Database Principles and Challenges in Text Analysis

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

ABSTRACT

A common conceptual view of text analysis is that of a two-step process, where we first extract relations from text documents and then apply a relational query over the result. Hence, text analysis shares technical challenges with, and can draw ideas from, relational databases. A framework that formally instantiates this connection is that of the document spanners. In this article, we review recent advances in various research efforts that adapt fundamental database concepts to text analysis through the lens of document spanners. Among others, we discuss aspects of query evaluation, aggregate queries, provenance, and distributed query planning.

1. INTRODUCTION

Different tools and paradigms have been developed over the past decades to facilitate the challenge of extracting structured information from text—a task generally referred to as *Information Extraction* (IE). Common textual sources include natural language from a variety of sources such as scientific publications, customer input and social media, as well as machine-generated activity logs. Instantiations of IE are central components in text analytics and include tasks such as segmentation, named-entity recognition, relation extraction, and coreference resolution [55]. Rules and rule systems have consistently been key components in such paradigms, yet their roles have varied and evolved over time. Systems such as Xlog [59] and IBM SystemT [13] use IE rules for materializing relations inside *relational query languages.* Machine-learning classifiers and probabilistic graphical models (e.g., Conditional Random Fields) use rules for *feature generation* [38, 62]. Rules serve as *weak constraints* in Markov Logic Networks [51] and in the DeepDive system [60]. Rules are also used for generating *noisy training data* ("labeling functions") in the state-of-the-art Snorkel system [54].

Even though there is a fundamental difference in the structure of the underlying data, there is a tight connection between IE rules and relational databases: both provide machinery for manipulating base relations, either given explicitly (relational databases) or extracted from the text (IE). In the latter case, the base relations are typically constructed via generic extractors implemented in a variety of ways, from regular expressions (e.g., dictionary lookups) to machine-learned networks. We refer to these extractors as *primitive extractors*; hence, we view IE as a process where relational operators are applied to the relations extracted via primitive extractors.

Given the conceptual connection between relational databases and IE, can we leverage the principles of relational data management, as established over decades of research and practice, in the world of IE (and text analytics in general)? Particular questions include the following.

- *What is the expressive power of extraction languages?* What is the contribution of the relational operators to the expressiveness of the language of the primitive extractors? Does the relational component add power or just facilitates query formulation?
- *What is the computational complexity of evaluating IE programs?* How does it depend on the query and the textual data (combined/data complexity)? What guarantees can be made by an algorithm for streaming out many answers (enumeration complexity)? Can we understand their fine-grained complexity as we do for database algorithms [20]? Can we evaluate *aggregate queries* efficiently without materializing the aggregated tuples?
- *How do we approach query planning for IE?* Can we come up with useful plans that *parallelize* the task at hand among many independent computational units? Can we analyze the query to infer such independence as done in the context of *parallel-correctness* in databases [5]?
- *How can we leverage and efficiently manage the* provenance *accumulated in the process of extracting information from text?* Can we use machinery that is based on firm mathematical foundations such as the *provenance semirings* in databases [30]?

The framework of *document spanners* (*spanners* for short) has been established with the aim of providing the theoretical basis to pursue the above questions and build the foundations of relational principles in IE [21]. It has been originally introduced as the theoretical basis underlying IBM SystemT. Formally, in this framework a *document* is a string d over a finite alphabet, a *span* of d represents a substring of d by its start and end positions, and a *spanner* is a function that maps every document d into a relation over the spans of d [21].

The most studied instantiation of spanners is the class of *regular spanners*—the closure of *regex-formulas* (regular expressions with capture variables) under the standard operations of the relational algebra (projection, natural join, union, and difference). Equivalently, the regular spanners are the ones expressible as *variable-set automata* (vset-automata for short)—nondeterministic finite-state automata that can open and close capture variables. These spanners extract from the text relations wherein the capture variables are the attributes. The vset-automata are computationally challenging since number of extracted tuples can be exponential in the size of the automaton; hence, combined with the input string, a vset-automaton constitutes a compact representation of a relation, similarly to the concept of a Factorized Database (FDB) [44]; and as in FDB, we aim to evaluate queries over the represented relations efficiently, without materializing these relations.

In the remainder of this article, we will describe some of the research progress that has been made over recent years in an attempt of addressing the aforementioned questions. While we skip (for lack of space) any discussion on aspects of expressiveness that have been thoroughly studied [21, 25, 41, 48, 50, 57, 58], we will review recent progress on the evaluation complexity of spanners [4, 6] (Section 3), the incorporation of provenance and aggregate queries [15, 18] (Section 4) and parallel evaluation [16] (Section 5).

We note that much of the content of this article is based on prior publications of the authors [14, 15, 16, 17, 18, 19], where the reader can find the technical details that we omit here.

2. SPANNERS IN A NUTSHELL

We view a *document* (or *word*) d as a finite sequence of symbols from a finite alphabet Σ. That is, we have $d = \sigma_1 \cdots \sigma_n$ where $\sigma_i \in \Sigma$ for every $i \in \{1, \ldots, n\}$. We denote the *length* n *of* d as $|d|$. A *span of* d is an expression of the form $[i, j\rangle$ with $1 \le i \le j \le n+1$, representing the interval of d that starts with the i-th symbol and ends right before the j-th symbol. For instance, the span $[23, 30\rangle$ on the document in Figure 1 represents the interval that starts at position 23 and ends right before position 30. For a span $[i, j\rangle$ of d, we denote by $d_{[i,j\rangle}$ the word $\sigma_i \cdots \sigma_{j-1}$. That is, for the document d in Figure 1, $d_{[23,30\rangle}$ is the word *Belgium*.

A *document spanner* is a function that transforms documents to *span relations*, which are database relations wherein every value is a span [21]. To a span relation we can associate a *string relation*, which is obtained by replacing every span $[i, j\rangle$ in the span relation with the string $d_{[i,j\rangle}$.

Example 2.1. *Consider the document in Figure 1. The table at the bottom left depicts a span relation over the document. The relation at the bottom right is the corresponding string relation, from which we see that the spanner extracts locations along with the corresponding number of events from the document.*

In more formal terms, spanners use *span variables* from an infinite set Vars, which is disjoint from the alphabet Σ. Let V be a finite subset of Vars. A *document spanner* (henceforth *spanner*) over V is a function S that maps each document d to a finite $|V|$-ary relation where the variables in V serve as attribute names and all the values are spans of d. (We denote by $|X|$ the cardinality of a set X.)

Spanners can be specified using a wide range of formalisms. We focus here on formalisms that are based on *regular languages*, but the literature has examples that go beyond that, such as core spanners [21, 57] (that can also be represented also via SpLog [25]), context-free languages [48], and Datalog for spanners [50].

2.1 Regular Spanners

The most studied class of spanners is the class of *regular spanners*. Such spanners can be defined using *generalized regex formulas*, which are regular expressions with capture variables. Formally, we define their syntax with the inductive rule

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

where ε denotes the empty word, $\sigma \in \Sigma$, $x \in$ Vars, $\vee$ denotes disjunction, $\cdot$ denotes concatenation, and * denotes Kleene closure. We usually leave the notation of concatenation implicit. Here $x{\vdash}$ and ${\dashv}x$ are the operations that do not match a symbol in the input document, but rather "open" and "close" the variable x, respectively. To each generalized regex formula α, we can associate a language $L(\alpha)$ over the set of symbols $\Sigma \uplus \{x{\vdash} \mid x \in \text{Vars}\} \uplus \{{\dashv}x \mid x \in \text{Vars}\}$ using the standard semantics of regular expressions. Furthermore, we denote the set of variables that occur in α by Vars(α) (similarly for words w). In order to properly define a spanner, generalized regex formulas need some syntactic restrictions. In particular, we need to ensure that

(a) every variable is "opened" before it is "closed": for every word $w \in L(\alpha)$ and every $x \in \text{Vars}(\alpha)$, we have that $x{\vdash}$ occurs before ${\dashv}x$ in w *and*

(b) every variable is "opened" and "closed" exactly once: for every word w in $L(\alpha)$ and every $x \in \text{Vars}(\alpha)$, each of $x{\vdash}$ and ${\dashv}x$ appears exactly once in w.

We call α *functional* if it meets these two conditions. From now on in the paper, we assume that all generalized regex formulas are functional.

`There␣are␣7␣events␣in␣Belgium,␣9-15␣in␣France,␣three␣in␣Berlin.`

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

x_{loc}	x_{events}
$[23, 30\rangle$	$[11, 12\rangle$
$[40, 46\rangle$	$[32, 36\rangle$
$[57, 63\rangle$	$[48, 53\rangle$

d	$f_w(d)$
7	7
9-15	9
three	3

$d_{x_{\mathsf{loc}}}$	$d_{x_{\mathsf{events}}}$	$w(d, t)$
Belgium	7	7
France	9-15	9
Berlin	three	3

Figure 1: A document d (top), a span relation R (bottom left), a partial function f_w (bottom middle), and the corresponding string relation with weights $w(d, t)$ from weight function $w := (x_{\mathsf{events}}, f_w)$ (bottom right).

Before we explain the semantics of generalized regexes, we look at an example, in which we use Σ as a shorthand for $\bigvee_{\sigma \in \Sigma} \sigma$.

Example 2.2. *Consider again the document in Figure 1. Extracting the numbers (sequences of digits) from this document can be done with the regex*

$$\Sigma^*(\sqcup \vee \texttt{-})\ x\vdash\ (0 \vee 1 \vee \cdots \vee 9)^*\ \dashv x\ (\sqcup \vee \texttt{-})\Sigma^*\ .$$

Intuitively, the this regex starts by reading an arbitrary prefix, then reads the blank symbol $\sqcup$ or a dash `-`*, and "opens" the variable x. The variable x is "closed" after reading an arbitrary number of digits, followed by a blank symbol or a dash and an arbitrary postfix. As such, the regex extracts the span relation containing the spans $[11, 12\rangle$, $[32, 33\rangle$, and $[34, 36\rangle$ from the document in Figure 1.*

We define the semantics of a (functional) regular spanner α. Observe that each word $w \in L(\alpha)$ encodes two things:

- a *document* $\mathrm{doc}(w)$, which is obtained from w by deleting the symbols of the form $x\vdash$ and $\dashv x$ for variables x; and
- a *tuple* $\mathrm{tup}(w)$ *of spans* that is obtained from the positions where $x\vdash$ and $\dashv x$ occur in w.

The former is rather clear, but we explain the latter a bit more precisely. Since α is functional, every word $w \in L(\alpha)$ can be written as

$$w\ =\ u_x\ x\vdash\ v_x\ \dashv x\ w_x\ ,$$

in a unique manner for each $x \in \mathrm{Vars}(\alpha)$. The tuple $\mathrm{tup}(w)$ maps each variable $x \in \mathrm{Vars}(w)$ to the span $[i_x, j_x\rangle$ where $i_x = |\,\mathrm{doc}(u_x)|$ and $j_x = i_x + |\,\mathrm{doc}(v_x)|$. Notice that doc ensures that the indices i_x and j_x refer to positions in the document and do not consider other variable operations.

We can now associate to α a spanner that maps every document d to

$$\alpha(d) := \{\mathrm{tup}(w) \mid w \in L(\alpha) \text{ and } \mathrm{doc}(w) = d\}\ .$$

Notice that $\alpha(d)$ is indeed a span relation.

Spanners can be defined using finite automata. Indeed, since $L(\alpha)$ is just a regular language, we can just as well use non-deterministic finite automata over the alphabet $\Sigma \uplus \{x\vdash \mid x \in \mathrm{Vars}\} \uplus \{\dashv x \mid x \in \mathrm{Vars}\}$ to define it. Such automata, that therefore also define the class of regular spanners, are called *variable-set automata* (*vset-automata* for short) in the literature. Just as generalized regexes, they can open and close variables.

2.2 Regular Spanners Are Robust

Generalized regex formulas can define words of the form $x\vdash a\, y\vdash b \dashv x\, c \dashv y$, where the opening and closing operations of variables is not "properly" nested. The more conventional way of incorporating variables (e.g., in Perl regular expressions) is via "capturing" (or "capture groups") where variables take on complete sub-expressions of the regular expression and, in particular, feature proper nesting of variable assignments. In our formalism, such expressions correspond to a restriction of the generalized regex formulas, as we do next.

We define the syntax of *regex formulas* with the recursive rule

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

Although spanners defined by regex formulas are expressively weaker than those defined by generalized regex formulas, their expressiveness becomes the same again when they are *closed under the relational algebra operators.* Since spanners produce span relations, thus objects that live in relational database systems, it is natural to ask ourselves whether the expressiveness of a class spanners increases if we additionally allow their output span relations to be manipulated by the relational algebra operations: select, project, join, union, and difference.

Note that the algebraic operators view spans as atomic values and, in particular, disregard the underlying document. For instance, in the natural join of two spanners S and S', the join condition on two tuples $t \in S(d)$ and $t' \in S'(d)$ is that, for all $x \in \mathrm{Vars}(S) \cap \mathrm{Vars}(S')$, the spans $[i_x, j_x\rangle := t(x)$ and $[i'_x, j'_x\rangle := t'(x)$ are equal: $i_x = i'_x$ and $j_x = j'_x$. Similarly, the atomic predicates of the selection involve equality and disequality of spans (and not their associated strings). Selection conditions that involve the equality of the associated substrings give rise to *core spanners* that we discuss in the next section.

For any class $\mathcal{S}$ of spanners, we write $\mathcal{S}$ *with RA* to denote the closure of $\mathcal{S}$ under these relational algebra operations.

Theorem 2.3 (Fagin et al. [21, 22]). *The following are equally expressive:*
(a) vset-automata
(b) generalized regex formulas
(c) regex formulas with RA
(d) vset-automata with RA
(e) generalized regex formulas with RA

The result shows that regular spanners are a very robust class. For this reason, we will focus our attention in this article mostly on regular spanners.

2.3 Extensions of Regular Spanners

Other languages have been studied in connection to regular spanners. For example, the core spanners are the closure of the regular spanners under the positive relational algebra (union, projection and natural join) along with the selection operator that is based on the string-equality predicate. Fagin et al. [21] showed that every core spanner can be represented as a regular spanner followed by a filtering based on string equality among variables (namely the *Core Simplification Lemma* [21, Lemma 4.19]). Peterfreund et al. [50] considered the application of Datalog on top of regular spanners, and showed that the resulting language has precisely the expressiveness of the class of spanners computable in polynomial time.

3. EVALUATION

We are now ready to discuss central problems that are associated to evaluating spanners.

3.1 Enumerating the Output

Similarly to database queries, complexity analysis for spanners should account for the fact that they are required to produce a stream with potentially many answers. Hence, we adopt complexity notions of enumeration problems. Moreover, the complexity of database queries has been studied under various yardsticks of tractability and hardness, including data complexity (where the query is fixed and the data is the input), combined complexity (where both are the input), course-grained complexity (e.g., polynomial vs. exponential time) and fine-grained complexity (e.g., linear vs. quadratic time). Notably, past research has established characterizations of conjunctive queries that admit enumeration with a constant delay (i.e., time between consecutive answers) after linear-time preprocessing, that is, the time that is required just to read the input and then write the answers one by one [7, 11]. The complexity of evaluating regular spanners has been studied under these complexity concepts, and the strongest guarantee is by Amarilli et al. [3].

Eval	
Input:	Spanner S and document d.
Task:	Compute $S(d)$.

Theorem 3.1 ([3, Theorem 1.1]). *Let α be a generalized regex formula and d be a document. It is possible to enumerate the span relation $\alpha(d)$ with linear preprocessing and constant delay in $|d|$, and polynomial preprocessing and delay in $|\alpha|$.*

This result was a culmination of a line of work on enumeration of the answers of MSO-definable queries on words and trees [1, 2, 4, 9, 24, 39, 42, 43, 46, 47]. It should be noted that some of this work considers the underlying word or tree to be static, while others allow updates [2, 4, 9, 24, 39, 42, 43, 46, 47], which is technically a very different setting.

3.2 Random Sampling and Counting

The problem of answer enumeration goes hand in hand with those of counting answers (i.e., calculating $|S(d)|$) and sampling answers (i.e., producing a random tuple $t \in S(d)$ with a uniform probability $1/|S(d)|$). When enumerating a large number of answers, one wishes to count the answers, faster than actually producing all answers, in order to know how far along she is, and one can sample answers in order to get insights with statistical significance about the answer space. The two tasks are related, as sampling techniques often require counting and (approximate) counting often requires sampling [6, 32].

We parameterize the counting problem by a class $\mathcal{S}$ of spanners, since the problem is intractable in general and we are interested in classes that make the problem efficiently solvable.

Count for $\mathcal{S}$	
Input:	Spanner S and document d.
Task:	Compute $|S(d)|$.

This problem has mostly been studied for spanners that are represented by vset-automata. The reason is that notions such as *unambiguity*, which lead to better complexities, are more established on automata models. Indeed, the following theorem states that, Count is tractable if the vset-automaton is unambiguous, whereas it is intractable for the general class of vset-automata. Loosely explained, a vset-automaton A is said to be unambiguous if, for every document d, every tuple in $A(d)$ is witnessed by a single run of A.

Throughout this article, we denote by VSA the set of all functional vset-automata and by uVSA the subset thereof that is unambiguous.

Theorem 3.2 ([6, 24]). *The following holds for spanners given as vset-automata:*
(a) Count *is* spanL*-complete.*
(b) Count *is in* FP *for* uVSA*.*
(c) Count *can be approximated by an* FPRAS*.*

Part (a) of the theorem was proved by Florenzano et al. [24, Theorem 5.2] and parts (b) and (c) were proved by Arenas et al. [6, Corollaries 4.1 and 4.2]. The complexity class spanL is generally considered to be an intractable class. A canonical problem that is spanL-complete is #NFA: given a non-deterministic finite automaton A and an integer n, how many words in $L(A)$ have length n? The #NFA problem has also been proved to be #P-complete by Kannan et al. [33]. (Indeed, under the type of reductions considered by Kannan et al., spanL = #P.)

Parts (b) and (c) of Theorem 3.2 are tractability results. That is, Count can be solved exactly in polynomial time (FP) if spanners are given by unambiguous vset-automata. Part (c) of Theorem 3.2 states that, even though exactly solving Count is intractable for VSA, which we know by part (a), it is possible to approximate the answer with a *fully polynomial-time approximation scheme* (FPRAS): a randomized algorithm that, given S, d and $\varepsilon > 0$, returns an approximation of $|S(d)|$ within a multiplicative factor of $1 \pm \varepsilon$, with high probability (say 3/4) and running time bounded by a polynomial in $|S|$, $|d|$ and $1/\varepsilon$.

Furthermore, the scientific breakthrough of finding the FPRAS in Theorem 3.2 also led to the discovery of a *polynomial-time Las Vegas uniform generator (PLVUG)* for the outputs of vset-automata: a polynomial-time randomized algorithm that, given S and d, returns a uniformly sampled $t \in S(d)$ and is allowed to fail with a small probability (say, 1/4).

Counting the answers, as well as sampling thereof, is a special form of an aggregation operation over the answers. In the next section, we look more generally at aggregate queries with spanners.

4. WEIGHTS AND AGGREGATION

The list of practically relevant computational tasks for spanners does not end with computing the entire output or counting the size of the output. In many text analysis tasks, it is desirable to compute aggregate functions over the output. For example, Radinsky et al. [52] investigate patterns in textual news in order to make predictions. In this context, one may be interested in sequences of events that are close to each other in the text and compute aggregate functions over values assigned to such events. Such values can be, for instance, monetary quantities if we focus on financial events, or casualty counts if we look for conflicts. Such scenarios are common in subsequence mining [8, 52].

We now give a drastically simplified scenario where aggregation plays a central role. Furthermore, the example illustrates why it may be interesting to avoid the computation of huge intermediate results.

Example 4.1. *Consider the following document d, describing car configurations.*

> *There are 30 additional options that you can add to the default configuration of your car. Option 1) for €140, 2) for €900, [...], and the last option for €405.*

Even though there are only 30 options that one can choose from, they give rise to 2^{30}*, hence over one billion possible configurations of cars. Let* α *be a spanner with* $\mathrm{Vars}(\alpha) = \{x_1, \ldots, x_{30}\}$*, extracting all possible car configurations, that is, each tuple* $t \in \alpha(d)$ *encodes one configuration. Therefore, the relation* $\alpha(d)$ *contains* $2^{30} = 1,073,741,824$ *tuples. Let* $w(d,t)$ *be the price of the configuration, encoded by* t*. If we would want to do aggregation over these tuples, like computing the average or median price of a car configuration, is it possible to avoid materializing the relation containing the* 2^{30} *tuples?*

The above example is indeed just a toy example, but in effect the question is whether the materialization of an intermediate result of size in $O(|d|^{|\alpha|})$ can be avoided if one is interested in computing an aggregate value. A scenario where this question may also arise is in the development phase, where one wishes to get quick statistics about intermediate IE functions in a live manner without actually spending the time computing the entire set of answers.

This setting poses a range of new research questions. In fact, computing aggregate queries for spanners gives rise to at least two research challenges:

(a) Spanners have tuples of spans as output, but aggregation functions act on numerical values. So, how do we assign such numerical values, i.e., *weights* to the tuples in $S(d)$?

(b) How do we compute the aggregation over these weights efficiently?

In the remainder of this section, we will discuss initial approaches that we have investigated to tackle these challenges.

4.1 Aggregation Functions

Before we dive into the two aforementioned research challenges, we give definitions of some commonly used aggregation functions in databases. The definitions assume the existence of a *weight function* w that assigns weights to tuples of spans. In fact, we will formally model weight functions more generally as functions of the signature

$$w : \Sigma^* \times T \to \mathbb{Q} ,$$

where Σ^* is the set of all documents (that use symbols in Σ) and T is the set of all tuples of spans. Using this definition, one can also take the context of a tuple inside the document into account.

Let d be a document and S be a spanner such that $S(d) \neq \emptyset$. Let w be a weight function. We define the following spanner aggregation functions:

$$\mathrm{Sum}(S,d,w) := \sum_{t \in S(d)} w(d,t)$$

$$\mathrm{Avg}(S,d,w) := \frac{\mathrm{Sum}(S,d,w)}{\mathrm{Count}(S,d)}$$
$$\mathrm{Median}(S,d,w) := \operatorname*{median}_{t\in S(d)} w(d,t)$$
$$\mathrm{Min}(S,d,w) := \min_{t\in S(d)} w(d,t)$$
$$\mathrm{Max}(S,d,w) := \max_{t\in S(d)} w(d,t)$$

It remains to discuss the weight functions in more detail. We will discuss one instantiation here and discuss less restrictive options in Section 4.3. The simplest weight functions that we have considered are *single-variable weight functions* w, which assign values based on the strings selected by one variable in the spanner. Single variable weight functions can be specified in the input as a pair (x, f_w) where x is a variable and f_w is a partial function from the set of all words to $\mathbb{Q}$. The weight of each tuple t is then defined as

$$w(d,t) = f_w(d_{t(x)}).$$

Recall that our tuples of spans are partial functions t that map variables to spans. Therefore, $t(x)$ is a span and $d_{t(x)}$ is a subword of d. We note that this notion can be easily extended to constantly many variables [14]. We illustrate these notions on a simple example.

Example 4.2. *Consider again the document in Figure 1 and assume that we wish to calculate the total number of mentioned events. The table at the bottom left depicts a possible extraction of locations with their number of evens. The table at the bottom middle depicts the partial function f_w and the table on the bottom right depicts the corresponding string relation with the associated weights. To get an understanding of the total number of events, we may want to take the sum over the weights of the extracted tuples, namely* $7 + 9 + 3 = 19$. □

4.2 Computational Complexity

A natural question is now in which cases it is possible to avoid the materialization of the potentially huge output $S(d)$, which can be in the order of $O(|d|^{2k})$, where k is the number of variables of the spanner, and at what computational cost. To this end, one can study computational problems such as the following, which are parameterized by a class $\mathcal{S}$ of spanners.

Sum for $\mathcal{S}$	
Input:	Spanner $S \in \mathcal{S}$, document $d \in \Sigma^*$, and a weight function w.
Task:	Compute $\mathrm{Sum}(S,d,w)$.

The problems Average, Median, Min, and Max for a class $\mathcal{S}$ of spanners are defined analogously and just use a different aggregation function.

Theorem 4.3 ([15])**.**
(a) Max *and* Min *are in* FP *for* VSA.
(b) Sum, Average, *and* Median *are in* FP *for* uVSA *and are intractable for* VSA.

Here, by *intractable* we mean spanL-hard or #P-hard. Note that Theorem 4.3 states that Sum, Average and Median behave similarly to Count. Furthermore, as long as the weights assigned to tuples are non-negative, we can obtain a similar result as Theorem 3.2: the output of these problems can be approximated by an FPRAS. If weight functions can assign both positive and negative weights (-1 and $+1$ suffice), then the output of these problems cannot be approximated unless commonly believed conjectures do not hold.[1]

4.3 More Powerful Weight Functions

In principle, the framework does not need to limit itself to single-variable weight functions—the weight of a tuple could be any function that maps the tuple (alongside the document) into a number: the product of multiple variables, the sum of all numbers in the tuple, the difference between the leftmost and rightmost, and so on. The difficulty with this formulation is that we need to assume that this function is given as input, yet its naive representation is a table with an exponential number of rows (in the size of the spanner) and it is not realistic to assume that one can prepare it in advance. Therefore, we need to consider compact specifications of weights, and we do so via machine representations.

Polynomial-Time Weight Functions. A *polynomial-time weight functions w* is given in the input as a polynomial-time Turing Machine M that maps (d,t)-pairs to values in $\mathbb{Q}$ and defines $w(d,t) = M(d,t)$.

Not surprisingly, there are multiple drawbacks of having arbitrary polynomial-time weight functions. The first is that all considered aggregates become intractable (i.e., #P-hard or OptP-hard), even if the vset-automata in the input are already unambiguous. On the other hand, a "positional" approximation of the median is possible in the following sense. Given a vset-automaton, a document, and a parameter $\varepsilon > 0$, there is a randomized algorithm that runs in time polynomial in the input and $1/\varepsilon$ and returns a value in the $(0.5 \pm \varepsilon)$-quantile of the data with probability at least $3/4$. (Notice that the median is the 0.5-quantile of the data.)

Regular Weight Functions. Since single-variable weight functions are too verbose and polynomial-time weight functions can be considered as too powerful, the question is which representation of weight functions strikes a nice balance between complexity and expressiveness. One candidate is the class of

[1]That is, depending on the aggregate, the existence of an FPRAS would either imply that RP = NP or that the polynomial hierarchy collapses.

regular weight functions that is based on the concept of a $\mathbb{K}$-*Annotator* [18].

We consider (unambiguous) functional weighted vset-automata over the tropical semiring (also called min/plus semiring)[2] and the numerical semiring.

A *regular weight function* w is represented by a functional weighted vset-automaton W (over a semiring $\mathbb{K}$) and defines $w(d, t)$ as the $\mathbb{K}$-sum of the weights that W assigns to the ref-words w that produce the tuple t on the document d, that is, the ref-words w such that $\text{doc}(w) = d$ and $\text{tup}(w)$ is the tuple t, restricted to the variables $\text{Vars}(W)$.

There is indeed a natural hierarchy in these classes of weight functions. If we denote by SVar the single-variable weight functions, Reg the regular weight functions, UReg the regular weight functions given by unambiguous weighted spanners, and Poly the polynomial-time weight functions, then we have the following.

Theorem 4.4 ([14, 15, 18])**.** SVar $\subseteq$ UReg $\subseteq$ Reg $\subseteq$ Poly.

We will now give some complexity results for regular weight funcitons.

Theorem 4.5 ([15])**.** *The following holds for spanners given as* uVSA-*automata and* UReg *weight functions over the tropical or numerical semiring:*

(a) The problems Min*,* Max*,* Sum*, and* Average *are in* FP*.*

(b) The problem Median *is* #P*-hard.*

Recall that, even though Median is #P-hard for UReg weight functions and uVSA-automata, the $(0.5 \pm \varepsilon)$-quantile can be approximated, even for Poly weight functions and vset-automata.

The complexity landscape for regular weight functions is a bit more involved and strongly depends on the semiring of the weight function. For instance, Sum and Average for uVSA and regular weight functions over the numerical semiring are tractable, whereas in the same setting Min and Max are intractable. Orthogonaly, Min is tractable for VSA and regular weight functions over the tropical semiring whereas Max, Sum, and Average are intractable. We refer to Doleschal et al. [15] and Doleschal [14] for a more complete list of results.

5. PARALLELIZATION

In this section, we discuss the aspect of *parallelization* in query evaluation. 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 [34, 35], and so are instances of *Relation Extraction* [40, 63]. Algorithms for *coreference resolution* (identification of spans that refer to the same entity) are typically bounded to limited-size windows; for instance, Stanford's well known *sieve* algorithm [53] for coreference resolution processes separately intervals of three sentences [36]. Sentiment extractors typically process individual paragraphs or even sentences [45]. It is also common for extractors to operate on windows of a bounded number N tokens (a.k.a. N*-grams* or *local contexts*) [12, 29]. 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 [31]), 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 ones, 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. We recently embarked on a principled exploration of automated inference of *split-correctness* for information extractors [17]: the ability 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 (like the aforelisted ones), 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 be reprocessed.

5.1 Splittability and Split-Correctness

A *splitter* is just a spanner with one variable. As such, it always transforms a document into a set of spans, which means that it can be understood as a spanner that splits the input document into pieces of text, hence the name *splitter.* Typical such pieces of text in practice are sentences, paragraphs,

[2]One can also consider the tropical semiring with max/plus, in which case the complexity results are analogous to the ones we have for the tropical semiring with min/plus, with Min and Max interchanged.

N-grams or HTTP requests. Notice that the spans in the output can overlap, as in N-grams.

In order to define splittability, we need to define the *composition* $S \circ P$ of a spanner S and a splitter P. Intuitively, $S \circ P$ is the spanner that results from evaluating S on every part of the document extracted by P, with a proper shift of the indices. More formally we obtain the output of $S \circ P$ on a document d as follows. For each span $s \in P(d)$, we consider the word d_s. We then run S on each such d_s and shift the indices of the output so that the spans refer to the intended place in d instead of d_s. Concretely, this means that, if $s = [i, j\rangle$ and if $[i_s, j_s\rangle \in S(d_s)$, then we output $[i+i_s-1, i+j_s-1\rangle$.

Example 5.1. *Consider the document in Figure 2 and a splitter P that extracts subsentences. This can be done, for example, by the following extended regular expression, where $\Sigma' = \Sigma - \{,\}$.*

$$(\varepsilon \vee (\Sigma^* \texttt{,}))\ x{\vdash}\ \Sigma'^*\ {\dashv}x\ ((\texttt{,}\Sigma^*) \vee \texttt{.})\ .$$

Figure 2 shows the corresponding span relation. The composition of the spanner S from our running example and P is obtained by executing S on the subdocuments extracted by P (cf. Figure 3) and taking the union of the three relations, where every span is shifted accordingly. That is, the spans of the second relation – $[10, 16\rangle, [2, 6\rangle$ – are shifted by $31 - 1$ and the spans of the last relation – $[11, 17\rangle, [2, 7\rangle$ – are shifted by $48 - 1$. Observe that the resulting span relation is exactly the span relation in Figure 1.

Since executing S on each individual output of P enables parallelization, it is interesting if there is a difference between the output of S and $S \circ P$ on some document d. This property clearly depends on the definitions of S and P. We define this formally next. For the following definitions, recall that a spanner is a function from documents to span relations. As such, we consider two spanners to be equal if this function is the same.

A spanner S is *self-splittable* by a splitter P if

$$S = S \circ P\ .$$

If this is the case, then one can always run P over the input document d, run S over every extracted subdocument *in parallel*, and output the union of the obtained results (with indices properly shifted). The obtained result is then the same as $S(d)$, for every document d.

Another interesting scenario is the more general one where we allow the spanner on the chunks produced by P to be some spanner S_P different from S. In this case, we say that S *is splittable by P via S_P*, which is formally defined as

$$S = S_P \circ P\ .$$

If, for given S and P, such a spanner S_P exists, we say that S is *splittable* by P.

With these definitions, we formally define the following computational problems, which are again parameterized by a class $\mathcal{C}$ of spanners.

Split-Correctness for $\mathcal{C}$

Input:	Spanners $S, S_P \in \mathcal{C}$ and splitter $P \in \mathcal{C}$.
Question:	Is S splittable by P via S_P? In other words, is $S = S_P \circ P$?

Splittability for $\mathcal{C}$

Input:	Spanner $S \in \mathcal{C}$ and splitter $P \in \mathcal{C}$.
Question:	Is S splittable by P? In other words, is there a spanner $S_P \in \mathcal{C}$, such that $S = S_P \circ P$?

Self-Splittability for $\mathcal{C}$

Input:	Spanner $S \in \mathcal{C}$ and splitter $P \in \mathcal{C}$.
Question:	Is S self-splittable by P? In other words, is $S = S \circ P$?

Note that the problem Self-Splittability is a special case of Split-Correctness by choosing $S_P = S$. It can also be seen as a restriction of Splittability in the sense that it implies Splittability.

We illustrate these notions by a few examples. Many spanners S that extract person names do not look beyond the sentence level. This means that, if P splits to sentences, it is the case that S is self-splittable by P. Now suppose that S 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 P is the N-gram splitter, then S is self-splittable by P for $N \geq 5$ but not for $N < 5$. As another example, suppose that we analyze financial reports. Assume that S extracts those paragraphs that contain specific keywords and that P splits reports into single paragraphs. Then S is self-splittable by P. It is also splittable by P via the spanner S_P that selects the entire document if it contains the keywords.

5.2 Splitter Synthesis

When a spanner S is splittable by P, it is natural to ask whether one can synthesise a spanner S_P such that $S = S_P \circ P$. It turns out that, in the case of regular spanners, this is indeed the case. One can define a *canonical spanner* S_P^{can} such that S is splittable by P via S_P^{can} if and only if S is splittable by P *at all*.

Theorem 5.2 ([17]). *Let S be a spanner and P be a splitter. Then S is splittable by P if and only if S is splittable by P via S_P^{can}.*

```
There⊔are⊔7⊔events⊔in⊔Belgium,⊔9-15⊔in⊔France,⊔three⊔in⊔Berlin.
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
```

x	d_x
$[1,30\rangle$	There are 7 events in Belgium
$[31,46\rangle$	9-15 in France
$[47,63\rangle$	three in Berlin

Figure 2: A document d (top) and the span relation (bottom) extracted by a splitter P, which splits a document into its sub sentences.

```
There⊔are⊔7⊔events⊔in⊔Belgium
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
```

x_{loc}	x_{events}
$[23,30\rangle$	$[11,12\rangle$

```
⊔9-15⊔in⊔France
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

x_{loc}	x_{events}
$[10,16\rangle$	$[2,6\rangle$

```
⊔three⊔in⊔Berlin
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

x_{loc}	x_{events}
$[11,17\rangle$	$[2,7\rangle$

Figure 3: The three sub sentences, extracted from the document in Figure 2 and the corresponding span relations.

The definition of S_P^{can} is the following:

$$S_P^{\text{can}}(d) := \big\{t \mid \forall d' \in \Sigma^*, \forall s \in P(d') \text{ such that } d'_s = d, \text{ it holds that } (t \gg s) \in S(d')\big\}.$$

Intuitively, S_P^{can} selects all tuples that are "safe to select", since they don't contradict splittability, independent of the context.

Here, if $s = [i,j\rangle$, then the tuple $(t{\gg}s)$ denotes the tuple obtained from t by "shifting it $i-1$ positions to the right,", that is, adding $i-1$ to every index in t. Interestingly, Theorem 5.2 does not assume that S or P are regular. But if they are, then the definition of S_P^{can} is useful to actually construct S_P.

5.3 Complexity of Splitting Problems

We now have a look at the complexity of the aforementioned splitting problems. It turns out that they are all decidable and even polynomial-time solvable if the involved spanners are unambiguous and they satisfy a condition that we call *highlander condition* that we explain next.

We say that $[i,j\rangle$ *covers* $[i',j'\rangle$ if $i \le i' \le j' \le j$. Furthermore, if t is a tuple, we say that $[i,j\rangle$ covers t if $[i,j\rangle$ covers $t(x)$ for every variable $x \in \text{Vars}(t)$. A spanner S and a splitter P *satisfy the highlander condition* if for every document d and every tuple $t \in S(d)$ there exists at most one span $s \in P(d)$ which covers t.[3]

We note that the highlander condition is expected to be satisfied in many cases in practice. For instance, if the splitter is *disjoint* (i.e., only outputs spans $[i,j\rangle$, $[i',j'\rangle$ such that $i \le j \le i' \le j'$ or $i' \le j' \le i \le j$) and the spanner is *proper* (which is the case if it has at least one variable and does not return empty spans) then the highlander condition is satisfied. Typical splitters that are used in the context of tokenization, sentence boundary detection, paragraph splitting, and paragraph segmentation are disjoint. Examples of *non-disjoint* splitters include N-grams and pairs of consecutive sentences.

Theorem 5.3 ([17]). *The following holds for spanners and splitters given as functional vset-automata:*

(a) The SPLIT-CORRECTNESS *problem is complete for* PSPACE, SPLITTABILITY *is* PSPACE*-hard and in* EXPSPACE, *and* SELF-SPLITTABILITY *is* PSPACE*-complete.*

(b) Assuming the highlander condition and unambiguity of vset-automata, SPLIT-CORRECTNESS *is in* PTIME *while* SPLITTABILITY *is* PSPACE*-complete and, yet,* SELF-SPLITTABILITY *is in* PTIME.

Finally, we note that the highlander condition can be efficiently tested.

Proposition 5.4 ([17]). *Let S be a regular spanner and P be a splitter, given as functional vset-automata. Then it can be checked in polynomial time whether S and P satisfy the highlander condition.*

5.4 Reasoning with Black-Box Spanners

While we have a good understanding of splittability in the case of regular spanners, spanners in practice are often not regular and can be defined by programs that are way more complex than regular expressions or automata. It is even possible that they are just given to us as black-box algorithms for which we know some properties, such as the fact that they do not look beyond chunks of consecutive

[3]This is in acclimation to the tagline "There can be only one" of the Highlander movie.

sentences. In the following examples, we denote by $S(x, y)$ that spanner S uses the variables x and y.

Example 5.5. *Consider the spanner S that seeks to extract adjectives for Galaxy phones from reports. We define this spanner by joining three spanners:*

The spanner $S_1(x, y)$ is given by the regex formula

$$\Sigma^* \, x{\vdash} \, \mathsf{Galaxy}[\mathsf{A}-\mathsf{Z}] \setminus \mathsf{d}^* \, {\dashv}x \, \Sigma^* \, y{\vdash} \, \Sigma^* \, {\dashv}y \, \Sigma^*$$

that extracts mentions of Galaxy brands (e.g., Galaxy A72 and Galaxy S21) followed by substrings y that occur right before a period.

The spanner $S_2(x, x')$ is a coreference resolver (e.g., the sieve *algorithm [53]) that finds spans x' that coreference spans x. The spanner $S_3(x', y)$ finds pairs of noun phrases x' and attached adjectives y (e.g., based on a Recursive Neural Network [61]).*

For example, consider the review "I am happy with my Galaxy A72. It is stable." *Here, in one particular match, x will match (the span of)* Galaxy A72*, x' will match* it *(being an anaphora for* Galaxy A72*), 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? Naively materializing each relation might be too costly: $S_1(x, y)$ may produce too many matches, and $S_2(x, x')$ and $S_3(x', y)$ may be computationally costly. Nevertheless, we may have the information that S_2 is splittable by paragraphs and that S_3 is splittable by sentences (hence, by paragraphs). This information suffices to determine that the entire join $S_1(x, y) \bowtie S_2(x, x') \bowtie S_3(x', y)$ is splittable, hence parallelizable, by paragraphs. □

Example 5.6. *Now consider the spanner that joins two spanners: $S(x)$ extracts spans x followed by the phrase* "is kind" *(e.g.,* "Barack Obama is kind"*). The spanner $S'(x)$ extracts all spans x that match person names. Clearly, the spanner $S(x)$ does not split by a natural splitter, since it includes, for instance, the entire prefix of the document before* "is kind." *However, by knowing that $S'(x)$ splits by sentences, we know that the join $S(x) \bowtie P'(x)$ splits by sentences. Moreover, by knowing that $S'(x)$ splits by 3-grams, we can infer that $S(x) \bowtie S'(x)$ splits by 5-grams. Here, again, the holistic analysis of the join infers splittability in cases where intermediate spanners are not splittable.* □

It is therefore also important to develop a theory of splittability in the presence of such black-box spanners. Next, we mention a few results that do not assume regularity. The first one states that the composition operator $\circ$ is associative and transitive.

Theorem 5.7 ([17])**.** *The spanner/splitter composition is associative and transitive. That is, for all spanners S and splitters P_1, P_2 it holds that*

(a) $S \circ (P_1 \circ P_2) = (S \circ P_1) \circ P_2$,

(b) if S is splittable by P_1 and, furthermore, P_1 is splittable by P_2, then S is splittable by P_2, and

(c) if S is self-splittable by P_1 and, furthermore, P_1 is self-splittable by P_2 then S is self-splittable by P_2.

As we will see in the following example, spanner composition does not distribute over the join operator in general.

Example 5.8. *Let*

$$\begin{aligned} S_1 &:= \Sigma^* \, x_1{\vdash} \, a \, {\dashv}x_1 \, x_2{\vdash} \, b \, {\dashv}x_2 \, \Sigma^* \ , \\ S_2 &:= \Sigma^* \, x_2{\vdash} \, b \, {\dashv}x_2 \, x_3{\vdash} \, a \, {\dashv}x_3 \, \Sigma^* \ , \text{ and} \\ P &:= \Sigma^* \, x{\vdash} \, \Sigma\Sigma \, {\dashv}x \, \Sigma^* \ . \end{aligned}$$

That is, S_1 extracts every "a" in variable x_1 which is followed by a "b", extracted in variable x_2, S_2 extracts every "b" in variable x_2 which is followed by an "a" that is extracted in variable x_3, and P extracts all substrings of length two.

Let $S := S_1 \bowtie S_2$ be the join of both spanners and let $d = aba$. It follows that $P(d) = \{[1, 3\rangle, [2, 4\rangle\}$ and $S(d) = \{t\}$, where $t(x_1) = [1, 2\rangle$, $t(x_2) = [2, 3\rangle$, and $t(x_3) = [3, 4\rangle$. As there is no span $s \in P(d)$ that covers $t \in S(d)$ it follows directly that S is not splittable by P and therefore $S \neq S \circ P$. However, both spanners, S_1 and S_2, are self-splittable by P which implies that

$$(S_1 \circ P) \bowtie (S_2 \circ P) = S_1 \bowtie S_2 = S \ .$$

It follows directly that

$$(S_1 \bowtie S_2) \circ P \quad \neq \quad (S_1 \circ P) \bowtie (S_2 \circ P).$$

Therefore, in general it is not true that spanner composition does distributes over join.

However, composition distributes over join if the splitter is disjoint, the spanners share at least one variable, and the join of both spanners restricted to the shared variables is proper.

Theorem 5.9 ([17])**.** *Let P be a disjoint splitter and S_1 and S_2 be spanners such that $X :=$ Vars$(S_1) \cap$ Vars$(S_2) \neq \emptyset$ and the spanner $S_1 \bowtie S_2$ restricted to the variables in X is proper. Then*

$$(S_1 \bowtie S_2) \circ P \quad = \quad (S_1 \circ P) \bowtie (S_2 \circ P) \ .$$

Hence, Theorem 5.9 illustrates that knowing properties of spanners allows to establish query plans that might considerably optimize the computation.

6. CONCLUSIONS AND OUTLOOK

Viewing IE as a relational query over relations extracted from text allows us to incorporate and benefit from database paradigms in text processing. We have given an overview of a list of research efforts in this general direction, including query evaluation and complexity analysis, aggregate queries, provenance, and parallel-correctness. This list excludes some other efforts (due to space limitation). In particular, other database problems have been studied in the context of document spanners, including recursion and expressiveness aspects [21, 25, 26, 27, 41, 48, 50, 57, 58], extracting incomplete information and data cleaning [17, 23, 41, 49], ranked enumeration [10, 18], queries over dynamic data (incremental maintenance) [28], and ontology mediated IE [37, 56].

Many directions are left open for future research, including aspects of system implementation, a theory of spanners based on artificial neural networks, aspects of explanations to query answers (that has gained considerable attention by the database community over the past decade), and so on. Moreover, some of the past research has only scratched the surface of the studied topics; to highlight a particular direction, we believe that there is much impactful investigation to be done on query optimization with black-box extractors based on known behavior constraints (e.g., splittability [17]). Finally, we believe that understanding queries over non-relational data using the relational view and its establishment over decades, as spanners facilitate in the case of text, could be followed by other modalities of data such as graphs, images, and voice.

Acknowledgment. This work was supported by the German-Israeli Foundation for Scientific Research and Development (GIF), grant I-1502-407.6/2019.

References

[1] A. Amarilli, P. Bourhis, L. Jachiet, and S. Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, pages 111:1–111:15, 2017.

[2] A. Amarilli, P. Bourhis, and S. Mengel. Enumeration on trees under relabelings. In *ICDT*, pages 5:1–5:18, 2018.

[3] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, pages 22:1–22:19, 2019.

[4] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Constant-delay enumeration for nondeterministic document spanners. *ACM Trans. Database Syst.*, 46(1):2:1–2:30, 2021.

[5] 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.

[6] M. Arenas, L. A. Croquevielle, R. Jayaram, and C. Riveros. Efficient logspace classes for enumeration, counting, and uniform generation. In *PODS*, pages 59–73, 2019.

[7] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, pages 208–222, 2007.

[8] K. Beedkar, R. Gemulla, and W. Martens. A unified framework for frequent sequence mining with subsequence constraints. *ACM Trans. Database Syst.*, 44(3), 2019.

[9] H. Björklund, W. Gelade, and W. Martens. Incremental XPath evaluation. *ACM Trans. Database Syst.*, 35(4):29:1–29:43, 2010.

[10] P. Bourhis, A. Grez, L. Jachiet, and C. Riveros. Ranked enumeration of MSO logic on words. In *ICDT*, pages 20:1–20:19, 2021.

[11] N. Carmeli and M. Kröll. Enumeration complexity of conjunctive queries with functional dependencies. *Theory Comput. Syst.*, 64(5):828–860, 2020.

[12] J. Chen, D. Ji, C. L. Tan, and Z. Niu. Unsupervised feature selection for relation extraction. In *IJCNLP*, 2005.

[13] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, and S. Vaithyanathan. SystemT: An algebraic approach to declarative information extraction. In *ACL*, pages 128–137, 2010.

[14] J. Doleschal. *Optimization and Parallelization of RegEx Based Information Extraction.* PhD thesis, University of Bayreuth and Hasselt University, 2021.

[15] J. Doleschal, N. Bratman, B. Kimelfeld, and W. Martens. The Complexity of Aggregates over Extractions by Regular Expressions. In *ICDT*, pages 10:1–10:20, 2021.

[16] J. Doleschal, B. Kimelfeld, W. Martens, Y. Nahshon, and F. Neven. Split-correctness in information extraction. In *PODS*, pages 149–163, 2019.

[17] J. Doleschal, B. Kimelfeld, W. Martens, F. Neven, and M. Niewerth. Split-correctness in information extraction. *CoRR*, abs/1810.03367, 2021.

[18] J. Doleschal, B. Kimelfeld, W. Martens, and L. Peterfreund. Weight annotation in information extraction. In *ICDT*, pages 8:1–8:18, 2020.

[19] J. Doleschal, B. Kimelfeld, W. Martens, and L. Peterfreund. Weight annotation in information extraction. *CoRR*, 2020.

[20] A. Durand. Fine-grained complexity analysis of queries: From decision to counting and enumeration. In *PODS*, pages 331–346. ACM, 2020.

[21] 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.

[22] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. A relational framework for information extraction. *SIGMOD Rec.*, 44(4):5–16, 2015.

[23] 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.

[24] F. Florenzano, C. Riveros, M. Ugarte, S. Vansummeren, and D. Vrgoč. Constant delay algorithms for regular document spanners. In *PODS*, pages 165–177, 2018.

[25] D. D. Freydenberger. A logic for document spanners. *Theory Comput. Syst.*, 63(7):1679–1754, 2019.

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

[27] D. D. Freydenberger and L. Peterfreund. Finite models and the theory of concatenation. *CoRR*, abs/1912.06110, 2019.

[28] D. D. Freydenberger and S. M. Thompson. Dynamic complexity of document spanners. In *ICDT*, pages 11:1–11:21, 2020.

[29] C. Giuliano, A. Lavelli, and L. Romano. Exploiting shallow linguistic information for relation extraction from biomedical literature. In *EACL*, 2006.

[30] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

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

[32] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169–188, 1986.

[33] S. Kannan, Z. Sweedyk, and S. Mahaney. Counting and random generation of strings in regular languages. In *SODA*, pages 551–557. SIAM, 1995.

[34] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer. Neural architectures for named entity recognition. In *NAACL-HLT*, pages 260–270, 2016.

[35] R. Leaman and G. Gonzalez. BANNER: an executable survey of advances in biomedical named entity recognition. In *PSB*, pages 652–663, 2008.

[36] 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 *CoNLL*, pages 28–34, 2011.

[37] D. Lembo, Y. Li, L. Popa, and F. M. Scafoglieri. Ontology mediated information extraction in financial domain with mastro system-t. In *DSMM*, 2020.

[38] Y. Li, K. Bontcheva, and H. Cunningham. SVM based learning system for information extraction. In *DSMML*, pages 319–339, 2004.

[39] K. Losemann and W. Martens. MSO queries on trees: enumerating answers under updates. In *LICS*, pages 67:1–67:10, 2014.

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

[41] F. Maturana, C. Riveros, and D. Vrgoc. Document spanners for extracting incomplete information: Expressiveness and complexity. In *PODS*, pages 125–136, 2018.

[42] M. Niewerth. MSO queries on trees: Enumerating answers under updates using forest algebras. In *LICS*, pages 769–778, 2018.

[43] M. Niewerth and L. Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *PODS*, pages 179–191, 2018.

[44] D. Olteanu and M. Schleich. Factorized databases. *SIGMOD Rec.*, 45(2):5–16, 2016.

[45] B. Pang and L. Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *ACL*, pages 271–278, 2004.

[46] Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *ICDT*, pages 47–63, 2003.

[47] S. Patnaik and N. Immerman. Dyn-fo: A parallel, dynamic complexity class. In *PODS*, pages 210–221, 1994.

[48] L. Peterfreund. Grammars for document spanners. In *ICDT*, pages 7:1–7:18, 2021.

[49] L. Peterfreund, D. D. Freydenberger, B. Kimelfeld, and M. Kröll. Complexity bounds for relational algebra over document spanners. In *PODS*, pages 320–334, 2019.

[50] L. Peterfreund, B. ten Cate, R. Fagin, and B. Kimelfeld. Recursive Programs for Document Spanners. In *ICDT*, pages 13:1–13:18, 2019.

[51] H. Poon and P. M. Domingos. Joint inference in information extraction. In *AAAI*, pages 913–918, 2007.

[52] K. Radinsky, S. Davidovich, and S. Markovitch. Learning causality for news events prediction. In *WWW*, pages 909–918, 2012.

[53] K. Raghunathan, H. Lee, S. Rangarajan, N. Chambers, M. Surdeanu, D. Jurafsky, and C. D. Manning. A multi-pass sieve for coreference resolution. In *EMNLP*, pages 492–501, 2010.

[54] A. Ratner, S. H. Bach, H. R. Ehrenberg, J. A. Fries, S. Wu, and C. Ré. Snorkel: Rapid training data creation with weak supervision. *PVLDB*, 11(3):269–282, 2017.

[55] S. Sarawagi. Information extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.

[56] F. M. Scafoglieri and D. Lembo. A formal framework for coupling document spanners with ontologies. In *AIKE*, pages 155–162, 2019.

[57] M. L. Schmid and N. Schweikardt. A purely regular approach to non-regular core spanners. In *ICDT*, pages 4:1–4:19, 2021.

[58] M. L. Schmid and N. Schweikardt. Spanner evaluation over SLP-compressed documents. In *PODS*, 2021.

[59] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using Datalog with embedded extraction predicates. In *VLDB*, pages 1033–1044, 2007.

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

[61] R. Socher, C. C. Lin, A. Y. Ng, and C. D. Manning. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, pages 129–136, 2011.

[62] C. A. Sutton and A. McCallum. An introduction to conditional random fields. *Foundations and Trends in Machine Learning*, 4(4):267–373, 2012.

[63] D. Zeng, K. Liu, Y. Chen, and J. Zhao. Distant supervision for relation extraction via piecewise convolutional neural networks. In *EMNLP*, pages 1753–1762, 2015.