A Framework for Extraction and Transformation of Documents

Cristian Riveros
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Markus L. Schmid
Humboldt-Universität zu Berlin, Germany

Nicole Schweikardt
Humboldt-Universität zu Berlin, Germany

Abstract

We present a theoretical framework for the extraction and transformation of text documents as a two-phase process: The first phase uses document spanners to extract information from the input document. The second phase transforms the extracted information into a suitable output.

To support several reasonable extract-transform scenarios, we propose for the first phase an extension of document spanners from span-tuples to so-called multispan-tuples, where variables are mapped to sets of spans instead of only single spans. We focus on multispanners described by regex formulas, and we prove that these have the same desirable properties as standard regular spanners. To formalize the second phase, we consider transformations that map every pair document-tuple, where each tuple comes from the (multi)span-relation extracted in the first phase, into a new output document. The specification of the two phases is what we call an extract-transform (ET) program, which covers practically relevant extract-transform tasks.

In this paper, our main technical goal is to identify a broad class of ET programs that can be evaluated efficiently. We specifically focus on the scenario of regular ET programs: the extraction phase is given by a regex multispanner and the transformation phase is given by a regular string-to-string function. We show that for any regular ET program, given an input document, we can enumerate all final output documents with output-linear delay after linear preprocessing. As a side effect, we characterize the expressive power of regular ET programs and also show that they have desirable properties, like being closed under composition.

2012 ACM Subject Classification Theory of computation → Database theory

Keywords and phrases Information extraction, Document spanners, Transducers, Query evaluation

Digital Object Identifier 10.4230/LIPIcs.ICDT.2025.18

Related Version *Full Version*: `https://arxiv.org/abs/2405.12350`

Funding *Cristian Riveros*: Supported by ANID Fondecyt Regular project 1230935, by ANID –Millennium Science Initiative Program – Code ICN17_002, and by the Alexander von Humboldt Foundation. This work was developed during a research fellowship at the Humboldt University, funded by the Alexander von Humboldt Foundation.
Markus L. Schmid: Supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) – project number 522576760 (gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 522576760).

1 Introduction

Information extraction (IE) of text documents found its theoretical foundations in the framework of document spanners [18]. Introduced by Fagin, Kimelfeld, Reiss, and Vansummeren one decade ago [17], this framework sets the grounds for rule-based IE through the notion of spanners and how to combine them by using relational operators. Moreover,

it inspired a lot of research on languages [21, 34], expressiveness [20, 29, 36, 14, 39], evaluation [19, 8, 14, 11, 22, 35, 24], provenance [15, 13], and compressed evaluation [40, 42, 31] for IE. Initially conceived to understand SystemT – IBM's rule-based IE system – it has found some recent promising implementations in [37]. See [41, 7] for surveys of the area.

Although IE is a crucial task for data management, document extraction is usually followed by a transformation into new data objects. Indeed, data transformation is essential for communicating middleware systems, where data does not conform to the required format of a third-party system and needs to be converted. This forms the main task of the so-called ETL technologies [44] (for Extract-Transform-Load), which are crucial parts of today's data management workflows. Even the search-and-replace operation over text, a ubiquitous task in all text systems, can be conceived as an extraction (i.e., search) followed by a transformation (i.e., replace). Hence, although document spanners provide the formal ground of ruled-based IE, they are incomplete without understanding the subsequent transformation processes.

In this paper, we study the extraction and transformation of text documents through the lens of document spanners. We formulate *extract-transform programs* (*ET programs* for short) as a two-phase process: first, an *extraction phase* extracts some information from the input document, and then a *transformation phase* maps the document and the extracted information into the desired output. Depending on the particular application context, the output may be a relational database, graph data, a single document, or a collection of documents. More specifically, the first phase is based on information extraction with document spanners, i. e., it extracts span-tuples from a document. The second phase is deliberately left underspecified at this point. In principle, any class of functions could be used here, depending on the expressive power and algorithmic properties that we aim for, which, in turn, depend on the specific application context.

Let us now explain and motivate our framework of ET programs with some examples. Consider the following document $\mathbf{D}$ over alphabet Σ that lists famous English-speaking singers in the format # [person] # [person] # ... # [person] # with [person] = [last name]; [first name]; [birthplace]; [opt]; [opt];..., where [opt] are optional attributes like, e. g., age, nickname, honorary titles etc, which follow no fixed scheme. This means that "#" and ";" are used as separators, and for convenience, we will use $\widehat{\Sigma} = \Sigma \setminus \{;, \#\}$.

$\mathbf{D}$ = # `Holiday`; `Billie`; `USA` # `Bush`; `Kate`; `England` # `Young`; `Neil`; `Canada`; 78; `"Godfather of Grunge"` # `King`; `Carole`; `USA`; 81 # `McCartney`; `Paul`; `England`; `Sir`; `CH`; `MBE` # `Mitchell`; `Joni`; `Canada`; `painter` #

As mentioned above, the extraction phase is performed by applying a document spanner to $\mathbf{D}$. For example, consider the document spanner specified by the following *regex formula* (cf. [18]): $E_1 := \Sigma^*\#\mathsf{x}\{\widehat{\Sigma}^*\};\mathsf{y}\{\widehat{\Sigma}^*\};\Sigma^*$ that uses a variable x to extract just any factor over $\widehat{\Sigma}$ that occurs between separators "#" and ";", and a variable y to extract the following factor over $\widehat{\Sigma}$ between the next two occurrences of ";". The construct $\mathsf{x}\{r\}$ creates a *span* $[i, j\rangle$ pointing to factor $\mathbf{D}[i..j-1] := \mathbf{D}[i]\mathbf{D}[i+1]\dots\mathbf{D}[j-1]$, satisfying the subexpression r. The regex formula E_1 specifies a spanner that, on our example document $\mathbf{D}$ produces the set $\{([2, 9\rangle, [10, 16\rangle), ([21, 25\rangle, [26, 30\rangle), ([39, 44\rangle, [45, 49\rangle), \dots\}$ of *span-tuples*. Hence, every span-tuple t can be interpreted as representing a pair of factors extracted from our document, i. e., the span-tuple $([2, 9\rangle, [10, 16\rangle)$ represents (`Holiday`, `Billie`), the span-tuple $([21, 25\rangle, [26, 30\rangle)$ represents (`Bush`, `Kate`) and so on. Consequently, this spanner extracts all the names of the singers of our input document.

As a reasonable transformation task, we now might want to produce lots of small documents, each consisting of the name of a person mentioned in $\mathbf{D}$, but in the format "[first name] [last name]". In the given example, this results in the collection of the fol-

lowing documents: `Billie Holiday`, `Kate Bush`, `Neil Young`, `Carole King`, `Paul McCartney`, `Joni Mitchell`. Formally, this transformation is done by a function that maps each $(\mathbf{D}, t_i)$ with $t_i = ([i_{\mathsf{x}}, j_{\mathsf{x}}\rangle, [i_{\mathsf{y}}, j_{\mathsf{y}}\rangle)$ to the string $\mathbf{D}[i_{\mathsf{y}}..j_{\mathsf{y}}{-}1] \sqcup \mathbf{D}[i_{\mathsf{x}}..j_{\mathsf{x}}{-}1]$. Note that the order of first and last names has to be swapped.

Let us move on to a more complicated example. We want to define an ET program which maps $\mathbf{D}$ to the following collection of XML documents:

⟨opt-attr⟩ ⟨singer⟩`Neil Young`⟨/singer⟩ ⟨opt⟩`78`⟨/opt⟩ ⟨opt⟩`"God..."`⟨/opt⟩ ⟨/opt-attr⟩

⟨opt-attr⟩ ⟨singer⟩`Carole King`⟨/singer⟩ ⟨opt⟩`81`⟨/opt⟩ ⟨/opt-attr⟩

⋮

This means that for every singer with optional attributes we want to construct an XML document that contains the name of the respective singer and all the optional attributes. A natural way to approach the corresponding extraction task is to extract the name and surname in two variables as before, and to extract each element of the unbounded list [opt-1]; [opt-2]; ..., [opt-k] in its own variable. This, however, goes beyond the capability of document spanners, since we would need an unbounded number of different variables.

As a remedy, in this paper we propose to extend the classical spanner framework to *multispanners*, which can extract in each variable a *set of spans* instead of only a single span. For example, assume that we apply to the document the *multispanner regex formula*: $E_2 \;:=\; \Sigma^*\#\,\mathsf{x}\{\widehat{\Sigma}^*\};\mathsf{y}\{\widehat{\Sigma}^*\};\widehat{\Sigma}^*;\mathsf{z}\{\widehat{\Sigma}^*\}(;\mathsf{z}\{\widehat{\Sigma}^*\})^*\,\#\,\Sigma^*$.

Then for `#Young`; `Neil`; `Canada`; 78; `"Godfather of Grunge"#`, we will extract *each* optional attribute as a single span of variable `z`, so the corresponding *multispan*-tuple is:

$$\big(\underbrace{\{[39,44\rangle\}}_{\mathsf{x}}, \underbrace{\{[45,49\rangle\}}_{\mathsf{y}}, \underbrace{\{[57,59\rangle, [60,81\rangle\}}_{\mathsf{z}}\big)\,.$$

Thus, by defining a suitable transformation T_2, we get an ET program (E_2, T_2) for our task.

Our contributions

We start the study of ET programs from an evaluation perspective. Towards this goal, we devise a specific instantiation of our ET framework that achieves a desirable balance between the following objectives:

Robustness. Rather than defining ad-hoc ET programs, we are interested in identifying a class of ET programs that, for the extraction and transformation phase, uses established and robust computational models that have several equivalent (and user-friendly) description mechanisms. For the extraction phase, we use a class of multispanners that can be described by a variant of the regex formulas already used for classical spanners (see [17]); see the examples E_1 and E_2 from above. This model is particularly user-friendly and has good algorithmic properties. Our transformation functions are based on the class of regular string-to-string functions, a well-understood formalism that allows many equivalent description mechanisms.

Expressive Power. Our instantiation of the ET-framework, called regular ET programs, can describe several common extract and transform tasks, including the two examples discussed above.

Efficient Enumeration. The most relevant computational task is to compute all outputs of a regular ET program. Since there are potentially many such outputs (even exponential in the number of variables), an enumeration algorithm is desirable. One could treat the two phases separately: Enumerate all span-tuples extracted in the first phase (a task for which algorithms are known [8, 31]) and then apply the transformation function over each enumerated span-tuple. However, the delay of such an enumeration procedure depends on the size of the input document, since it is an argument of the transformation function. Instead, we design an enumeration algorithm with linear preprocessing that directly enumerates all output documents with output-linear delay.

Composability. Regular ET programs satisfy a property that is in general desirable for practical applications: Composability. This means that if we take the outputs of one ET program and feed them in as inputs for another ET program, then the obtained transformation can also be described by a single ET program. In particular, this also means that we can efficiently enumerate the output documents of the transformation process described by composing several regular ET programs.

Outline

After reviewing further related work, Section 2 introduces the class of multispanners. In Section 3, we present the general setting of ET programs. Then, in Section 4, we instantiate this setting to the case of regular ET programs. In Section 5, we study the expressive power of regular ET programs, and in Section 6, we present our evaluation algorithm. In Section 7, we study the composition of such programs. Finally, we discuss future work in Section 8. Due to space limitations, proof details can be found in the online version [38].

Further related work

String transductions [9, 32, 10] – a classical model in computer science – have recently gained renewed attention with the theory of polyregular functions [2]. Although we can see an ET program as a single transduction, our work has several novel contributions compared to classical string transductions. Firstly, our framework models the process by two declarative phases (which is natural from a data management perspective), contrary to string transductions that model the task as a single process. Secondly, we are concerned with bag semantics, which are usually not considered in the context of transductions. Moreover, efficient enumeration algorithms have not been studied in this context.

On the practical side, there are systems for transforming documents into documents (e.g., [27]). For example, they used a combination of regular expressions with replace operators [28] or parsing followed by a transduction over the parsing tree [27]. Indeed, in practice, regular expressions support some special commands for transforming data (also called substitutions). Our study has a theoretical emphasis on information extraction. To the best of our knowledge, previous systems neither use the expressive power of document spanners nor regular functions. In particular, previous systems cannot define queries like (E_2, T_2) through regular expressions plus a transformation.

2 Multispanners

Let us first give some standard notations. By $\mathcal{P}(A)$ we denote the power set of a set A. Let $\mathbb{N} = \{1, 2, 3, \ldots\}$ and $[n] = \{1, 2, \ldots, n\}$ for $n \in \mathbb{N}$. For a finite alphabet A, let A^+ denote the set of non-empty words over A, and $A^* = A^+ \cup \{\varepsilon\}$, where ε is the empty word. For a

word $w \in A^*$, $|w|$ denotes its length (in particular, $|\varepsilon| = 0$). A word $v \in A^+$ is a *factor* of w if there are $u_1, u_2 \in A^*$ with $w = u_1 v u_2$; v is a *prefix* or *suffix* of w, if $u_1 = \varepsilon$ or $u_2 = \varepsilon$, respectively. For every $i \in [|w|]$, let $w[i]$ denote the symbol at position i of w. We use DFAs and NFAs (deterministic and nondeterministic finite automata, resp.) as commonly defined.

Multispans and multispanners

For a document $\mathbf{D} \in \Sigma^*$ and for every $i, j \in [|\mathbf{D}|+1]$ with $i \leq j$, $[i, j\rangle$ is a *span of* $\mathbf{D}$ and its *value*, denoted by $\mathbf{D}[i, j\rangle$, is the substring of $\mathbf{D}$ from symbol i to symbol $j-1$. In particular, $\mathbf{D}[i, i\rangle = \varepsilon$ (called an *empty span*) and $\mathbf{D}[1, |\mathbf{D}|+1\rangle = \mathbf{D}$. By $\mathsf{Spans}(\mathbf{D})$, we denote the set of spans of $\mathbf{D}$, and by Spans we denote the set of all spans $\{[i, j\rangle \mid i, j \in \mathbb{N},\ i \leq j\}$. Two spans $[i, j\rangle$ and $[i', j'\rangle$ are *disjoint* if $j \leq i'$ or $j' \leq i$. A *multispan* is a (possibly empty) set of pairwise disjoint spans. Let $\mathcal{X}$ be a finite set of variables. A *span-tuple* (over a document $\mathbf{D}$ and variables $\mathcal{X}$) [18] is a function $t\colon \mathcal{X} \to \mathsf{Spans}(\mathbf{D})$. We define a *multispan-tuple* as a function $t\colon \mathcal{X} \to \mathcal{P}(\mathsf{Spans}(\mathbf{D}))$ such that, for every $\mathsf{x} \in \mathcal{X}$, $t(\mathsf{x})$ is a multispan. Note that every span-tuple t can be considered as a special case of a multispan-tuple t' with $t'(\mathsf{x}) = \{t(\mathsf{x})\}$ for every $\mathsf{x} \in \mathcal{X}$. For simplicity, we usually denote multispan-tuples in tuple-notation, for which we assume an order $<$ on $\mathcal{X}$. For example, if $\mathcal{X} = \{\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3\}$ with $\mathsf{x}_1 < \mathsf{x}_2 < \mathsf{x}_3$, the multispan-tuple $t = (\{[1, 6\rangle\}, \emptyset, \{[2, 3\rangle, [5, 7\rangle\})$ maps x_1 to $\{[1, 6\rangle\}$, x_2 to $\emptyset$ and x_3 to $\{[2, 3\rangle, [5, 7\rangle\}$. Note that $[2, 3\rangle$ and $[5, 7\rangle$ are disjoint, while $[1, 6\rangle$ and $[5, 7\rangle$ are not, which is allowed, since they are spans of different multispans.

A *multispan-relation* (over a document $\mathbf{D}$ and variables $\mathcal{X}$) is a possibly empty set of multispan-tuples over $\mathbf{D}$ and $\mathcal{X}$. Given a finite alphabet Σ, a *multispanner* (over Σ and $\mathcal{X}$) is a function that maps every document $\mathbf{D} \in \Sigma^*$ to a multispan-relation over $\mathbf{D}$ and $\mathcal{X}$. Note that the empty relation $\emptyset$ is also a valid image of a multispanner.

▶ **Example 1.** Let S be a multispanner over alphabet $\{\mathtt{a}, \mathtt{b}\}$ and variables $\{\mathsf{x}, \mathsf{y}\}$ that maps every document $\mathbf{D} \in \{\mathtt{a}, \mathtt{b}\}^*$ to the set of all multispan-tuples t such that $t(\mathsf{x}) = \{[i, j\rangle\}$ where $\mathbf{D}[i, j\rangle$ is a factor that starts and ends with $\mathtt{b}$, and is not directly preceded or followed by another $\mathtt{b}$, and $t(\mathsf{y})$ is the multispan that contains a span for each maximal unary (i.e., of the form $\mathtt{a}^+$ or $\mathtt{b}^+$) factor of $\mathbf{D}[i, j\rangle$. For example, $t \in S(\mathtt{aababbbaab})$ with:

$$t(\mathsf{x}) = \{[3, 11\rangle\} \text{ and } t(\mathsf{y}) = \{[3, 4\rangle, [4, 5\rangle, [5, 8\rangle, [8, 10\rangle, [10, 11\rangle\}\,.$$

Similarly to the framework of document spanners [18], it is convenient to represent multispanners over Σ and $\mathcal{X}$ by formal languages over the alphabet $\Sigma \cup \{{}_{\mathsf{x}}\vdash, \dashv_{\mathsf{x}} \mid \mathsf{x} \in \mathcal{X}\}$. This allows us to represent multispanners by formal language descriptors, e. g., regular expressions.

Representing multispans by multiref-words

In this section, we adapt the concept of ref-words (commonly used for classical document spanners; see [23, 21, 14, 41]) to multispanners.

For any set $\mathcal{X}$ of variables, we shall use the set $\Gamma_{\mathcal{X}} = \{{}_{\mathsf{x}}\vdash, \dashv_{\mathsf{x}} \mid \mathsf{x} \in \mathcal{X}\}$ as an alphabet of meta-symbols. In particular, for every $\mathsf{x} \in \mathcal{X}$, we interpret the pair of symbols ${}_{\mathsf{x}}\vdash$ and $\dashv_{\mathsf{x}}$ as a pair of opening and closing parentheses. A *multiref-word* (*over alphabet* Σ *and variables* $\mathcal{X}$) is a word $w \in (\Sigma \cup \Gamma_{\mathcal{X}})^*$ such that, for every $\mathsf{x} \in \mathcal{X}$, the subsequence of the occurrences of ${}_{\mathsf{x}}\vdash$ and of $\dashv_{\mathsf{x}}$ is well-balanced and unnested, namely, has the form $({}_{\mathsf{x}}\vdash \dashv_{\mathsf{x}})^k$ for some $k \geq 0$.

Intuitively, any multiref-word w over Σ and $\mathcal{X}$ uniquely describes a document $\mathbf{D}_w \in \Sigma^*$ and a multispan-tuple t_w as follows. First, let $\mathbf{D}_w$ be obtained from w by erasing all symbols from $\Gamma_{\mathcal{X}}$. We note that, for every $\mathsf{x} \in \mathcal{X}$, every matching pair ${}_{\mathsf{x}}\vdash$ and $\dashv_{\mathsf{x}}$ in w (i. e., every

occurrence of ${}_{\mathtt{x}}\vdash$ and the following occurrence of $\dashv_{\mathtt{x}}$) uniquely describes a span of $\mathbf{D}_w$: ignoring all other occurrences of symbols from $\Gamma_{\mathcal{X}}$, this pair encloses a factor of $\mathbf{D}_w$. Consequently, we simply define that, for every $\mathtt{x} \in \mathcal{X}$, $\mathtt{t}_w(\mathtt{x})$ contains all spans defined by matching pairs ${}_{\mathtt{x}}\vdash$ and $\dashv_{\mathtt{x}}$ of w. The property that the subsequence of w of the occurrences ${}_{\mathtt{x}}\vdash$ and $\dashv_{\mathtt{x}}$ has the form $({}_{\mathtt{x}}\vdash \dashv_{\mathtt{x}})^k$ for some $k \geq 0$ implies that all spans of $\mathtt{t}_w(\mathtt{x})$ are pairwise disjoint.

▶ **Example 2.** Let us consider the following multiref-word over the finite alphabet Σ and variables $\mathcal{X} = \{\mathtt{x}, \mathtt{y}\}$: $w = \mathtt{aa}\,{}_{\mathtt{x}}\vdash {}_{\mathtt{y}}\vdash \mathtt{b}\dashv_{\mathtt{y}}{}_{\mathtt{y}}\vdash \mathtt{a}\dashv_{\mathtt{y}}{}_{\mathtt{y}}\vdash \mathtt{bbb}\dashv_{\mathtt{y}}{}_{\mathtt{y}}\vdash \mathtt{aa}\dashv_{\mathtt{y}}{}_{\mathtt{y}}\vdash \mathtt{b}\dashv_{\mathtt{y}}\dashv_{\mathtt{x}}$. By definition, w represents the document $\mathbf{D}_w = \mathtt{aababbbaab}$ and the multispan-tuple (from Example 1):

$$\mathtt{t}_w = (\underbrace{\{[3,11\rangle\}}_{\mathtt{x}}, \underbrace{\{[3,4\rangle, [4,5\rangle, [5,8\rangle, [8,10\rangle, [10,11\rangle\}}_{\mathtt{y}})$$

The advantage of the notion of multiref-words is that it allows to easily describe both a document and some multispan-tuple over this document. Therefore, one can use any set of multiref-words to define a multispanner as follows. A *multiref-language* (*over terminal alphabet* Σ *and variables* $\mathcal{X}$) is a set of multiref-words over Σ and $\mathcal{X}$. Any multiref-language L describes the multispanner $[\![L]\!]$ over Σ and $\mathcal{X}$ defined as follows. For every $\mathbf{D} \in \Sigma^*$: $[\![L]\!](\mathbf{D}) = \{\mathtt{t}_w \mid w \in L \text{ and } \mathbf{D}_w = \mathbf{D}\}$.

Analogously to classical spanners (cf. [39, 23, 21, 14, 41]), we define the class of *regular multispanners* as those multispanners S with $S = [\![L]\!]$ for some regular multiref-language L. Moreover, as done in [18] for classical spanners, we will use a class of regular expressions to define a subclass of regular multispanners that shall play a central role in the extraction phase of our extraction and transformation framework.

Regex multispanners

Let Σ be a finite alphabet and let $\mathcal{X}$ be a finite set of variables. We now define *multispanner-expressions* (*over* Σ *and* $\mathcal{X}$). Roughly speaking, these expresssions are a particular class of regular expressions for defining sets of multiref-words and therefore multispanners. A multispanner-expression R (over Σ and $\mathcal{X}$) satisfies the syntax:

$$R \;:=\; \varepsilon \mid a \in \Sigma \mid (R \cdot R) \mid (R + R) \mid R^* \mid \mathtt{x}\{R\}$$

for every $\mathtt{x} \in \mathcal{X}$ such that $\mathtt{x}$ does not appear in R. Such a multispanner-expression R naturally defines a set of multiref-words $\mathcal{L}(R)$ as follows: $\mathcal{L}(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}(a) = \{a\}$, $\mathcal{L}(R \cdot R') = \mathcal{L}(R) \cdot \mathcal{L}(R')$, $\mathcal{L}(R + R') = \mathcal{L}(R) \cup \mathcal{L}(R')$, $\mathcal{L}(R^*) = \mathcal{L}(R)^*$, and $\mathcal{L}(\mathtt{x}\{R\}) = \{{}_{\mathtt{x}}\vdash\} \cdot \mathcal{L}(R) \cdot \{\dashv_{\mathtt{x}}\}$, where, for every $L, L' \subseteq \Sigma^*$, $L \cdot L' = \{uv \mid u \in L \wedge v \in L'\}$, $L^0 = \{\varepsilon\}$, $L^i = L \cdot L^{i-1}$ for every $i \geq 1$, and $L^* = \bigcup_{i=0}^{\infty} L^i$. As usual, we use R^+ as a shorthand for $(R \cdot R^*)$.

One can easily prove that any multispanner-expression defines a multiref-language, since we do not allow expressions of the form $\mathtt{x}\{R\}$ whenever R mentions $\mathtt{x}$. Thus, we can define the *multispanner* $[\![R]\!]$ *specified by* R as $[\![R]\!] = [\![\mathcal{L}(R)]\!]$. Furthermore, we say that a multispanner S is a *regex multispanner* if $S = [\![R]\!]$ for some multispanner-expression R. Note that regex multispanners form a strict subclass of regular multispanners, similar to the class of regex spanners and regular spanners [18].

▶ **Example 3.** $R := (\varepsilon + \Sigma^*\mathtt{a}^+) \cdot \mathtt{x}\big\{(\mathtt{y}\{\mathtt{b}^+\} \cdot \mathtt{y}\{\mathtt{a}^+\})^* \cdot \mathtt{y}\{\mathtt{b}^+\}\big\} \cdot (\mathtt{a}^+\Sigma^* + \varepsilon)$ is a multispanner-expression with $[\![R]\!]$ being the multispanner S from Example 1; thus, S is a regex multispanner.

Comparison with classical spanners

Multispanners are designed to naturally extend the classical model of spanners from [18] to the setting where variables are mapped to sets of spans instead of single spans. Let us discuss a few particularities of our definitions.

We first note that since classical span-tuples, span-relations and spanners (in the sense of [18]) can be interpreted as special multispan-tuples, multispan-relations and multispanners, respectively, our framework properly extends the classical spanner framework.

A multispan-tuple t allows $[i,j\rangle \in t(\mathsf{x})$ and $[k,l\rangle \in t(\mathsf{y})$ with $i \leq k < j \leq l$ and $\mathsf{x} \neq \mathsf{y}$ (and this is also the case for classical span-tuples). However, $[i,j\rangle, [k,l\rangle \in t(\mathsf{x})$ with $i \leq k < j \leq l$ is not possible for multispan-tuples, since then representing $[i,j\rangle$ and $[k,l\rangle$ by parentheses ${}_{\mathsf{x}}\vdash \ldots \dashv_{\mathsf{x}}$ in the document cannot be distinguished from representing $[i,l\rangle$ and $[k,j\rangle$. Furthermore, for distinct $s, s' \in t(\mathsf{x})$ we require that s and s' are disjoint, which is motivated by the fact that without this restriction, the subsequence of all ${}_{\mathsf{x}}\vdash$ and $\dashv_{\mathsf{x}}$ occurrences could be an arbitrary well-formed parenthesised expression (instead of a sequence $({}_{\mathsf{x}}\vdash \dashv_{\mathsf{x}})^k$); thus, recognising whether a given string over $\Sigma \cup \Gamma_{\mathcal{X}}$ is a proper multiref-word cannot be done by an NFA, as is the case for classical spanners.

Our main motivation for multispanners is that they can express information extraction tasks that are of interest in the context of our extract-transform framework (and that cannot be represented by classical spanners in a convenient way). However, there are other interesting properties of multispanners not related to their application in our extract-transform framework, which deserve further investigation. For example, if L and L' are multiref-languages (i. e., $[\![L]\!]$ and $[\![L']\!]$ are multispanners), then $L \cup L'$, $L \cdot L'$ and L^* are also multiref-languages (and therefore $[\![L \cup L']\!]$, $[\![L \cdot L]\!]$ and $[\![L^*]\!]$ are also multispanners). For classical spanners, this is only true for the union. Consequently, multispanners show some robustness not provided by classical spanners.

3 The extract-transform framework

The setting

An *extract-transform program* (for short: *ET program*) is a pair (E, T) such that, for some finite alphabet Σ,

- E is a multispanner (over Σ and some finite set $\mathcal{X}$ of variables), and
- T is a function that, for every document $\mathbf{D} \in \Sigma^*$, maps $(\mathbf{D}, E(\mathbf{D}))$ to the desired output.

In other words, given a document $\mathbf{D}$, the multispanner E specifies how to extract the relevant data from $\mathbf{D}$ as a multispan-relation $E(\mathbf{D})$, and T specifies how to transform the extracted data $(\mathbf{D}, E(\mathbf{D}))$ into new data. The output of T is application-dependent, and various contexts are conceivable, such as transforming the data into a relational database, a graph database, a single document, or a collection of documents.

One may wonder why we must define the setting in two phases and why not simplify it into one phase. In the end, the purpose of users is to transform data, and then they may like to specify the transformation with a single query language. From a user perspective, we argue that it is useful to have a two-phase specification for several reasons. First, it is already the case that, in practice, people specify the transformation of documents with a two-phase approach. For example, search-and-replace is given by a pattern (i.e., E) and the string to be replaced (i.e., T). More generally, the so-called regex substitutions [33, 25] are specified by a regex (i.e., E) and a replacement pattern (i.e., T). Second, in a more general sense, data transformation today is performed between different data models by ETL programs [44] (for

Extract-Transform-Load) where the extract and transform steps are specified by different languages with different purposes. We made the same decision here, where multispanners are well-suited for the extraction phase, and we left open the language to be used for the transformation phase (that depends on the application). Last, one can argue that separating the process between extraction and transformation aids in decoupling the source from the target data, simplifying the overall specification for users. Indeed, an expert user in the source data can specify the multispanner E, whereas another expert user in the target data can provide the transformation T. Moreover, this separation allows that whenever the middle schema (i.e., $\mathcal{X}$) is not changed, one can update E or T without modifying the other part.

From documents to bags of documents

In this work, we focus on the case where the transformation T is given by a function M_T that maps a pair $(\mathbf{D}, t)$ to a document. This means that the transformation T is fully determined by the function M_T. For a given document $\mathbf{D}$, the output of an ET program (E, T) is a *bag of documents* defined as follows:

$$[\![E \cdot T]\!](\mathbf{D}) := \{\!\{M_T(\mathbf{D}, t) \mid t \in E(\mathbf{D})\}\!\}.$$

In the introduction, we presented two examples of this setting, where the ET programs map documents into a bag of documents. Next, we provide another example to show that this setting also includes the search-and-replace scenario as a special case.

▶ **Example 4.** Let $\mathbf{D}$ be a document over the alphabet $\{\mathtt{a}, \mathtt{b}\}$. Suppose that a user wants to search for all contiguous sequences of `a` and replace each such sequence with a single `a`. For example, the user wants to convert `baaabbaa` into `babba`. In this scenario, the user wants to map its document to a single document. For this goal, we can extract all contiguous sequences of `a` in a single multispan-tuple with the following expression: $E_3 := (\mathtt{b}^* \cdot \mathtt{x}\{\mathtt{a}^+\} \cdot (\mathtt{b}^+ \cdot \mathtt{x}\{\mathtt{a}^+\})^* \cdot \mathtt{b}^*) + \mathtt{b}^*$. The reader can note that the above expression will always capture a single multispan-tuple t where $t(\mathtt{x})$ contains a set of all spans of contiguous sequences of `a`. Then, with a function M_T that replaces each non-empty string $\mathbf{D}[i, j\rangle$ with a single `a` in $\mathbf{D}$ for every $[i, j\rangle \in t(\mathtt{x})$, $[\![E \cdot T]\!](\mathbf{D})$ will output a single document with the expected result. Note that we are extracting all spans with a single variable (i.e., no intersection between spans), which allows us to define the meaning of "replace" easily. Arguably, this provides some evidence to the need of multispanners for such applications.

Our decision to focus on bags (instead of sets) is that each multispan-tuple represents a data item and context in the document. Although T could map two data pieces to the same output document, a user may want to keep both copies, given that they come from different positions in the document. Let us motivate this briefly by a more practical example. Assume that the extraction phase marks the addresses in a list of customer orders, while the transformation phase then transforms these addresses into a format suitable for printing on the parcels to be shipped to the customers. In the likely situation that there are different orders by the same person, the extraction phase produces different markings that will all be transformed into the same address label. Such duplicates, however, are important, since we actually want to obtain an address label for each distinct order (i. e., distinct marking), even if these addresses are the same (although the orders are not). This means that the output sets of our ET programs should actually be bags for this scenario. Indeed, similar situations occur for relational data management systems (and database systems in general) where bag semantics is usually adopted [26]. Of course, one can also consider the scenario when the output is a single document or a set of documents. Both are interesting scenarios, and we leave them for future work (see Section 8 for further discussion).

The evaluation problem of ET-programs

The main technical goal of this paper is to identify a large class of ET programs (E, T) for which, upon input of any document $\mathbf{D}$, the output $[\![E \cdot T]\!](\mathbf{D})$ can be computed efficiently. Arguably, identifying such a large class of ET-programs is crucial for the foundations of extract-transform tasks. On the one hand, it determines which ET-programs can be used in practice and, on the other hand, it guides the design of query languages for the specifications E and T, which depends on the application.

In this work, we study the case when M_T is given by a regular function, an expressive class of string-to-string functions that has several equivalent characterizations and good algorithmic properties. In the sequel, we show how to combine the result of a regex multispanner E with a regular function to then present our approach to evaluate such ET programs efficiently.

4 Using regular functions for the transformation phase

Let us first present some more notation. As usual, $f : A \to B$ denotes a function from A to B. When f is partial, we write $f : A \rightharpoonup B$ and use $f(a) = \bot$ to indicate that f is undefined for element $a \in A$. Moreover, $\mathsf{dom}(f) = \{a \in A \mid f(a) \neq \bot\}$ denotes the domain of f. Every partial function $f : A^* \rightharpoonup B^*$ is called a *string-to-string function* with alphabets A and B.

In this paper, we study the case when we use *regular* string-to-string functions for the transformation phase of ET programs. Let us explain why the class of regular functions is a good choice for our setting. As mentioned before, the class of regular functions forms a well-understood formalism for transforming strings which allows for several equivalent representations like two-way transducers, MSO transductions [16], regular transducer expressions [12], or deterministic streaming string transducers (DSSTs) [3] (this last formalism will be crucial for the rest of this paper). Furthermore, regular functions can express most of the linear transformations that one can find in practice. For example, all the use cases presented so far can be defined by using regular functions as our basis for the transformation phase of ET programs. Finally, regular functions have good algorithmic properties, like being closed under composition, a property that we will exploit later (see Section 7).

For using regular functions as our mechanism for the tranformation phase, we need to first align the output of the extraction phase (i.e., a multispan relation) with the input of a regular function (i.e., a string). For this purpose, it is necessary to convert the output of a multispanner (i.e., multispan-tuples) into the input of a string-to-string function (i.e., strings). In the following, we present a unique way to encode multispan-tuples into multiref-words, which will serve as our input object for using regular functions.

A unique multiref-word representation

The representation of documents and multispan-tuples $(\mathbf{D}, t)$ by multiref-words allows us to define multispanners by multiref-languages. But this representation is not unique, which is inconvenient if we want to use it for encoding multispan-tuples as strings. As a remedy, we adopt the following approach: We represent a document $\mathbf{D} \in \Sigma^*$ and multispan-tuple t as a multiref-word w such that factors of w that belong to $\Gamma^*_{\mathcal{X}}$ (i.e., factors between two letters of $\mathbf{D}$) have the form: $(\dashv_{\mathsf{x}} + \varepsilon)({}_{\mathsf{x}}\vdash \dashv_{\mathsf{x}} + \varepsilon)({}_{\mathsf{x}}\vdash + \varepsilon)(\dashv_{\mathsf{y}} + \varepsilon)({}_{\mathsf{y}}\vdash \dashv_{\mathsf{y}} + \varepsilon)({}_{\mathsf{y}}\vdash + \varepsilon)\ldots$.

Specifically, let $\mathbf{D} \in \Sigma^*$ be a document and t a multispan-tuple over $\mathbf{D}$ and variables $\mathcal{X}$. For a variable $\mathsf{x} \in \mathcal{X}$ and $i \in [|\mathbf{D}| + 1]$, define $t[i, x] = (\dashv_{\mathsf{x}})^c({}_{\mathsf{x}}\vdash \dashv_{\mathsf{x}})^e({}_{\mathsf{x}}\vdash)^o$, where $c, e, o \in \{0, 1\}$ with $c = 1$ iff $[j, i\rangle \in t(\mathsf{x})$ for some $j \neq i$; $e = 1$ iff $[i, i\rangle \in t(\mathsf{x})$; and $o = 1$ iff $[i, j\rangle \in t(\mathsf{x})$ for some $j \neq i$. E. g., for the tuple t in Example 1, we have that $t[3, \mathsf{x}] = {}_{\mathsf{x}}\vdash$ and $t[4, \mathsf{y}] = \dashv_{\mathsf{y}}\,{}_{\mathsf{y}}\vdash$.

Let $\mathcal{X} = \{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_m\}$ with $\mathtt{x}_1 \preceq \mathtt{x}_2 \preceq \ldots \preceq \mathtt{x}_m$, where $\preceq$ is some fixed linear order on $\mathcal{X}$. For every $i \in [|\mathbf{D}| + 1]$, we define $t[i] := t[i, \mathtt{x}_1] \cdot t[i, \mathtt{x}_2] \cdot \ldots \cdot t[i, \mathtt{x}_m]$. We can then define the encoding of t and $\mathbf{D}$ as the multiref-word:

$$\mathtt{enc}(t, \mathbf{D}) := t[1] \cdot \mathbf{D}[1] \cdot t[2] \cdot \mathbf{D}[2] \cdot \ldots \cdot \mathbf{D}[|\mathbf{D}|] \cdot t[|\mathbf{D}|+1].$$

Coming back to Example 1, we have $\mathtt{enc}(t, \mathbf{D}) = \mathtt{aa}\,{}_{\mathtt{x}}\vdash{}_{\mathtt{y}}\vdash\mathtt{b}\dashv_{\mathtt{y}}{}_{\mathtt{y}}\vdash\mathtt{a}\dashv_{\mathtt{y}}{}_{\mathtt{y}}\vdash\mathtt{bbb}\dashv_{\mathtt{y}}{}_{\mathtt{y}}\vdash\mathtt{aa}\dashv_{\mathtt{y}}{}_{\mathtt{y}}\vdash\mathtt{b}\dashv_{\mathtt{y}}\dashv_{\mathtt{x}}$ by using the order $\mathtt{x} \preceq \mathtt{y}$. Note that $\mathtt{enc}(t, \mathbf{D})$ is a multiref-word, $\mathbf{D}_{\mathtt{enc}(t,\mathbf{D})} = \mathbf{D}$ and $\mathtt{t}_{\mathtt{enc}(t,\mathbf{D})} = t$. Thus, $\mathtt{enc}(t, \mathbf{D})$ is a correct and unique encoding for t and $\mathbf{D}$ as a multiref-word.

Regular ET-programs

We are now ready to formally define the following class of ET programs. An ET program (E, T) is called a *regular ET program* (from Σ to Ω) if E is a regex multispanner specified by some multispanner-expression R over Σ and some finite set $\mathcal{X}$ of variables, and T is specified by a regular string-to-string function f_T with input alphabet $\Sigma \cup \Gamma_{\mathcal{X}}$ and output alphabet Ω. The result of (E, T) on $\mathbf{D} \in \Sigma^*$ is defined as

$$[\![E \cdot T]\!](\mathbf{D}) := \{\!\{ f_T(\mathtt{enc}(t, \mathbf{D})) \mid t \in [\![R]\!](\mathbf{D}) \}\!\}$$

Note that this definition of $[\![E \cdot T]\!](\mathbf{D})$ means that we first apply the spanner specified by R on $\mathbf{D}$, which produces a multispan-relation $[\![R]\!](\mathbf{D})$. Then, for every multispan-tuple $t \in [\![R]\!](\mathbf{D})$, we apply f_T on the multiref-word $\mathtt{enc}(t, \mathbf{D})$ producing a new document in the output bag $[\![E \cdot T]\!](\mathbf{D})$. Note that while $\mathtt{enc}(\cdot, \cdot)$ is injective, the function f_T might not be. Hence, for $t, t' \in [\![R]\!](\mathbf{D})$ with $t \neq t'$ we might have $f_T(\mathtt{enc}(t, \mathbf{D})) = f_T(\mathtt{enc}(t', \mathbf{D}))$. This leads to duplicates, but, as explained before, we keep them, since we want each distinct $t \in [\![R]\!](\mathbf{D})$ of the extraction phase to correspond to a transformed document in the output.

So far, we have introduced our object of study, regular ET-programs, but we still need to define how to specify them. In particular, how to specify the regular string-to-string function f_T. Regular string-to-string functions are a robust class that admit several possible representations and people have proposed logic-based languages to specify them like MSO transductions and regular transducer expressions [10]. In particular, one can use any of them as the basis for a declarative query language for specifying f_T. In this paper, we concentrate on the evaluation problem of regular ET-programs. Toward this goal, we use deterministic streaming string transducers (DSSTs, [3]) as our model for defining regular functions.

Deterministic streaming string transducers

Let $\mathcal{R}$ be a finite set of registers and Ω a finite alphabet. An *assignment* is a partial function $\sigma : \mathcal{R} \rightharpoonup (\mathcal{R} \cup \Omega)^*$ that assigns each register $X \in \mathcal{R}$ to a string $\sigma(X)$ of registers and letters from Ω. We define the extension $\hat{\sigma} : (\mathcal{R} \cup \Omega)^* \rightharpoonup (\mathcal{R} \cup \Omega)^*$ of an assignment such that $\hat{\sigma}(\alpha_1 \ldots \alpha_n) = \sigma(\alpha_1) \cdot \ldots \cdot \sigma(\alpha_n)$ for every string $\alpha_1 \ldots \alpha_n \in (\mathcal{R} \cup \Omega)^*$ where $\sigma(\mathfrak{o}) = \mathfrak{o}$ for every $\mathfrak{o} \in \Omega$. Further, we assume that $\hat{\sigma}(\alpha_1 \ldots \alpha_n)$ is undefined iff $\sigma(\alpha_i)$ is undefined for some $i \in [n]$. Given two assignments σ_1 and σ_2, we define its composition $\sigma_1 \circ \sigma_2$ as a new assignment such that $[\sigma_1 \circ \sigma_2](X) = \hat{\sigma}_1(\sigma_2(X))$. We say that an assignment σ is *copyless* if, for every $X \in \mathcal{R}$, there is at most one occurrence of X in all the strings $\sigma(Y)$ with $Y \in \mathcal{R}$.

▶ **Example 5.** Consider $\mathcal{R} = \{X, Y\}$ and $\Omega = \{a, b\}$, and the following assignments $\sigma_1, \sigma_2, \sigma_3$, where we write $X := \alpha$ to mean $\sigma(X) = \alpha$: σ_1 is defined via $X := aXa$; $Y := XY$. σ_2 is defined via $X := bXb$; $Y := b$. σ_3 is defined via $X := baXab$; $Y := b$. One can check that $\sigma_1 \circ \sigma_2 = \sigma_3$. Also, σ_2 and σ_3 are copyless assignments, but σ_1 is not.

Note that copyless assignments are closed under composition, namely, if σ_1 and σ_2 are copyless assignments, then $\sigma_1 \circ \sigma_2$ is copyless as well. We denote by $\mathsf{Asg}(\mathcal{R}, \Omega)$ the set of all copyless assignments over registers $\mathcal{R}$ and alphabet Ω.

A *deterministic streaming string transducer* (DSST) [3] is a tuple $\mathcal{T} = (Q, \Sigma, \Omega, \mathcal{R}, \Delta, q_0, F)$, where Q is a finite set of states, Σ is the input alphabet, Ω is the output alphabet, $\mathcal{R}$ is a finite set of registers, $\Delta \colon Q \times \Sigma \rightharpoonup \mathsf{Asg}(\mathcal{R}, \Omega) \times Q$ is the transition function, q_0 is the initial state, and $F : Q \rightharpoonup (\mathcal{R} \cup \Omega)^*$ is a final partial function such that, for every $q \in Q$ and $X \in \mathcal{R}$, X appears at most once in $F(q)$. Intuitively, if $F(q)$ is defined, then q is a final (i.e., accepting) state.

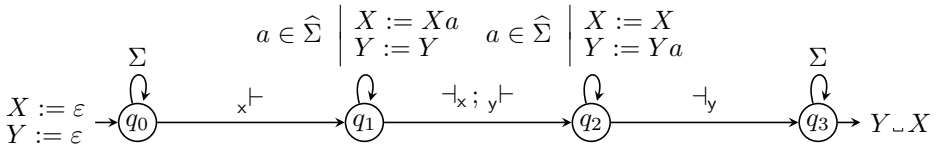
A *configuration* of a DSST $\mathcal{T}$ is a pair (q, ν) where $q \in Q$ and $\nu \in \mathsf{Val}(\mathcal{R}, \Omega)$, and $\mathsf{Val}(\mathcal{R}, \Omega)$ is the set of all assignments $\mathcal{R} \rightharpoonup \Omega^*$, which are called *valuations*. A *run* ρ of $\mathcal{T}$ over a string $a_1 \dots a_n$ is a sequence of configurations (q_i, ν_i) of the form:

$$\rho \;:=\; (q_0, \nu_0) \xrightarrow{a_1/\sigma_1} (q_1, \nu_1) \xrightarrow{a_2/\sigma_2} \dots \xrightarrow{a_n/\sigma_n} (q_n, \nu_n) \tag{*}$$

such that $\Delta(q_i, a_{i+1}) = (\sigma_{i+1}, q_{i+1})$, ν_0 is the empty assignment, i. e. $\nu_0(X) = \varepsilon$ for every $X \in \mathcal{R}$, and $\nu_{i+1} = \nu_i \circ \sigma_{i+1}$ for every $i < n$. A run ρ is called an *accepting run* if $\nu_n(F(q_n))$ is defined. The *output* of an accepting run ρ is defined as the string $\mathsf{out}(\rho) := \nu_n(F(q_n))$.

Since Δ is a partial function, we can see that DSSTs are deterministic, i.e., for every input word w, there exists at most one run ρ. Thus, every DSST $\mathcal{T}$ defines a string-to-string function $[\![\mathcal{T}]\!]$ such that $[\![\mathcal{T}]\!](w) = \mathsf{out}(\rho)$ iff ρ is the run of $\mathcal{T}$ over w and ρ is accepting.

▶ **Example 6.** Recall the first example from the introduction, where the spanner extracts span-tuples t where $t(\mathsf{x})$ and $t(\mathsf{y})$ refer to the spans containing a person's last and first name, respectively. The following illustration depicts a DSST that receives as input a multiref-word for $(\mathbf{D}, t)$ of the form $u\, {}_{\mathsf{x}}\vdash v_{\mathsf{x}} \dashv_{\mathsf{x}}; {}_{\mathsf{y}}\vdash v_{\mathsf{y}} \dashv_{\mathsf{y}} u'$ where $u, u' \in \Sigma^*$ and $v_{\mathsf{x}}, v_{\mathsf{y}} \in \widehat{\Sigma}^*$ (recall that $\widehat{\Sigma} = \Sigma \setminus \{;, \#\}$), and which outputs the word $v_{\mathsf{y}} \sqcup v_{\mathsf{x}}$ where transitions without annotations (i.e., assignments) use the identity assignment.



DSSTs define a well-behaved class of string-to-string functions, equivalent to the the class of regular string-to-string functions (see [10, 3]). In other words, we can use DSSTs as our computational model for specifying the function f_T since all other formalisms for declaring regular functions can be effectively compiled into DSSTs (e.g., MSO transductions or regular transducer expressions). Moreover, DSSTs perform a single pass over the input string, while other equivalent models (e.g., two-way transducers) may do several passes. This streaming behavior of DSSTs will be very useful for designing algorithms for regular ET programs. For this reason, we use DSSTs as our model of string-to-string functions for regular ET programs.

5 Expressiveness of regular ET programs

Nondeterministic SST

We extend DSSTs to the nondeterministic case and bag semantics. Let us explain the model by pointing out its differences to DSSTs.[1]

A *nondeterministic streaming string transducer* (NSST) is a tuple $\mathcal{T} = (Q, \Sigma, \Omega, \mathcal{R}, \Delta, I, F)$, where Q, Σ, Ω, $\mathcal{R}$ and F have the same meaning as for DSSTs. The partial function $I : Q \rightharpoonup \mathsf{Val}(\mathcal{R}, \Omega)$ plays the role of the initial state, i. e., a state q is a possible initial state if $I(q)$ is defined, and in this case $I(q)$ is an initial valuation of the registers. Moreover, Δ is a *bag* of elements from $Q \times \Sigma \times \mathsf{Asg}(\mathcal{R}, \Omega) \times Q$. Note that we define Δ as a bag and not as a set for technical reasons (e.g., the proof of Proposition 10 or Theorem 12).

The semantics of the model are as expected. A run over a string $a_1 \dots a_n$ is a sequence of configurations (q_i, ν_i) of the form (*) such that $I(q_0)$ is defined and $\nu_0 = I(q_0)$, $(q_i, a_{i+1}, \sigma_{i+1}, q_{i+1}) \in \Delta$ and $\nu_{i+1} = \nu_i \circ \sigma_{i+1}$ for every $i < n$. As for DSSTs, ρ is an *accepting run* if $\nu_n(F(q_n))$ is defined, and the output of an accepting run ρ is the string $\mathsf{out}(\rho) = \nu_n(F(q_n))$. We define $\mathsf{Runs}_{\mathcal{T}}(a_1 \dots a_n)$ as the bag of all accepting runs of $\mathcal{T}$ over $a_1 \dots a_n$.

Finally, we define the semantics of an NSST $\mathcal{T}$ over a string $w \in \Sigma^*$ as the bag:

$$[\![\mathcal{T}]\!](w) \;=\; \{\!\{\, \mathsf{out}(\rho) \mid \rho \in \mathsf{Runs}_{\mathcal{T}}(w) \,\}\!\}.$$

Equivalence of regular ET programs and NSSTs

We show that regular ET programs and NSSTs are equivalent. This is a fundamental insight with respect to the expressive power of ET programs. Moreover, the fact that the two-stage model of regular ET programs can be described by a single NSST will be important for our enumeration algorithm (Section 6) and their composition (Section 7).

Before going into the technical details, one may wonder why regular ET programs have two phases when this characterization of the expressive power shows that we can do the whole process with a single NSST. As we already argued in Section 3, a two-phase process is designed from a *user perspective* that aids the declaration of the whole process and helps for the future use of the specifications. On the contrary, the one-phase process of NSSTs is helpful for an *algorithmic perspective*, where we want to evaluate the ET program as a single process and in a single pass.

We first discuss how regular ET programs can be transformed into NSSTs. Every multiref-word over Σ and $\mathcal{X}$ describes a document $\mathbf{D}_w$ and a multispan-tuple $\mathtt{t}_w$. Hence, we can extend the encoding $\mathtt{enc}(\cdot)$ to multiref-words by setting $\mathtt{enc}(w) = \mathtt{enc}(\mathtt{t}_w, \mathbf{D}_w)$. Intuitively speaking, applying the function $\mathtt{enc}(\cdot)$ on a multiref-word w simply means that every maximal factor $u \in \Gamma_{\mathcal{X}}^*$ of w is re-ordered according to the order $\preceq$ on $\mathcal{X}$, and superfluous matching brackets ${}_{\mathsf{x}}\vdash \dashv_{\mathsf{x}}$ are removed (since several of those in the same maximal factor over $\Gamma_{\mathcal{X}}$ would describe the same empty span several times). We define $\mathtt{enc}(L) = \{\mathtt{enc}(w) \mid w \in L\}$ for multiref-languages L.

Let us consider a regular ET program $[\![E \cdot T]\!]$, i. e., E is a regex multispanner represented by some multispanner-expression r, and T is a regular function represented by some DSST $\mathcal{T}$. The high-level idea of the proof is to construct an NSST $\mathcal{T}'$ that simulates a DFA

[1] Note that nondeterministic streaming string transducers with *set semantics* instead of bag semantic have already been introduced in [4].

M for $\mathtt{enc}(\mathcal{L}(r))$ and the DSST $\mathcal{T}$ in parallel. More precisely, we read an input $\mathbf{D} \in \Sigma^*$, but between reading a symbol $\mathbf{D}[i]$ and the next symbol $\mathbf{D}[i+1]$, we pretend to read a sequence of symbols from $\Gamma_{\mathcal{X}}$ with $\mathcal{T}$ and M at the same time. Thus, we virtually read some multiref-word w with the property $\mathbf{D}_w = \mathbf{D}$. We need the DSST $\mathcal{T}$ for producing an output on that multiref-word, and we need the DFA M to make sure that the virtual multiref-word has the form $\mathtt{enc}(t, \mathbf{D})$ for some $t \in E(\mathbf{D})$.

One may wonder why we want M to be a DFA rather than an NFA. The reason is: Having to deal with bag semantics (rather than just set semantics) makes things more complicated. In particular, this means that we need M to be deterministic to have a one-to-one correspondence between the accepting runs of $\mathcal{T}'$ and the accepting runs of M on the corresponding multiref-word. Otherwise, if M was an NFA, the different possible accepting paths of M on the same multiref-word would translate into different accepting paths of $\mathcal{T}'$ which would cause erroneous duplicates in $[\![\mathcal{T}']\!](\mathbf{D})$.

We can transform r into a DFA M for $\mathtt{enc}(\mathcal{L}(r))$ by standard automata constructions, but the fact that M needs to be deterministic means that the construction is (one-fold) exponential in $|r|$, and the fact that it has to accept $\mathtt{enc}(\mathcal{L}(r))$ and not just $\mathcal{L}(r)$ means that the construction is also (one-fold) exponential in $|\mathcal{X}|$. In summary, we obtain the following.

▶ **Theorem 7.** *Given a regex multispanner E over Σ and $\mathcal{X}$ (represented by a multispanner-expression r), and a regular string-to-string function T with input alphabet $\Sigma \cup \Gamma_{\mathcal{X}}$ (represented by a DSST with h states), we can construct an NSST $\mathcal{T}$ with $[\![\mathcal{T}]\!] = [\![E \cdot T]\!]$ in time $\mathrm{O}(2^{4|r|+9|\mathcal{X}|}|\Sigma|^3|\mathcal{X}|^2h^2)$.*

Let us now move on to representing general NSSTs by ET programs. When an NSST $\mathcal{T}$ is in a state p and reads a symbol $b \in \Sigma$, then the number of possible nondeterministic choices it can make is the sum over all the multiplicities of elements $(p, b, \sigma, q) \in \Delta$ (recall that Δ is the bag of transitions). We shall call this number the *nondeterministic branching factor of p and b*. The *nondeterministic branching factor* of $\mathcal{T}$ is the maximum over all the nondeterministic branching factors of p and b for all $(p, b) \in Q \times \Sigma$.

The only obstacle in the simulation of an NSST by a DSST is that the latter cannot deal with the nondeterministic choices. However, in regular ET programs, a DSST gets an annotated version of the actual document $\mathbf{D}$ as input, i. e., a multiref-word w with $\mathbf{D}_w = \mathbf{D}$. Consequently, the DSST could interpret the additional information given by w in the form of the symbols from $\Gamma_{\mathcal{X}}$ as information that determines which nondeterministic choices it should make. More formally, we can construct a regex multispanner that, for every $i \in [1, 2, \ldots, |\mathbf{D}|]$, nondeterministically chooses some $\mathsf{x}_\ell \in \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_m\}$, where m is the NSST's nondeterministic branching factor, and puts the empty span $[i, i\rangle$ into $t(\mathsf{x})$. On the level of multiref-words, this simply means that every symbol is preceded by ${}_{\mathsf{x}_\ell}\vdash \dashv_{\mathsf{x}_\ell}$ for some ℓ. Such an occurrence of ${}_{\mathsf{x}_\ell}\vdash \dashv_{\mathsf{x}_\ell}$ can then be interpreted by the DSST as an instruction to select the ℓ^{th} nondeterministic choice when processing the next symbol from Σ. In summary, we obtain the following result.

▶ **Theorem 8.** *Given an NSST $\mathcal{T}$ with n states, input alphabet Σ and nondeterministic branching factor m, we can construct a regex multispanner E over Σ and $\{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_{\max\{n,m\}}\}$ and a regular function T with $[\![E \cdot T]\!] = [\![\mathcal{T}]\!]$. Moreover, E is represented by a multispanner-expression r with $|r| = \mathrm{O}(|\Sigma| + \max\{n, m\})$, T is represented by a DSST $\mathcal{T}'$ with $\mathrm{O}(\max\{n, m\} \cdot n)$ states, and both r and $\mathcal{T}'$ can be constructed in time $\mathrm{O}(|\mathcal{T}| + n \cdot \max\{n, m\} + |\Sigma|)$.*

6 Evaluation of regular ET programs

In this section, we present the main technical result of the paper, regarding the evaluation of regular ET programs. Specifically, we consider the following enumeration problem where E is a regex multispanner and T is specified by a regular function.

Problem:	EnumRegularET$[(E,T)]$
Input:	A document $\mathbf{D}$
Output:	Enumerate $[\![E \cdot T]\!](\mathbf{D})$

Notice that $[\![E \cdot T]\!](\mathbf{D})$ is a bag; thus, the task is to produce an enumeration that contains each element of the bag exactly once, e. g., (a, a, b, c, a) is a possible enumeration of $\{\!\{b, a, a, a, c\}\!\}$.

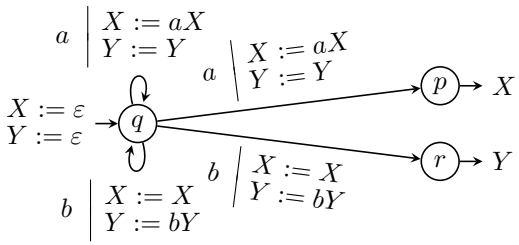
As usual, we measure the running time in *data complexity*, namely, we assume that E and T are fixed. Given this assumption, we can assume that E is given as a multispanner-expression, and T as a DSST. Otherwise, we can convert E and T to satisfy this requirement.

For this problem, we strive for an *enumeration algorithm with linear preprocessing and output-linear delay*, i. e., in a *preprocessing phase* it receives the input and produces some data structure `DS` which encodes the expected output, and in the following *enumeration phase* it produces a sequential enumeration $w_1, \ldots, w_\ell$ of the results from `DS`. Moreover, the time for the preprocessing phase is $O(|\mathbf{D}|)$, the time for producing w_1 is less than $c \cdot |w_1|$, and the time between producing w_{i-1} and w_i is less than $c \cdot |w_i|$, for some fixed constant c that does not depend on the input. As it is common [43], we assume the computational model of *Random Access Machines* (RAM) with uniform cost measure and addition and subtraction as basic operations [1]. We obtain the following result.

▶ **Theorem 9.** EnumRegularET$[(E,T)]$ *admits an enumeration algorithm with linear preprocessing time and output-linear delay.*

Due to space restrictions, all the details and the analysis of the enumeration algorithm are deferred to the online version [38]. In the following, we highlight the main technical challenges. For running the algorithm, we use Theorem 7 and convert the pair (E,T) into an NSST $\mathcal{T}_{E,T}$. This takes time exponential in $|E|$; nevertheless, it does not depend on $|\mathbf{D}|$ and so we can consider it as constant time. Our enumeration algorithm aims for computing $[\![\mathcal{T}_{E,T}]\!](\mathbf{D})$ with linear time preprocessing and output-linear delay.

For evaluating $\mathcal{T}_{E,T}$ over $\mathbf{D}$, the first challenge that we need to overcome is that its runs could maintain registers with content that is not used at the end of the run. For an illustration, consider the following NSST:



For each input word w with n a-symbols and m b-symbols, the NSST outputs a^n if w ends with a and b^m if w ends with b. Consequently, every run on a word that ends with a produces "garbage" in register Y, since the content of this register is not used for the output (and analogously with register X for inputs that end with b). This behavior of storing "garbage" will be problematic for our enumeration approach, given that the delay depends on the (potentially useless) contents of the registers.

Given the above discussion, we formalize the notion of "garbage" as follows. Consider an NSST $\mathcal{T} = (Q, \Sigma, \Omega, \mathcal{R}, \Delta, I, F)$. For $u \in (\mathcal{R} \cup \Omega)^*$, let $\mathsf{reg}(u)$ be the set of all registers $X \in \mathcal{R}$ that appear in u. For $\sigma : \mathcal{R} \rightharpoonup (\mathcal{R} \cup \Omega)^*$ let $\mathsf{reg}(\sigma) = \bigcup_{X \in \mathsf{dom}(\sigma)} \mathsf{reg}(\sigma(X))$, namely, $\mathsf{reg}(\sigma)$ is the set of all registers used by σ. We say that $\mathcal{T}$ is *garbage-free* if, and only if, for every string $w = a_1 \dots a_n$ and every accepting run ρ of the form (*) it holds that $\mathsf{dom}(\nu_i) = \mathsf{reg}(\sigma_{i+1})$ for every $i < n$, and $\mathsf{dom}(\nu_n) = \mathsf{reg}(F(q_n))$. In other words, the registers $\mathsf{dom}(\nu_i)$ that we have filled with content so far coincide with the registers $\mathsf{reg}(\sigma_{i+1})$ that we use on the right hand sides of the next assignment. The first challenge is to show how to make NSSTs garbage-free.

▶ **Proposition 10.** *For every NSST $\mathcal{T}$, there exists a garbage-free NSST $\mathcal{T}'$ such $[\![\mathcal{T}]\!](w) = [\![\mathcal{T}']\!](w)$ for every string w.*

The construction of Proposition 10 causes a blow-up that is exponential in the number of registers, since we turn an NSST with $|Q|$ states into an NSST with $|Q| \times 2^{|\mathcal{R}|}$ states. Of course, if we start with a garbage-free NSST, this blow-up can be avoided. Interestingly, we can show that one can check the garbage-free property in polynomial time.

▶ **Proposition 11.** *Given an NSST $\mathcal{T} = (Q, \Sigma, \Omega, \mathcal{R}, \Delta, I, F)$, we can decide in time $\mathrm{O}(|\Delta| \cdot |\mathcal{R}|)$ whether $\mathcal{T}$ is garbage-free.*

The second challenge is maintaining the set of outputs compactly for enumerating them. The main problem here is that the output of a run is not produced linearly as a single string (as is the case for classical spanners [8, 6]) but instead in parallel on different registers that are combined in some order at the end of the input. To solve this, we follow the approach in [30, 31] and present a non-trivial extension of *Enumerable Compact Sets* (ECS), a data structure that stores sets of strings compactly and retrieves them with output-linear delay. We modify ECS for storing sets of valuations and we call it *ECS with assignments*. For storing valuations, we use assignments in the internal nodes of the data structure, which allows us to encode the runs' output and keep the content of different registers synchronous.

Using assignments in the ECS requires to revisit the whole approach of the data structure. For example, although we can eliminate the garbage from an NSST (cf., Proposition 10), the machine can still swap and move registers during a run, producing assignments that permute the registers' content but do not contribute to a final output. We call these assignments *relabelings* and one of the main challenges is to take care of them. To solve this, we have to treat them as special objects and compact them in the data structure whenever possible. These extensions require to revisit ECS and to provide new ways for extending or unifying sets of valuations by also taking care of relabelings.

7 Composition of regular ET programs

A valuable property of regular ET programs is that they receive documents as input and produce documents as outputs. In this line, it is natural to think on reusing the output of one ET program (E_1, T_1) to feed a second ET program (E_2, T_2), namely, to compose them. Formally, we define the *composition* $(E_1 \cdot T_1) \circ (E_2 \cdot T_2)$ as a function $[\![(E_1 \cdot T_1) \circ (E_2 \cdot T_2)]\!]$ that maps documents to a bag of documents such that for every document $\mathbf{D}$:

$$[\![(E_1 \cdot T_1) \circ (E_2 \cdot T_2)]\!](\mathbf{D}) \;=\; \bigcup_{\mathbf{D}' \in [\![E_1 \cdot T_1]\!](\mathbf{D})} [\![E_2 \cdot T_2]\!](\mathbf{D}').$$

Note that we use here the union of bags, which is defined in the standard way.

For extraction and transformation of information, it is useful to evaluate the composition efficiently. One naive approach is to evaluate $[\![E_1 \cdot T_1]\!](\mathbf{D})$ (e.g., by using the evaluation algorithm of Section 6 in case that (E_1, T_1) is a regular ET program), and for every output $\mathbf{D}'$ in $[\![E_1 \cdot T_1]\!](\mathbf{D})$ compute $[\![E_2 \cdot T_2]\!](\mathbf{D}')$, gathering all the outputs together. Of course, this could be time consuming, since $|[\![E_1 \cdot T_1]\!](\mathbf{D})|$ could be exponential in $|\mathbf{D}|$ in the worst case.

Towards solving the previous algorithmic problem, in the next result we show that every composition of NSSTs can be defined by an NSST. Formally, let us denote the input and output alphabet of some NSST $\mathcal{T}$ by $\Sigma(\mathcal{T})$ and $\Omega(\mathcal{T})$, respectively. Given two NSSTs $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\Omega(\mathcal{T}_1) \subseteq \Sigma(\mathcal{T}_2)$, we define the composition $\mathcal{T}_1 \circ \mathcal{T}_2$ as the function from documents to bags of documents: $[\![\mathcal{T}_1 \circ \mathcal{T}_2]\!](\mathbf{D}) = \bigcup_{\mathbf{D}' \in [\![\mathcal{T}_1]\!](\mathbf{D})} [\![\mathcal{T}_2]\!](\mathbf{D}')$.

▶ **Theorem 12.** *For every pair of NSSTs $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\Omega(\mathcal{T}_1) \subseteq \Sigma(\mathcal{T}_2)$, there exists an NSST $\mathcal{T}$ such that $[\![\mathcal{T}]\!] = [\![\mathcal{T}_1 \circ \mathcal{T}_2]\!]$ and $|\mathcal{T}| = \mathrm{O}(2^{\mathsf{poly}(|\mathcal{T}_1|, |\mathcal{T}_2|)})$.*

The statement of Theorem 12 for set semantics rather than bag semantics was obtained by [4, 5]. The novelty of Theorem 12 is that we extend the result to bag semantics; namely, we need to maintain the multiplicities of the final outputs correctly. The proof revisits (and simplifies) the construction in [4, 5]: we simulate $\mathcal{T}_1$ over $\mathbf{D}$ while $\mathcal{T}_2$ runs over the registers' content of $\mathcal{T}_1$, compactly representing subruns of $\mathcal{T}_2$ by using pairs of states and assignment summaries [3]. The extension requires two changes for working under bag semantics. First, similar to the evaluation of NSSTs, garbage on $\mathcal{T}_1$-registers could generate additional runs for $\mathcal{T}_2$, producing outputs with wrong multiplicities. Therefore, our first step is to remove this garbage by constructing equivalent garbage-free NSSTs by using Propositon 10. Second, the construction in [4] guesses subruns of $\mathcal{T}_2$ for each register content and each starting state. In the end, we will use one of these guesses, and then unused subruns could modify the output multiplicity. Therefore, we simplify the construction in [4] by guessing a single subrun for each register content, using the non-determinism to discard wrong guesses. Interestingly, this simplification overcomes the error pointed out by Joost Engelfriet in the construction of [4], which was solved recently in [5]. Our construction does not need the machinery of [5]; thus, an adaptation of our construction for NSSTs (with set semantics) can be considered as a simplified version of the proof in [4, 5]. We conclude by mentioning that as an alternative proof of Theorem 12 we could move to MSO transductions [16], extend the logic with a bag semantics, and then do the composition at a logical level; however, this strategy would lead to a non-elementary blow-up.

By combining Theorem 7, 12 and 8, we get the following corollary regarding the expressiveness of regular ET programs and their composition.

▶ **Corollary 13.** *For every pair of regular ET programs (E_1, T_1) and (E_2, T_2), there exists a regular ET program (E, T) such that $[\![E \cdot T]\!] = [\![(E_1 \cdot T_1) \circ (E_2 \cdot T_2)]\!]$.*

We conclude this section by the following corollary regarding the evaluation of the composition of regular ET programs, obtained by combining Theorem 12 and 9.

▶ **Corollary 14.** *Given regular ET programs $(E_1, T_1), \ldots, (E_k, T_k)$ we can evaluate the composition $[\![(E_1 \cdot T_1) \circ \ldots \circ (E_k \cdot T_k)]\!](\mathbf{D})$ with linear time preprocessing and output-linear delay in data complexity.*

Finally, it is interesting to note that one could encode a multispanner as an NSST, by outputting directly $\mathtt{enc}(t, \mathbf{D})$ for each multispan-tuple t. Then Theorem 7 can be seen as a corollary of Theorem 12. However, the complexity of constructing an NSST from a regular ET program by using Theorem 12 will be worse than in Theorem 7.

8 Future Work

There are further research questions directly motivated by our results. First, the framework of multispanners deserves further investigation to understand which results and properties of classical spanners directly carry over to multispanners. Along the same lines, it will be interesting to understand the connections between classical spanners and subclasses of NSSTs. Second, one could consider other formalisms for the transformation phase, like the class of polyregular functions [10], that extends regular functions from linear to polynomial growth. Third, one can consider other algorithmic problems for the output, such as enumerating a set instead of a bag of results. Here, our algorithmic technique does not extend directly, since it would be required to remove duplicates in the NSST, or in the algorithmic evaluation, which is unclear. Last, the general setting allows for other application scenarios for the transformation phase, like producing graphs or relations. Furthermore, one can also extend the setting to consider input data with more structure, like nested documents (e.g., JSON, XML). We leave this and other open problems for future work.

References

1 Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

2 Rajeev Alur, Mikołaj Bojańczyk, Emmanuel Filiot, Anca Muscholl, and Sarah Winter. Regular Transformations (Dagstuhl Seminar 23202). *Dagstuhl Reports*, 13(5):96–113, 2023. doi:10.4230/DAGREP.13.5.96.

3 Rajeev Alur and Pavol Cerný. Expressiveness of streaming string transducers. In *FSTTCS*, pages 1–12, 2010. doi:10.4230/LIPICS.FSTTCS.2010.1.

4 Rajeev Alur and Jyotirmoy V. Deshmukh. Nondeterministic streaming string transducers. In *ICALP*, volume 6756, pages 1–20, 2011. doi:10.1007/978-3-642-22012-8_1.

5 Rajeev Alur, Taylor Dohmen, and Ashutosh Trivedi. Composing copyless streaming string transducers. *CoRR*, abs/2209.05448, 2022. doi:10.48550/arXiv.2209.05448.

6 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *PODS*, pages 89–103. ACM, 2019. doi:10.1145/3294052.3319702.

7 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. *SIGMOD Rec.*, 49(1):25–32, 2020. doi:10.1145/3422648.3422655.

8 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. *ACM Transactions on Database Systems (TODS)*, 46(1):1–30, 2021. doi:10.1145/3436487.

9 Jean Berstel. *Transductions and context-free languages*. Springer-Verlag, 2013.

10 Mikolaj Bojanczyk. Transducers of polynomial growth. In *LICS*, pages 1:1–1:27. ACM, 2022. doi:10.1145/3531130.3533326.

11 Pierre Bourhis, Alejandro Grez, Louis Jachiet, and Cristian Riveros. Ranked enumeration of MSO logic on words. In *24th International Conference on Database Theory, ICDT 2021, March 23-26, 2021, Nicosia, Cyprus*, pages 20:1–20:19, 2021. doi:10.4230/LIPICS.ICDT.2021.20.

12 Vrunda Dave, Paul Gastin, and Shankara Narayanan Krishna. Regular transducer expressions for regular transformations. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 315–324, 2018. doi:10.1145/3209108.3209182.

13 Johannes Doleschal, Benny Kimelfeld, and Wim Martens. The complexity of aggregates over extractions by regular expressions. *Logical Methods in Computer Science*, 19(3), 2023. doi:10.46298/LMCS-19(3:12)2023.

14 Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, and Frank Neven. Split-correctness in information extraction. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 149–163, 2019. doi:10.1145/3294052.3319684.

15 Johannes Doleschal, Benny Kimelfeld, Wim Martens, and Liat Peterfreund. Weight annotation in information extraction. *Logical Methods in Computer Science*, 18, 2022. doi:10.46298/LMCS-18(1:21)2022.

16 Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Trans. Comput. Log.*, 2(2):216–254, 2001. doi:10.1145/371316.371512.

17 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Spanners: a formal framework for information extraction. In *PODS*, pages 37–48. ACM, 2013. doi:10.1145/2463664.2463665.

18 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *Journal of the ACM (JACM)*, 62(2):1–51, 2015. doi:10.1145/2699442.

19 Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoč. Efficient enumeration algorithms for regular document spanners. *ACM Transactions on Database Systems (TODS)*, 45(1):1–42, 2020. doi:10.1145/3351451.

20 Dominik Freydenberger and Mario Holldack. Document spanners: From expressive power to decision problems. *Theory of Computing Systems*, 62:854–898, 2018. doi:10.1007/S00224-017-9770-0.

21 Dominik D. Freydenberger. A logic for document spanners. *Theory Comput. Syst.*, 63(7):1679–1754, 2019. doi:10.1007/S00224-018-9874-1.

22 Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. 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*, pages 137–149, 2018. doi:10.1145/3196959.3196967.

23 Dominik D. Freydenberger and Sam M. Thompson. Dynamic complexity of document spanners. In *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*, pages 11:1–11:21, 2020. doi:10.4230/LIPICS.ICDT.2020.11.

24 Dominik D. Freydenberger and Sam M. Thompson. 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)*, pages 10:1–10:18, 2022. doi:10.4230/LIPIcs.ICDT.2022.10.

25 Jeffrey Friedl. *Mastering regular expressions*. " O'Reilly Media, Inc.", 2006.

26 Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

27 Jerry R. Hobbs, Douglas E. Appelt, John Bear, David J. Israel, Megumi Kameyama, Mark E. Stickel, and Mabry Tyson. FASTUS: A cascaded finite-state transducer for extracting information from natural-language text. *CoRR*, cmp-lg/9705013, 1997. URL: http://arxiv.org/abs/cmp-lg/9705013.

28 Lauri Karttunen. The replace operator. *Finite-State Language Processing*, pages 117–147, 1997.

29 Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. Document spanners for extracting incomplete information: Expressiveness and complexity. In *PODS*, pages 125–136, 2018. doi:10.1145/3196959.3196968.

30 Martin Muñoz and Cristian Riveros. Streaming enumeration on nested documents. In *ICDT*, volume 220 of *LIPIcs*, pages 19:1–19:18, 2022. doi:10.4230/LIPICS.ICDT.2022.19.

31 Martin Muñoz and Cristian Riveros. Constant-delay enumeration for slp-compressed documents. In *ICDT*, volume 255, pages 7:1–7:17, 2023. doi:10.4230/LIPICS.ICDT.2023.7.

32 Anca Muscholl and Gabriele Puppis. The many facets of string transducers (invited talk). In Rolf Niedermeier and Christophe Paul, editors, *STACS*, volume 126 of *LIPIcs*, pages 2:1–2:21, 2019. doi:10.4230/LIPICS.STACS.2019.2.

33 https://perldoc.perl.org/perlre, 2024. Accessed on 2024-09-16.

34 Liat Peterfreund. Grammars for document spanners. In Ke Yi and Zhewei Wei, editors, *ICDT*, volume 186 of *LIPIcs*, pages 7:1–7:18, 2021. doi:10.4230/LIPICS.ICDT.2021.7.

35 Liat Peterfreund, Dominik D. Freydenberger, Benny Kimelfeld, and Markus Kröll. 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.*, pages 320–334, 2019. doi:10.1145/3294052.3319699.

36 Liat Peterfreund, Balder ten Cate, Ronald Fagin, and Benny Kimelfeld. Recursive programs for document spanners. In *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, pages 13:1–13:18, 2019. doi:10.4230/LIPICS.ICDT.2019.13.

37 Cristian Riveros, Nicolás Van Sint Jan, and Domagoj Vrgoc. Rematch: a novel regex engine for finding all matches. *VLDB*, 16(11):2792–2804, 2023. doi:10.14778/3611479.3611488.

38 Cristian Riveros, Markus L. Schmid, and Nicole Schweikardt. A framework for extraction and transformation of documents. *CoRR*, abs/2405.12350, 2024. doi:10.48550/arXiv.2405.12350.

39 Markus L. Schmid and Nicole Schweikardt. A purely regular approach to non-regular core spanners. In Ke Yi and Zhewei Wei, editors, *24th International Conference on Database Theory, ICDT 2021, March 23-26, 2021, Nicosia, Cyprus*, volume 186 of *LIPIcs*, pages 4:1–4:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ICDT.2021.4.

40 Markus L. Schmid and Nicole Schweikardt. 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*, pages 153–165, 2021. doi:10.1145/3452021.3458325.

41 Markus L. Schmid and Nicole Schweikardt. Document spanners - A brief overview of concepts, results, and recent developments. In Leonid Libkin and Pablo Barceló, editors, *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 139–150. ACM, 2022. doi:10.1145/3517804.3526069.

42 Markus L. Schmid and Nicole Schweikardt. Query evaluation over slp-represented document databases with complex document editing. In *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 79–89, 2022. doi:10.1145/3517804.3524158.

43 Luc Segoufin. Enumerating with constant delay the answers to a query. In *ICDT*, pages 10–20, 2013. doi:10.1145/2448496.2448498.

44 Panos Vassiliadis. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)*, 5(3):1–27, 2009. doi:10.4018/JDWM.2009070101.