




The Information Extraction Framework of Document Spanners - A Very Informal Survey

Markus L. Schmid^(✉) 

Humboldt-Universität zu Berlin, Berlin, Germany
MLSchmid@MLSchmid.de

Abstract. This document provides an intuitive and high-level survey of the information extraction framework of *document spanners* (Fagin, Kimelfeld, Reiss, and Vansummeren (PODS 2013, J. ACM 2015)). Originally, document spanners were presented as a formalisation of the query language AQL, which is used in IBM's information extraction engine SystemT, and over the last decade this framework is heavily investigated in the database theory community. The research topic of document spanners combines classical results from areas like formal languages, algorithms and database theory, while at the same time posing challenging new research questions.

This survey is aimed at a general theoretical computer science audience that is not necessarily familiar with database theory. Its focus are the topics of an invited talk at SOFSEM 2024.

Disclaimer

This survey particularly aims at providing an intuitive introduction to the topic of document spanners, and a list of pointers to the relevant literature. Whenever possible, we will neglect formal definitions, explain technical concepts with only examples, and discuss theoretical results in an intuitive way. We do not assume the reader to be familiar with aspects of data management that would exceed the common knowledge of most computer scientists. For more technically detailed surveys (that are particularly directed at a database theory audience), the reader is referred to [1, 28].

1 Document Spanners

Document spanners are a relatively new research area that has received a lot of attention in the database theory community over the last ten years or so. An interesting fact is that the topic is motivated by practical considerations, but its theoretical foundation uses very classical and old concepts from theoretical computer science, like regular expressions, finite automata and, in general, regular languages. In order to substantiate the claim that document spanners constitute

a quite relevant research area, we will just state a long list of recent papers (all of them published within the last 10 years) that all have to do with this topic. Here it comes: [2–4, 6–15, 19–24, 26, 27, 29].¹ Let us now explain what document spanners are.

Document spanners have been introduced in [9], and they are a framework for extracting information from texts (i.e., strings, sequences or words, or, as is the common term in the data management community, *documents*); it is therefore called an *information extraction framework*. Since strings are not tables as we know them from relational databases, the information they represent is usually considered by database people as being not structured, or, to use a less derogatory term, to be only *semi-structures*.² Therefore, we would like to process a string (or let’s try to stick to the term *document* in the following (but we keep in mind that documents are nothing but strings in the sense of finite sequences of elements from a finite alphabet)), so we would like to process a document and extract (some of) its information in a structured way, i.e., as a table as found in relational databases, so with a fixed label for every column. Relevant parts of the string should then appear as entries of the cells (put into relation by the rows of the table as usual), but we are not really interested in having substrings of our document in the cells of the table. Instead we use just pointers to substrings, which are represented by the start and the endpoint of the substring. Such a ‘pointer-to-substring’ is called a *span*. So the span (2, 4) represents the first occurrence of the substring **ana** in the document $\mathbf{D} = \mathbf{banana}$, and (4, 6) represents the second occurrence of **ana**.³ Not very surprisingly, we are not just interested in the existence of substrings, but also where they occur; thus, the span representation is useful. In the following, we use the notation $\mathbf{D}[i, j]$ to refer to the content of span (i, j) of document \mathbf{D} , so for $\mathbf{D} = \mathbf{banana}$, we have $\mathbf{D}[2, 4] = \mathbf{D}[4, 6] = \mathbf{ana}$.

In order to say how this table of spans that we want to extract from the document looks like, we also have to label its columns (thereby also postulating how many columns we have), and we do this by simply giving a set of *variables*, e.g., $\mathcal{X} = \{x, y, z\}$. So with respect to \mathcal{X} , a possible table to be extracted from a document can look like the table here to the right (such tables are also called *\mathcal{X} -span relations*, and their rows are called *span tuples* (or *\mathcal{X} -span tuples*):

¹ Whether a research area should be considered important or not is always quite subjective. At the very least we can observe that many researchers like to work in the area of document spanners right now.

² The fate of representing data in a way that is only semi-structured is also shared by trees and graphs.

³ In the literature, the span (4, 6) is actually represented as [4, 7), which has some reasons, but in this survey we abstract from several such details that are not needed on this high level discussion.

$\mathbf{D} = \text{abbabccabc}$	\implies	<table style="border-collapse: collapse; text-align: center;"> <tr> <td style="border: 1px solid black; padding: 2px 5px;">x</td> <td style="border: 1px solid black; padding: 2px 5px;">y</td> <td style="border: 1px solid black; padding: 2px 5px;">z</td> </tr> <tr> <td style="border: 1px solid black; padding: 2px 5px;">(2, 5)</td> <td style="border: 1px solid black; padding: 2px 5px;">(4, 7)</td> <td style="border: 1px solid black; padding: 2px 5px;">(1, 10)</td> </tr> <tr> <td style="border: 1px solid black; padding: 2px 5px;">(3, 5)</td> <td style="border: 1px solid black; padding: 2px 5px;">(5, 8)</td> <td style="border: 1px solid black; padding: 2px 5px;">(4, 7)</td> </tr> <tr> <td style="border: 1px solid black; padding: 2px 5px;">(1, 3)</td> <td style="border: 1px solid black; padding: 2px 5px;">(3, 10)</td> <td style="border: 1px solid black; padding: 2px 5px;">(2, 4)</td> </tr> <tr> <td style="border: 1px solid black; padding: 2px 5px;">⋮</td> <td style="border: 1px solid black; padding: 2px 5px;">⋮</td> <td style="border: 1px solid black; padding: 2px 5px;">⋮</td> </tr> </table>	x	y	z	(2, 5)	(4, 7)	(1, 10)	(3, 5)	(5, 8)	(4, 7)	(1, 3)	(3, 10)	(2, 4)	⋮	⋮	⋮
x	y	z															
(2, 5)	(4, 7)	(1, 10)															
(3, 5)	(5, 8)	(4, 7)															
(1, 3)	(3, 10)	(2, 4)															
⋮	⋮	⋮															

Any function that, for a fixed set \mathcal{X} of variables, maps each document to a (possibly empty) \mathcal{X} -span relation is called a *spanner* (or *document spanner*, if there is no page limit), and actually we should add \mathcal{X} somewhere and rather say \mathcal{X} -spanner, because the set of variables is obviously relevant. The little picture above demonstrates this scenario (and the reader should play the fun game of finding all the factors of `abbabccabc` the spans of the table point to).

But let's not overdo it with the informal style of this survey and maybe fix at least some more precise notation. A span of a document \mathbf{D} is an element $(i, j) \in \{1, 2, \dots, |\mathbf{D}|\} \times \{1, 2, \dots, |\mathbf{D}|\}$ with $i \leq j$, a span tuple is a mapping from \mathcal{X} to the set of spans, and a spanner is a function that maps a document to a set of span tuples. Since it is awkward to write span tuples as functions, we use a tuple notation, i.e., we write $t = ((2, 5), (4, 7), (1, 10))$ instead of $t(x) = (2, 5)$, $t(y) = (4, 7)$ and $t(z) = (1, 10)$ (this only works if we fix some linear order for \mathcal{X} , but this will always be clear from the context).

This brief and informal explanation of the concept of document spanners is a sufficient basis for explaining the further concepts and results that are to follow. On the other hand, it is overly simplistic and makes the model look somewhat primitive, which does not do justice to the original paper [9], which is indeed a *seminal* paper. In particular, besides establishing some important conventions about spans and spanners and documents as data sources, the paper [9] also convincingly explains (also for researchers not too familiar with database theory), why document spanners cover relevant information extraction and therefore data management tasks. There is no need to further motivate document spanners here, since this has been done by [9] and the many papers that followed (we will, however, cite more actual literature later on).

2 Representations of Document Spanners

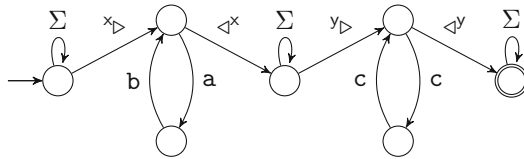
An important point is of course how to represent document spanners (so far, they are just abstractly defined as functions), and, as is common in database theory, we are not only interested in a mathematically rigorous formalisation, but we also want to provide a language for describing spanners that can be easily learned and applied by users (let's keep in mind that even though the theoretical research on document spanners is somewhat dominating, their original motivation was to describe a practically relevant information extraction framework, so a tool for users to tackle real-world data management tasks).

Historically, document spanners were defined by a two stage approach: First we use classes of regular language descriptors, like regular expressions and automata, to define spanners, and then we apply some relational algebra (i.e.,

operations on tables) on top of the span-relations that can be produced by those “regular spanners” (this term will be in quotation marks until we define it more formally, which will happen later on). This first stage makes a lot of sense from a data management perspective, because it means that if we throw a bunch of “regular spanners” at a document, then we actually turn it into a relational database. Using regular language descriptors is a great idea, since they are well understood, they have very nice algorithmic properties, they are still powerful enough to describe relevant computational tasks, and they are so simple that we can even teach them to students in the first year of their studies. Regarding the second stage: Everybody working in data management is able to manipulate relational tables with relational algebra or similar languages like SQL. So this approach is just natural.

2.1 Regular Spanners

Now how can regular language descriptors be used for describing document spanners? Well, just use a finite automaton, e.g., the following one:



This is an automaton over the alphabet $\Sigma = \{a, b, c\}$, but for the variable x it has a special “please start the span for variable x here”-symbol $x▷$, and a special “please let the span for variable x end here, thank you”-symbol $◁x$, and analogous special symbols for variable y (it’s helpful to see them as a pair of parentheses $x▷ \dots ◁x$). Intuitively, it is clear what’s going on: The automaton reads some input over Σ and whenever it takes an $x▷$ -arc, a span for x is created that ends when a $◁x$ -arc is traversed, and similar for y . When exactly such special-arcs are traversed obviously depends on the nondeterminism of the automaton, so the automaton can perform several different accepting runs on a fixed input, which yields several ways of extracting an $\{x, y\}$ -span tuple from the input. As is easy to see, the automaton describes the spanner that, for any document \mathbf{D} , produces the table of all span-tuples $((i, j), (k, l))$, where $j \leq k$ and $\mathbf{D}[i, j] = (ab)^m$ and $\mathbf{D}[k, l] = (cc)^n$ with $m, n \geq 0$. In a similar way, the regular expression $\Sigma^* x▷ (ab)^* ◁x \Sigma^* y▷ (cc)^* ◁y \Sigma^*$ describes the same spanner (note that it describes the same regular language over $\Sigma \cup \{x▷, ◁x, y▷, ◁y\}$ as the automaton).

However, in order to gain a better theoretical understanding of the model, it is somewhat more convenient to refrain from thinking about specific classes of spanner representations for a moment, and establish a quite general way of how document spanners can be described (so any possible functions that maps documents to span-relations, even undecidable ones). The most relevant class of “regular spanners” can then be easily obtained by just saying the word “regular” at the right place.

Spanners are functions with Σ^* as their domain, where Σ^* is the set of all words over the alphabet Σ . Consequently, it makes sense to describe the concept of spanners in a purely language theoretic setting, which is quite convenient.⁴

A word w and an \mathcal{X} -span-tuple t can be merged into a single string by simply marking in w the beginning of the span for x by the symbol \succ_x and the end of the span for x by the symbol \prec_x (and obviously in the same way for all other variables from \mathcal{X}), which is then a word over the alphabet $\Sigma \cup \Gamma_{\mathcal{X}}$, where $\Gamma_{\mathcal{X}} = \{\succ_x, \prec_x \mid x \in \mathcal{X}\}$. So merging the word **banana** and the span-tuple $((2, 4), (4, 6))$ yields **b \succ_x an \succ_y a \prec_x na \prec_y** . Words like this – i.e., words over $\Sigma \cup \Gamma_{\mathcal{X}}$ that encode a word and an \mathcal{X} -span-tuple for this word – are called *subword-marked words* (because that’s just what they are).⁵ In particular, we note that from such a subword-marked word, we can easily get the document it describes (by just deleting the special symbols from $\Gamma_{\mathcal{X}}$), and the span tuple it describes (by just looking up the positions of \succ_x and \prec_x for every $x \in \mathcal{X}$).

The important point is now that every set L of subword-marked words (over \mathcal{X}), which we also call a *subword-marked language*, describes an \mathcal{X} -document spanner. Why? Because for any given document \mathbf{D} , we can simply collect all subword-marked words $w \in L$ that represent \mathbf{D} and put the span-tuple represented by w in a table. So any subword-marked language L (over \mathcal{X}) has a natural interpretation as a function $\llbracket L \rrbracket$ that maps documents to \mathcal{X} -span relations, i.e., a spanner. As a concrete example, consider the subword-marked language $\{b \succ_x an \succ_y a \prec_x na \prec_y, b \succ_x anana \prec_x\}$, which extracts the span-relation $\{((2, 4), (4, 6)), ((2, 6), \perp)\}$ from **banana** (note that \perp means *undefined*), and the empty span relation from any other document. Or the subword-marked language $\{\succ_x b \prec_x u \mid b \in \Sigma, u \in \Sigma^*\}$, which represents a spanner that extracts the first symbol of any document in a span for variable x . Or the subword-marked language $\{u_1 \succ_x (ab)^m \prec_x u_2 \succ_y (cc)^n \prec_y u_3 \mid u_1, u_2, u_3 \in \Sigma^*, m, n \geq 0\}$ which describes the spanner also represented by the automaton from above.

But it also works the other way around. Let S be some \mathcal{X} -document spanner, so some function that maps documents to \mathcal{X} -span relations without any further restriction. Then S maps every given document to a set $\{t_1, t_2, \dots, t_n\}$ of \mathcal{X} -span tuples (note that this set is always finite, since \mathcal{X} is, but for our considerations this is not even important, it’s just more practically sane to have only finitely many variables). So we can simply merge \mathbf{D} with each of the span-tuples t_1, t_2, \dots, t_n to obtain subword-marked words w_1, w_2, \dots, w_n , and if we collect all these subword-marked words that we can obtain like this from every possible document, we have a huge subword-marked language L_S , which describes exactly the spanner S in the way explained in the previous paragraph, i.e., S equals the spanner interpretation $\llbracket L_S \rrbracket$ of L_S .

A Brief Interlude about Subword-marked Words: In the literature on spanners, subword-marked words are usually called *ref-words*. This has histori-

⁴ Maybe a bit confusingly, but justified by the fact that we are now in the realm of formal languages, we use the term *word* instead of *document* for a short while.

⁵ Obviously, we have to formalise when exactly a word over $\Sigma \cup \Gamma_{\mathcal{X}}$ is a proper subword-marked word, but this is not difficult.

cal reasons: Ref-words have originally been used in [25] (in the context of regular expressions with backreferences) as words that contain *references* \times to some of their subwords, which are explicitly marked by brackets $\times \triangleright \dots \triangleleft \times$. So these ref-words from [25] are strings in which subwords can be marked, but they can also contain references to marked substrings, represented by variable symbols in the string. The literature has adapted this technical tool for formalising document spanners, but for this, we only need the “subword-marking”-property, not the “subword-referencing”-property. However, the term “ref-word” has been used anyway and it stuck. Using the terms subword-marked words and ref-words synonymously is fine, as long as we only want to represent markings of subwords and no references. But in [26] – which we shall discuss in more detail below – it has been shown that the ref-words in the sense of [25] (so not only with marked subwords but also with reference-symbols) can be used for formalising a certain class of spanners. Hence, we need to distinguish between the ref-words that only mark subwords (and we call them subword-marked words here) on the one hand, and the ref-words from [25] that can also contain references to marked subwords. End of the interlude.

So we saw that subword-marked languages (over \mathcal{X}) and \mathcal{X} -document spanners are the same thing.⁶ In particular, we can now conveniently define certain classes of spanners by simply stating the underlying class of subword-marked languages. Like this: The *regular spanners* are exactly the spanners $\llbracket L \rrbracket$, where L is a regular subword-marked language. The reader is encouraged to try it for herself by replacing “regular” with her favourite language class (literally any language class! No judging!).

Obviously, for the applicability of a spanner class, the algorithmic properties of the underlying class of subword-marked languages is important. So for representing regular spanners, we can use automata or regular expressions or just anything that describes regular languages (note that we can always filter out strings over $\Sigma \cup \Gamma_{\mathcal{X}}$ that are not valid subword-marked words by intersection with a regular language (note that for doing this, we do not have to parse a well-formed parenthesised expression, we only have to check that if $\times \triangleright$ occurs, then it is followed by one occurrence of $\triangleleft \times$, and that this happens at most once)).

⁶ Note that this is not a one-to-one correspondence. While every subword-marked language L uniquely describes the spanner $\llbracket L \rrbracket$, there are in general several ways of representing a spanner by a subword-marked language. This is due to the fact that two subword marked words like $\mathbf{a} \times \triangleright \text{ } \triangleright \mathbf{b} \triangleleft \times \text{ } \triangleleft \times \mathbf{c}$ and $\mathbf{a} \triangleright \times \triangleright \mathbf{b} \triangleleft \times \triangleleft \times \mathbf{c}$ are different strings, but they nevertheless describe the same pair of document and span-tuple. Unfortunately, this can be very annoying from a technical point of view, and it can even lead to some problems for algorithms on spanners. But since there is no peer-reviewing for this article, we keep quiet and simply pretend that we did not notice this little flaw. This issue is anyway much discussed in the actual literature on document spanners and everybody is aware of it, we just neglect it in this survey because nobody stops us. For example, in order solve this issue, [7] introduces a fixed order on the consecutive occurrences of the symbols from $\Gamma_{\mathcal{X}}$, while [2, 27] simply replace sequences of symbols from $\Gamma_{\mathcal{X}}$ by sets of the symbols, thus using subsets of $\Gamma_{\mathcal{X}}$ as symbols. This was a long footnote, but better than having another “interlude”.

Coming back to the example automaton above: If interpreted as an NFA over $\Sigma \cup \Gamma_{\mathcal{X}}$, it obviously represents a regular subword-marked language, which is the same language represented by the regular expression $\Sigma^* \succ (\mathbf{ab})^* \prec^{\mathcal{X}} \Sigma^* \succ (\mathbf{cc})^* \prec^{\mathcal{Y}} \Sigma^*$, and it is easy to see that if we interpret this subword-marked language as a spanner, it is exactly the one described above, so all span-tuples $((i, j), (k, l))$, where $j \leq k$ and $\mathbf{D}[i, j] = (\mathbf{ab})^m$ and $\mathbf{D}[k, l] = (\mathbf{cc})^n$ with $m, n \geq 0$.

Whether we interpret the \succ and $\prec^{\mathcal{X}}$ transitions as special operations that trigger the construction of the span tuple, or whether we consider them as normal input symbols so that the automaton is a string-acceptor is merely a matter of taste. Although the second point of view seems to fit better to this general perspective of spanners as subword-marked languages.

Note that [9] also considers a proper subclass of regular spanners defined by so-called *regex formulas*, which is a certain class of regular expressions. The point is that regex formulas can enclose only proper sub-expressions in brackets \succ and $\prec^{\mathcal{X}}$. As a result, this formalism cannot describe overlapping spans.

2.2 Core-Spanners

Recall that the regular spanners describe just the first step of the original spanner framework from [9]. So let's move on to the second stage.

Assume that we have extracted from a string a span relation or several span relations by regular spanners. We could now manipulate these tables with relational algebra operations, and in [9], we use union, natural join, projection and – let us make a dramatic pause here, because this operation is a real game changer – *string equality selection*. Union is just the set union of span relations, natural join sort of glues together tables on their common attributes (if you know how natural join is defined, you are probably annoyed by this superficial explanation and would be bored by a detailed one, if you don't know it, you can google it) and projection just deletes columns. Now these are typical operations for relational data and they are not specific to our framework of information extraction of *textual data*. The string equality selection, on the other hand, is tailored to the situation that our tables are not just any tables, but span relations, so their entries are pointers to substrings of a document. The string equality selection is an operator that is parameterised by some subset $\mathcal{Y} \subseteq \mathcal{X}$. It looks at every span tuple of the span relation and checks whether all the spans of the variables in \mathcal{Y} point to an occurrence of the *same* substring. Although this is obvious, let us briefly observe that different spans might represent occurrences of the same substring, like (2, 4) and (4, 6) both represents the **ana** in **banana**. Every span tuple where this is *not* the case will be kicked out by this operator, so it selects span tuples from a span relation according to the equality of the substrings of certain spans. The next example shows what the string equality selector does with respect to $\mathcal{Y} = \{x, y\}$ on the following table that has been extracted from **banana**:

x	y
(1, 2)	(4, 6)
(2, 4)	(4, 6)
(3, 4)	(5, 6)
(2, 3)	(5, 6)

 \Rightarrow

x	y
(2, 4)	(4, 6)
(3, 4)	(5, 6)

The so-called *core spanners* are those spanners that can be obtained by first extracting a span relation from a document by a regular spanner, and then apply a finite sequence of any of the relational operators from above (including the string equality selection). As shown in [9], the operators of union, natural join and projection (but not string equality selection!) can all directly be pushed into the automaton for the regular spanner, meaning that tables extracted by a regular spanner followed by any sequence of these operators can also directly be extracted by a single regular spanner. Or, putting it differently, these simpler relational operators are “regular”. As a consequence, core spanners have a normal form: Every core spanner can be described by a regular spanner followed by a finite sequence of string equality selections followed by one projection.

Why the string equality selection makes such a huge difference (i.e., why core spanners are much more powerful than regular spanners) will be discussed in the next section. From an intuitive point of view, this is not surprising, since string equality selection is an inherently non-regular feature. For example, we can use a regular spanner over $\mathcal{X} = \{x, y\}$ that extracts from a document \mathbf{D} all span tuples $((1, k), (k + 1, |\mathbf{D}|))$ for every $k \in \{1, 2, \dots, |\mathbf{D}|\}$ (so it can arbitrarily split the document and store the two parts in the spans of the two variables), and then uses a string equality selection with respect to \mathcal{X} . This will turn every given document \mathbf{D} into the span-relation $\{((1, |\mathbf{D}|/2), (|\mathbf{D}|/2 + 1, |\mathbf{D}|))\}$ if $\mathbf{D} = ww$ (i.e., \mathbf{D} is a square) and into the span-relation \emptyset if \mathbf{D} is not a square. So it somehow recognises the non-regular copy language. But this is nothing! We can also get crazy and apply string equality selections to several spans that overlap each other in complicated ways to describe spanners that are not funny anymore (see [12] for further details).

The majority of the papers on document spanners is concerned with regular spanners, probably because core spanners have some issues with complexity and decidability. However, there are also several papers concerned with core spanners; see [11, 12, 14, 23, 26].

3 Problems on Regular Spanners and Core Spanners

Regular spanners are mild and core spanners are wild. Putting it more formally, regular spanners have excellent algorithmic properties (i.e., good complexities), while core spanners exhibit intractability and even undecidability for many of their relevant computational problems. As an example, let us consider some relevant problems like model checking (deciding whether a given span tuple t is in $S(\mathbf{D})$ for a given spanner S and document \mathbf{D}), non-emptiness (check whether $S(\mathbf{D}) \neq \emptyset$ for a given spanner S and document \mathbf{D}), satisfiability (for

given spanner S , decide whether there is a document \mathbf{D} with $S(\mathbf{D}) \neq \emptyset$, or inclusion (for given spanners S_1 and S_2 , decide whether $S_1(\mathbf{D}) \subseteq S_2(\mathbf{D})$ for every document \mathbf{D}).

For regular spanners, algorithms for model checking, non-emptiness and satisfiability have quite good polynomial running times (see, e.g., [2, 9, 10, 26]). The reason is that these problems reduce to problems on regular languages or finite automata (i.e., the good algorithmic properties of regular languages carry over to regular spanners). Moreover, inclusion for regular spanners is PSPACE-complete, which is not exactly tractable, but inclusion for regular languages is also PSPACE-complete, and the inclusion problem for regular spanners covers the inclusion problem for regular languages (see [19]). For core-spanners, the inclusion problem is even undecidable, and model checking, non-emptiness and satisfiability, which can be solved quite efficiently for regular spanners, are all NP-hard (see [12]).

All the aforementioned problems are typical decision problems, but in database theory, which always has an eye towards application, there is also a substantial interest in practically motivated problems. One key observation is that a computer program that merely says “yes” or “no” to Boolean database queries is of little use in the real world. Moreover, a program that computes the huge set (of potentially exponential size) of all possible answers to the query is also of questionable practical relevance. Therefore, it is common to investigate query evaluation (this term somewhat abstractly covers all scenarios where we want to evaluate a given query with respect to a given database, even if the database is just a single string) in terms of an enumeration problem. This means that we are interested in algorithms that produce a list of all answers to the queries (obviously, without repetitions). Such an algorithm is particularly worthwhile if it starts producing the list very fast, and if we do not have to wait too long to receive the next element. The optimal scenario here is therefore that the first element is produced after a running time that is only linear in the size of the data, which is called *linear preprocessing*. Note that the algorithm must somehow process the data that is queried, so assuming at least preprocessing linear in the size of the data is fair. Moreover, after one answer is produced, we would like the time we have to wait for the next element (which we call *delay*) to be completely independent from the size of the data, so the running time we need here is only a function of the size of the query (which is then called *constant delay*). We should mention here that these complexity requirements use the so-called *data-complexity* perspective, which measures running time only in the size of the data, and considers the size of the query as being constant. This is a quasi-standard in many areas of database theory, and it makes a lot of sense, since the data can be assumed to be quite large, while the query, in comparison, is tiny. The assumption that the data is large is justified by the buzzword “big data”. The queries are assumed to be small since they are – in most scenarios – written by human users. Of course, linear preprocessing and constant delay is not always possible, but it is the holy-grail for enumeration algorithms of query evaluation problems. See [31, 32] for surveys on the topic of enumeration algorithms in database theory, and [30] for a recent paper.

Coming back to document spanners, we are looking for an algorithm that, for some document \mathbf{D} and a spanner S , makes some preprocessing that is linear in $|\mathbf{D}|$ and then enumerates all span tuples from $S(\mathbf{D})$ with constant delay. For regular spanners, this is possible, but it does not directly follow from known algorithmic results about automata (see [2, 10] for details). Let us briefly discuss this on an intuitive level.

Assume that the spanner S is given by an NFA M that accepts a subword-marked language L (over Σ and \mathcal{X}) with $\llbracket L \rrbracket = S$. Now we are interested in all possible ways of shuffling the symbols $\Gamma_{\mathcal{X}} = \{\langle^{x_1}, \langle^{x_1}, x_2, \langle^{x_2}, \dots\}$ into \mathbf{D} such that we get a subword-marked word that is accepted by M , since these subword-marked words represent the span-tuples of $S(\mathbf{D})$. But these subword-marked words are represented by paths in M from the start state to an accepting state that are labelled with \mathbf{D} (and some symbols from $\Gamma_{\mathcal{X}}$). In fact, it is better to consider the DAG of nodes (p, i) , where p is a state of M and i is a position of \mathbf{D} , there is a $\mathbf{D}[i+1]$ -labelled edge from (p, i) to $(q, i+1)$ if in M we can change from p to q by reading $\mathbf{D}[i+1]$, and there is a γ -labelled edge from (p, i) to (q, i) if in M we can change from p to q by reading some symbol $\gamma \in \Gamma_{\mathcal{X}}$ (actually, we could even drop the edge labels from Σ , since they are not relevant). In this DAG, we are interested in all paths from $(q_0, 0)$, where q_0 is the start state, to some $(q_f, |\mathbf{D}|)$, where q_f is some accepting state. The labels from $\Gamma_{\mathcal{X}}$ on such paths represents the span-tuples that we enumerate. So we simply enumerate those paths, but this is a bit tricky because, firstly, there might be different paths representing the same span-tuple and, secondly, the paths are rather long (well, of size $|\mathbf{D}|$ actually), so we cannot afford to construct them explicitly, because this would break our bound on the delay. In other words, we have to enumerate those paths, but we have to efficiently skip over the parts of the paths that are labelled with symbols from \mathbf{D} .

4 An Approach to Tame Core Spanners

We called core spanners *wild* earlier in this article, because with this term in mind, it is appropriate to think about how we can *tame* them. And *taming* now means to make them a bit more like regular spanners in terms of their algorithmic properties, while still maintaining the most important features of their expressive power. An approach towards this goal has been presented in [26], which we shall now briefly explain.

A nice property of regular spanners is that we can purely describe them as special regular languages (i.e., regular subword marked languages), which means that we can use classical tools like regular expressions and finite automata to handle them. So it is worth thinking about to what extent we can use the same approach also for (subclasses of) core spanners. In particular, the goal is to describe core-spanners again just by certain regular languages (obviously, the subword-marked languages are not suitable for this, since the subword-marked languages of core-spanners are not necessarily regular languages).

Let L_S be the subword-marked language for a regular spanner S . If we use on S a string equality selection (i.e., we consider a core-spanner), say with respect to variables $\mathcal{Y} = \{y, z\}$, then any subword-marked word of L_S can be considered

irrelevant, if $\succ \dots \triangleleft^y$ and $\succ \dots \triangleleft^z$ enclose different factors. So with the application of the string equality selection in mind, we wish to directly get rid of such irrelevant words of L_S , and we achieve this by replacing every $\succ u \triangleleft^z$ factor of a subword-marked word by $\succ y \triangleleft^z$, where the symbol y is used as a *reference* signifying that the word enclosed by $\succ \dots \triangleleft^y$ is to be repeated here (otherwise, the subword-marked word would describe a span-tuple that would be killed by the string equality relation anyway).

As an example, consider the subword-marked language described by the regular expression

$$r := \succ a^* b \succ c \triangleleft^x b^* x' \triangleleft a^* bc \triangleleft^{x'} \triangleleft^y y' \triangleleft cb^* a^* bc \triangleleft^{y'}$$

and assume that we want to apply a string equality selection with respect to $\{x, x'\}$ followed by a string equality selection with respect to $\{y, y'\}$. The resulting core-spanner can as well be represented by the regular expression $r' := \succ a^* b \succ c \triangleleft^x b^* x' \triangleleft x \triangleleft^{x'} \triangleleft^y y' \triangleleft y \triangleleft^{y'}$, i.e., we simply replace $\triangleleft^x a^* bc \triangleleft^{x'}$ and $\triangleleft^y cb^* a^* bc \triangleleft^{y'}$ by $\triangleleft^x x \triangleleft^{x'}$ and $\triangleleft^y y \triangleleft^{y'}$, respectively. Now, we have represented a non-regular core-spanner “somehow” as a regular language (note that with help of the string equality selection, this core spanner checks whether some unbounded factors are repeated, which is an inherently non-regular property). But how exactly does the regular language describe the core spanner? Very easy: The regular expression r' can generate words like $\succ aab \succ c \triangleleft^x b x' \triangleleft x \triangleleft^{x'} \triangleleft^y y' \triangleleft y \triangleleft^{y'}$ and $\succ aaaab \succ c \triangleleft^x bbb \triangleleft x \triangleleft^{x'} \triangleleft^y y' \triangleleft y \triangleleft^{y'}$ and so on. So these words are almost subword-marked words, but they have occurrences of *references* x and y , which we need to replace. Hence, we interpret each such word as the subword-marked word that we get by simply replacing all the references. Applied to $\succ aab \succ c \triangleleft^x b x' \triangleleft x \triangleleft^{x'} \triangleleft^y y' \triangleleft y \triangleleft^{y'}$, the replacement $x \mapsto aabc$ gives us $\succ aab \succ c \triangleleft^x b x' \triangleleft \underline{aabc} \triangleleft^{x'} \triangleleft^y y' \triangleleft y \triangleleft^{y'}$, and then the replacement $y \mapsto cbaabc$ gives us the subword-marked word $\succ aab \succ c \triangleleft^x b x' \triangleleft \underline{aabc} \triangleleft^{x'} \triangleleft^y \triangleleft \underline{cbaabc} \triangleleft^{y'}$. And this final subword-marked word describes a document and a span-tuple in the usual way.

So what we do is that we also allow the variables from \mathcal{X} to occur in subword-marked words, and we call such words then *ref-words* (since these variables function a references to marked subwords). Then we consider *regular* languages of ref-words, and we interpret such ref-languages as spanners by first resolving all the references in all the ref-words (as sketched above) to get a subword-marked languages, which then describes a spanner as before. While the ref-language is necessarily regular (by definition), the subword-marked language that we get from it by resolving all references is not necessarily a regular language anymore (non-regularity must creep in somewhere, if we want to describe non-regular core-spanners).

Those so-called *refl-spanners* can still describe all regular spanners, but also many non-regular core-spanners.⁷

⁷ Technically, the formalism also allows to create an unbounded number of references by using a variable symbol under a Kleene-star, which results in spanners that are not even core-spanners. But since the formalism is introduced for describing a large class of core-spanners by regular languages, we ignore this issue.

Model-checking and satisfiability for refl-spanners can be solved as efficiently as for regular spanners (while for core-spanners these problems are intractable). On the other hand, non-emptiness for refl-spanners is NP-hard as for core spanners. Moreover, refl-spanner allow a certain restriction that yields decidable inclusion (recall that inclusion is undecidable for core-spanners).

It is intuitively clear that refl-spanners cannot describe all core-spanners. Note that the construction sketched above where we replaced parts of the regular spanner by references only works in special cases.

For example, let s, t and r be some regular expressions, and consider the regular spanner described by $\succ r \prec^x \succ s \prec^y \succ t \prec^z$. If we want to describe the core-spanner that we get by applying a string equality selection with respect to $\{x, y, z\}$, then we cannot simply replace each of s and t by an occurrence of x , since this would also represent ref-words $\succ u \prec^x \succ x \prec^y \succ x \prec^z$, where $u \in L(r)$, but $u \notin L(s)$ or $u \notin L(t)$, which gives us the subword-marked word $\succ u \prec^x \succ u \prec^y \succ u \prec^z$, which describes a span-tuple for the document uuu that should not be in the span-relation. However, in this case, we can use the refl-spanner $\succ h \prec^x \succ x \prec^y \succ x \prec^z$, where h is a regular expression that describes the intersection of $L(r)$, $L(s)$ and $L(t)$ (note that this construction causes an exponential blow-up).

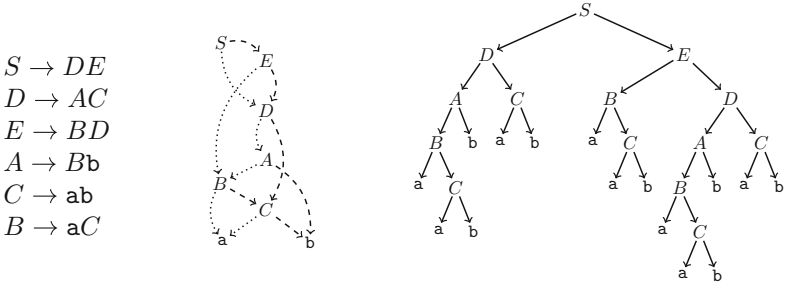
Another problem arises when we want to replace some $\succ s \prec^x$ by $\succ y \prec^x$, but s also contains symbols from $\Gamma_{\mathcal{X}}$, which we cannot afford to simply delete. Interestingly, in such cases it can help to first cut all the overlapping regions enclosed by the brackets $\succ \dots \prec^x$ into smaller parts (thereby introducing new variables, but only polynomially many), then, depending on the string equality selection, translating it into a ref-language, which then not quite describes the intended core spanner, but almost: It describes the core-spanner with the only difference that some spans are cut into a several smaller spans. This difference can then easily be repaired by just combining certain columns into one column and gluing the respective spans of those columns together. This result points out that refl-spanner can describe a large class of core-spanners.

5 Regular Spanners on SLP-Compressed Data

Since big data is so big, it is a good idea to compress it. The classical motivation is that compressed data can be stored in less space, or send somewhere in less time. But what about directly querying data in compressed form, i.e., without decompressing it? This would be very convenient, since then we do not have to decompress our data when we want to work with it, and algorithms for querying the data might even be faster, since their running times depend on the size of the compressed data, which might be much smaller than the uncompressed data size. Theoretically, any polynomial time algorithm that works on compressed data may outperform even a linear time algorithm for the uncompressed data, in the special case where the compressed data has size logarithmic in the uncompressed data (so measured in the uncompressed data it's polylogarithmic running time vs. linear running time).

This paradigm of *algorithmics on compressed inputs* is well-developed in the realm of string algorithms (see the explanations in [27, 29] and the general survey [18]), and since spanner evaluation is a string problem, it makes sense to investigate it in this compressed setting as well. More precisely, we are interested in evaluating a document spanner over a document that is compressed, and which should not be decompressed for this purpose. Moreover, the most relevant form of evaluation problem is enumeration (as explained above), and the best running time is linear preprocessing (but now linear with respect to the compressed size of the document!) and constant delay.

Before outlining some respective results, let us discuss the underlying compression scheme in this setting. A particularly fruitful approach to algorithmics on compressed strings are so-called *straight-line programs* (SLPs). The simplicity of SLPs is very appealing: We compress a string w by a context-free grammar (for convenience in Chomsky normal form) that describes the language $\{w\}$ (i.e., it can generate exactly one string, which happens to be w). This calls for an example: The string `aabbabaabaabbab` can be described by a context-free grammar with the rules shown here on the left (we use S, A, B, C, D, E as non-terminals, where S is the start non-terminal):



Above in the middle, it is demonstrated that we can interpret each SLP as a DAG in which each rule $A \rightarrow BC$ is represented by a node A with *left successor* B (indicated by dotted arrows) and *right successor* C (indicated by dashed arrows). Note that this is due to the fact that we assume the Chomsky normal form (without the Chomsky normal form, the outdegree of the nodes would be larger and we would need some way of expressing an order on the outgoing edges). The only possible derivation in this SLP (i.e., starting with S and then just applying the rules until we have a string over $\{a, b\}$) yields the compressed string `aabbabaabaabbab`. As is typical for derivations in context-free grammars, we can also consider the derivation tree, which is displayed above on the right (note, however, that the derivation tree is not a compressed representation, since it is at least as large as the string).

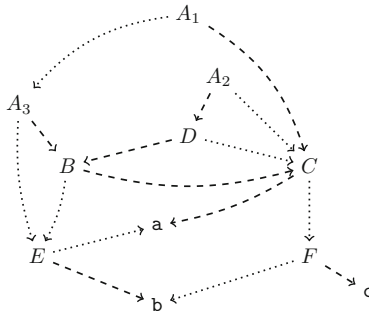
For the size of an SLP, we can take its number of rules (obviously, this ignores a factor of 3, but that's okay if we measure asymptotically). In order to see how the compression works, take a look at the derivation tree (recall that the derivation tree is an uncompressed representation): Here, we have to explicitly spell out each application of a rule in the construction of the string, while the actual SLP representation mentions each rule exactly once. Intuitively speaking,

an SLP just tells us how to replace repeating substrings by variables in a clever way (and this is hierarchical in the sense that substrings already containing variables are again replaced by variables and so on). Obviously, this should be called clever only if it achieves a decent compression.

An SLP might be much smaller than the string it represents; in fact, even exponentially smaller (to see this, just consider an SLP which just doubles a string in every rule). There are also strings that are not well compressible, but experimental analyses have shown that the compression achieved by SLPs on natural inputs is very good. On an intuitive level, this is not surprising: Texts in natural language have many repeating words, and the same syllables occur over and over again in different words. But also for any artificial string over a finite alphabet (e.g., the alphabet $\{A, G, C, T\}$ of DNA nucleobases), we will get repeating substrings if the string is long enough.

The success of SLP-compression for strings is due to the fact that SLPs achieve good compression (already mentioned above), but also that there are very fast approximation algorithms that achieve these good compression ratios.⁸ And, most importantly, there are many algorithms capable of solving basic, but important string problems directly on compressed strings (see [18] for an overview). As an example, let us recall that pattern matching (i.e., finding a given string P as substring in another string T) can be solved in time $O(|P| + |T|)$ by classical algorithms (the well-known Knuth-Morris-Pratt algorithm for example). But if T is given by an SLP S , so in SLP-compressed form, we can still solve pattern matching in linear time, but without having to decompress S , i.e., in time $O(|P| + |S|)$ (note that if $|S| \ll |T|$, then this directly translates into a faster running time); this has recently been shown in [16]. Although this is not always trivial to show, it often turns out that certain string problems can still be solved efficiently, even if we get the input string in form of an SLP. And it is the same for *regular* spanner evaluation (investigated in [20, 27, 29]).

For spanners (recall that we are in a data management setting), we assume that we have a whole database of documents, which is just a set of documents, e.g., $\mathcal{D} = \{\mathbf{D}_1, \mathbf{D}_2, \mathbf{D}_3\} = \{\text{ababbcabca}, \text{bcabcaabbca}, \text{ababbca}\}$. Such document databases can also be compressed by an SLP, for example, this one:



⁸ Full disclosure: Computing a minimal SLP for a string is NP-complete [5]. But this is no problem at all, due to these fast practical approximate compressors.

Recall that dotted arrows point to the left successors and dashed arrows point to the right successors. Here, the non-terminals A_1 , A_2 and A_3 derive exactly the documents \mathbf{D}_1 , \mathbf{D}_2 and \mathbf{D}_3 of the example document database \mathcal{D} mentioned above (the reader is welcome to verify this, although it is a bit painful).

We now get some regular spanner S and some index $i \in \{1, 2, \dots, |\mathcal{D}|\}$ (indicating the document that we want to query), and after a preprocessing linear in the size of \mathcal{D} , we want to be able to enumerate the elements of $S(\mathbf{D}_i)$. This is in fact possible, but with a delay of $\log(|\mathbf{D}_i|)$. The delay is therefore not constant and depends on the size of the *uncompressed* data, but only logarithmically. An important fact is that this delay is always at most logarithmic in $|\mathbf{D}_i|$ independent of the actual compression achieved by the SLP. In order to understand where this log-factor comes from, let us sketch the general approach of this algorithm (see [27] for details).

In the uncompressed setting, the enumeration of regular spanners relies on enumerating all subword-marked versions of the document that are accepted by the automaton that represents the spanner. So in the compressed setting, we have to enumerate all subword-marked versions of the derivation tree for \mathbf{D}_i that describe subword-marked documents accepted by the automaton. A subword-marked version of \mathbf{D}_i 's derivation tree is obtained by placing the symbols from $\Gamma_{\mathcal{X}}$ at their corresponding places into the derivation tree of \mathbf{D}_i . Of course, we cannot just naively construct \mathbf{D}_i 's derivation tree, because it is too big, but it is enough to sufficiently expand only those branches of the derivation tree that reveal positions of \mathbf{D}_i where symbols from $\Gamma_{\mathcal{X}}$ have to be placed. This means that we have to expand only $O(|\mathcal{X}|)$ branches of the derivation tree (recall that, due to the data-complexity perspective, $|\mathcal{X}|$ is a constant for us), and we do not have to expand branches completely, just a long path leading to the desired position of the document. However, each of these paths may lead to a position that is buried deep, deep down in \mathbf{D}_i 's derivation tree. So these branches leading to the positions that must be marked with symbols from $\Gamma_{\mathcal{X}}$ may be long. To clarify this, take a look again at our first example of an SLP from above (Page 13). The fourth letter **b** of the compressed document can be reached from S by a path of length 3, i.e., the path S, D, A, \mathbf{b} , while the 11th letter **a** needs a path of length 6. Hence, the cost of producing the next subword-marked variant of the (still partially compressed) derivation tree, can be rather high, when we have to mark symbols deep down in the derivation tree. In fact, the length of such a path can be $\Omega(|\mathbf{D}_i|)$, which is bad.

We solve this, by first *balancing* our SLP, where an SLP is balanced if the longest path from a non-terminal to a leaf is bounded logarithmically in the size of the string derived by that non-terminal. In particular, this means that all paths of \mathbf{D}_i 's derivation tree would be bounded by $\log(|\mathbf{D}_i|)$. So if the SLP is balanced, then constructing these subword-marked and partially decompressed variants of \mathbf{D}_i 's derivation tree can be done in time $\log(|\mathbf{D}_i|)$ which explains our logarithmic delay. But what if the SLP is not balanced? In this case, we can just balance it, which is possible in linear time (see [17]).

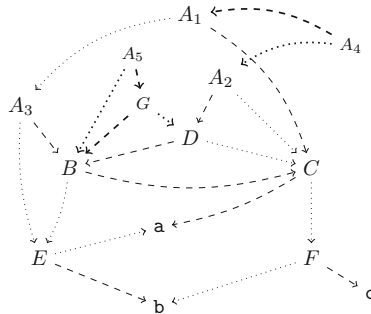
There is also a different approach to regular spanner evaluation over SLP-compressed documents that improves the logarithmic delay to constant delay (see [20]).

5.1 Updates

In addition to the enumeration perspective, another practically motivated perspective of query evaluation problems is the so-called *dynamic case*. This is based on the observation that in practice we usually query a fixed database that is updated from time to time. Hence, the same queries are evaluated over just slightly different versions of the same database (i.e., a few tuples are added, a few tuples are deleted, but it is relatively safe to assume that the database that we query today is quite similar to the database that we query tomorrow). Consequently, it would be nice if for a query q and a database D , it is enough to run the preprocessing for q and D only once (this preprocessing provides us with the necessary data structures to evaluate q on D efficiently), and whenever we update the database, we also update directly what we have computed in the preprocessing without repeating the complete preprocessing. Obviously, since an update changes just a tiny bit of our data, the work to be done after an update should be much less in comparison to re-running the complete preprocessing.

Document spanner evaluation in the SLP-compressed setting is particularly well-suited for this dynamic setting. Let us sketch why this is the case (see [29] for details).

In an uncompressed setting, we perform an update to the database (like adding or deleting some data element), which is easy, and then we have to take care of how to maintain the preprocessing data structures under the updates. In a compressed scenario, on the other hand, we also have to update the compressed representation of our data. And, needless to say, we do not want to completely decompress the data, make the update and then compress it again. But, fortunately, an SLP that represents a document database is somehow suitable for such updates. Take a look at the example SLP from above for the document database $\mathcal{D} = \{\mathbf{D}_1, \mathbf{D}_2, \mathbf{D}_3\} = \{\text{ababbcabca}, \text{bcabcaabbca}, \text{ababbca}\}$ (actually, take a look below, where we repeat this SLP, but now with some updates).



In this SLP, non-terminals A_1 and A_2 represent document \mathbf{D}_1 and \mathbf{D}_2 . So by adding a new non-terminal A_4 with rule $A_4 \rightarrow A_2A_1$ (as shown above), we automatically add the new document $\mathbf{D}_2\mathbf{D}_1 = \text{bcabcaabbcaabbbcabca}$ to our document database. Adding the non-terminals A_5 and G with rules $A_5 \rightarrow BG$ and $G \rightarrow DB$ adds the more complicated document $\mathbf{D}_B\mathbf{D}_D\mathbf{D}_B = \text{abbcabcaabbcaabbca}$, where $\mathbf{D}_B = \text{abbca}$ is the document represented by non-terminal B and $\mathbf{D}_D = \text{bcaabbca}$ is the document represented by non-terminal D . So as long as we want to add documents that can be pieced together from strings already represented by non-terminals, this is simply done by adding a few more non-terminals and rules. By slightly more complicated operations, we can also cut some already present document into two parts (meaning that we construct non-terminals that derive exactly the left and right part), and with such an operation, we can realise operations like copying factors and inserting them somewhere else etc. In summary, updates that consist in adding new documents that can be created from existing documents by a sequence of copy-and-paste-like operations (note that this is how we usually work with text documents) can be easily done for SLP-compressed document databases. But how much effort is this really?

On close inspection, we can see that for implementing any of these copy-and-paste-like operations, it is sufficient to manipulate a constant number of paths in the SLP. But how long is a path? Well, if the SLP is balanced, it is not so long, i.e., logarithmic in the (uncompressed) size of the represented document database. So updates can be performed in time logarithmic in the size of the uncompressed data (multiplied by the number of individual copy-and-paste-operations, should our desired update require several of those). The only catch is that our updates may destroy the balancedness property, so we should implement them in such a way that balancedness is maintained. This is possible, at least if we require a stronger balancedness property. The weaker balancedness property that we used for the enumeration algorithm only requires that every path that starts in a non-terminal A is logarithmically bounded in the size of the string represented by A , the stronger variant requires for every rule $A \rightarrow BC$ that the longest path from B and the longest path from C differ by at most one. The advantage of the weaker property is that we can transform any SLP in an equivalent weakly balanced one of asymptotically the same size in linear time, whereas the stronger property requires time $m \log(n)$, where m is the size of the SLP and n is the (uncompressed) size of the represented document database. But the stronger property can be maintained by our updates, so that we can guarantee that all our updates only cost time logarithmic in the size of the uncompressed data.

Nice, but this only updates the SLP. However, the data structures that we have computed in the preprocessing and that we need for enumerating the regular document spanner are basically just certain information for every non-terminal of the SLP, and we can easily update those while performing our updates on the SLP. Hence, as long as we have a strongly balanced SLP (which we can always get by paying with a logarithmic factor), we can make a preprocessing linear in

the size of the compressed data (which, potentially, is logarithmic in the actual data), then enumerate a regular spanner with delay that is guaranteed logarithmic in the data, then perform updates in time logarithmic in the data, then again enumerate the spanner with delay logarithmic in the data (but without re-running the preprocessing) and so forth. We only have to spend again preprocessing time linear in the compressed data if we want to evaluate a completely new query. Thus, we could also initially preprocess a finite set of document spanners (this costs time linear in the compressed data multiplied by the number of spanners) and then we can always enumerate any of these spanners with a delay logarithmic in the (current) data, which we can manipulate with updates that cost us time logarithmic in the (current) data.

References

1. Amarilli, A., Bourhis, P., Mengel, S., Niewerth, M.: Constant-delay enumeration for nondeterministic document spanners. *SIGMOD Rec.* **49**(1), 25–32 (2020). <https://doi.org/10.1145/3422648.3422655>
2. Amarilli, A., Bourhis, P., Mengel, S., Niewerth, M.: Constant-delay enumeration for nondeterministic document spanners. *ACM Trans. Database Syst.* **46**(1), 2:1–2:30 (2021). <https://doi.org/10.1145/3436487>
3. Amarilli, A., Jachiet, L., Muñoz, M., Riveros, C.: Efficient enumeration for annotated grammars. In: *PODS '22: International Conference on Management of Data*, Philadelphia, PA, USA, 12–17 June 2022, pp. 291–300 (2022). <https://doi.org/10.1145/3517804.3526232>
4. Bourhis, P., Grez, A., Jachiet, L., Riveros, C.: Ranked enumeration of MSO logic on words. In: *24th International Conference on Database Theory, ICDT 2021*, 23–26 March 2021, Nicosia, Cyprus, pp. 20:1–20:19 (2021). <https://doi.org/10.4230/LIPICS.ICDT.2021.20>
5. Casel, K., Fernau, H., Gaspers, S., Gras, B., Schmid, M.L.: On the complexity of the smallest grammar problem over fixed alphabets. *Theory Comput. Syst.* **65**(2), 344–409 (2021). <https://doi.org/10.1007/S00224-020-10013-W>
6. Doleschal, J., Bratman, N., Kimelfeld, B., Martens, W.: The complexity of aggregates over extractions by regular expressions. In: *24th International Conference on Database Theory, ICDT 2021*, 23–26 March 2021, Nicosia, Cyprus, pp. 10:1–10:20 (2021). <https://doi.org/10.4230/LIPICS.ICDT.2021.10>
7. Doleschal, J., Kimelfeld, B., Martens, W., Nahshon, Y., Neven, F.: 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 30–5 July 2019, pp. 149–163 (2019). <https://doi.org/10.1145/3294052.3319684>
8. Doleschal, J., Kimelfeld, B., Martens, W., Peterfreund, L.: Weight annotation in information extraction. In: *23rd International Conference on Database Theory, ICDT 2020*, March 30–2 April 2020, Copenhagen, Denmark, pp. 8:1–8:18 (2020). <https://doi.org/10.4230/LIPICS.ICDT.2020.8>
9. Fagin, R., Kimelfeld, B., Reiss, F., Vansummeren, S.: Document spanners: a formal approach to information extraction. *J. ACM* **62**(2), 12:1–12:51 (2015)
10. Florenzano, F., Riveros, C., Ugarte, M., Vansummeren, S., Vrgoc, D.: Efficient enumeration algorithms for regular document spanners. *ACM Trans. Database Syst.* **45**(1), 3:1–3:42 (2020). <https://doi.org/10.1145/3351451>

11. Freydenberger, D.: A logic for document spanners. *Theory Comput. Syst.* **63**(7), 1679–1754 (2019). <https://doi.org/10.1007/s00224-018-9874-1>
12. Freydenberger, D., Holldack, M.: Document spanners: from expressive power to decision problems. *Theory Comput. Syst.* **62**(4), 854–898 (2018)
13. Freydenberger, D.D., Kimelfeld, B., Peterfreund, L.: Joining extractions of regular expressions. In: *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, Houston, TX, USA, 10–15 June 2018, pp. 137–149 (2018). <https://doi.org/10.1145/3196959.3196967>
14. Freydenberger, D.D., Thompson, S.M.: Dynamic complexity of document spanners. In: *23rd International Conference on Database Theory, ICDT 2020*, March 30–2 April 2020, Copenhagen, Denmark, pp. 11:1–11:21 (2020). <https://doi.org/10.4230/LIPICs.ICDT.2020.11>
15. Freydenberger, D.D., Thompson, S.M.: Splitting spanner atoms: a tool for acyclic core spanners. In: *25th International Conference on Database Theory, ICDT 2022*, March 29 to 1 April 2022, Edinburgh, UK (Virtual Conference), pp. 10:1–10:18 (2022). <https://doi.org/10.4230/LIPICs.ICDT.2022.10>
16. Ganardi, M., Gawrychowski, P.: Pattern matching on grammar-compressed strings in linear time. In: *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference/Alexandria, VA, USA, 9–12 January 2022*, pp. 2833–2846 (2022). <https://doi.org/10.1137/1.9781611977073.110>
17. Ganardi, M., Jez, A., Lohrey, M.: Balancing straight-line programs. *J. ACM* **68**(4), 27:1–27:40 (2021). <https://doi.org/10.1145/3457389>
18. Lohrey, M.: Algorithmics on SLP-compressed strings: a survey. *Groups Complex. Cryptol.* **4**(2), 241–299 (2012). <https://doi.org/10.1515/gcc-2012-0016>
19. Maturana, F., Riveros, C., Vrgoc, D.: 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, 10–15 June 2018, pp. 125–136 (2018). <https://doi.org/10.1145/3196959.3196968>
20. Muñoz, M., Riveros, C.: Constant-delay enumeration for SLP-compressed documents. In: *26th International Conference on Database Theory, ICDT 2023*, 28–31 March 2023, Ioannina, Greece, pp. 7:1–7:17 (2023). <https://doi.org/10.4230/LIPICs.ICDT.2023.7>
21. Peterfreund, L.: *The Complexity of Relational Queries over Extractions from Text*. Ph.D. thesis (2019)
22. Peterfreund, L.: Grammars for document spanners. In: *24th International Conference on Database Theory, ICDT 2021*, 23–26 March 2021, Nicosia, Cyprus, pp. 7:1–7:18 (2021). <https://doi.org/10.4230/LIPICs.ICDT.2021.7>
23. Peterfreund, L., ten Cate, B., Fagin, R., Kimelfeld, B.: Recursive programs for document spanners. In: *22nd International Conference on Database Theory, ICDT 2019*, 26–28 March 2019, Lisbon, Portugal, pp. 13:1–13:18 (2019)
24. Peterfreund, L., Freydenberger, D.D., Kimelfeld, B., Kröll, M.: 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 30–5 July 2019, pp. 320–334 (2019)
25. Schmid, M.L.: Characterising REGEX languages by regular languages equipped with factor-referencing. *Inf. Comput. (I&C)* **249**, 1–17 (2016)
26. Schmid, M.L., Schweikardt, N.: A purely regular approach to non-regular core spanners. In: *24th International Conference on Database Theory, ICDT 2021*, 23–26 March 2021, Nicosia, Cyprus, pp. 4:1–4:19 (2021). <https://doi.org/10.4230/LIPICs.ICDT.2021.4>

27. Schmid, M.L., Schweikardt, N.: 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, 20–25 June 2021, pp. 153–165 (2021). <https://doi.org/10.1145/3452021.3458325>
28. Schmid, M.L., Schweikardt, N.: Document spanners - a brief overview of concepts, results, and recent developments. In: PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, 12–17 June 2022, pp. 139–150 (2022). <https://doi.org/10.1145/3517804.3526069>
29. Schmid, M.L., Schweikardt, N.: Query evaluation over SLP-represented document databases with complex document editing. In: PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, 12–17 June 2022, pp. 79–89 (2022). <https://doi.org/10.1145/3517804.3524158>
30. Schweikardt, N., Segoufin, L., Vigny, A.: Enumeration for FO queries over nowhere dense graphs. *J. ACM* **69**(3), 22:1–22:37 (2022). <https://doi.org/10.1145/3517035>
31. Segoufin, L.: A glimpse on constant delay enumeration (invited talk). In: 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, 5–8 March 2014, Lyon, France, pp. 13–27 (2014). <https://doi.org/10.4230/LIPICS.STACS.2014.13>
32. Segoufin, L.: Constant delay enumeration for conjunctive queries. *SIGMOD Rec.* **44**(1), 10–17 (2015)