Incorporating Information Extraction in the Relational Database Model

Yoav Nahshon
Technion - Israel Institute of Technology
Haifa, Israel
yoavn@cs.technion.ac.il

Liat Peterfreund
Technion - Israel Institute of Technology
Haifa, Israel
liatpf@cs.technion.ac.il

Stijn Vansummeren
Université Libre de Bruxelles (ULB)
Bruxelles, Belgium
stijn.vansummeren@ulb.ac.be

ABSTRACT

Modern information extraction pipelines are typically constructed by (1) loading textual data from a database into a special-purpose application, (2) applying a myriad of text-analytics functions to the text, which produce a structured relational table, and (3) storing this table in a database. Obviously, this approach can lead to laborious development processes, complex and tangled programs, and inefficient control flows. Towards solving these deficiencies, we embark on an effort to lay the foundations of a new generation of text-centric database management systems. Concretely, we extend the relational model by incorporating into it the theory of document spanners which provides the means and methods for the model to engage the Information Extraction (IE) tasks. This extended model, called Spannerlog, provides a novel declarative method for defining and manipulating textual data, which makes possible the automation of the typical work method described above. In addition to formally defining Spannerlog and illustrating its usefulness for IE tasks, we also report on initial results concerning its expressive power.

CCS Concepts

•Information systems → Information extraction; Structured Query Language; *Semi-structured data; Relational database model; Data analytics;* **•Theory of computation → Logic and databases; Data modeling;**

Keywords

Information extraction, spanners, relational model, Datalog

1. INTRODUCTION

Modern technological and contemporary trends, such as social networking, result in large amounts of textual data with a high potential value within. Moreover, available textual data on the web is usually accompanied with meta-data such as timestamps and identifiers that can easily be stored alongside the text in a relational database. However, data manipulation capabilities of traditional database systems are in fact not suitable for extracting significant knowledge from unstructured data (e.g., text). Consequently, high quality and meaningful text analytics typically require out-of-database solutions, often carried out by professionals with high academic degrees. The historical success of general-purpose database systems is largely due to the access and query model, most notably SQL, which requires fairly low skill to practice, and hence, facilitates data management for a large community of developers. Systems that do combine databases and text analytics are usually bundles of a variety of different technologies; these typically consist of a scripting language, an ordinary relational database, and a statistics/machine-learning library. This approach requires developers to master a collection of inherently different technologies and programming frameworks. Furthermore, a bundled package limits the ability of the system to understand and analyze the entire flow as a whole, and hence, misses significant opportunities of optimization (e.g., by constructing wisely the control flow, expensive natural language processing (NLP) algorithms can be applied at later stages where considerable amount of textual data has already been filtered out).

In our efforts towards overcoming these shortcomings we turn to the fundamentals of the relational model and extend it with respect to textual data. In this paper we present *Spannerlog* - an extended relational model that is built upon the formalism of *document spanners* for information extraction (IE) [9]. In Spannerlog we consider two types of atomic data values: strings and spans, where a single relation can have attributes of both types. On the basis of this data model, we define a new query language that is similar to Datalog [1]. This query language incorporates concepts from spanner theory and Xlog [15]. With Spannerlog we can naturally represent structured data, unstructured data, generic algorithms (NLP in particular) and ad-hoc programs. It is our hypothesis that systems based on Spannerlog will (1) provide significantly easier development of text-centric applications, (2) promote faster solutions via automatic optimization, and (3) establish the basis for future incorporation of soft logic (or features), uncertainty via probabilistic models and data cleaning [8, 13].

The paper is organized as follows. In sections 2 and 3 we survey related work and give preliminary definitions. In section 4 we formally define Spannerlog and its semantics, and we illustrate its applicability for IE tasks by means of an example. In section 5 we report on initial results concern-

WebDB'16, June 26-July 01 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-4310-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2932194.2932200

ing expressive power. Finally, in section 6 we conclude and discuss future research directions for Spannerlog in both the practical and the theoretical sense.

2. RELATED WORK

The creation and manipulation of relations extracted from text are an inherent part of information extraction (IE). Recently, the concept of *document spanners* [9] (or simply spanners) has been formalized for relational querying of text. Informally, spanners extract spans (intervals in the text specified by bounding indices) from a given text, and construct relations with those spans, while relational algebra manipulates these relations. The spanner framework is heavily inspired by SystemT [4], IBM's principal IE tool that features an SQL-like query language. This system is typically integrated within larger software bundles for data analytics (e.g., IBM BigInsights). While spanners can only represent relations over spans, in Spannerlog we consider relations that use spans and strings as their data values. Furthermore, in the context of SystemT, spanners are typically used as a single component in a large pipeline of an IE application. In our framework, there is a bidirectional link between spanners and the database. Therefore, in Spannerlog spanners have a more prominent role.

There is a large body of work on string databases (i.e., databases in which the data values are strings), and on designing query languages for them in order to solve different IE problems [2, 3, 11, 12]. Most notably, and closely related to our work, is Xlog [15], where the Datalog relational query language is extended with special primitive types such as *documents* (distinguished chunks of text) and *spans*, and matchers of built-in regular expressions. Such frameworks enable declarative methods for approaching IE tasks, as does Spannerlog. However, there are some key differences of these works with ours. Whereas we focus on the theoretical foundations of a similar language in this paper, Xlog focuses on practical query optimization. Moreover, in Spannerlog we leverage for the first time spanner theory in the context of databases by incorporating its relatively new formalism into the relational model. Our premise is that the notion of document spanners carries with it highly desirable qualities for establishing more expressive and more useful text-centric databases. A second and important difference is the use in Spannerlog of span attributes alongside string attributes in a single relation. Specifically, span attributes have a different semantics than that of string attributes with respect to algebraic operations such as natural join and selection.

In addition to solutions that include databases in their design, there are various frameworks that consider other approaches, and are widely adopted for IE tasks. A commonly used IE system is GATE [5], wherein a document is processed by a sequence of phases (called *cascades*), each annotating spans with types by applying grammar rules over previous annotations. Other industrial analytics tools with IE development include DeepDive, Attensity, HP Autonomy, Oracle Collective Intellect, SAS, SAP, and more. The IE framework we seek to establish takes a different approach of those mentioned above. We propose a declarative solution that is more generic, and more comprehensible for execution engines. We believe that this will lead to more succinct programs with more automatic optimization opportunities, compared to these systems.

3. PRELIMINARIES

3.1 String Basics

Strings and Spans. We fix a finite alphabet Σ of *symbols.* We denote by Σ^* the set of all finite strings over Σ, and by Σ^+ the set of all finite strings of length at least one over Σ. A language over Σ is a subset of Σ^*. A *span* identifies a substring of some string by specifying its bounding indices. Formally, a span has the form $[i, j\rangle$, where $1 \leq i \leq j$. We denote by Spans the set of all possible spans. Let $\mathbf{s} = \alpha_1 \cdots \alpha_n$ be a string where $\alpha_1, \dots, \alpha_n \in \Sigma^*$, and let σ be a span $[i, j\rangle$ such that $1 \leq i \leq j \leq n+1$. In that case, we say that the span σ can be *applied* on $\mathbf{s}$, and the expression $\mathbf{s}[\sigma]$ then denotes the substring $\alpha_i \cdots \alpha_{j-1}$. Note that here we deviate from the definition given by Fagin et al. [9] by allowing spans not to be associated with specific strings. In section 5.1 we elaborate on this point. Lastly, we denote by $\mathsf{Spans}(\mathbf{s})$ the set of all the spans that can be applied on $\mathbf{s}$.

EXAMPLE 3.1. In all the examples throughout the paper we consider the example alphabet Σ which consists of the English letters and the standard punctuation marks. We may use the underscore symbol ('_') to represent whitespace between words. Consider the following string $\mathbf{s}$ where the index of each character is given underneath it.

t	h	e	_	f	a	s	t	_	l	a	n	e
1	2	3	4	5	6	7	8	9	10	11	12	13	14

For the spans $\sigma_1 = [1, 4\rangle$ and $\sigma_2 = [10, 14\rangle$ we have $\mathbf{s}[\sigma_1] = \texttt{the}$ and $\mathbf{s}[\sigma_2] = \texttt{lane}$.

Regular Expressions. Regular expressions over Σ are defined by the language

$$\gamma := \emptyset \mid \epsilon \mid \alpha \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^*$$

where $\emptyset$ is the empty set, ϵ is the empty string, and $\alpha \in \Sigma$. Note that "$\vee$" is the disjunction operator, "$\cdot$" is the concatenation operator, and "$*$" is the Kleene-star operator. By abuse of notation, if $\Sigma = \{\alpha_1, \dots, \alpha_k\}$, then we use Σ itself as an abbreviation of the regular expression $\alpha_1 \vee \cdots \vee \alpha_k$. The language recognized by a regular expression γ (i.e., the set of strings $\mathbf{s} \in \Sigma^*$ that γ matches) is denoted by $\mathcal{L}(\gamma)$. A language L over Σ is *regular* if $L = \mathcal{L}(\gamma)$ for some regular expression γ.

3.2 Relations

We assume given two disjoint sets $\mathbf{V}_{\mathsf{str}}$ and $\mathbf{V}_{\mathsf{spn}}$ of string variables and span variables, respectively, which may be assigned strings and spans (respectively). For a variable $x \in \mathbf{V}_{\mathsf{str}}$ we denote by $dom(x)$ the set Σ^*; similarly, for $x \in \mathbf{V}_{\mathsf{spn}}$ we denote by $dom(x)$ the set Spans. A *relation schema* is a finite sequence $\mathbf{x} = x_1 \dots, x_n$ of variables in $\mathbf{V}_{\mathsf{str}} \cup \mathbf{V}_{\mathsf{spn}}$ that does not contain repetitions. In this context, we may refer to variables as *attributes.* A *tuple over* $\mathbf{x}$ is a function t that assigns to each $x_i \in \mathbf{x}$ a value in $dom(x_i)$. For convenience, we will often refer to t as the sequence $t(\mathbf{x}) = t(x_1), \dots, t(x_n)$ (even though a tuple is defined as a function). A *relation* over $\mathbf{x}$ is a finite set of tuples over $\mathbf{x}$, and is associated with a *relation name* (or *symbol*) R. We denote by $sch(R)$ the relation schema of R. Note that $sch(R) = \mathbf{x}$. A (*database*) *schema* $\mathcal{S}$ is a function that assigns a relation schema $\mathcal{S}(R)$ to a each relation name R

in a finite set of relation names, denoted by $dom(\mathcal{S})$. A (*database*) *instance* $\mathbf{I}$ over schema $\mathcal{S}$ is a function that maps each relation name $R \in dom(\mathcal{S})$ to a relation over $\mathcal{S}(R)$. A relation R is called a *span relation* (resp. *string relation*) if $sch(R)$ contains only span variables (resp. string variables). A relation R is *Boolean* if $sch(R) = \emptyset$. In that case, true denotes that R consists of the empty tuple, and false denotes that $R = \emptyset$.

3.3 Information Extraction Functions

An *information extraction function* (or *IE function* for short) is a function F that is associated with a relation schema denoted by $sch(F)$. The IE function F maps each string $\mathbf{s} \in \Sigma^*$ to a relation over $sch(F)$. There are various ways to define IE functions. For instance, standard NLP tools such as part-of-speech (POS) taggers and dependency parsers can be thought of as IE functions. For illustration, let us consider an example.

EXAMPLE 3.2. Let *Pos* be an IE function where $sch(Pos)$ consists of two attributes: a span attribute and a string attribute. For a given string $\mathbf{s}$ that contains textual data, each tuple in $Pos(\mathbf{s})$ is of the form $(\sigma, \boldsymbol{\tau})$, where σ is the span of a token (word) in $\mathbf{s}$, and $\boldsymbol{\tau}$ is the corresponding part-of-speech for that token. For instance, applying *Pos* on the string $\mathbf{s}$ described in Example 3.1 should yield the relation $\{([1,4\rangle, \mathrm{DT}), ([5,9\rangle, \mathrm{JJ}), ([10,14\rangle, \mathrm{NN})\}$, where 'DT', 'JJ' and 'NN' stand for determiner, adjective and noun, respectively. State-of-the-art NLP tools, such as the Stanford CoreNLP library [14], can be used as implementations of IE-functions that represent standard NLP algorithms, as in the IE-function *Pos* that has the functionality of a POS tagger.

As a special case of IE functions, Fagin et al. [9] have considered *document spanners* (or simply *spanners* for short), which are IE functions such that:

- $sch(F)$ consists only of span variables. That is, the relations returned by F contain only spans, and not strings.
- For each $\mathbf{s} \in \Sigma^*$ and for every span $[i,j\rangle$ occurring in $F(\mathbf{s})$, it is the case that $[i,j\rangle$ can be applied on s.

Fagin et al. have considered various formalisms for defining spanners. One such formalism is given by regex formulas: these are simply regular expressions with embedded variables (also called *capture variables*). Instead of giving the precise definition, we show an example.

EXAMPLE 3.3. Consider the following regex formula γ.

$$(\Sigma^* \cdot \_)^* \cdot z\Big\{x\{\gamma_{cap}\} \cdot \_ \cdot y\{\gamma_{cap}\}\Big\} \cdot (\_ \cdot \Sigma^*)^*$$

Here, γ_{cap} is the regular expression $(A \vee \ldots \vee Z) \cdot (a \vee \ldots \vee z)^*$. The spanner represented by γ extracts all the triples of spans of the form (x, y, z), where x and y delimit adjacent words starting with a capital letter, and z delimits these two words.

The order of variables in the corresponding relation schema is determined by the alphabetical order of the variables names. For convenience, we often refer to regex formulas as the spanners (or IE functions) they represent. We denote by RGX the IE functions class of regex formulas. We refer the reader to Fagin et al. [9] for the full and formal definition of regex formulas.

Lastly, we highlight a special type of IE functions. An IE function F is called *Boolean* if $sch(F) = \emptyset$. In that case, for a given string $\mathbf{s}$, $F(\mathbf{s}) = \mathsf{true}$ denotes that $F(\mathbf{s})$ consists of the empty tuple, and $F(\mathbf{s}) = \mathsf{false}$ denotes that $F(\mathbf{s}) = \emptyset$.

4. SPANNERLOG

In this section we formally define the query language for Spannerlog. For convenience, and with a slight abuse of the terminology, we use the model name Spannerlog to also refer to its query language. Spannerlog is similar to Datalog [1] with two notable modifications:

- Given a string $\mathbf{s}$ and a span σ that can be applied on $\mathbf{s}$, we can construct the substring of $\mathbf{s}$ spanned by σ, denoted by $\mathbf{s}[\sigma]$.
- We can call IE functions as subroutines.

Spannerlog is parameterized by a class $\mathcal{C}$ of IE functions that can be called. We use the notation Spannerlog($\mathcal{C}$) to specify the $\mathcal{C}$ parameter, but we may not explicitly do so if $\mathcal{C}$ is clear from the context or irrelevant.

Let $\mathcal{C}$ be a class of IE functions. The syntax of the query language Spannerlog($\mathcal{C}$) is formally defined as follows.

Terms. We use two types of *terms*:

- A *span term* is either a constant span $[i,j\rangle \in \mathsf{Spans}$ or a span variable $x \in \mathbf{V}_{\mathsf{spn}}$.
- A *string term* is inductively defined as follows.
  - Every constant string $\mathbf{s} \in \Sigma^*$ and every string variable $x \in \mathbf{V}_{\mathsf{str}}$ is a string term.
  - If $\mathbf{s}$ is a string term and σ is a span term, then $\mathbf{s}[\sigma]$ is a string term.

Atomic Formulas. Let $\mathcal{S}$ be a schema. Let R be a k-ary relation symbol in $dom(\mathcal{S})$, and let F be an IE function in $\mathcal{C}$. The sequence $t_1, \ldots, t_k$ of terms is *properly typed* for R (resp. F) if each t_i, $1 \le i \le k$, is (1) a string term if the i-th attribute of $\mathcal{S}(R)$ (resp. $sch(F)$) is a string attribute, or (2) a span term if the i-th attribute of $\mathcal{S}(R)$ (resp. $sch(F)$) is a span attribute.

An *atomic formula* has two forms:

- A *DB-atom* over $\mathcal{S}$ has the form $R(t_1, \ldots, t_k)$ where $R \in dom(\mathcal{S})$ and $t_1, \ldots, t_k$ is properly typed for R.
- An *IE-atom* over $\mathcal{C}$ has the form $F\langle\mathbf{s}\rangle(t_1, \ldots, t_k)$ where $F \in \mathcal{C}$, $\mathbf{s}$ is a string term, and $t_1, \ldots, t_k$ is properly typed for F.

Before proceeding, let us consider again the IE functions we saw in 3.2 and 3.3.

EXAMPLE 4.1. Let us denote by $\mathsf{CoreNLP}$ the IE functions class of the NLP tools provided by the Standford's CoreNLP library. The POS tagger *Pos* we defined in Example 3.2 is an IE function in $\mathsf{CoreNLP}$ if it is implemented by the CoreNLP library. The corresponding IE-atom is given by $Pos\langle\mathbf{s}\rangle(\sigma, \boldsymbol{\tau})$, where $\mathbf{s}$ and $\boldsymbol{\tau}$ are string terms, and σ is a span term.

EXAMPLE 4.2. Recall the regex formula γ we presented in Example 3.3. The corresponding IE-atom of γ is given by

$\gamma\langle \mathbf{s}\rangle(x,y,z)$. By a convenient abuse of notation, we can define the same IE-atom directly by the following expression.

$$\mathsf{RGX}\langle \mathbf{s}\rangle[(\Sigma^* \cdot \_)^* \cdot z\Big\{x\{\gamma_{cap}\} \cdot \_ \cdot y\{\gamma_{cap}\}\Big\} \cdot (\_ \cdot \Sigma^*)^*]$$

Here we use RGX to indicate that the IE-function of the atom is in the class RGX and its definition is given by the expression between the square brackets.

Rules. Let $\mathcal{I}$ and $\mathcal{E}$ be two schemas such that $dom(\mathcal{I})$ and $dom(\mathcal{E})$ are disjoint. A *rule* is an expression of the form $\varphi \leftarrow \psi_1, \ldots, \psi_m$ where φ is a DB-atom over $\mathcal{I}$ and each ψ_i is either a DB-atom (over $\mathcal{I}$ or $\mathcal{E}$) or an IE-atom. We call φ the *head* and $\psi_1, \ldots, \psi_m$ the *body*. Let $\mathbf{s}$ be a string term, ρ be a rule, and ψ be a DB-atom in the body of ρ of the form $R(t_1, \ldots, t_k)$. We say that $\mathbf{s}$ is *bound* by ψ if $\mathbf{s}$ is (1) a constant string, (2) one of the t_i-s, or (3) has the form $\mathbf{s}'[\sigma]$ where $\mathbf{s}'$ is bound by ψ. A rule is *safe* if (1) every variable in the head occurs at least once in the body, and (2) for every IE-atom $F\langle \mathbf{s}\rangle(t_1, \ldots, t_k)$ it is the case that $\mathbf{s}$ is bound.

Programs. Let $\mathcal{I}$ be $\mathcal{E}$ be defined as previously. A *program* over Spannerlog($\mathcal{C}$) is a set of safe rules. Let P be a program. The *extensional schema* of P, denoted by $edb(P)$, is $\mathcal{E}$. Similarly, the *intensional schema* of P, denoted by $idb(P)$, is $\mathcal{I}$. The schema of P, denoted by $sch(P)$, is $edb(P) \cup idb(P)$. Informally, the semantics of a program is a mapping of database instances over $edb(P)$ to database instances over $edb(P) \cup idb(P)$. Next we provide a formal definition of the semantics we consider in our model.

4.1 Semantics

In order to later analyze Spannerlog in a theoretical manner, we establish a link to logic-programming [1] by defining its semantics via the well-known notion of *valuation*. Recall that a *valuation function* maps variables to constants. We extend the definition of a valuation function to operate on string and span terms by defining a *grounding function*. Let v be a valuation function, $\mathbf{s}$ be a string term, σ be a span term, and t be a term (string or span). The grounding function of v, denoted by $\tilde{v}$, is inductively defined as follows.

$$\tilde{v}(t) = \begin{cases} t & \text{if } t \in \Sigma^* \cup \mathsf{Spans} \\ v(t) & \text{if } t \in \mathbf{V}_{\mathsf{str}} \cup \mathbf{V}_{\mathsf{spn}} \\ \tilde{v}(\mathbf{s})[\tilde{v}(\sigma)] & \text{if } t \text{ is of the form } \mathbf{s}[\sigma] \end{cases}$$

Note that $\tilde{v}$ is a well defined function. That is, it is defined for every span and string term. Moreover, the restriction of $\tilde{v}$ to variables is exactly v. We say that $\tilde{v}$ is *valid* for a term t if (1) t is either a constant or a variable, or (2) t has the form $\mathbf{s}[\sigma]$ and it is the case that $\tilde{v}(\sigma)$ is a span that can be applied on $\tilde{v}(\mathbf{s})$.

Let $\mathbf{t}$ be the sequence of terms $t_1, \ldots, t_k$. The *grounding* of $\mathbf{t}$ by v, denoted by $\tilde{v}(\mathbf{t})$, is the sequence $\tilde{v}(t_1), \ldots, \tilde{v}(t_k)$. Let $\mathcal{S}$ be a schema, $\mathbf{I}$ be an instance over $\mathcal{S}$, and R be a relation name in $dom(\mathcal{S})$. We say that the DB-atom $R(\mathbf{t})$ is *satisfied* under v by $\mathbf{I}$, denoted by $\mathbf{I} \models_v R(\mathbf{t})$, if (1) $\tilde{v}$ is valid for every $t_i \in \mathbf{t}$, and (2) $\tilde{v}(\mathbf{t}) \in \mathbf{I}(R)$. Let $\mathcal{C}$ be a class of IE functions, and F be an IE function in $\mathcal{C}$. We say that the IE-atom $F\langle \mathbf{s}\rangle(\mathbf{t})$ is *satisfied* under v by $\mathbf{s}$, denoted by $\mathbf{s} \models_v (F, \mathbf{t})$, if (1) $\tilde{v}$ is valid for $\mathbf{s}$ and for every $t_i \in \mathbf{t}$, and (2) $\tilde{v}(\mathbf{t}) \in F(\tilde{v}(\mathbf{s}))$.

Let P be a program in Spannerlog($\mathcal{C}$), ρ be a rule in P of the form $\phi \leftarrow \psi_1, \ldots, \psi_m$, and $\mathbf{I}$ be an instance over $sch(P)$.

Review

text	*bid*	*uid*	*rid*
Great food, and excellent wine, but for an exceptionally expensive price.	101	201	301
The best pancake I have ever eaten.	102	202	302

(a)

$$Signal(\mathbf{r}, x, y) \leftarrow Review(\mathbf{t}, \mathbf{b}, \mathbf{u}, \mathbf{r}), Dep\langle \mathbf{t}\rangle(x, y), Pos\langle \mathbf{t}\rangle(y, \text{`}\mathtt{JJ}\text{'})$$
$$Sentiment(\mathbf{r}, \mathbf{b}, \mathbf{t}[x], \mathbf{smt}) \leftarrow Review(\mathbf{t}, \mathbf{b}, \mathbf{u}, \mathbf{r}), Signal(\mathbf{r}, x, y), SA\langle \mathbf{t}[y]\rangle(\mathbf{smt})$$

(b)

Sentiment

rid	*bid*	*entity*	*class*
301	101	food	positive
301	101	wine	positive
301	101	price	negative
302	102	pancake	positive

(c)

Figure 1: A run of a Spannerlog program that performs entity-level sentiment analysis for a review database.

We say that $\mathbf{I}$ *satisfies* ρ by v, denoted by $\mathbf{I} \models_v \rho$, if it holds that if (1) for each DB-atom R in the body of ρ it holds that $\mathbf{I} \models_v R$, and (2) for each IE-atom $F\langle \mathbf{s}\rangle(\mathbf{t})$ in ρ it holds that $\mathbf{s} \models_v (F, \mathbf{t})$, then it holds that $\mathbf{I} \models_v \phi$. $\mathbf{I}$ is called a *model* of P if for every $\rho \in P$ and every valuation v it holds that $\mathbf{I} \models_v \rho$. Let $\mathbf{I}^{\mathcal{E}}$ be an instance over $\mathcal{E}$. As in Datalog, the *semantics* of the program P on input $\mathbf{I}^{\mathcal{E}}$, denoted by $P(\mathbf{I}^{\mathcal{E}})$, is the minimum model of P containing $\mathbf{I}^{\mathcal{E}}$, if it exists. In some contexts, we may refer to $P(\mathbf{I}^{\mathcal{E}})$ as the *output* of P.

Example 4.3. Figure 1 shows an example of a run of a Spannerlog program. Consider a database for businesses reviews. Reviews written by users are stored in the relation *Review*, given in (a), in which there are four attributes: *text* (the review body), *bid* (business ID), *uid* (user ID) and *rid* (review ID). For each review in *Review*, the program in (b) performs *entity-level sentiment analysis*. That is, for a given review, the program identifies the different aspects (or *entities*) of the business mentioned in the review, and then decides whether the review expresses positive, negative or neutral sentiment towards each one of them. For instance, in the first review given in (a), the review expresses a positive sentiment towards the food ("great food"), but expresses a negative sentiment towards the price ("exceptionally expensive price"). The program in (b) consists of two rules. The IE-atom $Dep\langle \mathbf{t}\rangle(x, y)$ in the first rule extracts a *dependency parse tree* of $\mathbf{t}$, where a *head* and a *modifier* of every pair in the tree are delimited by x and y, respectively. The motivation for extracting the parse tree stems from the observation that a modifier has an impact on the sentiment of its head expressed in the review. For instance, in the first review the word "food" is modified by the word "great", hence we deduce that the sentiment in the review towards the food is positive. We already encountered the IE-atom $Pos\langle \mathbf{t}\rangle(y, \text{`}\mathtt{JJ}\text{'})$ in Example 4.1. In this context, we use it to indicate that we are only interested in modifiers that are adjectives (JJ). The eligible head-modifier pairs are then stored in the *Signal* relation. In the second rule, we take use of a sentiment analysis (SA) tool to determine the sentiment for each entity; this is performed by the IE-atom $SA\langle \mathbf{t}[y]\rangle(\mathbf{smt})$. The output of the program is shown in (c).

5. ANALYSIS

In this section we provide fundamental results obtained from analyzing Spannerlog. Since our model is built on the formalism of spanners, our initial analysis involves comparing Spannerlog to spanners and their representations as they are defined in Fagin et al. [9]. In addition to the regex formulas (RGX) we have already encountered in section 3.3, we briefly describe two other spanner representations defined by Fagin et al. that we consider in our analysis.

- A *regular spanner* is a spanner that can be expressed in the closure of the regex formulas under relational algebra. More formally, we denote by REG the class of expressions in the closure of RGX under union ($\cup$), projection ($\pi_{\mathbf{x}}$ where $\mathbf{x}$ is a sequence of variables) and natural join ($\bowtie$). Note that the natural join is based on span equality, and not string equality. A spanner is *regular* if it is definable in REG.
- *Core spanners* extend the regular spanners with the string-equality selection, denoted by $\varsigma^{=}$. That is, given an expression γ in REG and two span variables x, y occurring in γ, the spanner defined by $\varsigma^{=}_{x,y}(\gamma)$ selects all those tuples from γ in which x and y span equal strings (though x and y can be different spans). The class Core of core spanners is the closure of RGX under union, projection, natural join and string-equality selection.

As in Datalog, the algebraic operators union, projection and natural join can be expressed in any Spannerlog program. As a consequence, the class of spanners definable in Spannerlog(RGX) is a superset of REG. Later we will see that this class is in fact a strict superset of Core.

Before proceeding, we define what it means for a language (i.e., a subset of Σ^*) to be *recognized* by a Spannerlog program and by a spanner. This will enable us to discuss the expressive power of Spannerlog compared to the different spanner classes defined in Fagin et al. Let $\mathcal{L}$ be a language over Σ. We say that $\mathcal{L}$ is *recognizable* by RGX, REG or Core if there exists a Boolean spanner S (recall the definition of Boolean IE-functions in section 3.3) in RGX, REG or Core, respectively, if for each $w \in \Sigma^*$, it holds that $S(w) = \mathsf{true}$ if and only if $w \in \mathcal{L}$. For Spannerlog we give the following definition. Let P be a program such that $edb(P)$ consists of only the relation name Doc, and the relation schema $edb(P)(Doc)$ has a single string attribute. Let Out be a relation name in $idb(P)$ such that the relation schema $idb(P)(Out)$ is Boolean. We say that $\mathcal{L}$ is *recognizable* by P, if for each $w \in \Sigma^*$ it holds that for the instance $\mathbf{I}$ over $edb(P)$, in which $\mathbf{I}(Doc)$ consists only of w, $P(\mathbf{I})(Out) = \mathbf{true}$ if and only if $w \in \mathcal{L}$. Note that the program P functions as a Boolean IE function, hence we may refer to P as a program over *Boolean Spannerlog*($\mathcal{C}$), where $\mathcal{C}$ is some IE functions class.

5.1 Span Semantics

We begin our analysis by discussing the span semantics in Spannerlog, compared to that in Fagin et al. In section 3.1 we defined a span σ to be simply an interval specified by bounding indices, and we allowed σ to be applied on some string $\mathbf{s}$ to extract a substring of $\mathbf{s}$. Therefore, a span is *string-independent* in that it can be applied on every string, whereas in Fagin et al. a span is defined similarly, but is also associated with the string in which it was originated, and can only be applied on that string; hence we can consider such span as *string-dependent*. We use the term *independent spans* to describe the span semantics used in Spannerlog, and the term *dependent spans* to describe the span semantics used in Fagin et al. Obviously, every language that can be recognized by a program that adopts the dependent spans semantics, also can be recognized by a program that adopts the independent spans semantics.

The following program over Boolean Spannerlog(RGX) illustrates the flexibility we gain by adopting the independent spans semantics.

$$\begin{aligned} \mathit{Out}() \leftarrow & \mathit{Doc}(\mathbf{s}),\ \mathsf{RGX}\langle \mathbf{s}\rangle[x\{a^*\}\cdot y\{b^*\}], \\ & \mathsf{RGX}\langle \mathbf{s}[y]\rangle[x\{b^*\}] \end{aligned} \tag{1}$$

The program in (1) consists of a single rule. The IE-atom $\mathsf{RGX}\langle \mathbf{s}\rangle[x\{a^*\}\cdot y\{b^*\}]$ demands the span variables x and y to be grounded to spans that span substrings of $\mathbf{s}$ of the form a^* and b^*, receptively. In the last atom $\mathsf{RGX}\langle \mathbf{s}[y]\rangle[x\{b^*\}]$, x is matched against the substring of $\mathbf{s}$ that contains only b's. Note that under the dependent spans semantics, the program is malformed: x is applied on two strings: $\mathbf{s}$ and $\mathbf{s}[y]$, rather than just one. Since x occurs in both the IE-atoms, any valuation function that satisfies the first two atoms can only satisfy the third atom if $\mathbf{s}$ has as many b's as a's. From here we conclude that the program in (1) recognizes the context-free language $\mathcal{L}_1 = \{a^n b^n \mid n \in \mathbb{N}\}$.

In a similar fashion, we define the following program over Boolean Spannerlog(RGX).

$$\begin{aligned} \mathit{Out}() \leftarrow & \mathit{Doc}(\mathbf{s}),\ \mathsf{RGX}\langle \mathbf{s}\rangle[x\{a^*\}\cdot y\{b^*\}\cdot z\{c^*\}], \\ & \mathsf{RGX}\langle \mathbf{s}[y]\rangle[x\{b^*\}],\ \mathsf{RGX}\langle \mathbf{s}[z]\rangle[x\{c^*\}] \end{aligned} \tag{2}$$

This program is similar in concept to that in (1). We leave it to the reader to verify that the program in (2) recognizes the non-context-free language $\mathcal{L}_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$. In Fagin et al. it was proven that Boolean regular spanners can recognize only regular languages. This leads us to the following theorem.

THEOREM 5.1. There exists a language that is not context-free, and yet, is recognizable by a program over Boolean Spannerlog(RGX). In particular, the class of spanners definable by Spannerlog(RGX) strictly contains the class of regular spanners.

5.2 String Equality

With core spanners we get string equality by applying the string-equality selection operator ($\varsigma^{=}$). Fagin et al. showed that string comparison cannot be expressed by a regular spanner. The following theorem shows it is possible to express string equality by a program over Spannerlog(RGX).

THEOREM 5.2. String equality is definable in Spannerlog(RGX).

To prove this, consider the following program.

$$Equals(\mathbf{s}[x], \mathbf{s}[x]) \leftarrow Doc(\mathbf{s}),\ \mathsf{RGX}\langle \mathbf{s}\rangle[\Sigma^*\cdot x\{\Sigma^*\}\cdot \Sigma^*] \tag{3}$$

The relation $Equals$ consists of all the pairs $(\mathbf{t}, \mathbf{t})$ such that $\mathbf{t}$ is a substring of $\mathbf{s}$. We can now use $Equals$ to recognize the non-context-free language $\mathcal{L}_3 = \{\mathbf{s} \cdot \mathbf{s} \mid \mathbf{s} \in \Sigma^*\}$ by adding the following rule.

$$Out() \leftarrow Doc(\mathbf{s}),\ \mathsf{RGX}\langle \mathbf{s}\rangle[x\{\Sigma^*\} \cdot y\{\Sigma^*\}],\ Equals(\mathbf{s}[x], \mathbf{s}[y])$$

Therefore, the above program serves as an alternative proof for theorem 5.1. Note that by expressing string equality, we have shown that any core Spanner is definable by a program over Spannerlog(RGX). Moreover, In Fagin et al. it was proven that the non-regular language $\mathcal{L}_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ is not recognizable by any Boolean core spanner. However, we have shown that $\mathcal{L}_1$ is recognizable by the program in (1). We conclude this discussion with the following theorem.

Theorem 5.3. The class of spanners definable by Spannerlog(RGX) strictly contains the class of core spanners.

5.3 Context-Free Languages

In addition to the previously discussed connections between Spannerlog and formal languages, we present the following result that shows a link to context-free languages.

Theorem 5.4. Any context-free language is recognizable in Spannerlog(RGX).

In other words, for every context-free grammar G there exists a program P over Boolean Spannerlog(RGX) such that $\mathcal{L}(G)$, the language produced by G, is recognizable by P. Next we describe the idea of the construction of P. In the following, capitalized letters denote nonterminals and Greek letters denote terminals. Let G be a context-free grammar, and assume w.l.o.g. that G is in the Chomsky normal form. That is, every production rule in G is of the form $A \to \alpha$ or $A \to BC$. We define a program P as follows. For every production rule of the form $A \to \alpha$ in G, we define the following rule in P.

$$A(\mathbf{s}[x]) \leftarrow Doc(\mathbf{s}),\ \mathsf{RGX}\langle \mathbf{s}\rangle[\Sigma^* \cdot x\{\alpha\} \cdot \Sigma^*]$$

For every production rule of the form $A \to BC$ in G, we define the following rule in P.

$$A(\mathbf{s}[z]) \leftarrow Doc(\mathbf{s}),\ \mathsf{RGX}\langle \mathbf{s}\rangle[\Sigma^* \cdot z\{x\{\Sigma^+\} \cdot y\{\Sigma^+\}\} \cdot \Sigma^*],$$
$$B(\mathbf{s}[x]),\ C(\mathbf{s}[y])$$

As the final step in constructing P, we add the following rule to decide whether a given string is in $\mathcal{L}(G)$.

$$Out() \leftarrow Doc(\mathbf{s}),\ S(\mathbf{s})$$

where S is the start symbol of G.

We conclude this discussion by noting that the other direction of the theorem does not hold. The context-sensitive language $\mathcal{L}_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is recognizable by the program given in (2), but it is well-known that there does not exist a context-free grammar G such that $\mathcal{L}(G) = \mathcal{L}_2$.

6. CONCLUSIONS AND FUTURE WORK

In this paper we introduced an extended relational model for IE, called Spannerlog, along with a suitable query language, in which generalized document spanners, called IE functions, are incorporated into traditional Datalog. Spannerlog offers a unified data model for the domains of relational databases and IE (or NLP more generally). Its query language serves as an interface layer between the application layer and the infrastructure layer where various techniques and methods are used, and thus eliminates the need for developers to master a variety of often-complex mechanisms, and to glue them together. Moreover, Spannerlog presents a high-level workflow, and as a consequence, new optimization opportunities arise. For instance, a lot of IE tasks are highly suitable for the MapReduce paradigm [6]. Therefore, the efficiency of a Spannerlog program may greatly be increased by identifying segments of the program that fit into the MapReduce framework, and then executing them accordingly. However, it is often the case that IE tasks are highly difficult to solve by means of deterministic rules, wherein the user fully specifies the outcome of a program. In text analysis, it has been found effective to interpret rules as soft constraints (e.g., by Markov Logic Networks [7]). In future work we will pursue an implementation that supports such an interpretation of Spannerlog rules.

In addition to exploring the applicability of Spannerlog, we also plan to proceed in the theoretical path. In section 5 we have discussed several initial results on the expressive power of Spannerlog. We plan to pursue this direction further, and in particular the relationship of Spannerlog to known concepts of formal languages. Aspects of complexity, however, have not been addressed in this paper, and will receive more focus in the future. Interestingly, a recent work [10] has reported results on various complexity aspects of document spanners. We plan to investigate whether these results can be extended to Spannerlog.

Acknowledgments

We are extremely grateful to Benny Kimelfeld for making this paper possible, and for helpful discussions.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Benedikt, L. Libkin, T. Schwentick, and L. Segoufin. Definable relations and first-order query languages over strings. *J. ACM*, 50(5):694–751, 2003.

[3] A. J. Bonner and G. Mecca. Sequences, datalog, and transducers. *J. CSS*, 57(3):234–259, 1998.

[4] 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.

[5] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, N. Aswani, I. Roberts, G. Gorrell, A. Funk, A. Roberts, D. Damljanovic, T. Heitz, M. A. Greenwood, H. Saggion, J. Petrak, Y. Li, and W. Peters. *Text Processing with GATE (Ver. 6)*. 2011.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[7] P. M. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2009.

[8] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Cleaning inconsistencies in information extraction via prioritized repairs. In *PODS*, pages 164–175, 2014.

[9] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to

information extraction. *J. ACM*, 62(2):12, 2015.

[10] D. D. Freydenberger and M. Holldack. Document spanners: From expressive power to decision problems. In *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016*, pages 17:1–17:17, 2016.

[11] S. Ginsburg and X. S. Wang. Regular sequence operations and their use in database queries. *J. Comput. Syst. Sci.*, 56(1):1–26, 1998.

[12] G. Grahne, M. Nykänen, and E. Ukkonen. Reasoning about strings in databases. *J. Comput. Syst. Sci.*, 59(1):116–162, 1999.

[13] B. Kimelfeld. Extending datalog intelligence. In *RR*, pages 1–10, 2015.

[14] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *ACL*, pages 55–60, 2014.

[15] 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.