and insertion. This is in contrast to Context-Free Languages which are not closed under deletion, but only under insertion.

2. We introduce the extended operations of $k$-bounded deletion and insertion which allow the deletion and insertion, respectively, of $k$ words in parallel. It is exactly these operations that are practical to use for evolving schemas for XML documents containing (or need to contain) multiple occurrences of words to be
deleted (or inserted). For instance, a day–specials schema contains \textit{price} for various items, and all these prices might need to be deleted. We show that the VPLs are closed under these extended operations as well.

3. We present an algorithm, which, given a schema VPL $L$, two sets $D$ and $I$ of allowed language deletions and insertions, respectively, and a positive integer $k$, produces a (succinctly represented) expanded language $L'$ by applying not more than $k$ operations in \textit{parallel} from $D \cup I$ on $L$. Language $L'$ contains \textit{all} the possible “$k$-evolution” of $L$ using operations from $D \cup I$. The difference from the $k$-bounded deletion and insertion is that now we allow these operations to be intermixed together.

4. We enhance the deletion and insertion operations by constraints that specify the allowed scope of the operations. We illustrate such constraints by using XPath expressions which select the XML elements of interest. We present an algorithm which computes the “$k$-evolution” of a given schema VPL $L$ in this constrained setting. The challenge is to be able to first mark non intrusively all the candidate spots for applying the operations, and then apply them. This is because applying an operation could possibly change the structure of the words and thus harm matching of the other constraints.

\textbf{Organization.} The rest of the chapter is organized as follows. In Section 4.2 we discuss related work. Section 4.3 reviews VPAs and VPTs (Visibly Pushdown Transducers). In Section 4.4.1 we study the deletion operation for VPLs. The $k$-bounded
deletion for VPLs is also introduced there. In Section 4.4.2 we study the insertion operation for VPLs. The $k$-bounded insertion for VPLs is also introduced there. In Section 4.4.3 we present an algorithm for evolving a schema VPL by applying at most $k$ language operations in parallel. In Section 4.5 we introduce constrained operations, and in Section 4.5.4 we present an algorithm to evolve schema VPLs using such operations. Finally, Section 4.6 concludes the chapter.

4.2 Related Work

The first to propose using pushdown automata for validating streaming XML are Segoufin and Vianu in [62]. The notion of auxiliary space for validating streaming XML is also defined in this work. Auxiliary space is the stack space needed to validate an XML document and is proportional to the depth of the document.

VPLs and their recognizing devices, VPAs, are introduced in [58]. Logic-based characterizations are provided in [63, 64]. In [59], it is argued that VPAs are the apt device for the validation of streaming XML and a direct construction is given for going from EDTDs to equivalent VPAs. Visibly Pushdown Transducers (VPTs) are introduced in [65] and [66]. The latter showed that VPLs are closed under transductions of VPTs which refrain from erasing open or close symbols. In this chapter we will use this class of VPTs for some auxiliary marking operations on VPLs.

The problem of error-tolerant validation has been studied in several works (cf. [67, 68, 69, 65]). These works use edit operations$^2$ to modify the XML documents and

\footnote{Using edit operations on regular path queries (RPQs) is studied in [70, 71, 24]. They revolve}
possibly make them fit the schema. The difference of our work from these works is that we consider language operations on the schema rather than edit operations on XML documents (trees).

We note that performing edit operations might not always capture the user intention of changing an XML document or schema. For example to delete a complex soup element we need several delete edit operations rather than just one language operation as in our setting. The latter, we believe, depending on the application, better captures a user intention of deleting such an element in one-shot. Furthermore, with our language operations, the user is given the opportunity to specify the structure of the elements to be deleted or inserted, which, as illustrated in the Introduction, is useful in practice.

The language operations of deletion and insertion are studied by Kari in [53] for regular and context free languages. As shown there, the regular languages are closed under deletion and insertion while context free languages are not closed under deletion, but closed under insertion. In this chapter we show that VPLs are closed under both deletion and insertion.

[53] also presents controlled operations (for regular languages) which restrict the insertion and deletion to take place only after or before specified symbols in the words of a language. These controlled operations have limited power in comparison to the constrained operations that we present in this chapter. The former do not consider the position of the control symbol, whereas the latter (besides being VPL operations) around the problem of finding paths in graph databases that approximately spell words in a given regular language.
can have a full control specification of where to perform the operation.

Evolution is studied in [72], [73], [74]. They present update operations for XML Schemas, regular tree grammars, and DTDs, respectively. Their operations are edit operations rather than language operations presented in our work.

In short, the novelties of our work compared to other works are: (1) we investigate language operations rather than edit operations, and (2) we consider VPA’s for XML schemas, thus solving the problem in streaming scenario.

4.3 Visibly Pushdown Automata and Transducers

4.3.1 Visibly Pushdown Automata

VPAs were introduced in [58] and are a special case of pushdown automata. Their alphabet is partitioned into three disjoint sets of call, return and local symbols, and their push or pop behavior is determined by the consumed symbol. Specifically, while scanning the input, when a call symbol is read, the automaton pushes one stack symbol onto the stack; when a return symbol is read, the automaton pops off the top of the stack; and when a local symbol is read, the automaton only moves its control state.

Formally, a visibly pushdown automaton (VPA) $A$ is a 6-tuple $(Q, (\Sigma, f), \Gamma, \tau, q_0, F)$, where

1. $Q$ is a finite set of states.
2. \(\Sigma\) is the alphabet partitioned into the (sub) alphabets \(\Sigma_c, \Sigma_l\) and \(\Sigma_r\) of call, local and return symbols respectively.

\(\bullet\) \(f\) is a one-to-one mapping \(\Sigma_c \to \Sigma_r\). We denote \(f(a)\), where \(a \in \Sigma_c\), by \(\bar{a}\), which is in \(\Sigma_r\).\(^3\)

3. \(\Gamma\) is a finite stack alphabet that (besides other symbols) contains a special “bottom-of-the-stack” symbol \(\bot\).

4. \(q_0\) is the initial state.

5. \(F\) is the set of final states.

6. \(\tau = \tau_c \cup \tau_r \cup \tau_l \cup \tau_\epsilon\) is the transition relation and \(\tau_c, \tau_l, \tau_r\) and \(\tau_\epsilon\) are as follows.

\(\bullet\) \(\tau_c \subseteq Q \times \Sigma_c \times Q \times (\Gamma \setminus \bot)\)

\(\bullet\) \(\tau_r \subseteq Q \times \Sigma_r \times \Gamma \times Q\)

\(\bullet\) \(\tau_l \subseteq Q \times \Sigma_l \times Q\)

\(\bullet\) \(\tau_\epsilon \subseteq Q \times \{\epsilon\} \times Q\)

We note that the \(\epsilon\)-transitions do not affect the stack and they behave like in an NFA. They can be easily removed by an \(\epsilon\)-removal procedure similar to the standard one for NFAs. However, we consider \(\epsilon\)-transitions as they make expressing certain constructions more convenient.

\(^3\)When referring to arbitrary elements of \(\Sigma_r\), we will use \(\bar{a}, \bar{b}, \ldots\) in order to emphasize that these elements correspond to \(a, b, \ldots\) elements of \(\Sigma_c\).
A run of VPA $A$ on word $w = x_0 \ldots x_{k-1}$ is a sequence $\rho = (q_i, \sigma_i), \ldots, (q_{i+k}, \sigma_{i+k})$, where $q_{i+j} \in Q$, $\sigma_{i+j} \in (\Gamma \setminus \{\bot\})^* \cdot \{\bot\}$, and for every $0 \leq j \leq k - 1$, the following holds:

- If $x_j$ is a call symbol, then for some $\gamma \in \Gamma$, $(q_{i+j}, x_j, q_{i+j+1}, \gamma) \in \tau_c$ and $\sigma_{i+j+1} = \gamma \cdot \sigma_{i+j}$ (Push $\gamma$).

- If $x_j$ is a return symbol, then for some $\gamma \in \Gamma$, $(q_{i+j}, x_j, \gamma, q_{i+j+1}) \in \tau_r$ and $\sigma_{i+j} = \gamma \cdot \sigma_{i+j+1}$ (Pop $\gamma$).

- If $x_j$ is a local symbol, then $(q_{i+j}, x_j, q_{i+j+1}) \in \tau_l$ and $\sigma_{i+j+1} = \sigma_{i+j}$.

- If $x_j$ is $\epsilon$, then $(q_{i+j}, x_j, q_{i+j+1}) \in \tau_\epsilon$ and $\sigma_{i+j+1} = \sigma_{i+j}$.

A run is accepting if $q_i = q_0$, $q_{i+k} \in F$, and $\sigma_{i+k} = \bot$. A word $w$ is accepted by a VPA if there is an accepting run in the VPA which spells $w$. A language $L$ is a visibly pushdown language (VPL) if there exists a VPA that accepts all and only the words in $L$. The VPL accepted by a VPA $A$ is denoted by $L(A)$.

When reasoning about XML structure and validity, the local symbols are not important, and thus, we consider the languages of XML schemas as VPLs on the alphabet $\Sigma_c \cup \Sigma_r$. Furthermore, we note that here, we are asking for an empty stack in the end of an accepting run because we are interested in VPLs of properly nested words.

**Example 2.** Suppose that we want to build a VPA accepting XML documents about movie collections. Such documents will have a collection element nesting any (non-
zero) number of movie elements in them. Each movie element will nest a title element and any number of star elements. A VPA accepting well-formed documents of this structure is \( A = (Q, (\Sigma, f), \Gamma, \tau, q_0, F) \), where

\[
Q = \{ q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7 \},
\]

\[
\Sigma = \Sigma_c \cup \Sigma_r = \{ \text{collection, movie, title, star} \} \cup \{ \overline{\text{collection, movie, title, star}} \},
\]

\( f \) maps the \( \Sigma_c \) elements into their “bar”-ed counterparts in \( \Sigma_r \),

\[
\Gamma = \{ \gamma_c, \gamma_m, \gamma_t, \gamma_s \} \cup \{ \bot \},
\]

\[
F = \{ q_7 \},
\]

\[
\tau = \{(q_0, \text{collection}, q_1, \gamma_c), (q_1, \text{movie}, q_2, \gamma_m), (q_2, \text{title}, q_3, \gamma_t),
(q_3, \overline{\text{title}}, q_4), (q_4, \text{star}, q_5, \gamma_s), (q_5, \overline{\text{star}}, q_6, \gamma_s, q_4),
(q_4, \overline{\text{movie}}, \gamma_m, q_6), (q_6, \text{collection}, \gamma_c, q_7), (q_6, \epsilon, q_1) \}.
\]

We show this VPA in Fig. 4.1.

![Figure 4.1: Example of a VPA.](image)

Processing a document with a VPA. As explained in [59], given a schema speci-
fication VPA \( A = (Q, (\Sigma, f), \Gamma, \tau, q_0, F) \), the (exact) validation of an XML document (word) \( w \) amounts to accepting or rejecting \( w \) using \( A \).

**Intersection with Regular Languages.** It can be shown that VPLs are closed under the intersection with regular languages. The construction is similar to the one showing closure of CFLs under intersection with regular languages. Formally we have

**Theorem 1.** Let \( L \) be a VPL and \( R \) a regular language. Then, \( L \cap R \) is a VPL.

**Proof.** Let \( A = (Q, (\Sigma, f), \Gamma, \tau_A, q_0, F) \) be an \( \epsilon \)-free VPA for \( L \), and \( B = (P, \Sigma, \tau_B, p_0, G) \) an \( \epsilon \)-free NFA for \( R \). Now, language \( L \cap R \) is accepted by the product VPA \( C = (Q \times P, (\Sigma, f), \Gamma, \tau_C, (q_0, p_0), F \times G) \), where

\[
\tau_C = \{(q, p, a, (q', p'), \gamma) : (q, a, q', \gamma) \in \tau_A \text{ and } (p, a, p') \in \tau_B\} \cup \\
\{(q, p, a, \gamma, (q', p')) : (q, a, \gamma, q') \in \tau_A \text{ and } (p, a, p') \in \tau_B\} \cup \\
\{(q, p, l, (q', p')) : (q, l, q') \in \tau_A \text{ and } (p, l, p') \in \tau_B\}.
\]

The Language of All Properly Nested Words. In our constructions we will often use the language of all properly nested words which we denote by \( PN \). All the VPLs of properly nested words are subsets of it. We consider the empty word \( \epsilon \) to also be in \( PN \).
4.3.2 Visibly Pushdown Transducers

A **visibly pushdown transducer** (VPT) $T$ is a 7-tuple $(P, (I, f), (O, g), \Gamma, \tau, p_0, F)$, where

1. $P$ is a finite set of states.

2. • $I$ is the input alphabet partitioned into the (sub) alphabets $I_c$ and $I_r$ of input call and return symbols.
   • $f$ is a one-to-one mapping $I_c \rightarrow I_r$. We denote $f(a)$, where $a \in I_c$, by $\bar{a}$.

3. • $O$ is the output alphabet partitioned into the (sub) alphabets $O_c$ and $O_r$ of output call and return symbols respectively.
   • $g$ is a one-to-one mapping $O_c \rightarrow O_r$. We denote $g(b)$, where $b \in O_c$, by $\bar{b}$.

4. $\Gamma$ is a finite stack alphabet that (besides other symbols) contains a special “bottom-of-the-stack” symbol $\bot$.

5. $p_0$ is the initial state.

6. $F$ is the set of final states.

7. $\tau = \tau_c \cup \tau_r \cup \tau_\epsilon$, where
   • $\tau_c \subseteq P \times I_c \times O_c \times P \times \Gamma$
   • $\tau_r \subseteq P \times I_r \times O_r \times \Gamma \times P$
   • $\tau_\epsilon \subseteq P \times \{\epsilon\} \times \{\epsilon\} \times P$. 
We define an accepting run for $T$ similarly as for VPA. Now, given a word $u \in I^*$, we say that a word $w \in O^*$ is an output of $T$ for $u$ if there exists an accepting run in $T$ spelling $u$ as input and $w$ as output.\footnote{In other words, we get $u$ and $w$ when concatenating the transitions' input and output components respectively.}

A transducer $T$ might produce more than one output for a given word $u$. We denote the set of all outputs of $T$ for $u$ by $T(u)$. For a language $L \subseteq I^*$, we define the image of $L$ through $T$ as $T(L) = \bigcup_{u \in L} T(u)$.

We note that in our definition of VPTs we disallow transitions which transduce a call or return symbol to $\epsilon$. As \cite{66} showed, VPLs are closed under the transductions of such non-erasing VPTs.

\section*{4.4 Language Deletion and Insertion}

\subsection*{4.4.1 Language Deletion}

In this section we present the language operation of deletion and show that VPLs are closed under this operation.

Let $L$ and $D$ be languages on $\Sigma$. The deletion of $D$ from $L$, denoted by $L \rightarrow D$, removes from the words of $L$ one occurrence of some word in $D$. For example if $L = \{abcd, ab\}$ and $D = \{bc, cd, a\}$, then $L \rightarrow D = \{ad, ab, bcd, b\}$.

Formally, the deletion of $D$ from $L$ is defined as:

$$L \rightarrow D = \{w_1w_2 : w_1vw_2 \in L \text{ and } v \in D\}.$$
Kari (in [53]) showed that the regular languages are closed under deletion, whereas context-free languages are not. We show here that VPLs are closed under deletion.

**Theorem 2.** If $L$ and $D$ are VPLs over $\Sigma$, then $L \rightarrow D$ is a VPL as well.

**Proof.** Construction. Let $A = (Q, (\Sigma, f), \Gamma, \tau, q_0, F)$, where $Q = \{q_0, \ldots, q_{n-1}\}$, be a VPA that accepts $L$. For every two states $q_i$ and $q_j$ in $Q$ define the VPA

$$A_{ij} = (Q, (\Sigma, f), \Gamma, \tau, q_i, \{q_j\}),$$

which is the same as $A$, but with initial and final states being $q_i$ and $q_j$, respectively.

The language $L(A_{ij})$ (which we also denote by $L_{ij}$) is a VPL for each $q_i, q_j \in Q$.

Consider now the VPA $A' = (Q, (\Sigma', f), \Gamma, \tau', q_0, F)$, with

1. $\Sigma' = \Sigma \cup \{\dag\}$, where $\dag$ is a fresh local symbol,
2. $\tau'_c = \tau_c$,
3. $\tau'_r = \tau_r$, and
4. $\tau'_l = \{(q_i, \dag, q_j) : q_i, q_j \in Q \text{ and } L_{ij} \cap D \neq \emptyset\}$.

We then define language $L' = L(A') \cap (\Sigma^* \cdot \{\dag\} \cdot \Sigma^*)$. This intersection extracts from $L(A')$ all the words marked by one $\dag$. Such words are derived from the words of $L$ containing some properly nested subword which is a word in $D$.

Now we obtain VPL $L''$ by substituting $\epsilon$ for $\dag$ in $L'$. This is achieved by replacing the local $\dag$ transitions, in the VPA for $L'$, by $\epsilon$ transitions. We have that
Lemma 1. $L \rightarrow D = L''$.

Proof. \textquoteleft\text{\subseteq\textquoteright}. Let $w \in L \rightarrow D$. There exists $u \in L$, $v \in D$ such that $u = w_1vw_2$ and $w = w_1w_2$. Hence, there exists an accepting run of $A$ for $u$:

$$\rho_u = (q_0, \sigma_0), \ldots, (q_i, \sigma_i), \ldots, (q_j, \sigma_j), \ldots, (q_f, \sigma_f),$$

where $(q_0, \sigma_0), \ldots, (q_i, \sigma_i)$ is a sub-run for $w_1$, $(q_i, \sigma_i), \ldots, (q_j, \sigma_j)$ is a sub-run for $v$, $(q_j, \sigma_j), \ldots, (q_f, \sigma_f)$ is a sub-run for $w_2$, and $q_f \in F$. As $v \in D$ is a properly nested word, we have that $\sigma_i = \sigma_j$. Therefore $v \in L_{ij}$, and so, $v \in L_{ij} \cap D$. The latter implies the transition $(q_i, \dag, q_j)$ exists in $\tau'$ and we have the following run in $A'$:

$$\rho = (q_0, \sigma_0), \ldots, (q_i, \sigma_i), (q_j, \sigma_j), \ldots, (q_f, \sigma_f),$$

where $\dag$ takes $(q_i, \sigma_i)$ to $(q_j, \sigma_j)$.

Thus, $w_1 \dag w_2 \in L(A')$ and also $w_1 \dag w_2 \in L' = L(A') \cap (\Sigma^* \cdot \{\dag\} \cdot \Sigma^*)$. This proves that $w = w_1w_2 \in L''$.

\textquoteleft\text{\supseteq\textquoteright}. Let $w \in L''$. As such, there exists a word $w' \in L' = L(A') \cap (\Sigma^* \cdot \{\dag\} \cdot \Sigma^*)$ which is of the form $w_1 \dag w_2$, where $w_1, w_2 \in \Sigma^*$, and $w_1w_2 = w$. For $w'$ there exists an accepting run in automaton of $L'$:

$$\rho_{w'} = (q_0, \sigma_0), \ldots, (q_i, \sigma_i), (q_j, \sigma_j), \ldots, (q_f, \sigma_f),$$

where $(q_0, \sigma_0), \ldots, (q_i, \sigma_i)$ is a sub-run for $w_1$, $(q_j, \sigma_j), \ldots, (q_f, \sigma_f)$ is a sub-run for $w_2$, and
and \((q_i, \sigma_i), (q_j, \sigma_j)\) is a sub-run for \(\dagger\) with \(\sigma_i = \sigma_j\) (since \(\dagger\) is a local symbol). This run indicates the existence of transition \((q_i, \dagger, q_j)\) in \(A'\), which means \(L_{ij} \cap D \neq \emptyset\). This also implies that there exists some properly nested word \(v \in D\) corresponding to a run \((q_i, \sigma_i), \ldots, (q_j, \sigma_j)\) in automaton \(A\). Hence, \(w_1vw_2 \in L\) and \(w = w_1w_2 \in L \rightarrow D\).

(Lemma 1)

Finally, the claim of the theorem follows from the above lemma and the fact that the visibly pushdown languages are closed under the intersection with regular languages.

(Theorem 2)

Based on the construction of the above theorem we have that

**Theorem 3.** *Computing a VPA recognizing* \(L \rightarrow D\) *can be done in PTIME.*

*Proof.* Straightforward based on the polynomial construction steps described in the proof of Theorem 2.

**k-Bounded Deletion.** We can also define a variant of deletion which allows for up to \(k\) deletions where \(k\) is a given positive integer. This is useful when we want to evolve a given schema language \(L\) by allowing the words of a given language \(D\) to be deleted (in parallel) from \(k\)-locations of the words of \(L\) rather than just one.

Let \(L\) and \(D\) be languages on \(\Sigma\). The \(k\)-bounded deletion of \(D\) from \(L\) is defined
as:

\[
L \xrightarrow{\leq k} D = \{ w_1w_2 \ldots w_iw_{i+1} : w_1v_1w_2 \ldots w_iv_iw_{i+1} \in L, \\
v_j \in D \text{ for } 1 \leq j \leq i, \text{ and } 1 \leq i \leq k \}.
\]

Observe that \( L \xrightarrow{\leq k} D \) cannot be obtained by iterative applications of \( k \) single deletions. For example, assume \( L = \{ abc\bar{c}\bar{b}a \} \) and \( D = \{ \bar{b}\bar{b}, c\bar{c} \} \). The language resulting from 2-bounded deletion of \( D \) from \( L \) is \( \{ a\bar{b}\bar{a} \} \). On the other hand, by applying 2 successive deletions of \( D \) from \( L \) we have \( (L \rightarrow D) \rightarrow D = \{ a\bar{a} \} \).

We show now that VPLs are closed under \( k \)-bounded deletion.

**Theorem 4.** If \( L \) and \( D \) are VPLs over \( \Sigma \), then \( L \xrightarrow{\leq k} D \) is a VPL as well.

**Proof. Construction.** Let \( A = (Q, (\Sigma, f), \Gamma, \tau, q_0, F) \) be a VPA that accepts \( L \).

We build a VPA \( A' \) as explained in the proof of Theorem 2. Now we set

\[
L_k' = L(A') \cap \bigcup_{1 \leq h \leq k} (\Sigma^* \cdot \{ \dagger \} \cdot \Sigma^*)^h.
\]

This intersection extracts from \( L(A') \) all the words marked by up to \( k \dagger \)'s. Such words are derived from the words of \( L \) containing properly nested subwords which are words in \( D \).

Now we obtain VPL \( L_k'' \) by substituting symbols \( \dagger \) in \( L_k' \) by \( \epsilon \). This is achieved by replacing the local \( \dagger \) transitions, in the VPA for \( L_k' \), by \( \epsilon \) transitions. We can show that
Lemma 2. $L \xrightarrow{\leq k} D = L''_k$.

Proof. We prove this by induction.

Basic step. For $k = 1$, we have $L \xrightarrow{\leq 1} D = L \rightarrow D$. On the other hand, we have that $L_1' = L(A') \cap (\Sigma^* \cdot \{\dagger\} \cdot \Sigma^*) = L'$, where $L'$ is defined as in Theorem 2. So, $L''_1 = L''$ and using Lemma 1 we conclude that $L \xrightarrow{\leq 1} D = L''_1$.

Induction step. Assume $L \xrightarrow{\leq n-1} D = L''_{n-1}$ (hypothesis). Now for $n$, we can write

$$L'_n = \left( L(A') \cap \bigcup_{1 \leq h \leq n-1} (\Sigma^* \cdot \{\dagger\} \cdot \Sigma^*)^h \right) \cup \left( (L(A') \cap (\Sigma^* \cdot \{\dagger\} \cdot \Sigma^*)^n) \right).$$

From the hypothesis we have

$$L'_n = L'_{n-1} \cup (L(A') \cap (\Sigma^* \cdot \{\dagger\} \cdot \Sigma^*)^n).$$

The intersection $L(A') \cap (\Sigma^* \cdot \{\dagger\} \cdot \Sigma^*)^n$ extracts from $L(A')$ all the words marked by exactly $n$ †’s. Let us call this language $L'_{=n}$. Now, the union $L'_n = L'_{n-1} \cup L'_{=n}$ is the language of all the words extracted from $L(A')$ marked by at most $n$ †’s. By substituting $\epsilon$ for † in the VPA of the above union we get $L''_n = L''_{n-1} \cup L''_{=n}$. By the hypothesis, $L''_{n-1} \cup L''_{=n} = \left( L \xrightarrow{\leq n-1} D \right) \cup L''_{=n}$. Language $L''_{=n}$ is the set of all words with exactly $n$ deletions (of words in $D$). Thus, based on the definition of the $k$-bounded deletion, $L''_n = \left( L \xrightarrow{\leq n-1} D \right) \cup L''_{=n} = L \xrightarrow{\leq n} D$, and this ends the induction.

(Lemma 2)

Based on this lemma, the claim of the theorem follows.
Based on the construction of the above theorem we have that

**Theorem 5.** Computing a VPA recognizing $L \xrightarrow{\leq k} D$ can be done in PTIME if $k$ is constant.

*Proof.* The construction of VPA $A'$ can be done in polynomial time. The automaton for $\bigcup_{1 \leq h \leq k}(\Sigma^* \cdot \{\dagger\} \cdot \Sigma^*)^h$ (in the proof of Theorem 4) has $k + 1$ states and constructing the intersection of this automaton with VPA $A'$ needs polynomial time. \qed

### 4.4.2 Language Insertion

Let $L$ and $I$ be languages on $\Sigma$. The *insertion of $I$ into $L$*, denoted by $L \leftarrow I$, inserts into the words of $L$ some word in $I$. For example if $L = \{ab, cd\}$ and $I = \{eg\}$, then $L \leftarrow I = \{egab, aegb, abeg, egcd, cegd, cdeg\}$.

Formally, the *insertion of $I$ into $L$* is defined as

$$L \leftarrow I = \{w_1vw_2 : w_1w_2 \in L, \text{ and } v \in I\}.$$

Kari (in [53]) showed that the regular languages and context free languages are closed under insertion. We show here that VPLs are closed under insertion, too.

**Theorem 6.** If $L$ and $I$ are VPLs over $\Sigma$, then $L \leftarrow I$ is a VPL as well.

*Proof. Construction.* Let $A = (Q, (\Sigma, f), \Gamma, \tau, q_0, F)$ and $A^I = (P, (\Sigma, f), \ldots$
\( \Gamma, \tau^I, p_0, F^I \) be the VPA's accepting \( L \) and \( I \), respectively. We construct VPA 
\( A' = (Q, (\Sigma', f), \Gamma, \tau', q_0, F^I) \), from \( A \) with

1. \( \Sigma' = \Sigma \cup \{\#\} \), where \( \# \) is a fresh local symbol,
2. \( \tau'_c = \tau_c \),
3. \( \tau'_r = \tau_r \), and
4. \( \tau'_t = \{(q_i, \#, q_i) : q_i \in Q\} \).

Then, we define language \( L' = L(A') \cap (\Sigma^* \cdot \{\#\} \cdot \Sigma^*) \). This intersection extracts from \( L(A') \) all the words marked by only one \( \# \). Let \( (q_i, \#, q_i) \) be the transition reading \( \# \). Next, we construct \( L'' \) by replacing the (local) transition \( (q_i, \#, q_i) \) by a fresh copy of \( A^I \) connected to state \( q_i \) by the following \( \epsilon \)-transitions,

\[ \{(q_i, \epsilon, p_0)\} \cup \{(p_f, \epsilon, q_i) : p_f \in F^I\}, \]

We show that

**Lemma 3.** \( L \leftarrow I = L'' \).

*Proof.* Let \( A'' \) be the VPA of \( L'' \).

"\( \subseteq \)." Let \( w \in L \leftarrow I \). There exist \( v \in I \) and \( x = w_1 w_2 \in L \) such that \( w = w_1 v w_2 \).

From the construction of \( L' \), there exists \( u \in L' \) such that \( u = w_1 \# w_2 \). Hence, there exists an accepting run:

\[ \rho_u = (q_0, \sigma_0), \ldots, (q_i, \sigma_i), (q_i, \sigma_i), \ldots, (q_f, \sigma_f), \]
for \( u \) in the automaton accepting \( L' \), where \((q_0, \sigma_0), \ldots, (q_i, \sigma_i)\) is a sub-run for \( w_1 \), \((q_i, \sigma_i), (q, \sigma_i)\) reads (local) symbol \#; \((q_i, \sigma_i), \ldots, (q_f, \sigma_f)\) is a sub-run for \( w_2 \), and \( q_f \in F \).

Based on the construction of \( L'' \), we have some accepting run of \( A'' \) of the form:

\[
(q_0, \sigma_0), \ldots, (q_i, \sigma_i), (p_0, \sigma_i), \ldots, (p_f, \sigma_i), (q_i, \sigma_i), \ldots, (q_f, \sigma_f),
\]

where \( p_f \in F^I \) and \((p_0, \sigma_i), \ldots, (p_f, \sigma_i)\) is a sub-run for some properly nested word \( v \in I \). Thus, since \((q_0, \sigma_0), \ldots, (q_i, \sigma_i)\) and \((q_i, \sigma_i), \ldots, (q_f, \sigma_f)\) spell \( w_1 \) and \( w_2 \) respectively, the above run is an accepting run for \( w_1vw_2 \), i.e. \( w_1vw_2 = w \in L'' \).

\( \supseteq \). Let \( w \in L'' \). By the construction of \( L'' \) (i.e. \( A'' \)), there exists an accepting run for \( w \) in \( A'' \):

\[
(q_0, \sigma_0), \ldots, (q_i, \sigma_i), (p_0, \sigma_i), \ldots, (p_f, \sigma_i), (q_i, \sigma_j), \ldots, (q_f, \sigma_f),
\]

where \( q_f \in F \) and \( p_f \in F^I \). The existence of sub-run \((p_0, \sigma_i), \ldots, (p_f, \sigma_i)\) implies that \( w \) contains a subword \( v \) which belongs to \( I \). Hence, \( w = w_1vw_2 \), where \( w_1 \) and \( w_2 \) correspond to sub-runs \((q_0, \sigma_0), \ldots, (q_i, \sigma_i)\) and \((q_i, \sigma_i), \ldots, (q_f, \sigma_f)\), respectively. Since, by the construction of \( A'' \), these two sub-runs are sub-runs of \( A \), we have \( w_1w_2 \in L \), i.e. \( w = w_1vw_2 \in L \leftarrow I \).

(Lemma 3)

Finally, the claim of the theorem follows from the above lemma and the fact that
visibly pushdown languages are closed under the intersection with regular languages.

(Theorem 6)

Based on the construction of Theorems 6, we have that

**Theorem 7.** Computing a VPA recognizing $L \leftarrow I$ can be done in PTIME.

**Proof.** The construction of $A'$ can be done in linear time in the number of states in $Q$. The automaton accepting $L'$ can be built in polynomial time by constructing the Cartesian product of $A'$ with the two state automaton accepting $\Sigma^* \cdot \{\#\} \cdot \Sigma^*$. The final stage, which is replacing $\#$ by a copy of $A'$, needs polynomial time. □

**k-Bounded Insertion.** We can also define a variant of insertion which allows for up to $k$ insertions where $k$ is a given positive integer. This is useful when we want to evolve a given schema language $L$ by allowing the words of a given language $I$ to be inserted (in parallel) into $k$-locations in the words of $L$ rather than just one.

Let $L$ and $I$ be languages on $\Sigma$. The *$k$-bounded insertion of $I$ into $L$* is defined as:

$$L \triangleleft^k I = \{ w_1v_1w_2\ldots w_iv_iw_{i+1} : w_1w_2\ldots w_{i+1} \in L, \quad v_j \in I \text{ for } 1 \leq j \leq i, \text{ and } 1 \leq i \leq k \}.$$  

We show here that VPLs are closed under the operation of $k$-bounded insertion.

**Theorem 8.** If $L$ and $I$ are VPLs over $\Sigma$, then $L \triangleleft^k I$ is a VPL as well.

**Proof. Construction.** Let $A = (Q, (\Sigma, f), \Gamma, \tau, q_0, F)$ be a VPA that accepts $L$.  

We build a VPA $A'$ as explained in the proof of Theorem 6. Then we define language

$$L'_k = L(A') \cap \bigcup_{1 \leq h \leq k} (\Sigma^* \cdot \{\#\} \cdot \Sigma^*)^h.$$  

The above intersection extracts from $L(A')$ all the words marked by one up to $k$ #’s.

Next, we construct $L''_k$ by replacing the (local) transitions reading #’s by fresh copies of $A'$ as explained in Theorem 6.

We show that

**Lemma 4.** $L \leq^k I = L''_k$.

**Proof.** This can be proved using induction similar to the proof of Lemma 2.

(Lemma 4) $\square$

Based on this lemma the claim of the theorem follows.

(Theorem 8) $\square$

Based on Theorem 8, we have that

**Theorem 9.** Computing a VPA recognizing $L \leq^k I$ can be done in PTIME if $k$ is constant.

**Proof.** Similar to the proof of Theorem 7, construction of VPA $A'$ can be done in linear time. If $k$ is constant, the automaton for $\bigcup_{1 \leq h \leq k} (\Sigma^* \cdot \{\#\} \cdot \Sigma^*)^h$ has $k + 1$ states and the automaton accepting $L'_k$ can be built in polynomial time by constructing the Cartesian product of this automaton and VPA $A'$. The final stage, which is replacing #’s by copies of $A'$, needs polynomial time. $\square$
4.4.3 Transforming a VPL with Language Operations

In practice it is more useful to allow the schema transformation to be achieved by a set of deletion and insertion operations. For example, we can define a set \( \mathcal{D} = \{D_1, \ldots, D_m\} \) and \( \mathcal{I} = \{I_1, \ldots, I_n\} \) of allowed language deletions and insertions, respectively. With slight abuse of notation we will consider \( D_1, \ldots, D_m \) and \( I_1, \ldots, I_n \) to also denote their corresponding delete and insert operations, respectively. What we would like now is to apply (in parallel) up to \( k \) operations from \( \mathcal{D} \cup \mathcal{I} \) on a given schema language \( L \).

For this, given a VPA \( A \) for \( L \), we extend the constructions described in the constructions of theorems 2 and 6. Specifically, let VPA \( A = (Q, (\Sigma, f), \Gamma, \tau, q_0, F) \) have the states numbered as \( \{q_0, \ldots, q_{n-1}\} \). Also, for every two states \( q_i \) and \( q_j \) in \( Q \) define VPA \( A_{ij} \) and its accepted language \( L_{ij} \) as described earlier (e.g. Theorem 2). We construct now the VPA \( A' = (Q, (\Sigma', f), \Gamma, \tau', q_0, F) \), with

1. \( \Sigma' = \Sigma_c \cup \Sigma_r \cup \{\dagger_1, \ldots, \dagger_m\} \cup \{\#_1, \ldots, \#_n\} \)

2. \( \tau'_c = \tau_c \),

3. \( \tau'_r = \tau_r \), and

4. \( \tau'_l = \{(q_i, \dagger_x, q_j) : q_i, q_j \in Q \text{ and } L_{ij} \cap D_x \neq \emptyset, \text{ for } 1 \leq x \leq m\} \cup \
\{(q_i, \#_y, q_i) : q_i \in Q \text{ and } 1 \leq y \leq n\} \).

VPA \( A' \) will accept language \( L' \) containing words with an arbitrary number of special local symbols. Each \( \dagger_x \) represents a deletion corresponding to \( D_x \), and each
#_y represents an insertion corresponding to I_y. What we want, though, is to extract only those words of L', whose total number of the special symbols is not more than k. For this we construct the following intersection

\[ L'' = L' \cap \bigcup_{1 \leq h \leq k} (\Sigma^* \cdot \{\dagger_1, \ldots, \dagger_m, \#_1, \ldots, \#_n\} \cdot \Sigma^*)^h. \]

Language L'' will contain all the words of L' with not more than k special symbols.

Then, we obtain language L''' by replacing

1. all the \( \dagger_x \), for 1 \( \leq x \leq m \), by \( \epsilon \), and

2. all the \( \#_y \), for 1 \( \leq y \leq n \), by the corresponding VPLs I_y (in the natural way described in Section 4.4.2).

It can be verified that

**Theorem 10.** L''' is the result of applying from one up to k operations from \( D \cup I \) on L.

Taking the union \( L \cup L''' \) gives us the new expanded (evolved) schema language.

Based on the above construction, we have that

**Theorem 11.** Computing a VPA recognizing L''' can be done in PTIME if k is constant.

**Proof.** The proof of the theorem follows from the proofs of Theorem 5 and Theorem 9. \( \square \)
4.5 Constrained Deletions and Insertions

Often we do not like the deletions and insertions to be performed in unrestricted places in the words of schemas for XML. Rather we would like them to apply only at certain parts of the words. For instance, taking the example given in the Introduction, one might want to apply operations only within a soup XML element.

For this, we assume that there is a given set of constraining rules that specify the conditions under which the operations can be applied on $L$.

We illustrate the conditions using XPath expressions\(^5\). The alphabet of the XPath expressions is the set of XML elements corresponding to $\Sigma_c$ (or $\Sigma_r$). Formally this alphabet is $\Sigma_e = \{\tilde{a} : a \in \Sigma_e\}$. As the validation problem considers the structure of XML only, we do not have data values in the XPath expressions.

**Definition 1.** A deletion rule is a tuple $(\pi, D)$, where $\pi$ is an XPath expression, and $D$ is a VPL.

Such a rule implies that the words of nested language $D$ can be deleted if they correspond to elements reached by XPath expression $\pi$. Of course, such elements need to further satisfy the structure imposed by $D$. An example of a deletion rule is $(/\tilde{a}/\tilde{b}/\tilde{c}, PN)$. Using this rule we can delete all the $\tilde{c}$ elements that are children of $\tilde{b}$ elements which in turn are children of $\tilde{a}$ elements. Surely, specifying $D = PN$ is a useful case in practice. However, we can further qualify the $D$ language to allow only the deletion of those $\tilde{c}$ elements which contain some particular child, say an element $\tilde{d}$.

\(^5\)We consider in fact unary CoreXPath expressions.
\( \tilde{d} \). In such a case, we set \( D = \{c\} \cdot PN \cdot \{d\} \cdot PN \cdot \{\tilde{d}\} \cdot PN \cdot \{\tilde{c}\} \).

We denote by \( \mathcal{D} \) the set of deletion rules.

**Definition 2.** An insertion rule is a tuple \((\pi, I)\), where \( \pi \) is an XPath expression, and \( I \) is a VPL.

Such a rule implies that the words of nested language \( I \) can only be inserted as children of elements reached by XPath expression \( \pi \). An example of an insertion rule is \((/\tilde{a}/\tilde{b}, PN)\).

We denote by \( \mathcal{I} \) the set of insertion rules.

As shown in [75, 76], a unary (Core) XPath query can be represented by a VPA with a single output variable attached to some call transitions. The set of query answers consists of all the elements whose call (open) symbol binds to the variable during an accepting run. We are aware that VPA representations of XPath queries can be exponential in the worst case (see [77]), but for the practical class of Downward XPath using a fixed number of test expressions the resulting VPA is of size polynomial in the size of the input XPath query (see [78]). Furthermore, the constraints can also be given directly as VPLs using approaches as in the example of [60], i.e. not specified by XPath (although an XPath specification is more user friendly). Detailing the practical classes for which the XPath to VPA translation is polynomial is outside the scope of this chapter.

Let \( \pi \) be an XPath expression and \( A_\pi \) a VPA for it. Let \( \tau'_c \) be the subset of the \( A_\pi \) transitions with the output variable attached to them. It is easy to identify with the
help of the stack the corresponding return transitions, $\tau'_r$. Since we are not interested
in outputting the result of the XPath queries, but rather just locate the elements of
interest, we will assume that instead of having a variable attached to the transitions
in $\tau'_c$, we have them simply marked by a dot (\'. Also, we will similarly assume the
transitions in $\tau'_r$ are marked as well.

We consider the alphabet of this VPA to be $(\Sigma_c \cup \hat{\Sigma}_c) \cup (\Sigma_r \cup \hat{\Sigma}_r)$, where $\hat{\Sigma}_c$ and
$\hat{\Sigma}_r$ are copies of $\Sigma_c$ and $\Sigma_r$, respectively, with the symbols being marked by (\').

When a rule is applied on a given word (of the schema VPL), we do not initially
perform deletion or insertion on the word, but only color the related symbols. This is
due to the fact that other rules can be further applied, and the XPath expressions for
those rules were written considering the original version of the given XML schema.

For example, suppose we have a set of deletion rules $\mathcal{D} = \{(\bar{\tilde{a}}/\tilde{b}, P N), (/\tilde{a},$
$\{a\tilde{b}\tilde{a}\})\}$, and a word $w = a\tilde{b}\tilde{a}c\tilde{c}$. Both of these rules can be applied on $w$. But if we
apply the first rule and truly delete $b\tilde{b}$, then the second rule can no longer be applied
on the result word, which is $w' = a\tilde{c}\tilde{c}$. On the other hand, only coloring $b$ and $\tilde{b}$,
does not prevent the second rule from being applied on the word.

In the following we construct coloring VPTs based on the VPAs for the given
rules. Then, we apply these VPTs on a schema language $L$.

4.5.1 Coloring VPTs for Deletion Rules

Let $(\pi_x, D_x)$ be a deletion rule in $\mathcal{D}$. For each such rule we choose a distinct red color,$r_x$. In the following construction, we use alphabets $\Sigma_c$, $\Sigma_r$, $\Sigma'_c$, $\Sigma'_r$, where the last
two are copies of the first two, respectively, having the symbols colored in \( r_x \). For the sake of the discussion, we will consider the symbols of \( \Sigma_c, \Sigma_r \) as being colored in black.

Using \( A_{\pi_x} \) we construct an \( r_x \)-coloring VPT \( T_{\pi_x}^{r_x} \), which has exactly the same states as \( A_{\pi_x} \), and transitions:

1. \((q, a, a, p, \gamma_a)\) for \((q, a, p, \gamma_a)\) in \( A_{\pi_x} \),
2. \((q, \bar{a}, \bar{a}, \gamma_a, p)\) for \((q, \bar{a}, \gamma_a, p)\) in \( A_{\pi_x} \),
3. \((q, a, a^{r_x}, p, \gamma_a^{r_x})\) for \((q, \dot{a}, p, \gamma_a)\) in \( A_{\pi_x} \),
4. \((q, \bar{a}, \bar{a}^{r_x}, \gamma_a^{r_x}, p)\) for \((q, \dot{\bar{a}}, \gamma_\bar{a}, p)\) in \( A_{\pi_x} \).

Intuitively, the “un-marked” transitions of \( A_{\pi_x} \) become “leave-unchanged” transitions in \( T_{\pi_x}^{r_x} \), whereas the “marked” transitions of \( A_{\pi_x} \) become “black to red” transitions in \( T_{\pi_x}^{r_x} \).

### 4.5.2 Coloring VPTs for Insertion Rules

Let \((\pi_y, I_y)\) be an insertion rule in \( \mathcal{I} \). For each such rule we choose a distinct green color, \( g_y \). In the following construction, we use alphabets \( \Sigma_c, \Sigma_r, \Sigma_c^{g_y}, \Sigma_r^{g_y} \), where the last two are copies of the first two, respectively, having the symbols colored in \( g_y \).

Using \( A_{\pi_y} \) we construct a \( g_y \)-coloring VPT \( T_{\pi_y}^{g_y} \), which has exactly the same states as \( A_{\pi_y} \), and transitions:

1. \((q, a, a, p, \gamma_a)\) for \((q, a, p, \gamma_a)\) in \( A_{\pi_y} \),
2. \((q, \bar{a}, \bar{a}, \gamma_a, p)\) for \((q, \bar{a}, \gamma_a, p)\) in \(A_{\pi_y}\),

3. \((q, a, a^{g_y}, p, \gamma_a^{g_y})\) for \((q, \bar{a}, p, \gamma_a)\) in \(A_{\pi_y}\),

4. \((q, \bar{a}, \bar{a}^{g_y}, \gamma_a^{g_y}, p)\) for \((q, \bar{a}, \gamma_a, p)\) in \(A_{\pi_y}\).

Intuitively, the “un-marked” transitions of \(A_{\pi_y}\) become “leave-unchanged” transitions in \(T_{\pi_y}^{g_y}\), whereas the “marked” transitions of \(A_{\pi_x}\) become “black to green” transitions in \(T_{\pi_y}^{g_y}\).

### 4.5.3 Color-Tolerant VPTs

Coloring VPTs presented in the two previous subsections can be applied only on black (normal) words (of a schema VPL). When we want to apply a coloring VPT on a word more than once or when we apply coloring VPTs for deletions and insertions one after the other, the VPTs have to be applicable also to words which have parts already colored. For example, suppose word \(w = abb\bar{a}c\bar{d}\bar{d}c\) has the \(\bar{b}\bar{b}\) part already colored in a red (being so ready for deletion). Word \(w\) might be needed next to have \(d\bar{d}\) colored in a green color (to become ready for an insertion). In order for a coloring VPT for insertion to be able to color \(d\bar{d}\) in green, it has to be “color-tolerant” while reading the prefix \(abb\bar{a}c\) of \(w\).

Let \(T_{\pi_x}\) be a coloring VPT for a deletion as described in Subsection 4.5.1. Now we make \(T_{\pi_x}\) color-tolerant by adding the following colored copies of its transitions.

1. \((q, a^{r_x}, a^{r_x}, p, \gamma_a^{r_x})\) and \((q, a^{g_y}, a^{g_y}, p, \gamma_a^{g_y})\), for each transition \((q, a, a, p, \gamma_a)\), and for every color \(r_x\) and \(g_y\).
2. \((q, \bar{a}^{r_x}, a^{r_x}, \gamma^{r_x}_a, p)\) and \((q, \bar{a}^{g_y}, a^{g_y}, \gamma^{g_y}_a, p)\), for each transition \((q, \bar{a}, a, \gamma_a, p)\), and for every color \(r_x\) and \(g_y\).

3. \((q, a^{r_x}, a^{r_x}, p, \gamma^{r_x}_a)\) and \((q, a^{g_y}, a^{r_x}, p, \gamma^{r_x}_a)\), for each transition \((q, a, a^{r_x}, p, \gamma^{r_x}_a)\), and for every color \(r_x \neq r_z\) and \(g_y\).

4. \((q, \bar{a}^{r_x}, a^{r_x}, \gamma^{r_x}_a, p)\) and \((q, \bar{a}^{g_y}, a^{r_x}, \gamma^{r_x}_a, p)\), for each transition \((q, \bar{a}, a^{r_x}, \gamma^{r_x}_a, p)\), and for every color \(r_x \neq r_z\) and \(g_y\).

Finally, we mention that color-tolerant VPTs for insertions can be constructed in a similar way. Specifically, wherever there is an \(r_z\) superscript there will be a \(g_z\) one.

### 4.5.4 Transforming a VPL with Constrained Operations

Let \(D = \{(\pi_1, D_1), \ldots, (\pi_m, D_m)\}\) and \(I = \{(\pi_1', I_1), \ldots, (\pi_n', I_n)\}\) be the sets of rules for the allowed deletions and insertions, respectively. What we want is to apply up to \(k\) operations corresponding to the rules in \(D \cup I\) on a given schema language \(L\).

We start by constructing coloring VPTs for each of the rules in \(D \cup I\), as described in Subsections 4.5.1, 4.5.2, and 4.5.3. Next, we transduce \(L\) by iteratively applying these VPTs one after the other (in no particular order) \(k\) times each. The result of this multiple transduction will be a language that is the same as \(L\) but with the words being colored to indicate the allowed places for deletions and insertions. For simplicity let us continue to use \(L\) for this colored version of the schema language.

Let \(A\) be a VPA for (the colored) \(L\). Let \(A_{ij}\), and its accepted language \(L_{ij}\), be defined as in theorems 2 and 6. Also, let \(\beta\) be a transformation that uncolors words
and languages. This transformation can be easily realized by a VPT. Now from $A$ we build VPA $A'$ keeping the same states and transitions, but adding the following transitions labeled by special local symbols.

$$\{(q_i, \dagger_x, q_j) : \beta((L_{ij} \cap (\Sigma^r_c \cdot PN \cdot \Sigma^r_r)) \cap D_x \neq \emptyset, \text{ for } 1 \leq x \leq m) \cup$$

$$\{(q_j, \#_y, q_j) : \text{there exists a transition } (\ldots, g^y_i, q_i, \ldots) \text{ and } L_{ij} \neq \emptyset\}.$$

The first set is for $\dagger$ transitions between the pairs of states connected by properly nested words in $D_x$. The words of interest in $L_{ij}$ are those with a first and last symbol colored in red ($r_x$). We determine these words by intersecting with $(\Sigma^r_c \cdot PN \cdot \Sigma^r_r)$. Then, we apply the $\beta$ transformation in order to uncolor the resulting language and proceed with intersecting with $D_x$.

The second set indicates that if there is a call transition $(\ldots, g^y_i, q_i, \ldots)$ colored by $g_y$ (due to an insertion rule $(\pi'_y, I_y)$) in $A$, then in $A'$ we have self-loop transitions labeled by $\#_y$ which are added to all states $q_j$ reachable from $q_i$ such that $A_{ij}$ accepts a properly nested language including the empty word.

VPA $A'$ will accept language $L'$ containing words with at most $k$ special symbols of each kind ($\dagger_1, \ldots, \dagger_m, \#_1, \ldots, \#_n$). Each $\dagger_x$ represents a deletion corresponding to $D_x$, and each $\#_y$ represents an insertion corresponding to $I_y$. What we want, though, is to extract only those words of $L'$, whose total number of the special symbols is not
more than $k$. For this we construct the following intersection

$$L'' = L' \cap \bigcup_{1 \leq h \leq k} \left( \Sigma^* \cdot \{\hat{1}_1, \ldots, \hat{1}_m, \#_1, \ldots, \#_n \} \cdot \Sigma^* \right)^h.$$ 

Language $L''$ will contain all the words of $L'$ with not more than $k$ special symbols.

Then, we obtain language $L'''$ by replacing

1. all the $\hat{1}_x$, for $1 \leq x \leq m$, by $\epsilon$, and

2. all the $\#_y$, for $1 \leq y \leq n$, by the corresponding VPLs $I_y$ (in the natural way described in Section 4.4.2).

Based on all the above we have that

**Theorem 12.** $L'''$ is the result of applying from one up to $k$ constrained operations from $D \cup I$ on $L$.

Taking the union $L \cup L'''$ gives us the new expanded (evolved) schema language.

**Complexity**

Regarding the complexity we have the following theorems.

**Theorem 13.** The colored VPA recognizing $L$ can be computed in $O(\delta^{(m+n)k})$ time, where $\delta$ is an upper bound on the size of rule automata.

*Proof.* The claim follows from the fact that each coloring VPT is applied $k$ times, and we have $n + m$ such VPTs. \hfill \Box

We note that in practice, the numbers $k$, $m$, and $n$ would typically be small.
**Theorem 14.** The automaton recognizing $L''$ can be constructed from the colored VPA $A$ accepting $L$ in polynomial time if $k$ is constant.

**Proof.** Let us refer to the steps for constructing $L''$. When constructing $A'$, we need to check the emptiness of $\beta(L_{ij} \cap (\Sigma^r_{c} \cdot PN \cdot \Sigma^c_{r})) \cap D_x$. For this we build Cartesian products of $A$ and a VPA for $(\Sigma^r_{c} \cdot PN \cdot \Sigma^r_{c})$ and then a VPA for $D_x$. This is done separately for $m$ different deletion rules requiring polynomial time in the size of the VPAs and $m$.

When adding the second set of transitions to $A'$, observe that the maximum number of $g_y$ colored call transitions [of the form of $(\_, s^y, q_i, \_)$] is $|Q|^2$, where $Q$ is the set of states in $A$. Therefore, this substep also requires only polynomial time.

Finally, constructing $L''$ and $L'''$ from $L'$ (represented by $A'$) is clearly polynomial.

\[\square\]

### 4.6 Conclusions

In this chapter we proposed modeling the schema evolution for XML by using the language operations of deletion and insertion on VPLs. We showed that the VPLs are well-behaved under these operations and presented constructions for computing the result of the operations. Then, we introduced constrained operations which are arguably more useful in practice. In order to compute the results of constrained operations we developed special techniques (such as VPA coloring) achieving a compatible application of a set of different operations. Based on our techniques, the
schema evolution operators can be applied in parallel without side-effect interactions.

Another variant of language operations is “unbounded” operations, which can be defined as $k$-bounded operations when $k = \infty$. It has been shown in [53] that regular languages are closed under unbounded (or “iterative sequential”) deletion but not insertion. Unbounded language operations for VPLs have not been explored in this thesis and we will consider it as a future work.
Chapter 5

Conclusions

In this dissertation we have addressed three important problems related to graph structured data (GSD), which has recently achieved an increasing popularity on account of its flexible nature.

First, we have focused on the problem of evaluating regular path queries (RPQs) on distributed GSD. We have presented a distributed algorithm which has optimal message complexity. Our proposed algorithm is truly distributed, evaluates the queries progressively, and uses the highest possible level of parallelism. We have also presented resilient variant of our algorithm that can handle any number of process failures in a transparent way. That is, query evaluation is not interrupted in face of process failures, but instead it preserves the computation done so far and outputs at least the best answer were the evaluation to restart using the remaining live processes.

Second, we have addressed the challenge of answering RPQs on incomplete GSD, where only some view of the original database is available. At first, we have shown
that the data complexity of computing certain answers is polynomial for local RPQS and this is due to the fact that for local RPQs the answers obtained from view-based rewriting, which can be computed in polynomial time, coincide with the certain answers. Then, we have presented an algorithm to compute maximal view-based rewriting for LRPQs. The rewriting is in the form of a multientry DFA and its expression complexity is an exponential order smaller than its counterpart for global RPQs.

The third problem we have considered is flexible validation of streaming XML in a realistic scenario, where the schema of data exchanging parties can evolve over time. We have utilized visibly pushdown automata (VPAs) to formalize XML schema, and then schema evolution have been modeled by applying formal language operations on the schema language. After showing the closure of VPLs under language operations of insertion and deletion, we have introduced $k$-bounded, as well as, constrained language operations on VPLs. Then, we have devised techniques to represent XML schema evolution using these operations.
Bibliography


http://www.census.gov/geo/www/tiger.


[38] “Planet-lab.” www.planet-lab.org.


