The Complexity of Relational Queries over Extractions from Text

Liat Peterfreund
The Complexity of Relational Queries over Extractions from Text

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Liat Peterfreund

Submitted to the Senate
of the Technion — Israel Institute of Technology
Av 5779 Haifa August 2019
This research was carried out under the supervision of Prof. Benny Kimelfeld, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Title</th>
<th>Conference/Book</th>
</tr>
</thead>
</table>
ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and gratitude to my advisor, Benny Kimelfeld, for his guidance and support. Working with Benny has taught me not only how to conduct profound and deep research but also how to see it within the broader context. I feel fortunate for having Benny as my guide in this fascinating journey.

I had the pleasure to work with my co-authors from which I have learned a great deal: Balder ten Cate, Ronald Fagin, Dominik D. Freydenberger, Daniel Genkin, Michael Kaminski, Markus Kröll, Yoav Nahshon, and Stijn Vansummeren.

I thank my candidacy exam committee, Oded Shmueli, Yehoshua Sagiv, Yael Amsterdamer and Eldar Fischer, for their questions and ideas. I also thank my final exam committee, Antoine Amarilli and Eldar Fischer, for their insightful comments.

I spent the summer of 2017 at IBM Almaden Research Labs and I am indebted to the Theory Group for hosting me and especially to my mentor there, Ronald Fagin, for a fun and fruitful collaboration.

The Computer Science Department was my second home throughout the last decade and I am thankful to its administrative and academic staff for that. I thank my wonderful friends that accompanied me throughout my undergraduate and graduate studies. I also thank my Master’s advisor, Michael Kaminski, for his advice along the way.

I am thankful to my family and in particular my parents and my parents-in-law for their help and encouragement. I am especially grateful to my kids Yonatan and Yuval for expanding my heart and making me happy.

Last but not least, I thank my husband, Ari. I could not have done this without you. This work is dedicated to you.

The Technion’s funding of this research is hereby acknowledged.
## Contents

1 Introduction 5

2 Preliminaries 15
   2.1 Document Spanners ........................................... 15
   2.2 Spanner Representations ..................................... 16
      2.2.1 Ref-Words ............................................. 16
      2.2.2 Regex Formulas ....................................... 17
      2.2.3 Variable-Set Automata ................................. 18
   2.3 Algebraic Operators for Spanners ......................... 19
   2.4 Regular and Core Spanners .................................. 20

3 Expressiveness and Descriptive Complexity 21
   3.1 RGXlog: Datalog over Regex Formulas .................... 21
   3.2 Comparison to Core Spanners ............................... 23
   3.3 Proof of Theorem 3.2.3 ................................... 26
      3.3.1 The “Only If” Direction .............................. 26
      3.3.2 The “If” Direction .................................. 28
   3.4 Equivalence to Polynomial Time ............................ 30
      3.4.1 Proof of Theorem 3.4.1 ............................... 30
      3.4.2 RGXlog over Monadic Regex Formulas ................. 33
   3.5 Concluding Remarks ......................................... 35

4 Combined Complexity 37
   4.1 Formal Setup ................................................. 37
      4.1.1 (Unions of) Conjunctive Queries ...................... 38
      4.1.2 Complexity Measures .................................. 40
   4.2 Lower Bounds of UCQ Evaluation ......................... 41
   4.3 Evaluating Vset-Automata .................................. 44
      4.3.1 Sequences of Variable Configurations ............... 44
      4.3.2 The Algorithm ....................................... 46
   4.4 Upper Bounds of UCQ Evaluation ......................... 53
   4.5 String Equality Selection ................................. 61
      4.5.1 Formal Setup ......................................... 61
      4.5.2 Lower Bound ......................................... 62
      4.5.3 Upper Bounds ........................................ 63
   4.6 The Difference Operator .................................. 66
   4.7 Concluding Remarks ....................................... 68
5 Complexity of Schemaless Spanners

5.1 Schemaless and Schema-based Spanners ............................................. 69
  5.1.1 Regex Formulas ................................................................. 70
  5.1.2 Vset-Automata ................................................................. 72
  5.1.3 Schemaless Spanner Algebra .................................................. 73
  5.1.4 Complexity ................................................................. 74

5.2 The Natural-Join Operator ................................................................. 75
  5.2.1 Bounded Number of Shared Variables ....................................... 77
  5.2.2 Restricting to Disjunctive Functional ....................................... 80

5.3 The Difference Operator ................................................................. 83
  5.3.1 Bounded Number of Common Variables ..................................... 83
  5.3.2 Proof of Lemma 5.3.1 ......................................................... 86
  5.3.3 Restricting the Disjunctions .................................................. 88
  5.3.4 Proof of Theorem 5.3.9 ....................................................... 92

5.4 Extraction Complexity of Schemaless Spanners .................................... 98

5.5 Concluding Remarks .................................................................... 101

6 Conclusions .................................................................... 103

A Appendix for Chapter 3

A.1 Extension to a Combined Relational/Textual Model ............................ 115
  A.1.1 Spannerlog ........................................................................... 115
  A.1.2 Equivalence to Polynomial Time ............................................. 117

A.2 Proof of Theorem A.1.1 ................................................................. 119
Abstract

Large amounts of textual data with high potential value are prevalent nowadays. Incorporating textual data within traditional database systems usually consists of two stages (a) extracting relations from the text using primitive extractors and (b) manipulating the extracted relations with relational algebra operations. Document spanners is a formal framework proposed by Fagin, Kimelfeld, Reiss, and Vansummeren, that captures the relational philosophy of commercial database systems for text analytics.

A document spanner (or spanner, for short) is a function that maps a string (document) into a relation over spans which are interval positions of the string. Evaluating spanners is a computational problem that involves both the atomic extractors, the relational algebra expression, and the document. We investigate the complexity of this problem from various angles, each regarding some components as fixed and regarding the rest as input. We focus on regular atomic extractor specified by means of either regular expressions or finite state automata. We present lower bounds and devise different restrictions and approaches to establish efficient upper bounds. We study the expressive power of spanners that involve recursion and the implications of extensions that support partial extractions in the flavor of SPARQL.
Chapter 1

Introduction

The abundance and availability of valuable textual resources position text analytics as a standard component in data-driven workflows. To facilitate the integration with textual content, a core operation in these workflows is that of Information Extraction (IE)—the extraction of structured data from text. IE arises in a large variety of domains, including social media analysis \[12\], healthcare analysis \[81\], customer relationship management \[4\], information retrieval \[86\], machine log analysis \[36\], and open-domain Knowledge Base construction \[44, 72, 75, 85\].

Rules for IE are used in commercial systems and academic prototypes for text analytics, either as a standalone extraction language or within machine-learning models. A common paradigm for rule programming is the one supported by IBM’s SystemT \[17, 53\], which exposes a collection of atomic extractors of relations from text (e.g., tokenizer, dictionary lookup, part-of-speech tagger and regular-expression matcher), together with a relational algebra for manipulating these relations. In Xlog \[71\], user-defined functions form the atomic extractors, and Datalog is used for relational manipulation. In DeepDive \[68\], rules are used for generating features that are translated into the factors of a statistical model with machine-learned parameters. Feature declaration combines atomic extractors alongside relational operators applied on the resulting extractions.

Document Spanners

In their efforts towards defining a formal model that is robust enough to capture the core of the aforementioned systems, and yet, abstract enough to yield useful insights, Fagin et al. \[27\] have proposed the framework of document spanners. In this framework, a document \(d\) is a string over a fixed finite alphabet, and a spanner is a function that extracts from a document a relation over the spans of \(d\): A span \(x\) of \(d\) represents a substring \(d_x\) of \(d\) that is identified by the start and end indices. A natural way to specify a spanner is by a regex formula: a regular expression with embedded capture variables that are viewed as relational attributes. The evaluation of a spanner represented as a regex formula on a document is obtained by matching the regex formula onto the document. Each possible matching corresponds with an assignment of spans to the variables of the regex formula.
The most studied language for specifying spanners is that of regular spanners: the closure of regex formulas under the classic relational algebra: projection, natural join, union, and difference \[27\]. An equally expressive formalism is variable-set automata (vset-automata for short), which are nondeterministic finite-state automata (NFA) that can open and close variables while running. Regular spanners were shown to be closed under difference \[27\] and hence can also be described as the closure of regex formulas under positive relational algebra: projection, natural join and union. Core spanners were defined by Fagin et al. \[27\] as the core language for AQL, IBM’s SystemT SQL-like declarative language, and are obtained by extending the positive relational algebra with string-equality selection on span variables. A syntactically different language for spanners that was shown by Freydenberger \[32\] to have precisely the same expressiveness of core spanners is SpLog which is based on the existential theory of concatenation. Core spanners can express more than regular spanners. A simple example is a spanner that extracts from the input \(d\) all spans \(x\) and \(y\) such that the substring \(d_x\) spanned by \(x\) is equal to the substring \(d_y\) spanned by \(y\). However, the class of core spanners does not behave as well as that of the regular spanners; for instance, core spanners are not closed under difference. Fagin et al. \[27\] proved this by showing that no core spanner extracts all spans \(x\) and \(y\) such that \(d_x\) is not a substring of \(d_y\). The proof is based on the core simplification lemma: every core spanner can be represented as a regular spanner followed by a sequence of string equalities and projections. The same technique has been used for showing that no core spanner extracts all pairs \(x\) and \(y\) of spans having the same length \[27\].

In this thesis, we explore the computational complexity of spanner evaluation with a special focus on the regular spanner representations (i.e., regex formulas and vset-automata) of the atomic extractors. Queries in the spanner framework are defined similarly to the world of relational databases, with two differences. First, the input is not a relational database, but rather a document. Second, the atoms are not relational symbols, but rather atomic extractors. Therefore, the relations on which the query is applied are implicitly defined as those obtained by evaluating each regex formula or vset-automaton over the input document.

Next, we introduce each chapter of the thesis.

**Expressive Power and Descriptive Complexity**

In Chapter 3 we explore the power of recursion in expressing spanners. The motivation came from SystemT developers, who have interest in recursion for various reasons, such as programming natural-language parsers by means of context-free grammars \[52\]. Specifically, we consider the language RGXlog of spanners that are defined via Datalog programs where regex formulas play the role of the extensional database (EDB). More precisely, given a document \(d\), the regex formulas extract EDB relations from \(d\), and a designated relation \(OUT\) captures the output of the program. Observe that such a program operates exclusively over the domain of spans of the input document. In particular, the output is a relation over spans of \(d\), and hence, RGXlog is yet another spanner representation language.

We explore the expressiveness of RGXlog. Without recursion, RGXlog captures
precisely the regular spanners \cite{28}. With recursion, several observations are quite straightforward. First, we can write a program that determines whether \(x\) and \(y\) span the same substring. Hence, we have string equality without explicitly including the string-equality selection. It follows that every core spanner can be expressed in \texttt{RGXlog}. Moreover, \texttt{RGXlog} can express more than core spanners. For instance, \texttt{RGXlog} can select all pairs of spans with the same length; this, as said earlier, is not expressible by a core spanner \cite{27}. What about upper bounds? A clear upper bound is polynomial time: every \texttt{RGXlog} program can be evaluated in polynomial time. Note that in our complexity analysis the program is regarded as fixed and the input consists of only the document. That is, we use data complexity. We can, therefore, conclude that \texttt{RGXlog} can express only spanners computable in polynomial time.

We continue our investigation by diving deeper into the relationship between \texttt{RGXlog} and core spanners. The inexpressiveness results to date are based on the aforementioned core simplification lemma \cite{27}. The proof of this lemma heavily relies on the absence of the difference operator in the algebra. In fact, Freydenberger and Holldack \cite{34} showed that it is unlikely that in the presence of difference, there is a result similar to the core simplification lemma. So, we extend the algebra of core spanners with the difference operator, and call a spanner of this extended language a generalized core spanner. We then ask whether (a) every generalized core spanner can be expressed in \texttt{RGXlog} (whose syntax is positive and excludes difference/negation), and (b) \texttt{RGXlog} can express only generalized core spanners.

The answer to the first question is positive. We establish a negative answer to the second question by deploying the theory of Presburger arithmetic \cite{67} (first-order theory of the natural numbers with the addition binary function and the constants zero and one). Specifically, we consider Boolean spanners on a unary alphabet. Each such spanner can be viewed as a predicate over natural numbers: the lengths of the documents that are accepted (evaluated to \texttt{true}) by the spanner. We prove that every predicate expressible by a Boolean generalized core spanner is also expressible in Presburger arithmetic. Yet, we show a very simple \texttt{RGXlog} program that expresses a selection operator that is not expressible in Presburger arithmetic, namely being a power of two \cite{50}.

We prove that \texttt{RGXlog} can express every spanner computable in polynomial time. Formally, recall that a spanner is a function \(P\) that maps an input document \(d\) into a relation \(P(d)\), over a fixed schema \(S_P\), over the spans of \(d\). We prove that the following are equivalent for a spanner \(P\): (a) \(P\) is expressible in \texttt{RGXlog}, and (b) \(P\) is computable in polynomial time. As a special case, Boolean \texttt{RGXlog} captures exactly the polynomial-time languages.

We prove equivalence to polynomial time via a result by Papadimitriou \cite{60}, stating that semipositive Datalog (i.e., Datalog where only EDB relations can be negated) can express every database property computable in polynomial time, under certain assumptions: (a) the property is invariant under isomorphism, (b) a successor relation that defines a linear order over the domain is accessible as an EDB, and (c) the first and last elements in the database are accessible as constants (or single-element EDBs). We show that in the case of \texttt{RGXlog}, we get
all of these for free, due to the fact that our EDBs are regex formulas. Specifically, in string logic (over a finite alphabet), isomorphism coincides with identity, the negation of EDBs (regex formulas) is expressible as EDBs (regex formulas), and we can express a linear order by describing a successor relation along with its first and last elements.

Interestingly, our construction shows that, to express polynomial time, it suffices to use regex formulas with only two variables. In other words, binary regex formulas already capture the entire expressive power. Can we get away with only unary regex formulas? Using past results on monadic Datalog [43] and non-recursive RGXlog [27] we conclude a negative answer—Boolean RGXlog with unary regex formulas can express precisely the class of Boolean regular spanners. Moreover, the class of non-Boolean spanners expressible by RGXlog with unary regex formulas is strictly contained in the class of non-Boolean regular spanners.

Lastly, in the Appendix, we analyze recursive Datalog programs in a theoretical framework that generalizes both the relational and the spanner model. The framework, introduced by Nahshon, Peterfreund and Vansummeren [57] and referred to as Spannerlog(RGX), aims to establish a unified query language for combining structured and textual data. In this framework, the input and output databases consist of relations that have two types of attributes: strings and spans. In the associated Datalog program, we refer to the relations of the input database as EDB relations and to those of the output database as IDB relations (that is, intensional, or inferred). The body of a Datalog rule may have three types of atoms: EDB, IDB, and regex formulas over string attributes. We prove that Spannerlog(RGX) with stratified negation, restricted to string EDB relations, can express precisely the functions that are computable in polynomial-time.

Summary of Contribution

- We show that the class of Boolean generalized core spanners over unary alphabets has the same expressive power as Presburger Arithmetics. That is, if we view a language over a unary alphabet as a set of positive integers (corresponding with the lengths of the words of the language) then every language accepted by a Boolean generalized core spanner is definable in Presburger arithmetic and vice-versa.

- We prove that recursive RGXlog captures exactly the class of spanners computable in polynomial time in data complexity. In particular, we prove that to capture the class of polynomial-time spanners it suffices to use regex formulas with only two variables. On the contrary, we show that recursive RGXlog over monadic regex formulas is less expressive than regular spanners.

- We extend our study to Spannerlog(RGX)—a recently proposed theoretical framework that generalizes both the relational and document spanners model. We show that in this framework, recursive Datalog-like programs with stratified negation, restricted to string EDB relations, can express precisely the class of polynomial-time computable functions.
Combined Complexity

Our previous analysis of the expressive power of RGXlog resulted in an exact characterization of the class of spanners that are computable in polynomial time. Note that our analysis treats the program as fixed. In Chapter 4, however, we treat the spanner representation as part of the input. For complexity analysis, there are important advantages to yardsticks that regard the spanner representation as input. First, the size of the regular atomic extractors (i.e., regex formulas and vset-automata) can be quite large in practice. Taking examples from RegExLib.com, each of the regexes for recognizing the RFC 2822 mailbox format (regexp id 711) and date format (regexp id 969) uses more than 350 ASCII symbols, and a regex for identifying US addresses (regexp id 1564) uses more than 2,000 ASCII symbols. Furthermore, automata may be constructed by automatic (machine-learning) processes that achieve accuracy through the granularity of the automaton. The paradigm of Artificial Neural Networks (ANNs) in natural language processing has motivated the conversion of ANN models such as recurrent neural networks and convolutional neural networks into automata [59, 63, 78], where the number of states may reach tens of thousands to match the expressiveness of the numeric parameters [78].

Architectures that motivated the modeling of the document spanners framework, e.g., SystemT and Xlog, uses Conjunctive Queries (CQs) and Unions of CQs (UCQs) over regex formulas. Such regex (U)CQs also constitute the language underlying more recent theoretical extensions [28, 57]. These queries are another representation language for regular spanners, yet, very little was initially known about the combined complexity of their evaluation (i.e., the complexity when considering both the document and query as input).

One might be tempted to claim that the literature on evaluation of CQs and UCQs should draw the complete picture on complexity, by what we refer to as the canonical relational evaluation: evaluate each individual regex formula on the input document, and compute the UCQ as if it were a query on an ordinary relational database. There are, however, several problems with this claim.

The first problem is that lower bounds on relational query evaluation do not carry over immediately to our settings since the input is not an arbitrary relational database, but rather a very specific one: every relation is obtained by applying a regex to the (same) document. However, we show that the standard lower bounds of NP-completeness [16] for Boolean queries (as well as W[1]-hardness for some parameters [61]) remain in our setting. The more surprising finding is that hardness holds even if the document is a single character. That is, even in terms of query complexity (where the document is regarded as fixed and the query is regarded as input) we hit hardness.

The second problem with the canonical relational evaluation approach is that it may be infeasible to materialize the relation defined by a regex formula, as the number of tuples in that relation may be exponential in the size of the input. This problem provably increases the complexity, as we show that Boolean regex-CQ evaluation is NP-complete even on acyclic CQs, and even on the more restricted gamma-acyclic CQs [26]. In contrast, acyclic CQs (and more generally CQs of...
bounded hypertree width \cite{11} admit polynomial-time evaluation in the relational world. Finally, even if we were guaranteed a polynomial bound on the result of each atomic regex formula, it would not necessarily mean that we can actually materialize the corresponding relation in polynomial time.

In spite of the above daunting complexity, we are able to establish some substantial upper bounds that are based on an algorithm that we devise for evaluating a vset-automaton over a document. More formally, recall that a vset-automaton $A$ represents a spanner, which we denote as $[A]$. When evaluating $A$ on a document $d$, the result is a relation $[A](d)$ over the spans of $d$. The number of tuples in $[A](d)$ can be exponential in the size of the input ($d$ and $A$). Our algorithm takes as input a document $d$ and a vset-automaton $A$ (or, more precisely, a functional vset-automaton \cite{33}) and enumerates the tuples of $[A](d)$ with polynomial delay \cite{18}. This is done by a nontrivial reduction to the problem of enumerating all the words of a specific length accepted by an NFA \cite{3}. Our algorithm implies several upper bounds, which we establish via two approaches: (a) the above-mentioned canonical relational evaluation, and (b) the compilation approach.

The first approach utilizes known algorithms for relational UCQ evaluation, by materializing the relations defined by the regex formulas. For that, we devise an efficient method to compile a regex formula into a vset-automaton, and establish that a regex formula can be evaluated in polynomial total time \cite{18} (and even polynomial delay). In particular, we can efficiently materialize the relations of each atom of a regex UCQ, whenever we have a polynomial bound on the cardinality of this relation. Hence, there is no need for a specialized algorithm for each cardinality guarantee—one algorithm fits all. Consequently, under such cardinality guarantees, canonical relational evaluation is efficient whenever the underlying UCQ is tractable (e.g., each CQ is acyclic).

In the second approach, compilation to automata, we compile the entire regex UCQ into a vset-automaton. Combining our polynomial-delay algorithm with known results \cite{27,32}, we conclude that regex UCQs can be evaluated with Fixed-Parameter Tractable (FPT) delay when the size of the UCQ is the parameter. Moreover, we prove that the compilation is efficient if every disjunct (regex CQ) has a bounded number of atoms. In particular, we show that every join of a bounded number of vset-automata can be compiled in polynomial time into a single vset-automaton, and every projection and union (with no bounds) over vset-automata can be compiled in polynomial time into a single vset-automaton. Hence, we establish that for every fixed $k$, the evaluation of regex $k$-UCQs (where each CQ has at most $k$ atoms) can be performed with polynomial delay.

Finally, we generalize our results to regex UCQs that include string-equality selections which are another representation system for the class of core spanners. Freydenberger et al. \cite{33} have shown that adding an unbounded number of string equalities can make a tractable regex UCQ intractable, even if that regex UCQ is a single regex formula. We prove that now we no longer retain the above FPT delay, as the problem is $\text{W}[1]$-hard when the parameter is the size of the query.

Nevertheless, much of the two evaluation approaches generalize to string equalities. While this generalization is immediate for the first approach, the
second faces a challenge—it is impossible to compile string equality into a vset-automaton [27]. Yet, we show that with our compilation techniques, one can compile in string equality for the specific input document at hand (i.e., not statically but rather at runtime). We then conclude that regex $k$-UCQs with a bounded number of string equalities can be evaluated with polynomial delay. We refer to this compilation method as dynamic compilation as opposed to static compilation that is independent of the document. We revisit these methods in the sequel in context of schemaless spanners.

Finally, we show that in the presence of the difference operator, emptiness becomes NP-hard. We also revisit the difference in context of schemaless spanners and present some restrictions that allow us to obtain a tractable evaluation.

**Summary of Contribution**

- We present two approaches for the evaluation of (U)CQs over regex formulas: (a) the canonical relational evaluation approach in which we materialize the regex formulas and then evaluate the query as if it were a standard relational query, and (b) the compilation approach in which we compile the query into a functional vset-automata and then run it on the document. We show that there are queries for which we obtain tractable evaluation with the first approach but not with the second, and vice-versa.

- We show that even if the underlying query of a regex CQ is tractable it does not imply that the evaluation can be done efficiently since the size of the materialized relations might be exponential. However, if the size of the materialized relations is polynomially bounded (and the underlying query is tractable) then the evaluation can be done efficiently.

- We obtain upper bounds for the evaluation of regex $k$-(U)CQs and extend these to queries that incorporate string equalities.

**Complexity of Schemaless Spanners**

Previously, we have considered schema-based spanners that extract relations from input documents and hence incapable of handling incomplete information. Since the data extracted from text is often incomplete, Maturana et al. [55] introduced the schemaless semantics that allows for incomplete extraction from documents, in the spirit of the SPARQL model [64]. In Chapter 5, we focus on spanners interpreted with the schemaless semantics, namely the schemaless spanners.

Recall that each schema-based spanner $P$ is associated with a fixed and finite set $V_P$ of variables, playing the roles of attributes in relational databases, so that every tuple that $P$ extracts from a document assigns a value to each of the variables of $V_P$. The regex formulas conform to this property are said to be functional. Freydenberger [32] has applied the property of functionality to vset-automata: a vset-automaton is functional if every accepting path properly opens and closes every variable exactly once. The functionality property can be
tested in polynomial time for both regex formulas \cite{27} and vset-automata \cite{31}. Moreover, functional vset-automata generalize functional regex formulas in the sense that every instance of the former can be transformed in linear time into an instance of the latter (but not necessarily the other way around). It is important to note that all of the results we have presented previously (and in particular the upper bounds) were obtained under the explicit assumption that we deal with only functional vset-automata and functional regex formulas.

In the schemaless case, two extracted tuples may assign spans to different sets of variables. The analog of functionality is sequentiality: a regex formula is sequential if every parse tree includes at most one occurrence of every variable, and a vset-automaton is sequential if every accepting path properly opens and closes every variable at most once. Again, in polynomial time we can test for sequentiality and transform a sequential regex formula into a sequential vset-automaton; moreover, Amarilli et al. \cite{7} have shown that sequential vset-automata can be evaluated with polynomial delay under combined complexity. In fact, the aforementioned algorithm of Amarilli et al. enumerates with polynomial delay under combined complexity, and, under data complexity, with constant delay following a linear pre-processing of the document \cite{1}. Since functional vset-automata are also sequential, this algorithm also applies to the schema-based spanners, and generalizes the applicability of the enumeration algorithm of Florenzano et al. \cite{30}.

The state of affairs leaves open a fundamental question regarding the combined complexity of query evaluation: Does the tractability for the positive relational algebra generalize from the schema-based case to the schemaless case? We prove that the answer to this question is negative. More specifically, it is NP-complete to determine whether the natural join of two sequential regex formulas is nonempty.

We formulate various syntactic restrictions that allow avoiding hardness. In particular, we show that polynomial delay is retained if we bound the number of common variables between the two operands of the natural join. We show that with the same restriction we obtain a polynomial delay evaluation also for the difference operator. Recall that we hit NP-hardness in the presence of difference, even when we considered only functional regex formulas. For the natural join, we also present a normal form for schemaless regex formulas and vset-automata, namely disjunctive functional, that are more restricted than, yet as expressive as, their sequential counterparts; the natural join of two disjunctive-functional vset-automata can be compiled into a disjunctive-functional vset-automaton in polynomial time and hence, evaluated with polynomial delay.

In contrast to the natural join, the tractability of the difference between sequential vset-automata with a bounded number of common variables cannot be established via compilation into a single vset-automaton. This is due to the simple reason that, in the case of Boolean spanners, the problem is the same as the difference between two NFAs, where the compilation necessitates an exponential blowup \cite{17}. Nevertheless, we establish the tractability by transforming the difference into a natural join with a special vset-automaton that is built ad-hoc for the input document using the aforementioned dynamic compilation approach.

\footnote{This is the spanner analog of a recent line of work on the enumeration complexity of database and string queries \cite{8,15,58,70}.}
In summary, our complexity upper bounds are established via two main methods within the compilation approach presented in Chapter 4. The first is based on a static document-independent compilation of the input vset-automata (or regex formulas) into a new vset-automaton. The second is based on a dynamic compilation of both the input vset-automata and the input document into a new, ad-hoc vset-automaton. Recall that in Chapter 4 we used static compilation to obtain an efficient evaluation of regex (U)CQs and dynamic compilation for evaluating queries with string equalities.

Finally, we compose our tractability results into more a general class of queries by proposing a new complexity measure that is specialized to spanners. Recall that the evaluation problem has three components: the document, the atomic extractors (e.g., regex formulas), and the relational algebra (RA) that combines these atomic extractors, which we refer to as the RA tree. Under combined complexity, all three are given as input; under data complexity, the document is given as input and the rest are fixed; there is also the expression complexity where the document is fixed and the rest is given as input. We propose the extraction complexity, where the RA tree is fixed, and the input consists of the document and the atomic spanners (mapped to their corresponding positions in the RA tree). We present and discuss conditions that cast the extraction complexity tractable (polynomial-delay evaluation) and intractable (NP-hard nonemptiness). Interestingly, since the tractability of an RA tree is based on a dynamic compilation, we can incorporate there any polynomial-time spanner, as long as its arity is bounded by a constant.

Summary of Contribution

- Although the evaluation of sequential regex formulas (for extracting incomplete information) was shown to be doable with polynomial delay (in combined complexity), we show that evaluating the join of sequential regex formulas is not tractable and devise restrictions that allow us to obtain an efficient evaluation. In particular, we suggest a new normal form for sequential regex formulas, namely disjunctive normal form that allows maintaining expressiveness yet compute the join efficiently.

- We distinguish between two methods within the compilation approach: (a) static compilation in which the resulting vset-automaton does not depend on the document, versus (b) dynamic compilation in which it does depend on the document. Dynamic compilation allows us to deal with black-box extractors. It also allows to include the difference operator and selection predicates that provably cannot be handled via the static compilation method.

- We suggest a novel complexity measure, namely extraction complexity, that lies within data and combined complexity and is unique for document spanners. Extraction complexity allows us to characterize the RA trees that can be evaluated efficiently.
Structure of the Thesis

We start in Chapter 2 with preliminaries and technical background of the document spanners framework. We then move on to discussing expressiveness and descriptive complexity of (recursive) Datalog-like programs in Chapter 3. We continue in Chapter 4 with the combined complexity analysis. In Chapter 5, we analyze the complexity of the evaluation of schemaless spanners for extracting incomplete information.

Figure 1.1: The expressive power of different classes of document spanners and equivalent formalisms that define these classes.
Chapter 2

Preliminaries

In this chapter, we introduce the basic terminology and notation that we use throughout the thesis, mainly from the literature on document spanners [27, 35, 66].

2.1 Document Spanners

We begin with basic definitions from the framework of document spanners [27].

We fix a finite alphabet \( \Sigma \) of symbols. A document (or string) \( d \) is a finite sequence \( \sigma_1 \cdots \sigma_n \) over \( \Sigma \) (i.e., each \( \sigma_i \in \Sigma \)). The length \( |d| \) of \( d := \sigma_1 \cdots \sigma_n \) is \( n \). We denote by \( \Sigma^* \) the set of all documents over \( \Sigma \). A language over \( \Sigma \) is a subset of \( \Sigma^* \). A span identifies a substring of \( d \) by specifying its bounding indices.

Formally, a span of \( d \) has the form \([i, j] \) where \( 1 \leq i \leq j \leq n + 1 \). If \([i, j] \) is a span of \( d \), then \( d_{[i,j]} \) denotes the substring \( \sigma_i \cdots \sigma_{j-1} \). Note that \( d_{[i,i]} \) is the empty document, and that \( d_{[1,n+1]} = \varepsilon \) where \( \varepsilon \) stands for the empty string. We denote by \( \text{Spans} \) the set of all spans of all documents, that is, all expressions \( [i, j] \) where \( 1 \leq i \leq j \). By \( \text{Spans}(d) \) we denote the set of spans of the document \( d \) (and in this case we have \( j \leq n + 1 \)).

We assume an infinite set \( Vars \) of variables, disjoint from \( \Sigma \). For a finite \( V \subset Vars \) and \( d \in \Sigma^* \), a \((V, d)\)-record is a mapping \( \mu \) that maps each variable in \( V \) to a span of \( d \). If context allows, we write \( V \text{-record} \) or record instead of \((V, d)\)-record. A set of \((V, d)\)-records is called a \((V, d)\)-relation. A document spanner, or spanner for short, is a function \( P \) that is associated with a finite variable set \( V \), denoted \( \text{Vars}(P) \), and that maps every document \( d \in \Sigma^* \) to a \((V, d)\)-relation \( P(d) \). A spanner \( P \) is Boolean if \( \text{Vars}(P) = \emptyset \). If \( P \) is Boolean, then either \( P(d) = \emptyset \) or \( P(d) \) contains only the empty \((\emptyset, d)\)-record; we interpret these two cases as false and true, respectively.

Example 2.1.1. In this and in the following examples of this chapter, we assume our alphabet consists of the lower-case and capitalized English alphabet letters \( a, \cdots, z, A, \cdots, Z \) and the symbol \( ' \) that stands for whitespace. Let us define

\(^1\)Fagin et al. [27] refer to \((V, d)\)-records as \((V, d)\)-tuples; we use “record” to avoid confusion with the concept of “tuple” that we later use in ordinary relations.
\[ d := \text{chocolate} \text{cookie} \] and let \( X = \{x, y\} \). We consider the following \((X, d)\)-relation which consists of the two \((X, d)\)-records \( \mu_1 \) and \( \mu_2 \).

\[
\begin{array}{c|cc}
  & x & y \\
\hline
\mu_1 & [1,9) & [10,16) \\
\mu_2 & [10,16) & [1,9) \\
\end{array}
\]

Note that \( d_{[1,9)} = \text{chocolate} \) and that \( d_{[10,16)} = \text{cookie} \). In Example 2.2.3, we show a document spanner \( P \) such that the above relation is the result of applying \( P \) on \( d \).

\[ \square \]

### 2.2 Spanner Representations

We use two models as basic building blocks for spanner representations: \textit{regex formulas} and \textit{vset-automata}. Regex formulas (vset-automata) can be understood as extensions of regular expressions (NFAs) with variables. Both models were introduced by Fagin et al. \[27\], and following Freydenberger \[32\] we define the semantics of these models using so-called ref-words \[60\] (short for reference-words). Before defining the semantics, we define ref-words.

#### 2.2.1 Ref-Words

For a finite variable set \( V \subset \text{Vars} \), ref-words are defined over the extended alphabet \( \Sigma \cup \Gamma_V \), where \( \Gamma_V \) consists of two symbols, \( x^+ \) and \( -x \), for each variable \( x \in V \). We assume that \( \Sigma \) and \( \Gamma_V \) are disjoint. Intuitively, the letters \( x^+ \) and \( -x \) represent opening or closing a variable \( x \). Hence, ref-words extend documents over \( \Sigma \) with an encoding of variable operations. As we shall see, treating these variable operations as letters allows us to adapt techniques from automata theory.

A ref-word \( r \in (\Sigma \cup \Gamma_V)^* \) is \textit{valid for} \( V \) if each variable of \( V \) is opened and then closed exactly once, or more formally, for each \( x \in V \) the string \( r \) has precisely one occurrence of \( x^+ \), precisely one occurrence of \( -x \), and the former occurrence takes place before (i.e., to the left of) the latter.

**Example 2.2.1.** Let \( V := \{x\} \), and define the ref-words \( r_1 := c \cdot x^+ \cdot \text{oo} \cdot -x \cdot \text{kie} \), \( r_2 := x^+ \cdot -x \), \( r_3 := -x \cdot a \cdot x^+ \), and \( r_4 := x^+ \cdot a \cdot -x \cdot x^+ \cdot a \cdot -x \). Then \( r_1 \) and \( r_2 \) are valid for \( V \), but \( r_3 \) and \( r_4 \) are not. Note that \( r_1 \) and \( r_2 \) are not valid for \( V' \) with \( V' \supset V \), as all variables of \( V' \) must be opened and closed.

\[ \square \]

If \( V \) is clear from the context, we simply say that a ref-word is valid. To connect ref-words to terminal strings and later to spanners, we define a morphism \( \text{clr} : (\Sigma \cup \Gamma_V)^* \rightarrow \Sigma^* \) by \( \text{clr}(\sigma) := \sigma \) for \( \sigma \in \Sigma \), and \( \text{clr}(a) := \epsilon \) for \( a \in \Gamma_V \). For \( d \in \Sigma^* \), let \( \text{Ref}(d) \) be the set of all valid ref-words \( r \in (\Sigma \cup \Gamma_V)^* \) with \( \text{clr}(r) = d \).

By definition, every \( r \in \text{Ref}(d) \) has a unique factorization \( r = r'_x \cdot x^+ \cdot r_x \cdot -x \cdot r''_x \) for each \( x \in V \). With these factorizations, we interpret \( r \) as a \((V, d)\)-record \( \mu^r \) by defining \( \mu^r(x) := [i, j) \), where \( i := |\text{clr}(r'_x)| + 1 \) and \( j := i + |\text{clr}(r_x)| \). Intuitively, \( \text{clr}(r_x) \) contains \( d_{\mu^r(x)} \), while \( \text{clr}(r'_x) \) contains the prefix of \( d \) to the left.
An alternative way of understanding $\mu^r = [i, j]$ is that $i$ is chosen such that $x^+$ occurs between the positions in $r$ that are mapped to $\sigma_{i-1}$ and $\sigma_i$, and $\neg x$ occurs between the positions that are mapped to $\sigma_{j-1}$ and $\sigma_j$ (assuming that $d = \sigma_1 \cdots \sigma_{|d|}$, and slightly abusing the notation to avoid a special distinction for the non-existing positions $\sigma_0$ and $\sigma_{|d|+1}$).

**Example 2.2.2.** Let $d := \text{cookie}$, and define $r_1 := c x^+ \text{oo} \neg x \text{kie}$, and let $r_2 := \text{cookie} x^+ \neg x$. Let $V := \{x\}$. Then $r_1, r_2 \in \text{Ref}(d)$, with $\mu^{r_1}(x) := [2, 4]$ and $\mu^{r_2}(x) := [7, 7]$. □

### 2.2.2 Regex Formulas

A *regex formula* (over $\Sigma$) is a representation of a spanner by means of a regular expression with *capture variables*. It is defined by

$$\alpha := \emptyset | \epsilon | \sigma | \alpha \lor \alpha | \alpha \cdot \alpha | \alpha^* | x\{\alpha\}$$

Here, $\epsilon$ stands for the empty string, $\sigma \in \Sigma$, and the extension beyond regular expressions is $x\{\alpha\}$ where $x$ is a variable in $\text{Vars}$. We denote the set of variables that occur in $\alpha$ by $\text{Vars}(\alpha)$. For convenience, we sometimes put regex formulas in parentheses and also omit parentheses, as long as the meaning remains clear. In addition, instead of $\alpha \cdot \alpha^*$ we use the abbreviation $\alpha^+$. For operator precedence, we assume that $\ast$ comes before $\cdot$, which comes before $\lor$.

We interpret each regex formula $\alpha$ as a generator of a ref-word language $\mathcal{R}(\alpha)$ over the extended alphabet $\Sigma \cup \Gamma_{\text{Vars}(\alpha)}$. If $\alpha$ is of the form $x\{\beta\}$, then $\mathcal{R}(\alpha) := x^+\mathcal{R}(\beta)^+\neg x$. Otherwise, $\mathcal{R}(\alpha)$ is defined like the language $\mathcal{L}(\alpha)$ of a regular expression; for example, $\mathcal{R}(\alpha \cdot \beta) := \mathcal{R}(\alpha) \cdot \mathcal{R}(\beta)$. For every document $d \in \Sigma^*$, we define $\text{Ref}(\alpha, d) = \mathcal{R}(\alpha) \cap \text{Ref}(d)$. In other words, $\text{Ref}(\alpha, d)$ contains exactly those valid ref-words from $\mathcal{R}(\alpha)$ that $\text{clr}$ maps to $d$.

Finally, the spanner $[\alpha]$ is the one that defines the following $(\text{Vars}(\alpha), d)$-relation for every document $d \in \Sigma^*$:

$$[\alpha](d) := \{\mu^r | r \in \text{Ref}(\alpha, d)\}$$

We say that a regex-formula $\alpha$ is *functional* if every ref-word in $\mathcal{R}(\alpha)$ is valid. Fagin et al. [27] gave an algorithm for testing whether a regex formula is functional. By analyzing the complexity of their construction, we conclude the following.

**Theorem 2.2.1.** [27] Whether a regex formula $\alpha$ with $v$ variables is functional can be tested in $O(|\alpha| \cdot v)$ time.

**Example 2.2.3.** Consider the following regex formula $\alpha$.

$$\Sigma^*(\{x\{\text{chocolate}\} \_ \_ y\{\text{cookie}\}\} \lor y\{\text{chocolate}\} \_ \_ x\{\text{cookie}\})$$

where `\_\_` stands for whitespace. It holds that $\text{Vars}(\alpha) = \{x, y\}$. For $d \in \Sigma^*$, the $(\text{Vars}(\alpha), d)$-relation $[\alpha](d)$ contains every $(\text{Vars}(\alpha), d)$-record $\mu$ that satisfies $(d_{\mu(x)} = \text{chocolate}$ and $d_{\mu(y)} = \text{cookie})$ or $(d_{\mu(y)} = \text{chocolate}$ and $d_{\mu(x)} = \text{cookie})$. Note that the relation in Example 2.1.1 is the result of applying $[\alpha]$ on $d$ (from the same example). □
Following the convention made by Fagin et al. [27], in what follows we assume that regex formulas are functional unless explicitly noted, and denote this class by \( \text{RGX} \). Only in Chapter 5 do we consider a broader class of regex formulas, namely sequential regex formulas, for extracting incomplete information [55].

### 2.2.3 Variable-Set Automata

Another basic way of representing spanners is by means of finite state machines. A variable-set automaton (or \( vset \)-automaton for short) with variables from a finite set \( V \subset \text{Vars} \) can be understood as an \( \epsilon \)-NFA (i.e., an NFA with epsilon transitions allowed) that is extended with edges labeled with variable operations \( x^+ \) or \( \neg x \) for \( x \in V \). Formally, a \( vset \)-automaton is a tuple \( A := (V, Q, q_0, q_f, \delta) \), where \( V \) is a finite set of variables, \( Q \) is the set of states, \( q_0, q_f \in Q \) are the initial and the final states, respectively, and \( \delta : Q \times (\Sigma \cup \{\epsilon\} \cup V) \to 2^Q \) is the transition function. By \( \text{Vars}(A) \) we denote the set \( V \).

The \( vset \)-automaton \( A := (V, Q, q_0, q_f, \delta) \) can be interpreted as a directed graph, where the nodes are the states, and every \( q \in \delta(p, a) \) is represented by an edge from \( p \) to \( q \) with the label \( a \). To define the semantics of \( A \), we first interpret \( A \) as an \( \epsilon \)-NFA over the alphabet \( \Sigma \cup \Gamma_V \), and define its ref-word language \( \mathcal{R}(A) \) as the set of all ref-words \( r \in (\Sigma \cup \Gamma_V)^* \) such that some path from \( q_0 \) to \( q_f \) is labeled with \( r \).

Let \( A \) be a \( vset \)-automaton. Analogously to regex formulas, we denote by \( \text{Ref}(A) \) the set of all ref-words in \( \mathcal{R}(A) \) that are valid for \( V \), and define \( \text{Ref}(A, d) \) and \( [A](d) \) accordingly for every \( d \in \Sigma^* \). We denote the class of \( vset \)-automata by \( \text{VA} \). A \( vset \)-automaton \( A \) is said to be functional if \( \text{Ref}(A) = \mathcal{R}(A) \); (i.e., every accepting run of \( A \) generates a valid ref-word). Two \( vset \)-automata \( A_1 \) and \( A_2 \) are equivalent if \( [A_1] = [A_2] \).

**Example 2.2.4.** Let \( A \) be the following \( vset \)-automaton:

\[
\begin{array}{c}
\text{x}^+, \text{a}, \neg \text{x} \\
\end{array}
\]

Then, the following hold.

\[
\mathcal{R}(A) = \{x^+, a, \neg x\}^* \\
\text{Ref}(A) = \{a^i x^+ a^j \neg x a^k \mid i, j, k \geq 0\}
\]

Hence, \( A \) is not functional, since \( \mathcal{R}(A) \) contains invalid ref-words such as \( \epsilon, x^+, \neg x a x^+, \) and \( x^+ a \neg x \neg x \). Now consider the \( vset \)-automaton \( A_{\text{fun}} \):

\[
\begin{array}{c}
\text{x}^+, \text{a}, \neg \text{x} \\
\end{array}
\]

Then \( \text{Ref}(A_{\text{fun}}) = \mathcal{R}(A_{\text{fun}}) = \text{Ref}(A) \). Hence, \( A_{\text{fun}} \) is functional, and \( A_{\text{fun}} \) and \( A \) are equivalent.

For all \( d \in \Sigma^* \setminus \{a\}^* \), \( [A](d) = \emptyset \); and for all \( d \in \{a\}^* \), \( [A](d) \) contains all possible \( \{x\}, d \)-records. \( \square \)
While $\text{Ref}(A_1) = \text{Ref}(A_2)$ is a sufficient criterion for the equivalence of $A_1$ and $A_2$, it is necessary only if the automata have at most one variable. For example, consider $r_1 := x^+ y^+ -xz+yz$ and $r_2 := y^+ -yx x^+ -xz$. Both are valid ref-words that define the same $(\{x, y\}, \epsilon)$-record $[1, 1]$, but $r_1 \neq r_2$.

Example 2.2.4 suggests that vset-automata can be converted into equivalent functional vset-automata; but Freydenberger [32] showed that although this is possible with standard automata constructions, the resulting blow-up may be exponential in the number of variables.

As shown in Lemma 4.4.1, every functional regex formula can be converted into an equivalent functional vset-automaton; and we use this connection as we shall see that functional vset-automata are a convenient tool for working with regex formulas. In contrast to this, vset-automata in general can be quite inconvenient (e.g., even deciding emptiness of $[A](\epsilon)$ is NP-complete if $A$ is not functional, see [32]).

Freydenberger [32] established the complexity of testing whether a given vset-automaton is functional.

**Theorem 2.2.2.** [32] Whether a given vset-automaton $\alpha$ with $n$ states, $m$ transitions and $v$ variables is functional can be tested in $O(vm + n)$ time.

A special property of functional vset-automata is that each state implicitly stores which variables have been opened and closed. We discuss it in detail in Chapter 4, Section 4.3.

### 2.3 Algebraic Operators for Spanners

Let $P$, $P_1$ and $P_2$ be spanners. The algebraic operators union, projection, natural join, and selection are defined as follows:

**Union** If $\text{Vars}(P_1) = \text{Vars}(P_2)$, their union $(P_1 \cup P_2)$ is defined by $\text{Vars}(P_1 \cup P_2) := \text{Vars}(P_1)$ and $(P_1 \cup P_2)(d) := P_1(d) \cup P_2(d)$ for all $d \in \Sigma^*$.

**Projection** Let $Y \subseteq \text{Vars}(P)$. The projection $\pi_Y P$ is defined by $\text{Vars}(\pi_Y P) := Y$ and $\pi_Y P(d) := P|_Y(d)$ for all $d \in \Sigma^*$, where $P|_Y(d)$ is the restriction of all $\mu \in P(d)$ to $Y$.

**Natural join** Let $V_i := \text{Vars}(P_i)$ for $i \in \{1, 2\}$. The (natural) join $(P_1 \bowtie P_2)$ of $P_1$ and $P_2$ is defined by $\text{Vars}(P_1 \bowtie P_2) := \text{Vars}(P_1) \cup \text{Vars}(P_2)$ and, for all $d \in \Sigma^*$, $(P_1 \bowtie P_2)(d)$ is the set of all $(V_1 \cup V_2, d)$-records $\mu$ for which there exist $\mu_1 \in P_1(d)$ and $\mu_2 \in P_2(d)$ with $\mu_i|_{V_i}(d) = \mu_i(d)$ and $\mu|_{V_2}(d) = \mu_2(d)$.

**Difference** If $\text{Vars}(P_1) = \text{Vars}(P_2)$, their difference $P_1 \setminus P_2$ is defined by $\text{Vars}(P_1 \setminus P_2) = \text{Vars}(P_1)$ and $(P_1 \setminus P_2)(d) = P_1(d) \setminus P_2(d)$. 

19
Selection Let \( R \subseteq (\Sigma^*)^k \) be a \( k \)-ary relation over \( \Sigma^* \). The selection operator \( \zeta_R \) is parameterized by \( k \) variables \( x_1, \ldots, x_k \in \text{Vars}(P) \), written as \( \zeta_R[x_1, \ldots, x_k] \). The selection \( \zeta_R[x_1, \ldots, x_k] P \) is defined by \( \text{Vars}(\zeta_R[x_1, \ldots, x_k] P) := \text{Vars}(P) \) and, for all \( d \in \Sigma^* \), \( \zeta_R[x_1, \ldots, x_k] P(d) \) is the set of all \( \mu \in P(d) \) for which \( (d_{\mu(x_1)}, \ldots, d_{\mu(x_k)}) \in R \). Following Fagin et al. [27], we almost exclusively consider binary string equality selections, \( \zeta_{x, y} P \), which selects all \( \mu \in P(d) \) where \( d_{\mu(x)} = d_{\mu(y)} \).

Note that the join operator joins tuples that have identical spans in their shared variables. In contrast, the selection operator compares the substrings of \( d \) that are described by the spans and does not distinguish between different spans that span the same substrings.

We allow the use of these operators for spanners represented by regex formulas or vset-automata and also for more complex spanner representations, e.g., \([A_1 \bowtie A_2]\). In this case, we use the abbreviated notation \([A_1 \bowtie A_2]\) instead of \([A_1 \bowtie A_2]\).

We use an equivalent abbreviated notation for all other operators as well.

**Example 2.3.1.** Let \( \alpha \) be the regex formula that captures all spans \( x \) and \( y \) such that each spans a token and the token spanned by \( x \) ends right before the token spanned by \( y \) begins:

\[
\alpha(x, y) := \Sigma^* \cup x\{\Delta^*\} \cup y\{\Delta^*\} \cup \Sigma^*
\]

where \( \Delta = \{a, \ldots, z, A_1, \ldots, Z\} \) and \( \Sigma = \Delta \cup \{\_\} \). The following algebraic expression is denoted by \( \beta \):

\[
\pi_{x, x'} \left( e_y x \left( \alpha(x, y) \bowtie \alpha(x', y') \right) \right)
\]

Observe that given a document \( d \), the spanner \([\beta]\) extracts all the spans \( x \) and \( x' \) that are followed by the same token. \( \square \)

### 2.4 Regular and Core Spanners

We refer to regex formulas and vset-automata as *primitive spanner representations*. If \( O \) is a finite set of the algebraic operators defined above and \( C \) is a class of primitive spanner representations, then by \([C^O]\) we denote the class of spanners that can be constructed by a (repeated) combination of the symbols for the operators from \( O \) with spanners from \( C \).

The class of *regular spanners* is then defined to be \([\text{RGX}^{(\pi, \cup, \bowtie)}]\). An equivalent definition of regular spanners is \([\text{VA}]\) (i.e., by means of vset-automata). Note that as the class of regular spanners is closed under difference [27], we have an additional equivalent definition of this class: \([\text{RGX}^{(\pi, \cup, \bowtie, \bowtie)}]\).

The class of *core spanners* is obtained by adding the string equality selection to the positive algebra, more precisely it is the class \([\text{RGX}^{(\pi, \bowtie)}]\). Contrary to the class of regular spanners, the class of core spanners is not closed under difference [27]. That is: \([\text{RGX}^{(\pi, \bowtie)}] \subset \text{RGX}^{(\pi, \bowtie, \cup, \bowtie)}\). We refer to \([\text{RGX}^{(\pi, \bowtie, \cup, \bowtie)}]\) as the class of *generalized core spanners*.  

20
Chapter 3

Expressiveness and Descriptive Complexity

Recall that regular spanners are expressed as the closure of regex formulas under relational operators – projection, union, and join. Equally expressive formalisms include non-recursive Datalog over regex formulas [28]. In this chapter, we explore the expressive power of recursive Datalog over regex formulas, which we call recursive RGXlog, and compare it to known formalisms such as core spanners (that are defined similarly to regular spanners but also allow the string-equality selection predicate on spans) and generalized core spanners (that extend core spanners with the difference operator). We show that recursive RGXlog expresses precisely the document spanners computable in polynomial time. Interestingly, our construction shows that to express polynomial time it suffices to use binary RGXlog programs, that is, those that are built upon regex formulas with only two variables. Nevertheless, Monadic RGXlog programs (that use only unary regex formulas) are strictly less expressive than their binary counterparts. In the appendix, we extend our study to a recently proposed framework that generalizes both the relational model and document spanners [57].

The chapter presents joint work with Balder ten Cate, Ronald Fagin and Benny Kimelfeld. The results were published and presented in the International Conference on Database Theory 2019 [66]. The chapter is organized as follows. In Section 3.1 we introduce RGXlog. In Section 3.2 we compare RGXlog with (generalized) core spanners and in Section 3.3 we prove the main theorem of Section 3.2. Our main result (equivalence to polynomial time) is proved in Section 3.4 in which we also discuss Monadic programs. Finally, we conclude in Section 3.5.

3.1 RGXlog: Datalog over Regex Formulas

We start with defining the spanner language RGXlog, pronounced “regex-log,” that generalizes regex formulas to (possibly recursive) Datalog programs.

We use the terminology and notation of ordinary relational databases, with the exception that database values are all spans. (In Section A.1 we allow more general values in the database.) Formally, a relation symbol $R$ has an associated arity that we denote by $\text{arity}(R)$, and a span relation over $R$ is a finite set of
tuples \( t \in \text{Spans}^{\text{arity}(R)} \) over \( R \). We denote the \( i \)th element of a tuple \( t \) by \( t_i \). A (relational) signature \( \mathcal{R} \) is a finite set \( \{R_1, \ldots, R_n\} \) of relation symbols. A span database \( D \) over a signature \( \mathcal{R} := \{R_1, \ldots, R_n\} \) consists of span relations \( \mathcal{R}^D \) over the \( R_i \). We call \( \mathcal{R}^D \) the instantiation of \( R_i \) by \( D \). Let \( \mathcal{R} \) be a signature. By an atom over \( \mathcal{R} \) we refer to an expression of the form \( R(x_1, \ldots, x_k) \) where \( R \in \mathcal{R} \) is a \( k \)-ary relation symbol and each \( x_i \) is a variable in \( \text{Vars} \). Note that a variable can occur more than once in an atom (i.e., we may have \( x_i = x_j \) for some \( i \) and \( j \) with \( i \neq j \)), and we do not allow constants in atoms. A RGXlog program is a triple \( (\mathcal{I}, \Phi, \text{OUT}(x)) \) where:

- \( \mathcal{I} \) is a signature referred to as the IDB signature;
- \( \Phi \) is a finite set of rules of the form \( \varphi \leftarrow \psi_1, \ldots, \psi_m \), where \( \varphi \) is an atom over \( \mathcal{I} \), and each \( \psi_i \) is either an atom over \( \mathcal{I} \) or a regex formula;
- \( \text{OUT} \in \mathcal{I} \) is a designated output relation symbol;
- \( x \) is a sequence of \( k \) distinct variables in \( \text{Vars} \), where \( k \) is the arity of \( \text{OUT} \).

If \( \rho \) is the rule \( \varphi \leftarrow \psi_1, \ldots, \psi_m \), then we call \( \varphi \) the head of \( \rho \) and \( \psi_1, \ldots, \psi_m \) the body of \( \rho \). Each variable in \( \varphi \) is called a head variable of \( \rho \). We make the standard assumption that each head variable of a rule occurs at least once in the body of the rule.

We now define the semantics of evaluating a RGXlog program over a document. Let \( Q = (\mathcal{I}, \Phi, \text{OUT}(x)) \) be a RGXlog program, and let \( d \) be a document. We evaluate \( Q \) on \( d \) using the usual fixpoint semantics of Datalog, while viewing the regex formulas as extensional-database (EDB) relations. More formally, we view a regex formula \( \alpha \) as a logical assertion over assignments to \( \text{Vars}(\alpha) \), stating that the assignment forms a tuple in \([\alpha](d)\). The span database with signature \( \mathcal{I} \) that results from applying \( Q \) to \( d \) is denoted by \( Q(d) \), and it is the minimal span database that satisfies all rules, when viewing each left arrow (\( \leftarrow \)) as a logical implication with all variables being universally quantified.

Next, we define the semantics of RGXlog as a spanner language. Let \( Q = (\mathcal{I}, \Phi, \text{OUT}(x)) \) be a RGXlog program. As a spanner, the program \( Q \) constructs the span database \( D = Q(d) \) and emits the relation \( \text{OUT}^D \) as assignments to \( x \). More precisely, suppose that \( x = x_1, \ldots, x_k \). The spanner \( P = [Q] \) is defined as follows.

- \( \text{Vars}(P) := \{x_1, \ldots, x_k\} \).
- Given \( d \) and \( D = Q(d) \), the set \( P(d) \) consists of all \( (\text{Vars}(P), d) \)-records \( r_a \) obtained from tuples \( a = (a_1, \ldots, a_k) \in \text{OUT}^D \) by setting \( r_a(x_i) = a_i \).

Finally, recursive and non-recursive RGXlog programs are defined similarly to ordinary Datalog (e.g., using the acyclicity of the dependency graph over the IDB predicates).

In the next example as well as throughout this chapter, we use the following abbreviations when we define regex formulas. We use \( \cdot \) instead of \( \bigwedge_{\sigma \in \Sigma} \sigma \) (e.g., we use \( \cdot \cdot \) instead of \( \bigwedge_{\sigma \in \Sigma} \sigma \)). For convenience, we put regex formulas
We assume our alphabet consists of lower and upper case letters from the English alphabet (i.e., $a, \ldots, z$ and $A, \ldots, Z$), the comma symbol “,” and the symbol ‘ ’ that stands for whitespace. We then define the following regex formula:

$$\gamma_{\text{prnt}}(x, y) := \{a - Z(a - z)^*\} \cup \text{son of } x\{a - Z(a - z)^*\} ,$$

It holds that $\gamma_{\text{prnt}}(x, y)$ extracts spans separated by $\cup$ (where $\text{prnt}$ stands for “parent”). Applying $[\gamma_{\text{prnt}}(x, y)]$ to the document $d$ of Figure 3.1 that describes family ancestry results in a $(\{x, y\}, d)$-relation that contains the $(\{x, y\}, d)$-records $r_1, r_2$ and $r_3$ that are defined by:

- $r_1(x) = [14, 18], r_1(y) = [2, 6]$;
- $r_2(x) = [32, 36], r_2(y) = [20, 24]$; and
- $r_3(x) = [51, 55], r_3(y) = [38, 43]$.

In this and in the following examples of programs, we use the cursor sign $\uparrow$ to indicate where a rule begins. Importantly, for brevity, we use the following convention: $\text{Out}(x)$ is always the left-hand side of the last rule. Consider the following recursive RGXlog program:

$$\Downarrow \text{Ancstr}(x, z) \leftarrow \gamma_{\text{prnt}}(x, z)$$
$$\Downarrow \text{Ancstr}(x, y) \leftarrow \text{Ancstr}(x, z), \gamma_{\text{prnt}}(z, y)$$

By our convention, $\text{Out}(x)$ is $\text{Ancstr}(x, y)$. This program returns the transitive closure of the relation obtained by applying the regex formula $\gamma_{\text{prnt}}(x, z)$.

### 3.2 Comparison to Core Spanners

We begin the exploration of the expressive power of RGXlog by a comparison to the class of core spanners and the class of generalized core spanners. We first recall the following observation by Fagin et al. [28] for later reference.

**Proposition 3.2.1.** [28] The class of spanners definable by non-recursive RGXlog is precisely the class of regular spanners, namely $[\text{RGX}^{(\cup, \pi, \text{E})}]$. 

---

Technion - Computer Science Department - Ph.D. Thesis PHD-2019-10 - 2019
In addition to $\text{RGXlog}$ being able to express union, projection and natural join, the following program shows that $\text{RGXlog}$ can express the string-equality selection predicate, namely $\zeta^\ast$.

- $\text{STREq}(x, y) \leftarrow \langle x\{\epsilon\}, y\{\epsilon\} \rangle$
- $\text{STREq}(x, y) \leftarrow \langle x\{\sigma\tilde{x}\{\ast\}\}, y\{\sigma\tilde{y}\{\ast\}\}, \text{STREq}(\tilde{x}, \tilde{y}) \rangle$

Here, the second rule is repeated for every alphabet letter $\sigma$. (Note that we are using the assumption that the alphabet is finite.) It thus follows that every core spanner is definable in $\text{RGXlog}$. The other direction is false. As an example, no core spanner extracts all spans $x$ and $y$ such that $s_x$ is not a substring of $s_y$ [27], or all pairs $x$ and $y$ of spans having the same length [28]. In the following example, we construct a $\text{RGXlog}$ program that extracts both of these relationships.

**Example 3.2.1.** In the following program, rules that involve $\sigma$ and $\tau$ are repeated for all alphabet letters $\sigma$ and $\tau$ such that $\sigma \neq \tau$, and the ones that involve only $\sigma$ are repeated for every $\sigma$.

- $\text{LEN}_\equiv(x, y) \leftarrow \langle x\{\epsilon\}, y\{\epsilon\} \rangle$
- $\text{LEN}_\equiv(x, y) \leftarrow \langle x\{\tilde{x}\{\ast\}\}, y\{\tilde{y}\{\ast\}\}, \text{LEN}_\equiv(\tilde{x}, \tilde{y}) \rangle$
- $\text{LEN}_>(x, y) \leftarrow \langle x\{\tilde{x}\{\ast\}\}, y\{\tilde{y}\{\ast\}\}, \text{LEN}_=(\tilde{y}, y) \rangle$
- $\text{NOTPrfx}(x, y) \leftarrow \langle x\{\ast\}, y\{\ast\} \rangle$
- $\text{NOTPrfx}(x, y) \leftarrow \langle x\{\tilde{x}\{\ast\}\}, y\{\tilde{y}\{\ast\}\}, \text{NOTPrfx}(\tilde{x}, \tilde{y}) \rangle$
- $\text{NOTSubstr}(x, y) \leftarrow \text{LEN}_(x, y)$
- $\text{NOTSubstr}(x, y) \leftarrow \text{NOTPrfx}(x, y), \langle y\{\tilde{y}\{\ast\}\}\rangle, \text{NOTSubstr}(x, \tilde{y})$

The program defines the following relations:

- $\text{LEN}_\equiv(x, y)$ contains all spans $x$ and $y$ of the same length.
- $\text{LEN}_>(x, y)$ contains all spans $x$ and $y$ such that $x$ is longer than $y$.
- $\text{NOTPrfx}(x, y)$ contains all spans $x$ and $y$ such that $d_x$ is not a prefix of $d_y$. The rules state that $d_x$ is not a prefix of $d_y$ if $d_x$ is nonempty but $d_y$ is empty, or the two begin with different letters, or the two begin with the same letter but the rest of $d_x$ is not a prefix of the rest of $d_y$.
- $\text{NOTSubstr}(x, y)$ contains all spans $x$ and $y$ such that $d_x$ is not a substring of $d_y$. The rules state that this is the case if $x$ is longer than $y$, or both of the following hold: $d_x$ is not a prefix of $d_y$, and $d_x$ is not a substring of the suffix following the first symbol of $d_y$.

In particular, the program defines both equal-length and non-substring relationships.

The impossibility proofs of Fagin et al. [27,28] are based on the core simplification lemma [27], which states that every core spanner can be represented as a regular spanner, followed by a sequence of string-equality selections ($\zeta^\ast$) and projections ($\pi$). In turn, the proof of this lemma relies on the absence of the
difference operator in the algebra. See Freydenberger and Holldack for an indication of why a result similar to the core simplification lemma is not likely to hold in the presence of difference. Do things change when we consider generalized core spanners, where the difference is allowed? To be precise, we are interested in two questions:

1. Can \text{RGXlog} express every generalized core spanner?

2. Is it true that every spanner definable in \text{RGXlog} is a generalized core spanner?

In Section 3.4 we show that the answer to the first question is positive. In the remainder of this section, we show that the answer to the second question is negative and in the next section, we complete the proof.

We begin by constructing the following \text{RGXlog} program, which defines a Boolean is spanner that returns \text{true} if and only if the length of the input \(d\) is a power of two. (Note that we use the relation \(\text{LEN} = (x_1, x_2)\) that was defined in Example 3.2.1)

\[
\begin{align*}
\text{Pow2}(x) &\leftarrow \langle x\{\} \rangle \\
\text{Pow2}(x) &\leftarrow \langle x\{.\}x_2\{.\}\rangle, \text{Pow2}(x_1), \text{LEN} = (x_1, x_2) \\
\text{OUT}(\cdot) &\leftarrow [x\{.\}], \text{Pow2}(x)
\end{align*}
\]

In the sequel (Theorem 3.2.3) we characterize the expressiveness of Boolean generalized core spanners and conclude the following direct consequence.

**Corollary 3.2.2.** There is no Boolean generalized core spanner that determines whether the length of the input document is a power of two.

We now give the formal setup for Theorem 3.2.3. Let \(a\) be a letter, and \(L_a\) the language of all documents \(d\) that consist of \(2^n\) occurrences of \(a\) for \(n \geq 0\), that is: \(L_a \text{ def } = \{d \in a^* \mid |d| \text{ is a power of } 2\}\). We will restrict our discussion to generalized core spanners that accept only strings in \(a^*\), and show that no such spanner recognizes \(L_a\). This is enough since every generalized core spanner \(S\) can be restricted into \(a^*\) by joining \(S\) with the regex formula \(a^*\). For simplicity, we will further assume that our alphabet consists of only the symbol \(a\). Then, a language \(L\) is identified by a set of natural numbers—the set of all numbers \(m\) such that \(a^m \in L\). We denote this set by \(\mathbb{N}(L)\).

Presburger Arithmetic (PA) is the first-order theory of the natural numbers with the addition (+) binary function and the constants 0 and 1. For example, the relationship \(x > y\) is expressible by the PA formula \(\exists z[x = y + z + 1]\) and by the PA formula \(x \neq y \land \exists z[x = y + z]\). As another example, the set of all even numbers \(x\) is definable by the PA formula \(\exists y[x = y + y]\). When we say that a set \(A\) of natural numbers is definable in PA we mean that there is a unary PA formula \(\varphi(x)\) such that \(A = \{x \in \mathbb{N} \mid \varphi(x)\}\). We can now characterize the expressiveness of Boolean generalized core spanners:

**Theorem 3.2.3.** A language \(L \subseteq \{a\}^*\) is recognizable by a Boolean generalized core spanner if and only if \(\mathbb{N}(L)\) is definable in PA.
Since it is well-known that being a power of two is not definable in PA [50], Corollary 3.2.2 follows directly from the above theorem.

### 3.3 Proof of Theorem 3.2.3

We begin with the “only if” direction, which is the more involved direction. This is the direction we are most interested in, since it gives us Corollary 3.2.2.

#### 3.3.1 The “Only If” Direction

Let \( \gamma \) be an expression in \( \text{RGX}\{[\,,\,\cdot,\cdot,\cdot,\cdot,\cdot]\} \). We say that \( \gamma \) is positional if for every expression \( x\{\delta\} \) that occurs in \( \gamma \) it is the case that \( \delta = \epsilon \). Therefore, all span variables are assigned empty spans, and hence represent positions in the input document \( d \). Observe that if \( \gamma \) is positional, then \( \zeta^n \) is redundant, since every two spans referenced by \( \gamma \) have the same string. To provide positional expressions with the needed expressive power, we also add the selection \( \zeta^n \) that takes as input four spans \( x_1, x_2, y_1 \) and \( y_2 \) and returns true if \( x_1 \) precedes \( x_2 \), and \( y_1 \) precedes \( y_2 \), and the string between \( x_1 \) and \( x_2 \) is equal to the string between \( y_1 \) and \( y_2 \).

Each expression \( \delta \) in \( \text{RGX}\{[\,,\,\cdot,\cdot,\cdot,\cdot,\cdot]\} \) is built out of applying operators to regex formulas: we call these regex formula components of \( \delta \). A direct consequence of the translation of regex formulas into path unions [27] shows the following.

**Lemma 3.3.1.** For every Boolean expression \( \gamma \) in \( \text{RGX}\{[\,,\,\cdot,\cdot,\cdot,\cdot,\cdot]\} \) there exists a positional Boolean expression \( \delta \) in \( \text{RGX}\{[\,,\,\cdot,\cdot,\cdot,\cdot,\cdot]\} \) such that all of the following hold.

1. Each component of \( \delta \) is of the form \([x\{\epsilon\}]*\), or \(\langle x\{\epsilon\}\alpha y\{\epsilon\}\rangle\), or \([\cdot x\{\epsilon\}]\), where \( \alpha \) is a (variable-free) regular expression.

2. \([\delta] = [\gamma]\); that is, \( \delta \) and \( \gamma \) accept the same strings.

Note that in the first condition of Lemma 3.3.1 the regex formula \([x\{\epsilon\}]*\) states that \( x \) is the first position, \([\cdot x\{\epsilon\}]\) states that \( x \) is the last position, and \(\langle x\{\epsilon\}\alpha y\{\epsilon\}\rangle\) states that the string between \( x \) and \( y \) satisfies the regular expression \( \alpha \).

By a slight abuse of notation, we view a variable \( x \) in a positional formula \( \gamma \) as a natural number that represents its location. For example, if \( x \) is assigned the span \([5,5]\) then we view \( x \) simply as 5. Our central lemma is the following.

**Lemma 3.3.2.** Let \( \delta \) be a Boolean positional expression as in Lemma 3.3.1. It is the case that \( \delta \) is equivalent to a disjunction of Boolean formulas of the form \( \omega(x_1, \ldots, x_n) \land \varphi(z_0, \ldots, z_n) \) where:

1. \( \omega(x_1, \ldots, x_n) \) specifies a total order \( x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_n} \) over \( x_1, \ldots, x_n \) (viewed as numeric positions).

2. \( \varphi(z_0, \ldots, z_n) \) is a PA formula, where each \( z_i \) represents the length of the \( i \)th segment (among the \( n+1 \) segments) of the input document \( d \) as defined in \( \omega(x_1, \ldots, x_n) \).
Note that the \( n+1 \) segments of \( d \) defined in \( \omega(x_1, \ldots, x_n) \) are the following spans:

\[
[1, x_{i_1} + 1), [x_{i_1} + 1, x_{i_2} + 1), \ldots, [x_{i_{n-1}} + 1, x_{i_n} + 1), [x_{i_n} + 1, |d| + 1).
\]

**Proof.** We prove the lemma by induction on the structure of \( \delta \).

**Induction Base.** We first handle the case where \( \delta \) is atomic, that is, one of the three forms of components in part (1) of Lemma 3.3.1.

- If \( \delta = [x\{\epsilon\}.*] \), then it is equivalent to \( \text{true} \land z_0 = 0 \).
- If \( \delta = \langle x\{\epsilon\} \alpha \{\epsilon\} \rangle \), then it is equivalent to \( (x \leq y) \land \varphi(z_1) \) where \( \varphi(z_1) \) is the PA formula stating that \( z_1 \) is a length of a string in \( a^* \) satisfying \( \alpha \). It is known that such \( \varphi \) exists, since \( \alpha \) is a regular expression [40, 52].
- If \( \delta = [\cdot \{\epsilon\} \cdot] \), then it is equivalent to \( \text{true} \land z_1 = 0 \).

**Induction Step.** Next, we consider algebraic expressions and use the induction hypothesis. We assume that \( \delta(x_1, \ldots, x_n) \) is equivalent to

\[
\bigvee_{i=1}^{k} \omega_i(x_1, \ldots, x_n) \land \varphi_i(z_{i,0}, \ldots, z_{i,n})
\]

and that \( \delta'(x'_1, \ldots, x'_l) \) is equivalent to

\[
\bigvee_{i=1}^{k'} \omega'_i(x'_1, \ldots, x'_l) \land \varphi'_i(z'_{i,0}, \ldots, z'_{i,l})
\]

We have the following operators:

- For \( \delta_1 \cup \delta_2 \), we assume union compatibility, which means that \( \{x_1, \ldots, x_n\} = \{x'_1, \ldots, x'_l\} \). Hence, we simply take the disjunction of the two disjunctions.
- For \( \pi_{y_1, \ldots, y_q}(\delta) \), we replace each \( \omega_i(x_1, \ldots, x_n) \) with \( \omega_i(y_1, \ldots, y_q) \) by simply restricting the total order to \( y_1, \ldots, y_q \). We also replace each \( \varphi_i(z_{i,0}, \ldots, z_{i,n}) \) with the PA formula

\[
\exists z_0^i, \ldots, z_q^i \left[ \varphi_i(z_{i,0}, \ldots, z_{i,n}) \land \xi(z_0^i, \ldots, z_q^i, z_{i,0}, \ldots, z_{i,n}) \right]
\]

where \( \xi(z_0^i, \ldots, z_q^i, z_{i,0}, \ldots, z_{i,n}) \) states the relationships between the lengths \( z_0^i, \ldots, z_q^i, z_{i,0}, \ldots, z_{i,n} \), stating that each \( z_j^i \) is the sum of some of variables from \( z_{i,0}, \ldots, z_{i,n} \). For example, if \( \omega_i(x_1, \ldots, x_n) \) is \( x_1 \leq x_2 \leq x_3 \) and the operation is \( \pi_{x_1,x_3}\delta \), then \( \xi \) will be

\[
z_0^i = z_{i,0} \land z_1^i = z_{i,1} + z_{i,2} \land z_2^i = z_{i,3}.
\]
- For \( \zeta_{(x_1,x_2,x_3,x_4)}(\delta) \), we replace each \( \varphi_i(z_{i,0}, \ldots, z_{i,n}) \) with the conjunction

\[
\varphi_i(z_{i,0}, \ldots, z_{i,n}) \land \xi(z_{i,0}, \ldots, z_{i,n})
\]

where \( \xi(z_{i,0}, \ldots, z_{i,n}) \) states the equality on the sum of corresponding segments expressed by the selection condition \( x_2 - x_1 = x_4 - x_3 \).
For \( \delta \cong \delta' \), we transform the join of each pair of disjuncts (one from \( \delta \) and one from \( \delta' \)) separately. To represent the join of \( \omega_i(x_1, \ldots, x_n) \land \varphi_i(z_{i,0}, \ldots, z_{i,n}) \) and \( \omega_j'(x'_1, \ldots, x'_k) \land \varphi_j'(z'_{j,0}, \ldots, z'_{j,k}) \) we take the disjunction over all total orders \( \omega(x_1, \ldots, x_n, x'_1, \ldots, x'_k) \)

obtained by interpolating the two total orders (hence, preserving each order separately). For each such interpolated order, we represent the conjunction

\[
\varphi_i(z_{i,0}, \ldots, z_{i,n}) \land \varphi_j'(z'_{j,0}, \ldots, z'_{j,k})
\]

by replacing each segment variable \( z \) with the corresponding sum of segments from the interpolated order \( \omega(x_1, \ldots, x_n, x'_1, \ldots, x'_k) \).

• For the difference (\( \setminus \)) we combine the previous (\( \cong \)) with \( \neg \delta \). For \( \neg \delta \), we take the disjunction over all total orders \( \omega(x_1, \ldots, x_n) \) of:

\[
\omega(x_1, \ldots, x_n) \land \bigwedge_{i=1}^{k} \psi_i(z_0, \ldots, z_n)
\]

where each \( \psi_i(z_0, \ldots, z_n) \) is defined as follows:

- If the total order \( \omega(x_1, \ldots, x_n) \) is \textit{incompatible} with the total order \( \omega_i(x_1, \ldots, x_n) \) then there is at least one segment that is empty according to one of the total orders and has a length larger than zero according to the other (e.g., \( x_1 \leq x_3 \) in \( \omega \) and \( x_3 \leq x_1 \) in \( \omega_i \)). In this case we set \( \psi_i(z_0, \ldots, z_n) \) to be a contradiction (e.g., \( \neg z_0 = z_0 \)).

- If the total order \( \omega(x_1, \ldots, x_n) \) is \textit{compatible} with the order \( \omega_i(x_1, \ldots, x_n) \), we set \( \psi_i(z_0, \ldots, z_n) \) to be \( \neg \varphi_i(z_{i,0}, \ldots, z_{i,n}) \).

This completes the induction step and hence the proof.

By applying Lemma 3.3.2 to a Boolean expression \( \delta \) we get the following.

**Lemma 3.3.3.** If \( \delta \) is a Boolean positional expression as in Lemma 3.3.1, then the language recognized by \( [\delta] \) is definable in PA.

Finally, combining Lemma 3.3.1 with Lemma 3.3.3 we conclude the “only if” direction of Theorem 3.2.3 as required.

### 3.3.2 The “If” Direction

Let \( \varphi(x) \) be a unary PA formula. For the “if” direction we need to show the existence of a Boolean generalized core spanner \( \delta \) that recognizes the language \( L \subseteq \{1\}^+ \) such that \( N(L) \) is the set of natural numbers defined by \( \varphi(x) \); that is, for all documents \( d \in a^* \) it is the case that \( [\delta](d) = \text{true} \) if and only if \( \varphi(|d|) \). This direction of Theorem 3.2.3 is simpler, due to a key result by Presburger [67] who proved that PA admits quantifier elimination (cf. [25] for a modern exposition). We make use of the following theorem.
Theorem 3.3.4. Let \( \varphi(x_1, \ldots, x_k) \) be a PA formula. There is an expression
\[ \gamma \in \operatorname{RGX}(\cup, \pi, \rho, \zeta, \land) \]
with \( \operatorname{Vars}(\gamma) = \{w_1, \ldots, w_k\} \) such that for all \( d \in a^* \), the following are equivalent for all \( (\operatorname{Vars}(\gamma), d) \)-records \( r \):

1. \( r \in \llbracket \gamma \rrbracket(d) \);
2. \( \varphi(\langle d_{w_1}(r) \rangle, \ldots, \langle d_{w_k}(r) \rangle) \).

Proof. Presburger [67] proved that every formula in PA is equivalent to a quantifier-
free formula built up from the following symbols:

- The constants 0 and 1;
- The + function;
- The binary predicate \(<\);
- The unary \( \text{divisibility} \) predicate \( \equiv_k \), for all \( k \in \mathbb{N} \), where \( \equiv_k(x) \) is interpreted as “\( x \) is divisible by \( k \).”

We first eliminate the use of complex terms at the cost of reintroducing quantifiers, but only of a particular, bounded form: by “bounded existential quantification” we mean existential quantification of the form \( \exists y[y < x \land \ldots] \), or written as \( \exists y < x \ldots \) for short, where \( x \) and \( y \) are distinct variables. It follows from Presburger [67] that every PA formula can be equivalently written as a formula built up from atomic formulas of the form \( x = 0 \), \( x = 1 \), \( x = y \), and \( x = y + z \), using the Boolean connectives and bounded existential quantification. In particular, \( x < y \) can be expressed as \( \exists z < y[x = z] \) and \( \equiv_k(x) \) can be expressed as

\[
x = 0 \lor \exists y_1 < x \exists y_2 < x \ldots \exists y_{k-1} < x \land \left( \bigwedge_{i=2}^{k-1} (y_i = y_{i-1} + y_1) \land x = y_{k-1} + y_1 \right)
\]

So, assuming that \( \varphi \) has the above structure, we continue the proof by induction. For the basis we have the following:

- For \( x_1 = 0 \) we use \( \langle w_1 \{e\} \rangle \).
- For \( x_1 = 1 \) we use \( \langle w_1 \{\} \rangle \).
- For \( x_1 = x_2 \) we use \( \zeta_{w_1, w_2}(\langle w_1 \{\} \rangle \bowtie \langle w_2 \{\} \rangle) \).
- For \( x_1 = x_2 + x_3 \) we use

\[
\pi_{w_1, w_2, w_3} \zeta_{w_{2, w_2}, w_2} \zeta_{w_{3, w_3}, w_3} \langle w_1 \{w_2 \{\} w_3 \{\} \} \bowtie \langle w_2 \{\} \bowtie \langle w_3 \{\} \rangle
\]

For the inductive step, we need to show closure under conjunction, negation, and bounded existential quantification. Let \( \varphi(x_1, \ldots, x_k) \) and \( \varphi'(x_1', \ldots, x'_k) \) be two PA formulas, and \( \gamma(w_1, \ldots, w_k) \) and \( \gamma'(w'_1, \ldots, w'_k) \) the corresponding expressions in \( \operatorname{RGX}(\cup, \pi, \rho, \zeta, \land) \). To express \( \varphi \land \varphi' \) we simply use \( \gamma \bowtie \gamma' \). To express \( \neg \gamma \) we use \( (\langle w_1 \{\} \rangle \bowtie \ldots \bowtie \langle w_k \{\} \rangle) \setminus \gamma \). Finally, for the formula \( \exists x_1 < x_2 \varphi(x_1, \ldots, x_k) \) we use \( \pi_{w_1, \ldots, w_k} \zeta_{w_{1, w_1}, w_1} \langle w_1 \{w_2 \{\} w_3 \{\} \} \bowtie \gamma \). This completes the proof. \( \square \)
From Theorem 3.3.4 we conclude the “if” direction of Theorem 3.2.3 as follows:
Let \( \varphi(x) \) be a unary PA formula. Let \( \gamma(w) \) be the corresponding expression of Theorem 3.3.4. We define a Boolean expression \( \delta \) in \( \text{RGX}^{(\cup, \pi, \partial, \xi, \psi)} \) as follows.
\[
\delta := \pi_\emptyset ([w\{.\}] \bowtie \gamma(w))
\]
Theorem 3.3.4 implies that \( \delta(d) \) is true if and only if \( \varphi(|d|) \) is true, as required.

3.4 Equivalence to Polynomial Time

While \text{RGXlog} programs output relations (which are sets of tuples), the result of evaluating a spanner on \( d \) is given as a set of \((V, d)\)-records. Therefore, to compare the expressiveness of \text{RGXlog} programs and spanners, in what follows we implicitly treat tuples as records and vice-versa as described now. We assume that there is a fixed predefined order on \( \text{Vars} \) (e.g., the lexicographic order on the names of the variables) and denote the \( i \)'th element in this order by \( v_i \). A tuple \( t \in \text{Spans}^n \) is viewed as the \( \{v_1, \ldots, v_n\} \)-record that maps each \( v_i \) to \( t_i \); a \((V, d)\)-record \( r \) is viewed as the tuple whose \( i \)'th element equals the value of \( r \) on \( v \) where \( v \) is the \( i \)'th variable of \( V \) according to the fixed predefined order on \( \text{Vars} \).

An easy consequence of existing literature \cite{2,33} is that every \text{RGXlog} program can be evaluated in polynomial time (as usual, under data complexity). Indeed, the evaluation of a \text{RGXlog} program \( P \) can be done in two steps: (1) materialize the regex atoms on the input document \( d \) and get relations over spans, and (2) evaluate \( P \) as an ordinary Datalog program over an ordinary relational database, treating the regex formulas as the names of the corresponding materialized relations. The first step can be completed in polynomial time (as we will see in the next chapter), and so can the second \cite{2}. Quite remarkably, \text{RGXlog} programs capture precisely the spanners computable in polynomial time.

**Theorem 3.4.1.** A spanner is definable in \text{RGXlog} if and only if it is computable in polynomial time.

Note that a direct consequence of this theorem is that every generalized core spanner is definable in \text{RGXlog}. Thus, the answer to the second question posed in Section 3.2 is positive. Another direct consequence is that, unlike core spanners, \text{RGXlog} is closed under difference.

3.4.1 Proof of Theorem 3.4.1

The proof of the “only if” direction is described right before the theorem. To prove the “if” direction, we need some definitions and notation.

**Definitions**

We apply ordinary Datalog programs to databases over arbitrary domains, in contrast to \text{RGXlog} programs that we apply to documents, and that involve databases over the domain of spans. Formally, we define a Datalog program as a quadruple \((E, I, \Phi, \text{OUT})\) where \( E \) and \( I \) are disjoint signatures referred to as the \( EDB \)
(input) and IDB signatures, respectively, Out is a designated output relation symbol in \( I \), and \( \Phi \) is a finite set of Datalog rules. As usual, a Datalog rule has the form \( \varphi \leftarrow \psi_1, \ldots, \psi_m \), where \( \varphi \) is an atomic formula over \( I \) and \( \psi_1, \ldots, \psi_m \) are atomic formulas over \( E \) and \( I \). We again require each variable in the head \( \varphi \) to occur in the body \( \psi_1, \ldots, \psi_m \). In this chapter, we restrict Datalog programs to ones without constants; that is, an atomic formula \( \psi_i \) is of the form \( R(x_1, \ldots, x_k) \) where \( R \) is a \( k \)-ary relation symbol and the \( x_i \) are (not necessarily distinct) variables. An input for a Datalog program \( Q \) is an instance \( D \) over \( E \) that instantiates every relation symbol of \( E \) with values from an arbitrary domain. The active domain of an instance \( D \), denoted \( \text{adom}(D) \), is the set of constants that occur in \( D \).

An ordered signature \( E \) is a signature that includes three distinguished relation symbols: a binary relation symbol \( \text{Succ} \), and two unary relation symbols \( \text{First} \) and \( \text{Last} \). An ordered instance \( D \) is an instance over an ordered signature \( E \) such that \( \text{Succ} \) is interpreted as a successor relation of some linear (total) order over \( \text{adom}(D) \), and \( \text{First} \) and \( \text{Last} \) determine the first and last elements in this linear order, respectively.

A semipositive Datalog program \( P \), or Datalog program in notation, is a Datalog program in which the EDB atoms (i.e., atoms over EDB relation symbols) can be negated. We make the safety assumption that in each rule \( \rho \), every variable that appears in the head of \( \rho \) also appear in a positive (i.e., non-negated) atom of the body of the rule. For an instance \( D \) over \( E \), we denote by \( P(D) \) the database with the signature \( I \) that results from applying \( P \) on \( D \).

A query \( Q \) over a signature \( E \) is associated with a fixed arity \( \text{arity}(Q) = k \), and it maps an input database \( D \) over \( E \) into a relation \( Q(D) \subseteq (\text{adom}(D))^k \). As usual, \( Q \) is Boolean if \( k = 0 \). We say that \( Q \) is invariant under isomorphisms if for all isomorphic databases \( D_1 \) and \( D_2 \) over \( E \), and for all isomorphisms \( \varphi : \text{adom}(D_1) \rightarrow \text{adom}(D_2) \), it is the case that \( \varphi(Q(D_1)) = Q(D_2) \).

The Proof

We now discuss the proof of the “if” direction which is based on Papadimitriou’s theorem \([60]\), stating a close connection between semipositive Datalog and polynomial time:

**Theorem 3.4.2.** \([21, 50]\) Let \( E \) be an ordered signature and let \( Q \) be a query over \( E \) such that \( Q \) is invariant under isomorphisms. Then \( Q \) is computable in polynomial time if and only if \( Q \) is computable by a Datalog program.

Our proof continues as follows. Let \( S \) be a spanner that is computable in polynomial time. We translate \( S \) into a \( \text{RGXlog} \) program \( P \) in two main steps. In the first step, we translate \( S \) into a Datalog program \( P_S \) by an application of Theorem 3.4.2. In the second step, we translate \( P_S \) into \( P \). To realize the first step of the construction, we need to encode our input document by a database.

\(^1\)Note that unlike \( \text{RGXlog} \), here there is no need to specify variables for Out. This is because a spanner evaluates to assignments of spans to variables, which we need to relate to Out, whereas a Datalog program evaluates to an entire relation, which is Out itself.
since $P_S$ operates over databases (and not over strings). To use Theorem 3.4.2, we need to make sure that this encoding is computable in polynomial time, and that it is invariant under isomorphisms, that is, the encoding allows to restore the document even if replaced by an isomorphic database. To realize the second step of the construction, we need to bridge several differences between RGXlog and Datalog. First, the former takes as input a document, and the latter a database. Second, the latter assumes an ordered signature while the former does not involve any order. Third, the former does not allow negation while in the latter EDB atoms can be negated.

For the first step of our translation, we use a standard representation (which we shall explain shortly) of a string as a logical structure and extend it with a total order on its active domain. Note that we have to make sure that the active domain contains the output domain (i.e., all spans of the input document). We define $R_{\text{ord}}$ to be an ordered signature with the unary relation symbols $R_{\sigma}$ for each $\sigma \in \Sigma$, in addition to the required $\text{Succ}$, $\text{First}$ and $\text{Last}$. Let $d = \sigma_1 \cdots \sigma_n$ be an input document. We define an instance $D_d$ over $R_{\text{ord}}$ by materializing the relations as follows.

- Each relation $R_{\sigma}$ consists of all tuples $([i, i + 1])$ such that $\sigma_i = \sigma$.
- $\text{Succ}$ consists of the pairs $([i, i'], [i, i'+1])$ and all pairs $([i, n+1], [i+1, i+1])$ whenever the involved spans are legal spans of $d$.
- $\text{First}$ and $\text{Last}$ consist of $[1, 1]$, and $[n + 1, n + 1]$, respectively.

Comment 3.4.3. Observe that we view the linear order as the lexicographic order over the spans. The only difference from the usual lexicographic order on ordered pairs $(i, j)$ in that for spans, we must have $i \leq j$. The successor relation $\text{Succ}$ is inferred from this order.

An encoding instance (or just encoding) $D$ is an instance over $R_{\text{ord}}$ that is isomorphic to $D_d$ for some document $d$. In this case, we say that $D$ encodes $d$. Note that the entries of a database encoding are not necessarily spans. Nevertheless, every encoding encodes a unique document. The following lemma is straightforward.

Lemma 3.4.4. Let $D$ be an instance over $R_{\text{ord}}$. The following hold:

1. Whether $D$ is an encoding can be determined in polynomial time.
2. If $D$ is an encoding, then there are unique document $d$ and isomorphism $\iota$ such that $D$ encodes $d$ and $\iota(D_d) = D$; moreover, both $d$ and $\iota$ are computable in polynomial time.

Let $S$ be a spanner. We define a query $Q_S$ over $R_{\text{ord}}$ as follows. If the input database $D$ is an encoding and $d$ and $\iota$ are as in Lemma 3.4.4, then $Q_S(D) = \iota([S](d))$; otherwise, $Q_S(D)$ is empty. To apply Theorem 3.4.2, we make an observation.
Observation 3.4.5. The query $Q_S$ is invariant under isomorphisms, and moreover, is computable in polynomial time whenever $S$ is computable in polynomial time.

We can now apply Theorem 3.4.2 on $Q_S$:

Lemma 3.4.6. If $S$ is computable in polynomial time, then there exists a Datalog$^\dagger$ program $P'$ over $\mathcal{R}_{\text{ord}}$ such that $P'(D) = Q_S(D)$ for every instance $D$ over $\mathcal{R}_{\text{ord}}$.

The second step of the translation simulates the Datalog$^\dagger$ program $P'$ using a RGXlog program. With RGXlog, we can construct $D_d$ from $d$ as follows:

- $R_s(x) \leftarrow \langle \{x\{\sigma}\} \rangle$
- $\text{Succ}(x_1, x_2) \leftarrow \langle x_2 \{x_1\{.*\}\} \rangle \vee [.*x_1\{.x_2\{x\}\}.*]\$
- $\text{First}(x) \leftarrow \langle [x\{.\}.*] \rangle$
- $\text{Last}(x) \leftarrow [.*[x\{.\}]]$

Indeed, if we evaluate the above RGXlog rules on a document $d$, we get exactly $D_d$. Note that rules in Datalog$^\dagger$ that do not involve negation can be viewed as RGXlog rules. However, since RGXlog do not allow negation, we need to include the negated EDBs as additional EDBs. Nevertheless, we can negate EDBs without explicit negation, because regular spanners are closed under difference and complement [27]. We therefore conclude the following lemma.

Lemma 3.4.7. If $P'$ is a Datalog$^\dagger$ program over $\mathcal{R}_{\text{ord}}$, then there exists a RGXlog program $P$ such that $P(d) = P'(D_d)$ for every document $d$.

To summarize the proof of the “if” direction of Theorem 3.4.1 let $S$ be a spanner computable in polynomial time. We defined $Q_S$ to be such that $Q_S(D_d) = [S](d)$ for all $d$. Lemma 3.4.6 implies that there exists a Datalog$^\dagger$ program $P'$ such that $P'(D_d) = Q_S(D_d)$ for all $d$. By Lemma 3.4.7, there exists a RGXlog program $P$ such that $P(d) = P'(D_d)$ for all $d$. Therefore, $P$ is the required RGXlog program for which $P(d) = [S](d)$ for all $d$.

3.4.2 RGXlog over Monadic Regex Formulas

Our proof of Theorem 3.4.1 shows that RGXlog programs over binary regex formulas (i.e., regex formulas with two variables) suffice to capture every spanner that is computable in polynomial time. Next, we show that if we allow only monadic regex formulas (i.e., regex formulas with one variable), then we strictly decrease the expressiveness. We call such programs regex-monadic RGXlog programs. We can characterize the class of spanners expressible by regex-monadic RGXlog programs, as follows.

Theorem 3.4.8. Let $S$ be a spanner. The following are equivalent:

1. $S$ is definable as a regex-monadic RGXlog program.
2. S is definable as a RGXlog program where all the rules have the form

\[ \text{OUT}(x_1, \ldots, x_k) \leftarrow \gamma_1(x_1), \ldots, \gamma_k(x_k), \gamma() \]

where each \(\gamma_i(x_i)\) is a unary regex formula and \(\gamma\) is a Boolean regex formula.

Proof. We first show the direction 1 \(\rightarrow\) 2. Suppose that S is definable by the regex-monadic RGXlog program \(P\). We can assume, without loss of generality, that every regex formula \(\gamma(x)\) in \(P\) appears only in a rule of the form \(R_i(x) \leftarrow \gamma(x)\) where \(x\) is the variable of \(\gamma\). The result of running the RGXlog program \(P\) is then the same as the result of running the ordinary Datalog program \(P'\) over an ordinary relational database where the relations in the EDB are the relations \(R_\gamma\), which are populated by the above rules. Since the relations in the EDB of \(P'\) are unary, it follows from Levy et al. [51] that \(P'\) is equivalent to a nonrecursive Datalog program \(P''\). In turn, the nonrecursive \(P''\) is equivalent to a union of conjunctive queries, and hence, we conclude that \(S\) is equivalent to a RGXlog program where all the rules have the form

\[ \text{OUT}(x_1, \ldots, x_k) \leftarrow \beta_1(y_1), \ldots, \beta_m(y_m) \]

where \(\{x_1, \ldots, x_k\} \subseteq \{y_1, \ldots, y_m\}\) and each \(\beta_i(y_i)\) is a unary regex formula.

Since the conjunction of regex formulas is a regex formula [27], we can group together regex formulas that have the same variable, and therefore we can assume that the \(y_i\)'s are unique (that is, \(y_i \neq y_j\) whenever \(i \neq j\)). If there is a variable \(y_i\) that is not in \(\{x_1, \ldots, x_k\}\), then we remove \(y_i\) from the regex formula \(\beta_i\). This removal does not affect the semantics of the rule, since \(y_i\) occurs only once. We can then compile all of the Boolean regex formulas that result from the removal of the \(y_i\) into one Boolean regex formula, and take it as our \(\gamma()\). If there are no such \(y_i\) (i.e., all body variables occur in the head), then we can define \(\gamma()\) vacuously as \([\cdot]\). Finally, to construct the formula as in the theorem, we take as \(\gamma_0(x_i)\) the atom \(\beta_i(y_j)\) where \(y_j = x_i\). This may require duplicating an atom (with no semantic impact) if a head variable occurs more than once, that is, \(x_i = x_j\) for some \(i \neq j\). For example, the rule \(\text{OUT}(x, x) \leftarrow \beta(x), \gamma()\) becomes \(\text{OUT}(x, x) \leftarrow \beta(x), \beta(x), \gamma()\), where redundancy is added to match the form of the theorem.

The direction 2 \(\rightarrow\) 1 is straightforward, since the form of Part 2 is “almost” regex-monadic. Indeed, while the regex formula \(\gamma()\) is of arity zero and not one, we can simply add a dummy variable to it, say \(x_0\). For example, if \(\gamma()\) is the regex formula \([\alpha]\) for a regular expression \(\alpha\), then we can replace \(\gamma()\) with, e.g., \(\gamma_0(x_0) = [x_0\{\} \cdot \alpha]\).

Note that in the second part of Theorem 3.4.8, the Boolean \(\gamma()\) can be omitted whenever \(k > 0\), since \(\gamma()\) can be compiled into \(\gamma_k(x_k)\). We draw the following direct consequence on Boolean programs.

**Corollary 3.4.9.** A language is accepted by a Boolean regex-monadic RGXlog program if and only if it is regular.
For non-Boolean spanners, we can use Theorem 3.4.8 to show that regex-monadic RGXlog programs are strictly less expressive than regular spanners. For instance, while the relation “the span $x$ contains the span $y$” is clearly regular, it is not expressible as a regex-monadic program. To show that, we start with the following lemma.

**Lemma 3.4.10.** Let $P$ be a regex-monadic RGXlog program. There is a constant natural number $K$ such that for all input documents $d$ there is an equivalence relation on spans, with at most $K$ equivalence classes, such that following holds. Every output record in $[P](d)$ remains an output record whenever a span is replaced with a span in the same equivalence class.

**Proof.** Assume that $P$ has the form of the second part of Theorem 3.4.8. We take as our equivalence relation the relation $x \equiv y$ stating that $x$ and $y$ are produced by the exact same set of regex formulas $\gamma_i(x_i)$ of the rules in $P$. The number of equivalence classes is then bounded by the number of sets of atoms in $P$. We then conclude the following from Theorem 3.4.8 and Lemma 3.4.10.

**Theorem 3.4.11.** The class of regex-monadic RGXlog programs is strictly less expressive than the class of regular spanners.

**Proof.** Since every program in the form of the second part of Theorem 3.4.8 is the union of joins of regex formulas, we get from known results [27] that every regex-monadic RGXlog program defines a regular spanner. To show that the expressive power is strictly smaller, we will show that containment of spans cannot be expressed by a regex-monadic RGXlog program. Formally, let $S$ be the spanner $[\gamma]$ where $\gamma$ is $\langle x\{.*y\{.*\}\}\rangle$, that is, $\gamma$ extracts all pairs $x$ and $y$ of spans such $x$ contains $y$. We will show that $S$ is not equivalent to any monadic RGXlog program.

Assume, by way of contradiction, that $S$ is equivalent to the monadic RGXlog program $P$, and let $K$ be the number in Lemma 3.4.10. We assume that $K$ is large enough so that the number of spans in a string of length $K$ is larger than $K$. We can make this assumption, since the number of spans of a document of length $n$ is $\Theta(n^2)$. Take $d$ to be a document of length $K$. Then $d$ has more than $K$ spans, so some equivalence class has at least two distinct spans $s$ and $t$ of $d$. Note that for every two distinct spans $s$ and $t$ there exists a span $u$ such that either (a) the span $u$ contains $s$ but does not contain $t$, or (b) the span $u$ contains $t$ but does not contain $s$. From Lemma 3.4.10 it follows that if $R(u, s)$ holds if and only if $R(u, t)$ also holds, hence a contradiction.

3.5 Concluding Remarks

In this chapter, we studied Datalog over regex formulas, namely RGXlog. We proved that this language expresses precisely the spanners computable in polynomial time. RGXlog is more expressive than the previously studied language of core spanners and, as we showed here, more expressive than even the language of generalized core spanners. We also observed that it takes very simple binary
regex formulas to capture the entire expressive power. Unary regex formulas, on the other hand, do not suffice: in the Boolean case, they recognize precisely the regular languages, and in the non-Boolean case, they produce a strict subset of the regular spanners.

The equivalence of the expressive power of RGXlog and polynomial-time is somewhat mysterious since we do not yet have a good understanding of how to phrase some simple polynomial-time programs naturally in RGXlog. The constructive proof simulates the corresponding polynomial-time Turing machine and does not give to understand the program. For instance, is there a natural program for computing the complement of the transitive closure of a binary relation encoded by the input? An interesting future work is to investigate this aspect by studying the complexity of translating simple formalisms, such as generalized core spanners, into RGXlog.

Related formalisms that capture polynomial time include the Range Concatenation Grammars (RCG) \[13\]. In RCG, the grammar defines derivation rules for reducing the input string into the empty string; if reduction succeeds, the string is accepted. Unlike context-free and context-sensitive grammars, RCGs have predicate names in addition to variables and terminals—this allows us to maintain connections between different parts of the input string. Another formalism that captures polynomial time is multi-head alternating automata \[49\], which are finite state machines with several cursors that can perform alternating transitions. Though related, these results do not seem to imply our results on spanners.
Chapter 4
Combined Complexity

Conjunctive Queries (CQs), and more generally Unions of CQs (UCQs), over regex formulas, are the basic queries in IE systems such as IBM’s SystemT [53] and Xlog [71]. For complexity analysis, there are advantages to yardsticks that take the regex query as input, rather than regarding it as small or fixed. While in the previous chapter we viewed the query as fixed, in this chapter we investigate the combined complexity (wherein both the query and the document are regarded as input) of querying text by CQs and UCQs on top of regex formulas. We show that the lower bounds (NP-completeness and W[1]-hardness) from the relational world also hold in our settings; moreover, hardness already holds for a single-character text. However, the upper bounds from the relational world do not carry over. Unlike the relational world, acyclic CQs, and even gamma-acyclic CQs, are hard to compute. The source of hardness is that it may be intractable to instantiate the relation defined by a regex formula, simply because it has an exponential number of tuples. Yet, we are able to establish general upper bounds. In particular, UCQs can be evaluated with polynomial delay, provided that every CQ has a bounded number of atoms (while unions and projections can be arbitrary). Furthermore, UCQ evaluation is solvable with FPT (Fixed-Parameter Tractable) delay when the parameter is the size of the UCQ.

The chapter presents joint work with Dominik D. Freydenberger and Benny Kimelfeld. The results were published and presented in the Symposium on Principles of Database Systems 2018 [35]. The chapter is organized as follows. In Section 4.1 we set the basic notation and terminology. In Section 4.2 we give our lower bounds for the evaluation problem of (U)CQs. In Section 4.4 we present upper bounds for the evaluation based on the polynomial-delay algorithm for evaluating vset-automata described in Section 4.3. In Section 4.5 we generalize our complexity results to UCQs with string equalities. In Section 4.6 we present lower bounds in the presence of the difference operator. We conclude in Section 4.7.

4.1 Formal Setup

We define (Unions of) Conjunctive Queries over regex formulas and describe the complexity measures we study.
4.1.1 (Unions of) Conjunctive Queries

We consider Conjunctive Queries (CQs) over regex formulas. Such queries are defined as the class of regular spanner representations that can be composed out of natural join and projection. Formally, a regex CQ is a regular spanner representation of the form

\[ q := \pi_Y (\alpha_1 \bowtie \ldots \bowtie \alpha_k) \]

where each \( \alpha_i \) is a regex formula. For clarity of notation, if \( Y = \{y_1, \ldots, y_m\} \), we sometimes write \( q(y_1, \ldots, y_m) \) when using \( q \) to make \( \text{Vars}(q) \) more explicit. Each regex formula \( \alpha_i \) is called a regex atom. We denote by \( \text{atoms}(q) \) the set of (regex) atoms of \( q \).

In the traditional relational model (see, e.g., Abiteboul et al. [2]), a CQ is phrased over a collection of relation symbols (called signature), each having a predefined arity. Formally, a relational CQ is an expression of the form

\[ Q(y_1, \ldots, y_m) := \varphi_1, \ldots, \varphi_k \]

where each \( y_i \) is a variable in \( \text{Vars} \) and each \( \varphi_i \) is an atomic relational formula (or simply atom), that is, an expression of the form \( R(x_1, \ldots, x_m) \) where \( R \) is an \( m \)-ary relation symbol and each \( x_j \) is a variable. Similarly to regex CQs, we denote by \( \text{atoms}(Q) \) the set of atoms of \( Q \), and by \( \text{Vars}(\gamma) \) the set of variables that occur in an atom \( \gamma \).

Let \( q(y_1, \ldots, y_m) \) be a regex CQ and let \( Q(y_1, \ldots, y_m) \) be a relational CQ. We say that \( q \) maps to \( Q \) if all of the following hold:

- No relation symbol occurs more than once in \( Q \) (i.e., \( Q \) has no self-joins).
- There is a bijection \( \mu : \text{atoms}(q) \rightarrow \text{atoms}(Q) \) that preserves the sets of variables, that is, for each \( \gamma \in \text{atoms}(q) \) we have \( \text{Vars}(\gamma) = \text{Vars}(\mu(\gamma)) \).

**Example 4.1.1.** The regex CQ \( q(x, z) := \pi_{x,z}(\alpha(x, y) \bowtie \beta(y, z)) \) where \( \alpha \) and \( \beta \) are regex formulas, each with two capture variables maps to the relational CQ \( Q(x, z) := R(x, y), S(y, z) \) but does not map to \( T(x, z) := R(x, y), R(y, z) \).

We now present the notion of acyclicity of relational CQs as defined by Fagin [26]. A hypergraph \( \mathcal{H} = (\mathcal{V}, \mathcal{E}) \) is a set \( \mathcal{V} \) of vertices and a set \( \mathcal{E} \) of non-empty subsets of \( \mathcal{V} \) called hyperedges. A join tree \( T_\mathcal{H} \) of \( \mathcal{H} \) is a tree whose nodes are the hyperedges of \( \mathcal{H} \), and the running intersection property holds, namely: for all \( u \in \mathcal{V} \) the set of tree nodes that contain \( u \), namely \( \{e \in \mathcal{E} \mid t \in e\} \), forms a connected subtree of \( T_\mathcal{H} \). A hypergraph \( \mathcal{H} \) is acyclic if there exists a join tree for \( \mathcal{H} \). A hypergraph \( \mathcal{H} \) is said to be gamma acyclic if applying the following operations repeatedly on \( \mathcal{H} \), in any order, until none can be applied results in the empty set of edges:

(a) if a node is isolated (that is, if it belongs to precisely one edge), then delete that node;

38
(b) if an edge is a singleton (that is, if it contains exactly one node), then delete that edge (but do not delete the node from other edges that might contain it);

(c) if an edge is empty, then delete it;

(d) if two edges contain precisely the same nodes, then delete one of these edges;

(e) if two nodes are edge-equivalent, then delete one of them from every edge that contains it. (We say that two nodes are edge-equivalent if they are in precisely the same edges.)

Example 4.1.2. The hyperedge $H = (V, E)$ for which $V = \{A, B, C, D, E, F\}$ and $E$ consists of the hyperedges $\{B, C, D, E, F\}$, $\{A, B, C, D\}$, $\{C\}$, $\{C, D\}$ and $\{E, F\}$.

Node $A$ is isolated, and hyperedge $\{C\}$ is a singleton, so both are deleted, by rules (a) and (b). We are left with the hyperedges $\{B, C, D, E, F\}$, $\{B, C, D\}$, $\{C, D\}$ and $\{E, F\}$.

Nodes $E$ and $F$ are edge-equivalent, and so, by rule (e), we delete $F$ from both edges that contain it. Similarly, nodes $C$ and $D$ are edge-equivalent, and so we delete $D$ from all three edges that contain it. We are left with $\{B, C, E\}$, $\{B, C\}$, $\{C\}$ and $\{E\}$.

The third and fourth edges above are singletons, and so they are eliminated. This leaves $\{B, C, E\}$ and $\{B, C\}$. Node $E$ is isolated; after it is deleted, we are left with $\{B, C\}$. Both nodes are now isolated, and so they are deleted. We are left with a single empty edge, which is deleted by rule (c). The end result is the empty set of edges, and so the original hypergraph is $\gamma$-acyclic.

We associate a CQ $Q$ with a hypergraph $H_Q$ whose vertices are the variables of $Q$, and every hyperedge is a set of variables occurring in a single atom of $Q$. A CQ $Q$ is said to be acyclic (or alpha-acyclic) if $H_Q$ is acyclic. A CQ $Q$ is said to be gamma-acyclic if $H_Q$ is gamma-acyclic.

Let $q$ be a regex CQ. We say that $q$ is acyclic if it maps to an acyclic (or alpha-acyclic) relational CQ and gamma-acyclic if it maps to a gamma-acyclic relational CQ. (For definitions of acyclicity see, e.g., Abiteboul et al. \cite{abiteboul1995foundations} or Fagin \cite{fagin1980relational}.) Recall that gamma-acyclicity is strictly more restricted than acyclicity (that is, every gamma-acyclic CQ is acyclic, and there are acyclic CQs that are not gamma-acyclic).

A Union of regex CQs, or regex UCQ for short, is a regular spanner $q$ of the form $q := \bigcup_{i=1}^{k} q_i$ where each $q_i$ is a regex CQ. (Recall that, by definition, $\text{Vars}(q_i) = \text{Vars}(q_j)$ must hold.) The following proposition follows quite directly from the results of Fagin et al. \cite{fagin2008relational} and was discussed in the previous chapter.

Proposition 4.1.1. The class of spanners expressible as regex UCQs is that of the regular spanners.
In the relational world, a relational UCQ is a query of the form \( \bigcup_{i=1}^{k} Q_i \), where each \( Q_i \) is a relational CQ. Given a regex UCQ

\[
q := \bigcup_{i=1}^{k} q_i(y_1, \ldots, y_m)
\]

and a relational UCQ

\[
Q := \bigcup_{i=1}^{l} Q_i(y_1, \ldots, y_m),
\]

we say that \( q \) maps to \( Q \) if \( k = l \) and each \( q_i \) maps to \( Q_i \).

A family \( \mathbf{P} \) of regex UCQs maps to a family \( \mathbf{Q} \) of relational UCQs if each UCQ in \( \mathbf{P} \) maps to one or more CQ in \( \mathbf{Q} \).

### 4.1.2 Complexity Measures

Under the measure of data complexity, where the UCQ \( q \) at hand is assumed fixed (and the document \( d \) is given as input), query evaluation can be done in polynomial time (as discussed in the previous chapter). Hence, our main measure of complexity is that of combined complexity where both \( q \) and \( d \) are given as input.

The task of evaluating a regex query \( q \) over a document \( d \) requires the solver algorithm to produce all tuples in \( J^\alpha_K(d) \) for each regex formula \( \alpha \) in \( q \). In the worst case there could be exponentially many tuples, and so polynomial time is not a proper yardstick of efficiency. For such problems, Johnson, Papadimitriou and Yannakakis [48] introduced several complexity guarantees, which we recall here. An enumeration problem \( \mathbf{E} \) is a collection of pairs \((x, Y)\) where \( x \) is an input and \( Y \) is a finite set of answers for \( x \), denoted by \( \mathbf{E}(x) \). In our case, \( x \) has the form \((q, d)\) and \( \mathbf{E}(x) = [q](d) \). A solver for an enumeration problem \( \mathbf{E} \) is an algorithm that, when given an input \( x \), produces a sequence of answers such that every answer in \( \mathbf{E}(x) \) is printed precisely once. We say that a solver \( S \) for an enumeration problem \( \mathbf{E} \) runs in polynomial total time if the total execution time of \( S \) is polynomial in \(|x| + |\mathbf{E}(x)|\); and in polynomial delay if the time between every two consecutive answers produced is polynomial in \(|x|\).

We also consider parameterized complexity [23] for various parameters determined by \( q \). Formally, a parameterized problem is a decision problem where the input consists of a pair \((x, k)\) where \( x \) is an ordinary input and \( k \) is a parameter (typically small, which relates to a property of \( x \)). Such a problem is Fixed-Parameter Tractable (FPT) if there is a polynomial \( p \), a computable function \( f \) and a solver \( S \), such that \( S \) terminates in time \( f(k) \cdot p(|x|) \) on input \((x, k)\). We similarly define FPT-delay for a parameterized enumeration algorithm: the delay between every two consecutive answers is bounded by \( f(k) \cdot p(|x|) \). A standard lower bound is W[1]-hardness, and the standard complexity assumption is that a W[1]-hard problem is not FPT [31].

Whenever we give an upper bound, it applies to general UCQs, and whenever we give a lower bound, it applies to Boolean CQs. When we give asymptotic running times, we assume the unit-cost RAM-model, where the size of each machine
word is logarithmic in the size of the input. Regarding our alphabet \( \Sigma \), our lower bounds and asymptotic upper bounds assume that it is fixed with at least two characters; our “polynomial” upper bounds hold even if \( \Sigma \) is given as part of the input.

### 4.2 Lower Bounds of UCQ Evaluation

In this section we give our main lower bounds for the evaluation of regex UCQs. Recall that Boolean CQ evaluation is \( \text{NP} \)-complete \cite{16}. This result does not extend simply to regex UCQs since relations are not given directly as input but rather extracted from an input string using regex formulas. However, quite unexpectedly, the evaluation of Boolean regex CQs remains \( \text{NP} \)-complete. What is less expected is that this holds even for documents that consist of a single fixed character. That is, we show that the evaluation of Boolean regex CQs is \( \text{NP} \)-complete also under the measure of query complexity, where the UCQ \( q \) is given as input and \( d \) is regarded as fixed.

**Theorem 4.2.1.** The evaluation of Boolean regex CQs is \( \text{NP} \)-complete, and remains \( \text{NP} \)-hard even under both of the following assumptions.

1. Each regex formula is of bounded size.
2. The document is of length one.

Note that our proof uses the fact that a document of length one has three different spans (i.e., two empty spans and a single non-empty span).

**Proof.** The upper bound is obvious (even for core spanners \cite{33}). For the lower bound, we construct a reduction from 3CNF-satisfiability (which is also known as 3SAT \cite{39}) to the evaluation problem of Boolean regex CQs. The input to 3CNF is a formula \( \psi \) with the free variables \( x_1, \ldots, x_n \) such that \( \psi \) has the form \( C_0 \land \cdots \land C_m \) where each \( C_j \) is a clause. Each clause is a conjunction of three literals from the set \( \{ x_i, \neg x_i \mid 1 \leq i \leq n \} \).

The goal is to determine whether there is an assignment \( \tau \) from \( \{ x_1, \ldots, x_n \} \) to \( \{ 0, 1 \} \) that satisfies \( \psi \). Given a 3CNF-formula \( \psi \), we construct a regex CQ \( q \) and an input document \( d \) such that there is a satisfying assignment for \( \psi \) if and only if \( [q](d) \neq \emptyset \). We define \( d := a \). To construct \( q \), we associate each variable \( x \) with a corresponding capture variable \( x \) and each assignment \( \tau \) with a regex formula that assigns \( x \) the span \([1, 1]\) if \( \tau(x) = 0 \) and \([2, 2]\) if \( \tau(x) = 1 \). For instance, given the assignment \( \tau \) such that \( \tau(x) = \tau(y) = 0 \) and \( \tau(z) = 1 \) its corresponding regex is \( x \{ y \{ \varepsilon \} \} \cdot a \cdot z \{ \varepsilon \} \). Note that since each clause \( C_j \) contains three variables, it has exactly seven satisfying assignments (out of all possible eight assignments to its variables). We denote these assignments by \( \gamma_1^j, \ldots, \gamma_7^j \) and their corresponding regex formulas by \( \gamma_1^j, \ldots, \gamma_7^j \). Next, we define \( \gamma_i \) and \( q \) as follows.

\[
\gamma_i := \bigvee_{j=1}^{7} \gamma_i^j \quad q := \pi_0 \bigwedge_{i=1}^{m} \gamma_i
\]
It is straightforward to show that if there exists a satisfying assignment \( \tau \) for \( \psi \) then there is at least one \( \mu \in \{ q \}(d) \). This \( \mu \) is obtained from \( \tau \) by defining \( \mu(x) := [1,1] \) if \( \tau(x) = 0 \) and \( \mu(x) := [2,2] \) if \( \tau(x) = 1 \). The other direction is shown analogously: If there is at least one \( \mu \in \{ q \}(d) \), then a satisfying assignment \( \tau \) for \( \psi \) is obtained by defining \( \tau(x) := 0 \) if \( \mu(x) = [1,1] \) and \( \tau(x) := 1 \) if \( \mu(x) = [2,2] \).

One might be tempted to think that the evaluation of a regex CQ over a document \( d \) can be done efficiently if the regex CQ maps to a relational CQ of a tractable class (e.g., acyclic CQs where evaluation is in polynomial total time \([84]\)), by applying what we refer to as the canonical relational evaluation:

(a) Evaluate each regex formula: \( r_i := [\alpha_i](d) \).

(b) Evaluate \( \pi_Y(r_1 \bowtie \ldots \bowtie r_k) \) (as a relational CQ).

This, however, is not true in the general case. There are two problems with the canonical relational evaluation. The first (and main) problem is that \( r_i \) may already be too large (e.g., has an exponential number of tuples). The second problem is that, even if \( r_i \) is of manageable size, it is not clear that it can be efficiently constructed. In the next section, we will show that the second problem is solvable: we can evaluate \( \alpha_i \) over \( d \) in polynomial total time. However, the first problem remains. In fact, the following theorem states that the evaluation of regex CQs is intractable, even if we restrict to ones that map to acyclic CQs, and even the more restricted gamma-acyclic CQs. In addition, we can show \( \text{W}[1] \)-hardness with respect to the number of variables or regex formulas.

**Theorem 4.2.2.** Evaluation of gamma-acyclic Boolean regex CQs is \( \text{NP} \)-complete. The problem is also \( \text{W}[1] \)-hard with respect to the number of (a) variables, and (b) atoms.

**Proof.** The \( \text{NP} \) upper bound was already discussed in the proof of Theorem 4.2.1. We prove both lower bounds at the same time by defining a polynomial time FPT-reduction from the \( k \)-clique problem. Given an undirected graph \( G := (V,E) \) and \( k \geq 2 \), this problem asks whether \( G \) contains a clique with \( k \) nodes. Let \( \Sigma := \{a,b,\triangleright,\#,#\} \). (Note that the proof can be adapted to a binary \( \Sigma \) with standard techniques.) We assume \( V = \{v_1, \ldots, v_n\} \), and associate each \( v_i \in V \) with a unique string \( v_i \in \{a,b\}^* \) such that \( |v_i| \) is \( O(\log n) \).

We define \( d := (e_{1,2} \cdots e_{1,n}) \cdot (e_{2,3} \cdots e_{2,n}) \cdots (e_{n-1,n}) \), where

\[
e_{i,j} := \begin{cases} 
\epsilon & \text{if } \{v_i, v_j\} \notin E, \\
\triangleright v_i \# v_j \triangleright & \text{if } \{v_i, v_j\} \in E.
\end{cases}
\]

for all \( 1 \leq i < j \leq n \). Thus, \( d \) encodes \( E \), such that an edge \( \{v_i, v_j \mid i < j\} \) precedes an edge \( \{v_{i'}, v_{j'} \mid i' < j'\} \) if \( i < i' \), or \( i = i' \) and \( j < j' \).

Next, we construct \( q \) such that \([q](d) \neq \emptyset\) if and only if there is a \( k \)-clique in \( G \). Note that a \( k \)-clique has \( k \) nodes \( v_{c(1)}, \ldots, v_{c(k)} \), and we assume that \( i < j \) implies \( c(i) < c(j) \). For each \( v_{c(l)} \), the query \( q \) shall contain the \((k - 1)\) variables

42
for all $1 \leq i < j \leq k$. The idea is that $x_{i,j}$ and $y_{i,j}$ respectively match $v_{c(i)}$ and $v_{c(j)}$ in $e$ in $d$, which uses the same order. We now want to ensure that for each $1 \leq l \leq k$, all $y_{i,l}$ and all $x_{i,l}$ with $1 \leq i < l < j \leq k$ have to be matched to various occurrences of the same substring. To ensure this, for each $1 \leq l \leq k$, we define the regex formula $\delta_l := \bigvee_{i=1}^{n} \delta_{l,v_i}$, where

$$\delta_{l,v_i} := \Sigma^* \# y_{1,l} \{\{v_i\}\} \# \Sigma^* \cdot \Sigma^* \# y_{l-1,l} \{\{v_i\}\} \# \Sigma^* \cdot \Sigma^* \# y_{l+1,l} \{\{v_i\}\} \# \Sigma^* \cdot \Sigma^* \# x_{l,k} \{\{v_i\}\} \# \Sigma^*.$$ 

Finally we define $q$ to be the query

$$q := \pi_{\emptyset} \left( \bigotimes_{1 \leq i \leq k-1} \delta_i \right)$$

Note that $q$ contains $O(k)$ atoms and $O(k^2)$ variables. Additionally, $q$ contains no gamma-cycles since each two different $\delta_i$ have no common variables. Moreover, as $|\gamma|$ is $O(k^2)$, and each $|\delta_i|$ is $O(kn \log n)$, it is the case that $|q|$ is $O(k^2 + k^2 n \log n) = O(k^2 n \log n)$. Furthermore, $|d|$ is $O(|E|)$. Hence, $q$ and $d$ can be constructed in polynomial time, and the construction is FPT with respect to the number of variables and atoms.

All that is left to show that the reduction is correct; that is $[q](d) \neq \emptyset$ if and only if $G$ contains a $k$-clique. Assume that $G$ contains a $k$-clique $\{v_{c(1)}, \ldots, v_{c(k)}\}$ where $c(i) < c(j)$ whenever $i < j$. Let $\mu$ be a record that is defined as follows: For all $1 \leq i < j \leq k$, it holds that $\mu$ maps the variables $x_{i,j}$ and $y_{i,j}$ to the span $s_{i,j}^1$ and $s_{i,j}^2$, respectively, where $\triangleright v_{c(i)} \# v_{c(j)} \triangleleft$ is a unique substring of $d$ and it holds that $d_{s_{i,j}^1} = v_{c(i)}$ and that $d_{s_{i,j}^2} = v_{c(j)}$ where $v_{c(i)}$ and $v_{c(j)}$ are substrings of the aforementioned substring. It holds that $\mu \in [\gamma](d)$, since each two nodes in the clique are connected and since the encoding of the edges in $d$ is ordered. Moreover, for each $l$, the restriction of $\mu$ to $\text{Vars}(\delta_l)$ is in $[\delta_l](d)$, since the strings spanned by the $y_{i,l}$ and the $x_{i,l}$ are equal, and $\mu$ respects the order of the variables in $\delta_l$.

Now assume that $\mu \in [q](d)$. We can now derive the nodes $v_{c(1)}$ to $v_{c(k)}$ of the clique directly from the variables $x_{i,j}$ and $y_{i,j}$; since for all $1 \leq i < l < j \leq k$, if $\mu \in [\delta_l](d)$ then there is a unique $c(l)$ such that $d_{\mu(y_{i,j})} = d_{\mu(x_{i,j})} = v_{c(l)}$. Furthermore, $\mu \in [\gamma](d)$ ensures that for all $i < j$, it is the case that $\mu(x_{i,j})$ and $\mu(y_{i,j})$ map to the encoding of the edge $\{v_{c(i)}, v_{c(j)}\}$ in $d$. Hence, $\{v_{c(1)}, \ldots, v_{c(k)}\}$ is a $k$-clique in $G$.

For example, consider the following graph:

```
  a -- b
  aa -- ab
```
with \( k = 3 \). Note that for simplicity the vertices are marked with their encoding strings over \( \Delta := \{a, b\} \). Thus, we have the following document

\[
d := \triangleright a \ # \ b \ # \ a \ # \ b \ # \ ab \ # \ ab \ # \ aa \ # \ ab \ # 
\]

In this case \( \gamma = \gamma_1 \gamma_2 \) where

\[
\begin{align*}
\gamma_1 &= \Sigma^* \triangleright x_1^2(\Delta^*) \ # \ x_2^1(\Delta^*) \ # \ x_3^2(\Delta^*) \ # \ x_3^3(\Delta^*) \ # \ x_3^3(\Delta^*) \ # \ x_3^3(\Delta^*) \ # \ x_3^3(\Delta^*) \ # \\
\gamma_2 &= \Sigma^* \triangleright x_2^3(\Delta^*) \ # \ x_2^3(\Delta^*) \ # \ x_2^3(\Delta^*) \ # \\
\end{align*}
\]

An assignment \( \mu \) to the variables in \( q \) assigns \( x_3^2 \) and \( x_3^3 \) the spans corresponding to the second occurrence of \( a \) and first of \( ab \), respectively.

In the sequel, we present an FPT result for the evaluation.

### 4.3 Evaluating Vset-Automata

Our upper bounds of UCQ evaluation (which are presented in Section 4.4) are based on the following algorithm that we devise for evaluating vset-automata over documents.

**Theorem 4.3.1.** Given a functional vset-automaton \( A \) with \( n \) states and \( m \) transitions, and a document \( d \), one can enumerate \([A](d)\) with polynomial delay of \( O(n^2|d|) \), following a polynomial preprocessing of \( O(n^2|d| + mn) \).

Note that Losemann [54] introduced an algorithm that evaluates vset-automata with logarithmic delay, but only if the number of variables is bounded\(^1\) Amarilli et al. [7] have improved and extended our algorithm to a broader class of vset-automata. (We revisit this result in Chapter 5.) In the remainder of this section, we discuss our algorithm and other details of the proof.

#### 4.3.1 Sequences of Variable Configurations

Before we discuss the actual algorithm, we first examine a property of functional vset-automata. Building on this, we introduce a novel way of working with vset-automata, which is a main contribution of our algorithm.

In order to introduce the concept, we consider an arbitrary functional vset-automaton \( A = (V, Q, q_0, q_f, \delta) \). We assume that \( q_f \) is reachable from \( q_0 \) since if it is not then for every \( d \) it holds that \([A](d) = \emptyset\) and the enumeration is trivial. We also ensure that all states are reachable from \( q_0 \), and that \( q_f \) is reachable from every state.

Then, for each state \( q \in Q \) and all \( x \in V \), each ref-word \( r \in (\Gamma_V \cup \Sigma)^* \) (see Section 2.2.1) that takes \( A \) from \( q_0 \) to \( q \) satisfies exactly one of these mutually exclusive conditions:

\(^1\)Otherwise, two steps of the algorithm become exponential, namely the encoding of vset-automata to another automata model and the actual enumeration. Using functional automata might make the first step polynomial, but not the second.
Example 4.3.1. Recall the functional \( V^T \)-automaton \( A_{\text{fun}} \) from Example 2.2.4.

\[
\begin{array}{c}
q_0 \xrightarrow{a} q_1 \xrightarrow{\neg x} q_f
\end{array}
\]

Then \( \vec{c}_{q_0}(x) = \mathbf{w} \), \( \vec{c}_{q_1}(x) = \mathbf{o} \), and \( \vec{c}_{q_f}(x) = \mathbf{c} \).

Before we use this for the enumeration algorithm, we consider a more general view on variable configurations that is independent of the automaton. As we shall see, the enumeration algorithm relies on the fact that for each \( d \in \Sigma^* \) \((d = \sigma_1 \cdots \sigma_N \text{ with } N \geq 0)\), each \((V, d)\)-record \( \mu \) can be interpreted as a sequence of \( N + 1 \) variable configurations \( \vec{c}_1, \ldots, \vec{c}_{N+1} \) in the following way: For \( x \in V \), assume that \( \mu(x) = [i, j] \). For \( 1 \leq l \leq N + 1 \), we define \( \vec{c}_l(x) := \mathbf{w} \) if \( l < i \), \( \vec{c}_l(x) := \mathbf{o} \) if \( i \leq l < j \), and \( \vec{c}_l(x) := \mathbf{c} \) if \( l \geq j \). The idea is that each \( \vec{c}_l \) is the variable configuration immediately before reading \( \sigma_i \).

Example 4.3.2. Let \( V := \{ x \} \), and let \( d := \mathbf{aa} \). The following table contains all possible \((V, d)\)-records, and the corresponding sequence \( \vec{c}_1(x), \vec{c}_2(x), \vec{c}_3(x) \):
The algorithm enumerates the $(V, \mathbf{d})$-records of $[A](\mathbf{d})$ by enumerating the corresponding sequences of $|\mathbf{d}| + 1$ variable configurations for $V$. In order to do so, we interpret each variable configuration as a letter of the alphabet $\mathcal{K} := \{ \tilde{c}_q \mid q \in Q \}$ (note that while there are $3^{|V|}$ possible letters that might occur in $\mathcal{K}$, its actual size is always bounded by $|Q|$). More specifically, the algorithm has the following two steps:

### 4.3.2 The Algorithm

The algorithm enumerates the $(V, \mathbf{d})$-records of $[A](\mathbf{d})$ by enumerating the corresponding sequences of $|\mathbf{d}| + 1$ variable configurations for $V$. In order to do so, we interpret each variable configuration as a letter of the alphabet $\mathcal{K} := \{ \tilde{c}_q \mid q \in Q \}$ (note that while there are $3^{|V|}$ possible letters that might occur in $\mathcal{K}$, its actual size is always bounded by $|Q|$). More specifically, the algorithm has the following two steps:

<table>
<thead>
<tr>
<th>$r$</th>
<th>$\mu^r(x)$</th>
<th>$\tilde{c}_1(x), \tilde{c}_2(x), \tilde{c}_3(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leftarrow xaa$</td>
<td>[1, 1]</td>
<td>c, c, c</td>
</tr>
<tr>
<td>$x \leftarrow a \leftarrow x a$</td>
<td>[1, 2]</td>
<td>o, c, c</td>
</tr>
<tr>
<td>$x \leftarrow aa \leftarrow x$</td>
<td>[1, 3]</td>
<td>o, o, c</td>
</tr>
<tr>
<td>$a \leftarrow x \leftarrow a a$</td>
<td>[2, 2]</td>
<td>w, c, c</td>
</tr>
<tr>
<td>$a \leftarrow a \leftarrow x$</td>
<td>[2, 3]</td>
<td>w, o, c</td>
</tr>
<tr>
<td>$aa \leftarrow x \leftarrow tw$</td>
<td>[3, 3]</td>
<td>w, w, c</td>
</tr>
</tbody>
</table>

Note that this is exactly $[A_{\text{fun}}](\mathbf{d})$, where $A_{\text{fun}}$ is the vset-automaton from Example 4.3.1.

We say that a sequence of variable configurations for $V$ is valid if it respects the order of variable states; i.e., $\tilde{c}_i(x) = c$ implies $\tilde{c}_{i+1}(x) = c$, and $\tilde{c}_i(x) = o$ implies $\tilde{c}_{i+1}(x) \in \{ o, c \}$ for all $x \in V$. Obviously, each valid sequence of $|\mathbf{d}| + 1$ variable configurations for $V$ can be interpreted as a $(V, \mathbf{d})$-record; and it is easy to see that this is a one-to-one correspondence.

To connect this point of view to the variable configurations of $A$, note that each $r \in \text{Ref}(A, \mathbf{d})$ can be written as $r = r_0 \cdot r_1 \cdot r_2 \cdots r_{N-1} \cdot r_N$, where $r_i \in \Gamma_V^*$. For $1 \leq i \leq N + 1$, we determine $\tilde{c}_i$ from $r_0 \cdot r_1 \cdots r_{i-1}$ (as in the definition of the $\tilde{c}_q$ above). This has the same effect as defining $\tilde{c}_i := \tilde{c}_{q_i}$ for any $q_i$ that can be reached by processing $r_0, r_1, \ldots, r_{i-1}$. In other words, in each run of $A$ on $r$, immediately before processing $\sigma_i$, $\sigma_i$ must be in a state $q_i$ with $\tilde{c}_i = \tilde{c}_{q_i}$. Thus, the sequence $\tilde{c}_1, \ldots, \tilde{c}_{N+1}$ corresponds to the $(V, \mathbf{d})$-record $\mu^r$.

Like ref-words, sequences of variable configurations can be understood as an abstraction of spanner behavior. In fact, both can be seen as successive steps of generalization: Ref-words hide the actual behavior of primitive spanner representations (i.e., the actual sequence of states in the vset-automaton, or which parts of the regex are mapped to which part of the input); they only express in which order variables are opened and closed. Sequences of variable configurations take this one step further, and compress successive variable operations (without terminals in-between) into a single step. Hence, treating $[A](\mathbf{d})$ as a language over the alphabet $\mathcal{V}^{|V|}$ is exactly the level of granularity that is needed to distinguish different tuples.

Note that although some proofs of Freydenberger use the concept of variable configurations for states, identifying $(V, \mathbf{d})$-records with sequences of variable configurations is a main conceptual contribution of this chapter.
1. Given $A$ and $d$, construct an NFA $A_G$ over the alphabet $\mathcal{K}$ such that $\mathcal{L}(A_G)$ contains exactly those strings $\kappa_1 \cdots \kappa_{|d|+1}$, $\kappa_i \in \mathcal{K}$, that correspond to the elements of $[A](d)$.

2. Enumerate $\mathcal{L}(A_G)$ with polynomial delay.

**Step 1: Constructing $A_G$**

The algorithm constructs the NFA $A_G$ by first constructing a graph $G$ whose nodes are tuples $(i,q)$ which encode that $A$ can be in state $q$ after reading $\sigma_1 \cdots \sigma_i$. The edges are drawn accordingly: There is an edge from $(i,p)$ to $(i+1,q)$ if $A$ can reach $q$ from $p$ by reading $\sigma_{i+1}$ and then arbitrarily many variable operations. The NFA $A_G$ is then directly obtained from $G$ by interpreting every edge from $(i,p)$ to $(i+1,q)$ as a transition for the letter $\bar{c}_q$.

Formally, we begin with the first step of computing the function $\bar{C}$ which maps each $q \in Q$ to its variable configuration $\bar{c}_q$. As $A$ is functional, all rewords that label paths which lead from $q_0$ to a state $q$ contain exactly the same symbols from $\Gamma_V$. Hence, $\bar{C}$ can be computed by executing a breadth-first-search that, whenever it proceeds from a state $p$ to a state $q$, derives $\bar{c}_q$ from $\bar{c}_p$ (if the transition opens or closes a variable $x$, $\bar{c}_p(x)$ is set to o or c, respectively; otherwise, both variable configurations are identical). As breadth-first-search is possible in $O(m+n)$, this step runs in time $O(vm+vn)$ where $v := |V|$. Note that this process can also be used to check whether $A$ is functional (this is the idea of the proof of Theorem 2.2.2 cf. [32]).

Next, we compute the $\epsilon$-closure as a function $\mathcal{E} : Q \rightarrow 2^Q$ directly from $A$. Using a standard transitive closure algorithm (see, e.g., Skiena [73]), $\mathcal{E}$ can be computed in time $O(n(n+m))$, which we can assume to be $O(mn)$ (as $v \leq n \leq m$, this dominates the complexity of computing the variable configurations).

To see how $A$ operates on variables when changing from a state $p$ to a state $q$, it suffices to compare $\bar{c}_p$ and $\bar{c}_q$. We now use this to define an $\epsilon$-NFA $A_N$ over $\Sigma$ that, when combined with $\bar{C}$, simulates $A$. Let $A_N := (Q, \delta_N, q_0, q_f)$, where $\delta_N$ is obtained from $\delta$ by defining, for all $q \in Q$, $\delta_N(q, \sigma) := \delta(q, \sigma)$ for all $\sigma \in \Sigma$, as well as

$$\delta_N(q, \epsilon) := \delta(q, \epsilon) \cup \bigcup_{a \in \Gamma_V} \delta(q, a).$$

In other words, $A_N$ is obtained from $A$ by replacing each transition that has a variable operation with an $\epsilon$-transition. Now, there is a one-to-one-correspondence between paths in $A_N$ and $A$. This also follows from the fact that $A$ is functional: In a general vset-automaton, it would be possible to have two states $p$ and $q$ such that there are two transitions with different variable operations from $p$ to $q$ (e.g. $x^+$ and $-x$, recall Example 2.2.4). But in functional automata, this is not possible (we already used this insight to define variable configurations).

Now, assume $d \in \mathcal{L}(A_N)$, and recall that $d = \sigma_1 \cdots \sigma_N$ with $N \geq 0$. Then there exists a sequence of states $q_0, q_0, \ldots, q_n, q_N \in Q$ such that

1. $\hat{q}_i \in \mathcal{E}(q_i)$ for $0 \leq i \leq N$,
2. $q_{i+1} \in \delta_N(\hat{q}_i, \sigma_{i+1})$ for $0 \leq i < N$,

3. $\hat{q}_i = q_f$.

In other words, each $q_i$ is the state that is entered immediately after reading $\sigma_i$, and $\hat{q}_i$ is the state immediately before $\sigma_{i+1}$ is read. Hence, $\hat{q}_{i+1}$ is reached from $\hat{q}_i$ solely using $\epsilon$-transitions (which correspond to variable operations or to $\epsilon$-transitions in $A$). Hence, in this point of view on runs, we only focus on the parts of an accepting run immediately before and after processing terminals. Take note that it is possible that $q_i = \hat{q}_i$, which corresponds to not operating on variables between reading $\sigma_i$ and $\sigma_{i+1}$.

We now discuss how this can be used to derive a $(V,d)$-tuple $\mu$. When introducing variable configurations, we already remarked that $\tilde{c}_{q_0}(x) = w$ and $\tilde{c}_{q_f}(x) = c$ must hold for all $x \in V$. The latter immediately implies $\tilde{c}_{\hat{q}_n} = c$. Furthermore, as terminal transitions cannot change variable configurations, $\tilde{c}_{\hat{q}_i} = \tilde{c}_{q_{i+1}}$ must hold as well. This allows us to define $\mu$ solely by considering the $\tilde{c}_{\hat{q}_i}$. As $A$ is functional, for each $x \in V$, there exist $i$ and $j$ with $0 \leq i \leq j \leq N$ such that $\tilde{c}_{q_i'} = w$ for all $i' < i$ and $\tilde{c}_{q_j'} = c$ for all $j' \geq j$. Less formally: In the variable configurations of the states $\hat{q}_0, \hat{q}_1, \ldots, \hat{q}_n$, there is first a (possibly empty) block of states where $x$ is waiting, followed by a (possibly empty) block where $x$ is open, and finally a non-empty block where $x$ is closed. (The block of open configurations is empty if and only if $x$ is opened and closed without reading a terminal letter). Hence, for every $x \in V$, there exist

1. a minimal $i$, such that $\tilde{c}_{q_i} \neq w$,

2. a minimal $j$, such that $\tilde{c}_{q_j} = c$.

We then define $\mu(x) := [i + 1, j + 1]$. The intuition behind this is as follows: The choice of $i$ means that in $A$, the operation $x^+$ occurs somewhere between $q_i$ and $\hat{q}_i$. Hence, the first position in $\mu(x)$ must belong to the next terminal, which is $\sigma_{i+1}$. Likewise, the choice of $j$ means that in $A$, the operation $\neg x$ occurs somewhere between $q_j$ and $\hat{q}_j$, which means that $\sigma_{j+1}$ must be the first letter after $\mu(x)$.

Thus, together with $\hat{C}$, every accepting run of $A_N$ for a string $d$ can be interpreted as a $\mu \in [A](d)$. As every accepting run in $A$ corresponds to an accepting run of $A_N$, in order to enumerate $[A](d)$, it suffices to enumerate all accepting runs of $A_N$. Take particular note that, instead of considering the states $\hat{q}_i$, it suffices to consider their variable configurations $\tilde{c}_{\hat{q}_i}$ in order to define $\mu$. Hence, a $(V,d)$-tuple $\mu$ can be derived from a sequence of $|d| + 1$ variable configurations. In fact, this approach also works in the reverse direction: Every $(V,d)$-tuple $\mu$ can be used to derive such a sequence, simply be examining how the variables are opened and closed with respect to the positions of $d$.

We use the NFA $A_N$ to construct a directed acyclic graph $G$ that can be interpreted as an NFA $A_G$ such that $L(A_G)$ encodes exactly the accepting runs of $A$ on $s$.

Instead of directly computing $G$, we first construct a graph $G' := (V', E')$
from $A$, where $V' := \bigcup_{i=0}^{N} V'_i$ with

$$V'_0 := \{(0, q) \mid q \in E(q_0)\},$$

$$V'_{i+1} := \{(i + 1, q) \mid q \in E(\delta_N(p, \sigma_{i+1})) \text{ for some } (i, p) \in V'_i\}$$

for all $0 \leq i < n$. In a slight abuse of notation, we restrict $V'_N$ to $V'_N \cap \{(N, q_f)\}$. Furthermore, without loss of generality, we can assume that $V'_N = \{(N, q_f)\}$, as otherwise, $[A](d) = \emptyset$. Less formally explained, $V'_i$ contains all $(i, q)$ such that $q$ can act as $q_i$ when using $A_N$ and $C$ to simulate $A$ as explained above.

Following this idea, we define $E'$ to simulate these transitions; in other words, we define $E' := \bigcup_{0 \leq i < N} E_i$, where

$$E_i := \{((i, p), (i + 1, q)) \mid (i, p) \in V'_i \text{ and } q \in E(\delta(p, \sigma_{i+1}))\}.$$ 

Observe that $|V'| \leq Nn + 1$, as $|V'_i| \leq N$ for $0 \leq i < N$. Also, as each node of some $V'_i$ with $i < N$ has at most $n$ outgoing edges, $|E'| \leq Nn^2$. Next, we obtain $G := (V, E)$ by removing from $G'$ all nodes (and associated edges) from which $(n, q_f)$ cannot be reached. Using standard reachability algorithms, this is possible in time $O(|V'| + |E'|) = O(Nn^2)$. For each $V'_i$, we define $V_i := V'_i \cap V$.

Now, every path $\pi$ from a node $(0, q)$ to $(N, q_f)$ in $G$ corresponds to an accepting run of $A_N$ (and for every accepting run of $A_N$, there is a corresponding path in $G$). Hence, if we consider a path $\pi = ((0, \hat{q}_0), (1, \hat{q}_1), \ldots, (N, \hat{q}_N))$ in $G$ (which implies $\hat{q}_N = q_N$), we can use the sequence of variable configurations $\vec{c}_{\hat{q}_0}, \vec{c}_{\hat{q}_1}, \ldots, \vec{c}_{\hat{q}_N}$ to define a $\mu^\pi \in [A](d)$.

While this allows us to enumerate all elements of $[A](d)$, two different paths $\pi_1$ and $\pi_2$ in $G$ might lead to the same sequence of variable configurations (which would imply $\mu^{\pi_1} = \mu^{\pi_2}$), which means that $G$ alone cannot be used to compute $[A](d)$ with polynomial delay.

The crucial part of the next step is interpreting the set of variable configurations of states as an alphabet $\mathcal{K} = \{\vec{c}_q \mid q \in Q\}$, and treating sequences of variable configurations as strings over $\mathcal{K}$. Then $G$ defines the language of all variable configurations that correspond to an element of $[A](d)$. By adding a starting state $q_0$, we turn $G$ into an NFA $A_G$ over $\mathcal{K}$ that accepts exactly this language. More specifically, we define $A_G := (Q_G, \delta_G, q_0, F_G)$, where

- $Q_G := V \cup \{q_0\}$,
- $F_G := V_N = \{(N, q_f)\}$,
- $\delta_G$ is defined in the following way:
  - for each $(0, q) \in V_0$, $\delta_G$ has a transition from $q_0$ to $q$ with label $\vec{c}_q$,
  - for each transition $((i, p), (i + 1, q)) \in E$, $\delta_G$ has a transition from $p$ to $q$ with label $\vec{c}_q$.

We observe that all incoming transitions of a state $(i, q)$ are labeled with the same terminal letter $\vec{c}_q$. Furthermore, although $|V|$ variables allow for $3^{|V|}$ distinct variable configurations, only at most $|Q|$ of these appear in $A_G$. 

49
Step 2: Enumerating $\mathcal{L}(A_G)$

To enumerate $\mathcal{L}(A_G)$ without repetitions, we tailor an optimized version of the algorithm by Ackerman and Shallit that, given $l \geq 0$ and an NFA $M$ over some alphabet $T$, enumerates $\mathcal{L}(M) \cap T^l$.

Before we proceed to the actual enumeration algorithm, we introduce another definition. We fix a total order $<_K$ on $K$, and extend this to the radix order $<_r$ on $K^{N+1}$ as follows: Given $u, v \in K^{N+1}$, we define $u <_r v$ if there exist $p, d_u, d_v \in K^*$ and $\kappa_u, \kappa_v \in K$ with $u = p \cdot \kappa_u \cdot d_u$, $v = p \cdot \kappa_v \cdot d_v$, and $\kappa_u <_K \kappa_v$.

As all strings in $\mathcal{L}(A_G)$ have length $N + 1$, we write them as $\kappa_0 \kappa_1 \cdots \kappa_N$, with $\kappa_i \in K$. These indices correspond to the level numbers that are encoded in the states of $Q_G$, i.e., a letter $\kappa_i$ takes a state $(i-1, p)$ to a state $(i, q)$. Consequently, $\kappa_N = \varepsilon_q$, for all strings of $\mathcal{L}(A_G)$.

The enumeration algorithm uses a global stack that stores sets $S_i \subset Q_G$, which we call the state stack. In fact, we shall see that $S_{i+1} \subseteq \{(i, q) : q \in Q\}$ shall hold. Intuitively, if the algorithm has constructed a string $\kappa_0 \cdots \kappa_i$, the state stack shall store sets $S_0, \ldots, S_{i+1}$ such that each $S_{i+1}$ contains the states that can be reached from the states of $S_i$ by processing the letter $\kappa_i$. As every state of $A_G$ is connected to the final state, and as $\kappa_N = \varepsilon_q$ must hold, the algorithm does not put $S_{N-1}$ or $S_N$ on the state stack.

Let $\perp \notin K$ be a new letter, and define $\kappa <_K \perp$ for all $\kappa \in K$. The enumeration algorithm uses the functions $\text{minLetter}: Q_G \rightarrow K$ and $\text{nextLetter}: Q_G \times K \rightarrow K \cup \{\perp\}$ as subroutines, which we define as follows for all $q \in Q_G$ and $c \in K$:

- $\text{minLetter}(q)$ is the $<_K$-smallest $\kappa \in K$ with $\delta_G(q, \kappa) \neq \emptyset$;
- $\text{nextLetter}(q, \kappa)$ is the smallest $\kappa' \in K$ with $\kappa <_K \kappa'$, and $\delta_G(q, \kappa') \neq \emptyset$; or $\perp$, if no such $\kappa'$ exists.

It is easily seen that these functions can be precomputed in time $O(n^2 N)$, ideally when computing $A_G$. The actual enumeration algorithm, $\text{enumerate}$, is given as Algorithm 1 below.

Algorithm 1: enumerate

1. $S_0 := \{q_0\}$;
2. $k = \text{minString}(0)$;
3. while $k \neq \perp$ do
   4. output $k$;
   5. $k = \text{nextString}(k)$;

The main idea is very simple: $\text{enumerate}$ calls $\text{minString}$ (Algorithm 2) to construct the $<_r$-smallest string of $\mathcal{L}(A_G)$, using the $\text{minLetter}$ functions. Starting at $S_0 = \{q_0\}$, $\text{minString}$ determines $\kappa_0 \cdots \kappa_N$ by first choosing $\kappa_0$ as the $<_K$-smallest letter that can be read when in $q_0$, and then computing $S_1$ as all states that can be reached from there by reading $\kappa_0$. For each $S_i$, the algorithm then applies $\text{nextLetter}$ to each state of $S_i$, chooses $\kappa_i$ as the $<_K$-smallest of these letters, and puts all states that can be reached by reading $\kappa_i$ when in a state in

50
$S_i$ into the set $S_{i+1}$. In other words, constructing the sets $S_i$ can be understood as an on-the-fly simulation of the power-set construction on $A_G$.

As all strings of $\mathcal{L}(A_G)$ have length $N+1$, and the final state of $A_G$ is reachable from every state, this is sufficient to determine the $<_r$-minimal string of the language.

To enumerate all further strings, enumerate uses the subroutine nextString (Algorithm 3) repeatedly. Given a string $k = \kappa_0 \cdots \kappa_N$, nextString finds the rightmost letter of $k$ that can be increased (according to $<_k$), and changes this $\kappa_i$ accordingly. It then calls minString to complete the string $<_r$-minimally, by finding letters $\kappa_{i+1}, \ldots, \kappa_N$ that are $<_k$-minimal. While doing so, minString updates the state stack, according to the computed letters.

**Algorithm 2: minString($l$)**

<table>
<thead>
<tr>
<th>Input</th>
<th>an integer $l$ with $0 \leq l \leq N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assumptions</td>
<td>state stack contains $S_0, \ldots, S_l$ for some $p \in \mathcal{K}_l$</td>
</tr>
<tr>
<td>Output</td>
<td>the $&lt;_r$-smallest $k \in \mathcal{K}^{N-l}$ that is accepted by $A_G$ when starting in a state of $S_l$</td>
</tr>
<tr>
<td>Side effects</td>
<td>updates the state stack to $S_0, \ldots, S_{N-1}$ for $p \cdot k$</td>
</tr>
</tbody>
</table>

1. for $i := l$ to $N - 1$ do 
2. \quad find $q \in S_i$ such that minLetter($q$) is $<_k$-minimal; 
3. \quad $\kappa_i := \text{minLetter}(q)$; 
4. \quad if $i < N - 1$ then 
5. \quad \quad $S_{i+1} := \bigcup_{p \in S_i} \delta_G(p, \kappa_i)$; 
6. \quad \quad push $S_{i+1}$ on state stack; 
7. \quad $\kappa_N = \tilde{c}_{q_f}$; 
8. return $\kappa_i \cdots \kappa_{N-1} \cdot \kappa_N$;

**Complexity** We now determine the complexity of the sub-routines. We begin with minString: The for-loop is executed $O(N)$ times. In each iteration of the loop, $q$ can be determined in time $O(n)$ by using the precomputed minLetter-function, and $\kappa_i$ can then be computed in $O(1)$. To build the set $S_{i+1}$, we need time $O(n^2)$. Hence, a call of minString takes at most time $O(Nn^2)$.

In each call of nextString, the for-loop is executed $O(N)$ times. In each iteration, $\kappa_i$ can be found in time $O(n)$, as nextLetter was precomputed. Furthermore, if $\kappa_i \neq \bot$, the subroutine updates the state stack in $O(n^2)$ and terminates after a single call of minString, which takes $O(Nn^2)$. Hence, the total complexity of nextString is $O(Nn + n^2 + Nn^2)$, or simply $O(Nn^2)$.

The total complexity of the preprocessing is obtained by adding up the complexity of computing $\mathcal{E}$ (and $\mathcal{C}$) of $O(mn)$, constructing $G$ and $A_G$ in $O(Nn^2)$, computing minLetter and nextLetter in $O(Nn^2)$, and a call of minString in $O(Nn^2)$. This adds up to $O(Nn^2 + mn)$ for the preprocessing and finding the first answer. Furthermore, as each call of nextString takes $O(Nn^2)$, the delay is clearly polynomial.
Algorithm 3: nextString(w)

Input: a string \( k = \kappa_0 \ldots \kappa_N, \kappa_i \in \mathcal{K} \)

Assumptions: state stack contains \( S_0, \ldots, S_{N-1} \) for \( k \)

Output: the \(<r\)-smallest word \( k' \in \mathcal{L}(A_G) \) with \( k < k' \), or \( \perp \) if no such \( k' \) exists

Side effects: updates the state stack to \( S'_0, \ldots, S'_{N-1} \) for \( k' \)

1. for \( i := N - 1 \) to 0 do
2. let \( \kappa_i \) be the \(<k\)-minimal element of \( \{ \text{nextLetter}(q, \kappa_i) \mid q \in S_i \} \);
3. if \( \kappa_i = \perp \) then pop \( S_i \) from the state stack;
4. else
5. if \( i < N - 1 \) then
6. \( S_{i+1} := \bigcup_{p \in S_i} \delta_G(p, \kappa_i) \);
7. push \( S_{i+1} \) on state stack;
8. return \( \kappa_0 \cdots \kappa_i \cdot \text{minString}(i+1) \);
9. return \( \perp \);

Figure 4.1: The NFA \( A_G \) from Example 4.3.3. For convenience, we write each variable configuration \( \tilde{c}_q \) as \( \tilde{c}_q(x) \).

Recall that by Lemma 4.4.1, \( m \) is in \( O(n) \) if we derived \( A \) from a regex formula. Hence, for regex formulas, the preprocessing is in \( O(Nn^2) \) as well.

Finally, note that if \( A_G \) is deterministic, the complexity of the delay drops to \( O(NN) \): As each \( S_i \) contains at most \( N \) states, both \text{minString} \( \) and \text{nextString} \( \) can be computed in \( O(N) \).

Example 4.3.3. Consider \( A_{\text{fun}} \) (from Example 4.3.1) and \( d := \text{aa} \). From \( A_{\text{fun}} \) and \( d \), the algorithm constructs the NFA \( A_G \) that is shown in Figure 4.1. To see that \( \mathcal{L}(A_G) \) corresponds to \([A_{\text{fun}}](d)\), take note that the table in Example 4.3.2 lists each element of \([A_{\text{fun}}](d)\) together with its corresponding sequence of variable configurations. \( \square \)

Note that the NFA \( A_G \) in Example 4.3.3 is deterministic, which means that enumerating \( \mathcal{L}(A_G) \) actually requires less effort than in the general case: As stated in Theorem 4.3.1 if \( A \) has \( n \) states, we can enumerate \([A](d)\) with a delay of \( O(n^2|d|) \). But if \( A_G \) is deterministic, this can be lowered to \( O(|d|) \).
While there are automata where the worst case for the algorithm is reached, the stated upper bounds only applies to automata where the states have many outgoing transitions (or a large number of $\epsilon$-transitions). For examples where we can use it for better upper bounds, see the proofs of Lemma 4.4.6 and Theorem 4.5.5.

As a final remark, we observe that the algorithm uses the variable operations on the transitions of $A$ only to compute the variable configurations. Afterwards, these transitions are treated as $\epsilon$-transitions instead. This chapter uses functional vset-automata to represent regex formulas. Instead, one could directly convert each regex formula into an $\epsilon$-NFA $A$ and a function that maps each state of $A$ to a variable configuration. The enumeration algorithm and the “compilation lemmas” from Section 4.4 would directly work with this model. Whether this actually allows constructions that are more efficient or significantly simpler remains to be seen.

4.4 Upper Bounds of UCQ Evaluation

In this section, we explain the implications of this Theorem 4.3.1 for both evaluation approaches.

Canonical Relational Approach

It was already shown by Fagin et al. [27] that every regex formula can be converted into an equivalent vset-automaton (where a vset-automaton $A$ and a regex formula $\alpha$ are said to be equivalent if $[A] = [\alpha]$). We next show that this is possible in a way that is efficient and results in functional vset-automata. (Recall that we assume regex formulas to be functional by convention.)

**Lemma 4.4.1.** Given a regex formula $\alpha$, one can construct in time $O(|\alpha|)$ a functional vset-automaton $A$ with $[A] = [\alpha]$.

**Proof.** Let $\alpha$ be a regex formula, and define $V := \text{Vars}(\alpha)$. Assume that $\alpha$ is represented as its syntax tree. We first rewrite $\alpha$ into a regular expression $\hat{\alpha}$ over the alphabet $\Sigma \cup \Gamma_V$ with $L(\hat{\alpha}) = R(\alpha)$. Recall that $R(\alpha)$ is the ref-word language of $\alpha$. This is done by recursively replacing every node that represents a variable binding $x\{\beta\}$ with the concatenation $x\cdot\beta\cdot\tau x$. Then the length of $\hat{\alpha}$ is linear in the length of $\alpha$; and the rewriting is possible in linear time as well.

Next, we convert $\hat{\alpha}$ into an $\epsilon$-NFA $A$ with $L(A) = L(\hat{\alpha})$. Using Thompson’s construction (cf. e.g., [45]), this is possible in linear time. Furthermore, both the number of states and the number of transitions of $A$ are linear in the length of $\hat{\alpha}$ (and, hence, also in the length of $\alpha$). Finally, note that the construction ensures that $A$ has only one accepting state. (Recall that vset-automata are required to have a single accepting state).

This allows us to interpret $A$ as a vset-automaton with variable set $V$, using $R(A) = L(A)$. As $L(A) = L(\hat{\alpha}) = R(\alpha)$ holds by definition, we know that $R(A) = R(\alpha)$. Furthermore, as $\alpha$ is functional, its ref-word language contains only the valid ref-words. That is, $R(\alpha) = \text{Ref}(\alpha)$ holds, which implies $\text{Ref}(A) = \text{Ref}(A)$. 

53
\( \mathcal{R}(A) = \text{Ref}(\alpha) \). This allows us to make two conclusions: Firstly, that \( \text{Ref}(A) = \mathcal{R}(A) \), hence \( A \) is functional. And, secondly, that \( [A] = [\alpha] \), as \( \text{Ref}(A) = \text{Ref}(\alpha) \) implies \( \text{Ref}(A, d) = \text{Ref}(\alpha, d) \) for all \( d \in \Sigma^* \).

Note that as the proof operates via ref-words, another method for converting regular expressions to NFAs could be chosen instead of Thompson’s construction. The same result was shown independently by Morciano et al. [56].

We note here that the complexity of the preprocessing stated in Theorem 4.3.1 holds for a general functional vset-automaton. If, however, the automaton is derived from a regex formula \( \alpha \) by the construction we use for proving Lemma 4.4.1, then the time of this preprocessing drops to \( O(n^2|d|) \), where \( n \) is \( O(|\alpha|) \).

As a consequence, we conclude that the canonical relational approach to evaluation is actually efficient for UCQs, under two conditions. The first condition is that the regex CQs map to a tractable class of relational CQs. More formally, by tractable class of relational CQs we refer to a class \( Q \) of CQs that can be evaluated in polynomial total time, such as acyclic CQs, or more generally CQs with bounded hypertree width \([7]\). But, as shown in Theorem 4.2.2, this condition is not enough. The second condition is that there is a polynomial bound on the number of tuples of each regex formula. More formally, we say that a class \( A \) of regex formulas is polynomially bounded if there exists a positive integer \( d \) such that for every regex formula \( \alpha \in A \) and document \( d \) we have that the number of records in \( [\alpha](d) \) is \( O(|d|^d) \).

Clearly, if every regex formula in \( A \) can be evaluated in polynomial time, then \( A \) is polynomially bounded. From Theorem 4.3.1 we conclude that the other direction also holds. Hence, from Theorem 4.3.1 and Lemma 4.4.1 we establish the following theorem.

**Theorem 4.4.2.** Let \( P \) be a class of regex UCQs. If the regex formulas in the UCQs of \( P \) belong to a polynomially bounded class, and \( P \) maps to a tractable class of relational UCQs, then UCQs in \( P \) can be evaluated in polynomial total time.

Notice that we could use other yardsticks of enumeration efficiency, such as polynomial delay and incremental polynomial time \([13]\). Then, every occurrence of “polynomial total time” in the definition of tractable class of relational CQs and in Theorem 4.4.2 would be replaced with these other yardsticks.

Examples of polynomially bounded classes of regex formulas are:

- The class of regex formulas with at most \( k \) variables, for some fixed \( k \).
- The class of regex formulas \( \alpha \) with a key attribute, that is, a variable \( x \in \text{Vars}(\alpha) \) with the property that for all documents \( d \) and tuples \( \mu \) and \( \mu' \) in \( [\alpha](d) \), if \( \mu(x) = \mu'(x) \) then \( \mu = \mu' \).

A key attribute implies a polynomial bound as the number of spans of a document \( d \) is quadratic in \( |d| \). Interestingly, its existence can be tested in polynomial time.

**Proposition 4.4.3.** Given a functional vset-automaton \( A \) with \( n \) states, and a variable \( x \in \text{Vars}(A) \), it can be decided in time \( O(n^4) \) whether \( x \) is a key attribute.
Proof. The proof uses a modification of the intersection construction for NFAs. Given a functional vsf-automaton $A = (V, Q, q_0, q_f, \delta)$, our goal is to construct an NFA $A_x$ such that $L(A_x)$ is empty if and only if $x$ is not a key attribute. More specifically, the language $L(A_x)$ consists of all $d$ for which there exist $\mu, \mu' \in [A](d)$ such that $\mu(x) = \mu'(x)$ and $\mu(y) \neq \mu'(y)$ for some $y \in V$. Without loss of generality, we can assume that $A$ has at least two variables, and that we consider only non-empty documents (as $[A](\epsilon)$ contains at most one element). For our complexity analysis, let $n := |Q|$ and $v := |V|$.

Before we discuss the actual construction, we bring $A$ into a more convenient form. As in the proof of Theorem 4.3.1, we use the fact that every state of $A$ has a uniquely defined variable configuration $\tilde{c}_\cdot : X \to V$ (also see Section 1.3). Hence, the first step of the construction is computing the variable configurations of $A$. Recall that, if $A$ has $n$ states and $m$ transitions, this can be done in time $O(m + n)$. We can also assume that every state is reachable from $q_0$, and that $q_f$ is reachable from each state. We replace all variable transitions with $\varepsilon$-transitions. We translate the resulting automaton to an equivalent automaton without epsilon-transition by changing the transition function $\delta$ into a new one, namely $\delta'$. We do that using the standard procedure, this takes time $O(n^3)$ (in contrast to previous proofs, we do not need to use the tighter bound $O(mn)$).

The main idea of the construction is that $A_x$ is an NFA over $\Sigma$ that simulates two copies of $A$ in parallel. Both copies have the same behavior on $x$, but different behavior on (at least) one witness variable $y$. In addition to this, $A_x$ uses its states to keep track of whether such a $y$ has been found. Hence, we define $A_x := (Q_x, q_0,x, q_f,x, \delta_x)$, where

$$Q_x := \{(0, q_1, q_2) \mid q_1, q_2 \in Q, \tilde{c}_q_1 = \tilde{c}_q_2\} \cup \{(1, q_1, q_2) \mid q_1, q_2 \in Q, \tilde{c}_q_1(x) = \tilde{c}_q_2(x)\}.$$  

Intuitively, each state triple contains the state of each of the two copies of $A$, and a bit that encodes with 1 or 0 whether a witness $y$ has been found or not (respectively). Hence, we require for all state pairs that their variable configurations on $x$ are identical; but as long as no witness $y$ has been found, the variable configurations of all other variables also have to be identical. Following this intuition, we define $q_{0,x} := (0, q_0, q_0)$ and $q_{f,x} := (1, q_f, q_f)$. Finally, we construct $\delta_x$ as follows: For each pair of transitions $q_1 \in \delta'(p_1, \sigma)$ and $q_2 \in \delta'(p_2, \sigma)$ that satisfies $\tilde{c}_q_1(x) = \tilde{c}_q_2(x)$, we first define $(1, q_1, q_2) \in \tilde{c}_x((1, p_1, p_2), \sigma)$ (as the bit signifies that a witness $y$ has been found, we simply continue simulating the two copies of $A$). In addition to this, if $\tilde{c}_q_1 = \tilde{c}_q_2$, we also define

- $(0, q_1, q_2) \in \tilde{c}_x((0, p_1, p_2), \sigma)$ if $\tilde{c}_q_1 = \tilde{c}_q_2$, or
- $(1, q_1, q_2) \in \tilde{c}_x((0, p_1, p_2), \sigma)$ if $\tilde{c}_q_1 \neq \tilde{c}_q_2$, i.e., there is a variable $y$ with $\tilde{c}_q_1(y) \neq \tilde{c}_q_2(y)$.

In the first case, all variables have identical behavior, so we have not found a witness $y$, and do not change the bit. In the second case, there is a witness $y$, which allows us to change the bit. Note that we can precompute the sets of all $(q_1, q_2) \in Q$ with $\tilde{c}_q_1 = \tilde{c}_q_2$ or $\tilde{c}_q_1 \neq \tilde{c}_q_2$ in time $O(vn^2)$.

55
Now $A_x$ describes exactly the language of strings $d$ for which $x$ does not have the key property. As $A'$ has $O(n^2)$ transitions, we can construct $A_x$ in time $O(vn^2 + n^4) = O(n^4)$, and emptiness of $L(A_x)$ can also be decided in time $O(n^4)$, by checking whether $A_x$ contains a path from $(0, q_0, q_0)$ to $(1, q_f, q_f)$. Note that such a path also provides a witness input string $d$, and, by decoding the variable configurations of the states along the path, the corresponding $\mu, \mu' \in \mathcal{L}[d]$ with $\mu(x) = \mu'(x)$ and $\mu \neq \mu'$. As a side-effect, this construction also shows that the shortest witness that $x$ does not have the key property is of length $O(n^2)$.

\section*{Compilation to Automata}

We now discuss the second evaluation approach: compiling the regex UCQ to a functional vset-automaton, and then applying Theorem 4.3.1. An immediate consequence of a combination with past results is as follows. There are computable (but potentially exponential) conversions of spanners in a regular representation into a vset-automaton \cite{22}, and of a vset-automaton into a functional vset-automaton \cite{31}. Hence, we get the following corollary of Theorem 4.3.1.

\begin{corollary}
Spanners $q$ in a regular representation (e.g., regex UCQs) can be evaluated with FPT delay for the parameter $|q|$.
\end{corollary}

The corollary should be contrasted with the traditional relational case, where Boolean CQ evaluation is W[1]-hard when the size of the query is the parameter \cite{47}. Hence, in that respect regex UCQ evaluation is substantially more tractable than UCQ evaluation in the relational model.

Our next results are established by applying efficient compilations. Such compilations were obtained independently by Morciano et al. \cite{67}; we discuss the differences in the approaches after each result. (Generally, while both proofs are based on standard constructions, ref-words allow us to take shortcuts.) Furthermore, our proofs also discuss the constructions with respect to Theorem 4.3.1.

We begin with the most straightforward result, the projection operator.

\begin{lemma}
Given a functional vset-automaton $A$ and $Y \subseteq \text{Vars}(A)$, one can construct in linear time a functional vset-automaton $A_Y$ with $[A_Y] = [\pi_Y(A)]$.
\end{lemma}

\begin{proof}
Let $A$ be a functional vset-automaton, $V := \text{Vars}(A)$, and $Y \subseteq V$. We define the morphism $h_Y : (\Gamma_V \cup \Sigma)^* \rightarrow (\Gamma_Y \cup \Sigma)^*$ for every $g \in (\Gamma_V \cup \Sigma)$ by $h_Y(g) = \epsilon$ if $g \in (\Gamma_V \setminus \Gamma_Y)$, and $h_Y(g) := g$ for all $g \in (\Gamma_Y \cup \Sigma)$. In other words, $h_Y$ erases all $x^+$ and $\neg x$ with $x \notin Y$, and leaves all other symbols unchanged.

We obtain $A_Y$ from $A$ by replacing the label $g$ of each transition with $h(g)$. In other words, for each $x \notin Y$, each transition with $x^+$ or $\neg x$ is replaced with an $\epsilon$-transition. Clearly, this is possible in linear time, and $R(A_Y) = h_Y(R(A))$. As $A$ is functional, this implies $R(A_Y) = h_Y(\text{Ref}(A))$.

Now assume that $A_Y$ is not functional, i.e., that there exists an $r \in (\text{Ref}(A_Y) \setminus R(A_Y))$. Then $r$ is not valid, which means that there is an $y \in Y$ such that $y^+$ or $\neg y$ does not occur exactly once, or both symbols occur in the wrong order. By definition of $A_Y$, there is an $\hat{r} \in R(A)$ with $h_Y(\hat{r}) = r$; and as $h_Y$ erases only symbols of $\Gamma_V \setminus \Gamma_Y$ and leaves all other symbols unchanged, this means that $\hat{r}$
is also not valid, which contradicts our assumption that $A$ is functional. Hence, $\text{Ref}(A_Y) = \mathcal{R}(A_Y) = h_Y(\text{Ref}(A))$ holds. This also implies $[A_Y] = [\pi_Y(A)]$. \hfill \qed

As the proof is obtained by replacing all transitions for operations on variables that are not in $Y$ with $\epsilon$-transitions, one of its advantages is that it showcases a nice use of the ref-word semantics. The situation is similar for the union operator.

**Lemma 4.4.6.** Given functional vset-automata $A_1, \ldots, A_k$ with $\text{Vars}(A_1) = \cdots = \text{Vars}(A_k)$, one can construct in linear time a functional vset-automaton $A$ with $[A] = [A_1 \cup \cdots \cup A_k]$.

**Proof.** For $1 \leq i \leq k$, let $n_i$ denote the number of states and $m_i$ denote the number of transitions of $A_i$. We can construct $A$ by using the standard construction for union: We add a new initial and a new final state, and connect these with $\epsilon$-transitions to the “old” initial and final states of the $A_i$. Clearly, $\mathcal{R}(A) = \bigcup_{i=1}^{k} \mathcal{R}(A_i)$, and as each $A_i$ is functional, $\text{Ref}(A) = \bigcup_{i=1}^{k} \text{Ref}(A_i)$ follows. Hence, $A$ is functional, and $[A] = \bigcup_{i=1}^{k} [A_i] = [A_1 \cup \cdots \cup A_k]$. Adding the new states takes time $O(k)$, and outputting $A$ takes time $O(\sum_{i=1}^{k} (n_i + m_i))$. As this is the same size as the input of the algorithm, the construction runs in linear time. \hfill \qed

Notice that the upper-bound for the worst case complexity of Theorem 4.3.1 is lower than the number of states of the constructed automaton. Observe that in Lemma 4.4.6, the number of states of the automata is not bounded. The situation is different for the join operator, which also uses a construction that is not completely straightforward.

**Lemma 4.4.7.** Given two functional vset-automata $A_1$ and $A_2$, each with $O(n)$ states and $O(v)$ variables, one can construct in time $O(vn^4)$ a functional vset-automaton $A$ with $[A_1 \bowtie A_2] = [A]$.

**Proof.** Assume we are given two functional vset-automata $A_1$ and $A_2$, with $A_i = (V_i, Q_i, q_{0,i}, q_{f,i}, \delta_i)$. Let $v_i := |V_i|$, $n_i := |Q_i|$, and let $m_i$ denote the number of transitions of $A_i$. Furthermore, let $V := V_1 \cup V_2$ and define $v := |V|$. We assume that in each $A_i$, all states are reachable from $q_{0,i}$, and $q_{f,i}$ can be reached from every state. In order to construct a functional vset-automaton $A$ with $[A] = [A_1 \bowtie A_2]$, we modify the product construction for the intersection of two NFAs.

Note that construction has to deal with the fact that the ref-words from $\mathcal{R}(A_1)$ and $\mathcal{R}(A_2)$ can use the variables in different orders. As a simple example, consider $r_1 := x+y \leftarrow a \rightarrow y$ and $r_2 := y \leftarrow x \rightarrow a \rightarrow x$. Then $\mu^{r_1} = \mu^{r_2}$, although $r_1 \neq r_2$. Hence, we cannot directly apply the standard construction for NFA-intersection; and for complexity reasons, we do not rewrite the automata to impose an ordering on successive variable operations. Instead, we use the variable configurations of $A_1$ and $A_2$. 

57
Construction: In order to simplify the construction, we slightly abuse the definition, and label the variable transitions of $A$ with sets of variable transitions, instead of single transitions. (Similarly to extended vsf-automata defined by Florenzano et al. [30].) We discuss the technical aspects of this at the end of the proof (in particular its effect on the complexity). This decision allows us to restrict the number of states in $A$, which in turn simplifies the construction: Like the standard construction of the intersection, the main idea of the proof is that $A$ simulates each of $A_1$ and $A_2$ in parallel. Hence, its state set is a subset of $Q_1 \times Q_2$. But as we shall see, this construction uses the variable configurations $\tilde{C}_i$ of $A_i$ to determine how $A$ should act on the variables (instead of the variable transitions). In particular, we shall require that all states of $A$ are consistent, where $(q_1, q_2) \in Q_1 \times Q_2$ is consistent if $\tilde{C}_1(q_1)(x) = \tilde{C}_2(q_2)(x)$ for all $x \in V_1 \cap V_2$. In other words, the variable configurations of $q_1$ and $q_2$ agree on the common variables of $A_1$ and $A_2$.

By using sets of variable operations, we can disregard the order in which $A_1$ and $A_2$ process the common variables; and as variable transitions connect only consistent states of $A$ that encode consistent pairs of states, the fact that $A_1$ and $A_2$ are functional shall ensure that $A$ is also functional.

Before the algorithm constructs $A$, it computes the following sets and functions:

1. the variable configurations function $\tilde{C}_i$ for $A_i$,
2. the set $Q$ of all consistent $(q_1, q_2) \in Q_1 \times Q_2$,
3. the sets $Q_i$ of all $(p, q) \in Q_i \times Q_i$ with $\tilde{C}_i(p) = \tilde{C}_i(q)$,
4. the $\epsilon$-closures $\mathcal{E}_i: Q_i \to 2^{Q_i}$, where for each $p \in Q_i$, $\mathcal{E}_i(p)$ contains all $q \in Q_i$ that can be reached from $p$ by using only $\epsilon$-transitions,
5. the variable-$\epsilon$-closures $\mathcal{VE}_i: Q_i \to 2^{Q_i}$, where for each $p \in Q_i$, the variable-$\epsilon$-closure $\mathcal{VE}_i(p)$ contains all $q \in Q_i$ that can be reached from $p$ by using only transitions from $\{\epsilon\} \cup \Gamma_{V_i}$,
6. for each $\sigma \in \Sigma$, a function $T_i^\sigma: Q_i \to 2^{Q_i}$, that is defined by $T_i^\sigma(p) := \bigcup_{q \in \delta_i(p, \sigma)} \mathcal{E}_i(q)$ for each $p \in Q_i$.

In other words, $\mathcal{VE}_i$ can be understood as replacing all variable transitions in $A_i$ with $\epsilon$-transitions, and then computing the $\epsilon$-closure; while $T_i^\sigma(q)$ contains all states that can be reached by processing exactly one terminal transition, and then arbitrarily many $\epsilon$-transitions.

As mentioned in the proof of Theorem 4.3.1 the variable configurations $\tilde{C}_i$ can be computed in $O(v_in_i)$. These can then be used to compute $Q$ in time $O(|V_1 \cap V_2| \cdot n_1 n_2)$, and each $Q_i$ in time $O(v_in_i^2)$. By using the standard algorithm for transitive closures in directed graphs (see e.g., Skiena [73]), each $\mathcal{E}_i$ and $\mathcal{VE}_i$ can be computed in time $O(n_in_i)$. Finally, each of the $T_i^\sigma$ can be computed in time $O(mn_i)$, by processing each transition of $A_i$ and then using $\mathcal{E}_i$.

Now, $O(v_in_i^2)$ dominates $O(v_in_i)$. Furthermore, for at least one $i$, it holds that $O(v_in_i^2)$ also dominates $O(|V_1 \cap V_2|n_1 n_2)$, as $|V_1 \cap V_2| \leq \min\{v_1, v_2\}$. Hence,
the total running time of this precomputations adds up to \(O(v_1n_1^2+m_1n_1+v_2n_2^2+m_2n_2)\).

We are now ready to define \(A := (V, Q, q_0, q_f, \delta)\), where \(Q\) is the set we defined above, \(q_0 := (q_{0,1}, q_{0,2})\), and \(q_f := (q_{f,1}, q_{f,2})\). Before we define \(\delta\), note that \(q_0\) and \(q_f\) are always consistent: as \(A_1\) and \(A_2\) are functional, the variable configurations of the initial and final states map all variables to \(0\) and \(c\), respectively. For the same reason, for all \(p \in Q_i\), all \(\sigma \in \Sigma\), \(\overline{C}_i(q) = \overline{C}_i(p)\) must hold for all \(q \in \mathcal{T}_i^\sigma(p)\) or \(q \in \mathcal{E}_i(p)\).

We now define \(\delta\) as follows:

1. For every pair of states \((q_1, q_2)\) with \(q_i \in \mathcal{E}_i(q_{0,i})\), we add an \(\epsilon\)-transition from \((q_{0,1}, q_{0,2})\) to \((q_1, q_2)\).
2. For every pair of states \((p_1, p_2)\) \(\in Q\) and every \(\sigma \in \Sigma\), we compute all \((q_1, q_2) \in \mathcal{T}_1^\sigma(p_1) \times \mathcal{T}_2^\sigma(p_2)\). To each such \((q_1, q_2)\), we add a transition with label \(\sigma\) from \((p_1, p_2)\).
3. For every pair of states \((p_1, p_2)\) \(\in Q\), we compute all \((q_1, q_2) \in V\mathcal{E}_1(p_1) \times V\mathcal{E}_2(p_2)\). If \((q_1, q_2) \in Q\), and \(\overline{C}_i(q_i) \neq \overline{C}_i(p_i)\), we add a variable transition from \((p_1, p_2)\) to \((q_1, q_2)\) that contains exactly the operations that map each \(\overline{C}_i(p_i)\) to \(\overline{C}_i(q_i)\).

Each of these three rules guarantees that the destination \((q_1, q_2)\) of each transition is indeed a consistent pair of states (the third rule requires this explicitly, while the other two use the properties of \(\mathcal{E}_i\) and \(\mathcal{T}_i^\sigma\) mentioned above together with the fact that \((p_1, p_2)\) is consistent).

**Correctness and Complexity:** To see why this construction is correct, consider the following: The basic idea is that \(A\) simulates \(A_1\) and \(A_2\) in parallel. The actual work of the simulation is performed by the transitions that were derived from the second or the third rule. The former simulate the behavior of \(A_1\) and \(A_2\) on terminal letters, while the letter simulate sequences of variable actions. In particular, as we use the variable-\(\epsilon\)-closures, a transition of \(A\) can simulate all sequences of variable actions that lead to a consistent state pair. Hence, it does not matter in which order \(A_1\) or \(A_2\) would process variables, the fact that the state of \(A\) is a consistent pair means that both automata agree on their common variables.

In order to keep the second and third rule simple, we also include the first rule: The effect of these transitions is that \(A\) can simulate that \(A_1\) or \(A_2\) changes states via \(\epsilon\)-transitions before it starts processing terminals or variables. Finally, to see that \(A\) is functional, assume that it is not. Then there is a ref-word \(r \in \mathcal{R}(A) \setminus \mathcal{Ref}(A)\) that is not valid. Let \(r_i\) be the result of projecting \(r\) to \(\Sigma \cup \Gamma_{V_j}\) (i.e., removing all elements of \(\Gamma_{V_j}\) with \(j \not= i\)). Then at least one of \(r_1\) or \(r_2\) is also not valid; we assume \(r_1\. But according to the definition of \(A\), this means that \(r_1 \in \mathcal{R}(A_1)\), which implies \(\mathcal{R}(A_1) \neq \mathcal{Ref}(A_1)\), and contradicts our assumption that \(A_1\) is functional. Hence, \(A\) must be functional.

Regarding the complexity of the construction, note that \(A\) has \(O(n_1n_2)\) states and \(O(n_1^2n_2^2)\) transitions, and can be constructed in time \(O(n_1^2n_2^2)\). All transitions
that follow from the first rule can obviously be computed in $O(n_1n_2)$. For the second rule, for each of $O(n_1n_2)$ many pairs $(p_1, p_2) \in Q$, we enumerate $O(n_1n_2)$ many pairs $(q_1, q_2) \in T(p_1) \times T(p_2)$, and check whether $(q_1, q_2) \in Q$. As we precomputed $Q$, this check is possible in $O(1)$, which means that these transitions can be constructed in time $O(n_1^2n_2^2)$. Likewise, for the third rule, we use that we precomputed the sets $Q_{v_1}^\pi$, which allows us to check $C_i^\pi(p) \neq C_i^\pi(q)$ in time $O(1)$ as well.

For the third rule, we need to take into consideration that the sets of variable operations have to be computed. Doing this naïvely would bring the complexity to $O((v_1 + v_2)n_1^2n_2^2)$, but using the right representation, these sets can be precomputed independently for each $A_i$ in time $O(v_im_i)$, which allows us to leave the time for the variable transitions unchanged.

Up to this point, the complexity of the construction with precomputations is $O(v_1n_1^2 + m_1n_1 + v_2n_2^2 + m_2n_2 + n_1^2n_2^2)$. For $v, m, n$ with $v_i \in O(v)$, $m_i \in O(m)$, $n \in O(n)$, this becomes $O(vn^2 + mn + n^4)$, which we can simplify to $O(n^4)$, as $v \leq n$, and $m \leq n^2$. Note that this is the same complexity as constructing the product automaton for the intersection of two NFAs.

If we want to strictly adhere to the definition of vset-automata, we can rewrite $A$ into an equivalent vset-automaton $A_{\text{strict}}$, by taking each variable transition that is labeled with a set $S$, and replacing it with a sequence of states and transitions that enumerates the elements of $S$. This would add $O((v_1 + v_2)n_1^2n_2^2)$ states and the same number of transitions, and increase the complexity of the construction by $O((v_1 + v_2)n_1^2n_2^2)$, netting a total complexity of $O(m_1n_1 + m_2n_2 + (v_1 + v_2)n_1^2n_2^2)$. Under the assumptions from the last paragraph, this can be simplified to $O(vn^4)$.

While a more detailed analysis or a more refined construction might avoid this increase, it might be more advantageous to generalize the definition of vset-automata to allow sets of variable operations on transitions, as the complexities of computing the variable configurations and of Theorem 4.3.1 generalize to this model (as long as the number of variables is linear in the number of states).

In particular, if our actual goal is evaluating $A_1 \bowtie A_2$ by using Theorem 4.3.1 we can skip the step of rewriting $A$ into $A_{\text{strict}}$, and directly construct $G$ from $A$ (or, without explicitly constructing $A$, from the $E_i$ and $T_i^\pi$ functions). The total time for preprocessing is then $O(v_1n_1^2 + m_1n_1 + v_2n_2^2 + m_2n_2 + Nn_1^2n_2^2)$, or, if simplified, $O(Nn_1^4)$. This is also the complexity of the delay.

Naturally, all these constructions can be extended to the join of $k$ vset-automata, where the complexities become $O(n^{2k})$ for the construction of the vset-automaton $A$ with sets of variable operations, as well as $O(vn^{2k})$ for the construction of $A_{\text{strict}}$ or $O(Nn^{2k})$ for the preprocessing and the delay of the enumeration algorithm.

Both this proof and the one by Morciano et al. [56] build on the standard construction for automata intersection; the key difference is that our proof takes advantage of variable configurations. Note that joining $k$ automata leads to a time of $O(vn^{2k})$, which is only polynomial if $k$ is bounded. Due to Theorem 4.2.2 this is unavoidable under standard complexity theoretic assumptions.

This motivates the following definition. Let $k \geq 0$ be fixed. A regex $k$-CQ is
a regex CQ with at most \( k \) atoms. A regex \( k \)-UCQ is a UCQ where each CQ is a \( k \)-CQ (i.e., a disjunction of \( k \)-CQs). From Lemmas 4.4.1 and 4.4.7 we conclude that we can convert, in polynomial time, a join of \( k \) regex formulas into a single functional vset-automaton. Then, Lemma 4.4.5 implies that projection can also be efficiently pushed into the functional vset-automaton. Hence, in polynomial time we can translate a \( k \)-CQ into a functional vset-automaton. Then, with Lemma 4.4.6 we conclude that this translation extends to \( k \)-UCQs. Finally, by applying Theorem 4.3.1 we arrive at the following main result.

**Theorem 4.4.8.** For every fixed \( k \), regex \( k \)-UCQs can be evaluated with polynomial delay.

Hence, where Theorem 4.4.8 applies, it is more powerful than Theorem 4.4.2: the former holds for all regex \( k \)-UCQs, while the latter has additional requirements, even when limited to \( k \)-UCQs.

In the next section, we extend the main theorems of this section, Theorems 4.4.2 and 4.4.8, to incorporate string equalities.

### 4.5 String Equality Selection

Our results in the previous two sections only apply to regex UCQs. As these are not allowed to use string equality selections, these have the same expressive power as regular spanner representations. In this section, we discuss how our results can be extended to include string equality selections, which allows us to reach the full expressive power of core spanners.

#### 4.5.1 Formal Setup

A regex CQ with string equalities is a core spanner representation of the form

\[
q := \pi_Y \left( \zeta_{x_1,x_2}^= \cdots \zeta_{x_l,x_{l+1}}^= (\alpha_1 \Join \cdots \Join \alpha_k) \right)
\]

for some \( k \geq 1 \) and \( l \geq 0 \). Also here, for clarity of notation, if \( Y = \{y_1, \ldots, y_m\} \), we sometimes write \( q(y_1, \ldots, y_m) \) when using \( q \) to make \( \text{Vars}(q) \) more explicit.

Similarly to regex CQs, each regex formula \( \alpha_i \) is called a regex atom and each selection \( \zeta_{x_j,x_{j+1}}^= \) is called an equality atom. We denote by \( \text{atoms}(q) \) the set of atoms of \( q \). For each equality atom \( \zeta_{x_j,x_{j+1}}^= \), we define \( \text{Vars}(\zeta_{x_j,x_{j+1}}^=) := \{x_j, x_{j+1}\} \). Note that our definition of the selection operator (Section 2.3) implies that every variable that occurs in an equality atom also appears in at least one regex atom.

A regex UCQ with string equalities is a spanner \( q \) of the form \( q := \bigcup_{i=1}^k q_i \) where each \( q_i \) can be a CQ with string equalities. The following propositions follows quite easily from the core simplification lemma [27].

**Proposition 4.5.1.** The class of spanners expressible as regex UCQs with string equalities is that of the core spanners.
Proof. It suffices to prove that every core spanner can be expressed by a regex UCQ with string equalities. For this, we use the core simplification lemma [27, Lemma 4.19], which states that every core spanner $P$ can be expressed as $\pi_Y SA$, where $A$ is a vsf-automaton, $S$ is a sequence of string equality selections over $V := \text{Vars}(A)$, and $Y \subseteq V$. We then use Proposition 4.1.1 to convert $A$ into an equivalent regex UCQ $q := \bigcup_{i=1}^{k} q_i$, and define the regex UCQ with string equalities $q' := \bigcup_{i=1}^{k} q'_i$, where each $q'_i$ is obtained by adding $\pi_Y$ and $S$ to $q_i$. Obviously, $q'$ and $P$ are equivalent.  

4.5.2 Lower Bound

The main difficulty when dealing with string equality selections is that this operator quickly becomes computationally expensive, even without using joins. More specifically, Freydenberger and Holldack [34] showed that combined with string equalities, a single regex formula and a projection to a Boolean spanner already lead to an intractable evaluation problem.

Theorem 4.5.2. [34] Evaluation of Boolean regex CQs with string equalities is NP-complete, even if restricted to queries of the form $\pi \zeta_{x_1, y_1} \cdots \zeta_{x_m, y_m} \alpha$.

In other words, even a single regex formula already leads to NP-hardness. The proof by Freydenberger et al. [34] uses a reduction from the membership problem for so-called pattern languages. It was shown by Fernau et al. [29] that this membership problem is $W[1]$-complete for various parameters. We prove that the situation is analogous for Boolean regex CQs with string equalities.

Theorem 4.5.3. Evaluation of Boolean regex CQs $q$ with string equalities is $W[1]$-hard for the parameter $|q|$, even if restricted to queries of the following form: $\pi \zeta_{x_1, y_1} \cdots \zeta_{x_m, y_m} \alpha$.

Proof. We prove the claim by modifying the proof of Theorem 4.2.2. Let $d$ and $\gamma$ be defined as in that proof. However, we do not use regex formulas $\delta_i$ to ensure that all variables $y_{i,j}$ and $x_{l,j}$ with $1 \leq i < l < j \leq k$ map to the same substring $v_l$. Instead, for each $1 \leq l \leq k$, we define a sequence $S_l$ of $k - 2$ binary string equality selections that is equivalent to the $(k - 1)$-ary string equality selection on the variables $y_{1,l}$ to $y_{l-1,l}$ and $x_{l,l+1}$ to $x_{l,k}$. We then define our query $q$ as

$$q := \pi \delta_1 \cdots S_k \gamma.$$ 

Note that being able to use string equality predicates, we do not need to iterate through all of the possible $v$ as in the $\delta_i$. Therefore the proof of correctness of the reduction used in the proof of Theorem 4.2.2 can be simply adapted to show correctness of this reduction. Observe that this is an FPT reduction with parameter $|q|$ since $|q|$ is of size $O(k^2)$ (i.e., the number of string equality predicates we use depends only on the clique size). Finally, we remark that like the proof of Theorem 4.2.2, this proof can be adapted to a binary alphabet using standard coding techniques. 

62
Hence, while a regex CQ that consists of a single regex formula can be evaluated efficiently (due to Lemmas 4.4.1 and 4.4.5 and Theorem 4.3.1), even limited use of string equalities can become expensive. This result can be obtained by combining the reduction by Fernau et al. [29] with the reduction by Freydenberger et al. [34], the former proof requires encodings that are needlessly complicated for our purposes (this is caused by the comparatively low expressive power of pattern languages). Instead, we take the basic idea of an FPT-reduction from the \( k \)-clique problem, and give a direct proof that is less technical. In fact, observe that the proof of Theorem 4.5.3 is similar to that of Theorem 4.2.2 except that in the former the query \( q \) constructed in the reduction is determined solely by the parameter \( k \), and not the size of the graph as in the latter.

The reader should contrast Theorem 4.5.3 with Corollary 4.4.4. The combination of the two complexity results shows that, with respect to the parameter \(|q|\), string equality considerably increases the parameterized complexity: Boolean regex CQ evaluation used to be FPT, and is now W[1]-hard.

Finally, note that queries as in Theorems 4.5.2 and 4.5.3 are always acyclic, as the variables of each equality atom also occur in \( \alpha \). Although they are not gamma-acyclic, we can rewrite each query into an equivalent gamma-acyclic query with \( k \)-ary string equalities, by merging each pair \( \zeta_X \) and \( \zeta_Y \) with sets \( X \cap Y \neq \emptyset \) into \( \zeta_{X,Y} \) until all equalities range over pairwise disjoint sets.

### 4.5.3 Upper Bounds

We now examine the upper bounds for our two evaluation strategies on regex UCQs with string equalities. Recall that as defined in Section 4.5.1, the atoms of regex CQs with string equalities consist of regex atoms along with equality atoms. Hence, if \( q(y_1, \ldots, y_m) \) is a regex CQ with string equalities and \( Q(y_1, \ldots, y_m) \) is a relational CQ we say that \( q \) maps to \( Q \) by treating \( q \) as a regex CQ with an extended set of atoms (see Section 4.1.1). The reader should contrast the following example with Example 4.1.1.

**Example 4.5.1.** The following regex CQ with string equalities:

\[
q(x, z) := \pi_{x,z}\zeta_{y,y'}(\alpha(x, y) \bowtie (y', z))
\]

maps to the relational CQ \( Q(x, z) := T(y, y'), R(x, y), S(y', z) \) but does not map to \( T(x, z) := T(y, y'), R(x, y), R(y', z) \) and not to \( T(x, z) := R(x, y), S(y', z) \).

We extend respectively the definition to (classes of) regex UCQs with string equalities.

For the canonical relational approach, we observe that each equality atom corresponds to a relation that is of polynomial size. Hence, we can directly use Theorem 4.4.2 to conclude the following.

**Corollary 4.5.4.** Let \( \mathcal{P} \) be a class of regex UCQs with string equalities. If the regex formulas in the UCQs of \( \mathcal{P} \) belong to a polynomially bounded class, and \( \mathcal{P} \) maps to a tractable class of relational UCQs, then UCQs in \( \mathcal{P} \) can be evaluated in polynomial total time.
Both have exploited the structure of $O_{rem 4.3.1}$, preprocessing and delay each have a complexity of $O(N)$ states and transitions. Moreover, we can build all possible paths to take, which means that $O_{rem 4.3.1}$ can be done naïvely in time $O(N^4)$. Using these pairs, we construct a functional vset-automaton $A_{eq}$ with $\text{Vars}(A_{eq}) = \{x_i, y_i \mid 1 \leq i \leq k\}$ such that $\mu \in [A_{eq}]$ if and only if $d_{\mu(x_i)} = d_{\mu(y_i)}$ holds for all $1 \leq i \leq k$. In other words, $A_{eq}$ describes exactly those spans of $d$ that satisfy the string equality selections.

We ensure that $A_{eq}$ has $O(N^{3k+1})$ states, and each state except the initial state and the final state has exactly one outgoing transition. More specifically, $A_{eq}$ encodes each possible $\mu \in [A](d)$ in a path of states. Each of these $\mu$ is characterized by selecting for each $1 \leq i \leq k$ the length of $d_{\mu(x_i)}$ (and, hence, $d_{\mu(y_i)}$), as well as the starting positions of $\mu(x_i)$ and $\mu(y_i)$. Hence, there are $O(N^{3k})$ possible $\mu$, and each can be encoded in a path of $O(N + k)$ successive states if we take variable operations into account. If we allow multiple variable actions on a single transition (see the Proof of Lemma 4.4.7), this is possible in $O(N)$ states; but even if we do not, we can safely assume that $k \leq N$, which also gives $O(N)$ states.

The initial state of $A_{eq}$ non-deterministically chooses with an $\epsilon$-transition which of these paths to take, which means that $A_{eq}$ can be constructed with $O(N^{3k+1})$ states and transitions. Moreover, we can build $A_{eq}$ while enumerating the pairs $s_1, s_2$ as described above, which means that the construction takes time $O(N^{3k+1})$. Note that if two selections have a common variable, we can lower the exponent by two, as we only need to account once for the length and the starting position. For example, $A_{eq}$ for $\zeta_{x,y} \zeta_{y,z} A$ only needs $O(N^5)$ states, instead of $O(N^7)$.

We can now use Lemma 4.4.7 to construct a functional vset-automaton $A_{join}$ with $[A_{join}] = [A \bowtie A_{eq}]$. As $[A_{join}](d) = [\zeta_{x_1,y_1} \cdots \zeta_{x_k,y_k} A](d)$ holds, this solves our problem.

Complexity analysis: If we directly use Lemma 4.4.7 together with Theorem 4.3.1 pre-processing and delay each have a complexity of $O(N(nN^{3k+1})^2) = O(N^{6k+3}n^2)$. While this is polynomial in $n$ and $N$, we can lower the degree by exploiting the structure of $A_{eq}$.

Take particular note of the structural similarity between $A_{eq}$ and the graphs that are used in the proof of Theorem 4.3.1 when matching on the string $d$: Both have $N$ levels, where a level $i$ represents that $i$ letters of $N$ have been
processed. Hence, if we wanted to enumerate $[A_{eq}](d)$ as described in the proof of Theorem 4.3.1, the constructed graph would only need to have $O(N^{3k+1})$ nodes, with exactly one outgoing edge for all nodes (except the initial and the final node). Consequently, the resulting NFA that is used for enumerate would only have $O(N^{3k+1})$ nodes and $O(N^{3k+1})$ transitions.

We now examine $A_{\text{join}}$. Each of its states $(q, q_{eq})$ encodes a state $q$ of $A$ and a state $q_{eq}$ of $A_{eq}$. The state $q$ has $O(n)$ successor states in $A$, and $q_{eq}$ has at most one successor state in $A_{eq}$ (unless it is the initial state; but as this state cannot be reached after it has been left, the number of transitions from it are dominated by the number of the other transitions). Hence, $(q, q_{eq})$ has at most $O(n)$ successors in $A_{\text{join}}$. In total, $A_{\text{join}}$ has $O(N^{3k+1}n)$ states, and $O(N^{3k+1}n^2)$ transitions. By combining these observations with the construction from the proof of Lemma 4.4.7, we see that $A_{\text{join}}$ can be constructed in time $O(vn^2 + mn + N^{3k+1}n^2)$.

Moreover, $A_{\text{join}}$ exhibits the same level structure as $A_{eq}$. Hence, in order to enumerate $[A_{\text{join}}](d)$, we can directly derive $G$ and $A_G$ from $A_{\text{join}}$, without needing the extra factor of $O(N)$ states to store how much of $d$ has been processed.

Thus, $A_G$ has $O(N^{3k+1}n)$ states and $O(N^{3k+1}n^2)$ transitions. We can construct $A_G$ from $A_{\text{join}}$, precompute its functions minLetter and nextLetter, and ensure that the final state is reachable from every state, and that each state is reachable in time $O(N^{3k+1}n^2)$. Finally, note that the alphabet $K$ of $A_G$ is the set of all variable configurations of $A$, as $\text{Vars}(A_{eq}) \subseteq \text{Vars}(A)$.

All that remains is calling enumerate on $A_G$. As in the proof of Theorem 4.3.1, its complexity is determined by the complexity of minString: The for-loop is executed $O(N)$ times. As each level of $A_G$ contains $O(N^{3k}n)$ nodes, the same bound holds for each $S_i$. Furthermore, as each of these nodes has $O(n)$ successors, each $S_{i+1}$ can be computed in time $O(N^{3k}n^2)$. Hence, executing minString requires time $O(N^{3k+1}n^2)$.

Hence, we can enumerate $[A_{\text{join}}](d)$ with a delay of $O(N^{3k+1}n^2)$. Including the construction of $A_{\text{join}}$, the preprocessing takes $O(vn^2 + mn + N^{3k+1}n^2)$, which we can simplify to $O(n^3 + N^{3k+1}n^2)$. Except for rather pathological cases, we can assume this to be $O(N^{3k+1}n^2)$.

Note that the compilation of $A_{eq}$ is dynamic, i.e., it depends on $d$; in fact, $[A_{eq}](d') = \emptyset$ for all strings $d' \neq d$. This dependency on $d$ is unavoidable: Recall that as shown by Fagin et al. [27], regular spanners are strictly less expressive than core spanners (i.e., string equality adds expressive power). If we could construct an $A_{eq}$ that worked on every document $d'$, this would immediately lead to a contradiction.

Analogously to the join in Section 4.4, the polynomial upper bound only holds for the construction if $m$ is fixed, and Theorem 4.5.3 suggests that this cannot be overcome under standard assumptions. Hence, for all fixed $k, m \geq 0$, we define the notion of a regex $k$-UCQ with up to $m$ string equalities analogously to a regex $k$-UCQ, with the additional requirement that each of the CQs uses at most $m$ binary string equality selections. It follows from Theorem 4.5.5 that for every constant $k$, adding a fixed number of string equalities to a regex $k$-UCQ does
not affect its enumeration complexity (in the sense that the complexity stays polynomial).

Corollary 4.5.6. For all fixed \( m \) and \( k \), regex \( k \)-UCQs with up to \( m \) string equalities can be evaluated with polynomial delay.

In other words, Theorem 4.4.8 can be extended to cover string equalities.

4.6 The Difference Operator

In the previous sections, we showed that we can compile all of the positive operators efficiently (i.e., in polynomial time) into a vset-automaton. For the string equality selection, we can do the compilation efficiently but for a certain document at hand. That is, the resulting vset-automaton is valid only for that document.

In the case of NFAs or regular expressions, compiling the complement into an NFA necessitates an exponential blowup in size \([24, 47]\). Since NFAs and regular expressions are the Boolean functional vset-automata and Boolean regex formulas, respectively, we conclude that constructing a vset-automaton that is equivalent to the difference of two vset-automata, or two regex formulas, entails an exponential blowup. Therefore, the static compilation (that is independent of the document) provably fails to yield tractability results for the difference.

In the case of NFAs and regular expressions, the membership of a string in the difference can be tested in polynomial time. In contrast, the following theorem states that, for regex formulas (and vset-automata), this is no longer true under the conventional complexity assumption \( P = NP \).

Theorem 4.6.1. The following problem is NP-complete. Given two functional regex formulas \( \gamma_1 \) and \( \gamma_2 \) with \( \text{Vars}(\gamma_1) = \text{Vars}(\gamma_2) \) and an input document \( d \), is \( \sqsubseteq \gamma_1 \setminus \gamma_2 \)(\( d \)) nonempty?

Proof. Membership in NP is straightforward: for functional regex formulas, membership can be decided in polynomial time \([32]\). Hence, we focus on NP-hardness.

We show a reduction from 3CNF-satisfiability. The input for 3SAT is a formula \( \varphi \) with the free variables \( x_1, \ldots, x_n \) such that \( \varphi \) has the form \( C_1 \land \cdots \land C_m \), where each \( C_j \) is a clause. In turn, each clause is a disjunction of three literals, where a literal has the form \( x_i \) or \( \neg x_i \) for \( i = 1, \ldots, n \). The goal is to determine whether there is an assignment \( \tau : \{ x_1, \ldots, x_n \} \rightarrow \{ \text{true, false} \} \) that satisfies \( \varphi \). Given a 3CNF formula \( \varphi \), we construct two sequential regex formulas \( \gamma_1 \) and \( \gamma_2 \) and an input document \( d \), such that there is a satisfying assignment for \( \varphi \) if and only if \( \sqsubseteq \gamma_1 \setminus \gamma_2 \)(\( d \)) \( \neq \emptyset \). We begin with the document \( d \), which is defined by \( d := a^n \).

The regex formulas \( \gamma_1 \) and \( \gamma_2 \) are constructed as follows. We associate every free variable \( x_i \) with a capture variable \( x_i \). We start by defining the auxiliary regex formulas

\[
\beta_i := ((x_i\{e\} \cdot a) \lor x_i\{a\})
\]

for \( 1 \leq i \leq n \) and then define \( \gamma_1 := \beta_1 \cdots \beta_n \). Intuitively, \( \gamma_1 \) encodes all of the legal assignments for \( \varphi \) in such a way that if \( x_i \) captures the substring ‘a’ then it...
Hence, deciding nonemptiness of the assignment $\mu_V(b)$ corresponds with assigning $\text{true}$ to the free variable $x_i$, and otherwise (in case it captures $\ell$), it corresponds with assigning to it $\text{false}$. Before defining $\gamma_2$, for each $1 \leq i \leq m$ we denote the indices of the literals that appear in $C_i$ by $i_1 < i_2 < i_3$ and define $\gamma_2$ as follows:

$$\gamma_2^i := \beta_1 \cdots \beta_{i-1} \cdot \delta_i \cdot \beta_{i+1} \cdots \beta_{i-2} \cdot \delta_{i+1} \cdots \beta_{i+1} \cdots \beta_n$$

where $\delta_i$ is defined as $(x_\ell \{\ell\} \cdot a)$ if $x_\ell$ appears as a literal in $C_i$ or as $(x_\ell \{a\})$ if $-x_\ell$ appears as a literal in $C_i$.

Intuitively, $\gamma_2^i$ encodes the assignments for which clause $C_i$ is not satisfied. We then set

$$\gamma_2 := \bigvee_{1 \leq i \leq m} \gamma_2^i.$$ 

We consider the following example:

$$\varphi = (x \lor y \lor z) \land (\neg x \lor y \lor \neg z)$$

We have $d := a^3$ since we have three variables $\{x, y, z\}$ and

$$\gamma_1 = \left( (x\{\ell\} \cdot a) \lor x\{a\} \right) \cdot \left( (y\{\ell\} \cdot a) \lor y\{a\} \right) \cdot \left( (z\{\ell\} \cdot a) \lor z\{a\} \right)$$

For the first clause we have

$$\gamma_2^1 := (x\{\ell\} \cdot a) \cdot (y\{\ell\} \cdot a) \cdot (z\{\ell\} \cdot a)$$

and for the second

$$\gamma_2^2 := (x\{a\}) \cdot (y\{\ell\} \cdot a) \cdot (z\{a\})$$

It is left to show that $\models \gamma_1 \backslash \gamma_2 \models (d) \neq \emptyset$ if and only if $\varphi$ has a satisfying assignment.

Note that for every assignment $\mu \in \models \gamma_1 \models (d)$ and for every $1 \leq j \leq n$, it holds that $\mu(x_j)$ is either $[j, j]$ or $[j, j + 1]$. Note also that the same is true also for $\mu \in \models \gamma_2 \models (d)$.

Let us assume that there exists a satisfying assignment $\tau$ for $\varphi$. We define $\mu$ to be the mapping that is defined as follows: $\mu(x_i) := [i, i]$ if $\tau(x_i) = \text{false}$ and $\mu(x_i) := [i, i + 1]$; otherwise (if $\tau(x_i) = \text{true}$). It then follows immediately from the definition of $\gamma_2$ that $\mu \in \models \gamma_1 \backslash \gamma_2 \models (d)$.

On the other hand, assume that $\mu \in \models \gamma_1 \backslash \gamma_2 \models (d)$. We can define an assignment $\tau$ in such a way that $\tau(x_i) = \text{true}$ if $\mu(x_i) = [i, i + 1]$ and $\tau(x_i) = \text{false}$ otherwise (if $\mu(x_i) = [i, i]$). It follows directly from the way we defined $\gamma_1$ and $\gamma_2$ that $\tau$ is a satisfying assignment for $\varphi$.

In our example, the assignment $\tau$ defined by $\tau(x) = \tau(y) = \text{true}$ and $\tau(z) = \text{false}$ is a satisfying assignment. Indeed, the mapping $\mu$ corresponds to this assignment that is defined by $\mu(x) = [1, 2]$, $\mu(y) = [2, 3]$ and $\mu(z) = [3, 3]$ is in $\models \gamma_1 \models (a^n)$ but is not in $\models \gamma_2 \models (a^n)$ since either (a) $\mu(x) = [1, 1]$ and $\mu(y) = [2, 2]$ or (b) $\mu(x) = [1, 2]$ and $\mu(y) = [2, 2]$.

Note also that the assignment $\mu$ defined by $\mu(x) = [1, 2]$, $\mu(y) = [2, 3]$ and $\mu(z) = [3, 4]$ is in $\models \gamma_1 \backslash \gamma_2 \models (a^n)$ since it is in $\models \gamma_1 \models (a^n)$ and not in $\models \gamma_2 \models (a^n)$. Indeed, the assignment $\tau$ for which $\tau(x) = \tau(y) = \tau(z)$ is a satisfying assignment for $\varphi$. Hence, deciding nonemptiness of $\models \gamma_1 \backslash \gamma_2 \models (d)$ is NP-hard.

We will revisit the difference operator in Section 5.3 where we present upper bounds for a broader class of spanners.
4.7 Concluding Remarks

We have studied the combined complexity of evaluating regex CQs and regex UCQs. We showed that the complexity is not determined only by the structure of the CQ as a relational query; rather, complexity can go higher since a single regex formula can already define a relation that is exponentially larger than the combined size of the input and output. Our upper bounds are based on an algorithm for evaluating a vset-automaton with polynomial delay. These bounds are based on two alternative evaluation strategies—the canonical relational evaluation and the compilation approach. In addition, we show that the difference operator does not behave as well as the positive RA operators as it is NP-hard to determine the emptiness of the difference of two regex formulas.

A crucial future direction is that of translating the upper bounds we presented into algorithms that substantially outperform the state of the art, at least when our tractability conditions hold. Beyond optimizing our translation and polynomial-delay algorithm, we would like to incorporate techniques of aggressive filtering for matching regular expressions \[82\] \[83\] and parallelizing polynomial-delay enumeration \[83\].
Chapter 5

Complexity of Schemaless Spanners

We continue our investigation of the combined complexity of regular spanners from the previous chapter. In this chapter, we explore the implication of adopting the “schemaless” semantics for spanners, as proposed and studied by Maturana et al. [55], that allow the extraction of incomplete information. While Amarilli et al. [7] and Florenzano et al. [30] have shown that the evaluation of schemaless spanners can be done in polynomial delay, we show that the schemaless semantics introduces computational hardness already for the join operator. Nevertheless, we propose and analyze syntactic constraints on the relational algebra (RA) expression and the regex formulas at hand, such that the expressive power is fully preserved and, yet, evaluation can be done with polynomial delay. Our technique is not (and provably cannot be) based on the static compilation of regex formulas, but rather on dynamic compilation into an automaton that incorporates both the query and the document. Recall that dynamic compilation was also used to compile the string equality selections in the previous chapter. With the similar dynamic compilation method, we present syntactic constraints for which we obtain efficient enumeration even in the presence of difference. Finally, we propose the extraction complexity, where the RA tree is fixed, and the input consists of the document and the atomic extractors (mapped to their corresponding positions in the RA tree). We present and discuss conditions that make the extraction complexity tractable and intractable.

The chapter presents joint work with Dominik D. Freydenberger, Benny Kimelfeld and Markus Kröll. The results were presented at the ACM Symposium on Principles of Database Systems 2019 [65]. The chapter is organized as follows: In Section 5.1, we present the basic terminology and concepts. We investigate the complexity of the natural join operator in Section 5.2 and the difference operator in Section 5.3. We extend our development to the extraction complexity in Section 5.4 and conclude in Section 5.5.

5.1 Schemaless and Schema-based Spanners

In Chapter 2, we defined the notion of a \((V, d)\)-record for a finite set \(V \subset \text{Vars}\) and a document \(d\). Here, we define the notion of a \(d\)-record: Given a document \(d\), a \(d\)-record is a mapping \(\mu\) from a finite set of variables, called the domain of \(\mu\) and
denoted $\text{dom}(\mu)$, into $\text{Spans}(d)$. If context allows, we use the term record instead of $d$-record. A schemaless spanner is a function $P$ that maps every document $d$ into a finite set $P(d)$ of $d$-records.

For a schemaless spanner $P$ and a document $d$, different $d$-records in $P(d)$ may have different domains. This stands in contrast to the spanners $P$ we defined in Chapter 2 where $P$ is associated with a set $V_P$ of variables and maps every document $d$ into a set of $(V_P, d)$-records.

Example 5.1.1. Let $\Gamma$ be the alphabet that consists of lowercase and uppercase English letters: $a, \cdots, z, A, \cdots, Z$; digits: $0, \cdots, 9$; and symbols: ‘’ that stands for whitespace, ‘.’ and ‘@’. Let $\Delta = \{\rightarrow\}$ where $\rightarrow$ stands for end of line. The input document $d_{\text{Students}}$ over $\Gamma \cup \Delta$ given in Figure 5.1 holds personal information on students. (Some of the positions are numbered for convenience.) Each line in the document describes information on a student in the following format: first name (if applicable), last name, phone number (if applicable) and email address. There are whitespaces in between these elements. The document spanner $P_{\text{StudentInfo}}$ extracts from the input document $d_{\text{Students}}$ the following set of records, given in a table for convenience.

<table>
<thead>
<tr>
<th>$x_{\text{first}}$</th>
<th>$x_{\text{last}}$</th>
<th>$x_{\text{mail}}$</th>
<th>$x_{\text{phone}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu_1$ : $[1, 7]$</td>
<td>$[8, 19]$</td>
<td>$[20, 29]$</td>
<td></td>
</tr>
<tr>
<td>$\mu_2$ : $[30, 37]$</td>
<td>$[46, 56]$</td>
<td>$[38, 45]$</td>
<td></td>
</tr>
<tr>
<td>$\mu_3$ : $[57, 62]$</td>
<td>$[63, 69]$</td>
<td>$[78, 89]$</td>
<td>$[70, 77]$</td>
</tr>
</tbody>
</table>

Note that the empty cells in the table stand for undefined. That is, we have $x_{\text{phone}} \notin \text{dom}(\mu_1)$ and $x_{\text{first}} \notin \text{dom}(\mu_2)$.

In what follows, we discuss different representation languages for schemaless spanners. Whenever a schemaless spanner is represented by a description $q$, we denote by $[q]$ the actual schemaless spanner that $q$ represents. We are using the notation $[\cdot]$ in order to clearly distinguish the schemaless semantics from the semantics $[\cdot]$ that was defined in Chapter 2; we refer to the latter as the schema-based semantics. We also refer to spanners that are defined via the schema-based semantics as schema-based spanners.

5.1.1 Regex Formulas

Recall the syntactic definition of regex formulas:

$$\alpha ::= \emptyset \mid \varepsilon \mid \sigma \mid \alpha \lor \alpha \mid \alpha \cdot \alpha \mid \alpha^* \mid x\{\alpha\}$$
where $\epsilon$ stands for the empty string, $\sigma \in \Sigma$, and the extension to regular expressions is $x\{\alpha\}$ where $x$ is a variable in $\text{Vars}$.

Following Maturana et al. [55], we interpret regex formulas as schemaless spanners in the following manner. The following sets define the application of a regex formula $\alpha$ on a document $d = \sigma_1 \cdots \sigma_n$, where the result is a pair $(s, \mu)$ where $s$ is a span of $d$ and $\mu$ is a mapping to $d$.

- $[\emptyset](d) := \emptyset$;
- $[\epsilon](d) := \{([i, i], \emptyset) \mid i = 1, \ldots, n\}$;
- $[\sigma](d) := \{([i, i + 1], \emptyset) \mid \sigma_i = \sigma\}$;
- $[x\{\alpha\}](d) := \{([i, j], \mu \cup \{x \mapsto [i, j]\}) \mid ([i, j], \mu) \in [\alpha](d) \text{ and } x \not\in \text{dom}(\mu)\}$;
- $[\alpha_1 \vee \alpha_2](d) := [\alpha_1](d) \cup [\alpha_2](d)$;
- $[\alpha_1 \cdot \alpha_2](d) := \{([i, j], \mu_1 \cup \mu_2) \mid \exists i' \text{ s.t. } ([i, i'], \mu_1) \in [\alpha_1](d), ([i', j], \mu_2) \in [\alpha_2](d), \text{ and } \text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset\}$;
- $[\alpha^*](d) := \bigcup_{i=0}^{\infty} [\alpha^i](d)$ where $\alpha^i$ stands for the concatenation of $i$ copies of $\alpha$.

The result of applying the schemaless spanner represented by $\alpha$ to $d$ is then defined as follows.

$$\llbracket \alpha \rrbracket (d) = \{\mu \mid ([1, |d| + 1], \mu) \in [\alpha](d)\}$$

**Syntactic Restrictions** Up to now we discussed the class of functional regex formulas (defined in Chapter 2) and denoted it by $\text{RGX}$. Since from now on we also consider more general regex formulas, we change the notation and denote the class of functional regex formulas by $\text{funcRGX}$. For the class $\text{funcRGX}$ of functional regex formulas, the schema-based and schemaless semantics definitions coincide. That is, if $\alpha \in \text{funcRGX}$ then $[\alpha] = \llbracket \alpha \rrbracket$.

Maturana et al. [55] pointed at a wider fragment of regex formulas, namely the sequential regex formulas, that has some desirable properties, as will be discussed later. A regex formula $\alpha$ is sequential if the following conditions hold:

- For every sub-formula $\alpha_1 \cdot \alpha_2$, we have $\text{Vars}(\alpha_1) \cap \text{Vars}(\alpha_2) = \emptyset$.
- For every sub-formula $\alpha^*$, we have $\text{Vars}(\alpha) = \emptyset$.
- For every sub-formula $x\{\alpha\}$, we have $x \not\in \text{Vars}(\alpha)$.

We denote by $\text{seqRGX}$ the class of sequential regex formulas.

As shown by Maturana et al. [55], every functional regex formula is sequential, but some sequential regex formulas are not functional as the following example illustrates.

---

1We added this restriction to the original definition [55] since it was mistakenly omitted, as the authors confirmed.
Example 5.1.2. Let us define the following regex formulas over the alphabet $\Gamma \cup \Delta$ from Example 5.1.1:

- $\alpha_{\text{mail}} := x_{\text{mail}}^\beta \beta \alpha$
- $\alpha_{\text{name}} := (x_{\text{first}}^\gamma \cup x_{\text{last}}^\gamma) \cup (x_{\text{first}}^\gamma)$
- $\alpha_{\text{phone}} := x_{\text{phone}}^\gamma (0 \cup \cdots \cup 9)^+$

where $\beta := (a \vee \cdots \vee z)^+$, and $\gamma := (A \vee \cdots \vee Z) \cdot (a \vee \cdots \vee z)^*$. Based on the previous regex formulas, we define the regex formula that represents the schemaless spanner $P_{\text{StudInfo}}$ from Example 5.1.1:

$$\alpha_{\text{info}} := ((\Gamma \cup \Delta)^* \cdot \epsilon) \cdot \alpha_{\text{name}} \cdot \epsilon \cdot (\alpha_{\text{phone}} \cdot \epsilon \vee \epsilon) \cdot \alpha_{\text{mail}} \cdot \epsilon \cdot (\Gamma \cup \Delta)^*$$

Note that $\alpha_{\text{info}}$ is sequential but not functional since the variables $x_{\text{first}}$ and $x_{\text{phone}}$ are optional.

For $R \in \{\text{funcRGX}, \text{seqRGX}\}$, we denote by $\llbracket R \rrbracket$ the class of schemaless spanners that can be expressed using regex formulas from $R$.

5.1.2 Vset-Automata

In addition to regex formulas, we use the vset-automata for representing schemaless spanners. We slightly change our definition from Chapter 2 of vset-automata and present an equivalent definition as we explain shortly: a vset-automaton is a tuple $(V, Q, q_0, F, \delta)$ where $V$ is a finite set of variables, $Q$ is the set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, and $\delta : Q \times (\Sigma \cup \{\epsilon\} \cup \Gamma_V) \rightarrow 2^Q$ is the transition function. This definition deviates from that in Chapter 2 since we replaced the final state $q_f$ with a set $F$ of accepting states. As we commented, this change does not affect the expressive power since we can simulate a multiple accepting states automaton by a single accepting state automaton with transitions of the form $(q, \epsilon, p)$. We say that the transition relation $\delta$ consists of epsilon transitions of the form $(q, \epsilon, p)$, letter transitions of the form $(q, \sigma, p)$ and variable transitions of the form $(q, v^+, p)$ or $(q, \neg v^+, p)$ where $q, p \in Q$, $\sigma \in \Sigma$, and $v \in \text{Vars}$.

Recall that in Chapter 2 we used the notion of ref-words to define the semantics of vset-automata. Here, in contrast, we define the schemaless semantics of vset-automata via the notion of runs: A run $\rho$ over a document $d := \sigma_1 \cdots \sigma_n$ is a sequence of the form

$$(q_0, i_0) \overset{o_1}{\rightarrow} \cdots (q_{m-1}, i_{m-1}) \overset{\sigma_m}{\rightarrow} (q_m, i_m)$$

where:

- the $i_j$ are indexes in $\{1, \ldots, n + 1\}$ such that $i_0 = 1$ and $i_m = n + 1$;
- each $o_j$ is in $\Sigma \cup \{\epsilon\} \cup \Gamma_{\text{Vars}(A)}$;
- $i_{j+1} = i_j$ whenever $o_j \in \Gamma_{\text{Vars}(A)}$, and $i_{j+1} = i_j + 1$ otherwise;
• for all \( j > 0 \) we have \((q_{j-1}, o_j, q_j) \in \delta\).

A run \( \rho \) is called valid if for every variable \( v \) the following hold:

• \( v \) is opened (or closed) at most once;

• if \( v \) is opened at some position \( i \) then it is closed at some position \( j \) with \( i \leq j \);

• if \( v \) is closed at some position \( j \) then it is opened at some position \( i \) with \( i \leq j \).

A run is called accepting if its last state is an accepting state (i.e., \( q_m \in F \)). For an accepting and valid run \( \rho \), we define \( \mu_\rho \) to be the mapping that maps the variable \( v \) to the span \([i_j, i_{j'}] \) where \( o_{i_j} = v^+ \) and \( o_{i_{j'}} = \neg v \).

The result \( \llbracket A \rrbracket(d) \) of applying the schemaless spanner represented by \( A \) on a document \( d \) is defined as the set of all assignments \( \mu_\rho \) for all valid and accepting runs \( \rho \) of \( A \) on \( d \).

We call a vset-automaton sequential if all of its accepting runs are valid. Recall that we already defined a functional vset-automaton in Chapter 2 through the notion of ref-words. An equivalent definition is the following: a vset-automaton is functional if each of its accepting runs is valid and it also includes all of its variables \( \text{Vars}(A) \). Note that for a functional vset-automaton \( A \) it holds that the schemaless and schema-based spanners represented by \( A \) are equivalent. That is, \( \llbracket A \rrbracket \equiv \llbracket A \rrbracket \) whenever \( A \) is functional.

Example 5.1.3. Let \( A \) be the following sequential vset-automaton:

\[
\begin{array}{c}
\Sigma \\
\xrightarrow{x^+} q_0 \\
\xrightarrow{-x} q_1 \\
\xrightarrow{\Sigma} q_2 \\
\end{array}
\]

Omitting the transition from \( q_0 \) to \( q_2 \) results in a functional vset-automaton. The same schemaless spanner as that represented by \( A \) is given by the sequential regex formula \( \alpha := (\Sigma^* x \{\Sigma^*\} \Sigma^*) \lor (\Sigma^+) \).

5.1.3 Schemaless Spanner Algebra

Before we define the algebra over schemaless spanners, we present some basic definitions. Two mappings \( \mu_1 \) and \( \mu_2 \) are compatible if they agree on every common variable, that is, \( \mu_1(x) = \mu_2(x) \) for all \( x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2) \). In this case, we define \( \mu := \mu_1 \cup \mu_2 \) as the mapping with \( \text{dom}(\mu) = \text{dom}(\mu_1) \cup \text{dom}(\mu_2) \) such that \( \mu(x) = \mu_1(x) \) for all \( x \in \text{dom}(\mu_1) \) and \( \mu(x) = \mu_2(x) \) for \( x \in \text{dom}(\mu_2) \).

The analog of relational algebra operators are defined similarly to the operators of the SPARQL1.0 formalism [47]. In particular, the operators union, projection, natural join, and difference are defined as follows for all schemaless spanners \( P_1 \) and \( P_2 \) and documents \( d \).
Union: The union $P := P_1 \cup P_2$ is defined by $P(d) := P_1(d) \cup P_2(d)$.

Projection: For $Y \subseteq \text{Vars}$, the projection $P := \pi_Y P_1$ is defined by $P(d) = \{\mu|_Y \mid \mu \in P_1(d)\}$ where $\mu|_Y$ is the restriction of $\mu$ to the variables in $\text{dom}(\mu) \cap Y$.

Natural join: The (natural) join $P := P_1 \bowtie P_2$ is defined to be such that $P(d)$ consists of all mappings $\mu_1 \cup \mu_2$ such that $\mu_1 \in P_1(d)$, $\mu_2 \in P_2(d)$ and $\mu_1$ and $\mu_2$ are compatible.

Difference: The difference $P := P_1 \setminus P_2$ is defined to be such that $P(d)$ consists of all mappings $\mu_1 \in P_1(d)$ such that no $\mu_2 \in P_2(d)$ is compatible with $\mu_1$.

String-equality selection: For variables $x$ and $y$ in $\text{Vars}$, the string-equality selection $P := \zeta_{x,y} P_1$ is defined to be such that $P(d)$ consists of all mappings $\mu_1 \in P_1(d)$ such that $x, y \in \text{dom}(\mu_1)$ and $d_{\mu_1(x)} = d_{\mu_1(y)}$.

We make the clear note that when the above operators are applied on schema-based spanners, they are the same as those defined in Chapter 2.

Example 5.1.4. Let us consider our input document $d_{\text{Students}}$ from Figure 5.1. Assume one wants to filter out from the results obtained by applying the spanner $P_{\text{StudInfo}}$ from Example 5.1.2 on $d_{\text{Students}}$ the mappings that correspond with students from universities within the UK. We consider that students study in the UK if and only if their email addresses end with the letters ‘uk’. We phrase the following regex formula that extracts such email addresses:

$$\alpha_{\text{UKm}} := \left( ((\Gamma \cup \Delta)^* \cdot \leftrightarrow \right) \lor \epsilon \cdot \Gamma^* \cdot x_{\text{mail}} \{ \beta @ \beta.\text{uk} \} \cdot \leftrightarrow \cdot (\Gamma \cup \Delta]^*$$

where $\beta$ is as defined in Example 5.1.2. In this case, the desired output is given by $[\alpha_{\text{info}} \setminus \alpha_{\text{UKm}}](d_{\text{Students}})$ which consists of the mappings $\mu_1$ and $\mu_2$ from Example 5.1.1.

5.1.4 Complexity

Let $\mathcal{L}$ be a representation language for schemaless spanners (e.g., the class of regex formulas or the class of vset-automata). Given $q \in \mathcal{L}$ and a document $d$, we are interested in the decision problem that checks whether $[q](d)$ is not empty. In that case, we are also interested in evaluating $[q](d)$. Recall that we study the combined complexity of these problems, as both $q$ and $d$ are regarded as input.

While deciding whether $[q](d) \neq \emptyset$ is NP-hard whenever $q$ is given as a vset-automaton [32], this is not the case for sequential (and hence functional) vset-automaton:

Theorem 5.1.1. [30] Given a sequential vset-automaton $A$ and a document $d$, one can enumerate $[A](d)$ with polynomial delay.
We call two schemaless spanner representations \( q_1 \) and \( q_2 \) equivalent if \( \llbracket q_1 \rrbracket \) and \( \llbracket q_2 \rrbracket \) are equal. Note that in Lemma 4.4.1 we showed that the translation of a functional regex formula to an equivalent functional vset-automata can be done in linear time. It also holds that the translation of sequential regex formulas to equivalent sequential vset-automata can also be achieved in linear time \[55\]. Hence, our lower bounds are usually shown for the nonemptiness of regex formulas and our upper bounds for the evaluation of vset-automata.

5.2 The Natural-Join Operator

In Chapter 4, we established complexity upper bounds on the evaluation of schema-based spanners by static compilation wherein we compiled the query (where the operands are regex formulas or vset-automata) into a single vset-automaton. In particular, we showed that two functional vset-automata can be compiled in polynomial time into a single equivalent vset-automaton that is also functional. Consequently, we concluded that we can enumerate with polynomial delay the mappings of \( \llbracket A_1 \bowtie A_2 \rrbracket (d) \) for functional vset-automata \( A_1 \) and \( A_2 \). The question is whether it generalizes to schemaless spanners: can we efficiently enumerate the mappings of \( \llbracket A_1 \bowtie A_2 \rrbracket (d) \), given sequential (but not necessarily functional) \( A_1 \) and \( A_2 \)? This is no longer the case, as the next theorem implies, even under the yardstick of expression complexity \[77\] in which the document is regarded as fixed. (Recall that a sequential regex formula can be translated in polynomial time into an equivalent vset-automaton \[55\].)

Theorem 5.2.1. The following problem is NP-complete. Given two sequential regex formulas \( \gamma_1 \) and \( \gamma_2 \) and an input document \( d \), is \( \llbracket \gamma_1 \bowtie \gamma_2 \rrbracket (d) \) nonempty? The problem remains NP-hard even if \( d \) is assumed to be of length one.

Proof. Membership in NP is straightforward, so we focus on NP-hardness. We use a reduction from 3CNF-satisfiability as in the proof of Theorem 4.6.1. Here, however, we are no longer restricted to functional regex formulas and therefore we can use the domains of the resulting mappings to encode the assignments. Recall that the input is a formula \( \varphi \) with the free variables \( x_1, \ldots, x_n \) such that \( \varphi \) has the form \( C_1 \land \cdots \land C_m \), where each \( C_i \) is a clause. In turn, each clause is a disjunction of three literals, where a literal has the form \( x_i \) or \( \neg x_i \).

Given a 3CNF formula \( \varphi \), we construct two sequential regex formulas \( \gamma_1 \) and \( \gamma_2 \) such that there is a satisfying assignment for \( \varphi \) if and only if \( \llbracket \gamma_1 \bowtie \gamma_2 \rrbracket (d) \neq \emptyset \), where \( d \) is the document that consists of a single letter \( a \).

To construct \( \gamma_1 \) and \( \gamma_2 \), we associate every variable \( x_i \) with 2\( m \) corresponding capture variables \( x_i^1, \ldots, x_i^m \) for \( 1 \leq j \leq m \) and \( \ell \in \{\text{true}, \text{false}\} \). We then define

\[
\gamma_1 := \gamma_{x_1} \cdots \gamma_{x_n} \cdot a,
\]

\[
\gamma_{x_i} := (x_i^1.\text{true} \{\epsilon\} \cdots x_i^m.\text{true} \{\epsilon\}) \lor (x_i^1.\text{false} \{\epsilon\} \cdots x_i^m.\text{false} \{\epsilon\}).
\]

Intuitively, \( \gamma_{x_i} \) verifies that the assignment to \( x_i \) is consistent in all of the clauses. We then define

\[
\gamma_2 := a \cdot (\delta_1 \cdots \delta_m)
\]
where $\delta_j$ is the disjunction of regex formulas $\beta$ such that $\beta = x_i^{j,\text{false}} \{\epsilon\}$ if $\neg x_i$ appears in $C_j$, and $\beta = x_i^{j,\text{true}} \{\epsilon\}$ if $x_i$ appears in $C_j$. Intuitively, $\gamma_2$ verifies that at least one disjunct in each clause is evaluate to true.

To emphasize the differences between this reduction and that in the proof of Theorem 4.6.1, we use the same $\varphi$ as an example:

$$\varphi := (x \lor y \lor z) \land (\neg x \lor y \lor \neg z).$$

In this case, we have

$$\delta_1 = x^{1,\text{true}} \{\epsilon\} \lor y^{1,\text{true}} \{\epsilon\} \lor z^{1,\text{true}} \{\epsilon\}$$

and, therefore,

$$\gamma_2 := a \cdot (x^{1,\text{true}} \{\epsilon\} \lor y^{1,\text{true}} \{\epsilon\} \lor z^{1,\text{true}} \{\epsilon\})$$

and $\varphi := (x^{1,\text{false}} \{\epsilon\} \lor y^{1,\text{true}} \{\epsilon\} \lor z^{1,\text{false}} \{\epsilon\})$.

We also have

$$\delta_1 = x^{1,\text{true}} \{\epsilon\} \lor y^{1,\text{true}} \{\epsilon\} \lor z^{1,\text{true}} \{\epsilon\}$$

and, therefore,

$$\gamma_2 := a \cdot (x^{1,\text{false}} \{\epsilon\} \lor y^{1,\text{false}} \{\epsilon\} \lor z^{1,\text{false}} \{\epsilon\}).$$

It follows directly from the definition that both $\gamma_1$ and $\gamma_2$ are sequential. Moreover, $\llbracket \gamma_1 \Join \gamma_2 \rrbracket (d)$ is nonempty if and only if there are compatible mappings $\mu_1 \in \llbracket \gamma_1 \rrbracket (d)$ and $\mu_2 \in \llbracket \gamma_2 \rrbracket (d)$. Since $\gamma_1$ ends with the letter $a$ whereas $\gamma_2$ starts with the letter $a$, it holds that $\mu_1 \in \llbracket \gamma_1 \rrbracket (d)$ and $\mu_2 \in \llbracket \gamma_2 \rrbracket (d)$ are compatible if and only if $\text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset$. We will show that $\llbracket \gamma_1 \Join \gamma_2 \rrbracket (d)$ is nonempty if and only if there is a satisfying assignment to $\varphi$.

The “only if” direction Suppose that $\llbracket \gamma_1 \Join \gamma_2 \rrbracket (d)$ is nonempty. In this case, a satisfying assignment $\tau$ to $\varphi$ is encoded by the domain of $\gamma_2$ in the following way: if $x_i^{j,\ell} \in \text{dom}(\mu_2)$ then $\tau(x_i) = \ell$; otherwise, $\tau(x_i) = \text{true}$. Note that if $x_i^{j,\ell} \in \text{dom}(\mu_2)$ then for every $j'$ it holds that $x_i^{j',\ell} \in \text{dom}(\mu_2)$. Note also that if both $x_i^{j,\text{true}} \not\in \text{dom}(\mu_2)$ and $x_i^{j,\text{false}} \not\in \text{dom}(\mu_2)$ then it is the case $x_i$ does not appear in $\varphi$ (and hence its assignment does not affect satisfiability). We can thus conclude that $\tau$ is well-defined.

In our example, the mapping $\mu_1 \in \llbracket \gamma_1 \rrbracket (a)$ with

$$\text{dom}(\mu_1) = \{x^{1,\text{true}}, x^{2,\text{true}}, y^{1,\text{false}}, y^{2,\text{false}}, z^{1,\text{false}}, z^{2,\text{false}}\}$$

and the mapping $\mu_2 \in \llbracket \gamma_2 \rrbracket (a)$ with

$$\text{dom}(\mu_2) = \{x^{1,\text{false}}, x^{2,\text{false}}, y^{1,\text{true}}, y^{2,\text{true}}, z^{1,\text{true}}, z^{2,\text{true}}\}$$

are compatible, and the satisfying assignment $\tau$ is encoded by $\text{dom}(\mu_2)$ and is given by $\tau(x) = \text{false}$, $\tau(y) = \text{true}$ and $\tau(z) = \text{true}$.
The “if” direction If there is a satisfying assignment \( \tau \) to \( \varphi \), then define the mappings \( \mu_1 \in \models \gamma_1(d) \) and \( \mu_2 \in \models \gamma_2(d) \) by \( x_i^{\mu_1} \in \text{dom}(\mu_2) \) whenever \( j = \tau(x_i) \) and \( x_i^{\mu_2} \in \text{dom}(\mu_1) \) whenever \( j \neq \tau(x_i) \). These mappings are compatible, since \( \text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset \). We conclude that \( \models \gamma_1 \bowtie \gamma_2(d) \) is nonempty.

We conclude the NP-hardness of determining whether \( \models \gamma_1 \bowtie \gamma_2(d) \) is nonempty, as claimed.

In what follows, we suggest two different approaches to deal with this hardness.

5.2.1 Bounded Number of Shared Variables

We now consider the task of computing \( \models A_1 \bowtie A_2(d) \), given sequential vset-automata \( A_1 \) and \( A_2 \) and a document \( d \). Next, we show that compiling the join into a new sequential vset-automaton is Fixed Parameter Tractable (FPT) when the parameter is the number of common variables.

Lemma 5.2.2. The following problem is FPT when parametrized by \( |\text{Vars}(A_1) \cap \text{Vars}(A_2)| \). Given two sequential vset-automata \( A_1 \) and \( A_2 \), construct a sequential vset-automaton that is equivalent to \( A_1 \bowtie A_2 \).

Since we have a polynomial delay algorithm for the evaluation of sequential vset-automata (Theorem 5.1.1) and the size of the resulting vset-automaton is FPT in \( |\text{Vars}(A_1) \cap \text{Vars}(A_2)| \), we have the following immediate conclusion.

Theorem 5.2.3. Given two sequential vset-automata \( A_1 \) and \( A_2 \) and a document \( d \), one can evaluate \( \models A_1 \bowtie A_2(d) \) with FPT delay parameterized by \( |\text{Vars}(A_1) \cap \text{Vars}(A_2)| \).

Proof of Lemma 5.2.2

We showed in Section 4.3 that if \( A \) is a functional vset-automaton then for every state \( q \) of \( A \) and every variable \( v \in \text{Vars}(A) \), all of the possible runs from the initial state \( q_0 \) to \( q \) include the same variable operations. Formally, for every state \( q \) there is a function \( c_q \), namely the variable configuration function, that assigns a label from \{o,c,w\}, standing for “open,” “close,” and “wait,” to every variable in \text{Vars}(A), as follows. First, \( c_q(x) = o \) if every run from \( q_0 \) to \( q \) opens \( x \) but does not close it. Second, \( c_q(x) = c \) if every run from \( q_0 \) to \( q \) opens and closes \( x \). Third, \( c_q(x) = w \) if no run from \( q_0 \) to \( q \) opens or closes variable \( x \).

In sequential vset-automata, however, not all of the accepting runs open and close all of the variables and therefore it makes more sense to replace the label \( w \) with the label \( u \) that stands for “unseen”. In addition, in sequential vset-automata as opposed to functional, there might be a state \( q \) for which there are two (different) runs from \( q_0 \) to \( q \) such that the first opens and closes the variable \( x \) whereas the second does not even open \( x \). For this case, we add to the set of labels the label \( d \) that stands for “done” meaning that variable \( x \) cannot be seen after reaching state \( q \). Hence, “done” can also be understood as “unseen or closed, depending on what happened before”. We formalize these notions right after the next example.
Example 5.2.1. Let us examine the following two accepting runs of the sequential vset-automaton $A$ from Example 5.1.3 on the input document $d := a$:

\[
\rho_1 := (q_0, 1) \xrightarrow{a} (q_1, 1) \xrightarrow{a} (q_1, 2) \xrightarrow{x} (q_2, 2)
\]

\[
\rho_2 := (q_0, 1) \xrightarrow{a} (q_2)
\]

The run $\rho_1$ gets to state $q_2$ after opening and closing $x$ while $\rho_2$ gets to $q_2$ without opening $x$. Thus, in state $q_2$ the variable configuration of $x$ is $d$.

This “nondeterministic” behavior of sequential vset-automata is reflected in an extended variable configuration function $\tilde{c}_q$ for every state $q$ whose co-domain is the set $\{u, o, c, d\}$. Since all of the accepting runs of a sequential vset-automaton are valid, given a state $q$, exactly one of the following holds:

- all runs from $q_0$ to $q$ open $x$; in this case $\tilde{c}_q(x) = o$;
- all runs from $q_0$ to $q$ (open and) close $x$; in this case $\tilde{c}_q(x) = c$;
- all runs from $q_0$ to $q$ do not open $x$; in this case $\tilde{c}_q(x) = u$;
- at least one run from $q_0$ to $q$ (opens and) closes $x$ and at least one does not open $x$; in this case $\tilde{c}_q(x) = d$.

A sequential vset-automaton $A$ is semi-functional for $x$ if for every state $q$ it holds that $\tilde{c}_q(x) \in \{u, o, c\}$. We say that $A$ is semi-functional for $X$ if it is semi-functional for every $x \in X$.

Example 5.2.2. The sequential vset-automaton $A$ from Example 5.1.3 is not semi-functional for $x$ because $\tilde{c}_{q_2}(x) = d$, as reflected from the runs $\rho_1$ and $\rho_2$ presented in the previous example. However, the following equivalent sequential vset-automaton $A'$ is semi-functional for $x$:

Observe that the ambiguity we had in state $q_2$ of $A$ is resolved since it is replaced with two states, each corresponding to a unique configuration.

We show that for every sequential vset-automaton $A$, every state $q$ of $A$ and every variable $v$, we can compute $\tilde{c}_q(v)$ efficiently, and based on that we can translate $A$ into an equivalent sequential vset-automaton that is semi-functional for $X$. We show that the total runtime is FPT parameterized by $|X|$.

Lemma 5.2.4. Given a sequential vset-automaton $A$ with $n$ states and $m$ transitions, and $X \subseteq \text{Vars}(A)$, one can construct in $O(2^{|X|}(n + m))$ time a sequential vset-automaton $A'$ that is equivalent to $A$ and semi-functional for $X$.  

78
Proof. To prove this lemma, we prove the following argument:

- Let $A$ be a sequential vset-automaton that is semi-functional for $Y \subseteq \text{Vars}(A)$ and let $x \in \text{Vars}(A) \setminus Y$. There is an algorithm that outputs a sequential vset-automaton $A'$ that is equivalent to $A$ and semi-functional for $Y \cup \{x\}$ with $O(2|Q|)$ states and $O(2|\delta|)$ transitions in $O(2(|Q| + |\delta|))$ steps where $Q$ is the set of states of $A$ and $\delta$ is $A$’s transition function. The lemma follows directly from the above argument as we can invoke the variable configuration function $\hat{c}(x)$ with $A$, $q_0$, and $\delta$, $F$.

We denote by $\hat{Q}$ the subset of $Q$ that consists of those states $\hat{\bar{q}}$ for which $\hat{c}_A^\hat{\bar{q}}(x) = d$, these are the states that we wish to replace. We then define the vset-automaton $A' := (V, \hat{Q}, q_0', \delta', F')$ such that

1. $Q' = (Q \setminus \hat{Q}) \cup \{\hat{\bar{q}}|\hat{\bar{q}} \in \hat{Q}\} \cup \{\hat{\bar{q}}|\hat{\bar{q}} \in \hat{Q}\}$ and
2. if $q_0 \in \hat{Q}$ then $q_0' = q_0^u$, otherwise $q_0' = q_0$, and
3. $F' = \{q|q \in F \setminus \hat{Q}\} \cup \{q^c, q^u|q \in F \cap \hat{Q}\},$

and the transition function $\delta'$ is as described now. For every $(p, o, q) \in \delta$ with $\hat{c}_q^A(x) \neq d$, we set $(p, o, q) \in \delta'$. In addition, for every $(p, o, q) \in \delta$ with $\hat{c}_q^A(x) = d$,

- if $\hat{c}_q^A(x) = d$ then $(p^u, o, q^u) \in \delta'$ and $(p^c, o, q^c) \in \delta'$;
- if $\hat{c}_q^A(x) = o$ then $(p, o, q^c) \in \delta'$ (and in this case $o = -x$);
- if $\hat{c}_q^A(x) = c$ then $(p^c, o, q^c) \in \delta'$.

It is left to show that

1. $A'$ is equivalent to $A$ and
2. $A'$ is semi-functional for $Y \cup \{x\}$.

To prove (1) we show that every mapping in $\|A'\|(d)$ is also a mapping in $\|A\|(d)$ and vice versa. Indeed, for every accepting run of $A'$ there might be two cases: (a) it includes only states from $Q$ or (b) it includes also states in $Q' \setminus Q$. If (a) then the claim is straightforward as $\delta' \cap (Q \times (\Sigma \cup \{\epsilon\} \cup \Gamma_{\text{Vars}(A)}) \times Q)$ equals $\delta$. If (b) then we can divide the corresponding run according to the variable configuration of $x$ and the claim then is a direct consequence of the definition of

79
δ’. To show that every mapping in ∥A∥(d) is also a mapping in ∥A’∥(d) we take the corresponding map in A and divide it into segments according to the variable configuration of A, we can then use the definition of δ’ to construct an accepting run in A’ for the same mapping.

To prove (3), we first observe that A’ is semi-functional for Y (since otherwise it would imply that A is not semi-functional for Y). It is therefore left to show that it is semi-functional for {x}. Note every state q in A with \( c^A_q(x) \neq d \) has two corresponding copies \( q^c, q^u \) in A’. Thus, the definition of δ’ implies that for every state \( p \in \{ q^u | q \in \tilde{Q} \} \) it holds that \( \tilde{c}^A_p(x) = w \) and for every state \( p \in \{ q^c | q \in Q \} \) it holds that \( \tilde{c}^A_p(x) = c \). For all other states r the \( \tilde{c}^A_r \) is identical to \( \tilde{c}^A \).

**Example 5.2.3.** The sequential vset-automaton A’ from Example 5.2.2 can be obtained from the automaton A from Example 5.1.3 by replacing \( q_2 \) with two states \( q^c_2 \) and \( q^u_2 \) such that \( q^c_2 \) corresponds with the paths in A from \( q_0 \) to \( q_2 \) in which variable x was unseen and \( q^u_2 \) corresponds with the paths in A from \( q_0 \) to \( q_2 \) in which variable x was closed, and by changing the transitions accordingly. The algorithm from the previous Lemma generalizes this idea.

We refer the reader to the definition of vset-automata we presented in the beginning of this chapter and note that, as in the previous example, there are cases where in order to be semi-functional, a vset-automaton must have more than a single accepting state.

If two sequential vset-automata are semi-functional for their common variables, their join can be computed efficiently:

**Lemma 5.2.5.** Given two sequential vset-automata A₁ and A₂ that are semi-functional for \( \text{Vars}(A_1) \cap \text{Vars}(A_2) \) one can construct in polynomial time a sequential vset-automaton A that is semi-functional for \( \text{Vars}(A_1) \cap \text{Vars}(A_2) \) and equivalent to \( A_1 \bowtie A_2 \).

The proof of this Lemma uses the same product construction as that for functional vset-automata in the proof of Lemma 4.4.7. What allows us to use the same construction is (a) the fact it ignores the non-common variables and (b) the fact we can treat both \( A_1 \) and \( A_2 \) as functional vset-automata over \( \text{Vars}(A_1) \cap \text{Vars}(A_2) \).

We can now conclude the proof of Lemma 5.2.2. Given two sequential vset-automata \( A_1 \) and \( A_2 \), we invoke the algorithm from Lemma 5.2.4 and obtain two equivalent sequential vset-automata \( \tilde{A}_1 \) and \( \tilde{A}_2 \), respectively, such that each \( \tilde{A}_i \) is semi-functional for \( \text{Vars}(A_i) \cap \text{Vars}(A_2) \). Then, we use Lemma 5.2.5 to join \( \tilde{A}_1 \) and \( \tilde{A}_2 \). Note that the runtime is indeed FPT parametrized by \( \text{Vars}(A_1) \cap \text{Vars}(A_2) \).

### 5.2.2 Restricting to Disjunctive Functional

Another approach to obtain a tractable evaluation of the join is by restricting the syntax of the regex formulas while preserving expressiveness. A regex formula \( \gamma \) is said to be **disjunctive functional** if it is a finite disjunction of functional regex formulas \( \gamma_1, \ldots, \gamma_n \). We denote the class of disjunctive functional regex formulas as \( \text{dfuncRGX} \).
Note that every disjunctive functional regex formula is also sequential. However, the regex formula \( z\{\Sigma^*\} \cdot (x\{\Sigma^*\} \cup y\{\Sigma^*\}) \) is sequential, yet it is not disjunctive functional. It also holds that every functional regex formula is a disjunctive functional regex formula with a single disjunct. We can therefore conclude that we have the following:

\[
\text{funcRGX} \subseteq \text{dfuncRGX} \subseteq \text{seqRGX}
\]

Note that here we treat the regex formulas as syntactic objects.

Similarly to a disjunctive functional regex formula, a disjunctive functional vset-automaton \( A \) is the sequential vset-automaton whose states are the disjoint union of the states of a finite number \( n \) of functional vset-automata \( A_1, \ldots, A_n \) and whose transitions are those of \( A_1, \ldots, A_n \), with the addition of a new initial state \( q_0 \) that is connected with epsilon transitions to each of the initial states of the \( A_i \)'s.

We observe that being disjunctive functional is only a syntactic restriction and not a semantic one, based on the following proposition.

**Proposition 5.2.6.** The following hold:

1. For every sequential regex formula there exists an equivalent disjunctive functional regex formula.

2. For every sequential vset-automaton there exists an equivalent disjunctive functional vset-automaton.

However, as we point at in the following observation, this translation is generally exponential. We move to the proof of this proposition.

**Proof.** We prove the two parts of the proposition.

1. Let \( \alpha \) be a sequential regex formula. We translate it into an equivalent disjunctive functional by defining the set \( A(\alpha) \) of its disjuncts recursively as follows:
   
   - if \( \alpha = \emptyset \) then \( A(\alpha) = \emptyset \);
   - if \( \alpha = \sigma \) then \( A(\alpha) = \{\sigma\} \);
   - if \( \alpha = \epsilon \) then \( A(\alpha) = \{\epsilon\} \);
   - if \( \alpha = \alpha_1 \lor \alpha_2 \) then
     - if \( \text{Vars}(\alpha_1) = \text{Vars}(\alpha_2) = \emptyset \) then \( A(\alpha) = \{\alpha_1 \lor \alpha_2\} \)
     - otherwise \( A(\alpha) = A(\alpha_1) \cup A(\alpha_2) \);
   - if \( \alpha = \alpha_1 \cdot \alpha_2 \) then \( A(\alpha) = \{\beta_1 \cdot \beta_2 | \beta_1 \in A(\alpha_1), \beta_2 \in A(\alpha_2)\} \);
   - if \( \alpha = (\alpha_1)^* \) then \( \text{Vars}(\alpha_1) = \emptyset \) and \( A(\alpha) = \{\alpha_1^*\} \);
   - if \( \alpha = x\{\alpha_1\} \) then \( A(\alpha) = \{x\beta | \beta \in A(\alpha_1)\} \)

We can then conclude the desired claim by proving the following argument using a simple induction on \( \alpha \)'s structure.
• If $\alpha$ is a sequential regex formula then $\|\alpha\| = \bigvee_{\gamma \in A(\alpha)} \|\gamma\|$ where each $\gamma$ is functional. In addition, $A(\alpha)$ is finite if $\alpha$ is finite.

2. Let $A$ be a sequential automaton with set of variables $\text{Vars}(A)$. We iterate through all of the possible subsets $V$ of $\text{Vars}(A)$ and for each such subset we create a new vset-automaton $A_V$ that consists of all of the accepting runs of $A$ that include exactly the variables of $V$ and only them (this can be done, for instance, by a BFS on $A$ starting from its initial state $q_0$). We then construct a new automaton $A'$ that is the disjoint union of all of those $A_V$’s by adding a new initial state with epsilon transitions to all of the initial states of the $A_V$’s.

Since $\text{funcRGX}$ corresponds to schema-based spanners whereas $\text{seqRGX}$ with schemaless and due to the previous proposition we can conclude the following:

$$\|\text{funcRGX}\| \subseteq \|\text{dfuncRGX}\| = \|\text{seqRGX}\|$$

Note that here we refer to the schemaless spanners represented by the regex formulas (and not to the syntactic expressions).

**Example 5.2.4.** Consider the following sequential regex formula:

$$(x_1\{\Sigma^*\} \lor y_1\{\Sigma^*\}) \cdots (x_n\{\Sigma^*\} \lor y_n\{\Sigma^*\})$$

Note that if we want to translate it into an equivalent disjunctive functional regex formula then we need at least one disjunct for each possible combination $z_1\{\Sigma^*\} \cdots z_n\{\Sigma^*\}$ where $z_i \in \{x_i,y_i\}$. This implies a lower bound on the length of the shortest equivalent disjunctive functional regex formula. Similarly, let us consider the following sequential vset-automaton:

An equivalent disjunctive functional vset-automaton has at least $2^n$ accepting states since the states encode the variable configurations.

We record this in the following observation.

**Observation 5.2.7.** For every natural number $n$ the following hold:

1. There exists a sequential regex formula $\gamma$ that is the concatenation of $n$ regex formulas of constant length such that each of its equivalent disjunctive functional regex formulas includes at least $2^n$ disjuncts.
2. There exists a sequential vset-automaton $A$ with $3n+1$ states such that each of its equivalent disjunctive functional vset-automaton has at least $2^n$ states.

That is, the translation from sequential to disjunctive functional might necessitate an exponential blow-up. Although the translation cannot be done efficiently in the general case, the advantage of using disjunctive functional vset-automata lies in the fact that we can compile the join of two disjunctive functional vset-automata efficiently into a disjunctive functional vset-automaton.

**Proposition 5.2.8.** Given two disjunctive functional vset-automata $A_1$ and $A_2$, one can construct in polynomial time a disjunctive functional vset-automaton $A$ that is equivalent to $A_1 \bowtie A_2$.

To prove this we can perform a pairwise join between the set of functional components of $A_1$ and those of $A_2$ and obtain a set of functional vset-automata for the join (Lemma 4.4.7).

Since disjunctive functional is a restricted type of sequential vset-automaton, we conclude the following.

**Corollary 5.2.9.** Given two disjunctive functional vset-automata $A_1$ and $A_2$ and a input document $d$, one can enumerate the mappings of $\|A_1 \bowtie A_2\|(d)$ in polynomial delay.

### 5.3 The Difference Operator

From Theorem 4.6.1 we conclude that, in contrast to the tractability of the natural join of disjunctive functional vset-automata (Corollary 5.2.9), here we are facing NP-hardness already for functional vset-automata. In the remainder of this section, we discuss syntactic conditions that allow to avoid this hardness.

#### 5.3.1 Bounded Number of Common Variables

Theorem 4.6.1 implies that no matter what approach we choose to tackle the evaluation of the difference operator, without imposing restrictions we hit NP-hardness. In this section, we investigate the restriction of an upper bound on the number of common variables shared between the operands. Recall that this restriction leads to an FPT static compilation for the natural join (Lemma 5.2.2). Yet, in the case of difference, such static compilation necessitates an exponential blow-up, even if there are no variables at all.

Therefore, instead of static compilation that is independent of the document, we apply dynamic compilation that depends on the specific document at hand. In this case, we refer to the resulting automaton as an ad-hoc vset-automaton since it is valid only for that specific document.

**Lemma 5.3.1.** Let $k$ be a fixed natural number. Given two sequential vset-automata $A_1$ and $A_2$ where $|\text{Vars}(A_1) \cap \text{Vars}(A_2)| \leq k$ and a document $d$, one can construct in polynomial time a sequential vset-automaton $A_d$ with $\|A_d\|(d) = \|A_1 \setminus A_2\|(d)$.
We present the proof of this lemma in Section 5.3.2. By Theorem 5.1.1, we can enumerate the results of a sequential vset-automaton in polynomial time. We can conclude the following.

**Theorem 5.3.2.** Let \( k \) be a fixed natural number. Given two sequential vset-automata \( A_1 \) and \( A_2 \) where \( |\text{Vars}(A_1) \cap \text{Vars}(A_2)| \leq k \) and a document \( d \), one can enumerate \( \|A_1 \setminus A_2\|(d) \) with polynomial delay.

Theorem 5.3.2 shows that we can enumerate the difference with polynomial delay when we restrict the number of common variables. A natural question is whether the degree of this polynomial depends on this number; the next theorem answers this question positively, under the conventional assumptions of parameterized complexity.

**Theorem 5.3.3.** The following problem is \( \text{W}[1] \)-hard parametrized by \( |\text{Vars}(\gamma_1) \cap \text{Vars}(\gamma_2)| \). Given two functional regex formulas \( \gamma_1 \) and \( \gamma_2 \) and an input document \( d \), is \( \|\gamma_1 \setminus \gamma_2\|(d) \) nonempty?

Note that while Theorem 4.6.1 deals with difference in the schema-based semantics, here we deal with difference in the schemaless semantics. Note also that the this theorem contrasts our FPT result for the natural join (Theorem 5.2.3).

The proof uses a reduction from the problem of determining whether a 3CNF formula has a satisfying assignment with at most \( k \) ones, where \( k \) is the parameter.

*Proof.* Let \( \varphi = C_1 \land \cdots \land C_m \) be a 3CNF formula with variables \( x_1, \ldots, x_n \) and denote by \( l_i, l_{i1}, l_{i2}, l_{i3} \) the literals of the \( i \)-th clause for \( 1 \leq i \leq m \). To show the reduction, we use the same idea of the reduction in the proof of Theorem 5.2.1.

Set \( d = s_1 \ldots s_n \), where every \( s_i \) is encoded by a string of length \( \ell := \lceil \log(n) \rceil \) over a binary alphabet. Let \( S = \{s_1, \ldots, s_n\} \). For \( S' \subseteq S \), we define the regex formula \( \alpha_{S'} \) as the disjunction of strings in \( S' \). We define the regex formula \( \alpha_1 \), using only the \( k \) variables \( V = \{y_1, \ldots, y_k\} \), as

\[
\alpha_1 = \alpha^*_S y_1 \{\alpha_S\} \alpha^*_S y_2 \{\alpha_S\} \ldots y_k \{\alpha_S\} \alpha^*_S.
\]

We next define for every \( 0 \leq i \leq m \) the regex \( \alpha_i \), such that \( \|\alpha_i\|(d) \) corresponds to all possible assignments of \( \varphi \) of weight \( k \), such that \( C_i \) is not satisfied. Let \( S_i \subseteq S \) such that \( s_j \in S_i \) iff \( x_j \) is contained in clause \( C_i \). Further let \( S_i^- \) denote the variables in \( S_i \) which appear negated in \( C_i \) (i.e., \( s_j \in S_i^- \) iff \( x_j \) is a negated variable in \( C_i \)) and define \( S_i^+ = S_i \setminus S_i^- \). Let \( \text{ind}(S_i^-) := \{1 \leq j \leq n \mid s_j \in S_i^-\} \) and similarly \( \text{ind}(S_i^+) := \{1 \leq j \leq n \mid s_j \in S_i^+\} \). In order to define \( \alpha_{C_i} \), we need to consider different cases on the number of positive and negative literals in \( C_i \).

- First, assume that \( |S_i^+| = 3 \) (i.e., \( C_i \) is a clause containing three positive variables). Then we set

\[
\alpha_{C_i} = \alpha^*_S y_1 \{\alpha_{S_i \setminus S_i^+}\} \alpha^*_S \ldots y_k \{\alpha_{S_i \setminus S_i^+}\} \alpha^*_S.
\]
• Next assume that $|S_i^+| = 2$ and thus there is some $1 \leq j \leq n$ such that 
\{j\} = \text{ind}(S_i^-)$. For every $1 \leq u \leq k$, we define
\[
\alpha(u)_{C_i} = \alpha^u_S y_1 \{\alpha_{S \setminus S_i^+} \} \cdots \alpha^u_S y_u \{s_j\} \\
\alpha^u_S y_{u+1} \{\alpha_{S \setminus S_i^+} \} \cdots \alpha^u_S y_k \{\alpha_{S \setminus S_i^+} \} \alpha^u_S.
\]
Note that the regex $\alpha(u)$ is built similarly to $\alpha$ in the first case, except for
the part $y_u \{s_j\}$ instead of $y_u \{S \setminus S_i^+\}$. We set $\alpha_C$, to be the disjunction
$\bigvee_{1 \leq u \leq k} \alpha(u)_{C_i}$.

• The case where $|S_i^+| = 1$ is defined similarly to the case where $|S_i^+| = 2$. Assume that
there are $1 \leq j_1 < j_2 \leq n$ such that $\{j_1, j_2\} = \text{ind}(S_i^-)$. Then for
every $1 \leq u_1 < u_2 \leq k$ we define
\[
\alpha(u_1, u_2)_{C_i} = \alpha^u_S y_1 \{\alpha_{S \setminus S_i^+} \} \cdots \alpha^u_S y_{u_1 \cdot 1} \{s_j\} \cdots \\
\alpha^u_S y_{u_2 \cdot 1} \{\alpha_{S \setminus S_i^+} \} \alpha^u_S y_{u_2 \cdot 2} \{s_j\} \cdots \alpha^u_S y_k \{\alpha_{S \setminus S_i^+} \} \alpha^u_S.
\]
and set $\alpha_C$ to be the disjunction $\bigvee_{1 \leq u_1 < u_2 \leq k} \alpha(u_1, u_2)_{C_i}$.

• If $|S_i^+| = 0$, then $\alpha_C$ is defined analogously to the case $|S_i^+| = 1$ but with
three indices $u_1, u_2, u_3$ instead of $u_1, u_2$.

To define $\alpha_2$, we set $\alpha_2 = \bigvee_{1 \leq i \leq m} \alpha_C$. Note that $|\alpha_2| \leq (m + 1) \cdot n^2 \cdot (m + 1)^3$.
It remains to show that $\varphi$ is satisfiable (with weight $k$) if $[\alpha_1 \wedge \alpha_2](d) \neq \emptyset$.

So let $\tau$ be a satisfying truth assignment of weight $k$. We claim that there
exists some $\mu \in [\alpha_1 \wedge \alpha_2](d)$, with $\mu(y_i) = [j_\ell + 1, j_\ell + 1 + \ell]$ for
every $1 \leq i \leq k$ and $1 \leq j \leq n$ such that $\tau(x_j) = \text{true}$, and $x_j$ is the $i$-th
such variable (i.e., among the variables $x_{j_1}, \ldots, x_{j_{k-1}}$, there are $i - 1$ that are set to true by $\tau$).
First consider the set $[\alpha_1](d)$. It is easy to see that $\mu' \in [\alpha_1](d)$ iff there
are $1 \leq j_1 < \ldots < j_k \leq n$ with $\mu' = \{y_1 \mapsto [j_1 + 1, j_1 + 1 + \ell], \ldots, y_k \mapsto [j_k + 1, j_k + 1 + \ell]\}$, hence $\mu \in [\alpha_1](d)$. To see that $\mu \not\in [\alpha_2](d)$, we need to
consider the set $[\alpha_2](d)$. By definition of $\alpha_2$, we have that $\mu' \in [\alpha_2](d)$ iff
there is some $1 \leq i \leq m$ such that $\mu' \in [\alpha_C](d)$. The containment $\mu' \in [\alpha_C](d)$
can be characterized as follows: There exist $j_1, \ldots, j_k \in \{1, \ldots, n\} \setminus \text{ind}(S_i^-)$
such that $1 \leq j_1 < \ldots < j_k \leq n$, $\text{ind}(S_i^-) \subset \{j_1, \ldots, j_k\}$ and $\mu' = \{y_i \mapsto [j_1 + 1, j_1 + 1 + \ell], \ldots, y_k \mapsto [j_k + 1, j_k + 1 + \ell]\}$.

For some $1 \leq i \leq n$, first assume that $C_i$ is a clause only containing positive
variables. Since $\tau$ is a satisfying assignment, there is some $i'$ such that and
$\tau(x_{i'}) = 1$. Hence $\mu(y_{i'}) = [j + 1, j + 1 + \ell]$ for some $1 \leq j \leq n$, and since
$i' \in \text{ind}(S_i^-)$ we have that $\mu \not\in [\alpha_2](d)$.

Fix some $1 \leq i \leq n$. Since $\tau$ is a satisfying assignment, $\tau(x_{i'}) = 1$ for some
$i' \in \text{ind}(S_i^+)$ or $\tau(x_{i'}) = 0$ for some $i' \in \text{ind}(S_i^-)$. In the first case, $j_1, \ldots, j_k \in \{1, \ldots, n\} \setminus \text{ind}(S_i^-)$ does not hold, and in the second case $\text{ind}(S_i^-) \subset \{j_1, \ldots, j_k\}$ is violated. Thus $\mu \not\in [\alpha_C](d)$ for every $1 \leq i \leq n$, and hence $\mu \not\in [\alpha_2](d)$.

For the other direction, let $\mu \in [\alpha_1 \wedge \alpha_2](d)$. Using the same arguments as
above, we can show that the truth assignment $\tau$ is a satisfying truth assignment of
\varphi with weight $k$, where $\tau(x_i) = 1$ iff $y_i = [j + 1, j + 1 + \ell]$ for some $1 \leq j \leq n$. \qed
5.3.2 Proof of Lemma 5.3.1

In this Section we present the proof of Lemma 5.3.1. Our goal is to construct a vset-automaton $A_d$ for which the conditions of the lemma hold. Note that if $\text{Vars}(A_2) = \emptyset$ (i.e., $A_2$ is Boolean) then there are two cases.

- $\|A_2\|(d) = \emptyset$ and then we set $A_d$ to be $A_1$.
- $\|A_2\|(d) \neq \emptyset$ and then $\|A_1 \setminus A_2\|(d) = \emptyset$ since any two mappings are compatible. In this case we set $A_d$ to be the Boolean vset-automaton that accepts precisely $\emptyset$.

Recall that determining whether $\|A_2\|(d) = \emptyset$ can be done in polynomial time (see Theorem 5.1.1). Thus, we can assume that $\text{Vars}(A_2)$ is not empty.

Second, we can assume that $A_2$ has no other variables except those that are in $A_1$ due to the following: $\|A_1 \setminus A_2\|(d) = \|A_1 \setminus \tau_{\text{Vars}(A_1)}A_2\|(d)$ since if a mapping $\mu_1 \in \|A_1\|(d)$ has a compatible mapping $\mu_2 \in \|A_2\|(d)$ then $\mu_1|_{\text{Vars}(A_1)}$ is also compatible for $\mu_1$ and is in $\|\tau_{\text{Vars}(A_1)}A_2\|(d)$. On the other hand, if a mapping $\mu_1 \in \|A_1\|(d)$ has a compatible mapping $\mu_2 \in \|\tau_{\text{Vars}(A_1)}A_2\|(d)$, then no matter how we extend $\mu_2$, as long as we extend it with variables that are not in $\text{Vars}(A_1)$, it remains compatible to that $\mu_1$. Thus, if a mapping $\mu_1 \in \|A_1\|(d)$ does not have a compatible mapping $\mu_2 \in \|A_2\|(d)$ then it also does not have such a mapping in $\tau_{\text{Vars}(A_1)}\|A_2\|(d)$. Therefore, we may assume that $\text{Vars}(A_1) \supseteq \text{Vars}(A_2)$ and we denote $\text{Vars}(A_2)$ by $V$.

Third, we can also assume that $A_1$ is semi-functional for $V$ where $V$ is defined as $\text{Vars}(A_2)$, since if it is not we can translate it into such in polynomial time (see Lemma 5.2.2).

We now move to the construction of sequential vset-automata $A$ and $B$ with fixed $|\text{Vars}(A)\cap\text{Vars}(B)|$ for which we will prove that $\|A \bowtie B\|(d) = \|A_1 \setminus A_2\|(d)$.

**Constructing $A$:** For every $X \subseteq V$ we define $X' = \{x' | x \in X\}$. We construct a vset-automaton $A$ that extends $A_1$ such that for every accepting run $\rho$ of $A_1$ that closes exactly the set $X$ of variables, there is a corresponding accepting run in $A$ that assigns the same values as $\rho$ to $X$, and in addition, assigns to each of the variables of $X'$ the span $[1, 1]$ (indicating that the variables of $X$ were closed throughout $\rho$) and to each of the variables in $V' \setminus X'$ the span $[|d| + 1, |d| + 1]$ (indicating that the variables of $V \setminus X$ were unseen throughout $\rho$).

More formally, the vset-automaton $A$ consists of the disjoint copies $A_1^X$ of $A_1$ for $X \subseteq V$, along with their transitions, of an initial state $q_0$, and of an accepting state $q_f$. We then extend $A$ as follows:

- for every $X$, we add a path from $q_0$ to the initial state of $A_1^X$ that opens and then immediately closes all of the variables in $X'$;
- for every $X$ and every accepting state $q$ of $A_1^X$ with $c_q(x) = c$ for every $x \in X$, we add a path from $q$ to the accepting state $q_f$ that opens and then immediately closes all of the variables in $V' \setminus X'$.
We then remove from $A$ all of the states that are not reachable throughout any accepting run and obtain a sequential vset-automaton.

Let $\mu$ be a mapping with $\text{dom}(\mu) \subseteq V$. A mapping $\mu'$ is called the marked extension of $\mu$ if the following hold:

- $\text{dom}(\mu') = \text{dom}(\mu) \cup V'$
- for every $x \in \text{dom}(\mu)$ it holds that $\mu'(x) = \mu(x)$;
- for every $x \notin \text{dom}(\mu)$ it holds that $\mu'(x') = [|d|+1, |d|+1)$.

We observe that the following holds:

**Lemma 5.3.4.** Let $\mu_1, \mu_2$ be two mappings with $\text{dom}(\mu_1) = \text{dom}(\mu_2) \subseteq V$ and let $\mu'_1$ be the marked extension of $\mu_1$ and $\mu'_2$ the marked extension of $\mu_2$. The mapping $\mu_1$ is compatible with $\mu_2$ if and only if $\mu'_1$ is compatible with $\mu'_2$.

**Proof.** Assume that $\mu_1$ is compatible with $\mu_2$. It is enough to show that $\mu'_1|_V$ and $\mu'_2|_V$ are compatible. This follows directly from the fact that $\text{dom}(\mu_1) = \text{dom}(\mu_2)$. On the other hand, assume that $\mu'_1$ is compatible with $\mu'_2$. Since $\mu'_1|_V$ is identical to $\mu_1$ and $\mu'_2|_V$ is identical to $\mu_2$ we conclude that $\mu_1$ is compatible with $\mu_2$. \qed

The following lemmas describe the connection between $A_1$ and $A$:

**Lemma 5.3.5.** $\mu_1 \in \llbracket A_1 \rrbracket(d)$ if and only if $\mu \in \llbracket A \rrbracket(d)$ where $\mu$ is the marked extension of $\mu_1$.

**Proof.** Assume that $\mu_1 \in \llbracket A_1 \rrbracket(d)$. Then there is an accepting run on $A_1$ on $d$ that corresponds with $\mu_1$. We build an accepting run of $A$ that corresponds with the marked extension $\mu$ of $\mu_1$ in the following way. We denote $X = \text{dom}(\mu_1)$. The run starts with a path from $q_0$ to the initial state of the copy $A^X_1$ that opens and then closes all of the variables in $X'$. We then continue with the accepting run of $A^X_1$ on $\mu_1$ (there is such a run since $A^X_1$ is a copy of $A_1$). Then from the accepting state $q$ that we reached, we continue with a path that opens and then closes all of the variables in $V' \setminus X'$. We can do that from the way we constructed $A$. Hence, we conclude that $\mu \in \llbracket A \rrbracket(d)$.

Assume that $\mu \in \llbracket A \rrbracket(d)$ is the marked extension of $\mu_1$. There is a run of $A$ on $d$ that corresponds with $\mu$. We take the sub-run on the copy of $A_1$ in $A$ and obtain an accepting run of $A$ on $d$ that corresponds with $\mu_1$. Thus, we can conclude that $\mu_1 \in \llbracket A_1 \rrbracket(d)$. \qed

**Constructing $B$:** We construct $B$ as follows: We compute all of the assignments in $\llbracket A_2 \rrbracket(d)$ and replace each of those with their marked extension to obtain the set $M$ of extended mappings. We then compute all of the possible extended mappings whose domain is contained in $V \cup V'$. For every such mapping if it is not in $M$ then we insert it to the set $M$.

We now define the vset-automaton $B$ as the disjoint union of all of the paths that correspond to the assignments in $\bar{M}$ along with an initial state $q_0$ and an accepting state $q$. For every path $P$ that corresponds with the assignment $\mu$ with $X := \text{dom}(\mu)$ we extend $B$ as follows:
we add a path that connects \( q_0 \) with the first state of \( P \) that opens and then immediately closes all of the variables in \( X' \);

- we add a path that connects the last state of \( P \) to the accepting state \( q \), that opens and then immediately closes all of the variables in \( Y' \) where \( Y = V \setminus X \).

Note that \( B \) is a sequential vset-automaton.

**Correctness:** The following lemma describes the connection between \( A_1, A_2, B \) and \( d \).

**Lemma 5.3.6.** A mapping \( \mu_1 \in \llbracket A_1 \rrbracket(d) \) has a compatible mapping \( \llbracket A_2 \rrbracket(d) \) if and only if the marked extension \( \mu \) of \( \mu_1 \) does not have a compatible mapping in \( \llbracket B \rrbracket(d) \).

**Proof.** Assume \( \mu_1 \in \llbracket A_1 \rrbracket(d) \) has a compatible mapping \( \mu_2 \in \llbracket A_2 \rrbracket(d) \) and assume by contradiction that \( \mu \) has a compatible mapping \( \mu'_2 \in \llbracket B \rrbracket(d) \). Then by Lemma 5.3.4, the restriction \( \mu'_2|_V \) is compatible with \( \mu_2 \) which is impossible due to \( B \)'s definition.

Assume that \( \mu_1 \in \llbracket A_1 \rrbracket(d) \) does not have a compatible mapping \( \mu_2 \in \llbracket A_2 \rrbracket(d) \). By Lemma 5.3.5, the extension \( \mu \) of \( \mu_1 \) is in \( \llbracket A \rrbracket(d) \). Lemma 5.3.4 and \( B \)'s definition imply that there is a mapping in \( \llbracket B \rrbracket(d) \) that is compatible with \( \mu \). \( \Box \)

We can conclude that \( \llbracket \pi_V (A \Join B) \rrbracket(d) = \llbracket A_1 \setminus A_2 \rrbracket(d) \)

Note that \( \text{Vars}(A) = \text{Vars}(B) = V \cup V' \) and since \( |V| \) is fixed this is also the case for \( \text{Vars}(A) \cap \text{Vars}(B) = \text{Vars}(A) \). Therefore, we can use Lemma 5.2.2 to conclude the proof.

**Complexity:** Note that throughout our construction we performed only polynomial time steps: In case the document is empty, checking the emptiness of \( \llbracket A_2 \rrbracket(d) \) can be done in polynomial time. Transforming \( A_1 \) into a semi-functional vset-automaton for \( V \) requires polynomial time assuming \( |V| \) is fixed. Constructing the extension \( A \) of \( A_1 \) requires polynomial time since we fix \( |V| \). Computing the set of extended mappings that are not compatible with any of the extended mappings that correspond with \( \llbracket A_2 \rrbracket(d) \) also requires polynomial time since \( |V| \) is fixed.

### 5.3.3 Restricting the Disjunctions

We now propose another restriction that guarantees a tractable evaluation, this time allowing the number of common variables to be unbounded. We begin with some definitions.

Let \( \gamma \) be a sequential regex formula and let \( x \in \text{Vars} \) be a variable. Then \( \gamma \) is synchronized for \( x \) if, for every subexpression of \( \gamma \) of the form \( \gamma_1 \lor \gamma_2 \), we have that \( x \) appears neither in \( \gamma_1 \) nor in \( \gamma_2 \). A regex formula \( \gamma \) is called synchronized for \( X \subseteq \text{Vars} \) if it is synchronized for every \( x \in X \).
This notion can also be defined to sequential vset-automata: A state \( q \) of a sequential vset-automaton \( A \) is called a unique target state for the variable operation \( \alpha \in \Gamma_{\text{Vars}(A)} \), if for every state \( p \) of \( A \) we have that \( (p, \omega, q) \in \delta \) implies \( q = q_\omega \) where \( \delta \) is the transition relation of \( A \). In other words, \( q_\omega \) is the only state that can be reached by processing \( \omega \). We say that \( A \) is synchronized for a variable \( x \in \text{Vars} \) if each of \( x^+ \) and \( \neg x \) has a unique target state and either all accepting runs of \( A \) open and close \( x \), or no accepting run of \( A \) operates on \( x \). Finally, \( A \) is synchronized for \( X \subseteq \text{Vars} \) if it is synchronized for every \( x \in X \).

**Example 5.3.1.** Consider the regex formula \( (x\{\Sigma^\ast\} \lor \epsilon) \cdot y\{\Sigma^\ast\} \) and this equivalent vset-automaton:

![Diagram](image_url)

Both are synchronized for \( y \) and not for \( x \): The regex formula has a subexpression of the form \( (x\{\Sigma^\ast\} \lor \epsilon) \), whereas the variable \( y \) does not appear under any disjunction. In the vset-automaton, although each variable operation has a unique target state, not all of the accepting runs include the variable operations \( x^+ \) and \( \neg x \) (as opposed to \( y^+ \) and \( \neg y \), which are included in every accepting run).

The following result states that conversions from regex formulas to vset-automata can preserve the property of being synchronized for \( X \).

**Lemma 5.3.7.** Let \( \gamma \) be a sequential regex formula that is synchronized for \( X \subseteq \text{Vars} \). One can convert \( \gamma \) in linear time into an equivalent sequential vset-automaton \( A \) that is synchronized for \( X \).

**Proof.** Similarly to the construction of Lemma 4.4.1, we obtain \( A \) using Thompson’s construction (cf. e.g., [15]) for converting a regular expression into an \( \epsilon \)-NFA, where we treat variable operations like symbols. More specifically, each occurrence of a variable binding \( x\{\gamma\} \) is interpreted like \( x^+ \cdot \gamma \cdot \neg x \).

As \( \gamma \) is sequential, \( A \) is also sequential. A feature of Thompson’s construction is that for every occurrence of a symbol the regular expression is converted into an initial state and an accepting state, where the former has a transition to the latter that is labeled with that symbol. All new transitions that enter to this sub-automaton pass through the initial state; the accepting state can only be reached through the original transition. This creates a new target state for every variable operation.

The other condition for vset-automata that are synchronizing for \( X \) follows directly from the fact that \( \gamma \) has no disjunctions over \( x \). Either \( \gamma \) never uses \( x \) (e.g., if \( \gamma = x\{\emptyset\} \); if \( \gamma = x\{\{a\} \emptyset \), or due to other uses of \( \emptyset \)); then the accepting runs of \( A \) also never operate on \( x \). Or \( \gamma \) always uses \( x \); then the same also holds for all accepting runs of \( A \).
As one might expect, vset-automata that are synchronized (for some non-empty set $X$ of variables) are less expressive than sequential or semi-functional vset-automata (that are defined in Section 5.2.1). In fact, even functional regex formulas can express spanners that are not expressible with vset-automata that are synchronized for all their variables:

**Proposition 5.3.8.** There is no sequential vset-automaton that is synchronized for $x$ and equivalent to $(a \cdot x\{\epsilon\} \cdot a) \lor (b \cdot x\{\epsilon\} \cdot b)$.

**Proof.** Note that $\gamma(A)(aa) \neq \emptyset$ and that also $\gamma(A)(bb) \neq \emptyset$; however, $\gamma(A)(ab) = \emptyset$. Note also that $\gamma(A)(aa) = \{\mu_1\}$ where $\mu_1$ is the mapping that maps $x$ to $[2, 3]$ and that $\gamma(A)(bb) = \{\mu_2\}$ where $\mu_2$ is the mapping that maps $x$ to $[2, 3]$.

We can conclude that there exists states $q_1, q_2, q_3 \in Q$ and $q_4 \in F$ such that the following is a valid an accepting run of $A$ (on $aa$):

$$\rho_1 := (q_0, 1) \xrightarrow{a} (q_1, 2) \xrightarrow{x} (q_2, 2) \xrightarrow{x} (q_3, 2) \xrightarrow{a} (q_4, 3)$$

Similarly, there exists states $p_1, p_2, p_3 \in Q$ and $p_4 \in F$ such that the following is a valid an accepting run of $A$ (on $bb$):

$$\rho_2 := (q_0, 1) \xrightarrow{b} (p_1, 2) \xrightarrow{x} (p_2, 2) \xrightarrow{x} (p_3, 2) \xrightarrow{b} (p_4, 3)$$

Let us assume by contradiction that $A$ is synchronized for the variable $x$. Thus, we obtain that $p_2 = q_2$. So we can take the first run of $\rho_1$ and the second run of $\rho_2$ and glue them together to obtain a new run:

$$\rho_{1,2} := (q_0, 1) \xrightarrow{a} (q_1, 2) \xrightarrow{x} (q_2, 2) \xrightarrow{x} (p_3, 2) \xrightarrow{b} (p_4, 3)$$

This run is valid and accepting and therefore we can conclude that $\gamma(A)(ab)$ is not empty (as it contains a mapping that maps $x$ to the span $[2, 3]$). However, we already noted that $\gamma(A)(ab) = \emptyset$ and we assumed that $A$ and $\gamma$ are equivalent, which yields the desired contradiction. $\square$

Hence, by using synchronized vset-automata we sacrifice expressive power. But this restriction also allows us to state the following positive result on the difference of vset-automata:

**Theorem 5.3.9.** Given an input document $d$ and two sequential vset-automata $A_1$ and $A_2$ such that, for $X := \text{Vars}(A_1) \cap \text{Vars}(A_2)$, $A_1$ is semi-functional for $X$ and $A_2$ is synchronized for $X$, one can construct a sequential vset-automaton $A_d$ with $\gamma(A_d)(d) = \gamma(A_1 \setminus A_2)\gamma(d)$ in polynomial time.

We now discuss some of the proof’s key ideas; the full proof is given in Section 5.3.4. The first key observation is that $A_2$ can be treated as a functional vset-automaton that uses only the common variables (similarly to the proof of Lemma 5.2.5). This allows us to work with the variable configurations of $A_2$, and construct the match structure $M(A_2, d)$ of $A_2$ on $d$. This model was introduced (without a name) in Section 5.3.2 to evaluate functional vset-automata with polynomial delay and was referred to as $A_G$. As explained there, every element of
\\[ A_2 \](d) can be uniquely expressed as a sequence of |d| + 1 variable configurations of \( A_2 \).

Every accepting run of \( A_2 \) on \( d \) can be mapped into such a sequence by taking the variable configurations of the states just before a symbol of \( d \) is read (and the configuration of the accepting state). The match structure \( M(A_2, d) \) is an NFA that has the set of variable configurations of \( A_2 \) as its alphabet; and its language is exactly the set of sequences of variable configurations that correspond to elements of \( \langle A_2 \rangle(d) \).

While determinizing match structures is still hard, the fact that \( A_2 \) is synchronizing on the common variables allows us to construct a deterministic match structure \( D_2 \) from \( M(A_2, d) \). Using a variant of the proof of Lemma 5.2.5, we can then combine \( A_1 \) and \( A_2 \) into an ad-hoc vset-automaton \( A_d \) with \( \langle A_d \rangle(d) = \langle A_1 \setminus A_2 \rangle(d) \).

After creating \( A_d \) according to Theorem 5.3.9, we can use Theorem 5.1.1 to obtain the following tractability result:

**Corollary 5.3.10.** Given an input document \( d \) and two sequential vset-automata \( A_1 \) and \( A_2 \) such that, for \( X := \text{Vars}(A_1) \cap \text{Vars}(A_2) \), \( A_1 \) is semi-functional for \( X \) and \( A_2 \) is synchronized for \( X \), one can enumerate the mappings in \( \langle A_1 \setminus A_2 \rangle(d) \) in polynomial delay.

We saw that disallowing disjunctions over the variables leads to tractability. Can we relax this restriction by allowing a fixed number of such disjunctions? Our next result is a step towards answering this question. A disjunction-free regex formula is a regex formula that does not contain any subexpression of the form \( \gamma_1 \lor \gamma_2 \).

**Proposition 5.3.11.** The following decision problem is NP-complete. Given two sequential regex formulas \( \gamma_1 \) and \( \gamma_2 \) with \( \text{Vars}(\gamma_1) = \text{Vars}(\gamma_2) \) and an input document \( d \) such that

- \( \gamma_1 \) is functional,
- \( \gamma_2 \) is a disjunction of regex formulas \( \gamma_2^i \) such that each is disjunction-free,
- for every variable \( x \in \text{Vars}(\gamma_2) \), it holds that \( x \) appears in at most 3 disjuncts \( \gamma_2^i \) of \( \gamma_2 \),

is \( \langle \gamma_1 \setminus \gamma_2 \rangle(d) \) nonempty?

**Proof.** This proof is an adaption of the proof of Theorem 4.6.1 using mostly the same notation. Instead of a general 3CNF formula, let \( \varphi = C_1 \land \ldots \land C_m \) be a CNF formula, such that every clause \( C_i \) contains either 2 or 3 literals, and each of the variables appears in at most 3 clauses. Deciding satisfiability for such a formula is still NP-complete [70].

For \( \gamma_1 \) to not have any disjunctions, we first set \( d = (bab)^n \) for some \( a, b \in \Sigma \). We then define

\[
\gamma_1 = (bx_1\{a^*\} \cdot a^*b) \cdots (bx_n\{a^*\} \cdot a^*b).
\]
Intuitively $\gamma_1$ encodes all of the possible assignments. The regex formula $\gamma_2$ is defined analogously to $\gamma_2$ in the proof of Theorem 4.6.1 with an adaptation to the new input document and a slight simplification of the $\gamma_2$'s (since we do not need $\gamma_2$ to be functional anymore). Formally, we set

$$\gamma_2^i = (bab)^{i_1 - 1} \delta_{i_1} (bab)^{i_2 - i_1 - 1} \delta_{i_2} (bab)^{n - i_2}$$

if only variables $x_{i_1}, x_{i_2}$ with $i_1 < i_2$ appear in clause $C_i$, and

$$\gamma_2^i = (bab)^{i_1 - 1} \delta_{i_1} (bab)^{i_2 - i_1 - 1} \delta_{i_2} (bab)^{i_3 - i_2 - 1} \delta_{i_3} (bab)^{n - i_3}$$

if variables $x_{i_1}, x_{i_2}, x_{i_3}$ with $i_1 < i_2 < i_3$ appear in clause $C_i$.

By the choice of the 3CNF formula $\varphi$, every variable $x_j$ appears in at most three regex formulas of the form $\gamma_2^i$. Correctness of this reduction can be shown analogously to that of Theorem 4.6.1. □

We conclude that evaluating $\gamma_1 \setminus \gamma_2$ remains hard even if $\gamma_1$ is functional (and hence also semi-functional for the common variables) and $\gamma_2$ is a disjunction of disjunction-free regex formulas, and each of $\gamma_2$’s variables appears in at most three such disjuncts.

It is open whether the problem becomes tractable if the variables are limited to at most one or two disjuncts.

5.3.4 Proof of Theorem 5.3.9

For two sequential vset-automata $A_1$ and $A_2$, let $X := \text{Vars}(A_1) \cap \text{Vars}(A_2)$, and assume that $A_1$ is semi-functional for $X$ and that $A_2$ is synchronized for $X$. Before we consider the document, we fix some assumptions on $A_2$ that are possible without loss of generality.

Assumptions on $A_2$:

First of all, we assume that $\text{Vars}(A_2) = X$. Variables in $\text{Vars}(A_2) \setminus \text{Vars}(A_1)$ play no role for the difference $A_1 \setminus A_2$, so this assumption does not change the correctness of the construction. We can guarantee this by replacing all transitions with operations for variables from $\text{Vars}(A_2) \setminus \text{Vars}(A_1)$ with $\epsilon$-transitions.

As pointed in Lemma 4.4.5 this change turns $A_2$ into an automaton for $\pi_X(A_2)$. Although Freydenberger et al. only remarks this for functional automata, the same argument translates to sequential automata; and the translation can be performed in time that is linear in the size of the transition relation of $A_2$.

As $A_2$ is synchronized for $X$, for every $x \in X$, either all accepting runs of $A_2$ operate on $x$, or no accepting run of $A$ operates on $x$. We assume without loss of generality that the former is the case (this can be achieved by trimming $A_2$ by removing all states that are not reachable from the initial state, and from which no accepting state can be reached). As every variable operation from $\Gamma_X$ has a unique target state, we know that $A_2$ is semi-functional for $X$. But as we assume that $\text{Vars}(A_2) = X$, and we have established that every accepting run of $A_2$ acts on all variables of $X$, this implies that $A_2$ is in fact functional.
Match structures in general:

We are now ready for the main part of the proof. It uses constructions from the proof of Theorem 4.3.1 which describes a polynomial delay algorithm for \( \| A \| (d) \) for functional vset-automata \( A \) and documents \( d \).

Given a document \( d = \sigma_1 \cdots \sigma_\ell \) with \( \ell \geq 1 \), the key idea of the construction is to represent each element of \( \| A \| (d) \) as a sequence \( c_0, \ldots, c_{\ell+1} \) of variable configurations of \( A \). Each \( c_i \) is the last variable configuration before \( \sigma_i \) is processed; and \( c_{\ell+1} \) is the configuration where all variables have been closed.

To obtain these configurations, we construct the match graph of \( A \) on \( d \), a directed acyclic graph \( G(A, d) \) that has one designated source node, and nodes of the form \((i, q)\), where \( 0 \leq i \leq \ell \) and \( q \) is a state of \( A \). Intuitively, the node \((i, q)\) represents that after processing the first \( i \) letters of \( d \) (i.e., \( \sigma_1 \cdots \sigma_i \)) \( A \) can be in state \( q \).

Consequently, an edge from \((i, p)\) to \((i+1, q)\) represents that if \( A \) is in state \( p \) and reads the symbol \( \sigma_{i+1} \), it can enter state \( q \) (not necessarily directly; it may process arbitrarily many variable operations or \( \epsilon \)-transitions after reading \( \sigma_{i+1} \)). The match graph \( G(A, d) \) can be constructed from \( A \) and \( d \) directly from the transition relation of \( A \) and by using standard reachability algorithms. These reachability algorithms can also be used to trim the match graph: We remove all nodes that cannot be reached from the source, and all nodes that cannot reach a node \((\ell + 1, q)\) where \( q \) is a accepting state of \( A \). As \( A \) is functional, each state \( q \) has a well-defined variable configuration \( c_q \), which means that we can also associate each node \((i, q)\) with that variable configuration.

We can now interpret the match graph \( G \) as an NFA \( M(A, d) \) over the alphabet of variable configurations in the following way: All nodes of \( G(A, d) \) become states of \( M(A, d) \). The source of \( G(A, d) \) becomes the initial state, and the accepting state of \( M(A, d) \) are those nodes \((\ell + 1, q)\) where \( q \) is a accepting state of \( A \). Finally, each edge from a node \( v \) to a node \((i, q)\) in \( G(A, d) \) becomes a transition with the letter \( c_q \). We call the automaton \( M(A, d) \) the match structure of \( A \) on the document \( d \).

As shown in the proof of Theorem 4.3.1, there is a one-to-one correspondence between the elements of \( \| A \| (d) \) and the words in the language of \( A_G \).

Preliminaries to determinizing \( M(A_2, d) \):

In the present proof, we start with the same construction, and first construct the match structure \( M(A_2, d) \) of \( A_2 \) on \( d \), where \( d = \sigma_1 \cdots \sigma_\ell \) with \( \ell \geq 1 \) is our specific input document.

We have already established that we can assume that \( A_2 \) is functional; hence, we can directly use the construction that we discussed previously. Our next goal is to determinize \( M(A_2, d) \), and use this to implicitly construct a vset-automaton for the complement of \( \| A_2 \| (d) \). This can then be combined with \( A_1 \) using a minor variation of the join-construction from Lemma 5.2.5.

In general, determinizing a match structure is not a viable approach (this was already observed in Chapter 4). But in our specific case, we can use that \( A_2 \) is synchronizing for \( X \) (which we assume to be identical with \( Vars(A_2) \)). As every
variable operation from $\Gamma_X$ has a unique target state (and as $A_2$ is functional), we know that every accepting run of $A_2$ executes the operations in the same order. Hence, we can define $\omega_1, \ldots, \omega_{2k} \in \Gamma_X$ with $k := |X|$ according to this order (i.e., the $i$-th operation in every accepting run is $\omega_i$). Accordingly, we define a sequence $c_0, c_1, \ldots, c_{2k}$ of variable configurations, where $c_0$ is the configuration that assigns $w$ to all variables of $x$, and $c_{i+1}$ is the configuration that is obtained from applying operation $\omega_i$ to configuration $c_i$. Note that, as $A_2$ is functional, we use variable configurations (that use $w$) instead of extended variable configurations (that use $u$). Later on, this will help us distinguish between the configurations of the match structure of $A_2$ and the extended configurations of $A_1$.

As the order of the possible variable operations of $A_2$ is fixed, the language of the match structure of $A_2$ on any document (not just our specific document $d$) is a subset of $c_0^* \cdot c_1^* \cdot \cdots \cdot c_{2k}^*$. Note that this does not mean that every configuration would appear; if two variable operations are always performed without consuming a symbol between them, the intermediate variable configuration would never appear. Consider for instance the following functional vsat-automaton that is synchronized for all its variables:

$$
\begin{array}{c}
\sum \\
\downarrow \\
q_0 \xrightarrow{x} q_1 \xrightarrow{y} \quad q_2 \xrightarrow{y} \quad q_3 \xrightarrow{y} \quad q_4
\end{array}
$$

Then on every document $d$, its match structure accepts only a single sequence of variable configurations, namely $c_{q_0^*} \cdot c_{q_1^*} \cdot \cdots \cdot c_{q_4^*}$. If we replace the state $q_2$ with an arbitrarily complicated NFA, the same observation would hold for all documents that belong to the language of the NFA (all others would be rejected).

The fact that $M(A_2, d)$ expresses a concatenation of unary languages is not enough to allow for determinization; for this, we need to use the unique target states. Assume that $M(A_2, d)$ reads a new variable configuration. More formally and more specifically, assume that $G(A_2, d)$ contains edges

- from some $(i, p)$ to some $(i + 1, q)$ with $c_p \neq c_q$, and
- from some $(i', p')$ to some $(i' + 1, q')$ with $c_{p'} \neq c_{q'}$,

such that $c_q = c_{q'}$. Choose $j$ such that $c_j = c_q$. Then the transition from $p$ to $q$ and the transition from $p'$ to $q'$ both execute $\omega_j$, which means that they pass through its unique target state. Both can then execute different sequences of epsilon transitions, which means that $q \neq q'$ may hold. But we can switch these sequences of transitions, and observe that $G(A_2, d)$ must also contain an edge from $(i, p)$ to $(i + 1, q')$, and an edge from $(i', p')$ to $(i' + 1, q)$.

Thus, for every variable configuration $c_j$, we can define a set $I_j$ that contains exactly those $q$ such that $M(A_2, d)$ contains an edge from some node $(i, p)$ to some node $(i + 1, q)$ with $c_p \neq c_q$. In other words, $I_j$ contains those states that of $A_2$ that are encoded in states that $M(A_2, d)$ can enter when first reading variable configuration $c_j$. And as we established in the previous paragraph, $A_2$ being synchronized for all its variables ensures that each $I_j$ is well-defined. We also use $Q_q$ to refer to that $I_j$ for which $c_j = c_q$ holds.
Determinizing $M(A_2, d)$:

We use these insights to turn the NFA $M(A_2, d)$ into a DFA $D_2$ over the alphabet $c_0, \ldots, c_{2k}$. Apart from the special initial state, each state of $D_2$ is a triple $(i, s, Q)$, where

- $i$ has the same role as in $G(A_2, d)$, meaning that it encodes how many positions of $d$ have been consumed,
- $s \leq i$ denotes when the current variable configuration was consumed the first time,
- $Q$ is a set of states of $A_2$, and all its elements have the same variable configuration.

We construct $D_2$ by performing a variant of the power set construction that, in addition to taking the special structure of $M(A_2, d)$ into account, also includes a reachability analysis.

The first set of states and transitions is computed as follows: For every variable configuration $c_j$ with $0 \leq j \leq 2k$, we compute a set $Q_j$ that contains exactly those states $q$ such that $c_j = c_j$ and $M(A_2, d)$ contains a state $(0, q)$. If $Q_j \neq \emptyset$, we extend $D_2$ with the state $(0, 0, Q_j)$ and a transition with label $c_j$ from the initial state to $(0, 0, Q_j)$. In this case, $Q_j = I_j$ holds.

To compute the successor states of some state $(i, s, P)$, we proceed as follows: First, we consider successors with the same variable configuration. Let $c_j$ be the variable configuration of all states in $P$. We compute the set $Q_j$ of all states $q$ such that $M(A_2, d)$ contains a transition with label $c_j$ from some state $(i, p)$ to $(i + 1, q)$. If $Q_j \neq \emptyset$, we extend $D_2$ with the state $(i + 1, s, Q_j)$ and a transition with label $c_j$ from the $(i, s, P)$ to $(i + 1, s, Q_j)$.

To determine successors with different configurations, we consider all $j'$ with $j < j' \leq 2k$. For every such $c_j$, we check if $M(A_2, d)$ contains a transition from some $(i, p)$ with $p \in P$ to some $(i + 1, q)$ with $c_j = c_{j'}$. If this is the case, we extend $D_2$ with an edge from the state $(i, s, P)$ to the state $(i + 1, i + 1, I_{j'})$ that is labeled with $c_{j'}$, and add the state $(i + 1, i + 1, I_{j'})$ to $D_2$ if it does not exist yet.

Intuitively, $D_2$ simulates $M(A_2, d)$ by disassembling it into sub-automata for the unary languages of each $c_j$. It needs to keep track of where in $d$ the current part started (using the middle component of each state triple) to avoid accepting words of the wrong length. Note that for each state $(i, s, Q)$ with $s = 0$, we have $Q = I_j$ for some variable component $c_j$ with $0 \leq j \leq 2k$. Hence, and by the definition of $D_2$, we observe that each component $Q$ is determined by the combination of $i$, $s$, and $c_j$. Therefore, we can bound the number of states in $D_2$ by $O(\ell^2 k)$, as $1 \leq s \leq i \leq \ell + 1$ and $0 \leq j \leq 2k$ hold. Likewise, the number of transitions is bounded by $O(\ell^2 k^2)$.

Combining $A_1$ and $D_2$:

The actual construction of $A_d$ is a variant of the proof of Lemma 5.2.5. To fix identifiers, we declare that $A_1 = (V_1, Q_{1,0,1}, F_1, \delta_1)$ and $D_2 = (Q_2, q_{0,2}, F_2, \delta_2)$.

95
As \( A_1 \) is sequential, every state \( q_i \in Q_1 \) has an extended variable configuration \( \tilde{c}_{q_i} \); and as \( A_1 \) is semi-functional for \( X \), we know that \( \tilde{c}_{q_i}(x) \in \{u, o, c\} \) for all \( x \in X \). Although \( D_2 \) is technically a DFA over the alphabet of variable configurations \( c_0, \ldots, c_{2k} \), its definition allows us to associate each state in \( Q_2 \) with a variable configuration \( c_q \) by considering the incoming transitions of \( q \) (which is equivalent to considering the variable configuration of the states that are encoded in \( q \)), and by setting \( c_{q_0,2}(x) = w \) for all \( x \in X \).

Recall that we aim to construct a sequential \( \text{vset-automaton} \) \( A \) with \( \|A_d\|(d) = \|A_1 \setminus A_2\|(d) \). In principle, \( A_d \) simulates \( A_1 \) and \( A_2 \) in parallel on the input document \( d \). But instead of using \( A_2 \), we use its deterministic representation \( D_2 \). If \( A_1 \) picks a next state \( q_1 \), its opponent \( D_2 \) tries to counter that by picking an a state that has consistent behavior on the variables of \( X \) (as \( D_2 \) has variable configurations as input, it hides the states of \( A_2 \) behind a layer of abstraction). If \( D_2 \) can pick such a set of its states that is consistent, it can follow \( A_1 \) for the current input. If all available states are inconsistent, it cannot follow \( A_2 \) and has to enter the trap state. This means that the finishing the current run of \( A_1 \) will lead to an element of \( \|A_1 \setminus A_2\|(d) \).

One technical problem is that \( A_1 \) may choose to leave variables from \( X \) undefined; and by definition, \( \|A_1 \setminus A_2\|(d) \) only takes the common variables into account. Thus, if \( D_2 \) is forced at some point to open a variable that was not yet processed, it cannot decide whether \( A_1 \) will open that variable later in the run; and using the finite control to keep track of the arising combinations would lead to an exponential blowup.

But we can work around this problem: As \( A_1 \) is semi-functional for \( X \), each accepting state contains information about which variables were skipped in every run that ends in that state. For each \( q \in F_1 \) and each variable \( x \in X \), we say that \( q \) skips \( x \) if \( \tilde{c}_{q}(x) = u \). Let \( S(q) \) denote the set of skipped variables in \( q \). The idea is to decompose \( A_1 \) into sub-automata that all skip the same variables. There are at most \( |F_1| \) different sets \( S(q) \) with \( q \in F_1 \). We call these \( S_1, \ldots, S_f \) with \( f \leq |F_1| \).

For each \( S_j \), we create a sub-automaton \( A_{1,j} = (V_1, Q_{1,j}, q_{0,1}, F_{1,j}, \delta_{1,j}) \) that accepts exactly those runs of \( A_1 \) that end in a state \( q \) with \( S(q) = S_j \). We first mark \( q_{0,1} \) and all \( q \in F_1 \) with \( S(q) = S_f \). Then we use a standard reachability algorithm to mark all states and transitions that lead from \( q_{0,1} \) to some marked finite state. Finally, we obtain \( A_{1,j} \) by removing all unmarked states and transitions. The resulting automaton is sequential and semi-functional for \( X \), and all accepting runs skip the same variables. Furthermore, we observe that \( \|A_{1,j}\| = \bigcup_{i=1}^{f} \|A_{1,i}\| \).

The last major step is creating a \( \text{vset-automaton} \) \( A_{d,j} \) such that \( \|A_{d,j}\|(d) = \|A_{1,j} \setminus A_2\|(d) \). Recall that each \( A_{1,j} \) skips exactly the same variables from \( S_j \).

For this, we define the notions of consistent and inconsistent state pairs. For each \( q_1 \in Q_{1,j} \), each \( q_2 \in Q_2 \), and each \( x \in X \), we say that \( q_1 \) and \( q_2 \) are consistent for \( x \) if one of the following conditions holds:

- \( x \in S_j \),
- \( \tilde{c}_{q_1}(x) = u \) and \( c_{q_2}(x) = w \),
- \( \tilde{c}_{q_1}(x) = c_{q_2}(x) \in \{o, c\} \).
Otherwise, $q_1$ and $q_2$ are inconsistent. Building on these definitions, we say that

- $q_1$ and $q_2$ are consistent if they are consistent for all $x \in X$,
- $q_1$ and $q_2$ are inconsistent if they are inconsistent for at least one $x \in X$.

The set of states of $A_{d,j}$ shall consist of states of the following type:

- consistent pairs $(q_1, q_2)$, with $q_1 \in Q_{1,j}$ and $q_2 \in Q_2$,
- pairs from $Q_{1,j} \times \{\text{trap}\}$,
- a number of unnamed helper states.

Here trap is a special trap state that we shall use to denote that something happened that made the parallel simulations of $A_{1,j}$ and $D$ inconsistent.

We define the initial state of $A_{d,j}$ as $(q_{0,1}, q_{0,2})$ and its set of accepting states as $F_1 \times \{\text{trap}\}$. Again, trap intuitively denotes that the two automata have inconsistent behavior on the variable operations. Following the same intuition, we define that for every transition from some state $p$ to some state $q$ in $A_{1,j}$, we have a transition with the same label from $(p, \text{trap})$ to $(q, \text{trap})$ in $A_{d,j}$.

To define the “main behavior” of $A_{d,j}$, we use the notion of variable-$\epsilon$-closure as in Lemma 5.2.5. For every $p \in Q_{1,j}$, we define $\mathcal{VE}(p)$ as all $q \in Q_{1,j}$ that can be reached from $p$ by using only transitions from $\{\epsilon\} \cup \Gamma_{V_1}$.

Let $\hat{Q}_2 \subset Q_2$ be the set of all states to which $q_{0,2}$ has a transition. For each $q_1 \in \mathcal{VE}(q_{0,1})$, we distinguish the following cases:

- If there is some $q_2 \in \hat{Q}$ such that $q_1$ and $q_2$ are consistent, we add a state $(q_1, q_2)$ to $A_d$.
- If all $q_2 \in \hat{Q}$ are inconsistent with $q_1$, we add a state $(q_1, \text{trap})$ to $A_d$.

In both cases, we connect $q_{0,1}$ with the new state using a sequence of helper states that has exactly the same variable operations and $\epsilon$-transitions as one that takes $Q_1$ from $q_{0,1}$ to $q_1$. We consider all states $(q_1, q_2)$ and $(q_1, P)$ that were introduced in this step states on level 0. Intuitively, state of the form $(q_1, q_2)$ describe cases where $D_2$ can follow the behavior of $A_{1,j}$, states of the form $(q_1, \text{trap})$ describe cases where it cannot follow the behavior and has to give up.

Now, we successively process the symbols $\sigma_i$ of $d = \sigma_1 \cdots \sigma_\ell$. For each $i$ with $1 \leq i \leq \ell$, we process the states on level $i - 1$ and compute their successors as follows:

For each state $(p_1, p_2)$ on level $i - 1$, we define $\hat{Q} \subset Q_2$ as the set of all states to which $p_2$ has a transition. We then consider each $q_1 \in \bigcup_{(p_1, \sigma_i, q') \in E_{1,j}} \mathcal{VE}(q')$ and distinguish the same cases as for level 0. Now, the new non-trap states are on level $i$; and in each case, $(p_1, p_2)$ is connected with the new state using a sequence of helper states that processes $\sigma_i$ and then exactly the same variable operations and $\epsilon$-transitions that take $A_1$ from $q'$ to $q_1$.

Now observe that every run of $A_{1,j}$ maps into a run of $A_{d,j}$ (and, likewise, every run of $A_1$ that skips exactly the variables of $S_j$ maps into a run of $A_{1,j}$). If this run can be matched by any run of $A_2$ (as realized by $D_2$), it ends in a state $(q_1, q_2)$.
and is not an accepting run of \( A_{d,j} \). But if it is not matched, then \( D_2 \) will be forced to admit the inconsistency, and redirect the simulation into the copy of \( A_{1,j} \) that assigns the trap state to \( D_2 \). In other words, \( \| A_{d,j} \|(d) = \| A_{1,j} \setminus A_2 \|(d) \).

Finally, we obtain \( A_d \) by taking all \( A_{d,j} \) and adding a new initial state that has an \( \varepsilon \)-transition to each initial state of some \( A_{d,j} \). Then

\[
\| A_d \|(d) = \bigcup_{j=1}^{f} \| A_{d,j} \|(d) = \bigcup_{j=1}^{f} \| A_{1,j} \setminus A_2 \|(d) = \| A_1 \setminus A_2 \|(d).
\]

As \( A_1 \) is sequential, every \( A_{1,j} \) is sequential. Therefore, every \( A_{d,j} \) is sequential, and so is \( A_d \).

**Complexity:**

Let \( m_i \) and \( n_i \) denote the number of transitions and states of \( A_i \); let \( \ell = |d| \), and \( k = |\text{Vars}(A_1) \cap \text{Vars}(A_2)| \). Let \( v := |\text{Vars}(A_1)| \). The match structure \( M(A_2, d) \) can be constructed in \( O(n_2^2) \) (see Theorem 4.3.1). Recall that \( D_2 \) has \( O(\ell^2k) \) states and \( O(\ell^2k^2) \) transitions. Each transition can be computed in \( O(n_2) \), which means that we can obtain \( D_2 \) from \( M(A_2, d) \) in \( O(\ell^2 k^2 n_2) \). Hence, the total time of computing \( D_2 \) from \( A_2 \) and \( d \) is \( O(\ell^2 k^2 n_2 + \ell n_2^2) \).

To construct an automata \( A_{d,j} \), we first pre-compute the comparisons of variable configurations in time \( O(n_1 k^2) \) (as \( D_2 \) has \( O(k) \) different variable configurations, and variable configurations can be compared in \( O(k) \)). We also pre-compute the variable-\( \varepsilon \)-closures, which takes \( O(m_1 n_1) \).

As \( A_1 \) has at most \( |F_1| \) different skip sets \( S_j \), we need to compute \( O(n_1) \) sub-automata \( A_j \). Each can be constructed with a standard reachability analysis in time \( O(m_1 + n_1) \). Hence, computing all automata \( A_{1,j} \) takes time \( O(m_1 n_1 + n_1^2) \). This will be subsumed by the complexity of the next steps.

For each \( A_{d,j} \), we have to compute \( \ell + 1 \) levels; in each level, we combine \( O(n_1 k) \) state pairs \( (p_1, p_2) \) with \( O(n_1 k) \) state pairs \( (q_1, q_2) \) (as \( D_2 \) is deterministic and over an alphabet of size \( 2k \), each state has \( O(k) \) outgoing transitions), and we need to include \( O(v) \) helper states. Hence, each \( A_{d,j} \) can be constructed in time \( O(\ell k^2 n_1^2 v) \) without pre-computations, and \( O(\ell k^2 n_1^2 v + m_1 n_1) \) including pre-computations.

Hence, computing \( A_d \) by constructing \( O(n_1) \) many \( A_{d,j} \), each in time \( O(\ell k^2 n_1^2 v + m_1 n_1) \), takes a total time of \( O(\ell k^2 n_1^2 v + m_1 n_1^2) \).

This combines to a total time of \( O(\ell^2 k^2 n_2 + \ell n_2^2 + \ell k^2 n_1^3 v + m_1 n_1^2) \). For a less precise estimation, let \( n \in \Omega(n_1 + n_2) \) and observe that \( k \leq v \) and \( m_1 \in O(n_2^2) \). Then the complexity becomes \( O(\ell^2 v^2 n^3 + \ell v^2 n^3 + \ell v^3 n^3 + n^4) = O(\ell^2 v^2 n + \ell v^3 n^3 + n^4) \). This completes our proof.

### 5.4 Extraction Complexity of Schemaless Spanners

In this section, we discuss queries that are defined as RA expressions over schemaless spanners given in a representation language \( L \) (e.g., regex formulas), which
we refer to as the language of the atomic spanners. Formally, an RA tree is a directed and ordered tree whose inner nodes are labeled with RA operators, the out-degree of every inner node is the arity of its RA operator, and each of the leaves is a placeholder for a schemaless spanner. For illustration, Figure 5.2 shows an RA tree $\tau$, where the placeholders are the rectangular boxes with the question marks; the dashed arrows should be ignored for now. The RA tree corresponds to the relational concept of a query tree or a logical query plan \cite{38,74}. We restrict the discussion to the RA operators projection, union, natural join, and difference.

Let $\mathcal{L}$ be a representation language for atomic spanners, and let $\tau$ be an RA tree. An instantiation of $\tau$ assigns a schemaless spanner representation from $\mathcal{L}$ to every placeholder, and a set of variables to every projection. For example, Figure 5.2 shows an instantiation $I$ for $\tau$ via the dashed arrows; here, we can think of $\mathcal{L}$ as the class of sequential regex formulas, and so, each $\alpha$ expression is a sequential regex formula.

An instantiation $I$ of $\tau$ transforms $\tau$ into an actual schemaless spanner representation, where $\tau$ is the parse tree of its algebraic expression. We denote this representation by $I[\tau]$. As usual, by $\|I[\tau]\|$ we denote the actual schemaless spanner that $I[\tau]$ represents.

**Example 5.4.1.** Assume that the input document $d_{\text{Students}}$ from the earlier examples is now extended and contains additional information about the students, including recommendations they got from their professors and previous hires. Let us assume that every line begins with a student’s name and contains information about that student. Let us also assume that we have the following functional regex formulas:

- a regex formula $\alpha_{\text{sm}}$ with capture variables $x_{\text{stdnt}}, x_{\text{ml}}$ that extracts names with their corresponding email addresses;
- a regex formula $\alpha_{\text{sp}}$ with capture variables $x_{\text{stdnt}}, x_{\text{phn}}$ that extracts names with their corresponding phone numbers;
- a regex formula $\alpha_{\text{nr}}$ with capture variables $x_{\text{stdnt}}, x_{\text{rcmnd}}$ that extracts names with their corresponding recommendations.

Note that all of the regex formulas are functional, that is, they do not output partial mappings. The following query extracts the students that do not have recommendations.

$$\pi_{\{x_{\text{stdnt}}\}}\left( (\alpha_{\text{sm}} \Join \alpha_{\text{sp}}) \setminus (\alpha_{\text{nr}}) \right)$$

This query is $I[\tau]$ for the RA tree $\tau$ and the instantiation $I$ of Figure 5.2. This query defines the spanner $\|I[\tau]\|$, and the set of extracted spans is $\|I[\tau]\|(d_{\text{Students}})$.

We present a complexity measure that is unique to spanners, namely the extraction complexity, where the RA tree $\tau$ is fixed and the input consists of both the instantiation $I$ and the input document $d$. Specifically, the evaluation problem for an RA tree $\tau$ is that of evaluating $\|I[\tau]\|(d)$, given $I$ and $d$. Similarly, the
nonemptiness problem for an RA tree \( \tau \) is that of deciding whether \( \mathcal{I}[\tau](\mathbf{d}) \) is nonempty, given \( \mathcal{I} \) and \( \mathbf{d} \).

Clearly, some RA trees have an intractable nonemptiness problem and, consequently, an intractable evaluation problem. For example, if \( \mathcal{L} \) is the class of sequential regex formulas and \( \tau \) is the RA tree that consists of a single natural-join node, then the nonemptiness problem for \( \tau \) is NP-complete (Theorem 5.2.1). Also, if \( \mathcal{L} \) is the class of functional regex formulas and \( \tau \) is the RA tree that consists of a single difference node, then the nonemptiness problem for \( \tau \) is NP-complete (Theorem 4.6.1). In contrast, by composing the positive results established in Sections 5.2 and 5.3, we obtain the following theorem, which is a consequence of Lemma 5.2.2 and Lemma 5.3.1.

**Theorem 5.4.1.** Let \( \mathcal{L} \) be the class of sequential vsset-automata. Let \( k \) be a fixed natural number and \( \tau \) an RA tree. The evaluation problem for \( \tau \) is solvable with polynomial delay, if we assume that, for all join and difference nodes \( v \) of \( \mathcal{I}[\tau] \), the left and right subtrees under \( v \) share at most \( k \) variables.

We restate that, while static compilation suffices for the positive operators, we need dynamic compilation to support the difference operator. Interestingly, the dynamic approach allows us to incorporate into the RA tree other representations of schemaless spanners, which can be treated as a black-box, as long as these spanners can be evaluated in polynomial time and are of a bounded degree. In turn, the degree of a schemaless spanner \( S \) is the maximal cardinality of a mapping produced over all possible documents, that is, \( \max\{|\text{dom}(\mu)| : \mathbf{d} \in \Sigma^*, \mu \in S(\mathbf{d})\} \).

Formalizing the above, we can conclude from Theorem 5.4.1 a generalization that allows for black-box schemaless spanners. To this end, we call a representation language \( \mathcal{L}' \) for schemaless spanners tractable if \( \mathcal{I}[\beta](\mathbf{d}) \) can be evaluated in polynomial time (for some fixed polynomial), given \( \beta \in \mathcal{L}' \) and \( \mathbf{d} \in \Sigma^* \), and we call \( \mathcal{L}' \) degree bounded if there is a fixed natural number that bounds the number of variables of all the schemaless spanners represented by expressions in \( \mathcal{L}' \).

**Corollary 5.4.2.** Let \( \mathcal{L}' \) be a tractable and degree-bounded representation system for schemaless spanners, and let \( \mathcal{L} \) be the union of \( \mathcal{L}' \) and the class of all sequential vsset-automata. Let \( k \) be a fixed natural number and let \( \tau \) be an RA tree. The evaluation problem of \( \tau \) is solvable with polynomial delay, if we assume that, for
all join and difference nodes \( v \) of \( I[\tau] \), the left and right subtrees under \( v \) share at most \( k \) variables.

Combining such black-box schemaless spanners in the instantiated RA tree increases the expressiveness, as it allows us to incorporate spanners that are not (and possibly cannot be) described as RA expressions over vset-automata, such as string equalities \([27]\). Other examples of such spanners are part of speech (POS) taggers, dependency parsers, sentiment analysis modules, and so on.

**Example 5.4.2.** Following Example 5.4.1, suppose that we now wish to extract the students that do not have any *positive* recommendations. Assume we have a black-box spanner for sentiment analysis, namely \( \text{PosRec} \), with the variables \( x_{\text{stdnt}} \) and \( x_{\text{posrec}} \), that extract names and their corresponding positive recommendation. Note that this spanner has degree 2. We can replace \( \not\in \) in the instantiation \( I \) of Figure 5.2 with \( \text{PosRec} \), and thereby obtain the desired result. If \( \text{PosRec} \) can be computed in polynomial time, then the resulting query can be evaluated in polynomial delay.

**5.5 Concluding Remarks**

We have studied the complexity of evaluating algebraic expressions over schemaless spanners that are represented as sequential regex formulas and sequential vset-automata. We have shown that we hit computational hardness already in the evaluation of the natural join and difference of two such spanners. In contrast, we have shown that we can compile the natural join of two sequential vset-automata (and regex formulas) into a single sequential vset-automaton, in polynomial time, if we assume a constant bound on the number of common variables of the joined spanners; hence, under this assumption, we can evaluate the natural join with polynomial delay. As an alternative to this assumption, we have proposed and investigated a new normal form for sequential spanners, namely disjunctive functional, that allows for such efficient compilation and evaluation.

Bounding the number of common variables between the involved spanners also allows to evaluate the difference with polynomial delay, even though this cannot be obtained by compiling into a vset-automaton—an exponential blowup in the number of states is necessary already for Boolean spanners. Evaluation with polynomial delay is then obtained via a dynamic compilation of both the spanners and the document into a vset-automaton. We have shown how the dynamic compilation approach can be used for establishing upper bounds on general RA trees over regex formulas, vset-automata, and even black-box spanners of a bounded degree. This has been done within the concept of *extraction complexity* that we have proposed, as a new lens to analyzing the complexity of spanners.
Chapter 6

Conclusions

In this thesis, we studied the complexity of evaluating document spanners. We started with a data complexity analysis where the spanner is assumed to be fixed and the document is given as input. We have established a descriptive complexity result according to which the class of spanners computable in polynomial time (in data complexity) is exactly the class of spanners expressible by recursive \( \text{RGXlog} \) programs (i.e., Datalog over regex formulas). We have shown that to capture polynomial time it suffices to use Binary regex formulas. However, \( \text{RGXlog} \) over monadic extractors expresses exactly the class of regular spanners.

As classical complexity theory often associates efficiency with polynomial time while assuming \( P \neq \text{NP} \), we can say that every \( \text{RGXlog} \) program can be evaluated efficiently. However, in real systems, the difference between quadratic and cubic run-time might be significant. Indeed, in recent years there is an increased interest in the qualitative distinction within the class of polynomial time \([1,5,79,80]\): this line of work is often referred to as fine-grained analysis. Establishing lower bounds is obtained by (a) selecting a key problem \( X \) that is conjectured to require essentially \( p(n) \) time for some polynomial \( p \), and (b) reducing \( X \) in a fine-grained way (as our next example illustrates) to any other problem of which we wish to show that it requires essentially \( p(n) \) time. If, for instance, the problem \( X \) is assumed to require quadratic time and we want to show that \( P \) also requires quadratic time, then the reduction of \( X \) to \( P \) should require strictly less than quadratic time. Adapting the fine-grained approach, it is only natural to ask: How can we characterize different classes within the class of polynomial time spanners? According to Theorem 4.3.1, we can evaluate regular spanners, represented by vset-automata, in linear delay after linear preprocessing (in data complexity). Following this, Amarilli et al. \([7]\) have improved our algorithm and presented an algorithm that enumerates the results in constant delay after linear preprocessing. Our preliminary investigation shows that string equalities change the complexity. In particular, we reduce the Matrix Multiplication problem \([8]\) to that of evaluating core spanners and hence show that under the Boolean Matrix Multiplication assumption \([6]\) core spanners cannot be evaluated in constant delay after linear preprocessing (in data complexity). While our repeated tries to connect the number of string equalities and the degree of the polynomial that describes the runtime complexity have failed, we are currently investigating a dif-
different approach based on different classes of string relations \[10\]. That is, we try to find a connection between the selections that are used to express the spanner and the evaluation complexity.

We then moved on to discuss the combined complexity of evaluating spanners, in which both the spanner and the document are regarded a input. We showed that there are inherent differences between relational queries and regex queries. While relational queries are evaluated on a database, regex queries are evaluated on a document. However, we can view regex queries as relational queries by interpreting their atoms as the actual result of applying the regex formula on the document. We refer to this approach as the canonical evaluation approach. The problem is that applying a single regex formula on the document might result in an exponential number of records. Thus, we hit hardness already for a rather simple class of queries. On the other hand, we presented the compilation approach that allows us to obtain efficient enumeration algorithms. While the compilation in the presence of string equality selections is dynamic and depends on a specific document, if the query does not incorporate string equalities we can do the compilation independently of the document. The dynamic compilation helps us to deal with the difference operator and black-box extractors as was discussed in Chapter 5.

It seems as if our upper bounds, and in particular those obtained with the compilation approach, can be generalized to a more general class of queries. To do so, we would like to have a robust definition of a class of algebraic expressions that we can efficiently translate into a vset-automaton. This desired generalization might be obtained by using known results on tractable fragments of conjunctive regular path queries \[11\] \[14\] \[20\] \[22\]. Fagin et al. \[27\] have shown that regular spanners, or equivalently regex UCQs (as we established in Theorem 4.1.1), can be phrased as Unions of Conjunctive Regular Path Queries (UCRPQs). However, their translation entails an exponential blowup. Moreover, even if the translation could be made efficiently, it is not at all clear that tractability properties of the regex UCQ (e.g., bounded number of atoms, or low hypertree width) would translate into tractable properties of UCRPQs. Importantly, in UCRPQs every atom involves two variables, and so, the problem of intractable materialization of an atom (i.e., the main challenge we faced here) does not occur. So, we would like to explore the relationship between the complexity results of the two frameworks.

In addition to data and combined complexity, in Chapter 5 we also proposed the concept of extraction complexity as a new lens to analyzing the complexity of spanners. In extraction complexity, the RA tree is fixed and the input consists of both the document and the atomic extractors. We believe that our analysis has merely touched the tip of the iceberg on the algorithms that can be devised under the guarantee of tractable extraction complexity. In particular, we have proposed sufficient conditions to avoid the inherent hardness of the natural join and difference operators, but it is quite conceivable that less restrictive conditions already suffice. Alternatively, we can ask whether there are conditions on the extractors (possibly incomparable to ours) that are both common in practice and useful to bound the extraction complexity?
Perspective

Our work on document spanners is highly inspired by the work on relational queries. In particular, the similarities between relational queries and IE queries give rise to a plethora of interesting questions (and meaningful insights). Because of this, there is a handful of future directions, part of which are described now.

The first suggested direction is the study of aggregate functions in the context of document spanners. Aggregate functions in the context of relational queries were extensively studied [18,19]. An aggregate function receives as input a set of values and returns a single value. For example, one of the very basic aggregate functions is the count function that returns the number of input values. As aggregate functions cover many data items while returning only a few, aggregate queries can be used to retrieve concise information from the input database. Different query languages in the relational model use aggregation, for instance, SQL, relational algebra and Datalog. Hence, it is natural to extend the framework of document spanners to include aggregations. An aggregate document spanner is a spanner that mentions an aggregate function as part of its output. Aggregate document spanners might provide tools for understanding and analyzing complex workflows that involve machine learning for text analytics.

The second suggested direction is the study of weighted extractors. Queries in the framework of document spanners can extract complex information. Indeed, we saw that every polynomial time spanner can be described as a recursive Datalog program over regex formulas. However, in order to formulate these expressive queries, the user needs to have quite detailed information on the input document. For instance, a single misspelling in one of the regex formulas might result in empty output. In addition, the document itself might contain typos the user is unaware of that might affect the evaluation. Motivated by this, it makes sense to extend the framework such that the result of evaluating a spanner on a document would be a weighted relation. We can view the weights as probabilities or rankings. In fact, we can generalize incomplete databases [46] and probabilistic databases [37] by adapting the framework of provenance semirings [42]. Hence, each weighted spanner will be parametrized by a semiring and will map an input document into a weighted relation over that semiring.

The third suggested direction lies within the intersection of databases and machine learning and can benefit from the previous two. Automatic extraction of structured information from text is a long-standing challenge. Roughly speaking, there are two main schools of thought to tackle IE: the rule-based approach and the machine learning (statistical) approach. Although different, these approaches can be combined in a way that takes the benefit of both worlds. First, we can use rules to extract features for machine-learning algorithms. As these features usually consist of numeric values we can use aggregate spanners or aggregations of weighted spanners for this purpose. Second, spanners may be constructed by machine-learning processes since, as commented in the introduction, even very simple extractors are quite complex and long, and hence manually defining these is a rather complicated task. The result of a learning algorithm would probably be an extractor which incorporates weights on its possible outputs, which is what
we refer to as a weighted spanner. Third, a recent line of work suggests that we can convert deep neural networks into finite state machines [59, 63, 78] to get a better understanding of the functionality as well as the complexity of these networks. A profound understanding of weighted spanners can lead to a better understanding of deep networks for NLP tasks.
Bibliography


Appendix A

Appendix for Chapter 3

A.1 Extension to a Combined Relational/Textual Model

In this section, we extend our main Theorem (Theorem 3.4.1) to Spannerlog—a data and query model introduced by Nahshon et al. [57] that unifies and generalizes relational databases and spanners by considering relations over both strings and spans.

A.1.1 Spannerlog

The fragment of Spannerlog that we consider is referred to by Nahshon et al. [57] as Spannerlog\(_{\text{RGX}}\), and we abbreviate it as simply \(\text{Spl}(\text{RGX})\). A **mixed signature** is a collection of **mixed relation symbols** \(R\) that have two types of attributes: **string** attributes and **span** attributes. We denote by \([R]_{\text{str}}\) and \([R]_{\text{spn}}\) the sets of string attributes and span attributes of \(R\), respectively, where an attribute is represented by its corresponding index. Hence, \([R]_{\text{str}}\) and \([R]_{\text{spn}}\) are disjoint and \([R]_{\text{str}} \cup [R]_{\text{spn}} = \{1, \ldots, \text{arity}(R)\}\). A **mixed relation** over \(R\) is a set of tuples \((a_1, \ldots, a_m)\) where \(m\) is the arity of \(R\) and each \(a_\ell\) is a string in \(\Sigma^*\) if \(\ell \in [R]_{\text{str}}\) and a span \([i, j]\) if \(\ell \in [R]_{\text{spn}}\). A **mixed instance** \(D\) over a mixed signature consists of a mixed relation \(R^D\) for each mixed relation symbol \(R\). A **query** \(Q\) over a mixed signature \(\mathcal{E}\) is associated with a mixed relation symbol \(R_Q\), and it maps every mixed instance \(D\) over \(\mathcal{E}\) into a mixed relation \(Q(D)\) over \(R_Q\).

A mixed signature whose attributes are all string attributes (in all of the mixed relation symbols) is called a **span-free signature**. A mixed relation over a relation symbol whose attributes are all string (respectively, span) attributes is called a **string relation** (respectively, **span relation**). To emphasize the difference between mixed signatures (respectively, mixed relation symbols, mixed relations) and the signatures that do not involve types (which we have dealt with up to this section), we often relate to the latter as **standard** signatures (respectively, standard relation symbols, standard relations).

We consider queries defined by \(\text{Spl}(\text{RGX})\) programs, which are defined as follows. We assume two infinite and disjoint sets \(\text{Vars}_{\text{str}}\) and \(\text{Vars}_{\text{spn}}\) of **string variables** and **span variables**, respectively. To distinguish between the two, we mark a string...
Now these are the generations of Terah: Terah begat Abram, Nahor, and Haran; And Haran died before his father Terah in...

These are the generations of Jacob. Joseph, being seventeen years old, was feeding the flock with his brethren; and the ...

Table A.1: An instance over the unary (span-free) relation Geneo

variable with an overline (e.g., \( \bar{x} \)). By a string term we refer to an expression of the form \( \bar{x} \) or \( \bar{x}_y \), where \( \bar{x} \) is a string variable and \( y \) is a span variable. In Spl(RGX), an atom over an \( m \)-ary relation symbol \( R \) is an expression of the form \( R(\tau_1, \ldots, \tau_m) \) where \( \tau_\ell \) is a string term if \( \ell \in [R]_{\text{str}} \) or a span variable if \( \ell \in [R]_{\text{spn}} \). A regex atom is an expression of the form \( \langle \tau \rangle[\gamma] \) where \( \tau \) is a string term and \( \gamma \) is a regex formula with, possibly, span variables as capture variables. Unlike RGXlog, in which there is a single input string, in Spl(RGX) a regex atom \( \langle \tau \rangle[\gamma] \) indicates that the input for \( \gamma \) is \( \tau \). An Spl(RGX) program is a quadruple \( \langle \mathcal{E}, \mathcal{I}, \Phi, \text{Out} \rangle \) where:

- \( \mathcal{E} \) is a mixed signature referred to as the EDB signature;
- \( \mathcal{I} \) is a mixed signature referred to as the IDB signature;
- \( \Phi \) is a finite set of rules of the form \( \varphi \leftarrow \psi_1, \ldots, \psi_m \) where \( \varphi \) is an atom over \( \mathcal{I} \) and each \( \psi_i \) is an atom over \( \mathcal{I} \), an atom over \( \mathcal{E} \), or a regex atom;
- \( \text{Out} \in \mathcal{I} \) is a designated output relation symbol.

We require the rules to be safe in the following sense: (a) every head variable, or each of its components if it is a string term, occurs at least once in the body of the rule, and (b) every string variable \( \bar{x} \) in the rule occurs, as a string term, in at least one relational atom (over \( \mathcal{E} \) or \( \mathcal{I} \)) in the rule.

We extend Spl(RGX) with stratified negation in the usual way: the set of relation symbols in \( \mathcal{E} \cup \mathcal{I} \) is partitioned into strata \( \mathcal{I}_0, \mathcal{I}_1, \ldots, \mathcal{I}_m \) such that \( \mathcal{I}_0 = \mathcal{E} \), the body of each rule contains only relation symbols from strata that precede or the same as that of the head, and negated atoms in the body are from strata that strictly precede that of the head. In this case, safe rules are those for which every head variable occurs at least once in a positive atom in the body of the rule and every string variable \( \bar{x} \) in the rule occurs, as a string term, in at least one positive relational atom (over \( \mathcal{E} \) or \( \mathcal{I} \)) in the rule.

The semantics of an Spl(RGX) program (with stratified negation) is similar to the semantics of RGXlog programs (with the standard interpretation of stratified negation in Datalog) with the natural semantics of string terms. Given a mixed instance \( D \) over \( \mathcal{E} \), the Spl(RGX) program \( P = \langle \mathcal{E}, \mathcal{I}, \Phi, \text{Out} \rangle \) computes the mixed instance \( P(D) \) over \( \mathcal{I} \) and emits the mixed relation \( \text{Out} \) of \( P(D) \). A query \( Q \) over \( \mathcal{E} \) is definable in Spl(RGX) if there exists an Spl(RGX) program \( P = \langle \mathcal{E}, \mathcal{I}, \Phi, \text{Out} \rangle \) such that \( \text{Out}^{P(D)} = Q(D) \) for all mixed instances \( D \) over \( \mathcal{E} \).

Example A.1.1. The following is an Spl(RGX) program over the mixed signature containing the relation symbol Geneo that consists of a single string attribute.
An example of an possible instance over this signature is given in Table A.1. As usual, OUT is the relation symbol in the head of the last rule, here NONRLTV.

\textbf{A.1.2 Equivalence to Polynomial Time}

Let $\mathcal{E}$ be a span-free signature, and $D$ an instance over $\mathcal{E}$. We define the \textit{extended active domain} of $D$, in notation $\text{adom}^+(D)$, to be the union of the following two sets: \textit{(a)} the set of all strings that appear in $D$, as well as all of their substrings; \textit{and (b)} the set of all spans of strings of $D$.

Notice that for every query $Q$ definable as an $\text{Spl}$(RGX) program, namely $P = \langle \mathcal{E}, \mathcal{I}, \Phi, \text{OUT} \rangle$, and every input database $D$ over $\mathcal{E}$, we have $\text{adom}(Q(D)) \subseteq \text{adom}^+(D)$, that is, every output string is a substring of some string in $D$, and every output span is a span of some string in $D$. Our result in this section states that, under this condition, we can express in $\text{Spl}$(RGX) with stratified negation every query $Q$ computable in polynomial time.

\textbf{Theorem A.1.1.} Let $Q$ be a query over a span-free signature $\mathcal{E}$, with the property that $\text{adom}(Q(D)) \subseteq \text{adom}^+(D)$ for all instances $D$ over $\mathcal{E}$. The following are equivalent:

1. $Q$ is computable in polynomial time.
2. $Q$ is computable in $\text{Spl}(\text{RGX})$ with stratified negation.

We remark that Theorem A.1.1 can be extended to general mixed signatures $\mathcal{E}$ if we assume that every span mentioned in the input database $D$ is within the boundary of some string in $D$. We also remark that Theorem A.1.1 is incorrect without negation, and this can be shown using standard arguments of monotonicity. In addition, since we use negation, in order to prevent ambiguity we use the stratified semantics.

Proof idea. We now discuss the proof idea of Theorem A.1.1. The direction $2 \rightarrow 1$ is straightforward, so we discuss only the direction $1 \rightarrow 2$. Let $Q$ be a query over a span-free signature $\mathcal{E}$, with the property that $\text{adom}(Q(D)) \subseteq \text{adom}^+(D)$ for all instances $D$ over $\mathcal{E}$. Assume that $Q$ is computable in polynomial time. We need to construct an $\text{Spl}(\text{RGX})$ program $P$ with stratified negation for computing $Q$. We do so in two steps. In the first step, we apply Theorem 3.4.1 to get a (standard) Datalog $\downarrow$ program $P'$ that simulates $Q$. Yet, $P'$ does not necessarily respect the typing conditions of $\text{Spl}(\text{RGX})$ with respect to the two types string and span. So, in the second step, we transform $P'$ to an $\text{Spl}(\text{RGX})$ program $P$ as desired. Next, we discuss each step in more detail.

First step. In order to produce the Datalog $\downarrow$ program $P'$, some adaptation is required to apply Theorem 3.4.1. First, we need to deal with the fact that the output of $Q$ may include values that are not in the active domain of the input (namely, spans and substrings). Second, we need to establish a linear order over the active domain. Third, we need to assure that the query that Theorem 3.4.1 is applied on respects isomorphism. To solve the first problem, we extend the input database $D$ with relations that contain every substring and every span of every string in $D$. This can be done using $\text{Spl}(\text{RGX})$ rules with regex atoms. For the second problem, we construct a linear order over the domain of all substrings and spans of strings of $D$, again using $\text{Spl}(\text{RGX})$ rules. For this part, stratified negation is needed. For the third problem, we show how our extended input database allows us to restore $D$ even if all values (strings and spans) are replaced with other values by applying an injective mapping.

Second step. In order to transform $P'$ into a “legal” $\text{Spl}(\text{RGX})$ program $P$ that obeys the typing of attributes and variables, we do the following. First, we replace every IDB relation symbol $R$ with every possible typed version of $R$ by assigning types to attributes. Semantically, we view the original $R$ as the union of all of its typed versions. Second, we replace every rule with every typed version of the rule by replacing relation symbols with their typed versions. Third, we eliminate rules that treat one or more variable inconsistently, that is, the same variable is treated once as a string variable and once as a span variable. The following example demonstrates the steps described above:

Example A.1.2. Let us consider the Datalog $\downarrow$ program that contains the rule $R(x, y) \leftarrow S(x), T(y, z)$. The relation atom $R(x, y)$ has four different typed versions, such as the following.

- $R_{\text{str,str}}(\overline{x}, \overline{y})$ wherein both attributes are string attributes.
\[
R_{\text{spn, str}}(x, \overline{y}) \quad \text{wherein the first attribute is a span attribute and the second is a string attribute.}
\]

The rule \( R(x, y) \leftarrow S(x), T(y, z) \) has \( 2^5 \) different typed versions, one for each “type assignment” for its variables, such as the following.

- \( R_{\text{str, str}}(x, \overline{y}) \leftarrow S_{\text{str}}(x), T_{\text{str, str}}(\overline{y}, \overline{z}) \)
- \( R_{\text{spn, str}}(x, \overline{y}) \leftarrow S_{\text{str}}(x), T_{\text{str, str}}(\overline{y}, \overline{z}) \)

Note that the second rule is type-inconsistent due to the variable \( x \) that is regarded as a span variable in the head atom and as a string variable in the atom \( S_{\text{str}}(x) \), and thus it is eliminated.

Finally, we prove that this replacement preserves the semantics of the program.

### A.2 Proof of Theorem [A.1.1]

Throughout this section, we fix a span-free signature \( \mathcal{E} \) as our input signature, and a query \( Q \) over \( \mathcal{E} \). We will prove that if \( Q \) is computable in polynomial time, then it can be phrased as an Spl(RGX) program with stratified negation. The other direction, that every Spl(RGX) program with stratified negation can be executed in polynomial time, is straightforward, similarly to ordinary Datalog. As described in Section [A.1], our proof is comprised of two steps, which we now explain.

**First Step** We first extend \( \mathcal{E} \) with additional relation symbols. We implicitly assume that each added relation symbol does not already belong to \( \mathcal{E} \).

- \( R_{\text{type}} \)

  The mixed signature \( R_{\text{type}} \) consists of the unary string relation \( \text{STR} \) and the unary span relation \( \text{SPN} \).

- \( R_{\Sigma} \)

  The mixed signature \( R_{\Sigma} \) consists of the relation symbols \( R_\sigma \) for all \( \sigma \in \Sigma \), where \( \text{arity}(R_\sigma) = 2 \), the first attribute is a string attribute (i.e., \( 1 \in [R_\sigma]_{\text{str}} \)) and the second attribute is a span attribute (i.e., \( 2 \in [R_\sigma]_{\text{spn}} \)).

- \( R_{\text{ord}} \)

  The mixed signature \( R_{\text{ord}} \) consists of the following relation symbols:

  - **FIRST** is a string relation with arity 1;
  - **SUCC\text{str}** is a string relation with arity 2;
  - **SUCC\text{mix}** is a mixed relation with arity 2 and \( 1 \in [\text{SUCC}_{\text{mix}}]_{\text{spn}} \) and \( 2 \in [\text{SUCC}_{\text{mix}}]_{\text{str}} \);
  - **SUCC\text{spn}** is a span relation with arity 2;
  - **LAST** is a span relation with arity 1.
We denote by $\mathcal{E}^+$ the signature $\mathcal{E} \cup \mathcal{R}^{\text{type}} \cup \mathcal{R}^{\Sigma} \cup \mathcal{R}^{\text{ord}}$. A mixed instance $E$ over $\mathcal{E}^+$ is said to encode an instance $D$ over $\mathcal{E}$ if all of the following conditions hold.

1. $R^E = R^D$ for all $R \in \mathcal{E}$.
2. The unary string relation $\text{STR}^E$ consists of all strings in $\text{adom}^+(D)$.
3. The unary span relation $\text{SPN}^E$ consists of all of the spans in $\text{adom}^+(D)$.
4. Each $R^E_\sigma$ consists of the tuple $(\overline{x}, y)$ where $\overline{x}$ is a string that occurs in $D$, and $y$ is a span of $\overline{x}$ of length one with $\overline{x}y = \sigma$.
5. The relations of $E$ that instantiate the signature $\mathcal{R}^{\text{ord}}$ interpret this signature so that the union $\text{Succ}^E_{\text{str}} \cup \text{Succ}^E_{\text{mix}} \cup \text{Succ}^E_{\text{spn}}$ is a successor relation of a linear order over $\text{adom}^+(D)$, wherein all strings precede all spans, and $\text{FIRST}$ and $\text{LAST}$ determine the first and last elements in this linear order, respectively.

Note the following in the last item above. Since the strings precede the spans in the linear order, the relation symbol $\text{FIRST}$ is a unary string relation and $\text{LAST}$ is a unary span relation. The relation $\text{Succ}^E_{\text{mix}}$ contains exactly one tuple $(\overline{x}, y)$, where $\overline{x}$ is the last string and $y$ is the first span.

We denote by $\text{Enc}(D)$ the mixed instance over $\mathcal{E}^+$ that encodes $D$. A mixed encoding of an instance $D$ over a span-free signature $\mathcal{E}$ is a mixed instance over $\mathcal{E}^+$ that is isomorphic to $\text{Enc}(D)$. We define the untyped encoding of a mixed encoding $D''$ to be the instance obtained from $D''$ by viewing it as an instance over the signature $\text{untyped}(\mathcal{E}^+)$, where $\text{untyped}(\mathcal{E}^+)$ is an ordinary signature obtained from $\mathcal{E}^+$ by (a) ignoring the types, and (b) relating to the relation symbols $\text{Succ}_{\text{spn}}, \text{Succ}_{\text{str}}$ and $\text{Succ}_{\text{mix}}$ uniformly as the binary successor relation symbol $\text{Succ}$.

Note that a mixed encoding has a unique untyped encoding, and vice versa. In particular, for every untyped encoding $D'$ there exists a unique mixed encoding $D''$ such that $D'$ is the untyped encoding of $D''$. This is true, since we can distinguish between spans and strings via the relations $\text{STR}'$ and $\text{SPN}'$.

Similarly to Lemma 3.4.4, we have the following.

**Lemma A.2.1.** Let $D'$ be an instance over $\text{untyped}(\mathcal{E}^+)$. The following hold:

1. Whether $D'$ is an untyped encoding can be determined in polynomial time.
2. If $D'$ is an untyped encoding, then there is a unique instance $D$ over $\mathcal{E}$ and isomorphism $\iota$ such that $\iota(\text{Enc}(D)) = D'$; moreover, both $D$ and $\iota$ are computable in polynomial time.

**Proof.** Note that an untyped encoding of an instance $D$ encodes each (string) entry of $D$ using relations over the mixed signature $\mathcal{R}^E$. Unlike Lemma 3.4.4, where we had a single string to encode, here we have a database of strings. Therefore, the mixed relations $R_\sigma$ hold an additional string attribute that indicates which entry in $D$ is encoded by the tuple. The rest of this proof is a straightforward adaptation of that of Lemma 3.4.4. \qed
Let \( Q \) be a query over a span-free signature \( \mathcal{E} \). We define the query \( Q^+ \) over untyped \( \mathcal{E}^+ \) on an input \( D_1 \) in the following way: If \( D_1 \) is an untyped encoding of \( D \) over \( \mathcal{E} \) then \( Q^+(D_1) = \iota(Q(D)) \) where \( \iota \) is as in Lemma A.2.1, otherwise \( Q^+(D_1) \) is empty. To apply Theorem 3.4.2 on \( Q^+ \), we make the following observation based on the definition of an untyped encoding and on Lemma A.2.1.

**Observation A.2.2.** The query \( Q^+ \) respects isomorphisms, and moreover, is computable in polynomial time whenever \( Q \) is computable in polynomial time.

Note also that the query \( Q^+ \) is defined over an ordered (standard) signature due to the relations \( \text{Succ}, \text{First} \) and \( \text{Last} \). Due to this and to Observation A.2.2, we can now apply Theorem 3.4.2 on \( Q^+ \) and obtain the following.

**Lemma A.2.3.** If \( Q \) is computable in polynomial time, then there exists a Datalog\(^+ \) program \( P_Q \) over untyped \( \mathcal{E}^+ \) such that for every instance \( D \) over \( \mathcal{E} \) and every untyped encoding \( D' \) of \( D \) it holds that \( P_Q(D') = Q^+(D') \).

This completes the first step, where we translate \( Q \) into an ordinary program \( P_Q \) over an ordinary signature. In the next step, we transform \( P_Q \) into an \( \text{Spl}(\text{RGX}) \) program over \( \mathcal{E} \).

**Second Step** Due to the syntactic resemblance between Datalog and \( \text{Spl}(\text{RGX}) \), one could suggest to consider Datalog rules over an ordinary signature simply as \( \text{Spl}(\text{RGX}) \) rules. However, there is a difference between the semantics of the languages since Datalog programs get standard input instances, as opposed to \( \text{Spl}(\text{RGX}) \) programs that get mixed instances and distinguish between types. We prove the following.

**Lemma A.2.4.** If \( Q \) is computable in polynomial time, then there exists an \( \text{Spl}(\text{RGX}) \) program \( P'_Q \) over \( \mathcal{E}^+ \) such that for every instance \( D \) over \( \mathcal{E} \) and every mixed encoding \( D'' \) of \( D \) it holds that \( P'_Q(D'') = Q^+(D') \) where \( D' \) is the untyped encoding of \( D'' \).

**Proof.** Due to Lemma A.2.3, it suffices to show how to translate \( P_Q \) to a \( \text{Spl}(\text{RGX}) \) program. A mixed version \( \rho^+ \) of a Datalog\(^+ \) rule \( \rho \) is obtained by replacing each relation atom \( R(x_1, \ldots, x_k) \) that appears in \( \rho \) by all of the atoms obtained from it by assigning its attributes all of the possible types. In the special case where \( R \) is the successor relation \( \text{Succ} \) we replace it with each of \( \text{Succ}_{\text{spn}}, \text{Succ}_{\text{str}} \) and \( \text{Succ}_{\text{mix}} \). Let \( P \) be the set of rules that is obtained from \( P \) by replacing each rule \( \rho \) in \( P \) with all of its mixed versions. A rule is called type-inconsistent if it is inconsistent with respect to the type restrictions imposed by \( \mathcal{E}^+ \). We omit from \( P \) rules \( \rho \) that are type-inconsistent and obtain \( P' \). Note that since we have omitted the type-inconsistent rules \( P' \) is a \( \text{Spl}(\text{RGX}) \) program. Since untyped encodings when viewed as mixed instances are consistent we obtain the desired result.

Note that Lemma A.2.4 compares between a mixed instance \( P'_Q(D'') \) and a standard one \( Q^+(D') \). However, this is well-defined since the comparison is done at the instance level.
Example A.2.1. The goal of this example is to demonstrate the construction in the previous proof (of Lemma A.2.4). Let us consider the Datalog\(^+\) program \(P\) that contains the rule \(R(x, y) \leftarrow S(x), T(y, z)\). The relation atom \(R(x, y)\) has four different mixed versions, such as the following.

- \(R_{\text{str}, \text{str}}(x, y)\) where both attributes are string attributes.
- \(R_{\text{spn}, \text{str}}(x, y)\) where the first attribute is a span attribute and the second is a string attribute.

The rule \(R(x, y) \leftarrow S(x), T(y, z)\) has \(2^5\) different mixed versions, one for each “type assignment” for its variables, such as the following.

- \(R_{\text{str}, \text{str}}(x, y) \leftarrow S_{\text{str}}(x), T_{\text{str,str}}(y, z)\)
- \(R_{\text{spn}, \text{str}}(x, y) \leftarrow S_{\text{str}}(x), T_{\text{str,str}}(y, z)\)

Note that these two are rules in the resulting \(\text{Spl(RGX)}\) program \(\bar{P}\). However, the second rule is type-inconsistent due to the variable \(x\) that is regarded as a span variable in the head atom and as a string variable in the atom \(S_{\text{str}}(\bar{x})\), and thus is not a rule in \(P'\).

Note that the input of the \(\text{Spl(RGX)}\) program from Lemma A.2.4 is a mixed encoding. We next show that \(\text{Spl(RGX)}\) with stratified negation is expressive enough to construct the mixed encoding of an instance over a span-free signature.

Lemma A.2.5. There exists an \(\text{Spl(RGX)}\) program \(P = \langle \mathcal{E}, \mathcal{I}, \Phi, \text{OUT} \rangle\) such that \(\mathcal{E} \cup \mathcal{I}\) contains \(\mathcal{E}^+\) and the following holds. For all instances \(D\) over \(\mathcal{E}\) and relation symbols \(R \in \mathcal{E}'\) we have that \(R^{(\text{Enc}(D))} = R^{(\bar{P}(D))}\).

Proof. We construct the program \(P\) as follows.

For every relation symbol \(R\) of \(\mathcal{E}\) and \(i = 1, \ldots, \text{arity}(R)\) we use the following rules:

\[
\text{STR}(\bar{x}_i) \leftarrow R(\bar{x}_i, \ldots, \bar{x}_i, \ldots, \bar{x}_{\text{arity}(R)}), \langle \bar{x}_i \rangle(y\{\} \})
\]
\[
\text{SPN}(y) \leftarrow R(\bar{x}_i, \ldots, \bar{x}_i, \ldots, \bar{x}_{\text{arity}(R)}), \langle \bar{x}_i \rangle(y\{\} \})
\]

(Recall that \(\mathcal{E}\) is span-free.) For all \(\sigma \in \Sigma\) we use the following rule:

\[
R_{\sigma}(\bar{x}, y) \leftarrow \text{STR}(\bar{x}), \langle \bar{x} \rangle(y\{\} \})
\]

In order to define the successor relation \(\text{SUCC}_{\text{str}},\) we define a strict total order \(\succ_{\text{str}},\) which is the usual lexicographic order. We denote our alphabet \(\Sigma\) by \(\{\sigma_1, \ldots, \sigma_n\}\).

In the usual lexicographic order, a string \(s\) follows \(s'\) in this order if either (1) \(s'\) is a strict prefix of \(s\) or (2) the first symbol in which they differ is \(\sigma_i\) in \(s\) and \(\sigma_j\) in \(s'\) where \(j < i\). This can be expressed with the following \(\text{Spl(RGX)}\) rules using the binary relation \(\text{STREQ}\) that holds pairs of equal strings and can be expressed in \(\text{Spl(RGX)}\) (see the comment in Example A.1.1). For case (1) we have:

\[
\succ_{\text{str}}(\bar{x}, \bar{x}') \leftarrow \text{STR}(\bar{x}), \text{STR}(\bar{x}'), \langle \bar{x} \rangle(y\{\} \}), \langle \bar{x}' \rangle(y\{\} \}), \text{STREQ}(\bar{x}_y, \bar{x}'_y)
\]

122
And for case (2):
\[ \succ_{str}(\bar{x}, \bar{x}') \leftarrow \text{STR}(\bar{x}), \text{STR}(\bar{x}'), (\bar{x})[y\{\cdot\}\sigma_i, \cdot], (\bar{x}')[y\{\cdot\}\sigma_j, \cdot], \text{STREQ}(\bar{x}_y, \bar{x}'_y) \]

This rule is repeated for every \(1 \leq j < i \leq n\). Based on \(\succ_{str}\), we use stratified negation to define the successor relation \(\text{Succ}_{str}\). To do that, we define the binary relation \(\text{NotSucc}_{str}\) that holds tuples \((\bar{x}', \bar{x})\) where \(\bar{x}'\) is not the successor of \(\bar{x}\) with respect to \(\succ_{str}\).

\[ \text{NotSucc}_{str}((\bar{x}_1, \bar{x}_2), (\bar{x}_1, \bar{x}_3), (\bar{x}, \bar{x}_2)) \]

and then,

\[ \text{Succ}_{str}((\bar{x}, \bar{x}'), (\bar{x}, \bar{x}'), \neg \text{NotSucc}_{str}((\bar{x}, \bar{x}')) \]

Note that the first string in the lexicographic order is always \(\epsilon\) (since for every \(D\), its extended active domain contains \(\epsilon\)). Therefore we have:

\[ \text{FIRST}(\bar{x}) \leftarrow \text{STR}(\bar{x}), (\bar{x})[x\{\epsilon\}] \]

To define the relation \(\text{Succ}_{mix}\) we need to find the last string in the extended active domain of the input instance. For this purpose, we define the relation \(\text{Last}_{str}\) as follows:

\[ \text{NotLast}(\bar{x}) \leftarrow \text{STR}(\bar{x}), \text{STR}(\bar{x}'), \succ_{str}(\bar{x}'', \bar{x}) \]

\[ \text{Last}_{str}(\bar{x}) \leftarrow \text{STR}(\bar{x}), \neg \text{NotLast}(\bar{x}) \]

We can now define the relation \(\text{Succ}_{mix}\). Note that in the sequel we define a strict total order \(\succ_{spn}\) on the spans in the extended active domain which is the lexicographic order (see Comment 3.4.3). The first span according to the lexicographic order is always \([1, 1]\) regardless of the input instance. We therefore use the following rule according to which the successor of the last string in the extended active domain of the input is the first span.

\[ \text{Succ}_{mix}(y, \bar{x}) \leftarrow \text{Last}_{str}(\bar{x}), (\bar{x})[y\{\epsilon\}.*] \]

Similarly to the definition of \(\text{Succ}_{str}\), we define \(\text{Succ}_{spn}\) by defining a strict total order \(\succ_{spn}\) on the spans in the extended active domain. Note that the span \(y\) follows \(y'\) in this order if either (1) \(y\) begins after \(y'\) begins or (2) they both start in the same position but \(y\) ends strictly after \(y'\) ends. For case (1) we have:

\[ \succ_{spn}(y, y') \leftarrow \text{STR}(\bar{x}), (\bar{x})[z\{\cdot\}y\{\cdot\}.*], (\bar{x}')[z'\{\cdot\}y'\{\cdot\}.*], (\bar{x})[z\{\cdot\}y\{\cdot\}.+.\}.+.*] \]

and for (2):

\[ \succ_{spn}(y, y') \leftarrow \text{STR}(\bar{x}), (\bar{x})[y\{\cdot\}y'\{\cdot\}.+.\}.+.*] \]

The relation \(\text{Succ}_{spn}\) is defined based on \(\succ_{spn}\) in a similar way to how we have defined \(\text{Succ}_{str}\) based on \(\succ_{str}\). Moreover, the relation \(\text{Last}\) is defined based on \(\succ_{spn}\) in a similar way to how \(\text{Last}_{str}\) was defined based on \(\succ_{spn}\). Therefore we skip these definitions. \(\square\)
We can now conclude the proof of Theorem A.1.1. Assume that $Q$ is computable in polynomial time. Let $D'$ be the untyped encoding of $D$ where $\iota$ from Lemma A.2.1 is the identity. Let $D''$ be the mixed encoding that corresponds with $D'$. Let $P'_Q$ be the $\mathsf{Spl}(\mathsf{RGX})$ program obtained from Lemma A.2.4. It holds that $P'_Q(D'') = Q^+(D')$. Due to the definition of $Q^+$ and since $\iota$ is the identity we have that $P'_Q(D'') = Q(D)$. Due to Lemma A.2.5 we can construct $D''$ from $D$ using a $\mathsf{Spl}(\mathsf{RGX})$ program $P'$. Combining $P'_Q$ with $P'$ into a single $\mathsf{Spl}(\mathsf{RGX})$ program completes the proof.
 iii

Technion - Computer Science Department - Ph.D. Thesis PHD-2019-10 - 2019
Data Complexity

Data complexity refers to the measures used to quantify the difficulty of processing, storing, or querying data. It is often used in computer science, particularly in database management and data mining, to understand and optimize the performance of data-intensive applications.

In the context of this dissertation, the focus is on the complexity of certain operations performed on data structures, such as trees or graphs. The analysis aims to identify patterns and strategies to improve the efficiency of these operations.

For instance, the dissertation explores the complexity of certain tree operations, such as searching for a particular node or traversing the entire tree. It also examines how the structure of the data can affect the complexity of these operations, leading to insights on optimizing data structures for specific use cases.

The research contributes to the field by providing a deeper understanding of the underlying principles that govern data complexity and by offering practical solutions for optimizing data processing tasks.

Overall, the dissertation is a valuable resource for researchers and practitioners interested in the theoretical and practical aspects of data complexity.
The title of the document is "ניקולר". The content appears to be in Hebrew and includes various technical terms and phrases related to information extraction and document spanners. The text is fragmented and appears to be part of a larger document or thesis, likely discussing methods and techniques in the field of computer science and information technology.

The first line mentions "University of California, Berkeley" and "techextraction Projects". This suggests the content might be related to research or projects conducted at the University of California, Berkeley, focusing on information extraction.

The text includes phrases like "EXTRACTORS", "PRIMITIVE", "INFORMATION EXTRACTION". These terms are commonly associated with natural language processing and data mining, indicating that the document likely deals with techniques for extracting useful information from natural language text.

The document seems to be part of a larger academic work, possibly a thesis or research paper, given the formal style and the presence of sections like "DOCUMENT SPANNERS" and "REGULAR SPANNERS". These terms refer to specific methods used in text analysis, where "spans" are segments of text that are identified and analyzed for various purposes, such as classification or information retrieval.

The text contains references to "Technion - Computer Science Department - Ph.D. Thesis PHD-2019-10 - 2019", indicating that this document is part of a Ph.D. thesis from the Technion, a well-known institution in Israel.

Overall, the document appears to be a technical and academic piece, likely intended for researchers or practitioners in the fields of computer science and natural language processing.


הטיפונים החיזוביים של ישראל
על תשובות מחודך תכנית

ת羀ור על מחקר

ליאת פטרוגיד

הוזמן לסכט הטכני --- מכון טכנולוגי לישראל
אבר החשע"ש לדינמה אוגוסט 2019
הסיבוכיות החישובית שלシアילות
על יחסם מתודת סקט

ליאת פטרפורניד