Split-Correctness in Information Extraction.

DOLESCHAL, Johannes; Kimelfeld, Benny; MARTENS, Wim; Nahshon, Yoav & NEVEN, Frank (2019) Split-Correctness in Information Extraction.. In: PODS '19 Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, ACM,p. 149-163.

DOI: 10.1145/3294052.3319684
Handle: http://hdl.handle.net/1942/29007

Split-Correctness in Information Extraction

Johannes Doleschal
University of Bayreuth & Hasselt University
johannes.doleschal@uni-bayreuth.de

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

Wim Martens
University of Bayreuth
wim.martens@uni-bayreuth.de

Yoav Nahshon
Technion, Israel
yoavn@cs.technion.ac.il

Frank Neven
Hasselt University & transnational University of Limburg
frank.neven@uhasselt.be

ABSTRACT

Programs for extracting structured information from text, namely *information extractors*, often operate separately on document segments obtained from a generic splitting operation such as sentences, paragraphs, k-grams, HTTP requests, and so on. An automated detection of this behavior of extractors, which we refer to as *split-correctness*, would allow text analysis systems to devise query plans with parallel evaluation on segments for accelerating the processing of large documents. Other applications include the incremental evaluation on dynamic content, where re-evaluation of information extractors can be restricted to revised segments, and debugging, where developers of information extractors are informed about potential boundary crossing of different semantic components.

We propose a new formal framework for split-correctness within the formalism of document spanners. Our preliminary analysis studies the complexity of split-correctness over regular spanners. We also discuss different variants of split-correctness, for instance, in the presence of black-box extractors with "*split constraints*".

CCS CONCEPTS

• **Information systems** → **Information extraction**; • **Theory of computation** → *Formal languages and automata theory*; *Parallel algorithms.*

PODS '19, June 30–July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6227-6/19/06…$15.00
https://doi.org/10.1145/3294052.3319684

KEYWORDS

Information Extraction, Spanners, Complexity

ACM Reference Format:
Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, and Frank Neven. 2019. Split-Correctness in Information Extraction. In *38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '19), June 30–July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3294052.3319684

1 INTRODUCTION

Information extraction (IE), the extraction of structured data from text, is a core operation when dealing with text in data analysis. Programming frameworks for IE, and especially *declarative* ones, are designed to facilitate the development of IE solutions. For example, IBM's SystemT [5] exposes an SQL-like declarative language, *AQL* (Annotation Query Language), which provides a collection of "primitive" extractors (e.g., tokenizer, dictionary lookup, Part-Of-Speech (POS) tagger, and regular-expression matcher) alongside the relational algebra for manipulating these relations. In Xlog [29], user-defined functions are used as primitive extractors, and Datalog is used for relation manipulation. In DeepDive [28, 30], rules are used for generating features that are translated into the factors of a statistical model with machine-learned parameters.

When applied to a large document, an IE function may incur a high computational cost and, consequently, an impractical execution time. However, it is frequently the case that the program, or at least most of it, can be distributed by separately processing smaller chunks in parallel. For instance, *Named Entity Recognition* (NER) is often applied separately to different sentences [17, 18], and so are instances of *Relation Extraction* [20, 36]. Algorithms for *coreference resolution* (identification of places that refer to the same entity) are typically bounded to limited windows; for instance, Stanford's well known *sieve* algorithm [27] for coreference resolution

processes separately intervals of three sentences [19]. Sentiment extractors typically process individual paragraphs or even sentences [26]. It is also common for extractors to operate on windows of a bounded number N of words (tokens), also known as *N-grams* or *local contexts* [4, 13]. Finally, machine logs often have a natural split into semantic chunks: query logs into queries, error logs into exceptions, web-server logs into HTTP messages, and so on.

Tokenization, N-gram extraction, paragraph segmentation (identifying paragraph breaks, whether or not marked explicitly [15]), sentence boundary detection, and machine-log itemization are all examples of what we call *splitters*. When IE is programmed in a development framework such as the aforementioned, we aspire to deliver the premise of being declarative—the developer specifies *what* end result is desired, and not *how* it is accomplished efficiently. In particular, we would like the system to automatically detect the ability to split and distribute. This ability may be crucial for the developer (e.g., data scientist) who often lacks the expertise in software and hardware engineering. In this paper, we embark on a principled exploration of automated inference of *split-correctness* for information extractors. That is, we explore the ability of a system to detect whether an IE function can be applied separately to the individual segments of a given splitter, *without changing the semantics*.

The basic motivation comes from the scenario where a long document is pre-split by some conventional splitters (as the above), and developers provide different IE functions. If the system detects that the provided IE function is correctly splittable, then it can utilize its multi-processor or distributed hardware to parallelize the computation. Moreover, the system can detect that IE programs are frequently splittable, and recommend the system administrator to materialize splitters upfront. Even more, the split guarantee facilitates *incremental maintenance*: when a large document undergoes a minor edit, like in the Wikipedia model, only the relevant segments (e.g., sentences or paragraphs) need to reprocessed. Later in this section, we discuss additional motivations to split-correctness.

Formal framework. Our framework adopts the formalism of *document spanners* (or just *spanners* for short) [7]. In this framework, we consider *documents* (strings) over a fixed finite alphabet. A spanner is associated with a relational schema, and it extracts from every input document a relation of intervals within the document. An interval, called *span*, is represented simply by its starting and ending indices in the document. An example of a spanner is a *regex formula*—a regular expression with capture variables that correspond to the relational attributes. The most studied spanner language is that of the *regular* spanners, which is the closure of regex formulas under a subset of relational algebra: projection, natural join, union, and difference [7].[1] Other equally expressive formalisms are non-recursive Datalog over regex formulas [8] and the *variable-set automaton* or (or VSet-automaton for short), which is an NFA that can open and close variables while running [7].

Our framework is based on the following formal concepts. A *splitter* is a spanner S that outputs a *set of intervals* (e.g., sentences, paragraphs, N-grams, HTTP requests, etc.). A spanner P is *self-splittable* by a splitter S if for all documents d, evaluating P on d gives the same result as the union of the evaluations of P on each of the chunks produced by S. We also consider the more general case where we allow the spanner on the chunks produced by S to be some spanner P_S different from P. In this case, we say that *P is splittable by S via P_S*. If, for a given P and S, such a spanner P_S exists, then we say that P is *splittable* by S. With these definitions, we formally define several computational problems, each parameterized by a class $\mathcal{C}$ of spanners. In the *split-correctness* problem, we are given P, S, and P_S, and the goal is to determine whether P is splittable by S via P_S. In the *splittability* (resp. *self-splittability*) problem, we are given P and S and the goal is to determine whether P is splittable (resp. *self-splittable*) by S.

In our preliminary analysis, we consider the classes of regex formulas and VSet-automata, as well as VSet-automata in known normal forms, namely *functional* and *deterministic*. As we discuss later on, we use a slightly stronger definition of determinism than [24].

We show several complexity results about the studied classes of spanners. For one, the problems Split-correctness and Self-splittability are PSPACE-complete for regex formulas and VSet-automata. For Splittability, we require an additional condition for PSPACE-completeness, namely that the splitter is *disjoint*. This is a natural property of splitters S, meaning that for all input documents, the spans produced by S are pairwise disjoint (non-overlapping), such as in the case of tokenization, sentence boundary detection, paragraph splitting, and paragraph segmentation. Examples of *non-disjoint* splitters include N-grams and pairs of consecutive sentences. It turns out that disjointness is also a useful condition for Split-correctness and Self-splittability. For VSet-automata that are both functional and deterministic, Split-correctness and Self-splittability are solvable in polynomial time if the splitter is disjoint.

Interestingly, to establish the tractability result, we needed to revisit past notions and findings regarding determinism in VSet-automata. Specifically, our notion of determinism is *stronger* than that of Maturana et al. [24] (without loss of expressive power). We require that whenever the VSet-automata handle multiple variables on the same position of

[1] Adding selection would result in *Core-Spanners*, which are more powerful.

the document, it does so in a predefined order on the variables. This requirement is crucial, since our tractability proof uses the fact that containment of functional and deterministic VSet-automata is solvable in polynomial time (and, in fact, in NL), which we prove in Section 4. In contrast, we show that with the determinism of Maturana et al. [24], containment is PSPACE-complete. As we explain in Section 4.3, this stands in contradiction to a claim they make about membership in coNP. Our notion of deterministic VSet-automata is similar to the extended deterministic VSet-automata by Florenzano et al. [9]. Note that these results are of independent interest.

Following our analysis of split-correctness and splittability for regular spanners, we turn to discussing additional problems that arise in our framework. In Section 6, we discuss basic *reasoning* tasks about spanners. For example, if we wish to materialize two splitters, can we evaluate one of them over the result of the other, possibly given that documents adhere to some regular language? As an example, if we wish to split by sentences and documents are already split by paragraphs, then we can parallelize the task by splitting each paragraph individually. Similarly, an K-gram extractor can be applied to the chunks of an N-gram extractor whenever $K \leq N$.

In Section 7, we discuss problems that arise by natural extensions of the basic framework. One of these problems captures the case where some of the spanners in the query are treated as *black boxes* in a formalism that we do not understand well enough to analyze (as opposed to, e.g., regex formulas), and yet, are known to be splittable by the splitters at hand. For example, a coreference resolver may be implemented as a decision tree over a multitude of features [32] but still be splittable by sequences of three sentences, and a NER and a POS tagger may be implemented by a bidirectional LSTM [6] and a bidirectional dependency network [34], respectively, but still be splittable by sentences. Additional problems we discuss are split-correctness and splittability under the assumption that the document conforms to a regular language.

Our framework can be seen as an extension of the *parallel-correctness* framework as proposed by Ameloot et al. [2, 3]. That work considers the parallel evaluation of relational queries. In our terms, that work studies self-splittability where spanners are replaced by relational queries and splitters by *distribution policies*.

Further motivation. Besides the obvious, there are additional, perhaps less straightforward, motivations. For one, even if the document is *not* split at evaluation time (as opposed to pre-split), this split allows to parallelize the evaluation *following* a sequential split. When the IE function is expensive, this can be quite beneficial. For example, we have extracted N-grams from 1.53 GB Wikipedia sentences and observed that this method (first split to sentences and then distribute) gives a runtime improvement of 2.1x for $N = 2$ and 3.11x for $N = 3$, all over 5 cores. In a similar experiment on 279 MB of PubMed[2] sentences, the speedup was 1.9x.

Another motivation comes from programming over distribution frameworks such as Apache Hadoop [14] and Apache Spark [35]. In common cases, the text is already given as a collection of small documents (e.g., tweets, reviews, abstracts) that allow for a parallel evaluation to begin with. While we have not seen this scenario as a motivation for our framework, it turns out that splitting can make a considerable difference even then. For illustration, we ran a simple event extractor of financial transactions between organizations from sentences of around 9,000 Reuters articles over Spark. When we broke each article into sentences, the running time reduced by 1.99x on a 5-node cluster. We ran a similar experiment on sentences of around 570,000 reviews from the Amazon Fine Food Reviews dataset,[3] where the goal is to extract targets of a negative sentiment; we observed a 4.16x speedup. We found this remarkable, because the same amount of parallelization was used both before and after splitting. To the best of our understanding, this improvement can be explained by the fact that splitting provides Spark with parallelizable tasks that are smaller in cost and larger in number; hence, we provide Spark with considerably more (smartly exploited) control over scheduling and resource allocation.[4]

Finally, another motivation comes from *debugging*. For illustration, suppose that the developer seeks HTTP requests to a specific host on a specific date, and for that she seeks Host and Date headers that are close to each other; the system can warn the developer that the program is not splittable by HTTP requests like other frequent programs over the log (i.e., it can extract the Host of one request along with the Date of another), which is indeed a bug in this case. In the general case, the system can provide the user with the different splitters (sentences, paragraphs, requests, etc.) that the program is split-correct for, in contrast with what the developer believes should hold true.

Organization. The remainder of the paper is organized as follows. In Section 2, we give preliminary definitions and notation. We define the concepts of splitter, split-correctness and splittability in Section 3, and present our analysis for regular spanners in Sections 4 and 5. We discuss the extensions of the framework to other problems in Section 6 and 7. Finally, we conclude and discuss open problems in Section 8. Due to space constraints, we sometimes omit proofs or only provide a proof sketch.

[2] https://www.ncbi.nlm.nih.gov/pubmed/

[3] https://www.kaggle.com/snap/amazon-fine-food-reviews

[4] See [25] for more detail on the experiments.

2 BASIC DEFINITIONS

Our framework is within the formalism of *document spanners* by Fagin et al. [7]. We first revisit some definitions from this framework. Let Σ be a finite set of symbols called the *alphabet*. By Σ^* we denote the set of all finite strings over Σ and by Σ^+ the set of all finite strings over Σ that have at least one symbol. A string s in Σ^* is also called a *document*.

Let $d = \sigma_1 \cdots \sigma_n \in \Sigma^*$ be a document, where every $\sigma_i \in \Sigma$. We denote by $|d|$ the length n of d. A *span* of d is an expression of the form $[i, j\rangle$ with $1 \leq i \leq j \leq n+1$. For a span $[i, j\rangle$ of d, we denote by $d_{[i,j\rangle}$ the string $\sigma_i \cdots \sigma_{j-1}$. For a document d, we denote by *Spans(d)* the set of all possible spans of d. Two spans $[i_1, j_1\rangle$ and $[i_2, j_2\rangle$ are *equal* if $i_1 = i_2$ and $j_1 = j_2$. In particular, $d_{[i_1,j_1\rangle} = d_{[i_2,j_2\rangle}$ does not imply that $[i_1, j_1\rangle = [i_2, j_2\rangle$. Two spans $[i, j\rangle$ and $[i', j'\rangle$ *overlap* if $i \leq i' < j$ or $i' \leq i < j'$, and are *disjoint* otherwise. Finally, $[i, j\rangle$ *contains* $[i', j'\rangle$ if $i \leq i' \leq j' \leq j$.

The framework focuses on functions that extract spans from documents and assigns them to variables. To this end, we fix an infinite set SVars of *span variables*, which range over spans, i.e., pairs of integers. The sets Σ and SVars are disjoint. For a finite set $V \subseteq$ SVars of variables and a document $d \in \Sigma^*$, a (V, d)-*tuple* is a mapping $t : V \to \text{Spans}(d)$ that assigns a span of d to each variable. If V is clear from context, or V is irrelevant, we may also just write *"d-tuple."* A set of (V, d)-tuples is called a (V, d)-*relation*, which is also called a *span-relation (over d)*.

A *document spanner* (or *spanner* for short) is a function that transforms a document into a span relation. More formally, a spanner is a function $P : \Sigma^* \to \mathcal{S}$ where Σ^* is the set of documents and $\mathcal{S}$ the set of span relations. A spanner is always associated with a finite set $V \in$ SVars and maps each document d to a (V, d)-relation $P(d)$. By SVars(P) we denote the set V. We say that a spanner P is *n-ary* if $|\text{SVars}(P)| = n$. We denote by $P = P'$ the fact that the spanners P and P' define the same function.

3 SPLITTERS AND MAIN PROBLEMS

In this work, we are particularly interested in spanners that split documents into (possibly overlapping) segments. Formally, a *document splitter* (or *splitter* for short) is a unary document spanner P, that is, $|\text{SVars}(P)| = 1$. Referring back to the Introduction, a splitter can split the document into paragraphs, sentences, N-grams, HTTP messages, error messages, and so on.

In the sequel, we denote a splitter by S and its unique variable by x_S or simply by x if S is clear from the context. Furthermore, since a splitter outputs unary span relations, its output on a document d can be identified with the set of spans $\{t(x) \mid t \in S(d)\}$. We often use this simplified view on a splitter in the paper and treat its output as a set of spans.

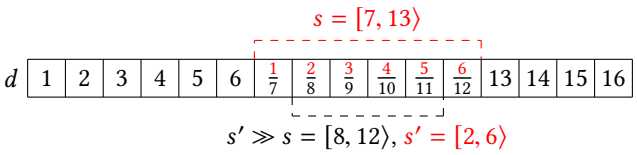

Figure 1: Visualization of the shift span operator, with $[8, 12\rangle = [2, 6\rangle \gg [7, 13\rangle$.

A splitter S is *disjoint* if the spans extracted by S are always pairwise disjoint, that is, for all $d \in \Sigma^*$ and $t, t' \in S(d)$, the spans $t(x)$ and $t'(x)$ are disjoint. For example, the paragraph and sentence splitters are disjoint, but N-gram extractors are not disjoint for $N > 1$.

Next, we want to define when a spanner is *splittable* by a splitter, that is, when documents can be decomposed such that the operation of a spanner can be distributed over the components. To this end, we first need some notation. Let d be a document, let $s = [i, j\rangle$ be a span of d, and let $s' = [i', j'\rangle$ be a span of the document $d_{[i,j\rangle}$. Then s' also marks a span of the original document d, namely the one obtained from s' by shifting it $i - 1$ characters to the right. We denote this shifted span by $s' \gg s$ (c.f., Figure 1). Hence, we have:

$$s' \gg s := [i' + (i-1), j' + (i-1)\rangle.$$

Let t be a (V, d)-tuple and $s = [i, j\rangle$ be a span. Slightly overloading notation, we define the (V, d)-tuple $t \gg s$ as the tuple that results from shifting each span in t by s. More formally, for all variables $x \in V$ we have:[5]

$$(t \gg s)(x) := (t(x)) \gg s.$$

As a splitter S always selects a set of unary tuples, in the following we abuse notation, and simply write s rather than $s(x_S)$ when $s \in S(d)$ for some document d.

We now define the *composition* $P \circ S$ of a spanner P and splitter S. Intuitively, $P \circ S$ is the spanner that results from evaluating P on every substring extracted by S, with a proper indentation of the indices. More formally, on every document d,

$$(P \circ S)(d) := \bigcup_{s \in S(d)} \{t \gg s \mid t \in P(d_s)\}.$$

As an example, if P extracts person names and S is a sentence splitter, then $P \circ S$ is the spanner obtained by applying P to every sentence independently and taking the union of the results. Furthermore, if P extracts close mentions of email addresses and phone numbers, and S is the 5-gram splitter, then $P \circ S$ is obtained by applying P to each 5-gram individually. An interesting question is if there is any difference between executing P and $P \circ S$. This property clearly depends

[5]Notice that when the first index of t is too large, $t \gg s$ could technically not be a (V, d)-relation anymore. However, we only use the operator in situations where this can never happen.

on the definitions of P and S. We will define it formally in Section 3.1 under the name *self-splittability*.

3.1 Splittability and Split-Correctness

A spanner P is *splittable* by a splitter S via a spanner P_S if evaluating P gives the same result as evaluating P_S on every substring extracted by S (again with proper indentation of the indices). If such a P_S exists, then we say that P is *splittable* by S; and if P_S is P itself, then we say that P is *self-splittable* by S. We define these notions more formally.

Definition 3.1. Let P be a spanner and S a splitter. We say that:

(1) P is *splittable by S via* a spanner P_S, if $P = P_S \circ S$;
(2) P is *splittable by S* if there exists a spanner P_S such that $P = P_S \circ S$;
(3) P is *self-splittable by S* if $P = P \circ S$.

We refer to P_S as the *split-spanner*.

As a simple example, suppose that we analyze a log of many HTTP requests separated by blank lines and assume for simplicity that the log only consists of GET requests. Furthermore, assume that S splits the document into individual requests (without the blank lines) and that P extracts the request line, which is always the first line of the request. If P identifies the request line as the one following the blank line, then P is splittable by S via P_S, which is the same as P but replaces the requirement to follow a blank line with the requirement of being the first line. If, on the other hand, P identifies the request line as being the one starting with the word GET, then P is self-splittable by S, since we can apply P itself to every HTTP message independently.

Other examples are as follows. Many spanners P that extract person names do not look beyond the sentence level. This means that, if S splits to sentences, it is the case that P is self-splittable by S. Now suppose that P extracts mentions of email addresses and phone numbers based on the formats of the tokens, and moreover, it allows at most three tokens in between; if S is the N-gram splitter, then P is self-splittable by S for $N \geq 5$ but not for $N < 5$.

3.2 Decision Problems

The previous definitions and the motivating examples from the introduction directly lead to the corresponding decision problems. We use C to denote a class of spanner representations (such as **VSA** or **RGX** that we define later on).

Split-correctness[C]	
Input:	Spanners $P, P_S \in C$ and splitter $S \in C$.
Question:	Is $P = P_S \circ S$?

Splittability[C]	
Input:	Spanner $P \in C$ and splitter $S \in C$.
Question:	Is P splittable by S?

In addition, Self-splittability[C] is the special case of the problem Splittability[C] where we ask if P is *self-splittable* by S.

4 PRELIMINARIES ON REGULAR SPANNERS

In this section, we recall the terminology and definition of *regular* spanners [7]. We use two main models for representing spanners: *regex-formulas* and *VSet-automata*. For both, we follow Freydenberger [10], defining the semantics of these models using so-called *ref-words*. We also introduce here a class of VSet-automata, **dfVSA**, that have determinism properties essential to the tractability of problems we study in the paper. In particular, we show that containment of regex formulas and VSet-automata is PSPACE-complete (Theorem 4.1), even under some determinism assumptions introduced in the past work [24] (Theorem 4.2), but is it solvable in polynomial time, and even NL, for **dfVSA** (Theorem 4.3).

Ref-words. For a finite set $V \subseteq \mathrm{SVars}$ of variables, ref-words are defined over the extended alphabet $\Sigma \cup \Gamma_V$, where $\Gamma_V := \{x\vdash, \dashv x \mid x \in V\}$. We assume that Γ_V is disjoint with Σ and SVars. Ref-words extend strings over Σ by encoding opening ($x\vdash$) and closing ($\dashv x$) of variables.

A ref-word $\mathbf{r} \in (\Sigma \cup \Gamma_V)^*$ is *valid for V* if each variable is opened and closed exactly once. More formally, for each $x \in V$, the string $\mathbf{r}$ has exactly one occurrence of $x\vdash$ and precisely one occurrence of $\dashv x$, which is after the occurrence of $x\vdash$. If V is clear from the context, we simply say that a ref-word is *valid*.

To connect ref-words to documents and spanners, we define a morphism $\mathrm{clr} : (\Sigma \cup \Gamma_V)^* \to \Sigma^*$ (pronounced "clear"), as $\mathrm{clr}(\sigma) := \sigma$ for every $\sigma \in \Sigma$ and $\mathrm{clr}(\sigma') := \varepsilon$ for every $\sigma' \in \Gamma_V$. For $d \in \Sigma^*$, let $\mathrm{Ref}(d) := \{\mathbf{r} \in (\Sigma \cup \Gamma_V)^* \mid \mathrm{clr}(\mathbf{r}) = d \text{ and } \mathbf{r} \text{ is valid}\}$ be the set of all valid ref-words with $\mathrm{clr}(\mathbf{r}) = d$ and for a regular language $\mathcal{L}$ we define $\mathrm{Ref}(\mathcal{L}) := \bigcup_{d \in \mathcal{L}} \mathrm{Ref}(d)$ to be the language of all valid ref-words over $\mathcal{L}$.

By definition, every valid $\mathbf{r} \in \mathrm{Ref}(d)$ over $(\Sigma \cup \Gamma_V)$ has a unique factorization $\mathbf{r} = \mathbf{r}_x^{\mathrm{pre}} \cdot x\vdash \cdot \mathbf{r}_x \cdot \dashv x \cdot \mathbf{r}_x^{\mathrm{post}}$ for each $x \in V$. We can therefore interpret $\mathbf{r}$ as a (V, d)-tuple $t^{\mathbf{r}}$ by defining $t^{\mathbf{r}}(x) := [i, j\rangle$, where $i := |\mathrm{clr}(\mathbf{r}_x^{\mathrm{pre}})| + 1$ and $j := i + |\mathrm{clr}(\mathbf{r}_x)|$.

4.1 Regex Formulas

A *regex-formula (over Σ)* is a regular expression that may also include variables (called capture variables). Formally, we define the syntax with the recursive rule

$$\alpha := \emptyset \mid \varepsilon \mid \sigma \mid (\alpha \vee \alpha) \mid (\alpha \cdot \alpha) \mid \alpha^* \mid x\{\alpha\}\,,$$

where $\sigma \in \Sigma$ and $x \in V$. We use α^+ as a shorthand for $\alpha \cdot \alpha^*$ and Σ as a shorthand for $\bigvee_{\sigma \in \Sigma} \sigma$. The set of variables that occur in α is denoted by $\mathrm{SVars}(\alpha)$ and the size $|\alpha|$ is defined as the number of symbols in α.

Every regex-formula can be interpreted as a generator of a (regular) ref-word language $\mathcal{R}(\alpha)$ over the extended alphabet $\Sigma \cup \Gamma_{\mathrm{SVars}(\alpha)}$. If α is of the form $x\{\beta\}$, then $\mathcal{R}(\alpha) := \{x\vdash\} \cdot \mathcal{R}(\beta) \cdot \{\dashv x\}$. Otherwise, $\mathcal{R}(\alpha)$ is defined as the language $\mathcal{L}(\alpha)$, that is $\mathcal{R}(\emptyset) := \emptyset$, $\mathcal{R}(a) := \{a\}$ for every $a \in \Sigma \cup \{\varepsilon\}$, $\mathcal{R}(\alpha \vee \beta) := \mathcal{R}(\alpha) \cup \mathcal{R}(\beta)$, $\mathcal{R}(\alpha \cdot \beta) := \mathcal{R}(\alpha) \cdot \mathcal{R}(\beta)$, $\mathcal{R}(\alpha^*) := \{\mathcal{R}(\alpha)^i \mid i \geq 0\}$.

By $\mathrm{Ref}(\alpha)$ we denote the set of all ref-words in $\mathcal{R}(\alpha)$ which are valid for $\mathrm{SVars}(\alpha)$;[6] and, for every string $d \in \Sigma^*$, we define $\mathrm{Ref}(\alpha, d) := \mathrm{Ref}(\alpha) \cap \mathrm{Ref}(d)$. In other words, $\mathrm{Ref}(\alpha, d)$ contains exactly those valid ref-words from $\mathrm{Ref}(\alpha)$ that clr maps to d. Finally, the spanner $[\![\alpha]\!]$ is the one that defines the following $(\mathrm{SVars}(\alpha), d)$-relation for every string $d \in \Sigma^*$:

$$[\![\alpha]\!](d) := \{t^{\mathbf{r}} \mid \mathbf{r} \in \mathrm{Ref}(\alpha, d)\}$$

Slightly abusing notation, we will sometimes simply write $\alpha(d)$ rather than $[\![\alpha]\!](d)$ to denote the span-relation α defines on d. We say that a regex-formula is *functional* if $\mathcal{R}(\alpha) = \mathrm{Ref}(\alpha)$; that is, every ref-word in $\mathcal{R}(\alpha)$ is valid. The set of all functional regex formulas is also denoted by **RGX**. Following previous work [7, 12], we assume regex formulas are functional unless explicitly stated otherwise.

4.2 Variable Set-Automata

A *variable-set automaton (VSet-automaton)* with variables from a finite set $V \subseteq \mathrm{SVars}$ can be understood as an ε-NFA that is extended with edges that are labeled with variable operations Γ_V. Formally, a VSet-automaton is a sextuple $A := (\Sigma, V, Q, q_0, Q_F, \delta)$, where Σ is a finite set of alphabet symbols, V is a finite set of variables, Q is a finite set of states, $q_0 \in Q$ is a start state, $Q_F \subseteq Q$ is a set of final states, and $\delta : Q \times (\Sigma \cup \{\varepsilon\} \cup \Gamma_V) \to 2^Q$ is the transition function. By $\mathrm{SVars}(A)$ we denote the set V. To define the semantics of A, we first interpret A as an ε-NFA over the terminal alphabet $\Sigma \cup \Gamma_V$, and define its *ref-word language* $\mathcal{R}(A)$ as the set of all ref-words $\mathbf{r} \in (\Sigma \cup \Gamma_V)^*$ that are accepted by the ε-NFA A. Analogously to regex formulas, we define $\mathrm{Ref}(A)$ as the set of ref-words in $\mathcal{R}(A)$ that are valid for V, and define $\mathrm{Ref}(A, d)$ and $[\![A]\!](d)$ accordingly for every $d \in \Sigma^*$. We say that A is *functional* if $\mathrm{Ref}(A) = \mathcal{R}(A)$, i.e., every accepting run of A generates a valid ref-word. Furthermore, two VSet-automata A_1, A_2 are *equivalent* if and only if $[\![A_1]\!] = [\![A_2]\!]$.

[6] Notice that not all ref-words in a ref-word language have to be valid. For instance, the ref-word $\varepsilon \in \mathcal{R}((x\{a\})^*)$ is not valid for $\mathrm{SVars}((x\{a\})^*) = \{x\}$, since it does not contain $x\vdash$ and $\dashv x$.

We refer to the set of all VSet-automata as **VSA**. Similar to regex formulas, we sometimes simply denote the relation $[\![A]\!](d)$ by $A(d)$, for a VSet automaton A.

Deterministic VSet-Automata. We use the notion of determinism for VSet-automata as introduced by [24], but refer to it as *weakly* deterministic because, as we will show, it still allows for sufficient nondeterminism to make reasoning equally hard than for general VSet-automata. Formally, a VSet-automaton $A = (\Sigma, V, Q, q_0, Q_F, \delta)$ is *weakly deterministic*, if it does not use ε-transitions and $|\delta(q, v)| \leq 1$ for every $q \in Q$ and every $v \in \Sigma \cup \Gamma_V$.

In Theorem 4.2 we show that weakly deterministic VSet-automata still have sufficient nondeterminism to make the containment problem PSPACE-hard.[7] We therefore define a stronger notion of determinism, which will lead to an NL-complete containment problem (Theorem 4.3).

We fix a total, linear order $\prec$ on the set Γ_{SVars} of variable operations, such that $v\vdash \prec \dashv v$ for every variable v. A VSet-automaton $A = (\Sigma, V, Q, q_0, Q_f, \delta)$ is *deterministic*, if

(1) $|\delta(q, v)| \leq 1$ for every $q \in Q$ and every $v \in \Sigma \cup \Gamma_V$;
(2) $v \prec v'$ for every $v, v' \in \Gamma_V$ for which there are $q_1, q_2, q_3 \in Q$ such that $\delta(q_1, v) = q_2$ and $\delta(q_2, v') = q_3$.

Condition (1) is the requirement for weakly deterministic automata, whereas condition (2) ensures that for every document $d \in \Sigma^*$ and every tuple $t \in A(d)$ there is exactly one ref-word $\mathbf{r} \in \mathrm{Ref}(A)$ with $t^{\mathbf{r}} = t$.[8] Moreover, all adjacent variable operations in $\mathbf{r}$ are ordered according to $\prec$. We discuss expressiveness and complexity of deterministic VSet-automata in Section 4.3. In particular, none of the conditions (1–2) restrict the expressiveness of regular spanners. In the following, we denote by **dVSA** (resp., **dfVSA**) the class of deterministic (resp., deterministic and functional) VSet-automata.

4.3 Complexity and Expressiveness

Containment is the problem that asks, given VSet-automata A and A', whether $A(d) \subseteq A'(d)$ for every document d. The next theorem establishes the complexity of containment. We note that the VSet-automata are not required to be functional. The following is essentially [24, Theorem 6.4].

Theorem 4.1. *Containment of regex-formulas (***RGX***) and VSet-automata (***VSA***) is PSPACE-complete.*

Since weakly deterministic VSet-automata restrict the transition function to a singleton set, which is a standard way

[7] This result contradicts Theorem 6.6 in Maturana et al. [24] unless coNP = PSPACE. We will further discuss this in Section 4.3.

[8] Deterministic VSet-automata are very similar to the *extended deterministic VSet-automata* by Florenzano et al. [9], which allow multiple variable operations on a single transition and force each variable transition to be followed by a transition processing an alphabet symbol.

to define determinism, one may expect their containment problem to be in NL (just as for their finite automata counterparts). However, the next theorem shows that containment is hard for PSPACE, as they can use different variable orderings to introduce nondeterministic choice.

Theorem 4.2. *Containment of weakly deterministic functional VSet-automata is PSPACE-hard.*

Proof sketch. We reduce from the PSPACE complete problem of DFA union universality [16]. Given deterministic finite automata $A_1, \ldots, A_n$ over the alphabet Σ, the union universality problem asks whether

$$L(\Sigma^*) \subseteq \bigcup_{1 \leq i \leq n} L(A_i). \qquad (\dagger)$$

As usual, $L(A_i)$ denotes the language accepted by A_i. We construct VSet-automata A, A' using the variable set $V = \{x_1, \ldots, x_n\}$, such that $A(d) \subseteq A'(d)$ for all documents $d \in \Sigma^*$ if and only if $(\dagger)$ holds. To this end, let A accept the language defined by the regex-formula

$$\alpha_A = x_1\{x_2\{\cdots x_n\{\Sigma^*\}\cdots\}\},$$

selecting the whole document with every variable. Clearly, the regex-formula α_A can be represented by a weakly deterministic VSet-automaton A. We now abuse notation and describe the language accepted by A' by a hybrid regex-formula

$$\alpha_{A'} = x_1\{\alpha_1\} + \cdots + x_n\{\alpha_n\},$$

where the DFAs A_i are plugged in. In particular,

$$\alpha_i = x_1\{\cdots x_{i-1}\{x_{i+1}\{\cdots\{x_n\{L(A_i)\}\}\cdots\},$$

for $1 \leq i \leq n$. Every branch i starts by first opening variable x_i, continues to open all other variables in increasing order, and finally selects for every variable the whole document when it is accepted by A_i. Clearly, as every disjunct starts with a different symbol, this hybrid formula can be transformed into an equivalent weakly deterministic VSet-automaton. It is easy to verify that $A(d) \subseteq A'(d)$ for every document $d \in \Sigma^*$ if and only if $(\dagger)$ holds. □

The previous theorem contradicts Theorem 6.6 in Maturana et al. [24] (unless coNP = PSPACE), where it is argued that containment for sequential weakly deterministic automata is in coNP (even allowing partial mappings to variables).[9] As the VSet-automata we consider are indeed sequential, we show that the latter problem is hard for PSPACE (already without allowing partial mappings). Our definition of determinism resolves this complexity issue.

[9]The error is in the upper bound of Maturana et al. [24], as can be seen in the version that includes the proofs [23]. The specific error is in the pumping argument for proving a polynomial size witness property for non-containment. The polynomial size witness property is not necessarily true, due to the nondeterminism entailed in the ability of the automaton to open variables in different orders. At any specific position in the string, the execution can be in $\Theta(n)$ possible states, where n is the number of states, implying that a minimal witness may require a length of $2^{\Theta(n)}$.

Theorem 4.3. *Containment for* **dfVSA** *is in NL.*

The following proposition shows that deterministic VSet-automata are equally expressive as VSet-automata in general.

Proposition 4.4. *For every VSet-automaton A there is a deterministic VSet-automaton A' such that $A(d) = A'(d)$ for every document $d \in \Sigma^*$. This property still holds if A' is additionally required to be functional.*

Finally, we recall that it is well known that **RGX** is less expressive than VSet-automata [7]. To reach the expressiveness of VSet-automata, **RGX** needs to be extended with projection, natural join, union, and difference.

5 SPLITTING WITH REGULAR SPANNERS

We now give preliminary results for the decision problems we introduced in Section 3 in the case of regular spanners. We consider split-correctness in Section 5.1, and discuss splittability and self-splittability in Sections 5.2 and 5.3, respectively. In particular, we show that split-correctness is PSPACE-complete for all spanners formalisms that we discuss in the paper (Theorem 5.1); nevertheless, if we make the natural assumption of *disjointness* of the splitter, then the complexity reduces to polynomial time for **dfVSA** (Theorem 5.7). For splittability, we show a PSPACE-completeness result under the assumption of splitter disjointness (Theorem 5.15), while membership in PSPACE for general splitters remains an open question. For self-splittability, we prove PSPACE-completeness (Theorem 5.16) in general, and again, that disjointness reduces the complexity to polynomial time for **dfVSA** (Theorem 5.17).

5.1 Split-Correctness

5.1.1 General cases. The following theorem states the complexity of Split-correctness for the classes defined in Section 4.

Theorem 5.1. *Split-correctness*$[C]$ *is PSPACE-complete for each of the classes* $C \in \{$**RGX**, **VSA**, **dVSA**, **dfVSA**$\}$.

Proof sketch. For the lower bound, we need to prove hardness separately for **dfVSA** and for **RGX**, since the models are incomparable (**dfVSA** are more expressive; **RGX** have more non-determinism). For the lower bound for **dfVSA**, we reduce from the PSPACE-complete problem of DFA union universality [16]. To this end, let $A_1, \ldots, A_n$ be regular languages, given as deterministic finite automata over the alphabet Σ and let $\Sigma' = \Sigma \uplus \{a\}$. Let $P = a^n \cdot y\{\Sigma^*\}$ and

$S = x\{a^n \cdot A_1\} + a \cdot x\{a^{n-1} \cdot A_2\} + \cdots + a^{n-1} \cdot x\{a \cdot A_n\}$. Furthermore, let $P_S = a^* \cdot z\{\Sigma^*\}$. Then $P = P_S \circ S$ if and only if $L(\Sigma^*) \subseteq \bigcup_{1 \leq i \leq n} L(A_i)$.

The lower bound for **RGX** is immediate by a reduction from the **RGX** containment problem, which is PSPACE-hard (c.f. Theorem 4.1). Indeed, let $P, P_S \in$ **RGX** be spanners and $S = x\{\Sigma^*\} \in$ **RGX** be a splitter. Then P is splittable by S via P_S if and only if $P \subseteq P_S$ and $P \supseteq P_S$.

For the upper bound, one can construct a spanner P' such that $P = P_S \circ S$ if and only if $P = P'$. (For readers familiar with spanner algebra, P' is the spanner $\pi_{\mathrm{SVars}(P)}((\Sigma^* \cdot x\{P_S\} \cdot \Sigma^*) \bowtie S)$.) Furthermore, P' can be constructed in polynomial time and therefore the upper bound follows from the fact that equivalence between spanners can be tested in PSPACE (Theorem 4.1). □

5.1.2 The cover condition. In this section, we introduce a necessary condition for splittability, called the *cover condition*. We show that it is PSPACE-complete to check if the condition holds for **RGX**, **VSA**, **dVSA**, and **dfVSA**. In Section 5.1.3, we leverage the condition to obtain a tractable split-correctness result. Furthermore, we use the cover condition as a part of the splittability condition to obtain a characterization for splittability in Section 5.2.

Definition 5.2 (Cover Condition)**.** A spanner P and splitter S satisfy the *cover condition* if, for every document d and every tuple $t \in P(d)$, there exists a span $s \in S(d)$ that contains every span in t. That is, s contains $t(v)$ for every $v \in \mathrm{SVars}(P)$. In this case, we also say that s *covers* t.

The cover condition is indeed necessary for splittability.

Lemma 5.3. *For a spanner P and a splitter S, if P is splittable by S, then P and S satisfy the cover condition.*

The following lemma determines the complexity of testing the cover condition.

Lemma 5.4. *Let P be a spanner and S be a splitter, both coming from one of the classes* **RGX**, **VSA**, **dVSA**, *or* **dfVSA**. *Then it is PSPACE-complete to check whether P and S satisfy the cover condition.*

Proof sketch. We start with the PSPACE upper bound. Let P be a spanner and let S be a splitter both in **VSA**. Let $V = \mathrm{SVars}(P)$. We assume $x_S \notin V$. We define a spanner P_V that selects every possible span in every variable. More formally: $P_V = (\Sigma, V, \{q_0\}, q_0, \{q_0\}, \delta)$, where $\delta = \{(q_0, c, q_0) \mid c \in \Sigma \cup \Gamma_V\}$. We argue next that the cover condition holds if and only if $P \subseteq P_V \circ S$. The PSPACE upper bound then follows from Theorem 4.1.

(if): Assume that the cover condition does not hold. Then there is a document $d \in \Sigma^*$ and a tuple $t \in P(d)$, such that there is no split $s \in S(d)$ which covers t. Therefore, $t \notin P_V \circ S$.

(only if): Assume that the cover condition holds. Let $d \in \Sigma^*$ be a document and $t \in P(d)$ be a tuple. Due to the cover condition, there is a split $s \in S(d)$ which covers t. Thus, per definition of P_V, there is a tuple $t' \in P_V$, such that $t = t' \gg s$. Therefore $P \subseteq P_V \circ S$ also holds.

For the lower bounds, we reduce from DFA union universality and regular expression union universality, which are both PSPACE-complete [16, 21]. We prove the lower bound for **dfVSA**. (The proof for **RGX** is analogous.) Let $A_1, \ldots, A_n$ be regular languages, given as deterministic finite automata over the alphabet Σ.

Let $\Sigma' = \Sigma \uplus \{a\}$. Furthermore, let $P = a^n \cdot y\{\Sigma^*\}$ be a spanner and $S = x\{a^n \cdot A_1\} + a \cdot x\{a^{n-1} \cdot A_2\} + \cdots + a^{n-1} \cdot x\{a \cdot A_n\}$ be a splitter. It is easy to see that P and S can be represented as deterministic functional VSet-automata whose size is quadratic in the input. For every document $d \in \Sigma'^*$ and every tuple $t \in P(d)$ there is a split $s \in S(d)$ that covers t if and only if $L(\Sigma^*) \subseteq \bigcup_{1 \leq i \leq n} L(A_i)$. □

5.1.3 Tractability for disjoint splitters. We now show that split-correctness for deterministic functional VSet-automata is decidable in polynomial time in the case where splitters are *disjoint*. The next proposition shows that deciding whether a splitter is disjoint is tractable.

Proposition 5.5. *For every splitter $S \in C$ it can be checked in NL whether S is disjoint, if C in* {**VSA**, **dVSA**, **dfVSA**}. *Furthermore, for $S \in$* **RGX**, *disjointness can be checked in PTIME.*

Recall from the Introduction that disjointness is a natural property that splitters often satisfy in real life (e.g., tokenization, sentence boundary detection, paragraph splitting and segmentation). Technically, we rely on a polynomial-time algorithm to test the cover condition for deterministic functional automata.

Lemma 5.6. *Let P be a spanner and S a disjoint splitter, both from* **dfVSA**. *Whether P and S satisfy the cover condition can be decided in polynomial time.*

Proof sketch. We reduce this problem to the containment problem of unambiguous finite automata, which can be solved in polynomial time [33]. Essentially we construct unambiguous automata A_P and A_S that accept *ref-words over an expanded alphabet*. That is, A_P accepts words $\mathbf{r}' = (\sigma_1, i_1) \cdots (\sigma_n, i_n)$ such that $\sigma_1 \cdots \sigma_n$ is a ref-word for P and $i_1 \cdots i_n \in 0^*1^+0^*$ is a bit sequence that indicates from where to where the ref-word uses its variables. The automaton A_S accepts precisely those words in the language of A_P that correspond to an output tuple that is covered by some span of S. (We need the disjointness of S for A_S to be unambiguous.) Therefore, $A_P \subseteq A_S$, if and only if for every output tuple of P, there is an output tuple of S that covers it. □

Theorem 5.7. *Let P, P_S be spanners and S be a disjoint splitter, all from* **dfVSA**. *Split-correctness for P, P_S, and S (i.e., whether $P = P_S \circ S$) is decidable in polynomial time.*

Proof Sketch. We explain how to decide the complement problem. We begin by checking the cover condition. If the cover condition is not satisfied, it follows by Lemma 5.3 that $P \neq P_S \circ S$. Using Lemma 5.6, this test can be done in polynomial time. Assuming that the cover condition holds, we look for a split s on which P and P_S behave differently. This can be done by guessing a ref-word $\mathbf{r} \in (\Sigma \cup \Gamma_{\mathrm{SVars}(P)} \cup \Gamma_{\{x_S\}})$, symbol by symbol and simulating P, S, and P_S on the fly. □

5.2 Splittability

In this section, we address the splittability problem. In particular, we show that Splittability[**VSA**] and Splittability[**RGX**] are PSPACE-complete for disjoint splitters.

Before we begin, it is useful to note the following insight about the splittability problem. When a spanner P is splittable by a splitter S, there can be different split-spanners P_S and P'_S witnessing splittability. This is illustrated next.

Example 5.8. Consider $P = ay\{b\}b$ and $S = x\{ab\}\, b + ax\{bb\}$. Then, both $P = P_S \circ S$ and $P = P'_S \circ S$ for $P_S = ay\{b\}$ and $P'_S = y\{b\}b$ but $P_S \neq P'_S$. The reason this happens is that S selects two different spans $s = [1,3\rangle$ and $s' = [2,4\rangle$ both containing the span $[2,3\rangle$ selected by P on abb. Since the selected spans are different, the split-spanners P_S and P'_S need to be different as well to be able to simulate P. Notice that S is not a disjoint splitter, as $[1,3\rangle \cap [2,4\rangle \neq \emptyset$. □

However, when S is disjoint and P is splittable by S, we show that there exists a *canonical* split-spanner P_S^{can} for which $P = P_S^{\mathrm{can}} \circ S$. In particular, for every document d, the canonical split-spanner selects a tuple t, if there is a larger document d' on which S selects d and P selects t on d', with proper indentation of the indexes. Formally, P_S^{can} is defined such that, for every document d,

$$P_S^{\mathrm{can}}(d) := \{t \mid \exists d' \in \Sigma^*, s \in S(d'), d'_s = d, \text{ and } (t \gg s) \in P(d')\}$$

In the following proposition, we prove that we can construct the canonical spanner in polynomial time. Essentially, P_S^{can} is a cross product between P and S, properly adjusted to use the same variables, operating on substrings of documents selected by S.

Proposition 5.9. *Let P be a spanner and S be a disjoint splitter, both coming from one of the classes* **RGX** *or* **VSA**. *Then a VSet-automaton can be constructed in time polynomial in the sizes of P and S that defines the spanner P_S^{can}.*

Proof sketch. Let $P = (\Sigma, V, Q_P, q_{0,P}, Q_{F,P}, \delta_P)$ be a VSet-automaton and let $S = (\Sigma, \{x\}, Q_S, q_{0,S}, Q_{F,S}, \delta_S)$ be a splitter, with $x \notin V$.

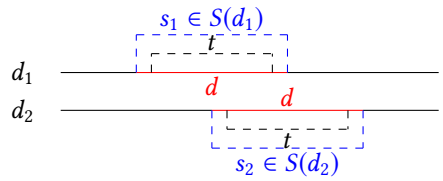

Figure 2: Visualization of the splittability condition (2).

By P^{Σ} we denote the VSet-automaton obtained from P by eliminating all Γ_V-transitions, that is, only keeping the Σ-transitions. For a symbol c, let $P \cdot_c P$ be the VSet-automaton consisting of two disjoint copies of P where every state of the first copy is connected to its corresponding state in the second copy by a transition labeled with c. Then define $P^x = P^{\Sigma} \cdot_{x\vdash} P \cdot_{\dashv x} P^{\Sigma}$. We denote by S^{+V} the VSet-automaton obtained from S by adding self-loops labeled with $v\vdash$ and $\dashv v$ to every state of S for $v \in V$. That is, for every state q of S, add the transitions $(q, v\vdash, q)$ and $(q, \dashv v, q)$ for every $v \in V$. Then, notice that both P^x and S^{+V} define ref-words over the same extended alphabet $\Sigma \cup \Gamma_{V \cup \{x\}}$. Moreover, for every document $d \in \Sigma^*$, the spanner $(P^x \cap S^{+V})$ defines precisely those tuples t where $t(x) \in S(d)$ and $t(v)$ is contained in $t(x)$ for every $v \in V$. Then, the spanner accepting the ref-word language $\{w_2 \mid w_1 \cdot x\vdash \cdot\, w_2 \cdot \dashv x \cdot w_3 \in \mathrm{Ref}(P^x \cap S^{+V})\}$ is exactly P_S^{can}. □

Notice that, due to Lemma 5.3, whenever P is splittable by S, $P \subseteq P_S^{\mathrm{can}} \circ S$. However, when S is not disjoint, the converse inclusion $P_S^{\mathrm{can}} \circ S \subseteq P$ does not always hold as the next example shows.

Example 5.10. Consider again P and S from Example 5.8 and let $d = abb$. Notice that $P(d) = \{[2,3\rangle\}$, and $P(d') = \emptyset$ for every $d' \neq d$. We now compute $(P_S^{\mathrm{can}} \circ S)(d)$. Since $S(d) = \{[1,3\rangle, [2,4\rangle\}$, we need to consider the documents $d_{[1,3\rangle} = ab$ and $d_{[2,4\rangle} = bb$ for the definition of P_S^{can}. We have $P_S^{\mathrm{can}}(ab) = \{[2,3\rangle\}$ and $P_S^{\mathrm{can}}(bb) = \{[1,2\rangle\}$. Finally,

$$\begin{aligned}(P_S^{\mathrm{can}} \circ S)(d) &= \bigcup_{s \in S(d)} \{t \gg s \mid t \in P_S^{\mathrm{can}}(d_s)\} \\ &= \bigcup_{s \in \{[1,3\rangle, [2,4\rangle\}} \{t \gg s \mid t \in P_S^{\mathrm{can}}(ab) \cup P_S^{\mathrm{can}}(bb)\} \\ &= \{[1,2\rangle, [2,3\rangle, [3,4\rangle\}\,.\end{aligned}$$

The reason is that the canonical split-spanner considers all combinations of output tuples $t' = t \gg s$ of P with splits s of S that cover t'. However, the split-spanner does not "remember" the relevant combinations of the t and s that lead to an output tuple t' and arbitrarily combines them. Therefore, if S is not disjoint and there is more than one split that covers some output tuple t', this may lead to combinations that don't correspond to outputs of P. □

We define the splittability condition that will characterize splittability for disjoint splitters.

Definition 5.11 (Splittability Condition)**.** Let P be a spanner and S a splitter. We say that P and S satisfy the *splittability condition* if the following holds:

(1) P and S satisfy the cover condition; and,
(2) for all $d, d^1, d^2 \in \Sigma^*$, $[i_1, j_1\rangle \in S(d^1), [i_2, j_2\rangle \in S(d^2)$ such that $d = d^1_{[i_1,j_1\rangle} = d^2_{[i_2,j_2\rangle}$ and for all $(\mathrm{SVars}(P), d)$-tuples t with $t_1 = t \gg [i_1, j_1\rangle$, $t_2 = t \gg [i_2, j_2\rangle$ it must hold that

$$t_1 \in P(d^1) \Leftrightarrow t_2 \in P(d^2).$$

Recall that the cover condition states that for every $d \in \Sigma^*$ and for all $t \in P(d)$ there is a span $s \in S(d)$ that covers t. The second requirement of the splittability condition is visualized in Figure 2. In essence, it says that whenever the same subdocument d is selected by S from two larger documents d_1 and d_2, and for every tuple t within d, if we consider the corresponding tuples t_1 and t_2 within the larger documents d_1 and d_2 then P should behave the same with respect to t_1 and t_2 on d_1 and d_2, respectively. That is, P selects t_1 in d_1 if and only if P selects t_2 in d_2.

Lemma 5.12. *Let P be a spanner and S be a disjoint splitter. Then the following three statements are equivalent:*

(1) P is splittable by S;
(2) P and S satisfy the splittability condition; and,
(3) $P = P^{\mathrm{can}}_S \circ S$.

Proof. Per definition (3) implies (1). Thus we need to prove that (1) implies (2) and that (2) implies (3).

(1) $\Rightarrow$ (2) : Let spanner P be splittable by S. The first requirement of the splittability condition follows directly from Lemma 5.3. It remains to show that the second requirement holds. Towards a contradiction, assume the requirement does not hold. Thus, there are $d, d^1, d^2 \in \Sigma^*$, $[i_1, j_1\rangle \in S(d^1), [i_2, j_2\rangle \in S(d^2)$ such that $d = d^1_{[i_1,j_1\rangle} = d^2_{[i_2,j_2\rangle}$ and there is a $(\mathrm{SVars}(P), d)$-tuple t with $t_1 = t \gg [i_1, j_1\rangle$, and $t_2 = t \gg [i_2, j_2\rangle$ such that $t_1 \in P(d^1)$ and $t_2 \notin P(d^2)$. Let P_S be a spanner for which P is splittable by S via P_S. Since S is disjoint, the span $[i_1, j_1\rangle \in S(d^1)$ is the only span of d^1 that covers t_1. Therefore, due to $t_1 \in P(d^1)$ and $P = P_S \circ S$ it must hold that $t \in P_S(d^1_{[i_1,j_1\rangle})$. However, using the same argument, $[i_2, j_2\rangle \in S(d^2)$ is the only span of d^2 that covers t_2. Thus, as $t_2 \notin P(d^2)$ and $P = P_S \circ S$ it must hold that $t \notin P_S(d^2_{[i_2,j_2\rangle})$. But $P_S(d^2_{[i_2,j_2\rangle}) = P_S(d^1_{[i_1,j_1\rangle})$ as $d = d^1_{[i_1,j_1\rangle} = d^2_{[i_2,j_2\rangle}$ which leads to the desired contradiction.

(2) $\Rightarrow$ (3) : Assume that the splittability condition holds. We show that P is splittable by S via the canonical split-spanner P^{can}_S.

We first argue that $P \subseteq P^{\mathrm{can}}_S \circ S$. Indeed, by construction, P^{can}_S simulates P on every span selected by S. Moreover, by the cover condition and disjointness of S, every tuple selected by P is included in exactly one split selected by S. Formally, let $d \in \Sigma^*$ be a document and $t \in P(d)$ be a tuple, selected by P. Then, there is exactly one $s \in S(d)$ such that s covers t. Let $d = d_{\mathrm{pre}} \cdot d_{\mathrm{mid}} \cdot d_{\mathrm{post}}$ such that $s = [|d_{\mathrm{pre}}| + 1, |d_{\mathrm{pre}} \cdot d_{\mathrm{mid}}| + 1\rangle$. Due to s covering t, there must be a tuple t' with $t = t' \gg s$. Thus it follows, per definition of P^{can}_S, that $t' \in P^{\mathrm{can}}_S(d_{\mathrm{mid}})$ and, therefore, $t \in (P^{\mathrm{can}}_S \circ S)(d)$.

We now show the other inclusion, that is, $P^{\mathrm{can}}_S \circ S \subseteq P$. To this end, let $d^1 \in \Sigma^*$ be a document and $t \in (P^{\mathrm{can}}_S \circ S)$ be a tuple. Thus, d^1 can be written as $d^1 = d^1_{\mathrm{pre}} \cdot d \cdot d^1_{\mathrm{post}}$, such that $[i_1, j_1\rangle = [|d^1_{\mathrm{pre}}| + 1, |d^1_{\mathrm{pre}} \cdot d| + 1\rangle \in S(d^1)$ and $t' \in P^{\mathrm{can}}_S(d)$ is a tuple with $t = t' \gg [i_1, j_1\rangle$. Per definition of P^{can}_S and $t' \in P^{\mathrm{can}}_S(d)$ there must be a document $d^2 = d^2_{\mathrm{pre}} \cdot d \cdot d^2_{\mathrm{post}} \in \Sigma^*$ such that $[i_2, j_2\rangle = [|d^2_{\mathrm{pre}}| + 1, |d^2_{\mathrm{pre}} d| + 1\rangle \in S(d^2)$ and $t' \gg [i_2, j_2\rangle \in P(d^2)$. By the second requirement of the splittability condition it directly follows that $t \in P(d_1)$. □

The following example shows that Lemma 5.12 does not hold in general if S is not disjoint.

Example 5.13. Let $P = aby\{b\} + cy\{b\}b$ and $S = x\{\Sigma^*\} + \Sigma^* x\{bb\}\Sigma^*$. Furthermore, let $d = bb, d_1 = abb$, $d_2 = cbb$, and $t = \{(y, [2, 3\rangle)\}$. With $s = [2, 4\rangle$, it is easy to see that $s \in S(d_1)$ and $s \in S(d_2)$. With $t_1 = t_2 = t \gg s$, we have that $t_1 \in P(d_1)$, but $t_2 \notin P(d_2)$. Therefore, the second requirement of the splittability condition does not hold. Nevertheless, it is easy to see that P is self-splittable. □

Lemma 5.14. *Let P, P_S be regular spanners and S be a disjoint splitter, such that $P = P_S \circ S$. Then $P^{\mathrm{can}}_S \subseteq P_S$.*

We note that the inclusion $P^{\mathrm{can}}_S \subseteq P_S$ does not hold if S is not disjoint, since P^{can}_S can select arbitrary tuples on documents for which S does not produce any output.

We are now ready to state the main complexity result of this section:

Theorem 5.15. *Deciding Splittability$[C]$ for disjoint splitters and $C \in \{\mathbf{RGX}, \mathbf{VSA}\}$ is PSPACE-complete.*

Proof sketch. The upper bound directly follow from Lemma 5.12, Proposition 5.9, and Theorem 5.1. For the lower bound, we give a reduction from the inclusion problem for regular expressions that is known to be PSPACE-complete. Let r_1 and r_2 be regular expressions. It can be shown that the boolean spanner $P = r_1$ is splittable by the disjoint splitter $S = x\{r_2\}$ if and only if $L(r_1) \subseteq L(r_2)$. □

5.3 Self-Splittability

Next, we discuss self-splittability.

Theorem 5.16. *Deciding Self-splittability$[C]$ for $C \in \{\mathbf{RGX}, \mathbf{VSA}\}$ is PSPACE-complete.*

Proof sketch. The upper bounds follow directly from Theorem 5.1. The lower bound for Self-splittability[**VSA**] follows directly from the lower bound of Self-splittability[**RGX**], since regex-formulas can be transformed into VSet-automata in polynomial time.

We give a reduction from the containment problem for **RGX** to Self-splittability[**RGX**]. To this end, let $r_1, r_2 \in \mathbf{RGX}$ be regex-formulas over the alphabet Σ with $\mathrm{SVars}(r_1) = \mathrm{SVars}(r_2)$ and let $a \notin \Sigma$ be a new symbol and $\Sigma' = \Sigma \cup \{a\}$. Furthermore, let $x, y \notin \mathrm{SVars}(r_1)$ be new variables. We define the spanner $P = r_1 + (a \cdot r_2)$ and splitter $S = a? \cdot x\{\Sigma^*\}$. Then P is self-splittable by S if and only if $[\![r_1]\!] \subseteq [\![r_2]\!]$. □

The following is immediate from Theorem 5.7.

Theorem 5.17. *Let P be a spanner and S be a disjoint splitter, both from* **dfVSA**. *Self-splittability for P and S is decidable in polynomial time.*

6 REASONING ABOUT SPLITTABILITY

In a complex pipeline that involves multiple spanners and splitters, it may be beneficial to reason about the manipulation or replacement of operators for the sake of query planning (in a similar way as we reason about query plans in a database system). In this section, we consider questions of this sort. For the class of regular spanners, we prove PSPACE-completeness for deciding on splitter *commutativity* (Theorem 6.2) and *subsumption*, that is, whether a splitter can always be executed after another (Theorem 6.3). We also discuss conditions for the *transitivity* of splitters (Observation 6.4 and Lemma 6.5).

Commutativity. An obvious problem is whether two splitters commute (possibly with respect to a context, which we abstract as a regular language). For example, suppose that we are given a program that processes a text document, and suppose that the query plan first splits by pages and then by paragraphs. This is the same as splitting by paragraphs and then by pages. So, the query planner can choose between the two options.

Formally, let S_1 and S_2 be two splitters and R a regular language. We say that S_1 and S_2 *commute w.r.t. R* if and only if $(S_1 \circ S_2)(d) = (S_2 \circ S_1)(d)$ for all $d \in R$. For instance, if S_1 and S_2 commute and S_1 is more selective than S_2, then it is beneficial to apply S_1 before S_2. The next lemma shows that the composition of splitters can be explicitly constructed.

Lemma 6.1. *Let S_1 and S_2 be splitters given as VSet-automata. Then a VSet-automaton for the splitter $(S_2 \circ S_1)$ can be computed in polynomial time.*

We now establish the complexity of the decision problem.

Theorem 6.2. *Let S_1, S_2, all coming from one of the classes* **RGX** *or* **VSA** *and let R be an NFA. Then deciding if S_1 and S_2 commute w.r.t. R is PSPACE-complete. The problem is PSPACE-hard even if $L(R) = \Sigma^*$.*

Subsumption. Another form of optimization is *subsumption*. We say that S *subsumes* S' *w.r.t.* R if $S(d) = (S' \circ S)(d)$ for all $d \in R$. For example, suppose that we are told that a spanner is splittable by a splitter S (e.g., the sentence splitter); does it also imply that it is splittable by S' (e.g., the paragraph splitter)? In Section 7, we discuss an extension of the framework where such knowledge is provided on arbitrary spanners as *split constraints*.

Theorem 6.3. *Let S, S', and R, all coming from one of the classes* **RGX** *or* **VSA**. *Then deciding if S subsumes S' w.r.t. R is PSPACE-complete.*

Proof. The upper bound is immediate from Lemma 6.1 and Theorem 4.1. The lower bound follows from regular expression universality. Let E be a regular expression and let $S' = x\{E\}$. Then $S = x\{\Sigma^*\}$ subsumes S' if and only if $L(E) = \Sigma^*$. □

Transitivity. We conclude the section with a few initial observations about the *transitivity* of splittability.

Observation 6.4. Let P, P_S be spanners and S_1, S_2 be splitters, such that $P = P_S \circ S_1$ and $S_1 = S_1 \circ S_2$. Then it is not necessarily true that $P = P_S \circ S_2$.

Proof. Let $P = \Sigma^* \cdot y\{a\} \cdot \Sigma^*$, $P_S = y\{a\}$, $S_1 = \Sigma^* \cdot x\{\Sigma\} \cdot \Sigma^*$, and $S_2 = \Sigma^* \cdot x\{\Sigma \cdot \Sigma\} \cdot \Sigma^* + x\{\Sigma\}$. It's easy to see that $P = P_S \circ S_1$ and $S_1 = S_1 \circ S_2$, but $P \neq P_S \circ S_2$. □

However, self-splittability transfers from one splitter to another more general splitter:

Lemma 6.5. *Let P be a spanner and S_1, S_2 be splitters, such that $P = P \circ S_1$ and $S_1 = S_1 \circ S_2$. Then $P = P \circ S_2$.*

7 BEYOND THE BASIC FRAMEWORK

We now discuss three problems that are based on the concept of splittability, but go beyond the basic framework discussed in the previous sections. The first problem is that of deciding on the splittability in the presence of *black-box* spanners that are known to follow *split constraints* (Section 7.1). The second problem is that of deciding on the splittability under a regular precondition on the input documents (Section 7.2). Finally, the third problem is that of deciding on the splittability under splitters that can *annotate* the splits as they run (Section 7.3).

7.1 Split-Constrained Black Boxes

We begin with motivating examples. Since the examples and our definitions afterwards use the natural join of spanners, we briefly recall it from Fagin et al. [7]. The spanner $P_1 \bowtie P_2$

is defined as follows. We have $\text{SVars}(P_1 \bowtie P_2) = \text{SVars}(P_1) \cup \text{SVars}(P_2)$, and $(P_1 \bowtie P_2)(d)$ consists of all d-tuples t that agree with some $t_1 \in P(d)$, and $t_2 \in P_2(d)$; note that the existence of t implies that t_1 and t_2 agree on the common variables of P_1 and P_2, that is, $t_1(x) = t_2(x)$ for all $x \in \text{SVars}(P_1) \cap \text{SVars}(P_2)$.

Example 7.1. In this example and the next, we'll denote by $P(x, y)$ that spanner P uses the variables x and y. Consider the spanner P that seeks to extract adjectives for Galaxy phones from reports. We define this spanner by joining three spanners:

The spanner $\alpha(x, y)$ is the regex formula

$$\Sigma^* \cdot x\{\text{Galaxy [A-Z]}\backslash\text{d}^*\} \cdot \Sigma^* \cdot y\{\Sigma^*\}. \cdot \Sigma^*$$

that extracts mentions of Galaxy brands (e.g., Galaxy A6 and Galaxy S8) followed by substrings y that occur right before a period.

The spanner $P_1(x, x')$ is a coreference resolver (e.g., the *sieve* algorithm [27]) that finds spans x' that coreference spans x. The spanner $P_2(x', y)$ finds pairs of noun phrases x' and attached adjectives y (e.g., based on a Recursive Neural Network [31]).

For example, consider the review *"I am happy with my Galaxy A6. It is stable."* Here, in one particular match, x will match (the span of) *Galaxy A6*, x' will match *it* (which is an anaphor for *Galaxy A6*), and y will match *stable*. (Other matches are possible too.)

How should a system find an efficient query plan to this join on a long report? Natively materializing each relation might be too costly: $\alpha(x, y)$ may produce too many matches, and $P_1(x, x')$ and $P_2(x', y)$ may be computationally costly. Nevertheless, we may have the information that P_1 is splittable by paragraphs, and that P_2 is splittable by sentences (hence, by paragraphs). This information suffices to determine that the entire join $\alpha(x, y) \bowtie P_1(x, x') \bowtie P_2(x', y)$ is splittable, hence parallelizable, by paragraphs. □

Example 7.2. Now consider the spanner that joins two spanners: $\alpha'(x)$ extracts spans x followed by the phrase *"is kind"* (e.g., *"Barack Obama is kind"*). The spanner $P'(x)$ extracts all spans x that match person names. Clearly, the spanner $\alpha'(x)$ does not split by any natural splitter, since it includes, for instance, the entire prefix of the document before *"is kind"*. However, by knowing that $P'(x)$ splits by sentences, we know that the join $\alpha'(x) \bowtie P'(x)$ splits by sentences. Moreover, by knowing that $P'(x)$ splits by 3-grams, we can infer that $\alpha'(x) \bowtie P'(x)$ splits by 5-grams. Here, again, the holistic analysis of the join infers splittability in cases where intermediate spanners are not splittable. □

We now formalize the splittability question that the examples give rise to. A *spanner signature* Π is a collection $\{\pi_1, \ldots, \pi_k\}$ of *spanner symbols*, where each π_i is associated with a set $\text{SVars}(\pi_i)$ of span variables. In Example 7.1, π_1 and π_2 would correspond to the name of a coreference resolver and an adjective extractor, respectively, with $\text{SVars}(\pi_1) = \{x, x'\}$ and $\text{SVars}(\pi_2) = \{x', y\}$. We assume that Π is connected, that is, the underlying hypergraph where every π_i is interpreted as the hyperedge consisting of $\text{SVars}(\pi_i)$ is connected. An *instance* I of Π associates with each spanner symbol π_i an actual spanner P_i such that $\text{SVars}(P_i) = \text{SVars}(\pi_i)$. In Example 7.1, these would be P_1 and P_2, respectively.

Let Π be a spanner signature, I an instance of Π, and α a regular spanner. We denote by $\alpha \bowtie I$ the spanner that is given by

$$\alpha \bowtie P_1 \bowtie \cdots \bowtie P_k.$$

We note that this is well-defined due to the associativity and commutativity of the $\bowtie$-operator.

A *regular split constraint* over a spanner signature Π is an expression of the form "π_i is self-splittable by the regular splitter S," which we denote by $\pi_i \sqsubseteq S$. An instance I of Π *satisfies* a set C of regular split constraints, denoted $I \models C$, if for every constraint $\pi_i \sqsubseteq S$ in C it is the case that P_i is self-splittable by S. The problem of *split-correctness with black boxes* is the following:

Black Box Split-Correctness	
Input:	A spanner signature Π, a set C of regular split constraints, a regular spanner α, and a splitter S.
Question:	Is $\alpha \bowtie I$ splittable by S whenever I is an instance of Π such that $I \models C$?

A natural question to ask is the following. Assume that α is self-splittable by S and we have all split constraints $\pi_i \sqsubseteq S$, that is, all spanners are self-splittable by the same splitter S. Is it the case that $\alpha \bowtie I$ is splittable by S? The following Lemma shows that this is not the case in general.

Lemma 7.3. *There are spanners P_1 and P_2 and a splitter S, such that P_1 and P_2 are self-splittable by S, but $P_1 \bowtie P_2$ is not splittable (and thus also not self-splittable) by S.*

Proof. Let $P_1 = \Sigma^* \cdot x_1\{a\} \cdot x_2\{b\} \cdot \Sigma^*$ and $P_2 = \Sigma^* \cdot x_2\{b\} \cdot x_3\{a\} \cdot \Sigma^*$ be spanners. Furthermore, let $S = \Sigma^* \cdot x\{(a \cdot \Sigma) + (\Sigma \cdot a)\} \cdot \Sigma^*$ be a splitter. Then both P_1 and P_2 are self-splittable by S. However, the join $P := P_1 \bowtie P_2$ can not be splittable by S since for $d = aba$, $S(d) = \{[1, 3\rangle, [2, 4\rangle\}$ and $P(d) = \{([1, 2\rangle, [2, 3\rangle, [3, 4\rangle)\}$, and therefore the cover condition is violated. □

The next result shows that in the presence of disjoint splitters the join operator preserves self-splittability.

Theorem 7.4. *Let S be a disjoint splitter, α be a spanner that is splittable by S, Π be a spanner signature, and $C = \{\pi_1 \sqsubseteq S, \ldots, \pi_k \sqsubseteq S\}$ be a set of regular split constraints. Then Π, C, α, and S are black box split-correct.*

Proof. Let I be an instance of Π, such that $I \models \Pi$ and let P_i be the spanner interpreting π_i. We have to show, that $\alpha \bowtie I = P_S \circ S$ for some spanner P_S. Per assumption, α is splittable by S and P_i is self-splittable by S for all $1 \leq i \leq k$. Let α_S be a spanner such that $\alpha = \alpha_S \circ S$ and let $P_S = \alpha_S \bowtie P_1 \bowtie \cdots \bowtie P_k$.

We argue that $\alpha \bowtie I$ is splittable by S via P_S (i.e. $\alpha \bowtie I = P_S \circ S$). To this end, let $d \in \Sigma^*$ be a document and $t \in \alpha \bowtie I$ be a tuple. Then there are tuples $t_\alpha \in \alpha(d)$, and $t_i \in P_i(d)$ for $1 \leq i \leq k$, such that $t = t_\alpha \bowtie t_1 \bowtie \cdots \bowtie t_k$. Furthermore, as S is disjoint and Π is connected there is a unique split $s \in S(d)$ covering all tuples $t, t_\alpha, t_1, \ldots, t_k$. By $\alpha = \alpha_S \circ S$ there is a tuple $t'_\alpha \in \alpha_S(d_s)$ such that $t_\alpha = t'_\alpha \gg s$. Furthermore, by self-splittability of P_i there also is a tuple $t'_i \in P_i(d_s)$ such that $t_i = t'_i \gg s$. Thus, it must hold that $t' = t'_\alpha \bowtie t'_1 \bowtie \cdots \bowtie t'_k \in P_S(d_s)$ and $t = t' \gg s$. Hence, $t \in P_S \circ S$.

For the other direction, assume that there is a document $d \in \Sigma^*$, a split $s \in S(d)$ and a tuple $t' \in P_S(d_s)$. Then it follows by the same argument, that the tuple $t = t' \gg s \in \alpha \bowtie I$. □

7.2 Regular Precondition

Sometimes a spanner is not splittable by a given splitter, because of a reason that seems marginal. For instance, the spanner may first check that the document conforms to some standard format, such as Unicode, UTF-8, CSV, HTTP, etc. We provide two ways to deal with such kind of scenarios: *regular preconditions* (here) and *annotated splitters* (in Section 7.3).

A *splitter with filter* is a pair (S, L) where S is a splitter and L is a word language. We denote such splitters as $S[L]$. The spanner defined by $S[L]$ is the function that maps each document d to $S(d)$ if $d \in L$ and to $\emptyset$ otherwise.

It is easy to see that splitters with filter are not more powerful than ordinary splitters. However, they give rise to new problems that can be studied. For instance, it may be the case that we already have a spanner and splitter available that we do not want to change, but we can use a regular language as a filter to obtain split-correctness on all documents that satisfy the filter.

Split-correctness[C] with regular filter	
Input:	Spanners $P, P_S \in C$ and splitter $S \in C$.
Question:	Is there a regular language L such that $P = P_S \circ S[L]$?

Splittability[C] with regular filter	
Input:	Spanners $P \in C$ and splitter $S \in C$.
Question:	Is there a regular language L such that P is splittable by $S[L]$?

Furthermore, Self-splittability[C] with regular filter is the special case of Splittability[C] with regular splitter where we ask if P is self-splittable by $S[L]$.

The next lemma shows that there is a minimal filter language for each spanner. For a regular spanner P, define $L_P = \{d \mid P(d) \neq \emptyset\}$. That is, L_P is the set of strings on which P produces a non-empty result.

Lemma 7.5. *Let P and P_S be spanners, S be a splitter and L be a regular language. Then, $P = P_S \circ S[L]$ implies that:*

(1) $L_P \subseteq L$; and,
(2) $P = P_S \circ S[L_P]$.

Proof. Per definition of L_P it holds that $d \in L_P \Leftrightarrow P(d) \neq \emptyset$, for every $d \in \Sigma^*$. For the sake of contradiction, assume that $L_P \nsubseteq L$ and $d \in L_P$ such that $d \notin L$. By $d \in L_P$, it holds that $P(d) \neq \emptyset$, but due to $d \notin L$ the document d is filtered out by $S[L]$ (i.e. $S[L](d) = \emptyset$), contradicting $P = P_S \circ S[L]$. The second condition (2) follows directly from (1) and $P = P_S \circ S[L]$. □

Due to Lemma 7.5, we can decide Split-correctness with regular filter by constructing a splitter S' that is equivalent to the splitter with filter $S[L_P]$ and then testing if $P = P_S \circ S'$.

Theorem 7.6. *Deciding Split-correctness[C] with regular filter and Self-splittability[C] with regular filter are PSPACE-complete, if $C \in \{\mathbf{RGX}, \mathbf{VSA}\}$ and P is functional.*

Proof sketch. If P is functional, we can construct a splitter S' in polynomial time, which computes the same function as $S[L_P]$ (for readers familiar with spanner algebra, $S' = S \bowtie \pi_\emptyset P$). Therefore, using Lemma 7.5 there is a regular language L such that $P = P_S \circ S[L]$ if and only if $P = P_S \circ S[L_P]$. Thus, PSPACE-completeness follows directly from Theorem 5.1 (and Theorem 5.16 in the case of Self-splittability[C] with regular filters). □

Theorem 7.7. *Deciding Splittability[C] with regular filter is PSPACE-complete, if $C \in \{\mathbf{RGX}, \mathbf{VSA}\}$, P is functional, and S is disjoint.*

Proof. Immediate from Theorem 7.6 and Theorem 5.15. □

7.3 Annotated Splitters

Annotated splitters form a natural extension of splitters that can propagate annotations to the splitted strings. That is, instead of mapping documents to relations of spans, annotated splitters map documents to relations of key-span pairs (in analogy to the key-value pairs from the MapReduce framework). They extend *splitters with filters* in the following way. A splitter with filter can be seen as a function that annotates each of the splits with the Boolean value 1 or 0, depending on whether the input document satisfies the precondition or not. More precisely, given a splitter with filter (S, L), we can define a function $S[L]_{\text{ann}}$ as $S[L]_{\text{ann}}(d) := S(d) \times \{1\}$ if $d \in L$ and $S[L]_{\text{ann}}(d) := S(d) \times \{0\}$ otherwise, for every document d. An annotated splitter generalizes this idea in the sense

that (1) it can choose from an arbitrary set of *keys* instead of the set of Boolean values and (2) it can annotate different splits with different keys.

As an example, assume that one wants to extract information from an HTTP log and wants to process the information extracted from GET requests differently than that extracted from POST requests. An annotated splitter can split the document and annotate each split with the type of request from which the split was extracted. This annotation can then be used to choose different split-spanners that work on the splits with different annotations.

We omit formal details, due to space constraints, but note that complexity results on *annotated* split-correctness and splittability that are in-line with those on ordinary split-correctness and splittability can be obtained.

8 CONCLUDING REMARKS

We embarked on an exploration of the task of automating the distribution of information-extraction programs across splitters. Adopting the formalism of *document spanners* [7] and the concept of *parallel-correctness* [2], our framework focuses on two computational problems, split-correctness and splittability, as well as their special case of self-splittability. We presented a preliminary analysis of these problems within the class of regular spanners. We have also discussed several natural extensions of the framework, considering the reasoning about the application about multiple splitters, black-box spanners with split constraints, preconditions, and splitters with annotation capabilities. Our principal goal is to open up new directions for research within the framework, and indeed, a plethora of open problems are left for future investigation. We discuss these problems in the remainder of this paper.

Among the basic problems, we know the least about the splittability problem. For example, we do not know even whether the problem is *decidable* without the assumption of disjointness. Moreover, what is the complexity of splittability when restricting to **dfVSA** and disjoint splitters? A fundamental problem is the existence of a *canonical split spanner* P_S such that, similarly to Proposition 5.9, if splittablity holds for P and S, then it can be realized by P_S.

We know more about split-correctness and self-splittablity, but there are some basic open problems there as well. Can we relax any of the assumptions of determinism and disjointedness and still retain tractability? What other natural assumptions lead to tractability? For instance, N-gram splitters and N-consecutive-sentence splitters are not disjoint; but do they possess any general (and easily detectable) properties that can be used for general efficient solvers of our problems? All of these open problems are within *regular* spanners; when considering more expressive languages for spanners (e.g., the class of *core* spanners [7, 11] that allow for string equalities), all problems reopen.

A variant of splittability that we have not touched upon is that of deciding, given a spanner P, whether it can be decomposed in a nontrivial way. We can show that this variant closely relates to the *language primality* problem—can a given regular language be decomposed as the concatenation of non-trivial regular languages? Interestingly, Martens et al. [22] showed that language primality is also related to the work of Abiteboul et al. [1] on typing in distributed XML, which is quite reminiscent, yet different from, our work.

For the extensions of reasoning about splitters, and deciding on splittability with black-box spanners, we barely scratched the surface. Specifically, we believe that reasoning about split constraints over black-box extractors can have a profound implication on the usability of IE systems to developers of varying degrees of expertise, while embracing the advances of the Machine Leaning and Natural Language Processling communities on learning complex functions such as artificial neural networks.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for PODS 2019 for many helpful remarks. The work of Johannes Doleschal was supported by the Special Research Fund (BOF) of Hasselt University and the Deutsche Forschungsgemeinschaft (DFG), Grant MA 4938/4-1. The work of Benny Kimelfeld and Yoav Nahshon was supported in part by the Israel Science Foundation (ISF), Grant 1295/15.

REFERENCES

[1] S. Abiteboul, G. Gottlob, and M. Manna. Distributed XML design. *Journal of Computer and System Sciences*, 77(6):936–964, 2011.

[2] T. J. Ameloot, G. Geck, B. Ketsman, F. Neven, and T. Schwentick. Parallel-correctness and transferability for conjunctive queries. *Journal of the ACM*, 64(5):36:1–36:38, 2017.

[3] T. J. Ameloot, G. Geck, B. Ketsman, F. Neven, and T. Schwentick. Reasoning on data partitioning for single-round multi-join evaluation in massively parallel systems. *Communications of the ACM*, 60(3):93–100, 2017.

[4] J. Chen, D. Ji, C. L. Tan, and Z. Niu. Unsupervised feature selection for relation extraction. In *International Joint Conference on Natural Language Processing (IJCNLP), Companion Volume*, 2005.

[5] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, and S. Vaithyanathan. SystemT: An algebraic approach to declarative information extraction. In *Annual Meeting on Association for Computational Linguistics (ACL)*, pages 128–137, 2010.

[6] J. P. C. Chiu and E. Nichols. Named entity recognition with bidirectional LSTM-CNNs. *Transactions of the ACL*, 4:357–370, 2016.

[7] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *Journal of the ACM*, 62(2):12:1–12:51, 2015.

[8] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Declarative cleaning of inconsistencies in information extraction. *ACM Transactions on Database Systems*, 41(1):6:1–6:44, 2016.

[9] F. Florenzano, C. Riveros, M. Ugarte, S. Vansummeren, and D. Vrgoc. Constant delay algorithms for regular document spanners. In *Symposium on Principles of Database Systems (PODS)*, pages 165–177, 2018.

[10] D. D. Freydenberger. A Logic for Document Spanners. In *International Conference on Database Theory (ICDT)*, volume 68, pages 13:1–13:18, 2017.

[11] D. D. Freydenberger and M. Holldack. Document spanners: From expressive power to decision problems. *Theory of Computing Systems*, 62(4):854–898, 2018.

[12] D. D. Freydenberger, B. Kimelfeld, and L. Peterfreund. Joining extractions of regular expressions. In *Symposium on Principles of Database Systems (PODS)*, pages 137–149, 2018.

[13] C. Giuliano, A. Lavelli, and L. Romano. Exploiting shallow linguistic information for relation extraction from biomedical literature. In *Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, 2006.

[14] Hadoop, apache, http://hadoop.apache.org, 2009.

[15] M. A. Hearst. Texttiling: Segmenting text into multi-paragraph subtopic passages. *Computational Linguistics*, 23(1):33–64, 1997.

[16] D. Kozen. Lower bounds for natural proof systems. In *Symposium on Foundations of Computer Science (FOCS)*, pages 254–266. IEEE, 1977.

[17] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer. Neural architectures for named entity recognition. In *Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, pages 260–270, 2016.

[18] R. Leaman and G. Gonzalez. BANNER: an executable survey of advances in biomedical named entity recognition. In *Pacific Symposium on Biocomputing (PSB)*, volume 13, pages 652–663, 2008.

[19] H. Lee, Y. Peirsman, A. Chang, N. Chambers, M. Surdeanu, and D. Jurafsky. Stanford's multi-pass sieve coreference resolution system at the conll-2011 shared task. In *Conference on Computational Natural Language Learning (CoNLL), Shared Task*, pages 28–34, 2011.

[20] A. Madaan, A. Mittal, Mausam, G. Ramakrishnan, and S. Sarawagi. Numerical relation extraction with minimal supervision. In *AAAI Conference on Artificial Intelligence*, pages 2764–2771, 2016.

[21] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM Journal on Computing*, 39(4):1486–1530, 2009.

[22] W. Martens, M. Niewerth, and T. Schwentick. Schema design for XML repositories: complexity and tractability. In *Symposium on Principles of Database Systems (PODS)*, pages 239–250, 2010.

[23] F. Maturana, C. Riveros, and D. Vrgoč. Document spanners for extracting incomplete information: Expressiveness and complexity. *CoRR*, abs/1707.00827, 2017.

[24] F. Maturana, C. Riveros, and D. Vrgoč. Document spanners for extracting incomplete information: Expressiveness and complexity. In *Symposium on Principles of Database Systems (PODS)*, pages 125–136, 2018.

[25] Y. Nahshon. Relational framework for information extraction. Master's thesis, Technion - Computer Science Department, 2018.

[26] B. Pang and L. Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *Annual Meeting on Association for Computational Linguistics (ACL)*, pages 271–278, 2004.

[27] K. Raghunathan, H. Lee, S. Rangarajan, N. Chambers, M. Surdeanu, D. Jurafsky, and C. D. Manning. A multi-pass sieve for coreference resolution. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 492–501, 2010.

[28] C. D. Sa, A. Ratner, C. Ré, J. Shin, F. Wang, S. Wu, and C. Zhang. DeepDive: Declarative knowledge base construction. *SIGMOD Record*, 45(1):60–67, 2016.

[29] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using Datalog with embedded extraction predicates. In *Conference on Very Large Data Bases (VLDB)*, pages 1033–1044, 2007.

[30] J. Shin, S. Wu, F. Wang, C. D. Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using DeepDive. *Proceedings of the VLDB Endowment (PVLDB)*, 8(11):1310–1321, 2015.

[31] R. Socher, C. C. Lin, A. Y. Ng, and C. D. Manning. Parsing natural scenes and natural language with recursive neural networks. In *International Conference on International Conference on Machine Learning (ICML)*, pages 129–136, 2011.

[32] W. M. Soon, H. T. Ng, and C. Y. Lim. A machine learning approach to coreference resolution of noun phrases. *Computational Linguistics*, 27(4):521–544, 2001.

[33] R. E. Stearns and H. B. Hunt III. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM Journal on Computing*, 14(3):598–611, 1985.

[34] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, 2003.

[35] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[36] D. Zeng, K. Liu, Y. Chen, and J. Zhao. Distant supervision for relation extraction via piecewise convolutional neural networks. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1753–1762, 2015.