Complexity Bounds for Relational Algebra over Document Spanners

Liat Peterfreund  
liatpf@cs.technion.ac.il  
Technion  
Haifa, Israel

Dominik D. Freydenberger  
ddfy@ddfy.de  
Loughborough University  
Loughborough, United Kingdom

Benny Kimelfeld  
bennyk@cs.technion.ac.il  
Technion  
Haifa, Israel

Markus Kröll  
kroell@dbai.tuwien.ac.at  
TU Wien  
Vienna, Austria

ABSTRACT
We investigate the complexity of evaluating queries in Relational Algebra (RA) over the relations extracted by regex formulas (i.e., regular expressions with capture variables) over text documents. Such queries, also known as the regular document spanners, were shown to have an evaluation with polynomial delay for every positive RA expression (i.e., consisting of only natural joins, projections and unions); here, the RA expression is fixed and the input consists of both the regex formulas and the document. In this work, we explore the implication of two fundamental generalizations. The first is adopting the “schemaless” semantics for spanners, as proposed and studied by Maturana et al. The second is going beyond the positive RA to allowing the difference operator.

We show that each of the two generalizations introduces computational hardness: it is intractable to compute the natural join of two regex formulas under the schemaless semantics, and the difference between two regex formulas under both the ordinary and schemaless semantics. Nevertheless, we propose and analyze syntactic constraints, on the 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. Unlike the previous work on RA over regex formulas, our technique is not (and provably cannot be) based on the static compilation of regex formulas, but rather on an ad-hoc compilation into an automaton that incorporates both the query and the document. This approach also allows us to include black-box extractors in the RA expression.

CCS CONCEPTS
• Theory of computation → Models of computation; Formal languages and automata theory; Design and analysis of algorithms; Fixed parameter tractability; Database theory; Data modeling; Database query languages (principles); Database query processing and optimization (theory); Logic and databases; • Information systems → Information extraction.

KEYWORDS
Information Extraction, Document Spanners, Regular Expressions, Relational Algebra, Polynomial Delay

ACM Reference Format:

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 is Information Extraction (IE)—the extraction of structured data from text. IE arises in a large variety of domains, including biology and biomedical analysis, social media analysis, cyber security,1 system and network log analysis, and business intelligence, to name a

1See, e.g., the TA-COS workshop at http://www.ta-cos.org/.
few [4, 27]. 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 [5, 18], which exposes a collection of atomic (sometimes called “primitive”) 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 [29], user-defined functions provide the atomic extractors, and Datalog is used for relational manipulation. In DeepDive [26], 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 thereof.

**Document spanners.** In this work, we explore complexity aspects of IE within the framework of document spanners (or just spanners for short) [8]. In this framework, a document is a string over a fixed finite alphabet, and a spanner extracts from every input document a relation of intervals within the document. An interval, called span, is represented by its starting and ending indices in the document.

An example of a spanner is a regex formula, which is a regular expression with capture variables that correspond to the relational attributes. The most studied language for specifying spanners is that of the regular spanners: the closure of regex formulas under the classic relational algebra: projection, natural join, union, and difference [8]. Equally expressive formalisms include non-recursive Datalog over regex formulas [9] and the variable-set automaton (vset-automaton for short), which is a nondeterministic finite-state automaton (NFA) that can open and close variables while running.

Since the framing of the spanner framework, there has been a considerable effort to delineate the computational complexity of spanner evaluation, with a special focus on the regular representations (regex formulas and vset-automata) of the atomic extractors.

Florenzano et al. [10] studied the data complexity (where the spanner is fixed and the input consists of only the document), and so did Peterfreund et al. [25] who showed that the closure of regex formulas under Datalog characterizes the class of polynomial-time spanners. Freydenberger et al. [11–13] studied the combined complexity (where the input consists of both the query and the document) for conjunctive queries, and unions of conjunctive queries, over spanners.

More recently, Amarilli et al. [1] presented an evaluation algorithm with tractability properties under both data and combined complexity; we further discuss this algorithm later on.

For complexity analysis, there are important advantages to yardsticks that take the atomic extractors (e.g., regex formulas or vset-automata) as input, rather than regarding them small or fixed. First, the size of these extractors 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 [21, 22, 33], where the number of states may reach tens of thousands to match the expressiveness of the numeric parameters [33]. Another advantage of regarding the atomic extractors as input is more technical: polynomial-time combined complexity allows to incorporate vset-automata whose size may depend on the input document. This approach allows to establish tractability even if we join with schemaless spanners that cannot be represented as RA expressions over regular spanners, such as string equality [13].

**Schema-based functionality vs schemaless sequentiality.** As defined by Fagin et al. [8], the spanners are schema-based in the sense that every spanner is associated with a fixed and finite set X of variables, playing the roles of attributes in relational databases, so that every tuple they extract from a document assigns a value to each variable of X. The regex formulas conform to this property in the sense that every parse tree contains exactly one occurrence of each variable; such regex formulas are said to be functional. Freydenberger [11] 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 [8] and vset-automata [12]. 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). Beyond that, functional vset-automata (and regex formulas) possess various desired tractability features [13]. First, they can be evaluated with polynomial delay under combined complexity. Second, the **natural join** of two functional vset-automata can be compiled in polynomial time into one functional vset-automaton, and so can the **union** of two vset-automata and the **projection** of a vset-automaton to a subset of its variables. Consequently, every combination of functional vset-automata can be evaluated with polynomial delay, as long as this combination is via the **positive** operators of the relational algebra.

More recently, Maturana et al. [19] introduced a schemaless version of spanners that allows for incomplete extraction
from documents, in the spirit of the SPARQL model [23]. There, 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, sequential vset-automata can be evaluated with polynomial delay under combined complexity [19]. In fact, the aforementioned algorithm of Amarilli et al. [1] enumerates with polynomial delay under combined complexity, and, under data complexity, with constant delay following a linear preprocessing of the document.\footnote{This is the spanner analog of a recent line of work on the enumeration complexity of database and string queries [2, 3, 20, 28].} Since functional vset-automata are also sequential, this algorithm also applies to the schema-based spanners, and improves upon (and, in fact, generalizes the applicability of) the constant-delay algorithm of Fornengo et al. [10].

**Contribution.** The state of affairs leaves open two fundamental questions 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?
- Does the tractability extend beyond the positive operators (in either the schema-based or schemaless case)? In particular, can we enumerate with polynomial delay the difference between two functional vset-automata?

We prove that the answers to both questions are negative. More specifically, it is NP-complete to determine whether the natural join of two sequential regex formulas is nonempty (Theorem 3.1), and it is NP-complete to determine whether the difference between two given functional regex formulas is nonempty (Theorem 4.1).

We formulate various syntactic restrictions that allow to avoid 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 and difference. 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; yet, the natural join of two disjunctive-functional vset-automata can be compiled into a disjunctive-functional vset-automaton in polynomial time (hence, evaluated with polynomial delay).

In contrast to the natural join, the tractability of the difference between 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 [16]. 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.

In summary, our complexity upper bounds are established in two main approaches. The first is based on a document-independent compilation of the input vset-automata (or regex formulas) into a new vset-automaton. The second is based on a compilation of both the input vset-automata and the input document into a new, ad-hoc vset-automaton. We refer to the first approach as static compilation and to the second as ad-hoc compilation.

We compose our tractability results into more general queries by proposing a new complexity measure that is specialized to spanners. Recall that the evaluation problem has three components: the document, the atomic spanners (e.g., regex formulas), and the relational algebra that combines the atomic spanners, 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 [32] where the document is fixed and the rest are 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 ad-hoc compilation, we can incorporate there any polynomial-time spanner, as long as its dimension is bounded by a constant.

**Organization.** The rest of the paper is organized as follows. In Section 2, we present the basic terminology and concepts. We investigate the complexity of the natural-join operator in Section 3 and the difference operator in Section 4. We extend our development to the extraction complexity in Section 5, and conclude in Section 6. To meet space constraints, some of the proof are given in the full version of the paper [24].

## 2 PRELIMINARIES

We first introduce the main definitions and terminology, mainly from the literature on document spanners [8, 19].

### 2.1 Document Spanners

**Documents and spans.** We fix a finite alphabet $\Sigma$ of symbols. By a *document* or *string* we refer to a finite sequence $d = \sigma_1 \cdots \sigma_n$ over $\Sigma$ (that is, each $\sigma_i$ is in $\Sigma$), that is, a member...
of $\Sigma^*$. The length $n$ of the document $d = \sigma_1 \cdots \sigma_n$ is denoted by $|d|$. A span is a pair $[i, j]$ of indices $1 \leq i \leq j \leq n + 1$ that marks a substring of $d$. The term $d_{[i, j)}$ denotes the substring $\sigma_i \cdots \sigma_{j-1}$.

Note that $d_{[i, j)}$ is the empty string $\epsilon$, and that $d_{[1, n+1)} = d$.

Note also that the spans $[i, i)$ and $[j, j)$, where $i \neq j$, are different objects, even though the substrings $d_{[i, i)}$ and $d_{[j, j)}$ are equal.

We denote by spans the set of all spans of all strings, that is, all expressions $[i, j)$ where $1 \leq i \leq j$. By spans($d$) we denote the set of all spans of $d$.

**Schemaless spanners.** We assume a countably infinite set $\text{Vars}$ of variables, and assume that $\text{Vars}$ is disjoint from $\Sigma$ and $\Sigma^*$. A schemaless (document) spanner is a function that maps each document into a finite collection of tuples (referred to as mappings) that assign spans to variables.

More formally, a mapping to $d$ is a function $\mu$ from a finite set of variables, called the domain of $\mu$ and denoted $\text{dom}(\mu)$, into spans($d$). A schemaless spanner is a function $\mu$ that maps every document $d$ into a finite set $P(d)$ of mappings.

For a schemaless spanner $\mu$ and a document $d$, different mappings in $P(d)$ may have different domains. This stands in contrast to the (schema based) spanners of Fagin et al. [8], where $P$ is such that there exists a set $\text{Vars}$ of variables where every document $d$ and mapping $\mu \in P(d)$ satisfy $\text{dom}(\mu) = \text{Vars}$; in this case, we may refer to $P$ as a schema-based spanner.

**Example 2.1.** Let $\Gamma$ be the alphabet consists of lower-case and uppercase English letters: a, b, c, d, named and symbols: that stands for whitespace, '.' and '@'. Let $\Delta = \{\epsilon\}$ where $\epsilon$ stands for end of line. The input document $d_{\text{Students}}$ over $\Gamma \cup \Delta$ given in Figure 1 holds personal information on students. (Some of the positions are marked underneath 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 spaces in between these elements. The schemaless document spanner $P_{\text{StudInfo}}$ extracts from the input document $d_{\text{Students}}$ the following set of mappings, 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>$\mu_1$:</td>
<td>(1, 7)</td>
<td>(8, 19)</td>
<td>(20, 29)</td>
</tr>
<tr>
<td>$\mu_2$:</td>
<td>(30, 37)</td>
<td>(46, 56)</td>
<td>(38, 45)</td>
</tr>
<tr>
<td>$\mu_3$:</td>
<td>(57, 62)</td>
<td>(63, 69)</td>
<td>(78, 89)</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 the next sections, 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 schema based semantics of Fagin et al. [8] who use $\llbracket \cdot \rrbracket$. This distinction is critical in the case of the vset-automata that we define later on.

### 2.2 Regex Formulas

One way of representing a schemaless spanner is by means of a regex formula, which is a regular expression with capture variables, as allowed by the grammar

$$\alpha := \emptyset \mid e \mid \sigma \mid (\alpha \lor \alpha) \mid (\alpha \cdot \alpha) \mid \alpha^* \mid x[\alpha]$$

where $\sigma \in \Sigma$ and $x \in \text{Vars}$. For convenience, we sometimes put regex formulas in parentheses and also omit parentheses, as long as the meaning remains clear. For operator precedence, we assume that $\lor$ comes before $\cdot$, which comes before $\lor$. We denote by Vars($\alpha$) the set of variables that appear in $\alpha$. By RGX we denote the class of regex formulas.

Following Maturana et al. [19], we interpret regex formulas as schemaless spanners in the following manner. The following grammar defines 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 $d$.

- $[\emptyset](d) := \emptyset$.
- $[e](d) := \{(i, i, \emptyset) \mid i = 1, \ldots, n\}$.
- $[\sigma](d) := \{(i, i + 1, \emptyset) \mid \sigma_1 = \sigma\}$.
- $[x[\alpha]](d) := \{(i, j, \mu \cup \{x \mapsto [i, j]\}) \mid (i, j), \mu \in [\alpha](d) \text{ and } x \notin \text{dom}(\mu)\}$.
- $[\alpha \lor \alpha](d) := [\alpha](d) \lor [\alpha](d)$.
- $[\alpha \cdot \alpha](d) := \{(i, j, \mu_1 \cdot \mu_2) \mid \exists \iota \text{ s.t. } (i, \iota', \mu_1) \in [\alpha](d), (\iota', j, \mu_2) \in [\alpha](d), \text{ and } \text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset\}$.
- $[\alpha^*](d) := \bigcup_{i=0}^{\infty} [\alpha^i](d)$ where $\alpha^i$ stands for the concatenation of $i$ copies of $\alpha$.

The result of applying $\alpha$ to $d$ is then defined as follows.

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

We denote by $[\text{RGX}]$ the class of schemaless spanners that can be expressed using the regex formulas. Similarly, for every subclass $R \subseteq \text{RGX}$, we denote by $[\text{R}]$ the class of spanners expressible by an expression in $R$.

**Syntactic restrictions.** Fagin et al. [8] introduced the class of regex formulas that are interpreted as schema-based spanners, namely the functional regex formulas. To define functional regex formulas, we first use the following inductive definition. A regex formula $\alpha$ is functional for a set $V \subseteq \text{Vars}$ of variables if:

- $\alpha \in \Sigma^*$ and $V = \emptyset$.
- $\alpha = \alpha_1 \lor \alpha_2$ and each $\alpha_i$ is functional for $V$.
- $\alpha = \alpha_1 \cdot \alpha_2$ and there exists $V_1 \subseteq V$ such that $\alpha_1$ is functional for $V_1$ and $\alpha_2$ is functional for $V \setminus V_1$.
where spanners, as defined by Maturana et al. [19] as a schema-
3 acceptance of the vset-automata of Fagin et al. [8].

(abbreviated since the variables

Example 2.1: $\beta$ and $\gamma$ are the set of

\[ \alpha = \alpha_1 \land \alpha_2, \quad \text{we have } Vars(\alpha_1) \cap Vars(\alpha_2) = \emptyset. \]

\[ \alpha = x(\alpha_0) \text{ and } \alpha_0 \text{ is functional for } \forall x \in Vars(\alpha). \]

We denote by funcRGX and seqRGX the classes of functional and sequential regex formulas, respectively.

As shown by Maturana et al. [19], it holds that funcRGX \(\subseteq\) seqRGX. That is, every functional regex formula is sequential, but some sequential regex formulas are not functional, as the next example illustrates.

Example 2.2. Let us define the following regex formulas over the alphabet \(\Gamma \cup \Delta\) from Example 2.1:

\[
\alpha_{\text{mail}} := x_{\text{mail}}(y@y)\,y \\
\alpha_{\text{name}} := (x_{\text{first}}(\delta),x_{\text{last}}(\delta)) \lor (x_{\text{last}}(\delta)) \\
\alpha_{\text{phone}} := x_{\text{phone}}(\beta^+) \\
\]

where \(y := (a \lor \cdots \lor z)^+\), \(\delta := (A \lor \cdots \lor Z) \cdot (a \lor \cdots \lor z)^+\), and \(\beta := (0 \lor \cdots \lor 9)^+\).

Based on the previous regex formulas, we define the regex formula that represents the schemaless spanner \(P_{\text{StudentInfo}}\) from Example 2.1:

\[
\alpha_{\text{info}} := \Gamma^* \cdot (\varepsilon \lor \varepsilon) \cdot \alpha_{\text{name}} \cdot (\alpha_{\text{phone}} \lor \varepsilon) \cdot \alpha_{\text{mail}} \leftrightarrow \Gamma^* \\
\]

Note that this is regex formula is sequential but not functional since the variables \(x_{\text{first}}\) and \(x_{\text{phone}}\) are optional.

2.3 Vset-Automata

In addition to regex formulas, we use the \textit{variable-set automata} (abbreviated \textit{vset-automata}) for representing schemaless spanners, as defined by Maturana et al. [19] as a schemaless adaptation of the vset-automata of Fagin et al. [8].

A \textit{vset-automaton}, VA for short, is a tuple \((Q,q_0,F,\delta)\), where \(Q\) is set of states, \(q_0 \in Q\) is the initial state, \(F \subseteq Q\) is the set of accepting states, 4 and \(\delta\) is a transition relation consisting of \textit{epsilon transitions} of the form \((q,\varepsilon,p)\), \textit{letter transitions} of the form \((q,\sigma,p)\) and \textit{variable transitions} of the form \((q,v\sigma,p)\) or \((q,v\sigma,\varepsilon)\) where \(q,p \in Q, \sigma \in \Sigma, \text{and } v \in Vars.\)

The symbols \(v\sigma\) and \(v\sigma\) are special symbols to denote the opening or closing of a variable \(v\). We define the set \(Vars(A)\) as the set of all variables \(v\) that are mentioned in some transition of \(A\). For every finite set \(V \subseteq Vars\) we define the set \(\Gamma_V := \{v\sigma,v\sigma : v \in V\}\) of variable operations.

A run \(\rho\) over a document \(d = \sigma_1 \cdots \sigma_n\) is a sequence of the form

\[
(q_0,i_0) \xrightarrow{\sigma_1} \cdots (q_{m-1},i_{m-1}) \xrightarrow{\sigma_m} (q_m,i_m) \\
\]

where:

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

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

\[
\begin{align*}
&v \text{ is opened (or closed) at most once;} \\
&\text{if } v \text{ is opened at some position } i \text{ then it is closed at some position } j \text{ with } i \leq j; \\
&\text{if } v \text{ is closed at some position } j \text{ then it is opened at some position } i \text{ with } i \leq j.
\end{align*}
\]

A run is called \textit{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 \([i_j, i_{j+1}]\) where \(o_{j+1} = v\sigma\) and \(o_j = v\sigma\).

The result \([A](d)\) 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 VA \textit{sequential} if all of its accepting runs are valid, and it is called \textit{functional} if each such run also include all of its variables \(Vars(A)\). Note that sequential VAs correspond to schemaless spanners, whereas functional correspond to complete.

In what follows, we assume that our VAs are \textit{trimmed}, that is, for every state \(q\) we have that (1) \(q\) is reachable from the initial state, and (2) there is at least one accepting state that can be reached from \(q\).

expressive power by simulating a multiple accepting states automaton by a single accepting state automaton with epsilon transitions.
Observe that given a VA we can construct an equivalent trimmed one in linear time.

**Example 2.3.** Let A be the following sequential VA:

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

Omitting the transition from \( q_0 \) to \( q_2 \) results in a functional VA. The same schemaless spanner as that represented by A is given by the sequential regex formula \( \alpha := (\Sigma^+ x (\Sigma^+)) \lor (\Sigma^+) \) where \( \Sigma^+ \) stands for \( \Sigma \cdot \Sigma^* \).

### 2.4 Algebraic Operators

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 correspondents of the relational-algebra operators are defined similarly to the SPARQL formalism [23]. 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:** The projection \( P := \pi_Y P_1 \) is defined by \( P(d) = \{ \mu \mid Y \subseteq \mu \in P_1(d) \} \) where \( \uparrow \) stands for 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 \( \mu_1 \).

We allow the use of these operators for spanners represented by regex formulas or VAs and also for more complex spanner representations, e.g., \([A_1] \Rightarrow [A_2]\). In this case, we use the abbreviated notation \([A_1] \Rightarrow [A_2]\) instead of \([A_1] \Rightarrow [A_2]\). We make the clear note that when the above operators are applied on schema-based spanners, they are the same as those of Fagin et al. [8].

**Example 2.4.** Let us consider our input document \( d_{\text{Students}} \) from Figure 1. Assume one wants to filter out from the results obtained by applying the spanner \( P_{\text{Student}} \) from Example 2.2 on \( d_{\text{Students}} \) the mappings that correspond with students from universities within the UK. It is given 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:

\[ \alpha_{UKm} \equiv (\varepsilon \lor (\Gamma^+ \cdot \leftarrow \cdot \Gamma^+) \cdot \chi_{\text{mail(y@y.uk)}} \cdot \leftarrow \cdot \Gamma^+) \]

where \( y \) is as defined in Example 2.2. In this case, the desired output is given by \([\alpha_{data} \setminus \alpha_{UKm}](d_{\text{Students}})\) which consists of the mappings \( \mu_1 \) and \( \mu_2 \) from Example 2.1. \( \square \)

### 2.5 Complexity

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

Under the combined complexity, “polynomial time” is not a proper yardstick of efficiency for evaluating \([q](d)\), since this set can contain exponentially many mappings. We thus use efficiency yardsticks of enumeration [17]. In particular, our evaluation algorithm takes \( q \) and \( d \) as input, and it outputs all the mappings of \([q](d)\), one by one, without duplicates. The algorithm runs in polynomial total time if its execution time is polynomial in the combined size of \( q \) and \( d \). The delay of the evaluation algorithm refers to the maximal time that passes between every two consecutive mappings. A well-known observation is that polynomial total time implies polynomial delay (but not necessarily vice versa), and that NP-hardness of the nonemptiness problem implies that no evaluation algorithm runs in polynomial delay, or else \( P = NP \).

While deciding whether \([q](d) \neq \emptyset\) is \( NP \)-hard whenever \( q \) is given as a VA [11], this is not the case for sequential (and hence functional) VA:

**Theorem 2.5.** [1] Given a sequential VA \( A \) 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 \([q_1] \equiv [q_2]\), that is, \([q_1]\) and \([q_2]\) are identical. Note that the translation of functional and sequential regex formulas to equivalent functional and sequential VAs, respectively, can be done in linear time [13, 19]. Hence, our lower bounds are usually shown for the nonemptiness of regex formulas and our upper bounds for the evaluation of VAs.

### 3 THE NATURAL-JOIN OPERATOR

To establish complexity upper bounds on the evaluation of schema-based spanners, Freydenberger et al. [13] used
Theorem 3.1. The following problem is NP-complete. Given two sequential regex formulas \( \gamma_1 \) and \( \gamma_2 \) and an input document \( d \), is \( \forall \gamma_1 \Rightarrow \gamma_2 \)(\(d \)) 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 show a reduction from 3-CNFSatisfiability which is also known as 3SAT [15]. 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 \( \tau : \{x_1, \ldots, x_n \} \rightarrow \{t, f\} \) that satisfies \( \varphi \). 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 \( \forall \gamma_1 \Rightarrow \gamma_2 \)(\(d \)) \( \neq \emptyset \), where \( d \) is the document that consists of a single letter \( a \).

To construct \( \gamma_1 \) and \( \gamma_2 \), we associate every variable \( x_j \) with 2\( m \) corresponding capture variables \( x_{j,1}^{\ell}, \ldots, x_{j,2}^{\ell} \) for \( 1 \leq j \leq m \) and \( \ell \in \{t, f\} \). Then define 
\[
\gamma_1 := \gamma_{x_1} \cdots \gamma_{x_n} \cdot a,
\]
where 
\[
\gamma_{x_j} := (x_{j,1}^{t} \cdot x_{j,1}^{f}) \cup (x_{j,2}^{t} \cdot x_{j,2}^{f}).
\]

Intuitively, \( \gamma_{x_j} \) verifies that the assignment to \( x_j \) is consistent in all of the clauses. We then define 
\[
\gamma_2 := a \cdot (\delta_1 \cdots \delta_m)
\]
where \( \delta_1 \) is the disjunction of regex formulas \( \beta \) such that \( \beta = x_{j,1}^{t} \cdot x_{j,1}^{f} \) if \( \neg x_j \) appears in \( C_j \), and \( \beta = x_{j,2}^{t} \cdot x_{j,2}^{f} \) if \( x_j \) appears in \( C_j \). Intuitively, \( \gamma_2 \) verifies that at least one disjunct in each clause is evaluated true.

Let us consider the following example where 
\[
\varphi := (x \lor y \lor z) \land (\neg x \lor y \lor \neg z).
\]
In this case, we have 
\[
\delta_1 = x_{1,1}^{t} \cup y_{1,1}^{t} \cup z_{1,1}^{t}
\]
and, therefore, 
\[
\gamma_2 := a \cdot (x_{1,1}^{t} \lor y_{1,1}^{t} \lor z_{1,1}^{t})
\]
\[
\cdot (x_{1,2}^{t} \lor y_{1,2}^{t} \lor z_{1,2}^{t}).
\]

We also have 
\[
\gamma_1 := (x_{2,1}^{t} \lor x_{2,1}^{f} \lor x_{2,2}^{f} \lor y_{2,1}^{f} \lor y_{2,2}^{f} \ordash z_{2,1}^{f} \lor z_{2,2}^{f}) \cdot a.
\]

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

The "only if" direction. Suppose that \( \forall \gamma_1 \Rightarrow \gamma_2 \)(\(d \)) is nonempty. In this case, a satisfying assignment \( \tau \) to \( \varphi \) is encoded by the domain of \( \gamma_2 \) in the following way: if \( x_{j,\ell}^{t} \in \text{dom}(\mu_2) \) then \( \tau(x_j) = \ell \). Observe that \( \tau \) is well defined since dom(\(\mu_1\)) \cap dom(\(\mu_2\)) = \emptyset.

In our example, the mapping \( \mu_1 \in \forall \gamma_1 \)(\(a \)) with 
\[
\text{dom}(\mu_1) = \{x_{1,1}^{t}, x_{1,2}^{t}, y_{1,1}^{f}, y_{1,2}^{f}, z_{1,1}^{f}, z_{1,2}^{f}\}
\]
and the mapping \( \mu_2 \in \forall \gamma_2 \)(\(a \)) with 
\[
\text{dom}(\mu_2) = \{x_{2,1}^{t}, x_{2,2}^{t}, y_{2,1}^{f}, y_{2,2}^{f}, z_{2,1}^{f}, z_{2,2}^{f}\}
\]
are compatible, and the satisfying assignment \( \tau \) is encoded by dom(\(\mu_2\)) and is given by \( \tau(x) = f, \tau(y) = t \) and \( \tau(z) = t \).

The "if" direction. If there is a satisfying assignment \( \tau \) to \( \varphi \), then define the mappings \( \mu_1 \in \forall \gamma_1 \)(\(d \)) and \( \mu_2 \in \forall \gamma_2 \)(\(d \)) by 
\[
x_{j,\ell}^{t} \in \text{dom}(\mu_2) \quad \text{whenever} \quad j = \tau(x_j) \quad \text{and} \quad x_{j,\ell}^{f} \in \text{dom}(\mu_1) \quad \text{whenever} \quad j \neq \tau(x_j).
\]

These mapping are compatible, since dom(\(\mu_1\)) \cap dom(\(\mu_2\)) = \emptyset. We conclude that \( \forall \gamma_1 \Rightarrow \gamma_2 \)(\(d \)) is nonempty.

We conclude the NP-hardness of the problem of determining whether \( \forall \gamma_1 \Rightarrow \gamma_2 \)(\(d \)) is nonempty, as claimed. \( \square \)

In what follows, we suggest two different approaches to deal with this hardness.
Lemma 3.2. The following problem is FPT when parameterized by $|\text{Vars}(A_1) \cap \text{Vars}(A_2)|$. Given two sequential VAs $A_1$ and $A_2$, construct a sequential VA that is equivalent to $A_1 \Leftrightarrow A_2$.

Since we have a polynomial delay algorithm for the evaluation of sequential VAs (Theorem 2.5) and the size of the resulting VA is FPT in $|\text{Vars}(A_1) \cap \text{Vars}(A_2)|$, we have the following immediate conclusion.

Theorem 3.3. Given two sequential VAs $A_1$ and $A_2$ and a document $d$, one can evaluate $[\lceil A_1 \Leftrightarrow A_2 \rceil](d)$ with FPT delay parameterized by $|\text{Vars}(A_1) \cap \text{Vars}(A_2)|$.

In the rest of this section, we discuss the proof of Lemma 3.2. As shown by Freydenberger et al. [13] if $A$ is a functional VA 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 $\text{Vars}(A)$, and $q$ is a single accepting state.

Example 3.4. Let us examine the following two accepting runs of the sequential VA $A$ from Example 2.3 on the input document $d := a$:

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

$$
\rho_2 := (q_0, 1) \xrightarrow{a} (q_2, 2)
$$

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

Example 3.5. The sequential VA $A$ from Example 2.3 is semi-functional for $x$ because $\tilde{c}_q(x) = d$, as reflected from the runs $\rho_1$ and $\rho_2$ presented in the previous example. However, the following equivalent sequential VA $A'$ is semi-functional for $x$:

$$
\begin{array}{c}
\Sigma \\
\xrightarrow{x} \\
\xrightarrow{\Sigma} \\
\xrightarrow{x} \\
\xrightarrow{\Sigma} \\
\xrightarrow{x} \\
\xrightarrow{\Sigma} \\
\xrightarrow{x} \\
\xrightarrow{\Sigma} \\
\end{array}
\begin{array}{c}
q_0 \\
q_1 \\
q_2 \\
q_3 \\
q_4 \\
q_5 \\
q_6 \\
q_7 \\
q_8 \\
\end{array}
\end{array}
$$

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 VA $A$, 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 VA that is semi-functional for $X$. We show that the total runtime is FPT parameterized by $|X|$.

Lemma 3.6. Given a sequential VA $A$ and $X \subseteq \text{Vars}(A)$, one can construct in $O(2^{|X|}(n + m))$ time a sequential VA $A'$ that is equivalent to $A$ and semi-functional for $X$ where $n$ is the number of states of $A$ and $m$ is the number of its transitions.

Example 3.7. The sequential VA $A'$ from Example 3.5 can be obtained from the automaton $A$ from Example 2.3 by replacing $q_2$ with two states $q_2^o$ and $q_2^c$ such that $q_2^o$ corresponds with the paths in from $q_0$ to $q_2$ in which variable $x$ was unseen and $q_2^c$ corresponds with the paths in 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 Footnote 4 in the definition of a VA and note that, as in the previous example, there are cases where, to be semi-functional, a VA must have more than a single accepting state.

If two sequential VAs are semi-functional for their common variables, their join can be computed efficiently:
**Lemma 3.8.** Given two sequential VAs $A_1$ and $A_2$ that are semi-functional for $\text{Vars}(A_1) \cap \text{Vars}(A_2)$ one can construct in polynomial time a sequential VA $A$ 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 VAs presented by Freydenberger et al. [13, Lemma 3.10]. What allow 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 VAs over $\text{Vars}(A_1) \cap \text{Vars}(A_2)$.

We can now move to compose the proof of Lemma 3.2: Given two sequential VAs $A_1$ and $A_2$, we invoke the algorithm from Lemma 3.6 and obtain two equivalent sequential VAs $\tilde{A}_1$ and $\tilde{A}_2$, respectively, such that each $\tilde{A}_i$ is semi-functional for $\text{Vars}(A_1) \cap \text{Vars}(A_2)$. Then, we use Lemma 3.8 to join $A_1$ and $A_2$. Note that the runtime is indeed FPT parametrized by $\text{Vars}(A_1) \cap \text{Vars}(A_2)$.

### 3.2 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 formula $\gamma_1, \ldots, \gamma_n$. 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 $z[\Sigma^*] \cdot (x[\Sigma^*] \lor y[\Sigma^*])$ is sequential, yet it is not disjunctive functional. It also holds that every functional regex formula is 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.

Equivalently, a **disjunctive functional** VA $A$ is the sequential VA whose states are the disjoint union of the states of a finite number $n$ of functional VAs $A_1, \ldots, A_n$ 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 semantic, based on the following proposition.

**Proposition 3.9.** The following hold:

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

Since funcRGX corresponds with schema-based spanners whereas 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.

**Example 3.10.** 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 VA:

![Diagram](image-url)

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

We record this in the following observation.

**Observation 3.11.** 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 VA $A$ with $3n + 1$ states such that each of its equivalent disjunctive functional VA 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 VAs lies in the fact that we can compile the join of two disjunctive functional VAs efficiently into a disjunctive functional VA.

**Proposition 3.12.** Given two disjunctive functional VAs $A_1$ and $A_2$, one can construct in polynomial time a disjunctive functional VA $A$ 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 VAs for the join [13, Lemma 3.10].
Since disjunctive functional is a restricted type of sequential VA, we conclude the following.

**Corollary 3.13.** Given two disjunctive functional VAs $A_1$ and $A_2$ and a input document $d$, one can enumerate the mappings of $\|A_1 \Rightarrow A_2\|(d)$ in polynomial delay.

### 4 THE DIFFERENCE OPERATOR

When we consider the class of functional VAs, we know that we can compile all of the positive operators efficiently (i.e., in polynomial time) into a functional VA [13]. In the case of NFAs or regular expressions, compiling the complement into an NFA necessitates an exponential blowup in size [7, 16].

Since NFAs and regular expressions are the Boolean functional VA and Boolean regex formulas, respectively, we conclude that constructing a VA is equivalent to the difference of two functional VAs, or two functional regex formulas, entails an exponential blowup. Therefore, the static compilation 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 decided in polynomial time. In contrast, the following theorem states that, for functional regex formulas (and VAs), this is no longer true under the conventional complexity assumption $P \neq NP$.

**Theorem 4.1.** The difference problem is $NP$-complete. Given two functional regex formulas $y_1$ and $y_2$ with $\text{Vars}(y_1) = \text{Vars}(y_2)$ and an input document $d$, is $\|y_1 \setminus y_2\|(d)$ nonempty?

**Proof.** Membership in NP is straightforward: for functional regex formulas, membership can be decided in polynomial time [11]. Hence, we focus on NP-hardness.

We use a reduction from 3SAT as in the proof of Theorem 3.1. Here, however, we are restricted to functional regex formulas and therefore we cannot use the domains of the resulting mappings to encode the assignments. Recall that the input is a formula $\phi$ with the free variables $x_1, \ldots, x_n$ such that $\phi$ 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 two literals, where a literal has the form $x_i$ or $\neg x_i$.

Given a 3CNF formula, we construct two functional regex formulas $y_1$ and $y_2$, and an input document $d$, such that there is a satisfying assignment for $\phi$ if and only if $\|y_1 \setminus y_2\|(d) \neq \emptyset$. We begin with the document $d$, which is defined by $d := a^n$.

The regex formulas $y_1$ and $y_2$ 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

$$\beta_i := ((x_i(\epsilon) \cdot a) \lor x_i(a))$$

for $1 \leq i \leq n$ and then define $y_1 := \beta_1 \cdots \beta_n$. Intuitively, $y_1$ encodes all of the legal assignments for $\phi$ in such a way that if $x_i$ captures the substring ‘$a’ then it corresponds with assigning $t$ to the free variable $x_i$, and otherwise (in case it captures $e$), it corresponds with assigning to it $f$. Before defining $y_2$, 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 $y'_2$ as follows:

$$y'_2 := \beta_1 \cdots \beta_{i_1} \cdot \delta_i \cdot \beta_{i_2} \cdots \beta_{i_3} \cdot \delta_i \cdot \beta_{i_1} \cdots \beta_{i_2} \cdot \delta_i \cdot \beta_{i_3} \cdots \beta_n$$

where $\delta_i$ is defined as $(x_i(\epsilon) \cdot a)$ if $x_i$ appears as a literal in $C_i$ or as $(x_i(a))$ if $\neg x_i$ appears as a literal in $C_i$.

Intuitively, $y'_2$ encodes the assignments for which clause $C_i$ is not satisfied. We then set

$$y_2 := \bigvee_{1 \leq i \leq m} y'_2.$$
for \( \varphi \). Hence, deciding nonemptiness of \(|y_1 \setminus y_2|\) is NP-hard.

From Theorem 4.1 we conclude that, in contrast to the tractability of the natural join of disjunctive functional VAs (Corollary 3.13), here we are facing NP-hardness already for functional VAs. In the remainder of this section, we discuss syntactic conditions that allow to avoid this hardness.

### 4.1 Bounded Number of Common Variables

Theorem 4.1 implies that no matter what approach we choose to tackle the evaluation of the difference, without imposing any 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 3.2).

Yet, in the case of difference, such static compilation necessitates an exponential blow-up, even if there are no variables at all (see the start of Section 4).

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

Ad-hoc VAs were introduced (without a name) by Freydenberger et al. [13] as a tool for evaluating functional VAs with polynomial delay. The next lemma is based on this idea.

**Lemma 4.2.** Let \( k \) be a fixed natural number. Given two sequential VAs \( A_1 \) 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 VA \( A_d \) with \(|A_d|(|d|) = |A_1 \setminus A_2|(|d|)\).

By Theorem 2.5, we can enumerate the results of a sequential VA in polynomial. We can conclude the following.

**Theorem 4.3.** Let \( k \) be a fixed natural number. Given two sequential VAs \( A_1 \) and \( A_2 \) where \(|\text{Vars}(A_1) \cap \text{Vars}(A_2)| \leq k \) and a document \( d \), one can enumerate \(|A_1 \setminus A_2|\) with polynomial delay.

**Proof Sketch of Lemma 4.2.** We construct two sequential VAs \( A \) and \( B \) (that share a bounded number of variables) such that evaluating the difference of \( A_1 \) and \( A_2 \) on \( d \) is the same as evaluating the natural join of \( A \) and \( B \) on \( d \). This natural join can be compiled into a sequential VA in polynomial time when the number of common variables is bounded by a constant (Theorem 3.3), and therefore, we establish the desired result.

Yet, unlike the schema-based model, difference in the schemaless case cannot be translated straightforwardly into a natural join (e.g., via complementation). For illustration, let us consider the case where there are \( \mu_1 \in |A_1|(|d|) \) and \( \mu_2 \in |A_2|(|d|) \) such that \( \text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset \). In this case, the assignment \( \mu_1 \) is not in \(|A_1 \setminus A_2|(|d|)\) since it is compatible with \( \mu_2 \). Nevertheless, \( \mu_1 \) will occur in the natural join of \( A_1 \) with every VA \( A_2' \), unless \( A_1 \) and \( A_2' \) share one or more common variables.

As a solution, we construct a VA that encodes information about the domains of the mappings \( \mu \), within the variables shared by \( A_1 \) and \( A_2 \), using new shared dummy variables. Specifically, we have a dummy variable \( \tilde{x} \) for every shared variable \( x \). If \( x \in \text{dom}(\mu) \), then \( \tilde{x} \) is assigned the first empty span \([1, 1]\), and if \( x \notin \text{dom}(\mu) \), then \( \tilde{x} \) is assigned the last empty span \([|d| + 1, |d| + 1]\). (Here, we assume that \( d \) is nonempty; we deal separately with the case \( d = \varepsilon \).)

We construct a VA \( A \) for the above extended mappings of \( A_1 \). In addition, we construct a VA \( B \) by iterating through all possible extended mappings over the shared variables, and for each such a mapping, if it is incompatible with all of the extended mappings of \( |A_2|(|d|) \), then we include it in \( B \). This construction can be done in polynomial time, since the number of common variables is bounded by a constant.

We conclude by showing that the extended mappings of \(|A_1|(|d|)\) that have compatible mappings in \(|B|(|d|)\) correspond to the mappings of \(|A_2|(|d|)\) that have no compatible mappings in \(|A_1|(|d|)\), and also that the extended mappings of \(|A|(|d|)\) that have compatible mappings in \(|B|(|d|)\) correspond to the mappings of \(|A_1|(|d|)\) that do not have compatible mappings in \(|A_2|(|d|)\).

**Theorem 4.4.** The following problem is \( \text{W}[1] \)-hard parameterized by \(|\text{Vars}(y_1) \cap \text{Vars}(y_2)| \). Given two functional regex formulas \( y_1 \) and \( y_2 \) and an input document \( d \), is \(|y_1 \setminus y_2|\) nonempty?

This result contrasts our FPT result for the natural join (Theorem 3.3). The proof uses a reduction from the problem of determining whether a 3-SAT formula has a satisfying assignment with at most \( p \) ones, where \( p \) is the parameter [6].

### 4.2 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 \( y \) be a sequential regex formula and let \( x \in \text{Vars} \) be a variable. Then \( y \) is synchronized for \( x \) if, for every subexpression of \( y \) of the form \( y_1 \lor y_2 \), we have that \( x \) appears neither
in $y_1$ nor in $y_2$. A regex formula $\gamma$ is called synchronized for $X \subseteq \text{Vars}$ if it is synchronized for every $x \in X$.

This notion generalizes to sequential VAs: A state $q$ of a sequential VA $A$ is called a unique target state for the variable operation $\omega \in \Gamma_{\text{Vars}(A)}$, 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 $-x$ 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 4.5.** Consider the regex formula $(x(\Sigma^* \vee \epsilon) \cdot (x(\Sigma^*))$ and this equivalent VA:

![Diagram of a variable automaton](image)

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

The following result states that conversions from regex formulas to VAs can preserve the property of being synchronized for $X$.

**Lemma 4.6.** 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 VA $A$ that is synchronized for $X$.

As one might expect, VAs that are synchronized (for some nonempty set $X$ of variables) are less expressive than sequential or semi-functional VAs (that are defined in Section 3.1). In fact, even functional regex formulas can express spanners that are not expressible with VAs that are synchronized for all their variables:

**Proposition 4.7.** There is no sequential VA that is synchronized for $x$ and equivalent to $(a \cdot x(\epsilon) \cdot a) \vee (b \cdot x(\epsilon) \cdot b)$.

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

**Theorem 4.8.** Given an input document $d$ and two sequential VAs $A_1$ 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 VA $A_4$ with $[A_4](d) = [A_1 \setminus A_2](d)$ in polynomial time.

The full proof can be found in the full version of the paper [24]; we discuss some of its key ideas. The first key observation is that $A_2$ can be treated as a functional VA that uses only the common variables (similarly to the proof of Lemma 3.8). 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) by Freydenberger et al. [13] to evaluate functional VAs with polynomial delay. 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 final 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 $[A_2](d)$.

While deterministic 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 $M(A, d)$. Using a variant of the proof of Lemma 3.8, we can then combine $A_1$ and $A_2$ into an ad-hoc VA $A_4$ with $[A_4](d) = [A_1 \setminus A_2](d)$.

After creating $A_4$ according to Theorem 4.8, we can use Theorem 2.5 to obtain the following tractability result:

**Corollary 4.9.** Given an input document $d$ and two sequential VAs $A_1$ 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 $[A_1 \setminus A_2](d)$ 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 $y_1 \vee y_2$.

**Proposition 4.10.** 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 $\gamma'_2$ 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 $\gamma'_2$ of $\gamma_2$,

is $[\gamma_1 \setminus \gamma_2](d)$ nonempty?

**Proof.** This proof is an adaption of the proof of Theorem 4.1, 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 \([31]\).

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

\[
\gamma_1 = (b \, x_1 \{ a \} \cdot a \cdot b) \cdot \ldots \cdot (b \, x_n \{ a \} \cdot a \cdot b).
\]

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 4.1 with an adaptation to the new input document and a slight simplification of the \( \gamma_i \)'s (since we do not need \( \gamma_2 \) to be functional any more). Formally, we set

\[
\gamma_2 = (b \, b \, x_1 \{ a \} \cdot a \cdot b) \cdot \ldots \cdot (b \, b \, x_n \{ a \} \cdot a \cdot b).
\]

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

\[
\gamma_2 = (b \, b \, x_1 \{ a \} \cdot a \cdot b) \cdot \ldots \cdot (b \, b \, x_n \{ a \} \cdot a \cdot b)
\]

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 \( \gamma_2 \). Correctness of this reduction can be shown analogously to that of Theorem 4.1.

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 conjunction 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 EXTRACTION COMPLEXITY

In this section, we discuss queries that are defined as RA expressions over schemaless spanners given in a representation language \( \mathcal{L} \) (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 its RA operator, and each of the leaves is a placeholder for a schemaless spanner. For illustration, Figure 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 \([14, 30]\).

As in the rest of the paper, 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 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 5.1.** 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:

- regex formula \( \alpha_{\text{sm}} \) with capture variables \( x_{\text{stdnt}}, x_{\text{ml}} \) that extracts names with their corresponding email addresses;
- regex formula \( \alpha_{\text{sp}} \) with variables \( x_{\text{stdnt}}, x_{\text{phn}} \) that extracts names with their corresponding phone numbers;
- regex formula \( \alpha_{\text{nr}} \) with variables \( x_{\text{stdnt}}, x_{\text{recommend}} \) 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.

\[
\pi_{\{x_{\text{stdnt}}\}}((\alpha_{\text{sm}} \bowtie \alpha_{\text{sp}}) \setminus (\alpha_{\text{nr}}))
\]

This query is \( I[\tau] \) for the RA tree \( \tau \) and the instantiation \( I \) of Figure 2. This query defines the spanner that \( I[\tau] \) corresponds to.

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 \( \|I[\tau]\|(d) \) is nonempty, given \( I \) and \( d \).

Clearly, some RA trees have an intractable nonemptiness and, consequently, an intractable evaluation. For example, if \( \mathcal{L} \) is the class of sequential regex formulas and \( \tau \) is the
Also, if \( L \) is the class of 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 3.1).

In contrast, by composing the positive results established in Sections 3 and 4, we obtain the following theorem, which is a consequence of Lemma 3.2 and Lemma 4.2.

**Theorem 5.2.** Let \( L \) be the class of sequential VAs. Let \( k \) be a fixed natural number and \( \tau \) an RA tree. The evaluation problem for \( \tau \) is solvable with polynomial delay, assuming that for all join and difference nodes \( v \) of \( I(\tau) \), 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 ad-hoc compilation to support the difference. Interestingly, the ad-hoc approach allows us to incorporate into the RA tree other representations of schemaless spanners, which can be treated as black-box schemaless spanners, 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, \(
\max(|\text{dom}(\mu)| \mid d \in \Sigma^*, \mu \in S(d)|)
\).

Formalizing the above, we can conclude from Theorem 5.2 a generalization that allows for black-box schemaless spanners. To this end, we call a representation language \( L' \) for schemaless spanners **tractable** if \( ||\beta||(d) \) can be evaluated in polynomial time (for some fixed polynomial), given \( \beta \in L' \) and \( d \in \Sigma^* \), and we call \( L' \) **degree bounded** if there is a fixed natural number that bounds the degree of all the schemaless spanners represented by expressions in \( L' \).

**Corollary 5.3.** Let \( L' \) be a tractable and degree-bounded representation system for schemaless spanners, and let \( L \) be the union of \( L' \) and the class of all sequential VAs. Let \( k \) be a fixed natural number and let \( \tau \) be an RA tree. The evaluation problem for \( \tau \) is solvable with polynomial delay, assuming 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 VAs, such as string equalities [8]. Other examples of such spanners are part of speech (POS) taggers, dependency parsers, sentiment analysis modules, and so on.

**Example 5.4.** Following Example 5.1, 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 PosRec, with the variables \( x_{\text{stdat}} \) and \( x_{\text{posrec}} \) that extract names and their corresponding positive recommendation. Note that this spanner has the degree 2. We can replace \( \alpha_0 \) in the instantiation \( I \) of Figure 2 with PosRec, and thereby obtain the desired result. If PosRec can be computed in polynomial time, then the resulting query can be evaluated in polynomial delay.

**6 CONCLUSIONS**

We have studied the complexity of evaluating algebraic expressions over schemaless spanners that are represented as sequential regex formulas and sequential VAs. 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 VAs (and regex formulas) into a single sequential VA, 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 VA—an exponential blowup in the number of states is necessary already for Boolean spanners. Evaluation with polynomial delay is then obtained via an ad-hoc compilation of both the spanners and the document into a VA. We have shown how the ad-hoc approach can be used for establishing upper bounds on general RA trees over regex formulas, VAs, and even black-box spanners of a bounded dimension. This has been done within the concept of extraction complexity that we have proposed as new lens to analyzing the complexity of spanners.

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, but it is quite conceivable that less restrictive conditions already suffice. Alternatively, are there conditions of extractors (possibly incomparable to ours) that are both common in practice and useful to bound the extraction complexity?

**ACKNOWLEDGMENTS**

The work of Liat Peterfreund was supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and Israel Cyber Bureau. The work of Benny Kimelfeld and Liat Peterfreund was supported by the ISF Grant 1295/15. The work of Markus Kröll was supported by the FWF Grants P30930-N35 and W1255-N23.
REFERENCES


