Document Spanners - A Brief Overview of Concepts, Results, and Recent Developments

Markus L. Schmid, Nicole Schweikardt

Document Spanners — A Brief Overview of Concepts, Results, and Recent Developments

Markus L. Schmid
MLSchmid@MLSchmid.de
Humboldt-Universität zu Berlin
Germany

Nicole Schweikardt
schweikn@informatik.hu-berlin.de
Humboldt-Universität zu Berlin
Germany

ABSTRACT

The information extraction framework of *document spanners* was introduced by Fagin, Kimelfeld, Reiss, and Vansummeren (PODS 2013, J. ACM 2015) as a formalisation of the query language AQL, which is used in IBM's information extraction engine SystemT. Since 2013, this framework has been investigated in depth by the principles of database management community and beyond. The present paper gives a brief overview of concepts, results, and recent developments concerning document spanners.

CCS CONCEPTS

• **Information systems** → *Information retrieval*; • **Theory of computation** → *Database query languages (principles)*; *Data structures and algorithms for data management*; *Design and analysis of algorithms*; *Automata extensions*; *Regular languages.*

KEYWORDS

Document Spanners, Information Extraction, Overview, Core Spanners, Regular Spanners, Refl-Spanners, SLP-represented documents

ACM Reference Format:
Markus L. Schmid and Nicole Schweikardt. 2022. Document Spanners — A Brief Overview of Concepts, Results, and Recent Developments. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3517804.3526069

1 INTRODUCTION

The information extraction framework of *document spanners* has been introduced by Fagin, Kimelfeld, Reiss, and Vansummeren [9] as a formalisation of the query language AQL, which is used in IBM's information extraction engine SystemT. A document spanner performs information extraction by mapping a *document* $\mathbf{D}$, formalised as a *word* (or *string*, *sequence*) over a finite alphabet Σ, to a relation over so-called *spans* of $\mathbf{D}$, which are intervals $[i, j\rangle$ with $1 \leqslant i \leqslant j \leqslant |\mathbf{D}| + 1$, where $|\mathbf{D}|$ denotes the length of $\mathbf{D}$. Intuitively, a span $[i, j\rangle$ of a document $\mathbf{D} = \mathsf{a}_1\mathsf{a}_2 \cdots \mathsf{a}_n$ represents the factor $\mathsf{a}_i\mathsf{a}_{i+1} \cdots \mathsf{a}_{j-1}$. More formally, $\mathrm{Spans}(\mathbf{D}) := \{[i, j\rangle : 1 \leqslant i \leqslant j \leqslant |\mathbf{D}| + 1\}$ is the set of spans of document $\mathbf{D}$. For a fixed finite set $\mathcal{X}$ of variables, an $(\mathcal{X}, \mathbf{D})$-*tuple* (or simply, *span tuple*) is a function $\mathcal{X} \to \mathrm{Spans}(\mathbf{D})$. An $(\mathcal{X}, \mathbf{D})$-*relation* is a set of $(\mathcal{X}, \mathbf{D})$-tuples. Now, a *document spanner (over Σ and $\mathcal{X}$)* (or simply, *spanner*) is a function that maps every document $\mathbf{D} \in \Sigma^*$ to an $(\mathcal{X}, \mathbf{D})$-relation.

To simplify the notation, one usually assumes the variables in $\mathcal{X} = \{x_1, \ldots, x_k\}$ to be ordered, so that a span-tuple $t : \mathcal{X} \to \mathrm{Spans}(\mathbf{D})$ is identified with the k-tuple $(t(x_1), \ldots, t(x_k))$.

Example 1.1. *Let $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ and let $\mathcal{X} = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$ with $\mathsf{x} < \mathsf{y} < \mathsf{z}$. Then the function S that maps documents $\mathbf{D} \in \Sigma^*$ to the $(\mathcal{X}, \mathbf{D})$-relation of all span tuples $([1, i\rangle, [i, i{+}1\rangle, [i{+}1, |\mathbf{D}|{+}1\rangle)$ such that the i^{th} position of $\mathbf{D}$ is an occurrence of b, is a spanner. For example, S maps the document* ababbab *to the following span-relation (represented as a table):*

x	y	z
$[1, 2\rangle$	$[2, 3\rangle$	$[3, 8\rangle$
$[1, 4\rangle$	$[4, 5\rangle$	$[5, 8\rangle$
$[1, 5\rangle$	$[5, 6\rangle$	$[6, 8\rangle$
$[1, 7\rangle$	$[7, 8\rangle$	$[8, 8\rangle$

The initial work on document spanners was dominated by their original motivation, i. e., a formalisation of the query language AQL of IBM's information extraction engine SystemT. In this regard, the class of document spanners presented in [9] follow a two-stage approach: *Primitive* spanners extract span relations directly from the input document (see Example 1.1), which are then further manipulated by using particular *algebraic operations*.

The primitive spanners of [9] are based on particular finite automata and regular expressions called *variable-set automata* (*vset-automata*, for short) and *regex-formulas*, respectively. In this paper, we use a slightly different formalism to represent primitive spanners illustrated in the following example and further explained in Section 2.1. It is straightforward to see that the spanner of Example 1.1 can be formalised by the regular expression

$$\alpha := \ {}^{\mathsf{x}}\triangleright\,(\mathsf{a} \vee \mathsf{b})^*\,\triangleleft^{\mathsf{x}} \cdot {}^{\mathsf{y}}\triangleright\,\mathsf{b}\,\triangleleft^{\mathsf{y}} \cdot {}^{\mathsf{z}}\triangleright\,(\mathsf{a} \vee \mathsf{b})^*\,\triangleleft^{\mathsf{z}}.$$

Here, the symbols ${}^{\mathsf{x}}\triangleright$, $\triangleleft^{\mathsf{x}}$ are meta-symbols called *markers*, that define the span extracted by variable x. As usual, we write $L(\alpha)$ for the language described by α, i.e., the set of all words (over $\Sigma \cup \{{}^{\mathsf{x}}\triangleright, \triangleleft^{\mathsf{x}} : \mathsf{x} \in \mathcal{X}\}$) that match the regular expression α. Referring to Example 1.1, the fact that ${}^{\mathsf{x}}\triangleright\,\mathsf{a}\,\triangleleft^{\mathsf{x}}\,{}^{\mathsf{y}}\triangleright\,\mathsf{b}\,\triangleleft^{\mathsf{y}}\,{}^{\mathsf{z}}\triangleright\,\mathsf{abbab}\,\triangleleft^{\mathsf{z}} \in L(\alpha)$ means that $([1, 2\rangle, [2, 3\rangle, [3, 8\rangle)$ is in the span relation $S(\mathsf{ababbab})$. In this sense, for a given document $\mathbf{D}$, the span tuples of $S(\mathbf{D})$ are described by all ways of how α can generate a string that equals $\mathbf{D}$ with the marker symbols shuffled in between the actual symbols. Likewise, one can define a finite automaton that, in addition to the symbols from Σ, may also read markers on its arcs. Nowadays, it is common to denote spanners that can be described in such a way as *regular*

PODS '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9260-0/22/06…$15.00
https://doi.org/10.1145/3517804.3526069

spanners (we shall discuss in Section 2.2 some particularities about different representations of regular spanner that, for simplicity, are neglected in the introduction).

The algebraic operations considered in [9] to further manipulate the primitive span relations extracted by regular spanner are the classical relational algebra operations of union $\cup$, natural join $\bowtie$, and projection π (with their obvious meanings), and additionaly, as a special operation that is particularly useful for information extraction from textual data, also the *string-equality selection* $\zeta^{=}$. String-equality selection is a unary operator parameterised by a set $\mathcal{Z} \subseteq \mathcal{X}$ of variables, and the operation $\zeta^{=}_{\mathcal{Z}}$ selects exactly those rows of the table, for which all spans of columns in $\mathcal{Z}$ refer to (potentially different occurrences of) the same subwords of $\mathbf{D}$. To clarify this operation, consider the regular spanner S_α described by

$$\alpha \;:=\; {}^{x}\triangleright\,(a \vee b)^*\,\triangleleft^{x} \cdot (a \vee b)^* \cdot {}^{y}\triangleright\,(a^*b^*)\triangleleft^{y}$$

and the string-equality selection $\zeta^{=}_{\{x,y\}}$. In $S_\alpha(\mathtt{abaaab})$ we have, among others, the span tuples $([1,3\rangle, [5,7\rangle)$ and $([1,3\rangle, [4,7\rangle)$. However, the string-equality selection $\zeta^{=}_{\{x,y\}}$ discards the latter and only selects the former.

In [9], the class of *core spanners* (capturing the *core* of SystemT's query language AQL) is defined as the closure of the class of primitive spanners that can be defined by a special class of regular expressions called *regex-formulas* (and denoted by RGX) under the relational algebra operations $\cup$, $\bowtie$, π and the string-equality selection $\zeta^{=}$ (comprising of the operator $\zeta^{=}_{\mathcal{Z}}$ for all $\mathcal{Z} \subseteq \mathcal{X}$). By using classical closure properties of finite automata, it is shown in [9] that the class of core spanners can also be characterised by regular spanners represented by a particular kind of finite automata called *vset-automata* and applications of string-equality selections, followed by a projection operation — this is known as the *core-simplification lemma*, which we further discuss in Section 2.3.

Since their introduction in [9], spanners have received a lot of attention (cf., e.g., [2, 8–13, 15, 16, 27, 30–33, 38–40]). A substantial part of the literature on document spanners has focused on *regular* spanners. In particular, it has been shown that results of regular spanners can be enumerated with linear preprocessing and constant delay [2, 10], the paper [27] is concerned with different semantics of regular spanners and their expressive power, and the articles [39, 40] study the evaluation of regular spanners on compressed representations of documents. The paper [32] investigates the evaluation of algebraic expressions over regular spanners. Articles that are concerned with string-equality selection are [12] in which many hardness results for core spanner evaluation are shown, [11] which, by presenting a logic that exactly covers core spanners, answers questions on the expressive power of core spanners, [33] which shows that datalog over regular spanners covers the whole class of core spanners, [13] which investigates conjunctive queries on top of regular spanners, and [38], which incorporates string-equality selections directly into the regular language that represents the underlying regular spanner. The paper [15] investigates the dynamic descriptive complexity of regular spanners and core spanners. The article [31] studies non-regular document spanners of a different kind, where non-regularity is not caused by string-equality selections, but rather by representing spanners by context-free language descriptors (in particular, grammars) instead of regular ones.

The remainder of this paper is structured as follows. Section 2 presents more details on aspects briefly discussed in the introduction, including a more detailed discussion of representations of spanners, different semantics, and the core-simplification lemma (Sections 2.1—2.3), and static analysis and query evaluation tasks associated with spanners (Sections 2.4 and 2.5). Section 3 gives an overview of the framework of *refl-spanners* of [38]. Section 4 focuses on the scenario of [39, 40] of spanner evaluation on compressed representations of documents.

2 DOCUMENT SPANNERS

This section presents more details on aspects briefly discussed in the introduction.

2.1 A General Declarative Approach to Formalise Document Spanners

The approach of [9] to formalise regular document spanners by vset-automata is of a somewhat imperative nature: An algorithm — in form of a finite automaton — is defined that reads an input document and nondeterministically produces a span tuple. The extracted span relation is then simply defined as all span tuples that can be extracted from the document by a run of this algorithm. A more declarative approach that is not restricted to regular document spanners but allows to formalise all document spanners was presented in [38] and is based on the simple observation that any pair of document $\mathbf{D} \in \Sigma^*$ and $(\mathcal{X}, \mathbf{D})$-tuple t can be represented as a string over the alphabet $\Sigma \cup \{{}^{x}\triangleright, \triangleleft^{x} : x \in \mathcal{X}\}$: We explicitly represent each span $t(x)$ in $\mathbf{D}$ with the markers ${}^{x}\triangleright$ and $\triangleleft^{x}$. As an example, for $\mathcal{X} = \{x, y, z\}$, consider the document $\mathbf{D} = \mathtt{abcacacbbaa}$ and the $(\mathcal{X}, \mathbf{D})$-tuple t with $t(x) = [2, 6\rangle$, $t(y) = [4, 8\rangle$, $t(z) = [1, 8\rangle$, which can be represented by the string

$${}^{z}\triangleright\,\mathtt{a}\,{}^{x}\triangleright\,\mathtt{bc}\,{}^{y}\triangleright\,\mathtt{ac}\,\triangleleft^{x}\,\mathtt{ac}\,\triangleleft^{y}\;\triangleleft^{z}\,\mathtt{bbaa}\,. \tag{1}$$

In the literature on spanners, such string representations of documents and span tuples are called *subword-marked words* or *ref-words* (over Σ and $\mathcal{X}$). Both these terms are derived from the work [37], where similar concepts are used to formalise capture groups and backreferences in regular expressions. A sound formalisation of subword-marked words over Σ and $\mathcal{X}$ can be obtained by simply restricting words over $\Sigma \cup \{{}^{x}\triangleright, \triangleleft^{x} : x \in \mathcal{X}\}$ in such a way that, for every $x \in \mathcal{X}$, there is exactly one occurrence of each ${}^{x}\triangleright$ and $\triangleleft^{x}$, and they occur in this order. Equivalently, we can just say that subword-marked words over Σ and $\mathcal{X}$ are exactly those strings that can be obtained by inserting the markers $\{{}^{x}\triangleright, \triangleleft^{x} : x \in \mathcal{X}\}$ into some document $\mathbf{D}$ as described by some $(\mathcal{X}, \mathbf{D})$-tuple t.

Any given subword-marked word w represents a document $\mathfrak{e}(w)$ (so $\mathfrak{e}(\cdot)$ just erases all markers) and a span tuple $\mathrm{st}(w)$ (the function $\mathrm{st}(\cdot)$ just transforms the information where the markers occur into spans in the obvious way). This means that for any spanner S and document $\mathbf{D}$, the span relation $S(\mathbf{D})$ with m rows can be represented as a set $L_{\mathbf{D}}$ of m subword-marked words (we just insert the markers into $\mathbf{D}$ as described by the span tuples). Referring to Example 1.1, $S(\mathtt{ababbab})$ can therefore be represented as

$$L_{\text{ababbab}} = \{\ {}^{x}\triangleright\, a\, \triangleleft^{x}\ {}^{y}\triangleright\, b\, \triangleleft^{y}\ {}^{z}\triangleright\, abbab\triangleleft^{z},\ {}^{x}\triangleright\, aba\, \triangleleft^{x}\ {}^{y}\triangleright\, b\, \triangleleft^{y}\ {}^{z}\triangleright\, bab\triangleleft^{z},\ {}^{x}\triangleright\, abab\, \triangleleft^{x}\ {}^{y}\triangleright\, b\, \triangleleft^{y}\ {}^{z}\triangleright\, ab\triangleleft^{z},\ {}^{x}\triangleright\, ababba\, \triangleleft^{x}\ {}^{y}\triangleright\, b\, \triangleleft^{y}\ {}^{z}\triangleright\ \triangleleft^{z}\ \}.$$

Consequently, any spanner S can be represented as a set of subword-marked words over Σ and $\mathcal{X}$, namely $\bigcup_{\mathbf{D}\in\Sigma^*} L_{\mathbf{D}}$. Conversely, *every* set L of subword-marked words over Σ and $\mathcal{X}$ necessarily describes a spanner $[\![L]\!]$ with $[\![L]\!](\mathbf{D}) = \{\mathrm{st}(w) : w \in L, \mathfrak{e}(w) = \mathbf{D}\}$. In the following, for any set L of subword-marked words or descriptor M of a set of subword-marked words, we use $[\![\cdot]\!]$ to denote the represented spanner.

We can now define classes of spanners independently of specialised machine models by defining classes of sets of subword-marked words. For example, the basic class of *regular* spanners is characterised by the class of those sets of subword-marked words that are regular languages (and any model for defining regular languages can then be employed as a model for describing regular spanners). Clearly, one now can replace "regular" by any established language class with their own toolkit of algorithmic models (the case where "regular" is replaced with "context-free" is studied in [31]). With respect to this approach it is also worth noting that for any language over the alphabet $\Sigma \cup \{{}^{x}\triangleright, \triangleleft^{x} : x \in \mathcal{X}\}$ that is not necessarily a subword-marked language, we can obtain its subset of subword-marked words by an intersection with a regular language. Hence, any class of languages closed under intersection with regular languages can directly be interpreted as a class of spanners (in this regard, note that closure under intersection with regular languages is one of the most natural language operations, and many relevant classes of formal languages are closed under this operation, e. g., (deterministic) context-free languages, and all language classes closed under finite state transductions, see [20, 26] for further details).

2.2 Particularities of Document Spanners

An important aspect of the semantics of spanners is whether span tuples are total functions $\mathcal{X} \to \mathrm{Spans}$ (as initially defined in [9]) or whether they can be partial mappings, i. e., for a $(\mathcal{X}, \mathbf{D})$-tuple t and $x \in \mathcal{X}$ it is possible that $t(x)$ is undefined (denoted as $t(x) = \bot$). This latter case is called the *schemaless semantics* and has been introduced and studied in [27]. In terms of the general approach described in Section 2.1, the difference between the classical semantics where span tuples are total functions and the schemaless semantics can directly be reflected in the definition of subword-marked words, i. e., we either require all markers to appear, or we allow that some markers may be missing.

A subword-marked word w over Σ and $\mathcal{X}$ and its associated span tuple $t = \mathrm{st}(w)$ are called *functional* if t is a total function on $\mathcal{X}$, a span relation is called functional if all its elements are functional, and a spanner S is called functional if $S(\mathbf{D})$ is functional for every document $\mathbf{D} \in \Sigma^*$.

The regex-formulas used in [9] are obtained from regular expressions over Σ in which proper sub-expressions can be enclosed by ${}^{x}\triangleright \cdots \triangleleft^{x}$. Such expressions, if interpreted as expressions over $\Sigma \cup \{{}^{x}\triangleright, \triangleleft^{x} : x \in \mathcal{X}\}$, can define subword-marked languages and therefore document spanners. Note that such spanners are necessarily *hierarchical* in the sense that the pairs of brackets ${}^{x}\triangleright \cdots \triangleleft^{x}$ for different $x \in \mathcal{X}$ are either strictly nested or disjoint; i. e., overlappings like in (1) cannot be described. In particular, this imlies that the class of spanners solely described by regex-formulas is strictly contained in those described by vset-automata. However — as a main observation of [9] — if one considers the $\{\cup, \bowtie, \pi\}$-closures of the class of spanners described by regex-formulas and the class of spanners described by vset-automata, one obtains the same class of spanners, namely, the class of spanners that can directly be described by vset-automata.

Subword-marked words (or runs of vset-automata) do not necessarily represent a document $\mathbf{D}$ and an $(\mathcal{X}, \mathbf{D})$-tuple t in a *unique* way. More precisely, the represented span tuple is invariant with respect to the order of consecutive occurrences of markers in the subword-marked word (or the order in which a vset-automata performs consecutive marker-transitions). We discuss in Section 2.4 how this issue can matter for decision problems for spanners. There are two obvious ways of how to deal with this non-uniqueness, which have both been considered in the literature. Option 1 is to *normalise* by restricting the model of subword-marked languages (and therefore automata accepting such languages), by fixing an order on $\{{}^{x}\triangleright, \triangleleft^{x} : x \in \mathcal{X}\}$ and requiring factors of *consecutive* markers of subword-marked words to respect this order (this has been done e.g. in [7, 9, 38]). Option 2 is to represent factors of consecutive markers as *sets* of markers; the according variant of vset-automata is called *extended vset-automata* [10]. In terms of the general approach described in Section 2.1, this means that subword-marked words now are built from letters in Σ and elements of the power set of $\{{}^{x}\triangleright, \triangleleft^{x} : x \in \mathcal{X}\}$. For example, the subword-marked word from (1) is represented by $\{{}^{z}\triangleright\}a\{{}^{x}\triangleright\}bc\{{}^{y}\triangleright\}ac\{\triangleleft^{x}\}ac\{\triangleleft^{y}, \triangleleft^{z}\}bbaa$.

2.3 Core-Simplification Lemma

As already mentioned in Section 1, the *core spanners* are defined as those spanners that can be described by applying the operations union, natural join, projection, and string-equality selection to spanners described by regex-formulas; in symbols, this class is denoted as $[\![\mathrm{RGX}]\!]^{\{\cup, \bowtie, \pi, \zeta^{=}\}}$. The *core-simplification lemma* states that any core spanner can be represented as

$$\pi_{\mathcal{Y}}(\zeta^{=}_{\mathcal{Z}_1}\zeta^{=}_{\mathcal{Z}_2}\cdots\zeta^{=}_{\mathcal{Z}_k}([\![M]\!])),$$

where $\mathcal{Y}, \mathcal{Z}_1, \ldots, \mathcal{Z}_k \subseteq \mathcal{X}$, and M describes a regular language of subset-marked words (e.g., M is a vset-automaton). For functional spanners, this has been proved in [9], and it also holds verbatim for the schemaless case (proved in [38] by combining the approach of [9] with results from [27]).

The core-simplification lemma tells us that, in terms of expressive power, the string-equality selection (followed by a projection) is the only feature of core spanners that properly exceeds the expressive power of regular languages. As recent research has proven, this seemingly "little" increase of expressivity has severe implications for the complexity (and even (un)decidability) of static analysis and evaluation problems; we discuss this in Section 2.4.

2.4 Decision Problems for Spanners

The following typical decision problems for spanners have been studied in the literature; the first two of these are *evaluation problems* whereas the other are *static analysis problems*.

- ModelChecking: Given a spanner S over Σ and $\mathcal{X}$, a document $\mathbf{D} \in \Sigma^*$, and an $(\mathcal{X}, \mathbf{D})$-tuple t, decide whether $t \in S(\mathbf{D})$.
- NonEmptiness: Given a spanner S over Σ and $\mathcal{X}$ and a document $\mathbf{D} \in \Sigma^*$, decide whether $S(\mathbf{D}) \neq \emptyset$.
- Satisfiability: Given a spanner S over Σ and $\mathcal{X}$, decide whether there exists a document $\mathbf{D} \in \Sigma^*$ with $S(\mathbf{D}) \neq \emptyset$.
- Hierarchicality: For a given spanner S over Σ and $\mathcal{X}$, decide whether S is hierarchical.
- Containment: Given two spanners S_1, S_2 over Σ and $\mathcal{X}$, decide whether $S_1(\mathbf{D}) \subseteq S_2(\mathbf{D})$ holds for all documents $\mathbf{D} \in \Sigma^*$.
- Equivalence: Given two spanners S_1, S_2 over Σ and $\mathcal{X}$, decide whether $S_1(\mathbf{D}) = S_2(\mathbf{D})$ holds for all documents $\mathbf{D} \in \Sigma^*$.

Obviously, the complexity and decidability of these problems depend on the class of spanners at hand.

For *regular* spanners, due to the positive algorithmic properties of regular languages, all these problems are decidable and have acceptable upper complexity bounds. Let us illustrate this for the problems ModelChecking and Equivalence and the setting where regular spanners are represented by non-deterministic finite automata (NFA) that accept subword-marked languages.

We can decide ModelChecking as follows. The input consists of an NFA M, a document $\mathbf{D}$ and a span tuple t. We can check whether $t \in [\![M]\!](\mathbf{D})$ by transforming $\mathbf{D}$ and t into a subword-marked word w and check whether M accepts w. The obvious problem with this approach is what we already have discussed in the last paragraph of Section 2.2: we do not know a priori in which order we have to insert consecutive marker symbols into $\mathbf{D}$ in order to construct w. There are several ways of handling this issue, e. g., we could require M to be normalised or to be an extended vset-automaton, or we could modify M such that it satisfies one of these requirements (potentially resulting in an exponential size blow-up), or we could try to utilise that we know that M will only accept proper subword-marked words to handle the question of the right order of the markers on-the-fly while processing w with M.

The situation is similar for the problem Equivalence: The input now consists of two NFAs M_1 and M_2 that accept subword-marked languages. One can reduce the question whether M_1 and M_2 describe the same spanner to the the question whether suitably modified NFAs M_1' and M_2' accept the same regular language.

The picture is quite different (i. e., worse) for core spanners. Let us first give an intuition before discussing known results. The power to define *repetitions* of strings is a feature that exceeds the expressive power of regular and even context-free languages (the *copy language* $\{ww : w \in \Sigma^*\}$ is the classical textbook example), and, in terms of the Chomsky-hierarchy, is covered by context-sensitive languages, which are not famous for having good algorithmic properties. Leaving classical questions of formal language theory aside, it is also well-known that the complexity of string processing and pattern matching problems substantially increases if extended by string repetition operators (but note that regular expressions as used in practical contexts are usually extended by such string repetition operators). Finally, string repetitions are also known to pose very hard combinatorial problems as usually investigated in combinatorics on words (e. g., word equations). We shall illustrate with three particular examples how these observations are crucial for core spanners.

First, consider the regular expression

$$\alpha := \ {}^{\mathsf{x}_1\triangleright}\Sigma^*\,{}^{\triangleleft\mathsf{x}_1}\ {}^{\mathsf{x}_2\triangleright}\Sigma^*\,{}^{\triangleleft\mathsf{x}_2}\cdots\,{}^{\mathsf{x}_n\triangleright}\Sigma^*\,{}^{\triangleleft\mathsf{x}_n}$$

that obviously describes a subword-marked language over Σ and $\mathcal{X} = \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n\}$ (in fact, α is a regex-formula). Now checking, for some $\mathcal{Z}_1, \mathcal{Z}_2, \ldots, \mathcal{Z}_k \subseteq \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n\}$ and a document $\mathbf{D}$, whether the empty tuple is in $\big(\pi_\emptyset(\varsigma^{=}_{\mathcal{Z}_1}\varsigma^{=}_{\mathcal{Z}_2}\cdots\varsigma^{=}_{\mathcal{Z}_k}([\![\alpha]\!]))\big)(\mathbf{D})$, is identical to checking whether $\mathbf{D}$ can be factorised into n factors such that for each $\mathcal{Z}_i$ all factors that correspond to the variables in $\mathcal{Z}_i$ are the same. This is the *pattern matching problem with variables* (also known as the *membership problem for pattern languages*, or the matching problem for *regular expressions with backreferences*), a well-known NP-complete problem (see, e. g., [24]).

As another example, let $r_1, r_2, \ldots, r_n$ be some regular expressions over Σ, and let $\alpha = {}^{\mathsf{x}_1\triangleright}r_1\,{}^{\triangleleft\mathsf{x}_1}\ {}^{\mathsf{x}_2\triangleright}r_2\,{}^{\triangleleft\mathsf{x}_2}\cdots\,{}^{\mathsf{x}_n\triangleright}r_n\,{}^{\triangleleft\mathsf{x}_n}$ (again, note that this is a regex-formula). Then there is a word w with $(\varsigma^{=}_{\{\mathsf{x}_1,\mathsf{x}_2,\ldots,\mathsf{x}_n\}}([\![\alpha]\!]))(w) \neq \emptyset$ if and only if $\bigcap_{i=1}^{n} L(r_i) \neq \emptyset$. Consequently, core spanners (with only one string-equality selection) can express *intersection non-emptiness of regular languages*, a well-known PSpace-complete problem.

Finally, let us discuss an example of how core spanners can describe relations typically described by *word equations* (see [12, Proposition 3.7, Example 3.8, Theorem 3.13] for details). For given strings $u, v \in \Sigma^*$, we write $u \sim_{\mathsf{com}} v$ if there exists a $p \in \Sigma^*$ such that $u, v \in \{p\}^*$, and we write $u \sim_{\mathsf{cyc}} v$ if u is a cyclic shift of v, i.e., there exist strings w_1, w_2 such that $u = w_1w_2$ and $v = w_2w_1$. It is a well-known fact that $\sim_{\mathsf{com}}$ can be expressed by the word equation $\mathsf{xy} = \mathsf{yx}$ (i. e., $u \sim_{\mathsf{com}} v$ if and only if the substitution $\mathsf{x} \mapsto u$ and $\mathsf{y} \mapsto v$ satisfies the equation), and $\sim_{\mathsf{cyc}}$ can be expressed by the word equation $\mathsf{xz} = \mathsf{zy}$ (i. e., $u \sim_{\mathsf{cyc}} v$ if and only if there is some string w such that the substitution $\mathsf{x} \mapsto u$, $\mathsf{y} \mapsto v$ and $\mathsf{z} \mapsto w$ satisfies the equation). It is shown in [12, Proposition 3.7] that one can define a core spanner S_{com} over Σ and $\{\mathsf{x}, \mathsf{y}\}$ such that $S_{\mathsf{com}}(\mathbf{D})$ represents exactly those pairs (u, v) of factors of $\mathbf{D}$ with $u \sim_{\mathsf{com}} v$; analogously, a core spanner $S_{\mathsf{cyc}}(\mathbf{D})$ can be constructed. In general, it is shown in [12] that, in a specific sense (see [12, Theorems 3.12 and 3.13] for details), core spanners are as expressive as *word equations with regular constraints*. This provides valuable insight in the complexity of core spanners, because word equations pose notoriously hard questions [6]. For example, whether solving word equations (without regular constraints) is in NP is still unknown — even for *quadratic equations* [5, 35] where every variable has at most two occurrences (which is the case for the two particular word equations mentioned above).

These explanations show how powerful the string-equality selection is, and, consequently, that extending regular spanners by this operation (i. e., extending regular spanners to core spanners) has severe consequences in terms of decidability and expressive power. In particular, from [12] we know that ModelChecking and NonEmptiness are NP-hard, Satisfiability and Hierarchicality are

PSpace-complete, and Containment and Equivalence are even undecidable.

2.5 The Enumeration Problem for Regular Spanners

A special focus of the recent research on spanners has been dedicated to the *enumeration problem*: Given a spanner S over Σ and $\mathcal{X}$, and a document $\mathbf{D}$, we want to enumerate, without repetition, all tuples in $S(\mathbf{D})$. In this setting, one is interested in algorithms that proceed in two phases: a *preprocessing phase* in which suitable data structures are built, and an *enumeration phase* in which these data structures are used to output the result tuples. One aims at run-time guarantees on the time spent for the preprocessing phase and the delay during the enumeration phase (i. e., the time needed between producing two elements of the result). Usually, these run-time guarantees are stated in terms of *data complexity*, i.e., the input size is measured in the length of the document $\mathbf{D}$ while the dependency on the (size of the representation of the) spanner S is hidden in the O-notation. The best one can hope for is *linear preprocessing* and *constant delay*. And for regular spanners, the enumeration problem can indeed be solved with linear preprocessing and constant delay: In [10] it is shown how to achieve this for regular spanners that are represented by extended deterministic vset-automata. An immediate consequence of this result is that we can also achieve linear preprocessing and constant delay enumeration for other representations of regular spanners (including nondeterministic automata, regular expressions, etc.) since the time needed for translating them into extended deterministic vset-automata vanishes in the data complexity perspective. An important improvement of this was achieved in [2] (see also [1]), who devised an enumeration algorithm directly for regular spanners represented by vset-automata; this algorithm has linear preprocessing and constant delay and, moreover, its preprocessing time and delay are also polynomial in the size of the vset-automaton's representation.

3 REFL-SPANNERS

As discussed in Section 2.4, static analysis and evaluation for core spanners is much more difficult (and in some cases even undecidable) compared to regular spanners. We have already seen, on an intuitive level, how the functionality of string-equality selections is responsible for this increase in complexity. In order to gain a better understanding of this phenomenon, let us take a more thorough look at the interplay of the core spanners' features.

As discussed in Section 2.4, checking for

$$\alpha \;:=\; {}^{\mathsf{x}_1\triangleright}\,\Sigma^*\,{}^{\triangleleft\mathsf{x}_1}\;{}^{\mathsf{x}_2\triangleright}\,\Sigma^*\,{}^{\triangleleft\mathsf{x}_2}\cdots{}^{\mathsf{x}_n\triangleright}\,\Sigma^*\,{}^{\triangleleft\mathsf{x}_n},$$

a document $\mathbf{D} \in \Sigma^*$, and sets $\mathcal{Z}_1, \ldots, \mathcal{Z}_k \subseteq \{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$ whether the empty tuple is in

$$\big(\pi_\emptyset(\varsigma^{=}_{\mathcal{Z}_1} \cdots \varsigma^{=}_{\mathcal{Z}_k}([\![\alpha]\!]))\big)\,(\mathbf{D})$$

encodes an intractable pattern matching problem. More precisely, in order to check this, we have to check whether there is at least one subword-marked word $w \in L(\alpha)$ with $\mathfrak{e}(w) = \mathbf{D}$, such that the markers ${}^{\mathsf{x}\triangleright} \cdots {}^{\triangleleft\mathsf{x}}$ enclose factors that satisfy the string-equality selections. Our intuition that in the worst-case we have to exhaustively check all such subword-marked words is confirmed by the known NP-completeness of this task.

At this point, however, one might ask whether a spanner of the form $\big(\pi_\emptyset(\varsigma^{=}_{\mathcal{Z}_1} \cdots \varsigma^{=}_{\mathcal{Z}_k}([\![\alpha]\!]))\big)$ really describes a reasonable information extraction task, or whether it can be considered as a rather special case. After all, this spanner considers all ways of extracting certain span tuples from a document (described by the regular spanner α), checks whether at least one of them satisfies a complex property described by the string-equality selections, and finally "returns" (by means of the projection $\pi_\emptyset$) either yes or no. This spanner implements a (rather complex) task of string analysis (or pattern matching or parsing): it answers the question "Does this string satisfy a certain property?" (or: "Does it have the right form?"), while it extracts either the empty set or the singleton set consisting of the empty tuple. This does not seem to fit well to the original intention of extracting information from a document, i. e., the assumption that documents contain information in a sequential way that we wish to extract and compile into a relational form. In fact, it is not far-fetched to assume that if we define a spanner for some information extraction task and this spanner contains a variable x, then we also want — at least as an intermediate result — the actual content extracted by variable x to appear in our constructed table (whether the x-column is projected away by some further processing of the data is another story).

It is easy to see that dropping $\pi_\emptyset$ from a core spanner can make a huge difference. For example, checking for a given span tuple t whether

$$t \;\in\; \big(\varsigma^{=}_{\mathcal{Z}_1}\varsigma^{=}_{\mathcal{Z}_2}\cdots\varsigma^{=}_{\mathcal{Z}_k}([\![\alpha]\!])\big)\,(\mathbf{D})$$

can be done quite efficiently (regardless of the actual string-equality selections), since t already tells us the contents of the extracted spans.

Since regular spanners may be non-hierarchical (i. e., variables may address overlapping spans, cf. Section 2.2), core spanners can use string-equality selections over overlapping spans. It is already somewhat difficult to come up with a practical example of a regular spanner that extracts overlapping spans. A possible scenario is that we do not have enough a-priori knowledge about the structure of the documents in order to define a spanner in a clean way, e. g., we know that regions of interest that we want to extract are separated by semicolons, but semicolons can also occur inside these regions with some other meaning. In this case, it can make sense to allow extracted spans to overlap each other in order to cover all possibilities. But a reasonable real-world example of an information extraction task for which it is necessary to extract overlapping spans with a regular spanner *and* then use string-equality selections *on these overlapping spans* is arguably rather hard to find. The fact that string-equality selections on overlapping spans can be used for describing complicated word combinatorial properties (see Section 2.4) raises the question whether we can avoid some complexity issues of core spanners by prohibiting overlapping spans to be subjected to string-equality selections. Note that such a restriction is much weaker than restricting to *hierarchical* spanners, since we can still extract overlapping spans, we just are not allowed to use the string-equality operator on overlapping spans (in particular, the full class of regular spanners — which are not necessarily hierarchical — is still covered).

The considerations from above are the motivation for the class of *refl-spanners* introduced in [38]. Refl-spanners are designed to

properly lie in between regular spanners and core spanners, with the overall goal of having an expressive power that covers the relevant features of core spanners, while at the same time having complexity and decidability properties that are better than that of the full class of core spanners. In the remainder of this section, we discuss the main idea of this class of spanners and give a brief overview of its properties.

3.1 String-Equalities as Meta Symbols

The only non-regular feature of core spanners is the string-equality selection. This is immediately clear on an intuitive level, and it is also demonstrated by the core-simplification lemma (cf. Section 2.3): a regular spanner plus string-equality selections plus a projection can describe every core spanner. The main idea of refl-spanners is to express — to at least some extent — the string-equality selections also as a regular feature, i. e., by meta symbols in the regular subword-marked language that describes the regular spanner. This yields a class of spanners that, just like regular spanners, are fully described by regular languages (with meta symbols) and therefore by automata, regular expressions etc. As a (intended) side effect of this different conceptual approach, refl-spanners are not capable to describe some of those features discussed above, i. e., features that are obviously problematic with respect to complexity and decidability while at the same time seem not to have substantial practical relevance.

The overall idea is very simple and directly extends a known-approach from regular expressions with backreferences (see [14, 37]): In addition to markers $\mathsf{x}\triangleright$ and $\triangleleft\mathsf{x}$ (that describe the span extracted by variable x) in subword-marked words, we also allow x as a meta symbol in subword-marked words, which describes a *copy* or *reference* of whatever factor is extracted in the span of variable x. In order to make this meaningful, we only have to forbid that x occurs between $\mathsf{x}\triangleright$ and $\triangleleft\mathsf{x}$. This extension of the concept of subword-marked words (and languages) is called *ref-words* (and *ref-languages*).[1]

Obviously, the idea is that references x in a ref-word describe string-equality selections. Let us illustrate this with an example. Consider the regular spanner represented by

$$\alpha \;:=\; \mathsf{a\,b}^*\,\mathsf{x}\triangleright\,(\mathsf{a}\vee\mathsf{b})^*\,\triangleleft\mathsf{x}\,(\mathsf{b}\vee\mathsf{c})^*\,\mathsf{y}\triangleright\,(\mathsf{a}\vee\mathsf{b})^*\,\triangleleft\mathsf{y}\,\mathsf{b}^*, \qquad (2)$$

and assume we are interested in the core spanner $S := \varsigma^{=}_{\{\mathsf{x},\mathsf{y}\}}(\llbracket\alpha\rrbracket)$. The subword-marked language $L(\alpha)$ contains words like

$$\mathsf{a}\,\mathsf{x}\triangleright\,\mathsf{aba}\,\triangleleft\mathsf{x}\;\;\mathsf{y}\triangleright\,\mathsf{ab}\,\triangleleft\mathsf{y} \qquad \text{and} \qquad \mathsf{a}\,\mathsf{x}\triangleright\,\mathsf{aa}\,\triangleleft\mathsf{x}\;\;\mathsf{y}\triangleright\,\mathsf{ab}\triangleleft\mathsf{y}$$

which, due to the string-equality selection $\varsigma^{=}_{\{\mathsf{x},\mathsf{y}\}}$ will not represent a row in any table extracted by the spanner S. In fact, the only part of $L(\alpha)$ that matters for S is its subset

$$\bigcup_{u\in\{\mathsf{a},\mathsf{b}\}^*} L(\mathsf{a\,b}^*\,\mathsf{x}\triangleright\,u\,\triangleleft\mathsf{x}\,(\mathsf{b}\vee\mathsf{c})^*\,\mathsf{y}\triangleright\,u\,\triangleleft\mathsf{y}\,\mathsf{b}^*).$$

Consequently, instead of using the subword-marked language given by α, we use the *ref-language* given by

$$\alpha' \;:=\; \mathsf{a\,b}^*\,\mathsf{x}\triangleright\,(\mathsf{a}\vee\mathsf{b})^*\,\triangleleft\mathsf{x}\,(\mathsf{b}\vee\mathsf{c})^*\,\mathsf{y}\triangleright\,\mathsf{x}\,\triangleleft\mathsf{y}\,\mathsf{b}^*. \qquad (3)$$

A given ref-word w describes a document and a span tuple as follows. First, by $\mathfrak{d}(w)$, we denote the "de-referenced version of w", i.e., the subword-marked word that is obtained from w by replacing each x by whatever appears in between $\mathsf{x}\triangleright$ and $\triangleleft\mathsf{x}$ (in general, this has to be done with some care due to possible nesting), e. g., $\mathfrak{d}(\mathsf{a}\,\mathsf{x}\triangleright\,\mathsf{aba}\,\triangleleft\mathsf{x}\;\;\mathsf{y}\triangleright\,\mathsf{x}\triangleleft\mathsf{y}) = \mathsf{a}\,\mathsf{x}\triangleright\,\mathsf{aba}\,\triangleleft\mathsf{x}\;\;\mathsf{y}\triangleright\,\mathsf{aba}\triangleleft\mathsf{y}$. Since $\mathfrak{d}(w)$ is a subword-marked word, it describes a document $\mathbf{D}$ and $(\mathcal{X},\mathbf{D})$-tuple as usual. For our particular example of (3), the ref-words of the regular ref-language $L(\alpha')$ describe exactly the non-regular core spanner $S := \varsigma^{=}_{\{\mathsf{x},\mathsf{y}\}}(\llbracket\alpha\rrbracket)$. In general, the formal definition of Section 2.1 extends as follows: Any regular ref-language L describes the *refl-spanner* $\llbracket L\rrbracket$ where, for every document $\mathbf{D}\in\Sigma^*$ we have

$$\llbracket L\rrbracket(\mathbf{D}) \;:=\; \big\{\,\mathsf{st}(\mathfrak{d}(w)) \;:\; w\in L,\;\; \mathfrak{e}(\mathfrak{d}(w)) = \mathbf{D}\,\big\}.$$

In symbols, this can also be written as $\llbracket L\rrbracket = \llbracket\mathfrak{d}(L)\rrbracket$ where $\mathfrak{d}(L) := \{\mathfrak{d}(w) : w\in L\}$ is just a subword-marked language. Obviously, this framework can be used for all kinds of ref-languages (not only regular ones), but the class of *refl-spanners* from [38] is based on *regular* ref-languages.

Just like regular spanners, refl-spanners are solely defined by finite automata (or other regular language description mechanisms), namely automata that accept regular ref-languages.

Let us consider a further, more involved example, where

$$\alpha \;:=\; \mathsf{x}\triangleright\,\mathsf{a}^*\,\mathsf{y}\triangleright\,\mathsf{b}^*\,\triangleleft\mathsf{x}\,\mathsf{a}^*\mathsf{c}^*\mathsf{x}\,\triangleleft\mathsf{y}\,\mathsf{abc\,y}.$$

An example of a ref-word in $L(\alpha)$ is $w := \mathsf{x}\triangleright\mathsf{aa}\,\mathsf{y}\triangleright\mathsf{bbb}\triangleleft\mathsf{x}\mathsf{cc}\,\mathsf{x}\triangleleft\mathsf{y}\,\mathsf{abc\,y}$. For obtaining the described document $\mathbf{D}$, the $\mathfrak{d}(\cdot)$ function needs to first substitute the reference x and then the reference y (since the latter depends on x):

$$\begin{aligned}
&\mathsf{x}\triangleright\,\mathsf{aa}\,\mathsf{y}\triangleright\,\mathsf{bbb}\,\triangleleft\mathsf{x}\,\mathsf{cc\,x}\,\triangleleft\mathsf{y}\,\mathsf{abc\,y} &&\rightsquigarrow\\
&\mathsf{x}\triangleright\,\mathsf{aa}\,\mathsf{y}\triangleright\,\mathsf{bbb}\,\triangleleft\mathsf{x}\,\mathsf{ccaabbb}\,\triangleleft\mathsf{y}\,\mathsf{abc\,y} &&\rightsquigarrow\\
&\mathsf{x}\triangleright\,\mathsf{aa}\,\mathsf{y}\triangleright\,\mathsf{bbb}\,\triangleleft\mathsf{x}\,\mathsf{ccaabbb}\,\triangleleft\mathsf{y}\,\mathsf{abcbbbccaabbb} &&\rightsquigarrow\\
&\mathsf{aabbbccaabbbabcbbbccaabbb}.
\end{aligned}$$

In the original core spanner formulation, we can define the spanner described by α as $\pi_{\{\mathsf{x},\mathsf{y}\}}\varsigma^{=}_{\{\mathsf{x},\mathsf{z}_1\}}\varsigma^{=}_{\{\mathsf{y},\mathsf{z}_2\}}(\llbracket\alpha'\rrbracket)$, where

$$\alpha' \;:=\; \mathsf{x}\triangleright\,\mathsf{a}^*\,\mathsf{y}\triangleright\,\mathsf{b}^*\,\triangleleft\mathsf{x}\,\mathsf{a}^*\mathsf{c}^*\,\mathsf{z}_1\triangleright\,\Sigma^*\,\triangleleft\mathsf{z}_1\;\;\triangleleft\mathsf{y}\,\mathsf{abc}\;\;\mathsf{z}_2\triangleright\,\Sigma^*\,\triangleleft\mathsf{z}_2\;.$$

Obviously, a reference x in a ref-language implements a string-equality selection *without* extracting a span, while a core spanner must extract spans in order to apply the string-equality selection. This is not a restriction for refl-spanners, since we can always also extract spans of references by using $\mathsf{z}_\mathsf{x}\triangleright\,\mathsf{x}\triangleleft\mathsf{z}_\mathsf{x}$ for a new variable z_x.

The crucial restriction of refl-spanners compared to core spanners is the following: If we want to implement a string-equality selection $\varsigma^{=}_{\{\mathsf{x},\mathsf{y}\}}$, but the brackets $\mathsf{x}\triangleright\cdots\triangleleft\mathsf{x}$ and $\mathsf{y}\triangleright\cdots\triangleleft\mathsf{y}$ enclose

[1] In the literature on spanners, subword-marked words (as defined here; see Section 2.1) are usually called *ref-words* (see, e. g., [7, 11, 13, 15]). This "misnomer" has historical reasons: Ref-words have originally been used in [37] (in the context of regular expressions with backreferences) as words that contain *references* x to some of their subwords, which are explicitly marked by brackets $\mathsf{x}\triangleright\cdots\triangleleft\mathsf{x}$. Consequently, the ref-words of [37] are syntactically the same as what is called *ref-words* in the context of refl-spanners. Inspired by the approach of [37], papers have adopted ref-words *without* any references x, but with the "subword marking property" by meta symbols $\mathsf{x}\triangleright\cdots\triangleleft\mathsf{x}$, in order to describe regular spanners. Especially for refl-spanners, we need ref-words in the sense of [37], i. e., with actual references, but also the variants without references (which are called *ref-words* in some papers), so [38] introduced the term *subword-marked word* for the latter. In the context of this development, it is also interesting to note that both the papers [9] (the origin of document spanners) and [37] (a work on regular expressions with backreferences) use — apparently without any knowledge of each other when published — similar automata and regular expression based concepts for describing spans. In fact, the *memory automata* from [37] can be seen as vset-automata that also evaluate string-equality selections on-the-fly.

other marker symbols, then we cannot simply replace one of them by a reference without substantially changing the meaning of the spanner. This obviously limits the expressive power of refl-spanners in comparison to the full class of core spanners. On the other hand, it also makes it impossible to define string-equality selections on overlapping spans, which, in terms of complexity and decidability, is an asset (see Section 3.3).

3.2 Expressive Power of Refl-Spanners

We first note that in order for refl-spanners to describe a subclass of core spanners, we have to restrict them in such a way that the underlying ref-language cannot have an unbounded number of references for some variable x, as, e. g., in the ref-language described by $^{x}\triangleright\, a^*b\, \triangleleft^{x}\, (x)^*$. We call a refl-spanner *reference-bounded* if it is described by a ref-language L for which there exists a number k such that for all $w \in L$ and $x \in \mathcal{X}$ we have $|w|_x \leqslant k$ (here, $|w|_x$ denotes the number of occurrences of the letter x in the word w). Without this restriction, regular ref-languages can describe spanners that provably are not core spanners, as demonstrated by the example $L(a^+\, ^{x}\triangleright\, b^+\, \triangleleft^{x}\, (a^+x)^*a^+)$ (see [9, Theorem 6.1])).

Any reference-bounded refl-spanner translates into a core spanner as follows. Intuitively, for every variable x, we use k new variables $y_{x,1}, y_{x,2}, \ldots, y_{x,k}$ (where k is the bound on the number of references), and we replace each reference x by $^{y_{x,i}}\triangleright\, \Sigma^*\, \triangleleft^{y_{x,i}}$ for some i, and finally use a string-equality selection $\zeta^{=}_{\{y_{x,1},\ldots,y_{x,k}\}}$.

The question which core spanners can be represented as refl-spanners is much more difficult. We already have seen a simple example above, namely, the core spanner $S := \zeta^{=}_{\{x,y\}}([\![\alpha]\!])$ from (2) can be represented by α' from (3). However, the situation slightly changes when replacing α with

$$\beta \;:=\; a\, b^*\, {}^{x}\triangleright\, a\, (a \vee b)^*\, \triangleleft^{x}\, (b \vee c)^*\, {}^{y}\triangleright\, (a \vee b)^*\, b\, \triangleleft^{y}\, b^*,$$

since now neither

$$a\, b^*\, {}^{x}\triangleright\, a\, (a \vee b)^*\, \triangleleft^{x}\, (b \vee c)^*\, {}^{y}\triangleright\, x\, \triangleleft^{y}\, b^*, \text{ nor}$$
$$a\, b^*\, {}^{x}\triangleright\, (a \vee b)^*\, b\, \triangleleft^{x}\, (b \vee c)^*\, {}^{y}\triangleright\, x\, \triangleleft^{y}\, b^*$$

is a correct representation. Luckily, in this case we can use

$$\beta' \;:=\; a\, b^*\, {}^{x}\triangleright\, \gamma\, \triangleleft^{x}\, (b \vee c)^*\, {}^{y}\triangleright\, x\, \triangleleft^{y}\, b^*,$$

where γ is a regular expression describing the language

$$L(a\,(a \vee b)^*) \;\cap\; L((a \vee b)^*\, b).$$

The situation gets much more difficult when considering core spanners with string-equality selections on nested spans, or even overlapping spans. For example, it seems difficult to transform

$$\zeta^{=}_{\{x,y\}}[\![L(\, {}^{x}\triangleright\, a^*\, \triangleleft^{x}\, {}^{y}\triangleright\, {}^{z}\triangleright\, a^*\, \triangleleft^{z}\, a^*\, \triangleleft^{y})]\!], \text{ or}$$
$$\zeta^{=}_{\{x,y\}}([\![L(\, {}^{x}\triangleright \ldots {}^{y}\triangleright \ldots \triangleleft^{x} \ldots \triangleleft^{y})]\!])$$

into a refl-spanner.

These examples suggest that the refl-spanner formalism is much less powerful than core spanners, which is to be expected, since we have to pay a price for the fact that we can solve many problems for refl-spanners much more efficiently than for core spanners (see Section 3.3).

As a main result of [38], it is shown that if a core spanner S only uses non-overlapping string-equality selections, then we can define a refl-spanner that extracts the same span relations as S, with the only difference that some columns are split into several columns. Let us explain this in some more detail.

A sequence of string-equality selections $\zeta^{=}_{\mathcal{Z}_1}, \ldots, \zeta^{=}_{\mathcal{Z}_k}$ is called *non-overlapping with respect to a regular spanner* S if there are no $x, y \in \mathcal{Z}_1 \cup \cdots \cup \mathcal{Z}_k$ with $x \neq y$ such that S extracts overlapping spans for x and y from any document.

Let t be a span tuple, let $\lambda \subseteq \mathcal{X}$ and let $x \notin \mathcal{X}$ be a new variable. We write $\uplus_{\lambda \to x}(t)$ to describe the span tuple t' in which the columns of variables in λ have been *fusioned* into a new column x as follows: $t'(x) = [i', j\rangle$, where i' is the minimum of all i that occur as "left bounds" of the spans $t(z) = [i, j\rangle$ of the variables $z \in \lambda$, and j' is the maximum of all j that occur as "right bounds" of the spans $t(z) = [i, j\rangle$ of the variables $z \in \lambda$. For example, let $t = ([1, 3\rangle, [2, 6\rangle, [3, 7\rangle)$ be a $(\{x_1, x_2, x_3\}, \mathbf{D})$-tuple, then $\uplus_{\{x_1,x_3\}\to y}(t) = ([1, 7\rangle, [2, 6\rangle)$ is a $(\{y, x_2\}, \mathbf{D})$-tuple. We lift this operation to an operation on spanners in the obvious way.

The mentioned result of [38] can now be stated as follows. For any core spanner $\zeta^{=}_{\mathcal{Z}_1} \cdots \zeta^{=}_{\mathcal{Z}_k}(S)$, such that S is a regular spanner and $\zeta^{=}_{\mathcal{Z}_1}, \ldots, \zeta^{=}_{\mathcal{Z}_k}$ are non-overlapping with respect to S, we can construct a refl-spanner S' such that

$$\zeta^{=}_{\mathcal{Z}_1} \cdots \zeta^{=}_{\mathcal{Z}_k}(S) \quad = \quad \biguplus_{\lambda_1 \to y_1} \cdots \biguplus_{\lambda_p \to y_p}(S').$$

Intuitively speaking, as long as the string-equality selections are non-overlapping, we can transform a core spanner into a refl-spanner, that describes the core spanner up to the difference that the actual columns are split into several columns.

3.3 Evaluation and Static Analysis of Refl-Spanners

With respect to the complexity and decidability of evaluation and static analysis problems, refl-spanners lie strictly between the classes of regular spanners and core spanners. Recall from Section 2.4 that the problem ModelChecking can be solved for regular spanners by checking if a finite automaton accepts a given word, while for the full class of core spanners the task is intractable. For refl-spanners, we can mimic the same approach that works for regular spanners — this is mainly due to the fact that refl-spanners can also be described by single NFAs. More precisely, if M is an NFA that represents a refl-spanner (i. e., it accepts a ref-language), then we can again combine t and $\mathbf{D}$ into a subword-marked word w. However, this w is not the ref-word for which we have to check acceptance with respect to M, but its image under $\mathfrak{d}(\cdot)$, i. e., the subword-marked word after replacing the references. Instead of checking $w \in L(M)$, we have to check whether M accepts some ref-word v with $\mathfrak{d}(v) = w$ (note that such a v is not uniquely determined by $\mathbf{D}$ and t). To this end, we interpret the x-arcs of M as paths reading the factor w_x of $\mathbf{D}$ that corresponds to reference x (note that w_x is uniquely described by $\mathbf{D}$ and the span $t(x)$), and then check whether w is accepted. If w is accepted, then the accepting run actually describes a ref-word v (via the replaced x-transitions), such that $\mathfrak{d}(v)$ is a subword-marked word with $\mathfrak{e}(\mathfrak{d}(v)) = \mathbf{D}$ and $\mathrm{st}(\mathfrak{d}(v)) = t$; thus, $t \in [\![M]\!](\mathbf{D})$. If w is rejected, then no such v can exist and therefore $t \notin [\![M]\!](\mathbf{D})$. Due to the replacement of x-arcs by paths of length $O(|\mathbf{D}|)$, this algorithm runs in time quadratic in $|\mathbf{D}|$, but, by using standard string data-structures, it can be improved to linear in $|\mathbf{D}|$ (i. e., with the *same* running time as for regular spanners); cf. [38].

In contrast, the problem NonEmptiness (i. e., checking $S(\mathbf{D}) \neq \emptyset$ when given a spanner S and a document $\mathbf{D}$), can be solved efficiently for regular spanners by just interpreting marker-transitions of the NFA as ε-transitions (where ε is the empty word) and checking if the word $\mathbf{D}$ is accepted by this modified NFA. For refl-spanners, however, this idea does not work and NonEmptiness is NP-hard [38] (just like for core spanners [12]).

The problem Satisfiability (i. e., checking for given spanner S whether there exists a document $\mathbf{D}$ with $S(\mathbf{D}) \neq \emptyset$), reduces to checking non-emptiness for an NFA in the case of regular spanners, and the same is true for refl-spanner. Thus, the problem can be solved efficiently for refl-spanners [38], while it is intractable for core-spanners [12].

The problem Containment is PSpace-complete for regular spanners, since the problem is closely related to the containment problem for regular languages. For core spanners, on the other hand, the problem is undecidable (in fact, not even semi-decidable) [12]. For refl-spanners, we can at least show that the problem is decidable if we restrict the refl-spanners in such a way that every reference is necessarily extracted by its own private extraction variable [38]. This seems like a strong restriction for refl-spanners, but note that for core spanners we necessarily have a rather similar situation: if we want to use string-equality selections on some spans, we have to explicitly extract them by variables first.

4 SPANNERS ON COMPRESSED DOCUMENTS

The underlying data model for document spanners are documents $\mathbf{D} \in \Sigma^*$, i.e., finite strings over a finite alphabet. Unlike relational data, strings usually contain many redundancies, and there are simple and straightforward techniques to use these redundancies for compression. This is especially true for natural language, but also for more abstract sequences like bio-sequences or sequential log-files of large systems.

A classical and widely used compression scheme for strings with substantial practical relevance are so-called *straight-line programs* (SLPs). Let us briefly describe this concept in a way that is tailored to the following exposition. An SLP $\mathcal{S}$ is a directed acyclic graph (DAG) whose nodes all have a *left* and a *right child*, except for the sinks T_x of the graph, which uniquely represent the symbols x of the alphabet Σ. For an illustration see Figure 1. Any node A with left and right children B and C represents the document $\mathfrak{D}(A) = \mathfrak{D}(B)\mathfrak{D}(C)$, where $\mathfrak{D}(T_x) = x$ for the sinks T_x. For example, for node B in Figure 1, we have

$$\mathfrak{D}(B) = \mathfrak{D}(E)\mathfrak{D}(C) = \mathfrak{D}(T_\mathsf{a})\mathfrak{D}(T_\mathsf{b})\mathfrak{D}(F)\mathfrak{D}(T_\mathsf{a}) \tag{4}$$

$$= \mathfrak{D}(T_\mathsf{a})\mathfrak{D}(T_\mathsf{b})\mathfrak{D}(T_\mathsf{b})\mathfrak{D}(T_\mathsf{c})\mathfrak{D}(T_\mathsf{a}) = \mathsf{abbca}\,. \tag{5}$$

By designating some of the nodes of an SLP to represent documents, an SLP represents a set of documents, which we will call a *document database*. In the example of Figure 1, these designated nodes are A_1, A_2 and A_3; thus, the SLP represents the document database

$$\mathsf{DDB}_\mathcal{S} = \{\mathfrak{D}(A_1), \mathfrak{D}(A_2), \mathfrak{D}(A_3)\}$$
$$= \{\mathsf{ababbcabca},\ \mathsf{bcabcaabbca},\ \mathsf{ababbca}\}\,.$$

The classical use of SLPs is as compressed representations of single strings. Their prominence in various areas of computer science is due to the fact that they are mathematically easy, and that they cover many practically applied dictionary-based compression schemes (e. g., run-length encoding and the Lempel-Ziv-family LZ77, LZ78, LZW, etc.). A comprehensive introduction to SLPs is beyond the scope of this overview (we refer to the survey [22] and the papers [21, 23, 28, 29, 36, 39, 41]). In the best case, SLPs can be exponentially smaller than the strings they represent, and since redundancies in texts are likely in practical scenarios, good compressibility is to be expected in most cases. Moreover, many fast (i. e., linear or near linear time) practical algorithms for computing SLPs exist.

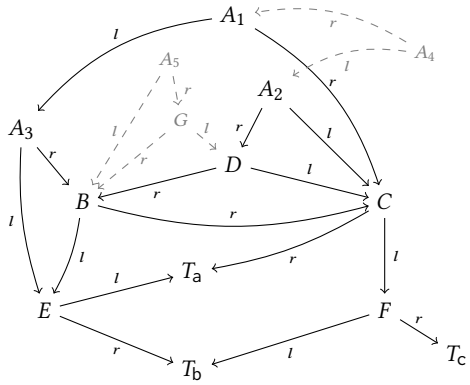

Figure 1: Graphical representation of an SLP (solid arcs). Left and right children are indicated by labels l and r, respectively. The grey parts illustrate the extensions discussed in Section 4.3. This figure is taken from [40].

The papers [39, 40] are devoted to the question of how regular document spanners can be evaluated directly over SLP-represented document databases. Let us explain the setting in a bit more detail. Our documents are stored in a document database $\mathsf{DDB} = \{\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_m\}$ that is represented by an SLP $\mathcal{S}$, i. e., for every i with $1 \leqslant i \leqslant m$ there is a node A_i in $\mathcal{S}$ with $\mathfrak{D}(A_i) = \mathbf{D}_i$ (again, see Figure 1 for an example of an SLP-represented document database that stores three documents). For a given regular spanner represented by an automaton M and some given i with $1 \leqslant i \leqslant m$, we wish to evaluate $[\![M]\!]$ on the document $\mathbf{D}_i$. The crucial point is that we want to evaluate $[\![M]\!]$ on $\mathbf{D}_i$ without explicitly constructing (i. e., decompressing) $\mathbf{D}_i$, which, after all, is stored in our document database (only) in a compressed form.[2] Since the SLP-compressed representation of the document might be exponentially smaller than the actual document, any algorithm that runs polynomially in the compressed size is, in the best case, only polylogarithmic in the actual document's size. This means that even an algorithm that is linear in the size of the document may be outperformed by an algorithm that is polynomial in the compressed size. For document spanners, this is witnessed by the main result of [39]: Given a document database $\mathsf{DDB} = \{\mathbf{D}_1, \ldots, \mathbf{D}_m\}$ represented by an SLP $\mathcal{S}$, an $i \in [m]$ and a regular spanner M, after a linear preprocessing in $\mathrm{O}(|\mathcal{S}|)$ (in data complexity),[3] we can enumerate $[\![M]\!](\mathbf{D}_i)$ with

[2] Such a task is also called *algorithmics on compressed strings* in the literature.
[3] We shall only discuss data complexity here; for upper bounds in combined complexity, we refer to the original papers.

delay $O(\log |\mathbf{D}_i|)$ (in data complexity). Let us discuss this result in a bit more detail (we discuss some technical aspects in Section 4.2).

First, observe that the preprocessing time $O(|\mathcal{S}|)$ depends on the compressibility of $\mathbf{D}_i$, i. e., in the best case, we have $|\mathcal{S}| = O(\log |\mathbf{D}_i|)$, and in the worst case, we have $|\mathcal{S}| = \Omega(|\mathbf{D}_i|)$. In contrast to this, the delay is logarithmic in $|\mathbf{D}_i|$ independently of the compressibility achieved by $\mathcal{S}$. This is obviously incomparable to the uncompressed setting, where one can achieve constant delay after $O(|\mathbf{D}_i|)$ preprocessing (cf. the results discussed in Section 2.5). There are nevertheless practical scenarios, where the compressed setting seems particularly useful. The compressed setting assumes that we get our documents already in an SLP-compressed form. This assumption is actually not a strong restriction, since there are many practical algorithms that compute SLPs for strings, and that both in terms of running time and achieved compressibility are — although not optimal[4] — rather efficient.[5] In the best case where the SLP is logarithmic in $|\mathbf{D}_i|$, spanner evaluation can be done logarithmically in the data size (i. e., sublinear in data complexity).

The paper [40] is devoted to the *dynamic setting* of spanner evaluation for SLP-represented document databases. The dynamic setting deals with the situation where the database is subject to update operations, and we wish to evaluate queries over the updated database without re-running the preprocessing from scratch. I. e., instead of treating the evaluation of the same query over a slightly changed database as a completely new evaluation problem, we want to profit from the fact that we have already done the preprocessing with respect to the query and the original database. For classical relational database systems, this scenario is completely natural: the system builds data structures that allow to efficiently compute the query result and maintains these data structures when the database is updated. The particular setting introduced in [40] is as follows.

We have available an SLP $\mathcal{S}$ that stores a document database $\mathrm{DDB} = \{\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_m\}$. Based on the results of [39], we have several data structures at hand that allow the enumeration of the results of each of the spanners $M_1, M_2, \ldots, M_k$ on any document $\mathbf{D}_i$ in DDB with delay $O(\log |\mathbf{D}_i|)$. We then want to manipulate the existing documents of DDB by a sequence of text-editing operations and evaluate the spanners $M_1, \ldots, M_k$ on the resulting document. For example, we may cut the subword from position 5 to 21 from document $\mathbf{D}_7$, insert it at position 12 into document $\mathbf{D}_3$, append this document to $\mathbf{D}_1$, and insert the resulting document into our document database — and we want to do this in such a way that each of the spanners $M_1, \ldots, M_k$ can be evaluated efficiently also on this new document. The way we build the new document is called *complex document editing* and can be described by an expression φ of a suitable algebra. In [40] it is shown that if $\mathcal{S}$ is *strongly balanced* (to be explained in more detail in Section 4.1), then we can construct the document $\mathbf{D}_\varphi$ described by φ and add it to the SLP representing DDB, update all data structures needed for enumerating the spanners $M_1, \ldots, M_k$, and maintain the balancedness property of the SLP — and all this can be done in time $O(k \cdot |\varphi| \cdot (|\varphi| + \log \mathbf{L}))$, where $\mathbf{L} := \max\{|\mathbf{D}_i| : 1 \leqslant i \leqslant m\}$. In other words, we can perform rather complex updates with only logarithmic dependency on the lengh of the actual documents, and — a rather important aspect — this is generally the case, independently from the SLP-compressibility of the documents.

The remainder of this section is devoted to some technical aspects of spanner evaluation on SLP-represented document databases. Section 4.1 discusses balancedness properties of SLPs, while Sections 4.2 and 4.3 focus on spanner evaluation in the static setting and the dynamic setting, respectively.

4.1 Balanced Straight-Line Programs

The results of [39, 40] hinge on *balancing* properties of SLPs, which we shall now briefly describe. The *order* of a node A (denoted by $\mathrm{ord}(A)$) of an SLP is $k+1$, where k is the longest path from A to a leaf (i. e., leafs have order 1). In particular, $\mathrm{ord}(A)$ denotes the number of iterative applications of the *derivation function* $\mathfrak{D}(\cdot)$ that are necessary in order to obtain the string over Σ represented by A (see (4) and (5)). We say that node A is *c-shallow* for some positive integer c if $\mathrm{ord}(A) \leqslant c \cdot \log |\mathfrak{D}(A)|$. If A has left child B and right child C, then we define $\mathrm{bal}(A) = \mathrm{ord}(B) - \mathrm{ord}(C)$, and we say that A is *balanced* if $\mathrm{bal}(A) \in \{-1, 0, 1\}$; and A is *strongly balanced* if A and all its descendants are balanced. We call an SLP $\mathcal{S}$ *strongly balanced*, (*c-shallow*, resp.) if all its inner nodes are strongly balanced (*c*-shallow, resp.). Obviously, *c*-shallowness as well as strongly balancedness can be viewed as natural properties of balancedness. Moreover, strongly balancedness corresponds to the balancing property that is also used for balanced search trees, e. g., AVL-trees. See [17, 18, 36] for more details on SLP-balancing.

Let us again consider the example of Figure 1. The orders of $\mathcal{S}$'s nodes are as follows: $\mathrm{ord}(F) = \mathrm{ord}(E) = 2$, $\mathrm{ord}(C) = 3$, $\mathrm{ord}(B) = 4$, $\mathrm{ord}(D) = \mathrm{ord}(A_3) = 5$, $\mathrm{ord}(A_1) = \mathrm{ord}(A_2) = 6$. In particular, all nodes are balanced except for A_1, A_2, A_3, since $\mathrm{bal}(A_1) = 2$ and $\mathrm{bal}(A_2) = \mathrm{bal}(A_3) = -2$.

We can observe the following important fact about strongly balanced nodes: Every directed path from a strongly balanced node A to some leaf has length at least $\frac{1}{2} \log |\mathfrak{D}(A)|$ and at most $2 \log |\mathfrak{D}(A)|$. Furthermore, $\log |\mathfrak{D}(A)| \leqslant \mathrm{ord}(A) - 1 \leqslant 2 \cdot \log |\mathfrak{D}(A)|$. In particular, this means that any strongly balanced SLP is also 2-shallow.

The main result of [39] crucially builds upon the balancing theorem of [18] stating that there is a constant c such that any given SLP $\mathcal{S}$ with a root A can be transformed in time $O(|\mathcal{S}|)$into a *c-shallow* SLP $\mathcal{S}'$ with a root A' such that $|\mathcal{S}'| = O(|\mathcal{S}|)$ and $\mathfrak{D}(A') = \mathfrak{D}(A)$. This means that when dealing with SLPs, we may assume that the given SLPs are *c*-shallow (since we can always ensure this property in linear time in a preprocessing step). Moreover, while we cannot assume that all documents are highly compressible by SLPs, we *can* always ensure *c*-shallowness, independently of compressibility issues.

For the more restrictive property of being *strongly balanced*, the situation is slightly different: From [36] we know that any given SLP $\mathcal{S}$ with a root A can be transformed into a strongly balanced SLP $\mathcal{S}'$ with a root A' such that $\mathfrak{D}(A') = \mathfrak{D}(A)$, but this construction takes time $O(|\mathcal{S}| \cdot \mathrm{ord}(A))$. Since we may assume that $\mathcal{S}$ is c-shallow, we can assume that this run-time, as well as $|\mathcal{S}'|$,

[4] Note that the problem of computing a *smallest* SLP for a string is NP-complete (see [3, 4]).

[5] In fact, the literature on algorithmics on compressed strings suggest that most basic string analysis tasks can be performed directly on SLPs. Thus, whenever we are dealing with textual data, it is even a likely scenario that strings are generally represented by SLPs, simply because we have the algorithmic machinery for working directly on SLPs.

are in $O(|\mathcal{S}| \cdot \log |\mathfrak{D}(A)|)$; and from [17] it is known that the factor $\log |\mathfrak{D}(A)|$ cannot be avoided.

4.2 Enumeration in the Compressed Setting

Intuitively speaking, the compression of an SLP is done by representing several occurrences of the same factor of a document by just a single node. Referring to Figure 1,

$$\mathfrak{D}(A_1) \;=\; \mathfrak{D}(E)\mathfrak{D}(E)\mathfrak{D}(C)\mathfrak{D}(C) \;=\; \mathsf{ababbcabca}\,.$$

Thus, the two occurrences of factor ab are represented by the same node E, and the two occurrences of factor bca are represented by the same node C. However, the span-tuples to be extracted may treat different occurrences of the same factor compressed by the same node in different ways. For example, a spanner may extract the span-tuple that corresponds to $\mathsf{ababb}\,{}^{x}\!\triangleright\,\mathsf{cabc}\,\triangleleft^{x}\,\mathsf{a}$. This messes up the compression, since the two occurrences of $\mathfrak{D}(C) = \mathsf{bca}$ have now become two different factors: $\mathsf{b}\,{}^{x}\!\triangleright\,\mathsf{ca}$ and $\mathsf{bc}\,\triangleleft^{x}\,\mathsf{a}$. So it seems that extracting a span-tuple enforces at least a partial decompression of $\mathcal{S}$, because different occurrences of the same factor need to be treated differently.

The technical challenge that we face also becomes clear by a comparison to the approach of [2] for spanner evaluation in the uncompressed case. This approach first computes in the preprocessing *one* data structure that represents the whole solution set (i. e., the product graph of spanner and document), and then the enumeration is done by systematically searching this data structure (with the help of additional, pre-computed information). Since each position of the document might be the start or end position of some extracted span, it is difficult to imagine such a data structure that is not at least as large as the whole document. In any case, such a data-structure cannot explicitly contain each position of the represented document, and therefore must still respect the compression.

Let us take a look at the (classical and well-investigated) task of checking whether an SLP-compressed string is accepted by a given NFA (note that algorithms for evaluating regular spanners over SLP-compressed documents will necessarily also implicitly solve this task in some way). Let S be some node of an SLP $\mathcal{S}$ (over Σ), let M be an NFA over Σ with states $\{s_1, s_2, \ldots, s_n\}$. We want to check whether $\mathfrak{D}(S) \in L(M)$. The general idea is to compute, for each node A that is reachable from S, a Boolean $(n \times n)$ matrix M_A whose entries indicate from which state we can reach which state by reading the string $\mathfrak{D}(A)$. This can be done recursively bottom-up along the DAG rooted by S: the matrices M_{T_x} for the leaf non-terminals are directly given by M's transition function, and for every node A with left and right children B and C, we have $M_A = M_B \cdot M_C$, where $\cdot$ denotes the Boolean matrix multiplication. This means that we can in fact check $\mathfrak{D}(S) \in L(M)$ in time $O(|\mathcal{S}|n^3)$ (this observation is well-known; see, e. g., [22, 25, 34])).

In principle, the enumeration algorithm of [39] extends the idea sketched above, but in a non-trivial way. It uses suitable variants and extensions of the matrices M_A. In the enumeration phase it considers trees that represent partially decompressed versions of the SLP. The logarithmic delay bound is ensured by transforming (in the preprocessing phase) the SLP into a c-shallow SLP (cf. Section 4.1).

4.3 Complex Document Editing

We can update an SLP-represented document database DDB by directly adding new nodes (with left and right arcs) to the SLP. As an example, consider the grey part of Figure 1: by adding the new nodes A_4, A_5 and G in the way described in Figure 1, we add to DDB a document $\mathbf{D}_4 = \mathbf{D}_2 \cdot \mathbf{D}_1$ represented by A_4, and a document

$$\mathbf{D}_5 = \mathfrak{D}(B)\mathfrak{D}(G) = \mathfrak{D}(B)\mathfrak{D}(D)\mathfrak{D}(B) = \mathsf{abbcabcaabbcaabbca}\,,$$

represented by A_5. Similarly, we can add any document that is defined as concatenation of documents that are already represented by individual nodes of the SLP. Furthermore, the information computed in the preprocessing of the enumeration algorithm (see Section 4.2) can be easily updated.

The situation becomes more difficult if we want to do more complicated updates, such as "construct the document obtained by inserting the subword from position 5 to 21 of document $\mathbf{D}_7$ at position 12 into document $\mathbf{D}_3$ and appending this document to $\mathbf{D}_1$", and if we want to maintain some kind of balancing property. The latter will be important not only for ensuring a logarithmic delay in the enumeration of spanners, but also for performing updates in logarithmic time.

The types of updates considered in [40] are expressions over the following basic operations (note that the latter three can be described by suitable combinations of the former two):

- $\mathsf{concat}(\mathbf{D}, \mathbf{D}')$**:** Concatenate $\mathbf{D}$ and $\mathbf{D}'$.
- $\mathsf{extract}(\mathbf{D}, i, j)$**:** Extract the factor from position i to j from $\mathbf{D}$.
- $\mathsf{delete}(\mathbf{D}, i, j)$**:** Delete the factor from position i to j from $\mathbf{D}$.
- $\mathsf{insert}(\mathbf{D}, \mathbf{D}', k)$**:** Insert $\mathbf{D}'$ at position k of $\mathbf{D}$.
- $\mathsf{copy}(\mathbf{D}, i, j, k)$**:** Considering $\mathbf{D}$, copy the factor from position i to j and paste it at position k.

A *CDE-expression* φ over a document database $\mathsf{DDB} = \{\mathbf{D}_1, \ldots, \mathbf{D}_m\}$ is obtained by nested application of the basic CDE-algebra operations described above. By $\mathsf{eval}(\varphi)$ we denote the document described by φ. The update task can now be formalised as follows. We have a document database DDB represented by an SLP $\mathcal{S}$, we receive a CDE-expression φ, and we want to modify $\mathcal{S}$ in such a way that it describes $\mathsf{DDB} \cup \{\mathsf{eval}(\varphi)\}$. As illustrated by Figure 1, this can be done easily in the special case that φ only contains operation concat. The approach for arbitrary CDE-expressions is as follows.

We assume that the SLP that represents DDB is *strongly balanced* (see Section 4.1).

Let us consider the operation $\mathsf{concat}(\mathfrak{D}(B), \mathfrak{D}(C))$, where B and C are some nodes of the SLP. If we just naively add a new node A with left and right children B and C, then it represents $\mathsf{concat}(\mathfrak{D}(B), \mathfrak{D}(C))$. However, if $\mathsf{ord}(B) \geqslant \mathsf{ord}(C) + 2$, then A is unbalanced. This case has been handled in [36] by first identifying in the part of the SLP that is rooted by B a suitable position where C can be inserted in such a way that we only obtain nodes that are unbalanced by at most 2 or -2; such mildly unbalanced nodes can then be re-balanced by performing suitable rotations (these are rotations that are similar to the classical rotations that are used for rebalancing AVL-trees). An analogous (but substantially more involved) construction (desribed in [40]) also works for the operation $\mathsf{extract}(\mathfrak{D}(A), i, j)$. The crucial point is that for these constructions, we only have to move (and manipulate) nodes

for a constant number of times along a path starting in A. Since the original SLP is strongly balanced, this implies that the overall running time is bounded by $O(\mathrm{ord}(A)) = O(\log |\mathfrak{D}(A)|)$.

Given a CDE-expression φ, we can add $\mathrm{eval}(\varphi)$ to the SLP representing the document database by applying these constructions inductively, i. e., bottom-up to the syntax tree of φ. This yields the following main result from [40]: Let DDB be a document database that is represented by a strongly balanced SLP $\mathcal{S}$. When given a CDE-expression φ over DDB, we can turn $\mathcal{S}$ into a strongly balanced SLP that represents $\mathrm{DDB} \cup \{\mathrm{eval}(\varphi)\}$. Moreover, this construction is carried out in time $O(|\varphi| \cdot \log \mathbf{d})$, where $\mathbf{d}$ is the maximum length of any document of DDB that occurs at a leaf of φ's syntax tree, any intermediate document that is represented by a subexpression of φ, and the finally resulting document $\mathrm{eval}(\varphi)$.

Within the running time of performing a CDE-update, we can also update the data structures that are necessary to perform the enumeration phase of the algorithm sketched in Section 4.2. . This implies that if we have computed these data structures for spanners $M_1, M_2, \ldots, M_k$ (represented by NFA M_i), then, for every $i \in [k]$ and CDE-expression φ, we can always enumerate $[\![M_i]\!](\mathrm{eval}(\varphi))$ by first updating DDB according to φ and then running the enumeration algorithm. Both the required update as well as the delay is only logarithmic in the data size.

If, for some CDE-expression φ, the actual document $\mathrm{eval}(\varphi)$ is to be queried just once without the necessity of permanently storing it in the document database, then we can just remove all the new nodes after having queried it, i. e., this can be done within the time required for constructing the node that represents $\mathrm{eval}(\varphi)$.

In comparison to the enumeration algorithm sketched in Section 4.2, the mentioned results on CDE-updates require the SLP to be strongly balanced instead of only c-shallow. As discussed in Section 4.1, the strongly balancedness property is stronger than c-shallowness, and it cannot be ensured in linear time.

Let us conclude with a discussion on how to build, from scratch and with acceptable running time, an SLP-representation for a document database. Let $\mathrm{DDB} = \{\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_m\}$. If the documents $\mathbf{D}_i$ are given as SLPs $\mathcal{S}_i$, then we can make each of those strongly balanced in time $O(|\mathcal{S}_i| \log |\mathbf{D}_i|)$ (see Section 4.1), and then we can just combine all these SLPs in order to get an SLP for DDB. In this regard, it is also worth noting that many practical compression schemes can be transformed directly into SLPs with moderate size blow-ups (see [19]). If, on the other hand, the documents are given in an uncompressed form, then we first have to compress them by running one of the many SLP-compression algorithms. In this setting, in order to achieve a better overall compression rate, it can also make sense to construct an SLP for the single document $\mathbf{D}_1\mathbf{D}_2 \cdots \mathbf{D}_m$, which can then be made strongly balanced, and then transformed into an SLP that contains a node for each $\mathbf{D}_i$ (this latter step can be done by CDE-operations).

ACKNOWLEDGMENTS

The first author is supported by the German Research Foundation — project number 416776735 (gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 416776735).
The second author is partially supported by the ANR project EQUUS ANR-19-CE48-0019; funded by the German Research Foundation — project number 431183758 (gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 431183758).

REFERENCES

[1] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2020. Constant-Delay Enumeration for Nondeterministic Document Spanners. *SIGMOD Rec.* 49, 1 (2020), 25–32. https://doi.org/10.1145/3422648.3422655

[2] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2021. Constant-Delay Enumeration for Nondeterministic Document Spanners. *ACM Trans. Database Syst.* 46, 1 (2021), 2:1–2:30. https://doi.org/10.1145/3436487 Confernce version appeared in Proc. ICDT 2019..

[3] K. Casel, H. Fernau, S. Gaspers, B. Gras, and M.L. Schmid. 2020. On the Complexity of the Smallest Grammar Problem over Fixed Alphabets. *Theory of Computing Systems* (2020). https://doi.org/10.1007/s00224-020-10013-w Conference version appeared in Proc. ICALP 2016..

[4] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. 2005. The smallest grammar problem. *IEEE Transactions on Information Theory* 51, 7 (2005), 2554–2576.

[5] Joel D. Day and Florin Manea. 2020. On the Structure of Solution Sets to Regular Word Equations. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*. 124:1–124:16. https://doi.org/10.4230/LIPIcs.ICALP.2020.124

[6] Volker Diekert. 2015. More Than 1700 Years of Word Equations. In *Algebraic Informatics - 6th International Conference, CAI 2015, Stuttgart, Germany, September 1-4, 2015. Proceedings*. 22–28. https://doi.org/10.1007/978-3-319-23021-4_2

[7] Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, and Frank Neven. 2019. Split-Correctness in Information Extraction. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. 149–163. https://doi.org/10.1145/3294052.3319684

[8] Johannes Doleschal, Benny Kimelfeld, Wim Martens, and Liat Peterfreund. 2020. Weight Annotation in Information Extraction. In *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*. 8:1–8:18. https://doi.org/10.4230/LIPIcs.ICDT.2020.8

[9] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. 2015. Document Spanners: A Formal Approach to Information Extraction. *J. ACM* 62, 2 (2015), 12:1–12:51. Conference version appeared in Proc. PODS 2013..

[10] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. 2020. Efficient Enumeration Algorithms for Regular Document Spanners. *ACM Trans. Database Syst.* 45, 1 (2020), 3:1–3:42. https://doi.org/10.1145/3351451 Conference version appeared in Proc. PODS 2018..

[11] D. Freydenberger. 2019. A Logic for Document Spanners. *Theory Comput. Syst.* 63, 7 (2019), 1679–1754. https://doi.org/10.1007/s00224-018-9874-1 Conference version appeared in Proc. ICDT 2017..

[12] D. Freydenberger and M. Holldack. 2018. Document Spanners: From Expressive Power to Decision Problems. *Theory Comput. Syst.* 62, 4 (2018), 854–898. Conference version appeared in Proc. ICDT 2016..

[13] Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. 2018. Joining Extractions of Regular Expressions. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*. 137–149. https://doi.org/10.1145/3196959.3196967

[14] Dominik D. Freydenberger and Markus L. Schmid. 2019. Deterministic regular expressions with back-references. *J. Comput. Syst. Sci.* 105 (2019), 1–39. https://doi.org/10.1016/j.jcss.2019.04.001 Conference version appeared in Proc. STACS 2017..

[15] Dominik D. Freydenberger and Sam M. Thompson. 2020. Dynamic Complexity of Document Spanners. In *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*. 11:1–11:21. https://doi.org/10.4230/LIPIcs.ICDT.2020.11

[16] Dominik D. Freydenberger and Sam M. Thompson. 2022. Splitting Spanner Atoms: A Tool for Acyclic Core Spanners. In *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference)*. 10:1–10:18. https://doi.org/10.4230/LIPIcs.ICDT.2022.10

[17] Moses Ganardi. 2021. Compression by Contracting Straight-Line Programs. In *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*. 45:1–45:16. https://doi.org/10.4230/LIPIcs.ESA.2021.45

[18] Moses Ganardi, Artur Jez, and Markus Lohrey. 2021. Balancing Straight-line Programs. *J. ACM* 68, 4 (2021), 27:1–27:40. https://doi.org/10.1145/3457389

[19] K. Goto, S. Maruyama, S. Inenaga, H. Bannai, H. Sakamoto, and M. Takeda. 2011. Restructuring Compressed Texts without Explicit Decompression. *CoRR* abs/1107.2729 (2011). http://arxiv.org/abs/1107.2729

[20] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to automata theory, languages, and computation, 3rd Edition*. Addison-Wesley.

[21] J. C. Kieffer and E.-H. Yang. 2000. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. on Information Theory* 46, 3 (2000), 737–754.

[22] M. Lohrey. 2012. Algorithmics on SLP-compressed strings: A survey. *Groups Complex. Cryptol.* 4, 2 (2012), 241–299. https://doi.org/10.1515/gcc-2012-0016

[23] M. Lohrey. 2014. *The Compressed Word Problem for Groups* (Springer Briefs in Mathematics ed.). Springer.

[24] Florin Manea and Markus L. Schmid. 2019. Matching Patterns with Variables. In *Combinatorics on Words - 12th International Conference, WORDS 2019, Loughborough, UK, September 9-13, 2019, Proceedings*. 1–27. https://doi.org/10.1007/978-3-030-28796-2_1

[25] N. Markey and P. Schnoebelen. 2004. A PTIME-complete matching problem for SLP-compressed words. *Inf. Process. Lett.* 90, 1 (2004), 3–6.

[26] Alexandru Mateescu and Arto Salomaa. 1997. Aspects of Classical Language Theory. In *Handbook of Formal Languages, Volume 1: Word, Language, Grammar*. 175–251. https://doi.org/10.1007/978-3-642-59136-5_4

[27] Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. 2018. Document Spanners for Extracting Incomplete Information: Expressiveness and Complexity. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*. 125–136. https://doi.org/10.1145/3196959.3196968

[28] C. Nevill-Manning and I. Witten. 1997. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *J. Artif. Intelligence Research* 7 (1997), 67–82.

[29] C. G. Nevill-Manning. 1996. *Inferring Sequential Structure*. Ph. D. Dissertation. University of Waikato, NZ.

[30] L. Peterfreund. 2019. *The Complexity of Relational Queries over Extractions from Text*. Ph. D. Dissertation.

[31] Liat Peterfreund. 2021. Grammars for Document Spanners. In *24th International Conference on Database Theory, ICDT 2021, March 23-26, 2021, Nicosia, Cyprus*. 7:1–7:18. https://doi.org/10.4230/LIPIcs.ICDT.2021.7

[32] Liat Peterfreund, Dominik D. Freydenberger, Benny Kimelfeld, and Markus Kröll. 2019. Complexity Bounds for Relational Algebra over Document Spanners. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. 320–334.

[33] Liat Peterfreund, Balder ten Cate, Ronald Fagin, and Benny Kimelfeld. 2019. Recursive Programs for Document Spanners. In *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*. 13:1–13:18.

[34] W. Plandowski and W. Rytter. 1999. Complexity of Language Recognition Problems for Compressed Words. In *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*. 262–272.

[35] John Michael Robson and Volker Diekert. 1999. On Quadratic Word Equations. In *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings*. 217–226. https://doi.org/10.1007/3-540-49116-3_20

[36] W. Rytter. 2003. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.* 302, 1-3 (2003), 211–222.

[37] Markus L. Schmid. 2016. Characterising REGEX languages by regular languages equipped with factor-referencing. *Information and Computation (I&C)* 249 (2016), 1–17. Conference version appeared in Proc. DLT 2014..

[38] Markus L. Schmid and Nicole Schweikardt. 2021. A Purely Regular Approach to Non-Regular Core Spanners. In *24th International Conference on Database Theory, ICDT 2021, March 23-26, 2021, Nicosia, Cyprus*. 4:1–4:19. https://doi.org/10.4230/LIPIcs.ICDT.2021.4

[39] Markus L. Schmid and Nicole Schweikardt. 2021. Spanner Evaluation over SLP-Compressed Documents. In *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Virtual Event, China, June 20-25, 2021*. 153–165. https://doi.org/10.1145/3452021.3458325

[40] Markus L. Schmid and Nicole Schweikardt. 2022. Query Evaluation Over SLP-Represented Document Databases With Complex Document Editing. In *PODS'22: Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*.

[41] J. A. Storer and T. G. Szymanski. 1982. Data compression via textual substitution. *Journal of the ACM* 29, 4 (1982), 928–951.